Name: Sujan Neupane

UNI ID: 2058939

I ran this code on `Satrun Cloud` platform's GPU. So, I uploaded the dataset to google drive and downloaded the data into satrun cloud's GPU runtime using gdown package and extracted iamges using zipfile package.

In [2]:
```
# !pip install gdown
```

In [4]:
```
# !gdown '1ROrEh7iCYHGQ_nRIw9d0EuyLRaNd656t'
```

Unzipping Files

In [2]:
```
'''
This package is used to extract all images from the zipped data
'''

import zipfile

zip_ref = zipfile.ZipFile('Flower Classification.zip', 'r')
zip_ref.extractall(".")
zip_ref.close()
```

Importing the dependencies

In [2]:
```
'''
Following packages, libraries and frameworks are used for data understanding, cleaning and modeling purpose.

1. NumPy : Performing image and array manipulations
2. Pillow : Loading and working with jpg images
3. torch : PyTorch framework for working with tensors, GPU and models
4. Sklearn : Useful library that provides very useful functions mainly used to evaluate the performance of our model
```

```
5. Seaborn and Matplotlib : Data visualization
'''

import os
import torch
import torchvision
import tarfile
import torch.nn as nn
import numpy as np
import random
from tqdm import tqdm
import pandas as pd
import seaborn as sns
import torch.nn.functional as F
from PIL import Image
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import ConcatDataset
from torchvision.utils import make_grid
from torchsummary import summary
from sklearn.metrics import accuracy_score, classification_report, roc_auc_score, confusion_matrix, roc_curve, precisio
import torchvision.models as models
import warnings
warnings.filterwarnings("ignore")
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as tt
from torch.utils.data import random_split
from torchvision.utils import make_grid
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

matplotlib.rcParams['figure.facecolor'] = '#ffffff'
```

Data Understanding

Lets start with the number of classes and number of samples in each class in the entire dataset.

In [31]:
```
# this dictionary will hold names of classes as keys and number of samples in corresponding class as the number of valu
number_of_samples_per_class = {}
```

In [32]:
```
# the path to our entire data

path = r'C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Train'
```

In [33]:
```
# this will return the folders present in the given path.
# In this case, the names of classes

n_classes = os.listdir(path)
print("The total number of classes present in the dataset are: {}\nand they are: {}".format(len(n_classes), n_classes))
```

```
The total number of classes present in the dataset are: 5
and they are: ['daisy', 'dandelion', 'rose', 'sunflower', 'tulip']
```

`daisy` Class

In [34]:
```
# loading the first sample from daisy class

sample1 = os.listdir(path+'/daisy')[0]
sample1 = os.path.join(path+"/daisy", sample1)

print("daisy")
daisy = Image.open(sample1)
print("Image Shape: ", np.array(daisy).shape)
daisy
```

```
daisy
Image Shape:  (263, 320, 3)
```

Out[34]:

This is the first sample present in the daisy class present in the dataset. We have used pillow library to import and present the image. It can be further seen that the image has 3 channels with height and width of 333 and 555 pixels respectively.

In [35]:
```
number_of_samples_in_daisy_class = len(os.listdir(path+'/daisy'))
```

In [36]:
```
print("There are a total of {} number of samples in {} class.".format(number_of_samples_in_daisy_class, 'daisy'))
```

```
There are a total of 761 number of samples in daisy class.
```

We can clearly see that there are 761 samples in this class.

In [37]:
```
number_of_samples_per_class['daisy'] = number_of_samples_in_daisy_class
```

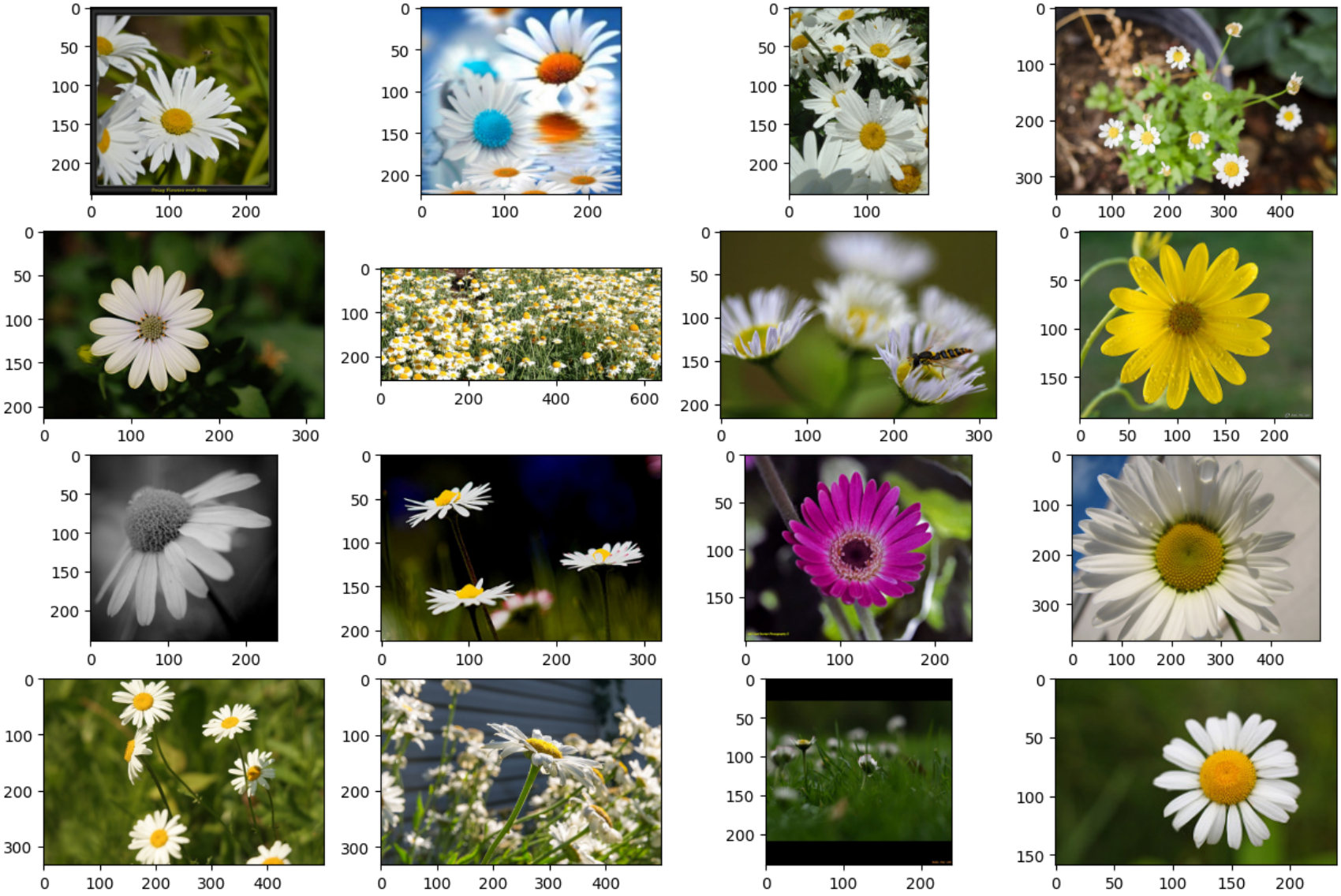
In [38]:
```
number_of_samples_per_class
```

Out[38]:
```
{'daisy': 761}
```

In [39]:
```
'''
Lets plot 16 random daisy samples in matplotlib subplot.
'''

sixteen_daisy_samples = random.sample(os.listdir(path+'/daisy'), 16)

fig, axes = plt.subplots(4, 4, figsize=(15, 10))
```

```
print("16 Random daisy image samples plot")
for i, img in tqdm(enumerate(sixteen_daisy_samples)):
    ax1 = plt.subplot(4, 4, i+1)
    sample1 = os.path.join(path+"/daisy", img)
    sample1 = Image.open(sample1)
    ax1.imshow(sample1)
```

```
16 Random daisy image samples plot
16it [00:00, 292.24it/s]
```



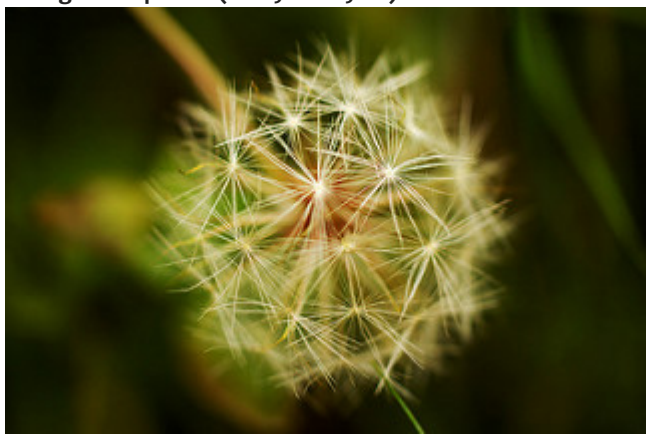
we can clearly see that there are a variety of image size in daisy class.

For dandelion class

In [40]:
```
# loading the first sample from dandelion class

sample1 = os.listdir(path+'/dandelion')[0]
sample1 = os.path.join(path+"/dandelion", sample1)

print("dandelion")
dandelion = Image.open(sample1)
print("Image Shape: ", np.array(dandelion).shape)
dandelion
```

```
dandelion
Image Shape:  (213, 320, 3)
```

Out[40]:



This is the first sample present in the dandelion class present in the dataset. We have used pillow library to import and present the image. It can be further seen that the image has 3 channels with height and width of 180 and 240 pixels respectively.

In [41]:
```
number_of_samples_in_dandelion_class = len(os.listdir(path+'/dandelion'));number_of_samples_in_dandelion_class
```

Out[41]:
```
1051
```

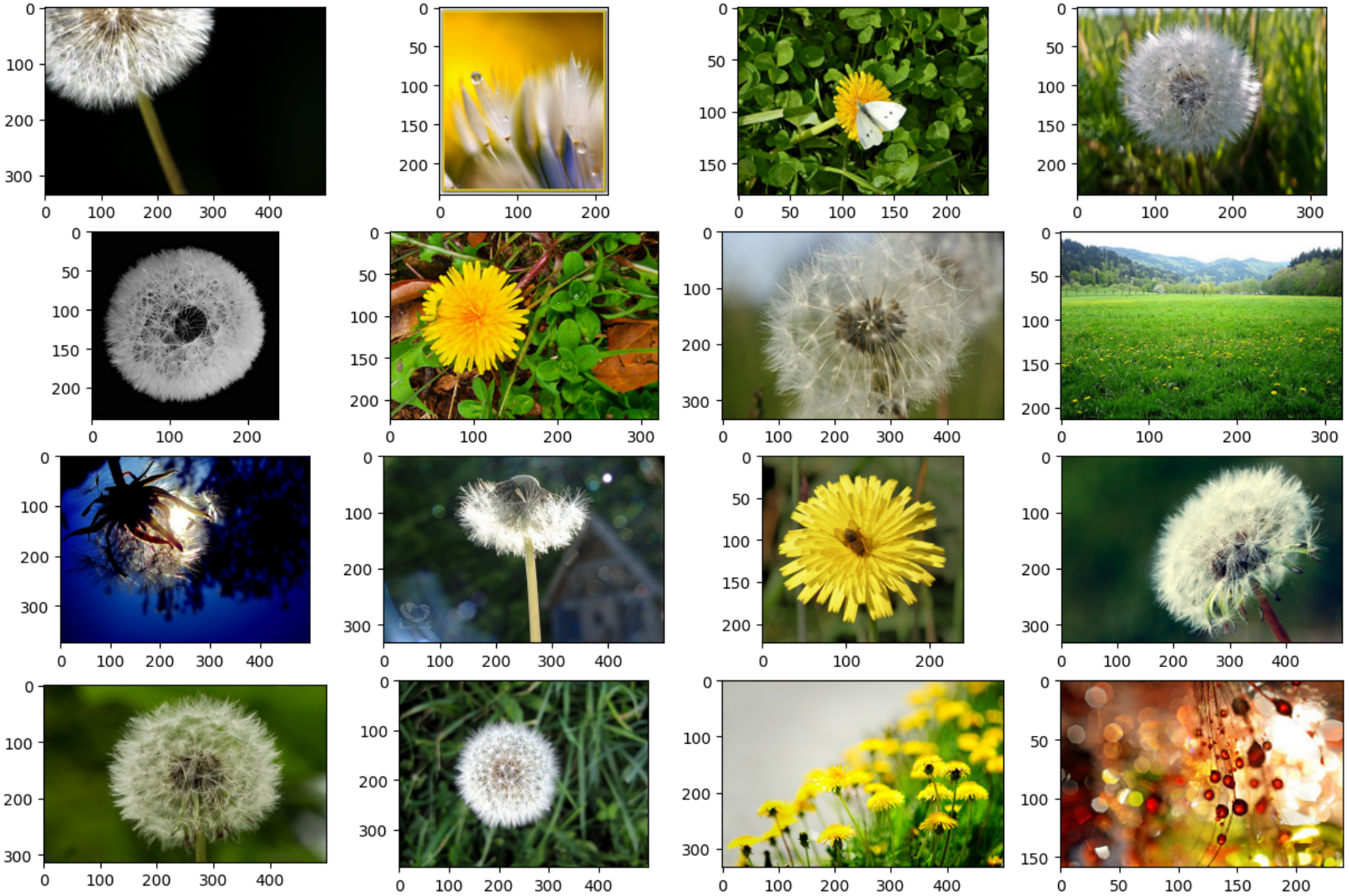
In [42]:
```
number_of_samples_per_class['dandelion'] = number_of_samples_in_dandelion_class
```

In [43]:
```
number_of_samples_per_class
```

Out[43]:
```
{'daisy': 761, 'dandelion': 1051}
```

In [44]:
```
'''
Lets plot 16 random dandelion samples in matplotlib subplot.
'''

sixteen_dandelion_samples = random.sample(os.listdir(path+'/dandelion'), 16)

fig, axes = plt.subplots(4, 4, figsize=(15, 10))

print("16 Random dandelion image samples plot")
for i, img in tqdm(enumerate(sixteen_dandelion_samples)):
    ax1 = plt.subplot(4, 4, i+1)
    sample1 = os.path.join(path+"/dandelion", img)
    sample1 = Image.open(sample1)
    ax1.imshow(sample1)
```

```
16 Random dandelion image samples plot
16it [00:00, 243.28it/s]
```



Likewise, there are a variety of image sizes in this class.

For rose class

In [45]:
```
# loading the first sample from rose class
```

```
sample1 = os.listdir(path+'/rose')[0]
sample1 = os.path.join(path+"/rose", sample1)

print("rose")
rose = Image.open(sample1)
print("Image Shape: ", np.array(rose).shape)
rose
```

```
rose
Image Shape:  (240, 179, 3)
```

Out[45]:

This is the first sample present in the rose class present in the dataset. We have used pillow library to import and present the image. It can be further seen that the image has 3 channels with height and width of 325 and 500 pixels respectively.

In [46]:
```
number_of_samples_in_rose_class = len(os.listdir(path+'/rose'));number_of_samples_in_rose_class
```

Out[46]:
```
783
```

Similarly, there are 783 samples in this class.

In [47]:
```
number_of_samples_per_class['rose'] = number_of_samples_in_rose_class
```

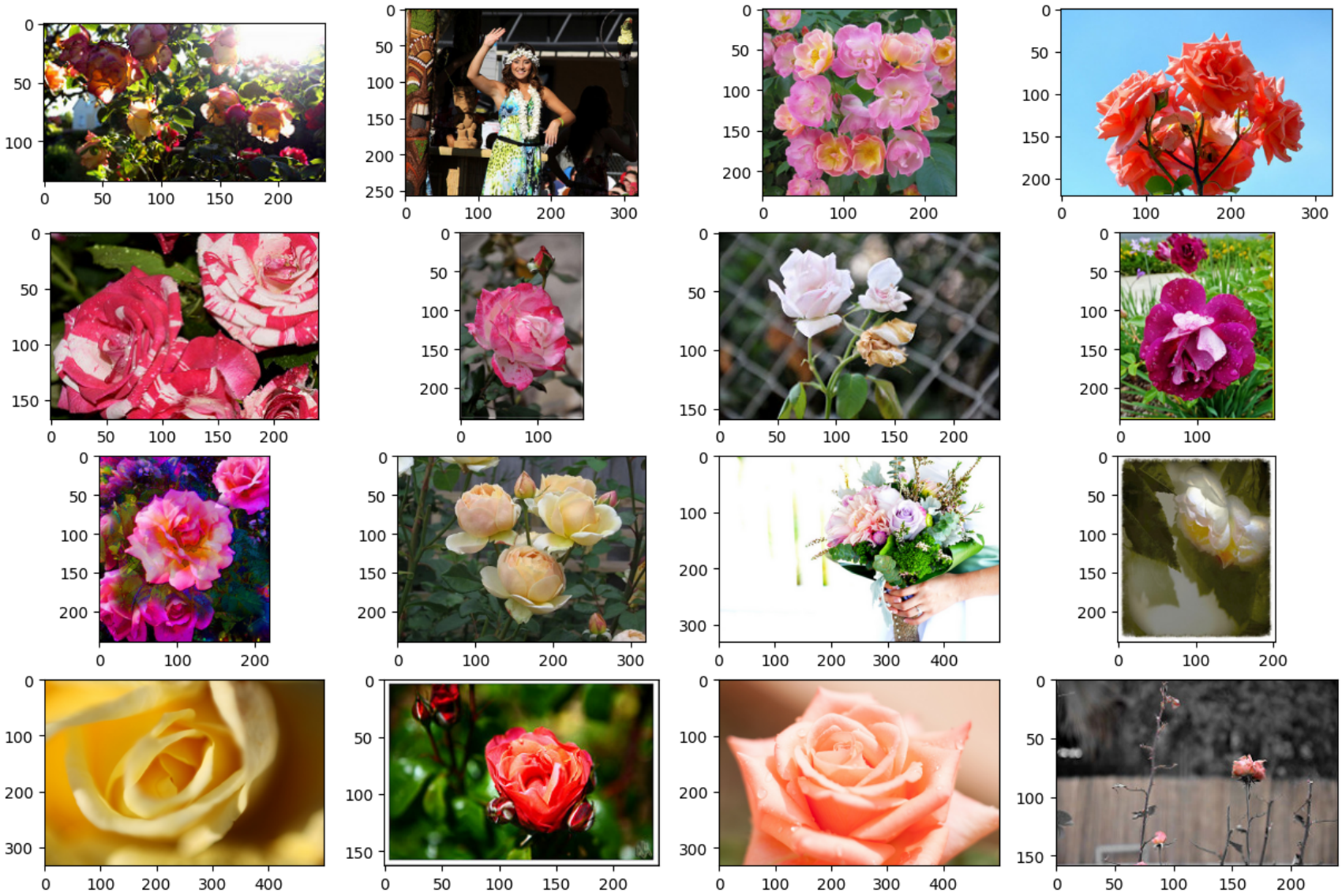
In [48]:
```
number_of_samples_per_class
```

Out[48]:
```
{'daisy': 761, 'dandelion': 1051, 'rose': 783}
```

In [49]:
```
'''
Lets plot 16 random dandelion samples in matplotlib subplot.
```

```
'''

sixteen_rose_samples = random.sample(os.listdir(path+'/rose'), 16)
fig, axes = plt.subplots(4, 4, figsize=(15, 10))

print("16 Random rose image samples plot")
for i, img in tqdm(enumerate(sixteen_rose_samples)):
    ax1 = plt.subplot(4, 4, i+1)
    sample1 = os.path.join(path+"/rose", img)
    sample1 = Image.open(sample1)
    ax1.imshow(sample1)
```

```
16 Random rose image samples plot
16it [00:00, 300.77it/s]
```



There are a variety of images in this class as well.

For sunflower class

In [50]:
```
# loading the first sample from sunflower class
```

```
sample1 = os.listdir(path+'/sunflower')[0]
sample1 = os.path.join(path+"/sunflower", sample1)

print("sunflower")
sunflower = Image.open(sample1)
print("Image Shape: ", np.array(sunflower).shape)
sunflower
```

```
sunflower
Image Shape:  (330, 500, 3)
```

Out[50]:

This is the first sample present in the sunflower class present in the dataset. We have used pillow library to import and present the image. It can be further seen that the image has 3 channels with height and width of 240 and 320 pixels respectively.

In [51]:
```
number_of_samples_in_sunflower_class = len(os.listdir(path+'/sunflower'));number_of_samples_in_sunflower_class
```

Out[51]:
```
732
```

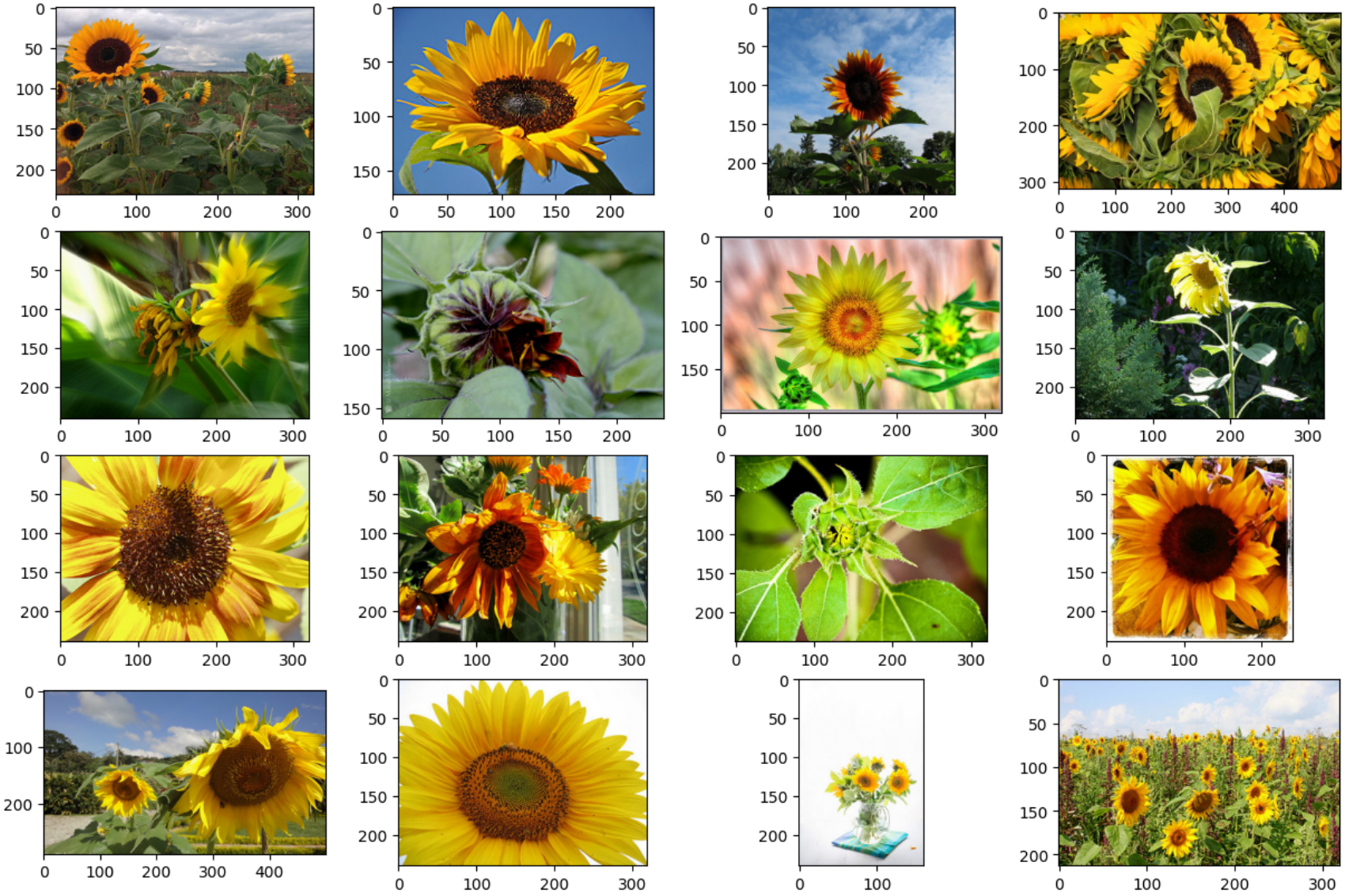
In [52]:
```
number_of_samples_per_class['sunflower'] = number_of_samples_in_sunflower_class
```

In [53]:
```
number_of_samples_per_class
```

Out[53]:
```
{'daisy': 761, 'dandelion': 1051, 'rose': 783, 'sunflower': 732}
```

In [54]:
```
'''
Lets plot 16 random sunflower samples in matplotlib subplot.
'''

sixteen_sunflower_samples = random.sample(os.listdir(path+'/sunflower'), 16)

fig, axes = plt.subplots(4, 4, figsize=(15, 10))

print("16 Random sunflower image samples plot")
for i, img in tqdm(enumerate(sixteen_sunflower_samples)):
    ax1 = plt.subplot(4, 4, i+1)
    sample1 = os.path.join(path+"/sunflower", img)
    sample1 = Image.open(sample1)
    ax1.imshow(sample1)
```

```
16 Random sunflower image samples plot
16it [00:00, 295.81it/s]
```



As we can see, there are a variety of sunflower images in this class.

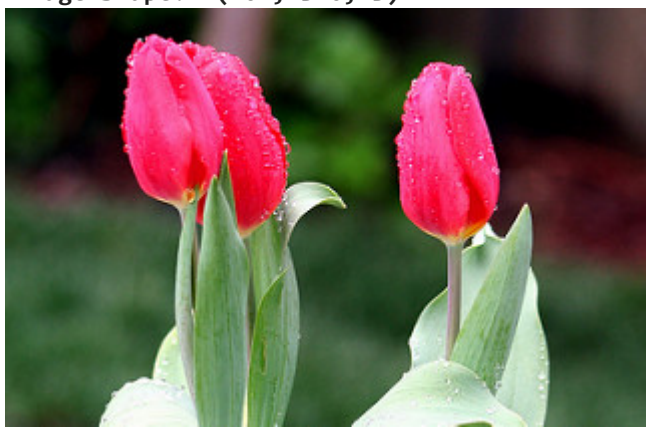
For tulip class

In [55]:
```
# loading the first sample from tulip class
```

```
sample1 = os.listdir(path+'/tulip')[0]
sample1 = os.path.join(path+"/tulip", sample1)

print("tulip")
tulip = Image.open(sample1)
print("Image Shape: ", np.array(tulip).shape)
tulip
```

```
tulip
Image Shape:  (209, 320, 3)
```

Out[55]:



This is the first sample present in the tulip class present in the dataset. We have used pillow library to import and present the image. It can be further seen that the image has 3 channels with height and width of 249 and 320 pixels respectively.

In [56]:
```
number_of_samples_in_tulip_class = len(os.listdir(path+'/tulip'));number_of_samples_in_tulip_class
```

Out[56]:
```
983
```

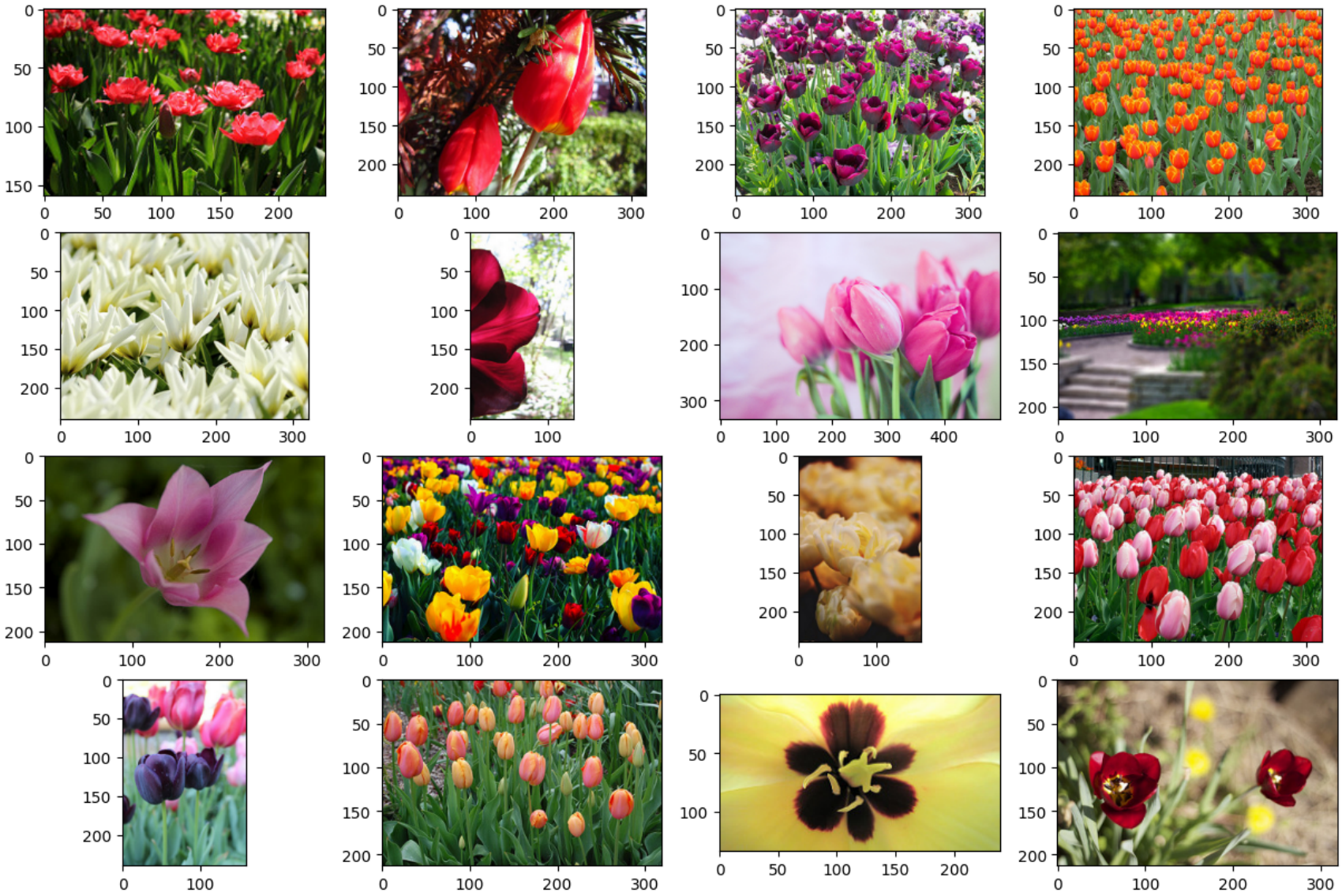
In [57]:
```
number_of_samples_per_class['tulip'] = number_of_samples_in_tulip_class
```

In [58]:
```
'''
Lets plot 16 random tulip samples in matplotlib subplot.
'''



sixteen_tulip_samples = random.sample(os.listdir(path+'/tulip'), 16)

fig, axes = plt.subplots(4, 4, figsize=(15, 10))

print("16 Random tulip image samples plot")
for i, img in tqdm(enumerate(sixteen_tulip_samples)):
```

```
    ax1 = plt.subplot(4, 4, i+1)
    sample1 = os.path.join(path+"/tulip", img)
    sample1 = Image.open(sample1)
    ax1.imshow(sample1)
```

16 Random tulip image samples plot

```
16it [00:00, 300.43it/s]
```



As we can see, this class also has a variety of images.

In [59]:
```
number_of_samples_per_class
```

Out[59]:
```
{'daisy': 761, 'dandelion': 1051, 'rose': 783, 'sunflower': 732, 'tulip': 983}
```

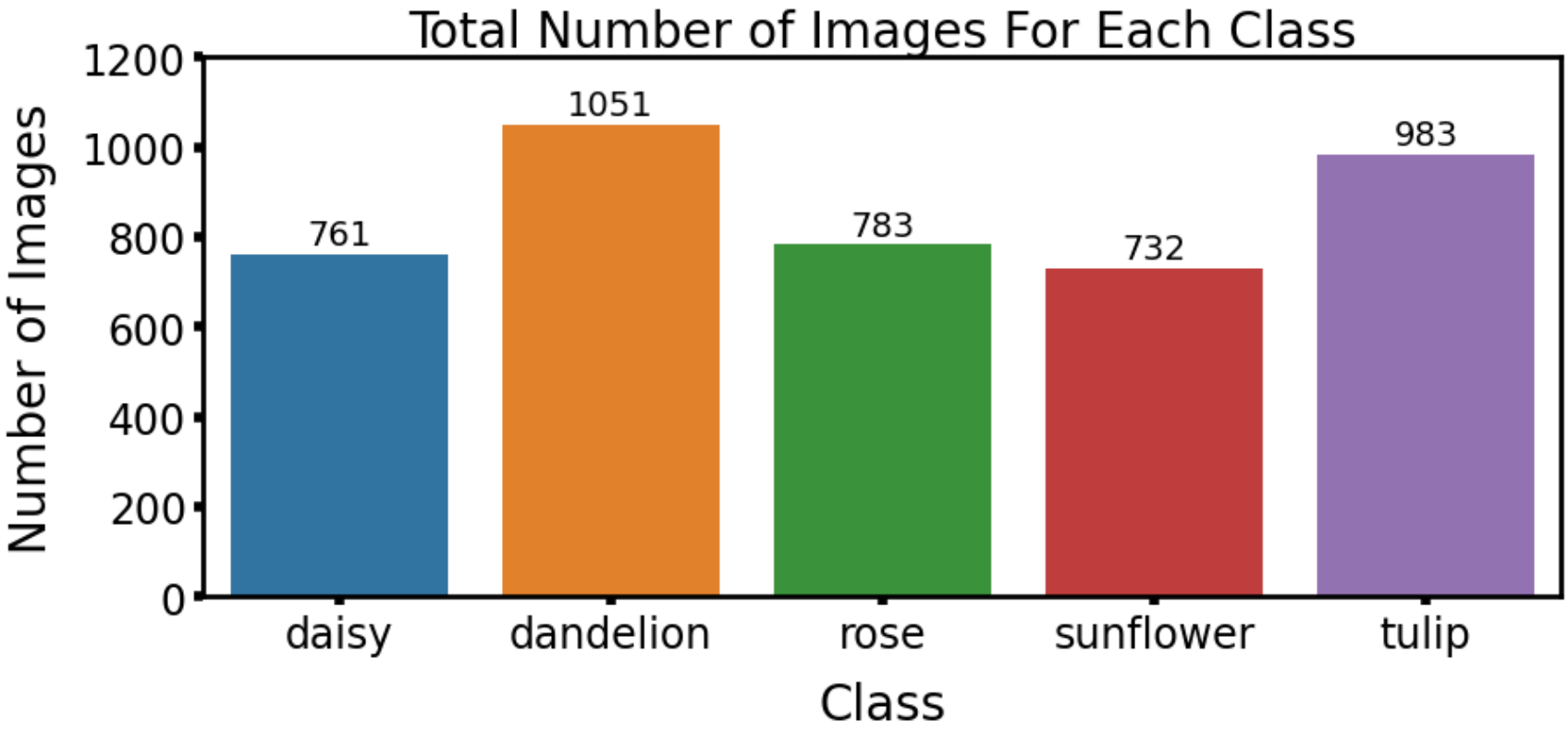
In [60]:
```
number_of_samples = pd.DataFrame(np.array(list(number_of_samples_per_class.values()))).T
number_of_samples.columns = list(number_of_samples_per_class.keys())
number_of_samples
```

Out[60]:

	daisy	dandelion	rose	sunflower	tulip
0	761	1051	783	732	983

In [61]:
```
fig, axes = plt.subplots(1, 1, figsize=(10, 4))

ax1 = plt.subplot(1,1,1)
sns.barplot(y = list(number_of_samples_per_class.values()), x = list(number_of_samples_per_class.keys()),\
            ax = ax1)

ax1.text(0.5, 1.02, "Total Number of Images For Each Class",
    horizontalalignment='center',
    fontsize=20,
    transform = ax1.transAxes);

ax1.set_ylim([0, 1200])

# iterate through each bar and add the number on top
for i, bar in enumerate(ax1.containers):
    ax1.bar_label(bar, label_type='edge', fontsize=14, padding=1, labels=[list(number_of_samples_per_class.values())[i]


# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)

ax1.set_ylabel('Number of Images', fontsize = 20, labelpad = 10, fontweight = 'normal')
ax1.set_xlabel('Class', fontsize = 20, labelpad = 10, fontweight = 'normal')
```

```
ax1.tick_params(axis ='both', labelcolor = 'black',labelsize=17, width=4)
```



In [62]:
```
total_images_in_dataset = sum(list(number_of_samples_per_class.values()))
print("Total Images in Dataset is: {}".format(total_images_in_dataset))
```

```
Total Images in Dataset is: 4310
```

From the barplot above, we can cleary see that the dandelion class has the highest samples equaling 1051. Similarly, the number of samples in other classes are very similar. This clears any confusion about the imbalanceness in the data as we can cleary see that the data set is not imbalanced and all classes have very similar number of sampls respectively.

Calculating Mean and Standard Deviation for each Channel

In [63]:
```
# creating the train dataset and dataloader for calculating the mean and standard deviation of our samples
image_size = (224, 224)
```

```
batch_size = 64
CHANNELS_IMG = 3
```

In [34]:
```
''' Creating a pytorch dataset using imagefolder class .Also, resizing all images to 224 * 224 width and height.
Finally, converting the images into pytorch tensor '''

dataset = ImageFolder(path, transform=tt.Compose([
                                        tt.Resize(image_size),
                                        tt.ToTensor()]))

''' creating pytorch dataloader with batch size of 64 because we want to train our model in mini batches respectively''
dataset_dl = DataLoader(dataset, batch_size, shuffle=True, num_workers=4, pin_memory=True)
```

In [35]:
```
len(dataset) # the size of all samples accross all class in our daraset
```

Out[35]:
```
4312
```

In [36]:
```
def get_mean_and_std(dataloader):

    # Initializing variables which will be utilized to keep track of the sum of each channel and the number of mini-bat
    sum_across_each_channels, squared_sum_accross_each_channels, batch_count = 0, 0, 0

    # Iterate over each batch in the dataloader
    for images, labels in tqdm(dataloader):

        # Compute the sum of each channel over the batch, height and width dimensions
        sum_across_each_channels += torch.mean(images, dim=[0,2,3])

        # Compute the sum of the squared value of each channel over the batch, height and width dimensions
        squared_sum_accross_each_channels += torch.mean(images**2, dim=[0,2,3])

        # Increment the number of batches
        batch_count += 1

    # Compute the mean of each channel over all the batches
    mean_accross_channels = sum_across_each_channels / batch_count

    # Compute the standard deviation of each channel over all the batches
    # using the formula std = sqrt(E[X^2] - (E[X])^2)
    std_accross_channels = (squared_sum_accross_each_channels / batch_count - mean_accross_channels ** 2) ** 0.5

    # Return the computed mean and standard deviation as output
    return mean_accross_channels, std_accross_channels
```

In [36]:
```
mean, std = get_mean_and_std(dataset_dl)
```

```
100%|██████████| 68/68 [00:12<00:00,  5.30it/s]
```

In [37]:
```
print("The calculated mean is: ", mean)
print("The calculated standard deviation is: ", std)
```

```
The calculated mean is:  tensor([0.4588, 0.4198, 0.3006])
The calculated standard deviation is:  tensor([0.2941, 0.2635, 0.2864])
```

In [65]:
```
# mean = [0.4588, 0.4198, 0.3006]
# std = [0.2941, 0.2635, 0.2864]
```

Here, we have sucessfully calculated the mean and standard deviation of our images for each channels.

Now that we have mean and standard deviation of each channel in our data, we will normalize the pixels in each channel using these values.

We can normalize the pixels of our images using the calculated mean and standard deviation by subtracting each pizel with mean pixel and dividing by standard deviation accross each channel. This will result in resultant pixels centering around zero with a unit variance.
This will result in pixels accross each channel to be scaled into a consistent range. We do this to ensure that the pixels of individual channels don't disproportionately impact gradients in backpropagation. During backprogation, if one of the channels have much larger pixels, it will result in larger gradients as compared to other channels. This will result in one channel severy impacting the model's learning as compared to other channels. So, to ensure one channel does not disproportionately impact gradients, we normalize pixels accross each channel.

Train, Validation and Test Datasets

We create training set to train the model and validation set to validate our mode. We improve the performance of our model on unseen data. Finally, after optimizing the model, we can make final predictions in the test data

Creating Datasets

In [66]:
```
image_size = (224, 224) # our image tensors will be resized to 224 pixels
CHANNELS_IMG = 3
```

In [67]:
```
random_seed = 42
torch.manual_seed(random_seed);
```

In [68]:
```
''' Creating a pytorch dataset using imagefolder class .Also, resizing all images to 224 * 224 width and height.
Finally, converting the images into pytorch tensor.
Additionally, we will also normalize the input using calculated mean and standard deviations
'''

# following transformations will be performed on each image
transformations_to_perform = transform=tt.Compose([
                                tt.Resize(image_size),
                                tt.ToTensor(),
                                tt.Normalize(mean, std)])


# Training dataset
train_ds = ImageFolder(path, transformations_to_perform)
```

Here, data generators referets to different techniques used to generate data for training and validating the mode. For example, resizing the image, or normalizing the image and augmentation techniques.

In [69]:
```
train_ds
```

Out[69]:
```
Dataset ImageFolder
    Number of datapoints: 4310
    Root location: C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Train
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=None)
               ToTensor()
               Normalize(mean=[0.4588, 0.4198, 0.3006], std=[0.2941, 0.2635, 0.2864])
           )
```

In [70]:
```
classes = train_ds.classes;classes # classes in our data
```

Out[70]:
```
['daisy', 'dandelion', 'rose', 'sunflower', 'tulip']
```

Randomly splitting data into train, validation, and test set with sample ratio of 60:20:20

In [71]:
```
[int(len(train_ds)*0.8)+1, int(len(train_ds)*0.2)]
```

Out[71]:
```
[3449, 862]
```

In [72]:
```
'''
The original data will be split into training, validation, and testing set in the ratio of 60:20:20 respectively.
'''

train_sample_size = int(len(train_ds) * 0.6)
valid_sample_size = int(len(train_ds) * 0.2)
test_sample_size  = int(len(train_ds) * 0.2)
```

In [73]:
```
train_sample_size + valid_sample_size + test_sample_size, len(train_ds)
```

Out[73]:
```
(4310, 4310)
```

In [75]:
```
from torch.utils.data import random_split

# Split the original dataset into a training set and a validation/test set
train_val_ds, test_ds = random_split(train_ds, [train_sample_size + valid_sample_size, test_sample_size])

# Split the training set into a training set and a validation set
train_ds, val_ds = random_split(train_val_ds, [len(train_val_ds)-valid_sample_size, valid_sample_size])

# Print the sizes of the resulting datasets
print(f"Training set size: {len(train_ds)}")
print(f"Validation set size: {len(val_ds)}")
print(f"Test set size: {len(test_ds)}")
```

```
Training set size: 2586
Validation set size: 862
Test set size: 862
```

In [78]:
```
2586 + 862 + 862
```

Out[78]:
```
4310
```

Number of samples in each class in training dataset

In [79]:
```
from collections import defaultdict

# Define a dictionary to store the counts of each class
```

```
class_counts = defaultdict(int)

# Loop through the dataset and count the number of samples present in each class
for x, y in train_ds:
    class_counts[y] += 1

# Print the class counts
for c, count in class_counts.items():
    print(f"Class {c}: {count} samples")
```

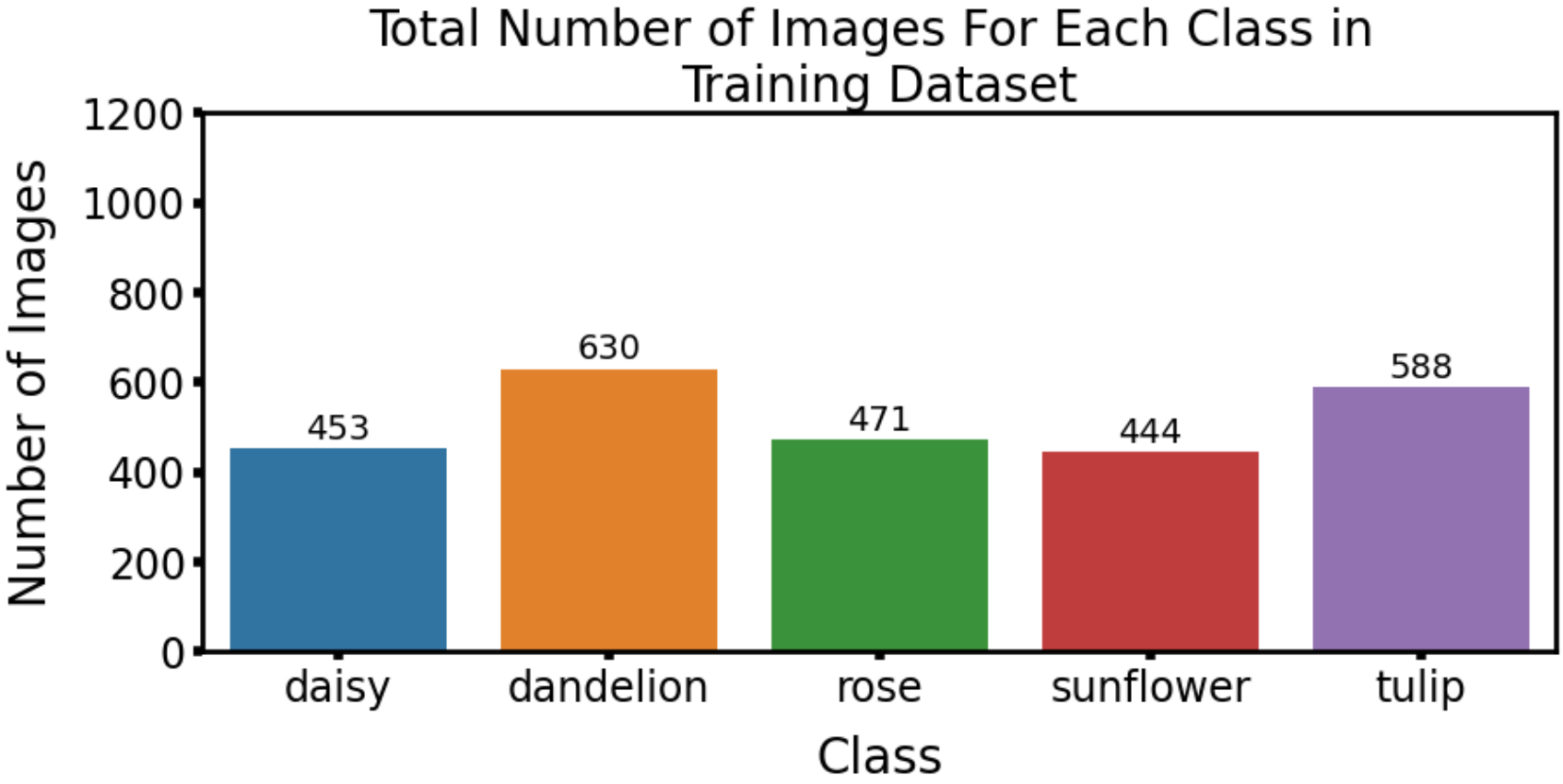
```
Class 1: 630 samples
Class 0: 453 samples
Class 2: 471 samples
Class 4: 588 samples
Class 3: 444 samples
```

In [80]:
```
# Sort the dictionary based on keys
sorted_dict = dict(sorted(dict(class_counts).items()));sorted_dict
```

Out[80]:
```
{0: 453, 1: 630, 2: 471, 3: 444, 4: 588}
```

In [83]:
```
fig, axes = plt.subplots(1, 1, figsize=(10, 4))

ax1 = plt.subplot(1,1,1)
sns.barplot(y = list(sorted_dict.values()), x = [classes[i] for i in list(sorted_dict.keys())],\
            ax = ax1)

ax1.text(0.5, 1.02, "Total Number of Images For Each Class in \nTraining Dataset",
    horizontalalignment='center',
    fontsize=20,
    transform = ax1.transAxes);

ax1.set_ylim([0, 1200])

# iterate through each bar and add the number on top
for i, bar in enumerate(ax1.containers):
    ax1.bar_label(bar, label_type='edge', fontsize=14, padding=1, labels=list(sorted_dict.values()))


# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
```

```
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)

ax1.set_ylabel('Number of Images', fontsize = 20, labelpad = 10, fontweight = 'normal')
ax1.set_xlabel('Class', fontsize = 20, labelpad = 10, fontweight = 'normal')

ax1.tick_params(axis ='both', labelcolor = 'black',labelsize=17, width=4)
```



There does not seem to be too much difference in number of samples in each class in training dataset. Next, we will perform 3 different types of augmentation techniques for all images in training set and concatenate 3 augmented datasets to the original dataset. This will effectively increase our training dataset 4 times.

In [86]:
```
def denormalize_images(image, mean_, std_):
    '''
    This function will denormalize pixels. it undos the standard scaling that we performed accross each channel using
```

```
    calculated mean and standard deviation
    '''
    mean = torch.tensor(mean_).reshape(1, 3, 1, 1)
    standard_deviation = torch.tensor(std_).reshape(1, 3, 1, 1)

    images_that_are_denormalized = image * standard_deviation + mean
    return images_that_are_denormalized

def show_first25_images(dataset, batch_size=25):
    # Create a dataloader from the dataset
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=0)

    # Iterate over the batches and display the images
    for images, labels in dataloader:
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.set_xticks([]) # sisabling the pixels in x and y axis labels
        ax.set_yticks([])
        images_that_are_denormalized = denormalize_images(images, mean, std)
        ax.imshow(make_grid(images_that_are_denormalized, nrow=5).permute(1, 2, 0))
        break # breaking after the first batch by plotting first 5 rows and columns in the grid using makegrid function
```

In [87]:
```
show_first25_images(train_ds) # 25 images randomly plotted in a grid using trainiig set
```

Creating 3 different copies of train_ds and perform 3 different augmentations and concatenate them

In [88]:
```
# Define a custom dataset class that inherits from torch.utils.data.Dataset
class CreateTransformedDataset(Dataset):
    def __init__(self, my_list):
        self.data = my_list

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
```

```
        image, label = self.data[idx]
        return image, label
```

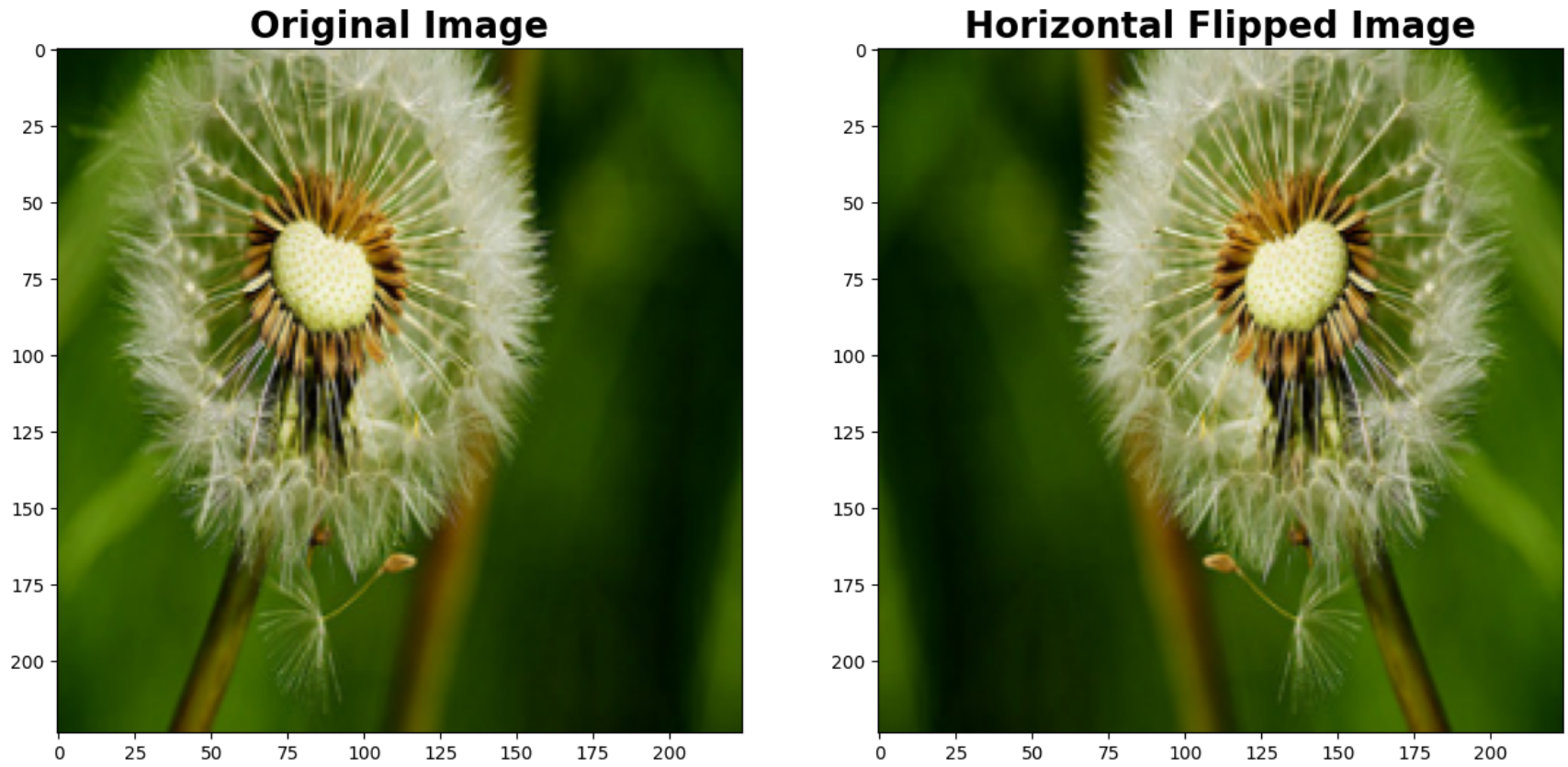
Applying Random Horizontal Flip with probability 1 to each image in original training dataset

In [89]:
```
train_ds_RandomHorizontalFlip = train_ds
transform = tt.RandomHorizontalFlip(p=1)

# Apply the transformation to each item in the dataset
transformed_data = [(transform(image), label) for image, label in tqdm(train_ds_RandomHorizontalFlip)]
train_ds_RandomHorizontalFlip = CreateTransformedDataset(transformed_data)
```

```
100%|██████████| 2586/2586 [00:09<00:00, 283.69it/s]
```

In [93]:
```
fig, axes = plt.subplots(1, 2, figsize=(15, 20))

ax = plt.subplot(1, 2, 1)
ax.imshow(denormalize_images(train_ds[250][0], mean, std)[0].permute((1, 2, 0)))
ax.set_title("Original Image", fontsize = 20, fontweight = 'bold')

ax = plt.subplot(1, 2, 2)
ax.imshow(denormalize_images(train_ds_RandomHorizontalFlip[250][0], mean, std)[0].permute((1, 2, 0)))
ax.set_title("Horizontal Flipped Image", fontsize = 20, fontweight = 'bold')
```

Out[93]:
```
Text(0.5, 1.0, 'Horizontal Flipped Image')
```



The image on the left is the original image in training set. The right one is the corresponding image that is horizontally flipped. We will use this augmentation to increse the size of our training data.

Applying Random Vertical Flip with probability 1 to each image in original training dataset

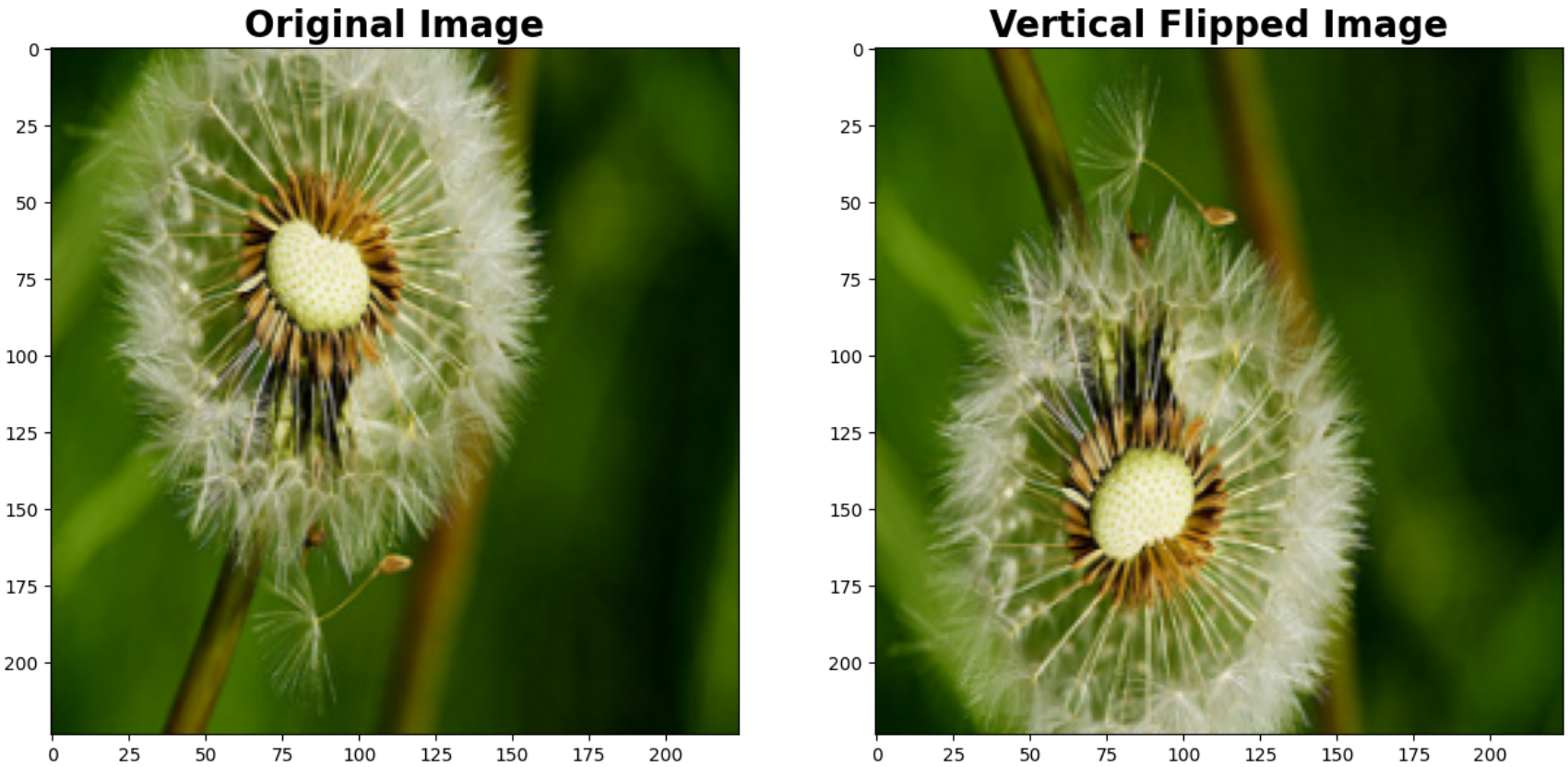
In [94]:
```
train_ds_RandomVerticalFlip = train_ds
transform = tt.RandomVerticalFlip(p=1)

# Apply the transformation to each item in the dataset
transformed_data = [(transform(image), label) for image, label in tqdm(train_ds_RandomVerticalFlip)]
train_ds_RandomVerticalFlip = CreateTransformedDataset(transformed_data)
```

```
100%|██████████| 2586/2586 [00:09<00:00, 278.25it/s]
```

In [95]:
```
fig, axes = plt.subplots(1, 2, figsize=(15, 20))
```

```
ax = plt.subplot(1, 2, 1)
ax.imshow(denormalize_images(train_ds[250][0], mean, std)[0].permute((1, 2, 0)))
ax.set_title("Original Image", fontsize = 20, fontweight = 'bold')

ax = plt.subplot(1, 2, 2)
ax.imshow(denormalize_images(train_ds_RandomVerticalFlip[250][0], mean, std)[0].permute((1, 2, 0)))
ax.set_title("Vertical Flipped Image", fontsize = 20, fontweight = 'bold')
```

Out[95]:
```
Text(0.5, 1.0, 'Vertical Flipped Image')
```



The image on the left is the original image in training set. The right one is the corresponding image that is vretically flipped. We will use this augmentation to increse the size of our training data.

Applying Random Rotation with degrees (-45, 45) to each image in original training dataset

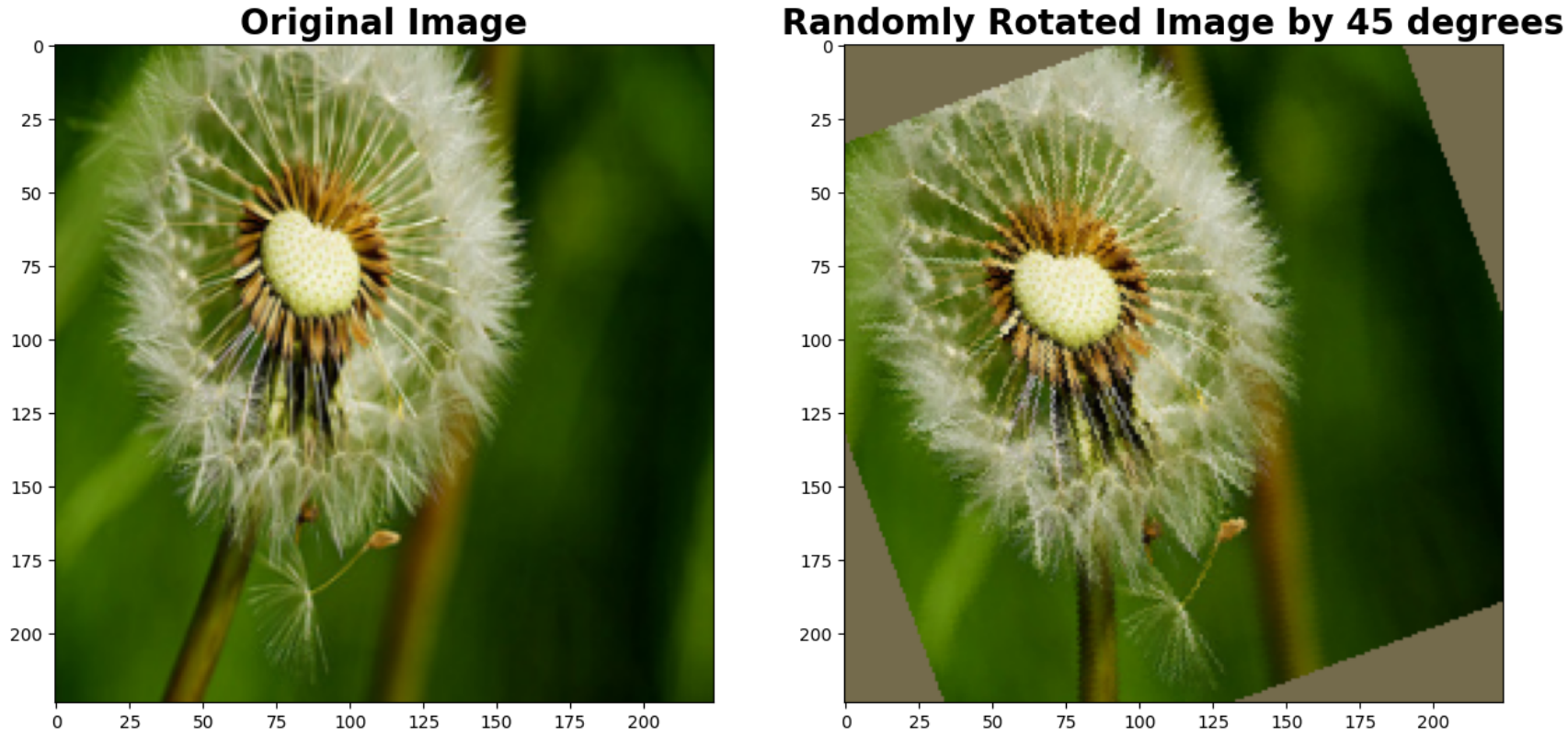
In [96]:
```
train_ds_RandomRotation = train_ds
transform = tt.RandomRotation(degrees = 45)
```

```
# Apply the transformation to each item in the dataset
transformed_data = [(transform(image), label) for image, label in tqdm(train_ds_RandomRotation)]
train_ds_RandomRotation = CreateTransformedDataset(transformed_data)
```

```
100%|██████████| 2586/2586 [00:21<00:00, 120.79it/s]
```

In [97]:
```
fig, axes = plt.subplots(1, 2, figsize=(15, 20))

ax = plt.subplot(1, 2, 1)
ax.imshow(denormalize_images(train_ds[250][0], mean, std)[0].permute((1, 2, 0)))
ax.set_title("Original Image", fontsize = 20, fontweight = 'bold')

ax = plt.subplot(1, 2, 2)
ax.imshow(denormalize_images(train_ds_RandomRotation[250][0], mean, std)[0].permute((1, 2, 0)))
ax.set_title("Randomly Rotated Image by 45 degrees", fontsize = 20, fontweight = 'bold')
```

Out[97]:
```
Text(0.5, 1.0, 'Randomly Rotated Image by 45 degrees')
```



The image on the left is the original image in training set. The right one is the corresponding image that is randomly rotated. We will use this augmentation to increse the size of our training data.

Concatenating all training datasets

In [98]:
```
# concatenate the datasets into one
train_ds_concatenated = ConcatDataset([train_ds, train_ds_RandomHorizontalFlip, train_ds_RandomVerticalFlip, \
                                       train_ds_RandomRotation])
```

In [99]:
```
len(train_ds_concatenated)
```

Out[99]:
```
10344
```

In [100…
```
len(train_ds) * 4
```

Out[100]:
```
10344
```

our original dataset has been scaled by 4 times

In [101…
```
print("Final Augmented and Concatenated Training Dataset")
show_first25_images(train_ds_concatenated)
```

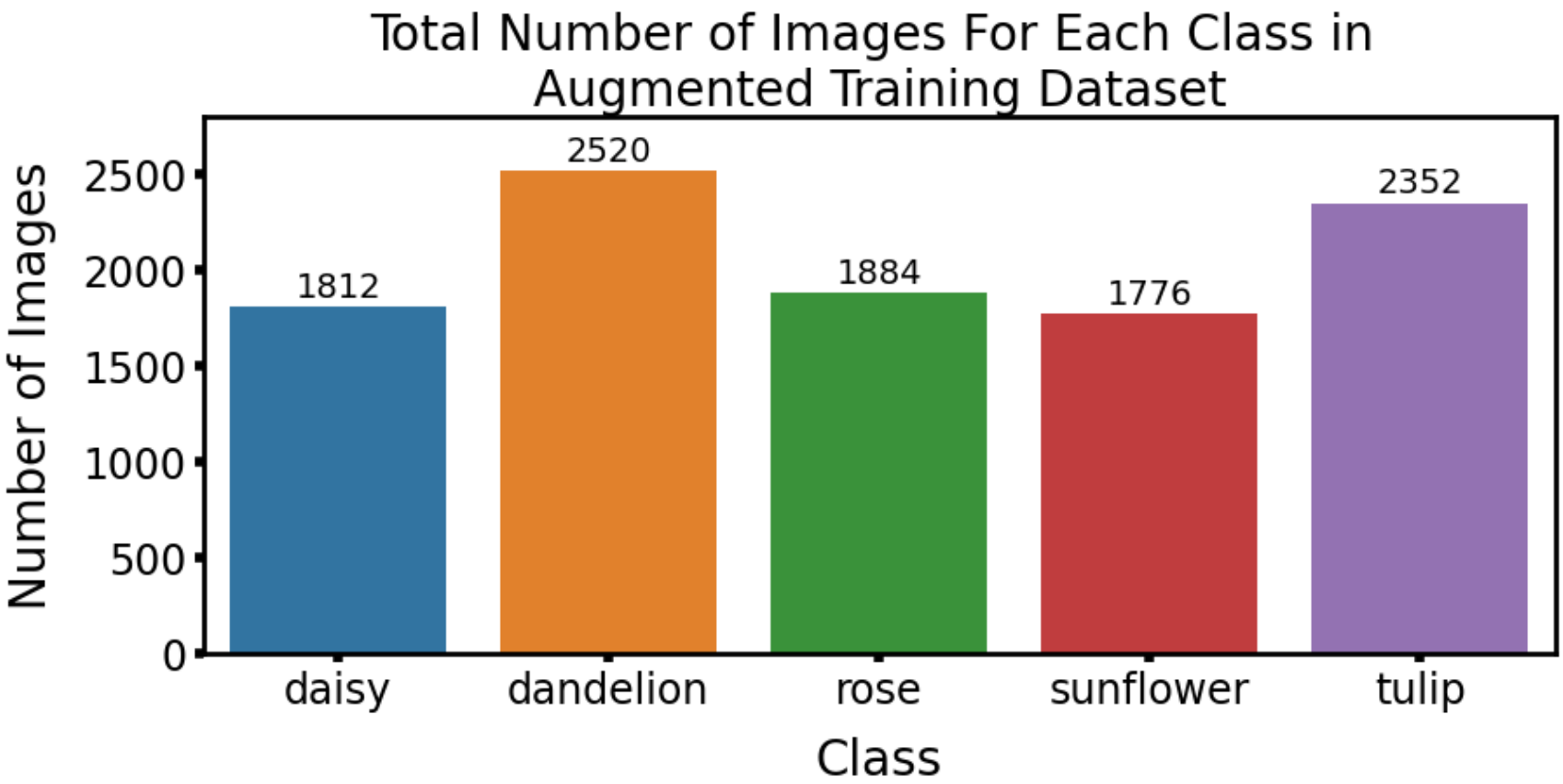
```
Final Augmented and Concatenated Training Dataset
```

In [102…

```
# Define a dictionary to store the counts of each class
class_counts = defaultdict(int)

# Loop through the dataset and count the number of instances in each class
for x, y in tqdm(train_ds_concatenated):
    class_counts[y] += 1

# Print the class counts
for c, count in class_counts.items():
    print(f"Class {c}: {count} samples")

# Sort the dictionary based on keys
sorted_dict = dict(sorted(dict(class_counts).items()));sorted_dict
```

```
fig, axes = plt.subplots(1, 1, figsize=(10, 4))

ax1 = plt.subplot(1,1,1)
sns.barplot(y = list(sorted_dict.values()), x = [classes[i] for i in list(sorted_dict.keys())],\
            ax = ax1)

ax1.text(0.5, 1.02, "Total Number of Images For Each Class in \nAugmented Training Dataset",
    horizontalalignment='center',
    fontsize=20,
    transform = ax1.transAxes);

ax1.set_ylim([0, 2800])

# iterate through each bar and add the number on top
for i, bar in enumerate(ax1.containers):
    ax1.bar_label(bar, label_type='edge', fontsize=14, padding=1, labels=list(sorted_dict.values()))


# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)

ax1.set_ylabel('Number of Images', fontsize = 20, labelpad = 10, fontweight = 'normal')
ax1.set_xlabel('Class', fontsize = 20, labelpad = 10, fontweight = 'normal')

ax1.tick_params(axis ='both', labelcolor = 'black',labelsize=17, width=4)
```

```
100%|██████████████████████████████████████████████████████████████████████████| 10344/10344 [00:08<00:00, 1155.51it/s]
Class 1: 2520 samples
Class 0: 1812 samples
Class 2: 1884 samples
Class 4: 2352 samples
Class 3: 1776 samples
```



As we can clearly see, the number of samples in each of the class in training set has quadrupled thanks to 3 different augmentation techniques. We applied 3 different augmentation techniques to each of the samples in each class to create 3 new datasets for each augmentation. Finally, we concatenated them to create our finalized training set.

Number of samples in each class in validation dataset

In [103…

```
print("For Validation Dataset")
show_first25_images(val_ds)
```

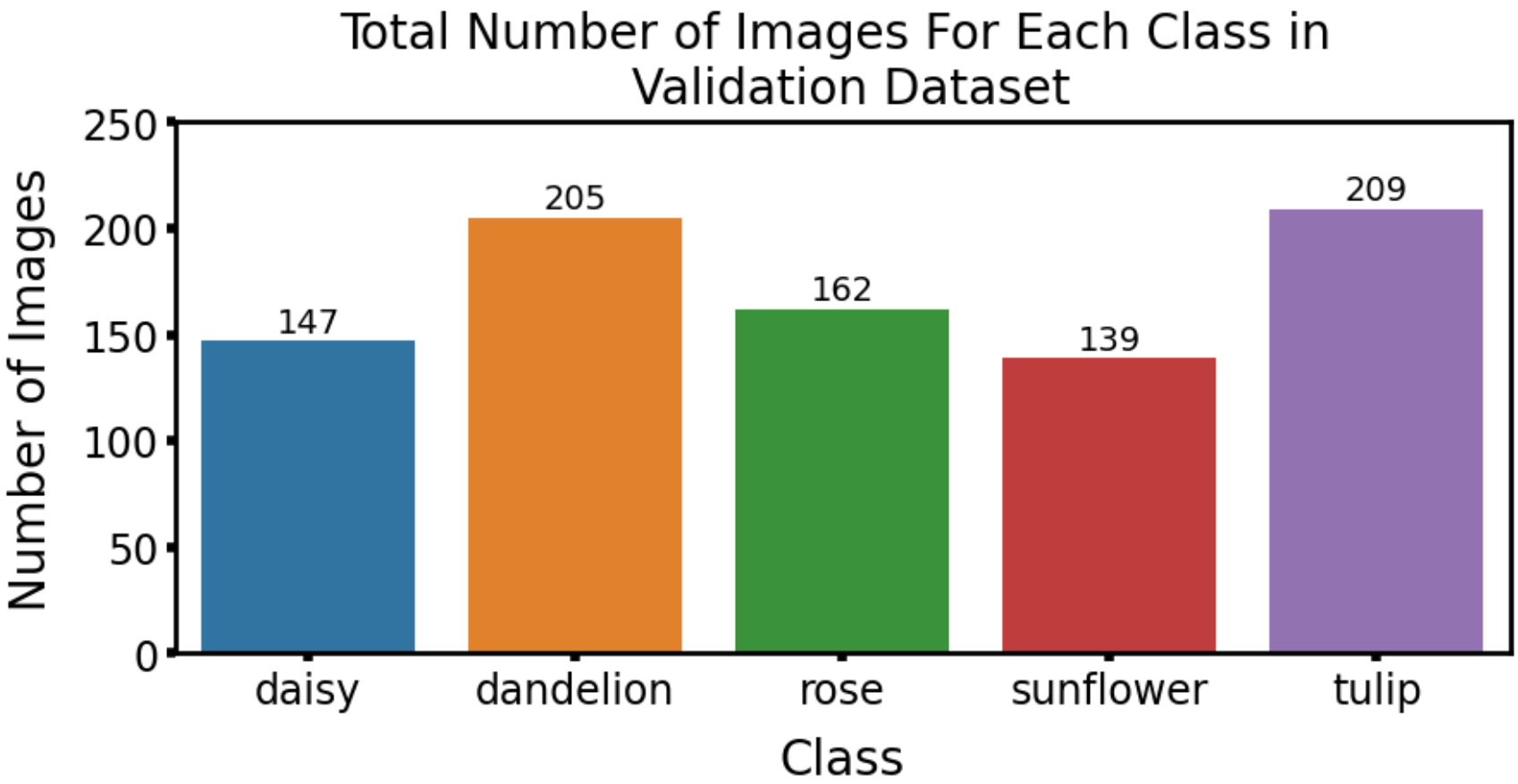
```
For Validation Dataset
```

In [104…

```
# Define a dictionary to store the counts of each class
class_counts = defaultdict(int)

# Loop through the dataset and count the number of instances in each class
for x, y in tqdm(val_ds):
    class_counts[y] += 1

# Print the class counts
for c, count in class_counts.items():
    print(f"Class {c}: {count} samples")

# Sort the dictionary based on keys
sorted_dict = dict(sorted(dict(class_counts).items()));sorted_dict
```

```
fig, axes = plt.subplots(1, 1, figsize=(10, 4))

ax1 = plt.subplot(1,1,1)
sns.barplot(y = list(sorted_dict.values()), x = [classes[i] for i in list(sorted_dict.keys())],\
            ax = ax1)

ax1.text(0.5, 1.035, "Total Number of Images For Each Class in \n Validation Dataset",
    horizontalalignment='center',
    fontsize=20,
    transform = ax1.transAxes);

ax1.set_ylim([0, 250])

# iterate through each bar and add the number on top
for i, bar in enumerate(ax1.containers):
    ax1.bar_label(bar, label_type='edge', fontsize=14, padding=1, labels=list(sorted_dict.values()))


# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)

ax1.set_ylabel('Number of Images', fontsize = 20, labelpad = 10, fontweight = 'normal')
ax1.set_xlabel('Class', fontsize = 20, labelpad = 10, fontweight = 'normal')

ax1.tick_params(axis ='both', labelcolor = 'black',labelsize=17, width=4)
```

```
100%|██████████| 862/862 [00:03<00:00, 263.00it/s]
Class 0: 147 samples
Class 3: 139 samples
Class 1: 205 samples
Class 4: 209 samples
Class 2: 162 samples
```



As we can see, the number of samples for each class in validation dataset is fairly balanced.

Number of samples in each class in test dataset

In [105…

```
print("For Test Dataset")
show_first25_images(test_ds)
```

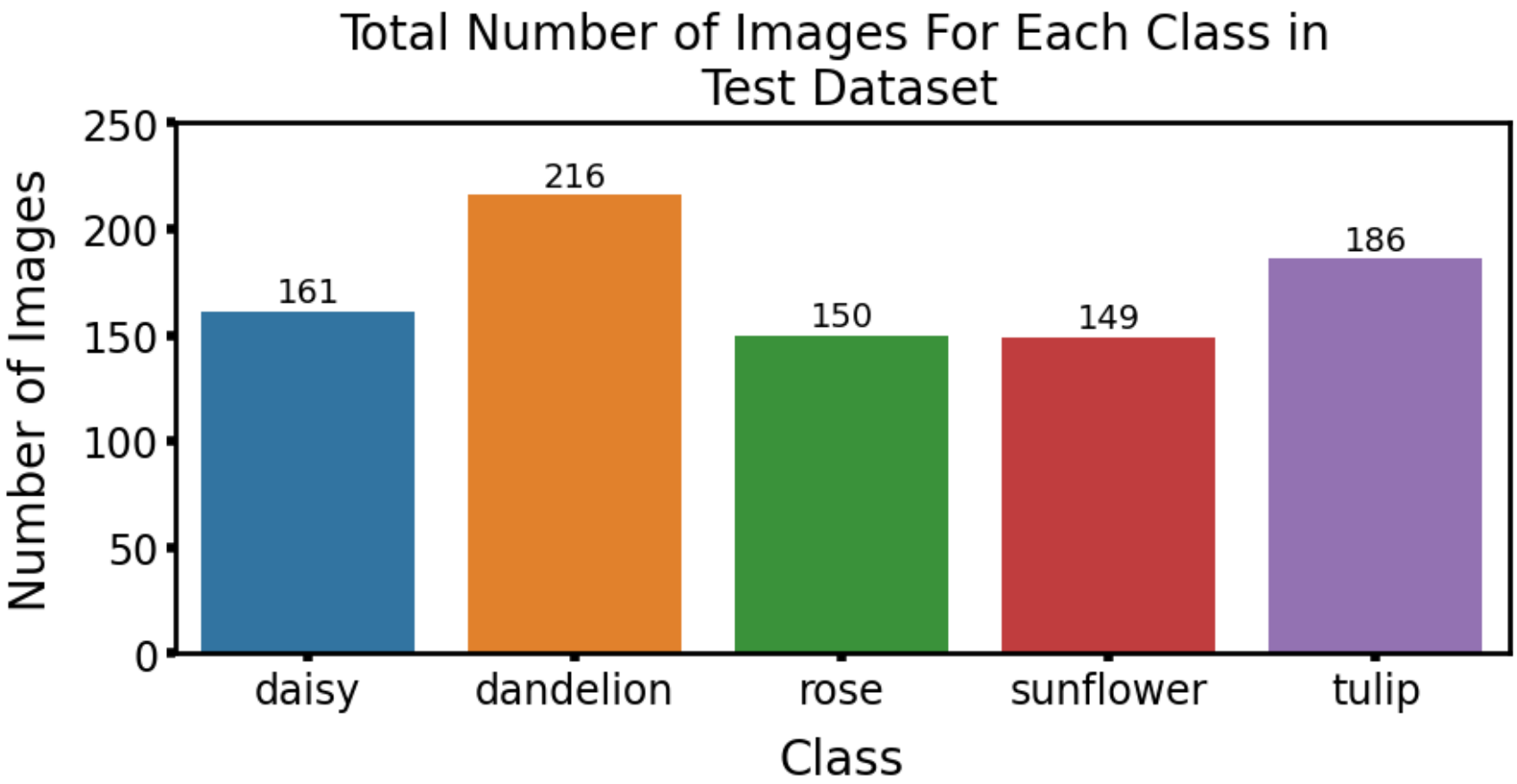
```
For Test Dataset
```

In [107…

```
# Define a dictionary to store the counts of each class
class_counts = defaultdict(int)

# Loop through the dataset and count the number of instances in each class
for x, y in tqdm(test_ds):
    class_counts[y] += 1

# Print the class counts
for c, count in class_counts.items():
    print(f"Class {c}: {count} samples")

# Sort the dictionary based on keys
sorted_dict = dict(sorted(dict(class_counts).items()));sorted_dict
```

```
fig, axes = plt.subplots(1, 1, figsize=(10, 4))

ax1 = plt.subplot(1,1,1)
sns.barplot(y = list(sorted_dict.values()), x = [classes[i] for i in list(sorted_dict.keys())],\
            ax = ax1)

ax1.text(0.5, 1.035, "Total Number of Images For Each Class in \n Test Dataset",
    horizontalalignment='center',
    fontsize=20,
    transform = ax1.transAxes);

ax1.set_ylim([0, 250])

# iterate through each bar and add the number on top
for i, bar in enumerate(ax1.containers):
    ax1.bar_label(bar, label_type='edge', fontsize=14, padding=1, labels=list(sorted_dict.values()))


# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)

ax1.set_ylabel('Number of Images', fontsize = 20, labelpad = 10, fontweight = 'normal')
ax1.set_xlabel('Class', fontsize = 20, labelpad = 10, fontweight = 'normal')

ax1.tick_params(axis ='both', labelcolor = 'black',labelsize=17, width=4)
```

```
100%|██████████████████████████████████████████████████████████████████████████| 862/862 [00:03<00:00, 253.73it/s]
Class 2: 150 samples
Class 3: 149 samples
Class 4: 186 samples
Class 1: 216 samples
Class 0: 161 samples
```



As we can see, the number of samples accross each class in test dataset is also fairly balanced.

Creating DataLoaders and Moving them to GPU

In [73]:
```
'''
Creating training dataloader to with batch size = 64 to perform mini batch gradient descent
'''

train_dl = DataLoader(train_ds_concatenated, batch_size, shuffle=True, num_workers = 2)
```

In [74]:
```
'''
Creating validation and test dataloader to with batch size = 64 to perform mini batch gradient descent
'''
```

```
val_dl = DataLoader(val_ds, batch_size, shuffle=True, num_workers = 2)
test_dl = DataLoader(test_ds, batch_size, shuffle=True, num_workers = 2)
```

Setting num_workers=-1 in the PyTorch DataLoader constructor will use all available CPU cores to load the data in parallel. In other words, it will automatically determine the number of workers to use based on the available CPU resources on your machine.

In [75]:
```
del dataset, train_ds_concatenated, train_ds, val_ds, test_ds
```

Building the Model

In [7]:
```
'''
This is a VGG19 model created using PyTorch. Additionally, batch normalization has been integrated after each convoluti
- further reduce internal covariate shift, which can help model converge faster
'''

class VGG19(nn.Module):
    def __init__(self, num_classes=5):
        super(VGG19, self).__init__()
        self.features = nn.Sequential(
                nn.Conv2d(3, 64, kernel_size=3, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.Conv2d(64, 64, kernel_size=3, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.Conv2d(64, 128, kernel_size=3, padding=1),
                nn.BatchNorm2d(128),
                nn.ReLU(inplace=True),
                nn.Conv2d(128, 128, kernel_size=3, padding=1),
                nn.BatchNorm2d(128),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.Conv2d(128, 256, kernel_size=3, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),
                nn.Conv2d(256, 256, kernel_size=3, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),
                nn.Conv2d(256, 256, kernel_size=3, padding=1),
```

```
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))

        ''' FULLY CONNECTED LAYER '''
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.BatchNorm1d(4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.BatchNorm1d(4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
```

```
                nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1) # flattening into a 1-d array to pass into the multilayer perceptron
        x = self.classifier(x)
        return x

model = VGG19()
```

In [8]:
```
model
```

Out[8]:
```
VGG19(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (25): ReLU(inplace=True)
    (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (27): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (34): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (35): ReLU(inplace=True)
    (36): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (37): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (38): ReLU(inplace=True)
    (39): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (42): ReLU(inplace=True)
```

```
    (43): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (44): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (45): ReLU(inplace=True)
    (46): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (47): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (48): ReLU(inplace=True)
    (49): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (50): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (51): ReLU(inplace=True)
    (52): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): Dropout(p=0.5, inplace=False)
    (8): Linear(in_features=4096, out_features=5, bias=True)
  )
)
```

In [10]:
```
model = model.to('cuda') # moving the model into gpu
```

In [11]:
```
print(model.parameters().__next__().device)
```

```
cuda:0
```

In [101…
```
''' Summary of the VGG19 architecture '''
summary(model, (3, 224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
       BatchNorm2d-2         [-1, 64, 224, 224]             128
              ReLU-3         [-1, 64, 224, 224]               0
            Conv2d-4         [-1, 64, 224, 224]          36,928
       BatchNorm2d-5         [-1, 64, 224, 224]             128
              ReLU-6         [-1, 64, 224, 224]               0
         MaxPool2d-7         [-1, 64, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]          73,856
       BatchNorm2d-9        [-1, 128, 112, 112]             256
             ReLU-10        [-1, 128, 112, 112]               0
           Conv2d-11        [-1, 128, 112, 112]         147,584
      BatchNorm2d-12        [-1, 128, 112, 112]             256
             ReLU-13        [-1, 128, 112, 112]               0
        MaxPool2d-14          [-1, 128, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         295,168
      BatchNorm2d-16          [-1, 256, 56, 56]             512
             ReLU-17          [-1, 256, 56, 56]               0
           Conv2d-18          [-1, 256, 56, 56]         590,080
      BatchNorm2d-19          [-1, 256, 56, 56]             512
             ReLU-20          [-1, 256, 56, 56]               0
           Conv2d-21          [-1, 256, 56, 56]         590,080
      BatchNorm2d-22          [-1, 256, 56, 56]             512
             ReLU-23          [-1, 256, 56, 56]               0
           Conv2d-24          [-1, 256, 56, 56]         590,080
      BatchNorm2d-25          [-1, 256, 56, 56]             512
             ReLU-26          [-1, 256, 56, 56]               0
        MaxPool2d-27          [-1, 256, 28, 28]               0
           Conv2d-28          [-1, 512, 28, 28]       1,180,160
      BatchNorm2d-29          [-1, 512, 28, 28]           1,024
             ReLU-30          [-1, 512, 28, 28]               0
           Conv2d-31          [-1, 512, 28, 28]       2,359,808
      BatchNorm2d-32          [-1, 512, 28, 28]           1,024
             ReLU-33          [-1, 512, 28, 28]               0
           Conv2d-34          [-1, 512, 28, 28]       2,359,808
      BatchNorm2d-35          [-1, 512, 28, 28]           1,024
             ReLU-36          [-1, 512, 28, 28]               0
           Conv2d-37          [-1, 512, 28, 28]       2,359,808
      BatchNorm2d-38          [-1, 512, 28, 28]           1,024
             ReLU-39          [-1, 512, 28, 28]               0
        MaxPool2d-40          [-1, 512, 14, 14]               0
           Conv2d-41          [-1, 512, 14, 14]       2,359,808
      BatchNorm2d-42          [-1, 512, 14, 14]           1,024
```

```
              ReLU-43         [-1, 512, 14, 14]               0
            Conv2d-44         [-1, 512, 14, 14]       2,359,808
       BatchNorm2d-45         [-1, 512, 14, 14]           1,024
              ReLU-46         [-1, 512, 14, 14]               0
            Conv2d-47         [-1, 512, 14, 14]       2,359,808
       BatchNorm2d-48         [-1, 512, 14, 14]           1,024
              ReLU-49         [-1, 512, 14, 14]               0
            Conv2d-50         [-1, 512, 14, 14]       2,359,808
       BatchNorm2d-51         [-1, 512, 14, 14]           1,024
              ReLU-52         [-1, 512, 14, 14]               0
         MaxPool2d-53           [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-54           [-1, 512, 7, 7]               0
            Linear-55                [-1, 4096]     102,764,544
       BatchNorm1d-56                [-1, 4096]           8,192
              ReLU-57                [-1, 4096]               0
           Dropout-58                [-1, 4096]               0
            Linear-59                [-1, 4096]      16,781,312
       BatchNorm1d-60                [-1, 4096]           8,192
              ReLU-61                [-1, 4096]               0
           Dropout-62                [-1, 4096]               0
            Linear-63                   [-1, 5]          20,485
================================================================
Total params: 139,618,117
Trainable params: 139,618,117
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 352.05
Params size (MB): 532.60
Estimated Total Size (MB): 885.23
----------------------------------------------------------------
```

Model Training

In [89]:

```
'''

here, we are training our model for a total of 30 epochs. We are using adam optimizer along with mini-batch gradient de
to train our model. Performance of our model on training and validation datasets on each epoch is also recorded.

'''


epochs = 30
```

```
train_losses = []
train_accs = []
val_losses = []
val_accs = []

for epoch in range(epochs):
    train_loss = 0.0
    train_total = 0
    train_correct = 0

    # training loop
    for images, labels in tqdm(train_dl):
        images = images.to(device), labels = labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()

    train_loss /= len(train_dl)
    train_acc = train_correct / train_total
    train_losses.append(train_loss)
    train_accs.append(train_acc)

    val_loss = 0.0
    val_total = 0
    val_correct = 0

    # validation loop
    with torch.no_grad():
        model.eval()
        for images, labels in val_dl:
            images = images.to(device), labels = labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            val_total += labels.size(0)
```

```
            val_correct += (predicted == labels).sum().item()

        val_loss /= len(val_dl)
        val_acc = val_correct / val_total
        val_losses.append(val_loss)
        val_accs.append(val_acc)

    print(f"Epoch {epoch+1}:")
    print(f"Training Loss: {train_loss:.4f} | Training Accuracy: {train_acc*100:.4f}%")
    print(f"Validation Loss: {val_loss:.4f} | Validation Accuracy: {val_acc*100:.4f}%")

    model.train()  # set model back to training mode
```

```
100%|██████████| 162/162 [03:18<00:00,  1.23s/it]
Epoch 1:
Training Loss: 1.4500 | Training Accuracy: 41.3543%
Validation Loss: 1.3593 | Validation Accuracy: 48.4919%
100%|██████████| 162/162 [03:22<00:00,  1.25s/it]
Epoch 2:
Training Loss: 1.3020 | Training Accuracy: 48.4254%
Validation Loss: 1.1974 | Validation Accuracy: 49.6520%
100%|██████████| 162/162 [03:23<00:00,  1.26s/it]
Epoch 3:
Training Loss: 1.2989 | Training Accuracy: 51.9900%
Validation Loss: 1.2455 | Validation Accuracy: 57.7726%
100%|██████████| 162/162 [03:16<00:00,  1.21s/it]
Epoch 4:
Training Loss: 1.2709 | Training Accuracy: 57.0711%
Validation Loss: 1.2885 | Validation Accuracy: 53.0162%
100%|██████████| 162/162 [03:14<00:00,  1.20s/it]
Epoch 5:
Training Loss: 1.1965 | Training Accuracy: 58.4911%
Validation Loss: 1.3769 | Validation Accuracy: 60.4408%
100%|██████████| 162/162 [03:20<00:00,  1.24s/it]
Epoch 6:
Training Loss: 1.2313 | Training Accuracy: 57.7570%
Validation Loss: 1.3022 | Validation Accuracy: 52.9002%
100%|██████████| 162/162 [03:12<00:00,  1.19s/it]
Epoch 7:
Training Loss: 1.2369 | Training Accuracy: 54.7430%
Validation Loss: 1.2882 | Validation Accuracy: 55.1044%
100%|██████████| 162/162 [03:11<00:00,  1.18s/it]
```

```
Epoch 8:
Training Loss: 1.1423 | Training Accuracy: 59.5730%
Validation Loss: 1.1976 | Validation Accuracy: 55.8005%
100%|██████████| 162/162 [03:15<00:00,  1.21s/it]
Epoch 9:
Training Loss: 1.0929 | Training Accuracy: 60.5294%
Validation Loss: 1.1318 | Validation Accuracy: 58.8167%
100%|██████████| 162/162 [03:21<00:00,  1.24s/it]
Epoch 10:
Training Loss: 1.0918 | Training Accuracy: 60.9061%
Validation Loss: 1.2057 | Validation Accuracy: 58.8167%
100%|██████████| 162/162 [03:21<00:00,  1.25s/it]
Epoch 11:
Training Loss: 0.9898 | Training Accuracy: 64.0553%
Validation Loss: 1.0015 | Validation Accuracy: 61.9490%
100%|██████████| 162/162 [03:22<00:00,  1.25s/it]
Epoch 12:
Training Loss: 0.9480 | Training Accuracy: 64.4320%
Validation Loss: 0.9609 | Validation Accuracy: 63.9211%
100%|██████████| 162/162 [03:22<00:00,  1.25s/it]
Epoch 13:
Training Loss: 0.9268 | Training Accuracy: 64.7798%
Validation Loss: 1.0260 | Validation Accuracy: 59.3968%
100%|██████████| 162/162 [03:21<00:00,  1.25s/it]
Epoch 14:
Training Loss: 0.8866 | Training Accuracy: 65.0985%
Validation Loss: 0.9430 | Validation Accuracy: 64.2691%
100%|██████████| 162/162 [03:17<00:00,  1.22s/it]
Epoch 15:
Training Loss: 0.8256 | Training Accuracy: 66.8953%
Validation Loss: 0.9305 | Validation Accuracy: 64.1531%
100%|██████████| 162/162 [03:12<00:00,  1.19s/it]
Epoch 16:
Training Loss: 0.8006 | Training Accuracy: 67.3493%
Validation Loss: 0.9662 | Validation Accuracy: 63.6891%
100%|██████████| 162/162 [03:19<00:00,  1.23s/it]
Epoch 17:
Training Loss: 0.7776 | Training Accuracy: 68.6631%
Validation Loss: 0.8253 | Validation Accuracy: 70.3016%
100%|██████████| 162/162 [03:18<00:00,  1.22s/it]
```

```
Epoch 18:
Training Loss: 0.7562 | Training Accuracy: 70.0734%
Validation Loss: 0.8214 | Validation Accuracy: 67.9814%
100%|██████████| 162/162 [03:10<00:00,  1.18s/it]
Epoch 19:
Training Loss: 0.7194 | Training Accuracy: 71.8219%
Validation Loss: 0.8672 | Validation Accuracy: 69.1415%
100%|██████████| 162/162 [03:14<00:00,  1.20s/it]
Epoch 20:
Training Loss: 0.6990 | Training Accuracy: 73.5220%
Validation Loss: 0.8128 | Validation Accuracy: 69.8376%
100%|██████████| 162/162 [03:16<00:00,  1.21s/it]
Epoch 21:
Training Loss: 0.6832 | Training Accuracy: 74.6039%
Validation Loss: 0.7685 | Validation Accuracy: 72.8538%
100%|██████████| 162/162 [03:10<00:00,  1.18s/it]
Epoch 22:
Training Loss: 0.6428 | Training Accuracy: 75.6182%
Validation Loss: 0.8007 | Validation Accuracy: 72.2738%
100%|██████████| 162/162 [03:16<00:00,  1.21s/it]
Epoch 23:
Training Loss: 0.6463 | Training Accuracy: 75.4154%
Validation Loss: 0.7690 | Validation Accuracy: 69.1415%
100%|██████████| 162/162 [03:19<00:00,  1.23s/it]
Epoch 24:
Training Loss: 0.6106 | Training Accuracy: 77.2315%
Validation Loss: 0.6823 | Validation Accuracy: 75.1740%
100%|██████████| 162/162 [03:17<00:00,  1.22s/it]
Epoch 25:
Training Loss: 0.6126 | Training Accuracy: 77.1542%
Validation Loss: 0.7117 | Validation Accuracy: 74.2459%
100%|██████████| 162/162 [03:08<00:00,  1.17s/it]
Epoch 26:
Training Loss: 0.5858 | Training Accuracy: 78.2264%
Validation Loss: 0.6631 | Validation Accuracy: 76.1021%
100%|██████████| 162/162 [03:13<00:00,  1.20s/it]
Epoch 27:
Training Loss: 0.5755 | Training Accuracy: 78.9026%
Validation Loss: 0.6623 | Validation Accuracy: 75.7541%
100%|██████████| 162/162 [03:18<00:00,  1.22s/it]
```

```
Epoch 28:
Training Loss: 0.5752 | Training Accuracy: 79.3470%
Validation Loss: 0.7221 | Validation Accuracy: 74.0139%
100%|██████████| 162/162 [03:11<00:00,  1.18s/it]
Epoch 29:
Training Loss: 0.5546 | Training Accuracy: 79.9652%
Validation Loss: 0.7244 | Validation Accuracy: 73.6659%
100%|██████████| 162/162 [03:10<00:00,  1.17s/it]
Epoch 30:
Training Loss: 0.5180 | Training Accuracy: 81.1148%
Validation Loss: 0.7289 | Validation Accuracy: 75.8701%
```

In [28]:
```
# torch.save(model.state_dict(), 'VGG19_30_epochs.pth')
```

In [91]:
```
training_stats = pd.DataFrame({"Training_Accuracy": train_accs,
                               "Training_Loss": train_losses,
                               "Val_Loss": val_losses,
                               "Val_Accuracy": val_accs})

training_stats.to_csv("VGG19_Training Statistics.csv", index = True)
```

Plotting the loss and accuracy for training and validation set during model training

In [38]:
```
training_stats = pd.read_csv('VGG19_Training Statistics.csv')
training_stats = training_stats[['Training_Accuracy',   'Training_Loss', 'Val_Loss', 'Val_Accuracy']]
```

In [40]:
```
training_stats.head()
```

Out[40]:

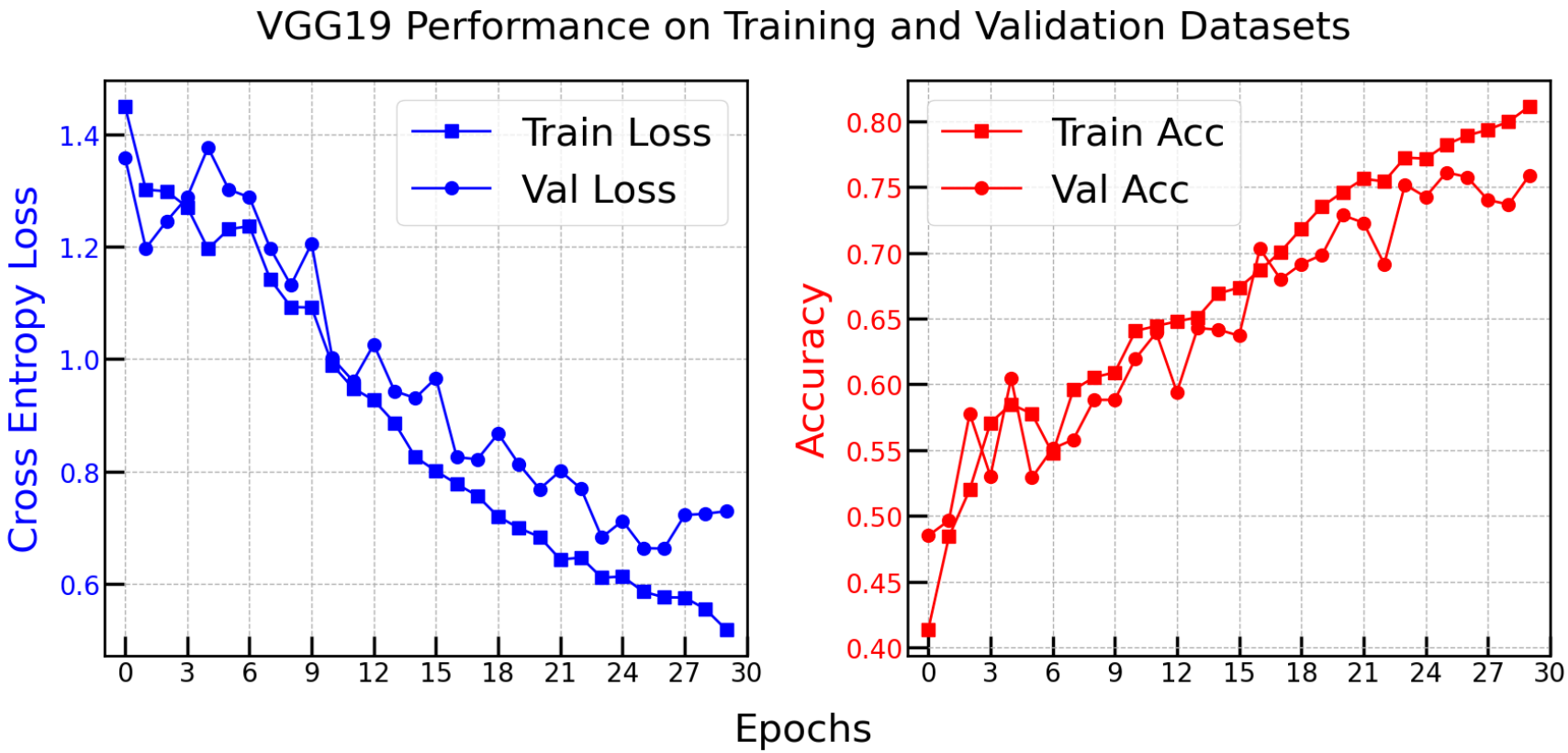
	Training_Accuracy	Training_Loss	Val_Loss	Val_Accuracy
0	0.413543	1.450039	1.359274	0.484919
1	0.484254	1.301991	1.197390	0.496520
2	0.519900	1.298880	1.245546	0.577726
3	0.570711	1.270896	1.288489	0.530162
4	0.584911	1.196527	1.376896	0.604408

In [92]:
```
fig, axes = plt.subplots(1, 2, figsize=(20, 8))

ax1 = plt.subplot(1,2, 1)
```

```
''' Left plot contains the validation and training lossess '''
plot_1 = ax1.plot(range(0, 30), training_stats['Training_Loss'], color = 'blue', label = 'Train Loss',\
            marker = 's', linewidth=2.0, markersize = 10)

plot_2 = ax1.plot(range(0, 30), training_stats['Val_Loss'], color = 'blue', label = 'Val Loss',\
            marker = 'o', linewidth=2.0, markersize = 10)

ax1.tick_params(axis ='y', labelcolor = 'blue',labelsize=20, width=3)
ax1.tick_params(axis ='x', labelcolor = 'black',labelsize=20, width=3)
ax1.legend(fontsize = 30)
plt.xticks(range(0,33,3))
ax1.set_xlim([-1, 30])
ax1.set_ylabel("Cross Entropy Loss", fontsize = 30, labelpad = 10, color = 'blue')


''' Right plot contains the training and validation accuracies '''
ax1a = plt.subplot(1,2, 2)
plot_11 = ax1a.plot(range(0, 30), training_stats['Training_Accuracy'], color = 'red', label = 'Train Acc',\
            marker = 's', linewidth=2.0, markersize = 10)

plot_22 = ax1a.plot(range(0, 30), training_stats['Val_Accuracy'], color = 'red', label = 'Val Acc',\
            marker = 'o', linewidth=2.0, markersize = 10)
ax1a.legend(fontsize = 30)
plt.xticks(range(0,33,3))
ax1a.set_xlim([-1, 30])
ax1a.tick_params(axis ='y', labelcolor = 'red',labelsize=20, width=3)
ax1a.tick_params(axis ='x', labelcolor = 'black',labelsize=20, width=3)
ax1a.set_ylabel("Accuracy", fontsize = 30, labelpad = 10, color = 'red')


# for ax1
ax1.tick_params(which='both', width=2.5)
ax1.tick_params(which='major', length=15)
ax1.tick_params(which='minor', length=5)
ax1.tick_params(which = 'both', direction = 'in')

# for ax1a
ax1a.tick_params(which='both', width=2.5)
ax1a.tick_params(which='major', length=15)
ax1a.tick_params(which='minor', length=5)
ax1a.tick_params(which = 'both', direction = 'in')

# set various colors
ax1a.spines['bottom'].set_color('black')
```

```
ax1a.spines['top'].set_color('black')
ax1a.spines['right'].set_color('black')
ax1a.spines['right'].set_linewidth(2)
ax1a.spines['top'].set_linewidth(2)
ax1a.spines['bottom'].set_linewidth(2)
ax1a.spines['left'].set_color('black')
ax1a.spines['left'].set_lw(2)

# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)


ax1.grid(True, which = 'major', alpha = 1, linestyle='--', linewidth = 1)
ax1a.grid(True, which = 'major', alpha = 1, linestyle='--', linewidth = 1)


plt.subplots_adjust(wspace=0.25,hspace=0.)
fig.text(0.5, 0.01, 'Epochs', ha='center', va='center', fontsize = 30)

fig.text(0.5, 0.95, 'VGG19 Performance on Training and Validation Datasets', ha='center', va='center', fontsize = 30)
```

Out[92]:
```
Text(0.5, 0.95, 'VGG19 Performance on Training and Validation Datasets')
```



The above subplot visualizes the training and validation accuracies and loss for a total of 30 epochs. The x-axis in both plots contains the corresponding epoch and y axis contains the cross entropy loss and accuracy of the model recorded at each epoch. It can be cleary seen that the training loss is fairly smooth and is decreasing whereas the validation loss is oscillating. Similarly, the validation accuracy starts to become slightly stagnant after 25 epochs. The trend also indicates that at the end of 27th epoch, the validation loss is diverging whereas training loss is still reducing. It could also mean that, should the model be trained any longer, the model might overfit the training data.

Evaluating Model Performance on Test Data

In [89]:
```
model.load_state_dict(torch.load('VGG19_BN_30_epochs.pth'))
model.eval()
```

Out[89]:
```
VGG19(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (25): ReLU(inplace=True)
    (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (27): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (34): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (35): ReLU(inplace=True)
    (36): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (37): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (38): ReLU(inplace=True)
    (39): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (42): ReLU(inplace=True)
```

```
    (43): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (44): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (45): ReLU(inplace=True)
    (46): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (47): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (48): ReLU(inplace=True)
    (49): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (50): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (51): ReLU(inplace=True)
    (52): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): Dropout(p=0.5, inplace=False)
    (8): Linear(in_features=4096, out_features=5, bias=True)
  )
)
```

In [90]:
```
print(model.parameters().__next__().device)
```

```
cuda:0
```

In [365…
```
'''
This custom function takes in a trained model and test dataloader.
It makes prediction and returns predicted class, actual class, the predicted probability for predicted class and
- predicted probabilities for each class. This way, we can compare performance of our model accross wide range of metri
'''

def make_predictions_on_dataloaders(trained_model, dataloader):
    target = []
    probabilities = []
    predictions = []
    pred_probs_for_all_class = []

    with torch.no_grad():
        trained_model.eval()
        for images, labels in tqdm(test_dl):
            images = images.to(device), labels = labels.to(device)
```

```
            yb = trained_model(images)
            out = nn.Softmax(dim = 1)(yb)
            out = out.cpu()
            probs, preds  = torch.max(out, dim=1)
            preds = np.array(preds.cpu().detach())
            probs = probs.cpu().detach().numpy()
            labels = labels.cpu().detach().numpy()
            out = out.detach().numpy()


            target.append(labels)
            probabilities.append(probs)
            predictions.append(preds)
            pred_probs_for_all_class.append(out)

    return target, probabilities, predictions, pred_probs_for_all_class
```

In [93]:
```
# making prediction
target, probabilities, predictions, pred_probs_for_all_class = make_predictions_on_dataloaders(model, test_dl)
```

```
100%|██████████| 14/14 [00:07<00:00,  1.77it/s]
```

In [94]:
```
def flatten(input_arr):
    output = []
    for i in input_arr:
        for j in i:
            output.append(j)
    return output
```

In [95]:
```
predictions = flatten([list(i) for i in predictions])
target = flatten([list(i) for i in target])
probabilities = flatten([list(i) for i in probabilities])
pred_probs_for_all_class = np.array(flatten(pred_probs_for_all_class))
```

Lets check accuracy, precision, f1 score, recall and auc score on the test dataset

In [97]:
```
print("The testing accuracy is: {}".format(accuracy_score(target, predictions)*100))
```

```
The testing accuracy is: 77.03016241299304
```

In [98]:
```
print("Precision (Test): ", precision_score(target, predictions, average = 'weighted'))
print("Recall (Test): ", recall_score(target, predictions, average = 'weighted'))
print("F1 (Test): ", f1_score(target, predictions, average = 'weighted'))
```

```
Precision (Test):  0.7792436524605408
Recall (Test):  0.7703016241299304
F1 (Test):  0.7726826775646839
```

In [99]:
```
print("Classification Report")
print(classification_report(target, predictions))
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.90      0.76      0.82       157
           1       0.81      0.83      0.82       202
           2       0.62      0.72      0.67       155
           3       0.85      0.76      0.80       135
           4       0.73      0.77      0.75       213

    accuracy                           0.77       862
   macro avg       0.78      0.77      0.77       862
weighted avg       0.78      0.77      0.77       862
```

The precision of our model is approximately 77.92 % . It indicates that out of all positive predictions made by the model, about 77.92 % of them are actually correct. Furthermore, the recall of our model is about 77% which indicates taht out of all positives in our target, the model was able to correctly classify about 77% of them corretly.

The choice of precision over recall depends upon the problem statement. Lets suppose that I am building a flower classification model to identify toxic or dangerous plants, it may be more important to have high precision. This is to ensure that I can minimize false positives and avoid misclassifying a safe plant as dangerous. However, if I am building a model to identify rare or endangered plant species, it might be in my interest to have high recall, so that I am able to identify as many positive cases as possible, even if it results in some false positives.

Similarly, the f1 score of our model is 77.2 %, which is the harmonic mean of precision and recall. This f1 score indicates that our model is correctly classifying positive cases very good by reducing the number of misclassificaitons.

A f1 score of 77.2 % means that when the model makes a positive prediction for a given class, it is correct about 77% of the time, and when a true positive case occurs in the dataset, the model is able to correctly identify it about 77% of the time.

It can be seen that model is performing pretty well accross all classes with a fairly high test accuracy of 77 % approximately. Accuracy alone is a weak metric to truly understand the performance of our model. Lets look at confusion matrix to understand the fa;se positives and negatives predictions made by our model.

Lets create a confusion matrix to further understand the performance of our model

In [100…
```
cf_matrix = confusion_matrix(target, predictions)
dataframe = pd.DataFrame(cf_matrix, index = classes, columns = classes)
```

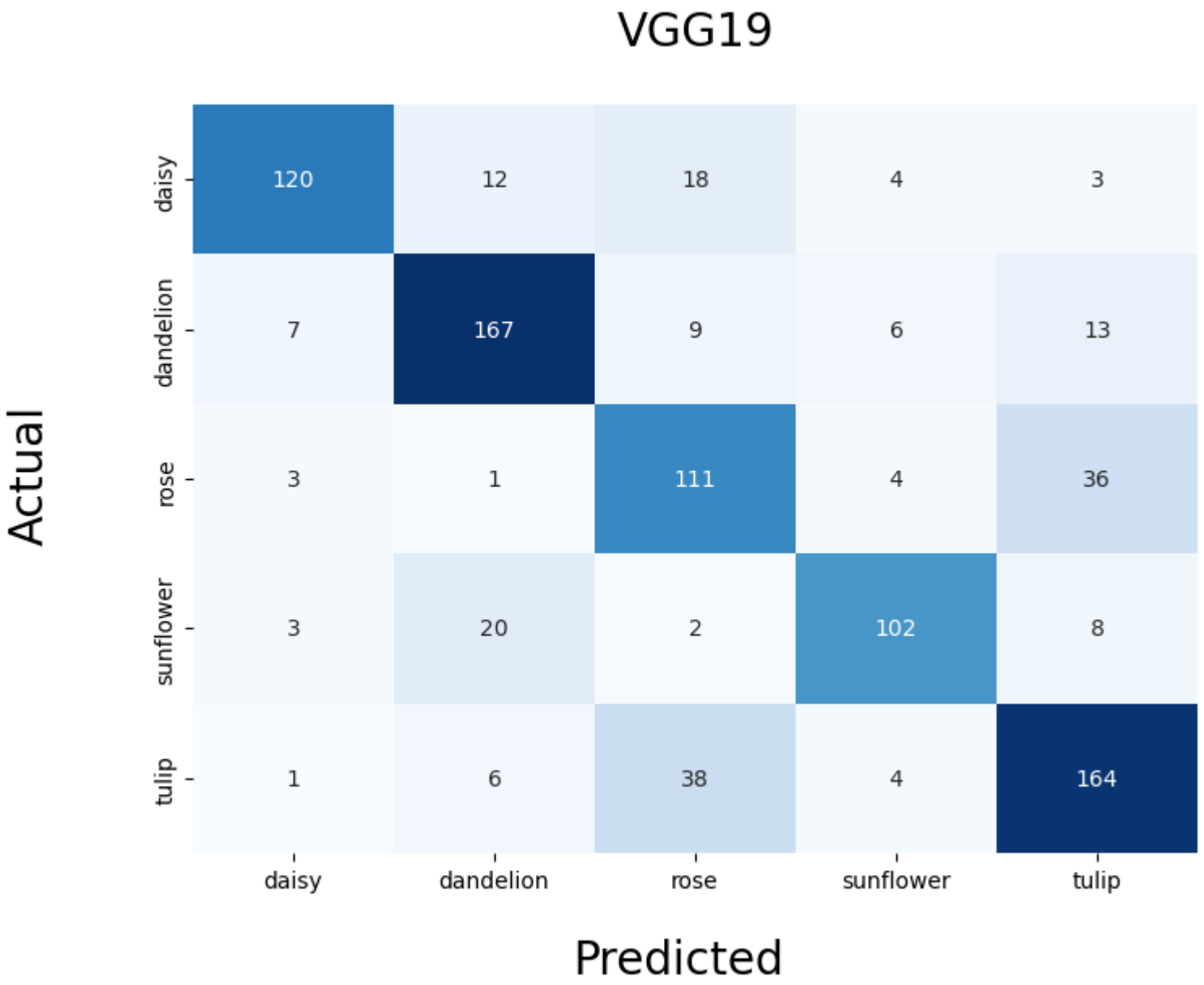
In [101…
```
dataframe
```

Out[101]:

	daisy	dandelion	rose	sunflower	tulip
daisy	120	12	18	4	3
dandelion	7	167	9	6	13
rose	3	1	111	4	36
sunflower	3	20	2	102	8
tulip	1	6	38	4	164

A confusion matrix showcases the true labels and predicted labels for all classes in a table, as shown below.

In [102…
```
fig, axes = plt.subplots(1, 1, figsize=(8, 6))

ax1 = plt.subplot(1, 1, 1)

sns.heatmap(dataframe, cmap="Blues", annot = True, fmt="d", cbar =False)
fig.text(0.5, 0.00, 'Predicted', ha='center', va='center', fontsize = 20)
fig.text(0.0, 0.5, 'Actual', ha='center', va='center', rotation='vertical', fontsize = 20)
ax1.text(0.5, 1.08, 'VGG19',
    horizontalalignment='center',
    fontsize=20,
    transform = ax1.transAxes);
```



In the confusion matrix, we can cleary see that out of 157 daisy samples, only 120 were correctly classified. Similarly, out of all samples belonging to dandelion class, 167 were correctly clasified. Out of all samples belonging to rose class, 111 were correctly clasified. out of all samples belonging to sunflower class, 102 were correctly clasified. Finally, out of all samples belonging to tulip class, 164 were correctly clasified.

In [103…
```
# Calculating False positive rates and True Positive Rates
fprs = {}
tprs = {}
thresholds ={}

for i in range(5):
    fprs[i], tprs[i], thresholds[i] = roc_curve(target, pred_probs_for_all_class[:,i], pos_label=i)
```

In [104…
```
auc_score_ovr = roc_auc_score(target, pred_probs_for_all_class, multi_class='ovr', average = 'weighted')
print("AUC SCORE (Test Set): {}".format(auc_score_ovr))
```

```
AUC SCORE (Test Set): 0.9443184074462758
```

An auc score of 0.5 means that our model is making random predictions. Similarly, an auc score of 0 means that the model is predicting positive classes and negative and vice versa. An auc score of about 94.4 % means that the model has good ability to diffrentiate between positive and negative classes.

An AUC score of 0.9 indicates that the model has a high probability of correctly ranking a randomly chosen positive instance higher than a randomly chosen negative instance. Specifically, if a positive instance is randomly selected from the dataset and compared to a negative instance that was also randomly selected, then there is a 90% chance that the model will assign a higher predicted probability to the positive instance than to the negative instance.

Therefore, an AUC score of 0.9 suggests that the model is highly capable of distinguishing between the positive and negative classes, and it has a high true positive rate while maintaining a low false positive rate, which is desirable for many classification tasks.

In [105…
```
one_vs_rest_auc = roc_auc_score(target, pred_probs_for_all_class, multi_class='ovr', average = None)
```

In [106…
```
one_vs_rest_auc
```

Out[106]:
```
array([0.94725573, 0.95881338, 0.91382945, 0.96508228, 0.93743354])
```

In [107…
```
classes
```

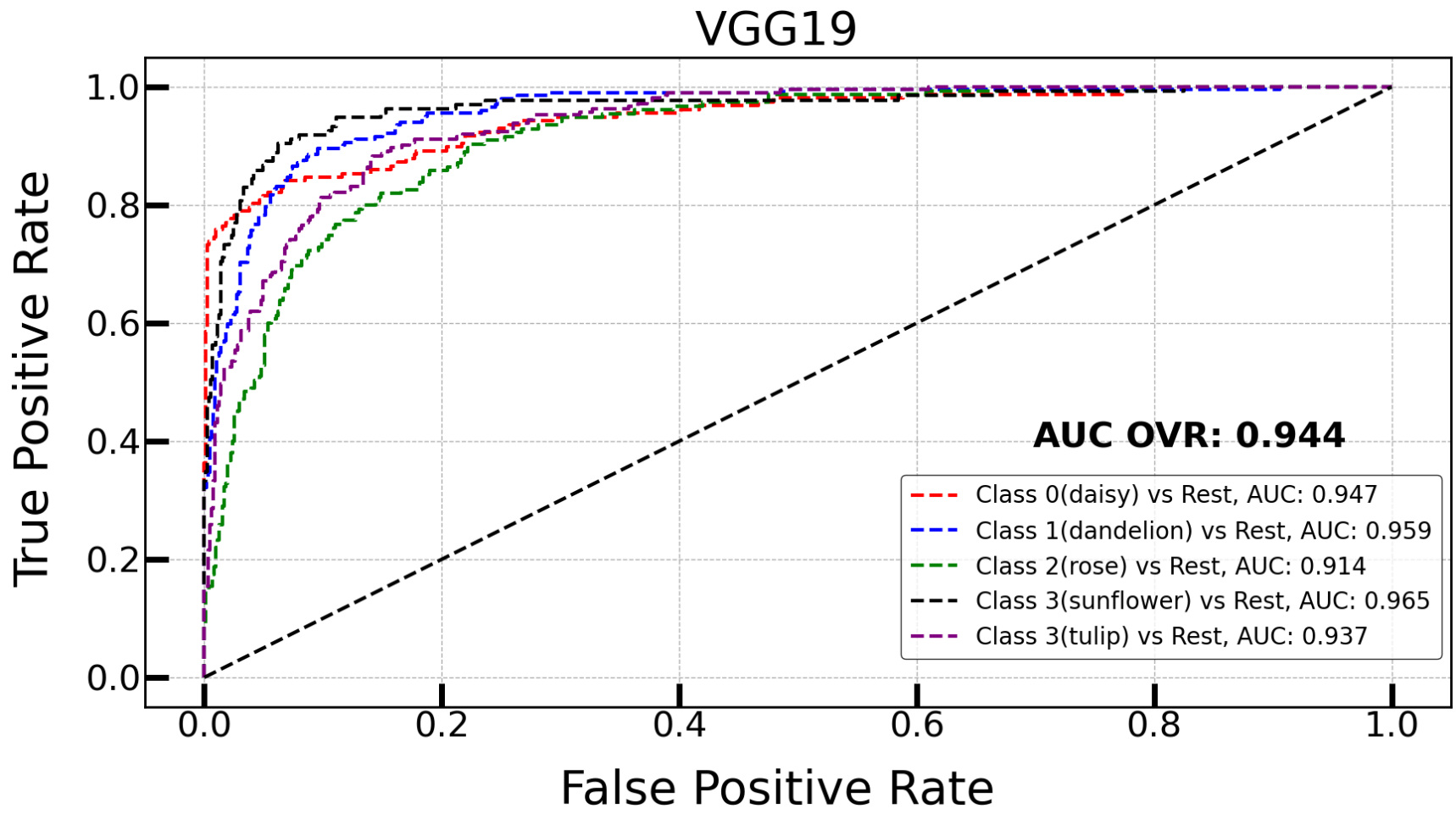
Out[107]:
```
['daisy', 'dandelion', 'rose', 'sunflower', 'tulip']
```

In [108…
```
fig, axes = plt.subplots(1, 1, figsize=(20, 10))

ax1 = plt.subplot(1, 1, 1)
```

```
plt.plot(fprs[0], tprs[0], linestyle='--',color='red', label='Class 0(daisy) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[0], 3)), linewidth = 3)
plt.plot(fprs[1], tprs[1], linestyle='--',color='blue', label='Class 1(dandelion) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[1], 3)), linewidth = 3)
plt.plot(fprs[2], tprs[2], linestyle='--',color='green', label='Class 2(rose) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[2], 3)), linewidth = 3)
plt.plot(fprs[3], tprs[3], linestyle='--',color='black', label='Class 3(sunflower) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[3], 3)), linewidth = 3)
plt.plot(fprs[4], tprs[4], linestyle='--',color='purple', label='Class 3(tulip) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[4], 3)), linewidth = 3)


plt.plot([0, 1], [0, 1], color="black", linewidth = 3, linestyle = '--')


ax1.tick_params(axis ='y', labelcolor = 'black',labelsize=30, width=3)
ax1.tick_params(axis ='x', labelcolor = 'black',labelsize=30, width=3)



# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)



# for ax1
ax1.tick_params(which='both', width=5)
ax1.tick_params(which='major', length=20)
ax1.tick_params(which='minor', length=5)
ax1.tick_params(which = 'both', direction = 'in')


ax1.grid(True, which = 'major', alpha = 1, linestyle='--', linewidth = 1)
lines_labels = [[ax.get_legend_handles_labels() for ax in fig.axes][0]]
lines, labels = [sum(lol, []) for lol in zip(*lines_labels)]
leg = ax1.legend(lines, labels, loc="upper left", bbox_to_anchor=(0.57, 0.375), fontsize = 20, ncol = 1,markerscale = 1
leg.get_frame().set_edgecolor('black')
```

```
ax1.text(0.8, 0.4, 'AUC OVR: '+ str(round(auc_score_ovr, 3)),
    horizontalalignment='center',
    fontsize=30,fontweight = 'bold',
    transform = ax1.transAxes);


plt.subplots_adjust(wspace=0.2,hspace=0.3)
fig.text(0.5, 0.01, 'False Positive Rate', ha='center', va='center', fontsize = 40)
fig.text(0.06, 0.5, 'True Positive Rate', ha='center', va='center', rotation='vertical', fontsize = 40, color = 'black'
fig.text(0.5, 0.91, 'VGG19', ha='center', va='center', fontsize = 40)
```

Out[108]: Text(0.5, 0.91, 'VGG19')



AUC is a widely used evaluation metric for binary classification problems. It measures the area under the receiver operating characteristic (ROC) curve, which is a plot of the true positive rate (sensitivity) against the false positive rate (1-specificity) at different classification thresholds. The auc score for individual classes, follwoing one vs rest approach, is also given in the plot.

Transfer Learning

We will make use of VGG19 model trained on imagenet dataset that consists of 1000 classses. We will freeze weights of all layers except for final linear layer. The final linear layer on vgg19 classifies 1000 classes. But, our dataset only has 5 class. Therefore, we will replace that layer with a new linear layer such that the final layer will have 5 outputs neurons instead of 1000. Finally, we will train our new VGG19 model by freezing all layers except for the final classification layer and check the performance

In [331…
```
# Load pre-trained VGG19 model
vgg19_pretrained = models.vgg19(pretrained=True)
```

In [332…
```
vgg19_pretrained
```

Out[332]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
```

```
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Freeze Early layers

Inserting a Custom Classififer

The original VGG19 trained on imagenet data has 1000 classes. We will remove or replace it with another linear layer that will have 5 output neurons to predict 5 classes. Lets do that.

In [334…
```
vgg19_pretrained.classifier
```

Out[334]:
```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
)
```

we can clearly see that in this sequential object's last linear layer has 1000 output neurons. Lets extract the last or sixth layer and change its output neurons.

`Instead of just replacing the final linear layer such that it will have 5 output neurons, I will insert a sequential object that will contain multiple linear layers: A Custom multi layer percepton`

In [335…
```
# original final linear layer
vgg19_pretrained.classifier[6]
```

Out[335]:
```
Linear(in_features=4096, out_features=1000, bias=True)
```

In [336…
```
# Add a custom classifier
vgg19_pretrained.classifier[6] = nn.Sequential(
```

```
                      nn.Linear(4096, 2048),
                      nn.BatchNorm1d(2048),
                      nn.LeakyReLU(negative_slope=0.01),
                      nn.Dropout(0.25),
                      nn.Linear(2048, 1024),
                      nn.BatchNorm1d(1024),
                      nn.LeakyReLU(negative_slope=0.01),
                      nn.Dropout(0.25),
                      nn.Linear(1024, 5))

# vgg19_pretrained.classifier[6] = nn.Linear(4096, 6, bias=True)
```

When using batch normalization between linear layers, it is recommended to add the batch normalization layer before the activation function. This is because the output of the activation function can be affected by the mean and variance normalization performed by batch normalization, which can lead to reduced performance or convergence issues.

In [337…

```
# new final linear layer
vgg19_pretrained.classifier[6]
```

Out[337]:

```
Sequential(
  (0): Linear(in_features=4096, out_features=2048, bias=True)
  (1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
  (3): Dropout(p=0.25, inplace=False)
  (4): Linear(in_features=2048, out_features=1024, bias=True)
  (5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (6): LeakyReLU(negative_slope=0.01)
  (7): Dropout(p=0.25, inplace=False)
  (8): Linear(in_features=1024, out_features=5, bias=True)
)
```

In [338…

```
vgg19_pretrained.classifier
```

Out[338]:
```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Sequential(
    (0): Linear(in_features=4096, out_features=2048, bias=True)
    (1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01)
    (3): Dropout(p=0.25, inplace=False)
    (4): Linear(in_features=2048, out_features=1024, bias=True)
    (5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.01)
    (7): Dropout(p=0.25, inplace=False)
    (8): Linear(in_features=1024, out_features=5, bias=True)
  )
)
```

Here, the final linear classifying layer of pretrained VGG19 has been replaced by a sequential object containing a Multilayer perceptron. It will classify the input to 5 outputs.

Lets check the entire architecture of our new model

In [339…
```
vgg19_pretrained
```

Out[339]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
```

```
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Sequential(
        (0): Linear(in_features=4096, out_features=2048, bias=True)
        (1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.01)
        (3): Dropout(p=0.25, inplace=False)
        (4): Linear(in_features=2048, out_features=1024, bias=True)
        (5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (6): LeakyReLU(negative_slope=0.01)
        (7): Dropout(p=0.25, inplace=False)
        (8): Linear(in_features=1024, out_features=5, bias=True)
      )
    )
  )
```

In [340…

```
# passing the new model to GPU
vgg19_pretrained = vgg19_pretrained.to('cuda')
```

In [341…

```
summary(vgg19_pretrained, (3, 224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         295,168
             ReLU-12          [-1, 256, 56, 56]               0
           Conv2d-13          [-1, 256, 56, 56]         590,080
             ReLU-14          [-1, 256, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         590,080
             ReLU-16          [-1, 256, 56, 56]               0
           Conv2d-17          [-1, 256, 56, 56]         590,080
             ReLU-18          [-1, 256, 56, 56]               0
        MaxPool2d-19          [-1, 256, 28, 28]               0
           Conv2d-20          [-1, 512, 28, 28]       1,180,160
             ReLU-21          [-1, 512, 28, 28]               0
           Conv2d-22          [-1, 512, 28, 28]       2,359,808
             ReLU-23          [-1, 512, 28, 28]               0
           Conv2d-24          [-1, 512, 28, 28]       2,359,808
             ReLU-25          [-1, 512, 28, 28]               0
           Conv2d-26          [-1, 512, 28, 28]       2,359,808
             ReLU-27          [-1, 512, 28, 28]               0
        MaxPool2d-28          [-1, 512, 14, 14]               0
           Conv2d-29          [-1, 512, 14, 14]       2,359,808
             ReLU-30          [-1, 512, 14, 14]               0
           Conv2d-31          [-1, 512, 14, 14]       2,359,808
             ReLU-32          [-1, 512, 14, 14]               0
           Conv2d-33          [-1, 512, 14, 14]       2,359,808
             ReLU-34          [-1, 512, 14, 14]               0
           Conv2d-35          [-1, 512, 14, 14]       2,359,808
             ReLU-36          [-1, 512, 14, 14]               0
        MaxPool2d-37            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-38            [-1, 512, 7, 7]               0
           Linear-39                 [-1, 4096]     102,764,544
             ReLU-40                 [-1, 4096]               0
          Dropout-41                 [-1, 4096]               0
           Linear-42                 [-1, 4096]      16,781,312
```

```
              ReLU-43                 [-1, 4096]               0
            Dropout-44                 [-1, 4096]               0
             Linear-45                 [-1, 2048]       8,390,656
        BatchNorm1d-46                 [-1, 2048]           4,096
          LeakyReLU-47                 [-1, 2048]               0
            Dropout-48                 [-1, 2048]               0
             Linear-49                 [-1, 1024]       2,098,176
        BatchNorm1d-50                 [-1, 1024]           2,048
          LeakyReLU-51                 [-1, 1024]               0
            Dropout-52                 [-1, 1024]               0
             Linear-53                    [-1, 5]           5,125
================================================================
Total params: 150,070,341
Trainable params: 150,070,341
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 238.77
Params size (MB): 572.47
Estimated Total Size (MB): 811.82
----------------------------------------------------------------
```

Lets freeze the parameters of the pretrained vgg19 completely. we can do that by seeting the required_grad attributes of our parameters to False.

In [342…
```
# Freeze all layers
for param in vgg19_pretrained.features.parameters():
    param.requires_grad = False
```

In [343…
```
# Count the number of convolutional layers
num_conv_layers = sum(1 for layer in vgg19_pretrained.features if isinstance(layer, nn.Conv2d))
print("Number of convolutional layers in VGG19:", num_conv_layers)
```

```
Number of convolutional layers in VGG19: 16
```

In [344…
```
# Check if parameters of nth convolutional layer are frozen

for n in range(num_conv_layers):
    try:
        if vgg19_pretrained.features[n].weight.requires_grad:
            print("Parameters of thos convolutional layer are not frozen")
        else:
            print("Parameters of this convolutional layer are frozen")
```

```
    except:
        pass
```

```
Parameters of this convolutional layer are frozen
Parameters of this convolutional layer are frozen
Parameters of this convolutional layer are frozen
Parameters of this convolutional layer are frozen
Parameters of this convolutional layer are frozen
Parameters of this convolutional layer are frozen
Parameters of this convolutional layer are frozen
```

lets check if the final classifier's weights are frozen or not

In [345…
```
# Freeze all layers
for param in vgg19_pretrained.classifier[:6].parameters():
    param.requires_grad = False
```

We will also freeze weights for each linear layer except for the final classification layer that is instead replaced with a MLP Classifier.

In [346…
```
if vgg19_pretrained.classifier[0].weight.requires_grad:
    print("Parameters of thos linear layer are not frozen")
else:
    print("Parameters of this linear layer are frozen")
```

```
Parameters of this linear layer are frozen
```

In [347…
```
if vgg19_pretrained.classifier[3].weight.requires_grad:
    print("Parameters of thos linear layer are not frozen")
else:
    print("Parameters of this linear layer are frozen")
```

```
Parameters of this linear layer are frozen
```

In [348…
```
vgg19_pretrained.classifier
```

Out[348]:
```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Sequential(
    (0): Linear(in_features=4096, out_features=2048, bias=True)
    (1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01)
    (3): Dropout(p=0.25, inplace=False)
    (4): Linear(in_features=2048, out_features=1024, bias=True)
    (5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.01)
    (7): Dropout(p=0.25, inplace=False)
    (8): Linear(in_features=1024, out_features=5, bias=True)
  )
)
```

In [349…
```
if vgg19_pretrained.classifier[6][0].weight.requires_grad:
    print("LAYER: ", vgg19_pretrained.classifier[6][0])
    print("Parameters of this linear layer are not frozen")
else:
    print("Parameters of this linear layer are frozen")
```

```
LAYER:  Linear(in_features=4096, out_features=2048, bias=True)
Parameters of this linear layer are not frozen
```

In [350…
```
if vgg19_pretrained.classifier[6][4].weight.requires_grad:
    print("LAYER: ", vgg19_pretrained.classifier[6][4])
    print("Parameters of this linear layer are not frozen")
else:
    print("Parameters of this linear layer are frozen")
```

```
LAYER:  Linear(in_features=2048, out_features=1024, bias=True)
Parameters of this linear layer are not frozen
```

In [351…
```
if vgg19_pretrained.classifier[6][8].weight.requires_grad:
    print("LAYER: ", vgg19_pretrained.classifier[6][8])
    print("Parameters of this linear layer are not frozen")
else:
    print("Parameters of this linear layer are frozen")
```

```
LAYER:  Linear(in_features=1024, out_features=5, bias=True)
Parameters of this linear layer are not frozen
```

Thus, the Custom linear layers are not fronzen. Thus, after extracting the features from conv layers whose weights are frozen, we perform classification. Lets do it.

Training the pretrained model by freezing weights

In [352…
```
vgg19_pretrained.classifier[6]
```

Out[352]:
```
Sequential(
  (0): Linear(in_features=4096, out_features=2048, bias=True)
  (1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
  (3): Dropout(p=0.25, inplace=False)
  (4): Linear(in_features=2048, out_features=1024, bias=True)
  (5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (6): LeakyReLU(negative_slope=0.01)
  (7): Dropout(p=0.25, inplace=False)
  (8): Linear(in_features=1024, out_features=5, bias=True)
)
```

In [355…
```
# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(vgg19_pretrained.classifier[6].parameters(), lr=0.001)
```

In [356…
```
epochs = 10
train_losses = []
train_accs = []
val_losses = []
val_accs = []

for epoch in range(epochs):
    train_loss = 0.0
    train_total = 0
    train_correct = 0

    # training loop
    for images, labels in tqdm(train_dl):
        images = images.to(device), labels = labels.to(device)
        optimizer.zero_grad()
        outputs = vgg19_pretrained(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()

    train_loss /= len(train_dl)
    train_acc = train_correct / train_total
    train_losses.append(train_loss)
    train_accs.append(train_acc)

    val_loss = 0.0
    val_total = 0
    val_correct = 0

    # validation loop
    with torch.no_grad():
        vgg19_pretrained.eval()  # set model to evaluation mode
        for images, labels in val_dl:
            images = images.to(device), labels = labels.to(device)
            outputs = vgg19_pretrained(images)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            val_total += labels.size(0)
            val_correct += (predicted == labels).sum().item()

        val_loss /= len(val_dl)
        val_acc = val_correct / val_total
        val_losses.append(val_loss)
        val_accs.append(val_acc)

    print(f"Epoch {epoch+1}:")
    print(f"Training Loss: {train_loss:.4f} | Training Accuracy: {train_acc*100:.4f}%")
    print(f"Validation Loss: {val_loss:.4f} | Validation Accuracy: {val_acc*100:.4f}%")

    vgg19_pretrained.train()  # set model back to training mode
```

```
100%|██████████| 162/162 [00:58<00:00,  2.75it/s]
Epoch 1:
Training Loss: 0.5529 | Training Accuracy: 79.2890%
Validation Loss: 0.3834 | Validation Accuracy: 86.3109%
100%|██████████| 162/162 [00:59<00:00,  2.73it/s]
```

```
Epoch 2:
Training Loss: 0.4131 | Training Accuracy: 84.6696%
Validation Loss: 0.3820 | Validation Accuracy: 86.4269%
100%|██████████| 162/162 [00:56<00:00,  2.87it/s]
Epoch 3:
Training Loss: 0.3713 | Training Accuracy: 85.8192%
Validation Loss: 0.3570 | Validation Accuracy: 87.8190%
100%|██████████| 162/162 [00:56<00:00,  2.88it/s]
Epoch 4:
Training Loss: 0.3295 | Training Accuracy: 87.8671%
Validation Loss: 0.3911 | Validation Accuracy: 87.0070%
100%|██████████| 162/162 [00:56<00:00,  2.87it/s]
Epoch 5:
Training Loss: 0.3094 | Training Accuracy: 88.2631%
Validation Loss: 0.4200 | Validation Accuracy: 87.8190%
100%|██████████| 162/162 [00:56<00:00,  2.88it/s]
Epoch 6:
Training Loss: 0.2803 | Training Accuracy: 89.4223%
Validation Loss: 0.4036 | Validation Accuracy: 86.5429%
100%|██████████| 162/162 [00:56<00:00,  2.87it/s]
Epoch 7:
Training Loss: 0.2605 | Training Accuracy: 90.1565%
Validation Loss: 0.3803 | Validation Accuracy: 88.8631%
100%|██████████| 162/162 [00:56<00:00,  2.87it/s]
Epoch 8:
Training Loss: 0.2459 | Training Accuracy: 90.6491%
Validation Loss: 0.4121 | Validation Accuracy: 87.7030%
100%|██████████| 162/162 [00:56<00:00,  2.86it/s]
Epoch 9:
Training Loss: 0.2291 | Training Accuracy: 91.1804%
Validation Loss: 0.3879 | Validation Accuracy: 88.9791%
100%|██████████| 162/162 [00:58<00:00,  2.75it/s]
Epoch 10:
Training Loss: 0.2257 | Training Accuracy: 91.5379%
Validation Loss: 0.4259 | Validation Accuracy: 88.0510%
```

In [357…

```
training_stats = pd.DataFrame({"Training_Accuracy": train_accs,
                               "Training_Loss": train_losses,
                               "Val_Loss": val_losses,
                               "Val_Accuracy": val_accs})

training_stats.to_csv("VGG19_Training_Statistics_Pretrained.csv", index = True)
```

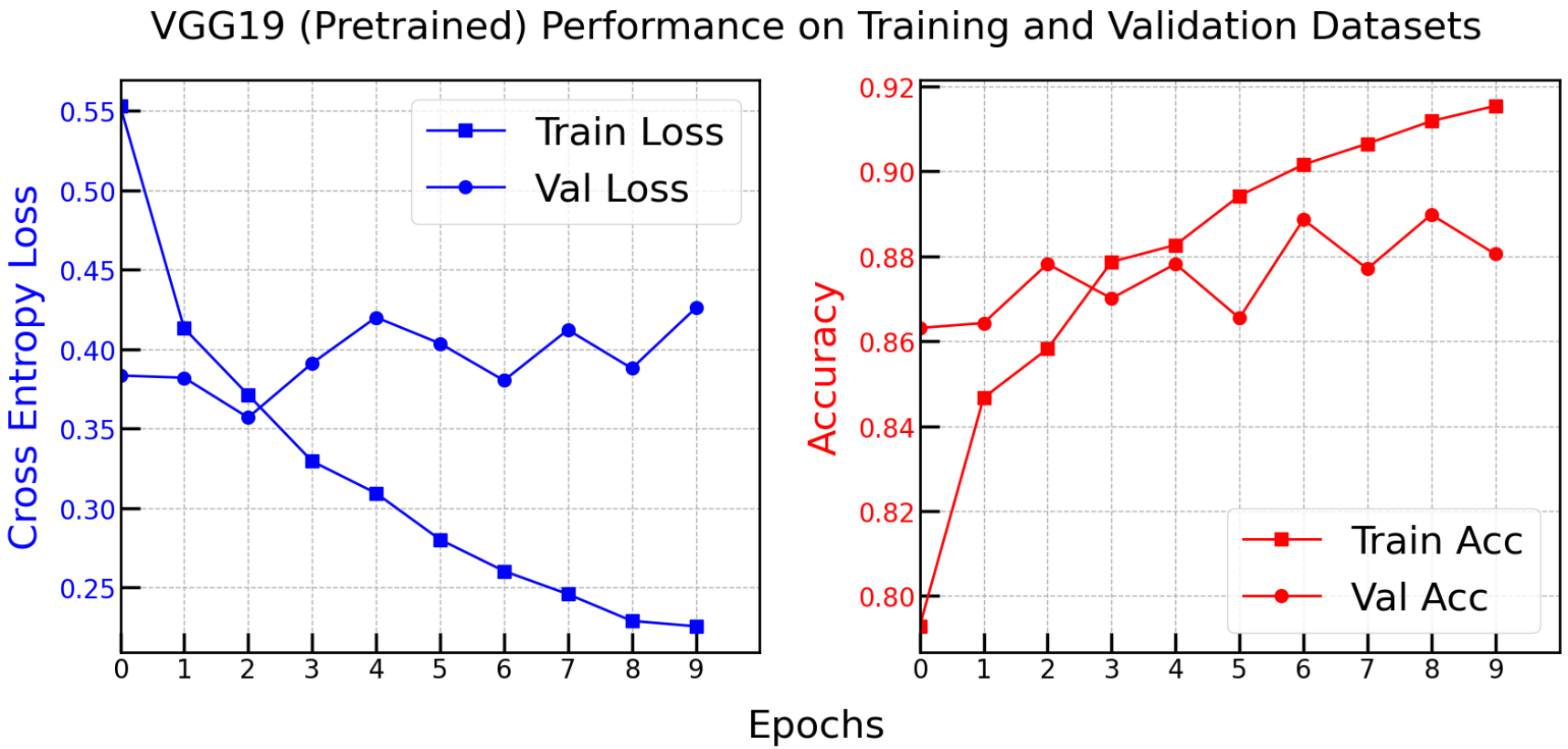
In [358…
```
torch.save(vgg19_pretrained.state_dict(), 'VGG19_Pretrained_30_epochs.pth')
```

Plotting the loss and accuracy for training and validation set during model training

In [359…
```
fig, axes = plt.subplots(1, 2, figsize=(20, 8))

ax1 = plt.subplot(1,2, 1)

''' Left plot contains the validation and training lossess '''
plot_1 = ax1.plot(range(0, 10), training_stats['Training_Loss'], color = 'blue', label = 'Train Loss',\
            marker = 's', linewidth=2.0, markersize = 10)

plot_2 = ax1.plot(range(0, 10), training_stats['Val_Loss'], color = 'blue', label = 'Val Loss',\
            marker = 'o', linewidth=2.0, markersize = 10)

ax1.tick_params(axis ='y', labelcolor = 'blue',labelsize=20, width=3)
ax1.tick_params(axis ='x', labelcolor = 'black',labelsize=20, width=3)
ax1.legend(fontsize = 30, loc = 'best')
plt.xticks(range(0,10,1))
ax1.set_xlim([0, 10])
ax1.set_ylabel("Cross Entropy Loss", fontsize = 30, labelpad = 10, color = 'blue')


''' Right plot contains the training and validation accuracies '''
ax1a = plt.subplot(1,2, 2)
plot_11 = ax1a.plot(range(0, 10), training_stats['Training_Accuracy'], color = 'red', label = 'Train Acc',\
            marker = 's', linewidth=2.0, markersize = 10)

plot_22 = ax1a.plot(range(0, 10), training_stats['Val_Accuracy'], color = 'red', label = 'Val Acc',\
            marker = 'o', linewidth=2.0, markersize = 10)
ax1a.legend(fontsize = 30, loc = 'best')
plt.xticks(range(0,10,1))
ax1a.set_xlim([0, 10])
ax1a.tick_params(axis ='y', labelcolor = 'red',labelsize=20, width=3)
ax1a.tick_params(axis ='x', labelcolor = 'black',labelsize=20, width=3)
ax1a.set_ylabel("Accuracy", fontsize = 30, labelpad = 10, color = 'red')


# for ax1
ax1.tick_params(which='both', width=2.5)
ax1.tick_params(which='major', length=15)
ax1.tick_params(which='minor', length=5)
ax1.tick_params(which = 'both', direction = 'in')
```

```
# for ax1a
ax1a.tick_params(which='both', width=2.5)
ax1a.tick_params(which='major', length=15)
ax1a.tick_params(which='minor', length=5)
ax1a.tick_params(which = 'both', direction = 'in')

# set various colors
ax1a.spines['bottom'].set_color('black')
ax1a.spines['top'].set_color('black')
ax1a.spines['right'].set_color('black')
ax1a.spines['right'].set_linewidth(2)
ax1a.spines['top'].set_linewidth(2)
ax1a.spines['bottom'].set_linewidth(2)
ax1a.spines['left'].set_color('black')
ax1a.spines['left'].set_lw(2)

# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)


ax1.grid(True, which = 'major', alpha = 1, linestyle='--', linewidth = 1)
ax1a.grid(True, which = 'major', alpha = 1, linestyle='--', linewidth = 1)


plt.subplots_adjust(wspace=0.25,hspace=0.)
fig.text(0.5, 0.01, 'Epochs', ha='center', va='center', fontsize = 30)

fig.text(0.5, 0.95, 'VGG19 (Pretrained) Performance on Training and Validation Datasets', ha='center', va='center', fon
```

Out[359]:
```
Text(0.5, 0.95, 'VGG19 (Pretrained) Performance on Training and Validation Datasets')
```



Unlike the scratch trained vgg19 model that could barely reach 70+ validation accuracy, the vggpretrained model with weights freezon reached more than 86% accuracy in the first epocs. We only trained this model for 10 epochs becauuse the model started to overfit the training data. We can see that the validaation loss starts to divege whereas training loss keeps on converging after 3rd epoch.

Furthermore, lets check the performance of this model on test set.

In [ ]:

Evaluating Pretrained Model's performance on Test Data

In [366…

```
target, probabilities, predictions, pred_probs_for_all_class = make_predictions_on_dataloaders(vgg19_pretrained, test_d
```

```
100%|██████████| 14/14 [00:05<00:00,  2.58it/s]
```

In [367…
```
predictions = flatten([list(i) for i in predictions])
target = flatten([list(i) for i in target])
probabilities = flatten([list(i) for i in probabilities])
pred_probs_for_all_class = np.array(flatten(pred_probs_for_all_class))
```

In [368…
```
print("The testing accuracy using PRETRAINED MODEL is: {}".format(accuracy_score(target, predictions)*100))
```

```
The testing accuracy using PRETRAINED MODEL is: 88.8631090487239
```

In [369…
```
print("Precision (Test): ", precision_score(target, predictions, average = 'weighted'))
print("Recall (Test): ", recall_score(target, predictions, average = 'weighted'))
print("F1 (Test): ", f1_score(target, predictions, average = 'weighted'))
```

```
Precision (Test):  0.8905921241016348
Recall (Test):  0.888631090487239
F1 (Test):  0.8890252526279564
```

In [370…
```
print("Classification Report")
print(classification_report(target, predictions))
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.90      0.90      0.90       157
           1       0.95      0.89      0.92       202
           2       0.84      0.86      0.85       155
           3       0.84      0.92      0.88       135
           4       0.89      0.87      0.88       213

    accuracy                           0.89       862
   macro avg       0.89      0.89      0.89       862
weighted avg       0.89      0.89      0.89       862
```

The pretrained model performed significantly better than scratch-trined vgg19. The original vgg19 achieved test accutacy of about 77% whereas the pretrained vgg19 achieved test accuracy of 88.86 % which is almost 12 % boost. Similarly, the precision recall and f1 score of our pretrained vgg is also much better than the original model.

In [371…
```
cf_matrix = confusion_matrix(target, predictions)
dataframe = pd.DataFrame(cf_matrix, index = classes, columns = classes)
```

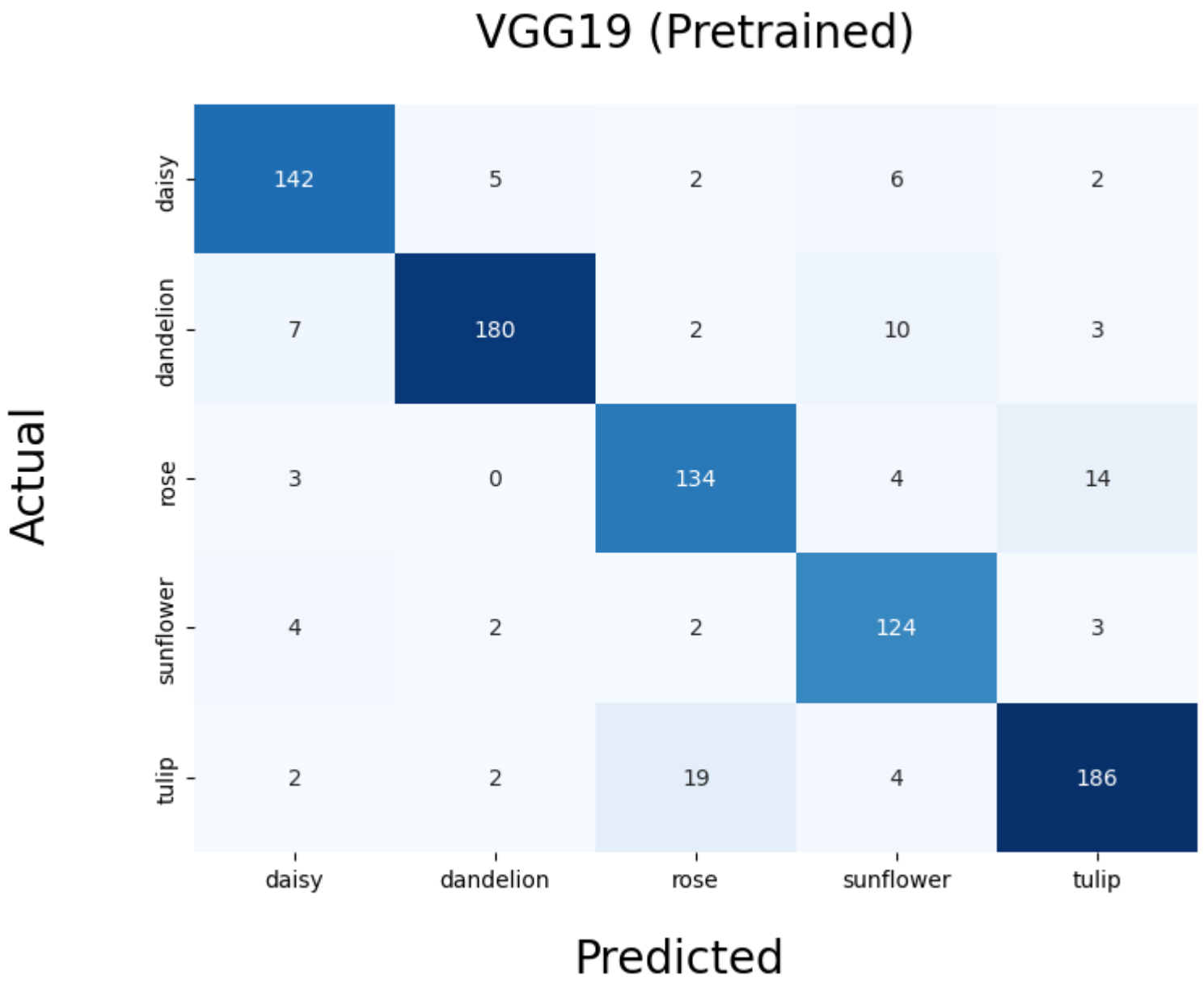
In [372…
```
dataframe
```

Out[372]:

	daisy	dandelion	rose	sunflower	tulip
daisy	142	5	2	6	2
dandelion	7	180	2	10	3
rose	3	0	134	4	14
sunflower	4	2	2	124	3
tulip	2	2	19	4	186

In [373…

```
fig, axes = plt.subplots(1, 1, figsize=(8, 6))

ax1 = plt.subplot(1, 1, 1)

sns.heatmap(dataframe, cmap="Blues", annot = True, fmt="d", cbar =False)
fig.text(0.5, 0.00, 'Predicted', ha='center', va='center', fontsize = 20)
fig.text(0.0, 0.5, 'Actual', ha='center', va='center', rotation='vertical', fontsize = 20)
ax1.text(0.5, 1.08, 'VGG19 (Pretrained)',
    horizontalalignment='center',
    fontsize=20,
    transform = ax1.transAxes);
```



The following paragraph was the analysis of confusion matrix of original VGG19 on test data.

In the confusion matrix, we can cleary see that out of 157 daisy samples, only 120 were correctly classified. Similarly, out of all samples belonging to dandelion class, 167 were correctly clasified. Out of all samples belonging to rose class, 111 were

correctly clasified. out of all samples belonging to sunflower class, 102 were correctly clasified. Finally, out of all samples belonging to tulip class, 164 were correctly clasified.

The following is the explanation of confusion matrix of pretrained vgg19 on test data.

In the confusion matrix, we can cleary see that out of 157 daisy samples, 142 were correctly classified. Similarly, out of all samples belonging to dandelion class, 180 were correctly clasified. Out of all samples belonging to rose class, 134 were correctly clasified. out of all samples belonging to sunflower class, 124 were correctly clasified. Finally, out of all samples belonging to tulip class, 186 were correctly clasified.

This proves that the pretrained vgg19 had much lower false positives and negatives as compared to the original model.

In [374…
```
# Calculating False positive rates and True Positive Rates
fprs = {}
tprs = {}
thresholds ={}

for i in range(5):
    fprs[i], tprs[i], thresholds[i] = roc_curve(target, pred_probs_for_all_class[:,i], pos_label=i)
```

In [375…
```
auc_score_ovr = roc_auc_score(target, pred_probs_for_all_class, multi_class='ovr', average = 'weighted')
print("AUC SCORE (Test Set): {}".format(auc_score_ovr))
```

```
AUC SCORE (Test Set): 0.9883210264136475
```

Similarly, the AUC score of the pretrained vgg is almot 99 % which is signigicantly higher than the scratch trained vgg19.

In [376…
```
one_vs_rest_auc = roc_auc_score(target, pred_probs_for_all_class, multi_class='ovr', average = None)
```

In [377…
```
one_vs_rest_auc
```

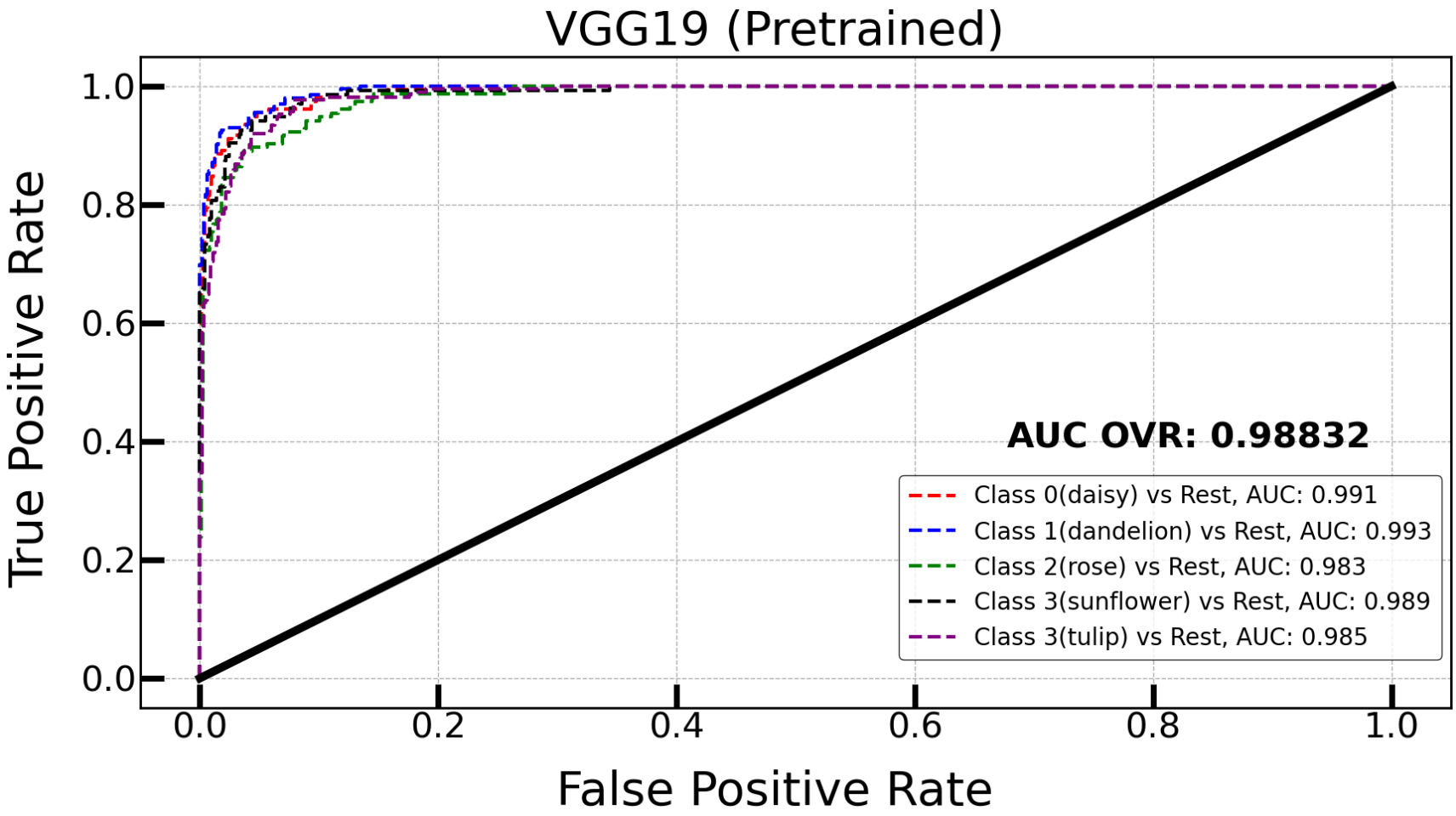
Out[377]:
```
array([0.99088404, 0.99344434, 0.98279874, 0.98870039, 0.98535124])
```

In [379…
```
fig, axes = plt.subplots(1, 1, figsize=(20, 10))

ax1 = plt.subplot(1, 1, 1)

plt.plot(fprs[0], tprs[0], linestyle='--',color='red', label='Class 0(daisy) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[0], 3)), linewidth = 3)
plt.plot(fprs[1], tprs[1], linestyle='--',color='blue', label='Class 1(dandelion) vs Rest, AUC: '+ \
```

```
        str(round(one_vs_rest_auc[1], 3)), linewidth = 3)
plt.plot(fprs[2], tprs[2], linestyle='--',color='green', label='Class 2(rose) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[2], 3)), linewidth = 3)
plt.plot(fprs[3], tprs[3], linestyle='--',color='black', label='Class 3(sunflower) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[3], 3)), linewidth = 3)
plt.plot(fprs[4], tprs[4], linestyle='--',color='purple', label='Class 3(tulip) vs Rest, AUC: '+ \
        str(round(one_vs_rest_auc[4], 3)), linewidth = 3)


plt.plot([0, 1], [0, 1], color="black", linewidth = 7)


ax1.tick_params(axis ='y', labelcolor = 'black',labelsize=30, width=3)
ax1.tick_params(axis ='x', labelcolor = 'black',labelsize=30, width=3)



# set various colors
ax1.spines['bottom'].set_color('black')
ax1.spines['top'].set_color('black')
ax1.spines['right'].set_color('black')
ax1.spines['right'].set_linewidth(2)
ax1.spines['top'].set_linewidth(2)
ax1.spines['bottom'].set_linewidth(2)
ax1.spines['left'].set_color('black')
ax1.spines['left'].set_lw(2)



# for ax1
ax1.tick_params(which='both', width=5)
ax1.tick_params(which='major', length=20)
ax1.tick_params(which='minor', length=5)
ax1.tick_params(which = 'both', direction = 'in')


ax1.grid(True, which = 'major', alpha = 1, linestyle='--', linewidth = 1)
lines_labels = [[ax.get_legend_handles_labels() for ax in fig.axes][0]]
lines, labels = [sum(lol, []) for lol in zip(*lines_labels)]
leg = ax1.legend(lines, labels, loc="upper left", bbox_to_anchor=(0.57, 0.375), fontsize = 20, ncol = 1,markerscale = 1
leg.get_frame().set_edgecolor('black')


ax1.text(0.8, 0.4, 'AUC OVR: '+ str(round(auc_score_ovr, 5)),
    horizontalalignment='center',
```

```
    fontsize=30,fontweight = 'bold',
    transform = ax1.transAxes);


plt.subplots_adjust(wspace=0.2,hspace=0.3)
fig.text(0.5, 0.01, 'False Positive Rate', ha='center', va='center', fontsize = 40)
fig.text(0.06, 0.5, 'True Positive Rate', ha='center', va='center', rotation='vertical', fontsize = 40, color = 'black'
fig.text(0.5, 0.91, 'VGG19 (Pretrained)', ha='center', va='center', fontsize = 40)
```

Out[379]: Text(0.5, 0.91, 'VGG19 (Pretrained)')



AUC is a widely used evaluation metric for binary classification problems. It measures the area under the receiver operating characteristic (ROC) curve, which is a plot of the true positive rate (sensitivity) against the false positive rate (1-specificity) at different classification thresholds. The auc score for individual classes, follwoing one vs rest approach, is also given in the plot.

Making Individual Predictions

In [14]:
```
VGG_Custom = VGG19()
vgg19_pretrained = models.vgg19(pretrained=False)
vgg19_pretrained.classifier[6] = nn.Sequential(
                    nn.Linear(4096, 2048),
                    nn.BatchNorm1d(2048),
                    nn.LeakyReLU(negative_slope=0.01),
                    nn.Dropout(0.25),
                    nn.Linear(2048, 1024),
                    nn.BatchNorm1d(1024),
                    nn.LeakyReLU(negative_slope=0.01),
                    nn.Dropout(0.25),
                    nn.Linear(1024, 5))
```

In [15]:
```
VGG_Custom.load_state_dict(torch.load('VGG19_BN_30_epochs.pth'))
VGG_Custom.eval()
```

Out[15]:
```
VGG19(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (25): ReLU(inplace=True)
    (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (27): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (34): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (35): ReLU(inplace=True)
    (36): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (37): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (38): ReLU(inplace=True)
    (39): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (42): ReLU(inplace=True)
```

```
    (43): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (44): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (45): ReLU(inplace=True)
    (46): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (47): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (48): ReLU(inplace=True)
    (49): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (50): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (51): ReLU(inplace=True)
    (52): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): Dropout(p=0.5, inplace=False)
    (8): Linear(in_features=4096, out_features=5, bias=True)
  )
)
```

In [16]:

```
vgg19_pretrained.load_state_dict(torch.load('VGG19_Pretrained_30_epochs.pth'))
vgg19_pretrained.eval()
```

Out[16]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
```

```
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Sequential(
      (0): Linear(in_features=4096, out_features=2048, bias=True)
      (1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01)
      (3): Dropout(p=0.25, inplace=False)
      (4): Linear(in_features=2048, out_features=1024, bias=True)
      (5): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (6): LeakyReLU(negative_slope=0.01)
      (7): Dropout(p=0.25, inplace=False)
      (8): Linear(in_features=1024, out_features=5, bias=True)
    )
  )
)
```

In [17]:
```
VGG_Custom = VGG_Custom.to('cuda')
vgg19_pretrained = vgg19_pretrained.to('cuda')
```

we will make use of original mean and standard deviation for making individual predctions.

In [39]:
```
image_size = (224, 224)
mean = [0.4588, 0.4198, 0.3006]
std = [0.2941, 0.2635, 0.2864]

# following transformations will be performed on each image
transformations_to_perform = transform=tt.Compose([
                                tt.Resize(image_size),
                                tt.ToTensor(),
                                tt.Normalize(mean, std)])
```

In [99]:
```
'''
This function takes in an image path, gpu or cpu, and a model and makes prediction
'''

def make_individual_predictions(img_path, device, model_local):
    img = transformations_to_perform(Image.open(img_path))
    img = img.reshape(1, 3, 224, 224)
    img = img.to(device)
    output = model_local(img)
    output_softmax = nn.Softmax(dim = 1)(output).cpu().detach().numpy()
    pred = np.argmax(output_softmax)
```

```
    prob = output_softmax[0][pred]

    return pred, prob, dict(zip(n_classes, list(output_softmax[0])))
```

Individual Prediction 1

PREDICTIONS MADE BY THE VGG19 TRAINED FROM SCRATCH

In [111…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\16159487_3a6615a565_n.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, VGG_Custom)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: dandelion, with Predicted Probability of: 0.9578337073326111.
Predicted Probabilities for Each Class:
 {'daisy': 0.018404493, 'dandelion': 0.9578337, 'rose': 0.0033045793, 'sunflower': 0.0032980843, 'tulip': 0.017159138}
```

PREDICTIONS MADE BY PRETRAINED VGG19

In [112…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\16159487_3a6615a565_n.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, vgg19_pretrained)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: dandelion, with Predicted Probability of: 0.9943735599517822.
Predicted Probabilities for Each Class:
 {'daisy': 0.004123726, 'dandelion': 0.99437356, 'rose': 1.5118842e-07, 'sunflower': 0.0013923338, 'tulip': 0.000110192
83}
```

Individual Prediction 2

PREDICTIONS MADE BY THE VGG19 TRAINED FROM SCRATCH

In [113…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\40410814_fba3837226_n.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, VGG_Custom)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: sunflower, with Predicted Probability of: 0.7464743852615356.
Predicted Probabilities for Each Class:
 {'daisy': 0.013468694, 'dandelion': 0.23312251, 'rose': 0.0012241956, 'sunflower': 0.7464744, 'tulip': 0.00571024}
```

PREDICTIONS MADE BY PRETRAINED VGG19

In [114…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\40410814_fba3837226_n.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, vgg19_pretrained)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: sunflower, with Predicted Probability of: 0.9997443556785583.
Predicted Probabilities for Each Class:
 {'daisy': 0.000253121, 'dandelion': 2.4485105e-06, 'rose': 1.7555228e-08, 'sunflower': 0.99974436, 'tulip': 1.5682708e
-08}
```

In [ ]:

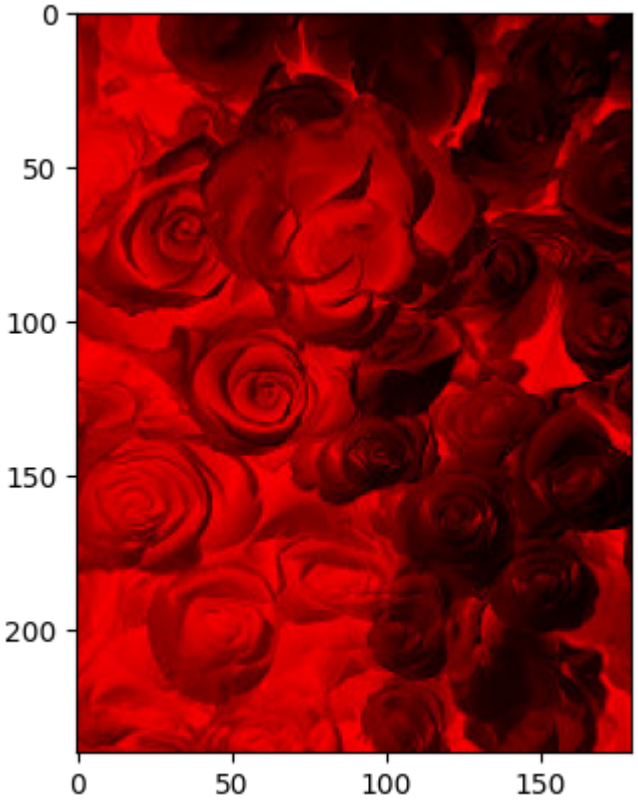
Individual Prediction 3

PREDICTIONS MADE BY THE VGG19 TRAINED FROM SCRATCH

In [116…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\110472418_87b6a3aa98_m.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, VGG_Custom)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: rose, with Predicted Probability of: 0.7779853343963623.
Predicted Probabilities for Each Class:
 {'daisy': 0.052972645, 'dandelion': 0.009258535, 'rose': 0.77798533, 'sunflower': 0.007339353, 'tulip': 0.15244417}
```

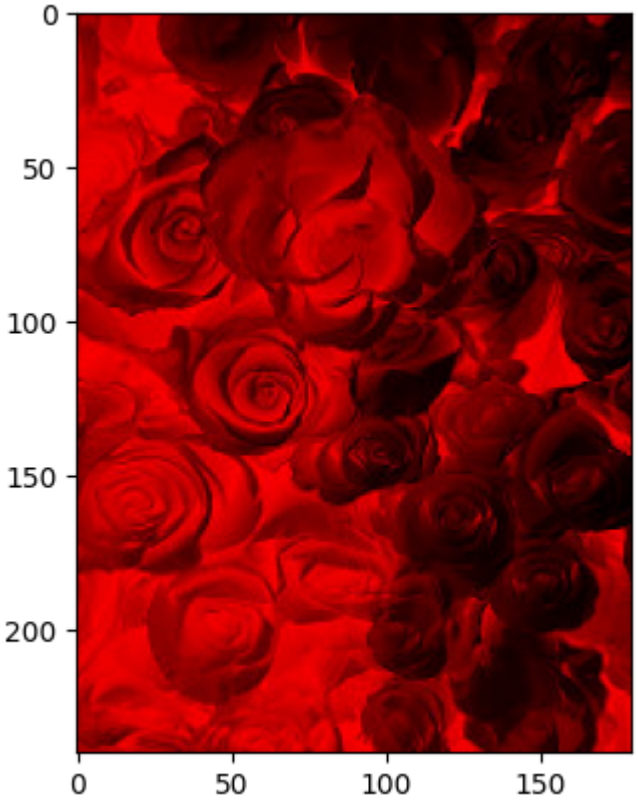


PREDICTIONS MADE BY PRETRAINED VGG19

In [117…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\110472418_87b6a3aa98_m.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, vgg19_pretrained)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: rose, with Predicted Probability of: 0.9999852180480957.
Predicted Probabilities for Each Class:
 {'daisy': 1.943462e-08, 'dandelion': 2.722418e-06, 'rose': 0.9999852, 'sunflower': 3.7585437e-09, 'tulip': 1.2060518e-
05}
```



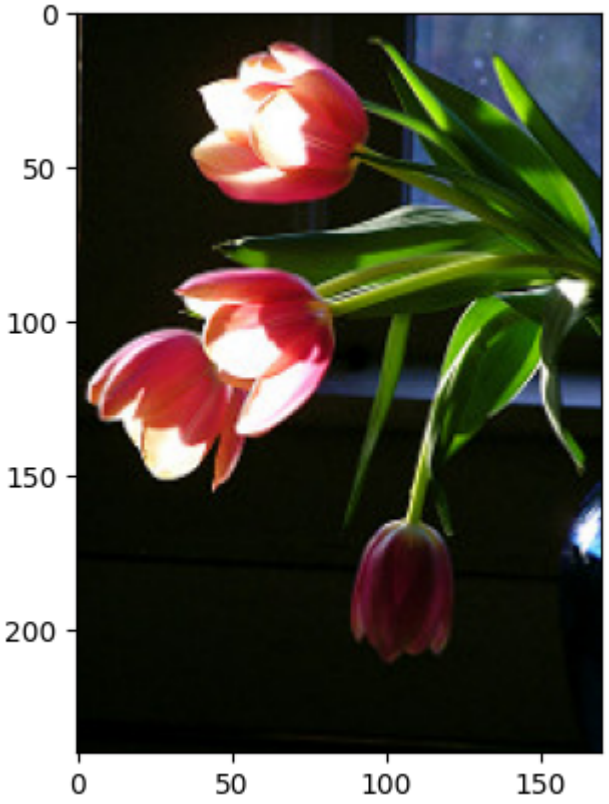
Individual Prediction 4

PREDICTIONS MADE BY THE VGG19 TRAINED FROM SCRATCH

In [118…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\113960470_38fab8f2fb_m.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, VGG_Custom)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: rose, with Predicted Probability of: 0.8201496005058289.
Predicted Probabilities for Each Class:
 {'daisy': 0.050419174, 'dandelion': 0.00597161, 'rose': 0.8201496, 'sunflower': 0.0034130963, 'tulip': 0.12004653}
```

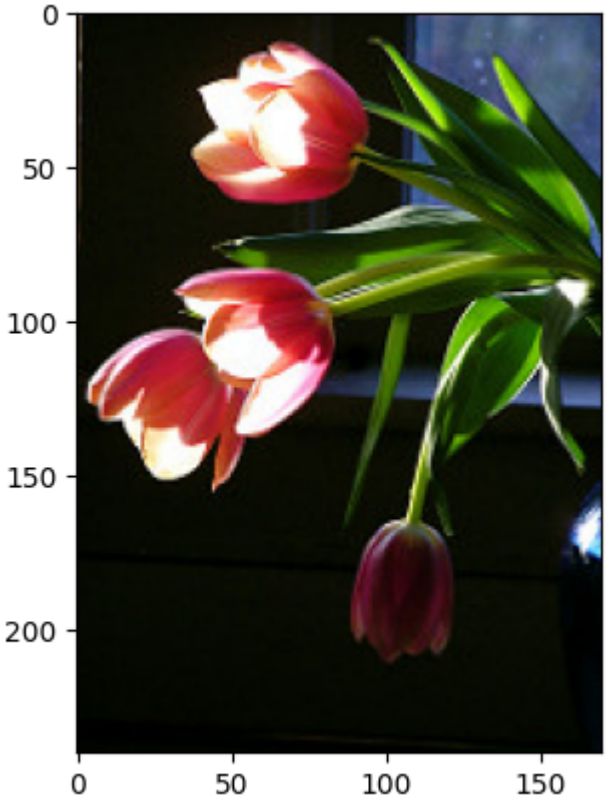


PREDICTIONS MADE BY PRETRAINED VGG19

In [119…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\113960470_38fab8f2fb_m.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, vgg19_pretrained)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: tulip, with Predicted Probability of: 0.8167306780815125.
Predicted Probabilities for Each Class:
 {'daisy': 0.16324136, 'dandelion': 0.0002718655, 'rose': 0.019185064, 'sunflower': 0.0005710468, 'tulip': 0.8167307}
```



Individual Predicion 5

PREDICTIONS MADE BY THE VGG19 TRAINED FROM SCRATCH

In [120…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\153210866_03cc9f2f36.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, VGG_Custom)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: daisy, with Predicted Probability of: 0.7628017067909241.
Predicted Probabilities for Each Class:
 {'daisy': 0.7628017, 'dandelion': 0.03406669, 'rose': 0.13118899, 'sunflower': 0.01895561, 'tulip': 0.052986976}
```

PREDICTIONS MADE BY PRETRAINED VGG19

In [121…

```
img_path = r"C:\Users\Lenovo\Desktop\Dataset_Compressed\Flower Classification\Test\153210866_03cc9f2f36.jpg"

pred, prob, prob_for_all_class = make_individual_predictions(img_path, device, vgg19_pretrained)
print("Predicted Class: {}, with Predicted Probability of: {}.".format(n_classes[pred], prob))
print("Predicted Probabilities for Each Class: \n",prob_for_all_class)
ax = plt.subplot()
ax.imshow(Image.open(img_path))
plt.show()
```

```
Predicted Class: daisy, with Predicted Probability of: 0.9992722868919373.
Predicted Probabilities for Each Class:
 {'daisy': 0.9992723, 'dandelion': 1.8675715e-06, 'rose': 2.7194588e-07, 'sunflower': 0.0007254924, 'tulip': 1.6886469e
-07}
```

In [ ]:

Conclusion

We trained two VGG variants: VGG19 from scratch and VGG19 pretrained model that was trained with imagenet weights. The final classification layer in VGG19-pretrained was replaced with a custom multi-layer perceptron. The scratched-trainedVGG19 achieved the maximum test accuracy of about 77%, whereas the pretrained vgg19 achieved maximum test accuracy of almost 89 %. Since the vgg19 was trained on imagenet dataset that has 1000 class, it could be that the upper layers in vgg19-pretrained were able to capture and extract features from the data more effectively than the scratch-trained vgg19. This showcases the importance and effectiveness of transfer learning, as transfer learing can be used to extract important featrues from pretrained weights to perform predictions on data from different domain.

In this case, the transfer learning resulted in much higher accuracy in just 10 epochs as compared to VGG19 that was trained from scratch for 30 epochs and it could barely reach 77 % accuracy on the test dataset. It highlights the importance and effectiveness of concept of transfer learning.