Project

Training Neural Networks for Handwritten Digit Recognition

Neural Networks

Alia Medhat Mohamed | Artificial Intelligence|20221440735

Malak Mahmoud Aref | Artificial Intelligence|20221445867

Faculty of Computers and Data Sciences,

Alexandria University.

May 2023

Table of Contents

Introduction

The "Petals to Metals" dataset is a comprehensive collection of images used for flower classification tasks. It encompasses 104 different types of flowers, sourced from five publicly available datasets. This dataset aims to classify flowers based on their visual characteristics and features.

Within the dataset, you will encounter classes that are quite specific, representing particular sub-types of flowers, such as pink primroses. On the other hand, some classes are broader and encompass multiple sub-types, like wild roses. The dataset provides a diverse range of flower species, allowing for a comprehensive exploration of flower classification algorithms and techniques.

Importing Packages

Will import all the needed packages we will use in the project

- numpy → is a library for numerical computing in Python.
- matplotlib → is a popular library to plot graphs in Python.
- pandas → is a Python package providing fast, flexible, and expressive data structures
- Sklearn (short for scikit-learn) → is a library for machine learning in Python. It provides tools for data preprocessing, feature selection, model selection, and model evaluation, among other things.
- Math → is a built-in Python module that provides mathematical functions and constants.
- OS → for file and directory operations
- Random → generating random numbers
- Seaborn → creating visualization
- PIL (Python Imaging Library) → module provides image processing capabilities, and ImageFile helps in handling corrupt or incomplete image files.
- TensorFlow → a popular framework for deep learning. It provides tools for building and training neural networks.

Dataset

The "Petals to Metals" dataset is a dataset specifically created for flower classification tasks. It consists of images of flowers collected from five different publicly available datasets. The goal of this dataset is to classify 104 different types of flowers based on their visual features and characteristics.

The dataset includes a wide range of flower species, and each species is represented as a separate class. Some classes in the dataset are quite narrow, meaning they contain only a specific sub-type of a flower. For example, there might be a class specifically for pink primroses, which focuses on images of that particular variation of primroses. On the other hand, some classes in the dataset are broader and encompass multiple sub-types of flowers. For instance, there could be a class for wild roses, which includes various sub-types of roses that are considered wild.

The dataset offers a diverse collection of flower images, allowing for the exploration and development of classification algorithms and models. By utilizing this dataset, researchers and practitioners can train machine learning models to recognize and classify different types of flowers accurately.

It's important to note that without the actual dataset or additional information, the specific details about the number of images per class, image resolutions, or other dataset characteristics cannot be provided.

Downloading the Data

First Option

First, we downloaded the data on google drive. Then we imports the `drive` module from the `google.colab` package. The `drive` module provides functionality to connect and access files from your Google Drive within the Colab environment. Once mounted, we can access files and folders stored in your Google Drive using this directory path.

The purpose of mounting Google Drive is to access and work with files stored in your Drive directly from the Colab notebook. This can be useful when we need to load or save datasets, models, or other files during your development or analysis process.

Then we use a code to extract the zipped file of the data. We created a `ZipFile` object named `zip_ref` by specifying the path to the zip file of our data we want to extract. The second argument `r` indicates that the zip file is opened in read mode.

Then extracts all the contents of the zip file to the specified directory '/content/data'. The `extractall()` method is used to extract all the files and folders from the zip file.

Moreover, the `zip_ref.close()` closes the `ZipFile` object `zip_ref`. It's good practice to close the file after we have finished working with it.

The purpose of this part is to extract the files from the specified zip file into the '/content/data' directory in Google Colab. After executing these lines, the files from the zip file will be available for further processing and analysis in the '/content/data' directory.

Second Option

In Kaggle, we imported the flower classification data and we downloaded vgg16 and resnet50. We also used gpu T4 x2 accelrator. And this is the option we used in our project.

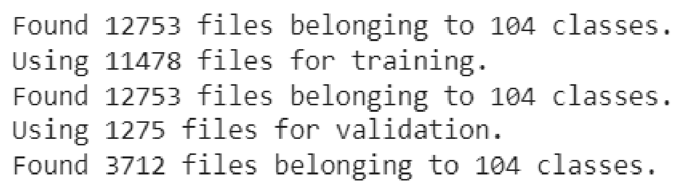
Loading the Dataset

We will load the data using TensorFlow.Keras to create datasets from image files in the given directory paths. The first part creates a training dataset (train_data) using images found in the train path with 10% of data reserved for validation set. It sets the batch size to be 32 and all images are resized to have dimensions (192, 192) pixels before being fed into the model during training.

The second part is similar but instead creates a validation dataset (val_data) using same source folder as above but only includes 10% of data that was not included when creating train dataset earlier during split.

Finally, third part creates a test dataset (test_data) using images found in test directory path and sets its batch size again equal to 32 after resizing each image down to (192,192) pixel resolution like others.

All these datasets will be useful during various stages of deep learning model preparation such as preprocessing inputs or evaluating trained models on unseen data.



```
Found 12753 files belonging to 104 classes.
Using 11478 files for training.
Found 12753 files belonging to 104 classes.
Using 1275 files for validation.
Found 3712 files belonging to 104 classes.
```

Moreover, create a function that loads images and their corresponding labels from a given directory path. It takes the directory path as an argument and returns two numpy arrays: one containing the images and another containing their corresponding labels.

The function first initializes two empty lists for images and labels. It then iterates over each folder in the given directory path and checks if it is a directory. If it is not a directory, it skips to the next iteration. If it is a directory, it extracts the label from the folder name and assigns it to the variable 'label'. It then iterates over each image in the folder and reads it using OpenCV's imread() function. The image is then appended to the 'images' list and its corresponding label is appended to the 'labels' list.

The advantage of this code is that it allow to easily load a large number of images and their corresponding labels from a directory path.

Data Split

Then split the data into training and validation sets. It uses scikit-learn's train_test_split() function to randomly split the data into two sets. The function takes four arguments: the training data (images), the training labels, the test size (which specifies the proportion of the data to use for validation), and a random state (which ensures that the same split is obtained each time the code is run).

The function returns four numpy arrays: x_train (the training images), x_val (the validation images), y_train (the training labels), and y_val (the validation labels).

By splitting the data into training and validation sets, you can train your machine learning model on the training set and evaluate its performance on the validation set. This can help us avoid overfitting your model to the training data and ensure that it generalizes well to new data.

Data Visualization

We will begin by visualizing a subset of the training set randomly by defining a function called `visualize_images` that visualizes a specified number of randomly selected images from each class in a given dataset directory. The function that takes three arguments:

data_dir: A string representing the directory path where images are stored.

num_images_per_class: An integer specifying the number of images to display per class.

images: A list containing image files.

Initialize figure for visualization. The function initializes a figure using `plt.figure` and sets its size to (15, 200) using `figsize`. This figure will be used to display the images.

Then the function iterates over all classes present in the provided dataset directory. The iteration is done using Python's built-in range() function which takes as input length of 'image_files' list (which should be equal to number of classes), and returns sequence numbers starting from 0 up to (length-1).

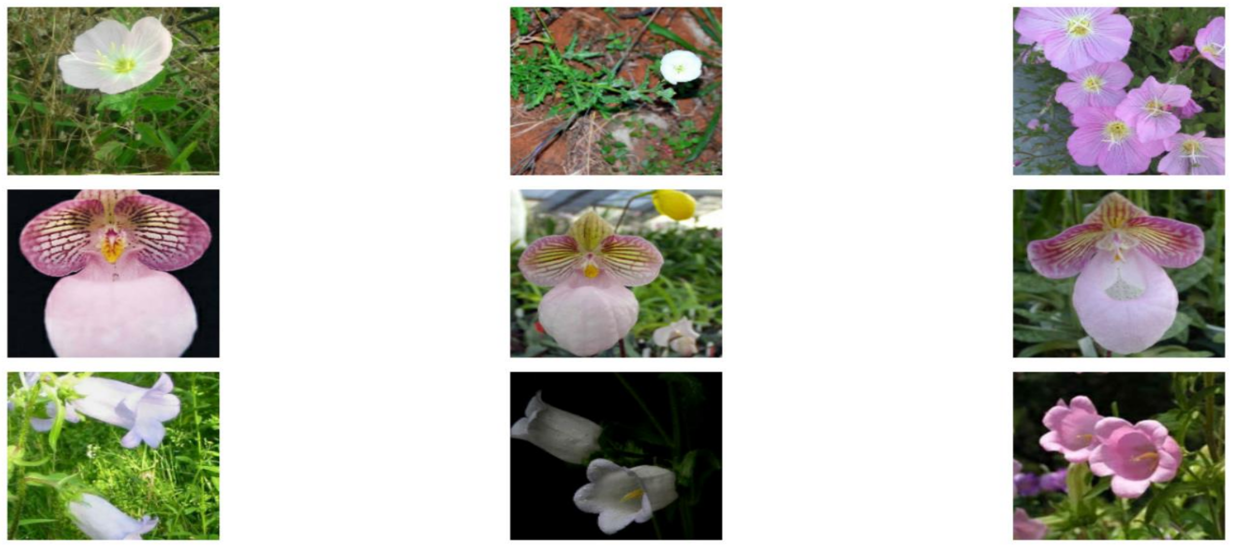
For each class i, it randomly selects num_images_per_class images using random.sample() method applied on list 'images'. These selected images are stored in another variable named 'selected_images'.

Inside second loop for j indexing through selected_images, the corresponding image file path is generated from its filename and parent folder name/class_dir ('data_dir/class_folder/image_file') using os.path.join(). Then OpenCV's cv2.imread method reads this image at path into memory as numpy array img.

After reading an image file successfully, it creates subplot objects inside main Matplotlib figure object one-by-one via fig.add_subplot(len(classes), 3, i * num_images_per_class + j + 1). Here len(classes) represents total number of unique classes or labels present in data set. It then displays this current subplot containing loaded/selected image onto screen via ax.imshow(image) call followed by hiding axis information such as ticks or ticklabels around displayed image area via ax.axis('off').

Finally after iterating over all available images across different classes/subsets,it arranges subplots in a tight grid using plt.tight_layout() and displays the entire figure on screen using plt.show(). The resulting figure will have three columns of images per class, with each row representing one class.

Sample Output



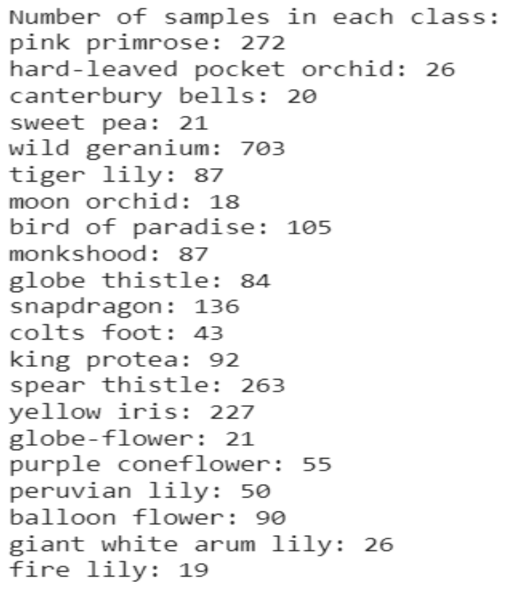
Then extract class names from the text file named 'Classes.txt', create a dictionary object with keys representing each unique class name, initialized to 0 count, and then print out the number of samples in each class present in the dataset.

The code reads the contents of 'Classes.txt' file using Python's built-in open() function. It extracts all available classes information from this file by splitting its first line at '=' sign position followed by some string replace calls used for removing brackets or quotes around individual class names if any. The extracted classes list is then stored inside variable classes.

After initializing empty dictionary class_counts, it prints out label/class counts per category on screen via iteration over each key-value pair within previously created class_counts dictionary. For every key-value pair found inside this dict object,it extracts both current label/classname (class_name) as well as its corresponding sample count/occurrences frequency(count). These values are then printed using f-string formatted output syntax which allows us to insert variables directly into strings.

The resulting output will show total number of samples belonging to each label/category that were counted so far during data preparation phase before training/testing machine learning models.

Number of samples in each class



```
Number of samples in each class:
pink primrose: 272
hard-leaved pocket orchid: 26
canterbury bells: 20
sweet pea: 21
wild geranium: 703
tiger lily: 87
moon orchid: 18
bird of paradise: 105
monkshood: 87
globe thistle: 84
snapdragon: 136
colts foot: 43
king protea: 92
spear thistle: 263
yellow iris: 227
globe-flower: 21
purple coneflower: 55
peruvian lily: 50
balloon flower: 90
giant white arum lily: 26
fire lily: 19
```

Data Preprocessing

Will create two image data generators for the training and test sets. Image data generators are used to preprocess images before they are fed into a machine learning model.

The first data generator (train_generator) applies several image augmentation techniques to the training images. These techniques include rescaling the pixel values between 0 and 1, vertical and horizontal transpositions, random rotations at 90 degrees, shifting the height of the image by 30%, and decreasing/increasing the brightness of the image within a specified range.

The second data generator (test_generator) applies a preprocessing function called preprocess_input() to the test images. This function is specific to the VGG16 model and is used to preprocess the images in a way that is compatible with the model's architecture.

By using these data generators, we can preprocess our images in a standardized way that is compatible with our machine learning model. This can help improve the performance of the model and make it more robust to variations in the input data.

Then, will define the path to the training directory to get a list of all the subfolders in the training directory using os.listdir(). The subfolders are sorted in ascending order using the sort() method with the key=float argument.

The key argument specifies a function of one argument to extract a comparison key from each element in the list. In this case, the key is set to float, which converts each subfolder name to a float before sorting them. This ensures that the subfolders are sorted in numerical order rather than lexicographical order. By sorting the subfolders in ascending order, we can ensure that the labels assigned to each class are consistent across different runs of the code. This can help avoid errors and make your code more reproducible.

Finally, create three data generators for the training, validation, and test sets.

The first data generator (traingen) is created using the train_generator object created earlier. It takes the training images (x_train) as input and generates batches of augmented images for training the machine learning model. The batch size is set to 32, which means that each batch will contain 32 images. The shuffle parameter is set to True, which means that the order of the images in each batch will be randomized. The seed parameter is set to 42, which ensures that the same random order is obtained each time the code is run.

The second data generator (validgen) is created in a similar way to traingen, but it takes the validation images (x_val) as input instead.

The third data generator (testgen) is created using the test_generator object created earlier. It takes the test images (imagestest) as input and generates batches of preprocessed images for testing the machine learning model. The batch size, shuffle, and seed parameters are set in the same way as traingen and validgen.

By using these data generators, we can efficiently generate batches of training and test data for our machine learning model. This can help improve the performance of the model and make it more robust to variations in the input data.

Training The Preprocessed Data

Build CNN Model

Define a convolutional neural network (CNN) model using the Keras framework.

The function `model_cnn` takes two arguments: `input_shape`, which represents the shape of the input data, and `num_classes`, which represents the number of classes in the classification problem.

The `Sequential` class is initialized to create a linear stack of layers for the model.

The first layer added to the model is a `Conv2D` layer with 32 filters, a kernel size of 3x3, and a ReLU activation function. This layer processes the input data by convolving the filters over the input image, producing feature maps. The `input_shape` argument is used to specify the shape of the input data for the first layer.

After the convolutional layer, a `BatchNormalization` layer is added. This layer normalizes the activations of the previous layer, helping with the gradient flow and improving training stability.

Two more sets of convolutional layers with batch normalization follow. Each set contains a `Conv2D` layer with 32 filters, a kernel size of 3x3, and a ReLU activation function. These layers extract additional features from the previous layers' outputs.

The next convolutional layer has 32 filters, a kernel size of 5x5, and a stride of 2. The stride of 2 reduces the spatial dimensions of the output, effectively downsampling the feature maps. This layer helps capture higher-level features by looking at larger receptive fields.

Another `BatchNormalization` layer is added after the downsampling convolutional layer.

A `Dropout` layer is introduced with a dropout rate of 0.4. Dropout randomly sets a fraction of input units to 0 during training, which helps prevent overfitting by reducing interdependencies between neurons.

Two more sets of convolutional layers are added, similar to the previous ones, but with 64 filters instead of 32. These layers further capture and transform the learned features.

Another downsampling convolutional layer with 64 filters, a kernel size of 5x5, and a stride of 2 is added. This layer reduces the spatial dimensions further.

Another `BatchNormalization` layer is added after the second downsampling convolutional layer.

A `Dropout` layer with a dropout rate of 0.6 is introduced.

A `Conv2D` layer with 128 filters and a kernel size of 4x4 is added. This layer captures more complex and abstract features from the previous layers.

A `BatchNormalization` layer is added after the last convolutional layer.

The feature maps are flattened using the `Flatten` layer to prepare them for the fully connected layers.

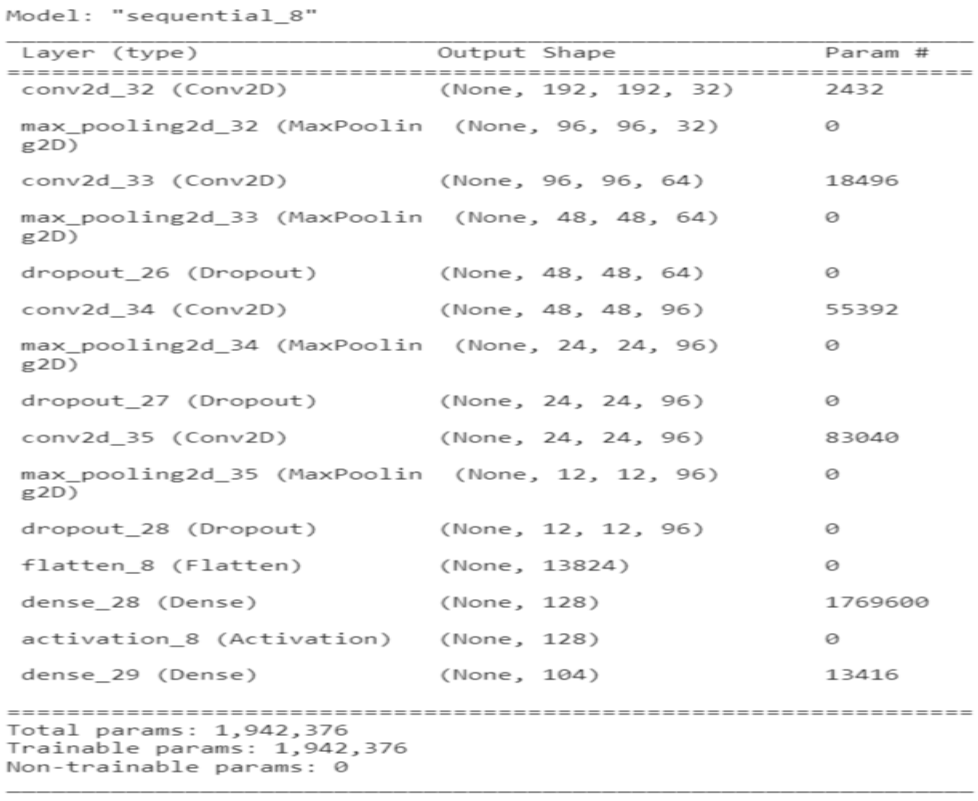
Another `Dropout` layer with a dropout rate of 0.4 is added.

The last layer of the model is a fully connected `Dense` layer with the number of neurons equal to the `num_classes`. This layer uses a softmax activation function to output probabilities for each class.

The model is compiled using the Adam optimizer, sparse categorical cross-entropy loss, and accuracy as the evaluation metric. Finally, the compiled model is returned.

Overall, this CNN architecture consists of several convolutional layers, batch normalization layers, dropout layers, and a fully connected layer, designed to extract features from input images and classify them into the specified number of classes.

Model Summary



```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_32 (Conv2D)          (None, 192, 192, 32)      2432

 max_pooling2d_32 (MaxPoolin  (None, 96, 96, 32)       0
 g2D)

 conv2d_33 (Conv2D)          (None, 96, 96, 64)        18496

 max_pooling2d_33 (MaxPoolin  (None, 48, 48, 64)       0
 g2D)

 dropout_26 (Dropout)        (None, 48, 48, 64)        0

 conv2d_34 (Conv2D)          (None, 48, 48, 96)        55392

 max_pooling2d_34 (MaxPoolin  (None, 24, 24, 96)       0
 g2D)

 dropout_27 (Dropout)        (None, 24, 24, 96)        0

 conv2d_35 (Conv2D)          (None, 24, 24, 96)        83040

 max_pooling2d_35 (MaxPoolin  (None, 12, 12, 96)       0
 g2D)

 dropout_28 (Dropout)        (None, 12, 12, 96)        0

 flatten_8 (Flatten)         (None, 13824)             0

 dense_28 (Dense)            (None, 128)               1769600

 activation_8 (Activation)   (None, 128)               0

 dense_29 (Dense)            (None, 104)               13416

=================================================================
Total params: 1,942,376
Trainable params: 1,942,376
Non-trainable params: 0
_________________________________________________________________
```

`PlotLossesCallback()` is a callback function that plots the training and validation loss at the end of each epoch.

The function is used to visualize the training and validation loss during training. It is useful for monitoring the progress of the model during training and identifying overfitting.

The `ModelCheckpoint()` callback saves the best weights of the model during training. The `EarlyStopping()` callback stops training when a monitored metric has stopped improving.

Train the model

Train the Keras model on the training data.

The first block of code trains the model using the fit() method. It takes the training images (imagestrain) and labels (labelstrain) as input, along with several other parameters. The epochs parameter specifies the number of times to iterate over the entire training dataset. The verbose parameter controls the amount of output printed during training. The validation_data parameter specifies the validation dataset to use during training. Finally, the callbacks parameter specifies a list of callbacks to use during training.

The second block of code is similar to the first, but it uses data generators (traingen and valgen) instead of raw data arrays (imagestrain and labelstrain). This can help improve the performance of your model and make it more robust to variations in the input data.

By training the model on the training data, we can optimize its parameters to perform well on your image classification task.

Plotting the Accuracy and Loss

Plot the training and validation metrics of the Keras model.

The plot_metrics() function takes two arguments: the history object returned by the fit() method and a list of metrics to plot. It creates a subplots figure with one subplot for each metric in the list. For each metric, it plots the training and validation values over the epochs of training.

The metric_list argument specifies which metrics to plot. In this case, it includes the loss and accuracy metrics.

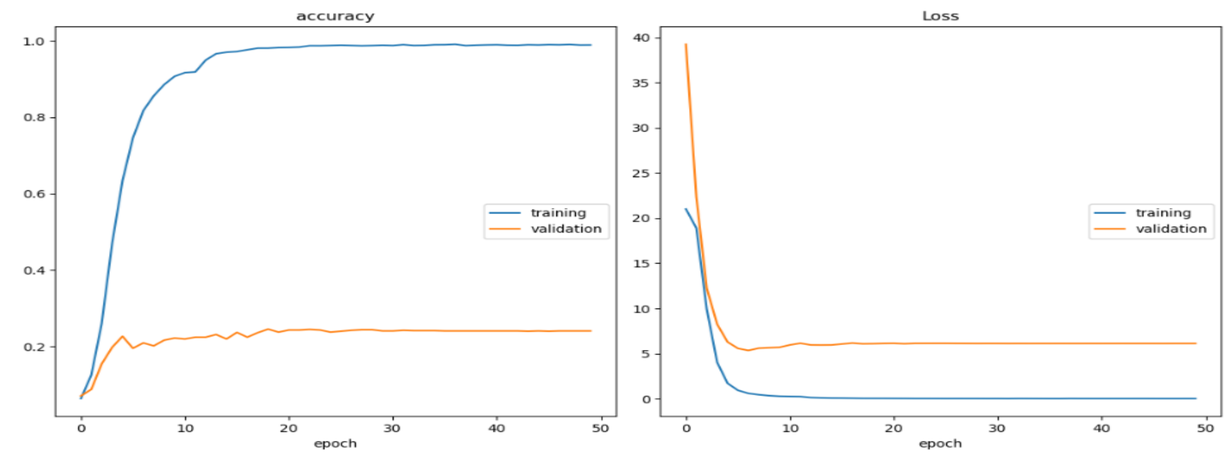
By plotting the training and validation metrics, we can visualize how well the model is performing during training and identify any issues that may need to be addressed.

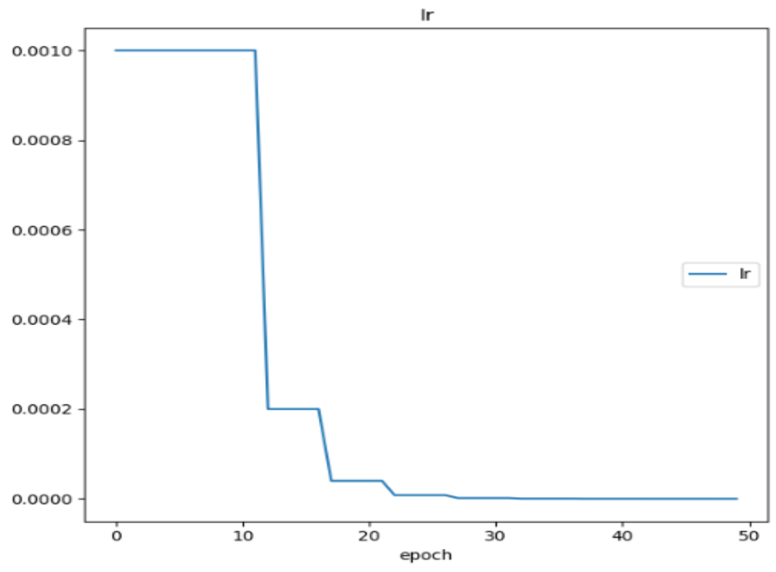
Plot the training and validation accuracy & loss of the Keras model before data preprocessing

The plot shows the training and validation loss over the epochs of training. The x-axis represents the number of epochs, and the y-axis represents the loss value. The blue line represents the training loss, and the orange line represents the validation loss.

The plot shows the training and validation accuracy over the epochs of training. The x-axis represents the number of epochs, and the y-axis represents the accuracy value. The blue line represents the training accuracy, and the orange line represents the validation accuracy.

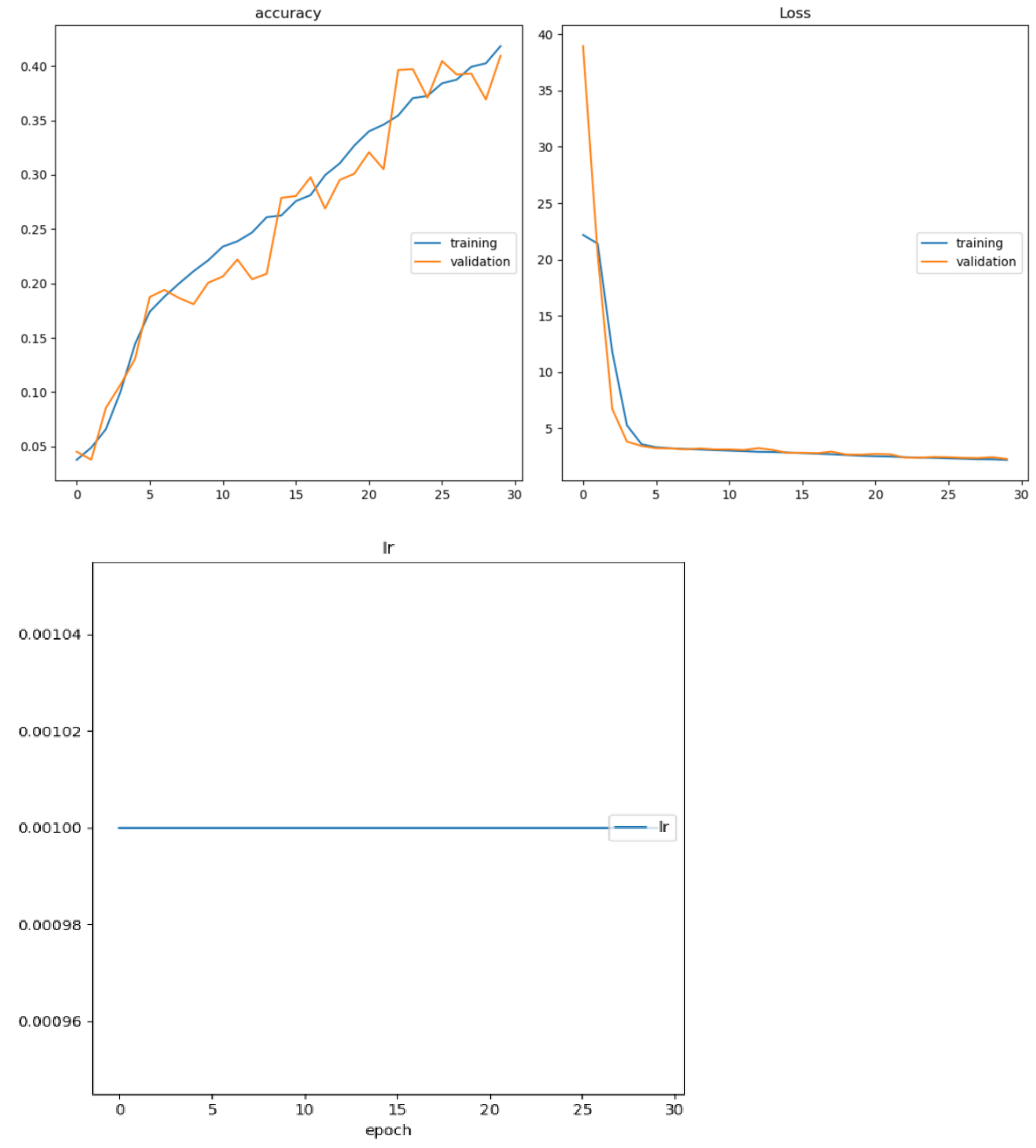
50 epochs





```
accuracy
        training                 (min:    0.064, max:    0.991, cur:    0.989)
        validation               (min:    0.071, max:    0.245, cur:    0.241)
Loss
        training                 (min:    0.031, max:   21.008, cur:    0.034)
        validation               (min:    5.358, max:   39.232, cur:    6.136)
lr
        lr                       (min:    0.000, max:    0.001, cur:    0.000)
359/359 - 73s - loss: 0.0342 - accuracy: 0.9893 - val_loss: 6.1360 - val_accuracy: 0.2408 - lr: 2.5600e-09 - 73s/epoch - 203ms/step
CPU times: user 1h 10min 29s, sys: 2min 58s, total: 1h 13min 27s
Wall time: 1h 6min 22s
```

Plot the training and validation accuracy & loss of the Keras model after data preprocessing

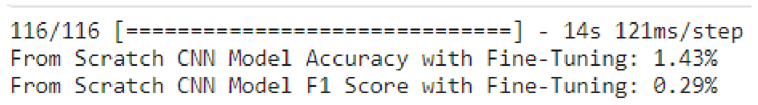


```
accuracy
        training                  (min:    0.038, max:    0.418, cur:    0.418)
        validation                (min:    0.038, max:    0.410, cur:    0.410)
Loss
        training                  (min:    2.214, max:   22.181, cur:    2.214)
        validation                (min:    2.289, max:   38.945, cur:    2.289)
lr
        lr                        (min:    0.001, max:    0.001, cur:    0.001)
360/360 - 177s - loss: 2.2139 - accuracy: 0.4184 - val_loss: 2.2890 - val_accuracy: 0.4095 - lr: 0.0010 - 177s/epoch - 493ms/step
CPU times: user 1h 44min 52s, sys: 1min 23s, total: 1h 46min 15s
Wall time: 1h 31min 38s
```

Evaluate the performance of a Convolutional Neural Network (CNN) model. The model is trained on a dataset and then tested on a separate dataset.

First, predict the output of the model on the test dataset. Then returns the predicted class for each input in the test dataset.

Calculate the accuracy score of the model by comparing the predicted classes with the true classes. And calculates the F1 score of the model. Then print out the accuracy score and F1 score of the model.



```
116/116 [==============================] - 14s 121ms/step
From Scratch CNN Model Accuracy with Fine-Tuning: 1.43%
From Scratch CNN Model F1 Score with Fine-Tuning: 0.29%
```

VGG16

Create a Keras model that integrates with the VGG16 pretrained layers.

The create_model() function takes three arguments: the input shape of the images (width, height, channels), the number of classes for the output layer, and an optimizer to use for training. It also includes an optional parameter for fine-tuning the pretrained layers.

The function first loads the VGG16 convolutional base with Imagenet weights (imported from another data resource from kaggle) and excludes the fully-connected layers. It then defines how many layers to freeze during training based on the fine-tuning parameter. If fine_tune is greater than 0, it freezes all but the last fine_tune layers of the convolutional base. Otherwise, it freezes all layers of the convolutional base.

The function then creates a new 'top' of the model by adding fully-connected layers onto the pretrained layers. The output layer has a softmax activation function and a number of units equal to the number of classes.

Finally, the function compiles the model for training using the specified optimizer and loss function.

By creating a model that integrates with pretrained layers, we can take advantage of their feature extraction capabilities and improve the performance of your image classification task.

Compile a Keras model that integrates with the VGG16 pretrained layers by defining several variables, including the input shape of the images (width, height, channels), the optimizer to use for training, the number of classes for the output layer, and the batch size for training.

It then calculates the number of steps per epoch for the training and validation datasets based on the batch size. Finally, it creates a new Keras model using the create_model() function defined earlier and compiles it for training using the specified optimizer and fine-tuning parameter. By compiling your model with an optimizer and fine-tuning parameter, we can train it on the image classification and optimize its performance.

In TensorFlow, `tf.test.is_gpu_available()` is a function that returns a Boolean indicating whether TensorFlow can access at least one GPU. The function is used to check if the system has a GPU available for use by TensorFlow. If the function returns True, it means that TensorFlow can access at least one GPU and can use it for computations.

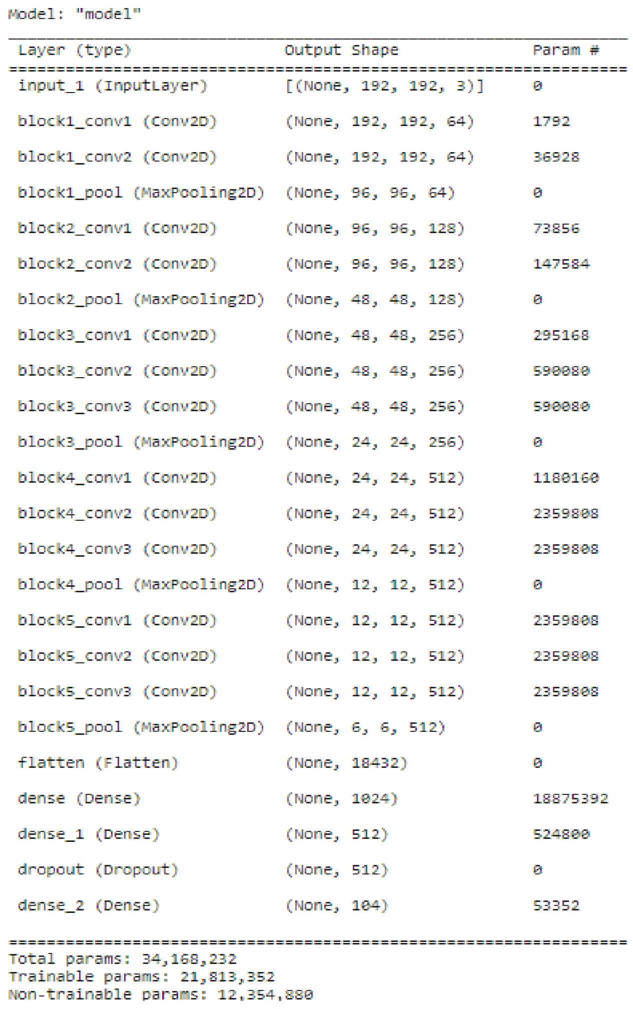
`PlotLossesCallback()` is a callback function that plots the training and validation loss at the end of each epoch.

The function is used to visualize the training and validation loss during training. It is useful for monitoring the progress of the model during training and identifying overfitting.

The `ModelCheckpoint()` callback saves the best weights of the model during training. The `EarlyStopping()` callback stops training when a monitored metric has stopped improving.

Recompile the model, with setting the `fine_tune` parameter to 2 which means that the last two layers of the VGG16 model will be left unfrozen for fine-tuning. This allows the model to learn more specific features from the dataset.

Model Summary



```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 192, 192, 3)]     0

 block1_conv1 (Conv2D)       (None, 192, 192, 64)      1792

 block1_conv2 (Conv2D)       (None, 192, 192, 64)      36928

 block1_pool (MaxPooling2D)  (None, 96, 96, 64)        0

 block2_conv1 (Conv2D)       (None, 96, 96, 128)       73856

 block2_conv2 (Conv2D)       (None, 96, 96, 128)       147584

 block2_pool (MaxPooling2D)  (None, 48, 48, 128)       0

 block3_conv1 (Conv2D)       (None, 48, 48, 256)       295168

 block3_conv2 (Conv2D)       (None, 48, 48, 256)       590080

 block3_conv3 (Conv2D)       (None, 48, 48, 256)       590080

 block3_pool (MaxPooling2D)  (None, 24, 24, 256)       0

 block4_conv1 (Conv2D)       (None, 24, 24, 512)       1180160

 block4_conv2 (Conv2D)       (None, 24, 24, 512)       2359808

 block4_conv3 (Conv2D)       (None, 24, 24, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 12, 12, 512)       0

 block5_conv1 (Conv2D)       (None, 12, 12, 512)       2359808

 block5_conv2 (Conv2D)       (None, 12, 12, 512)       2359808

 block5_conv3 (Conv2D)       (None, 12, 12, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 6, 6, 512)         0

 flatten (Flatten)           (None, 18432)             0

 dense (Dense)               (None, 1024)              18875392

 dense_1 (Dense)             (None, 512)               524800

 dropout (Dropout)           (None, 512)               0

 dense_2 (Dense)             (None, 104)               53352

=================================================================
Total params: 34,168,232
Trainable params: 21,813,352
Non-trainable params: 12,354,880
```

Train the model

In Keras, `fit()` is a method that trains the model for a fixed number of epochs on the training data .

The `%%time` command is used to measure the execution time of the cell.

Train the model using the training data (`traingen`) and validate it using the validation data (`validgen`). The `batch_size` parameter specifies the number of samples per gradient update while `epochs` specifies the number of epochs to train the model.

The `steps_per_epoch` parameter specifies the number of steps (batches) to yield from the generator before declaring one epoch finished and starting the next epoch.

The `validation_steps` parameter specifies the number of steps (batches) to yield from the validation generator before stopping at the end of every epoch.

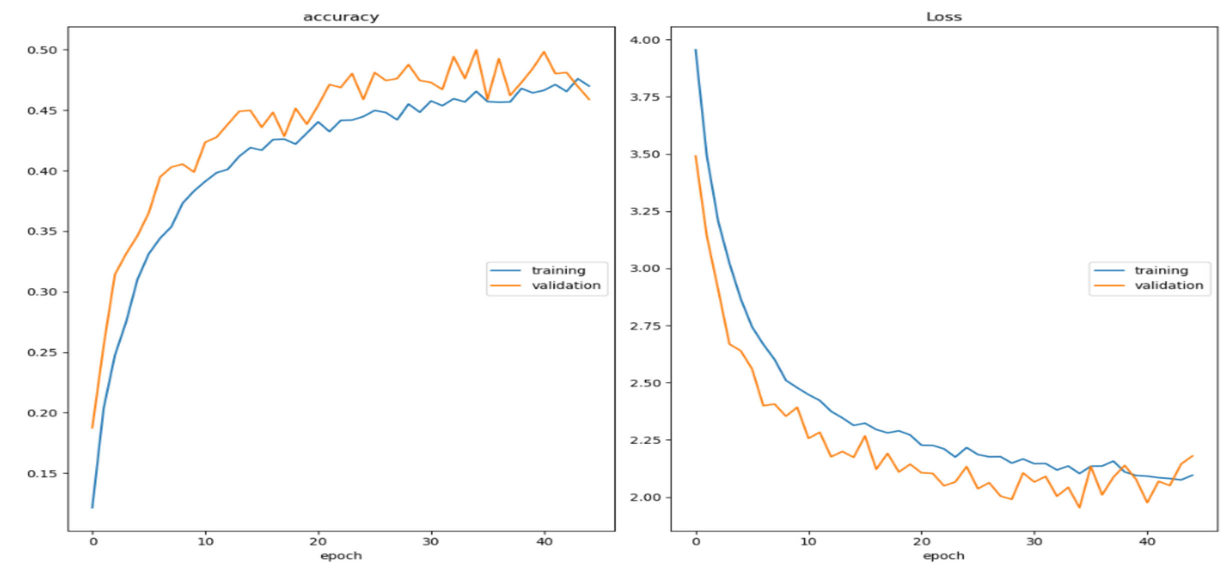
The `callbacks` parameter specifies a list of callbacks to apply during training.

The `tl_checkpoint_1`, `early_stop`, and `plot_loss_1` are used as callbacks.

Plotting the Accuracy and Loss

Plot the training and validation accuracy & loss of the Keras model model after data preprocessing

50 epochs

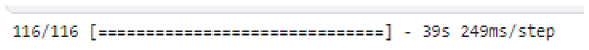


```
accuracy
        training                 (min:    0.121, max:    0.476, cur:    0.470)
        validation               (min:    0.188, max:    0.500, cur:    0.459)
Loss
        training                 (min:    2.075, max:    3.956, cur:    2.096)
        validation               (min:    1.953, max:    3.491, cur:    2.180)
360/360 [==============================] - 164s 456ms/step - loss: 2.0956 - accuracy: 0.4699 - val_loss: 2.1797 - val_accuracy: 0.4589
CPU times: user 1h 59min 18s, sys: 2min 6s, total: 2h 1min 24s
Wall time: 2h 11min 31s
```

Generate predictions for the model by loading the best trained weights for the model.

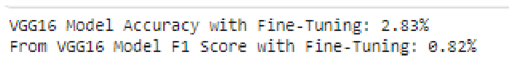
Then gets the true classes for the test dataset and the class indices for the training dataset. Creates a dictionary that maps the class indices to their respective class names.

Then generates predictions for the test dataset using the trained model and returns the predicted class for each input in the test dataset.



```
116/116 [==============================] - 39s 249ms/step
```

Calculate the accuracy and F1 score for the VGG16 model by comparing the true classes to the predicted classes. And calculates the F1 score of the model.



```
VGG16 Model Accuracy with Fine-Tuning: 2.83%
From VGG16 Model F1 Score with Fine-Tuning: 0.82%
```

ResNet50

Import the ResNet50 model from Keras and set its parameters.

The `for layer in imported_model.layers: layer.trainable=False` part is used to freeze the weights of the layers in the ResNet50 model so that they are not updated during training. This is done to prevent overfitting and improve the performance of the model.

Create a sequential model in Keras. The `dnn_model.add(imported_model)` is used to add the ResNet50 model to the sequential model.

Then, flatten the output of the ResNet50 model so that it can be passed through a dense layer.

Add a dense layer with 256 neurons and a ReLU activation function to the model.

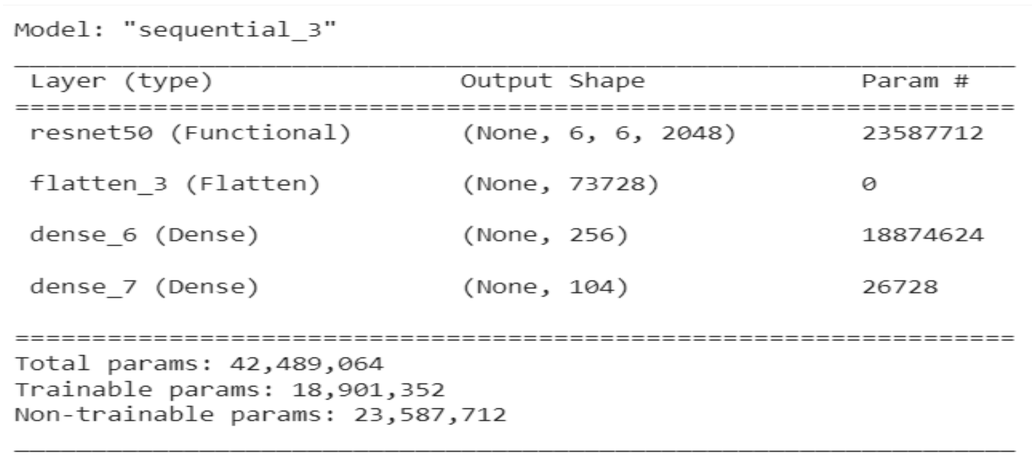
Add another dense layer with 104 neuron and a softmax activation function to the model.

In Keras, the `summary()` method is used to print a summary of the model.

Use summary() function to print a summary of the `dnn_model` model that we created.

The summary includes information about the layers in the model, the output shape of each layer, and the number of parameters in each layer.

Model Summary



```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 6, 6, 2048)        23587712

 flatten_3 (Flatten)         (None, 73728)             0

 dense_6 (Dense)             (None, 256)               18874624

 dense_7 (Dense)             (None, 104)               26728

=================================================================
Total params: 42,489,064
Trainable params: 18,901,352
Non-trainable params: 23,587,712
_________________________________________________________________
```

`PlotLossesCallback()` is a callback function that plots the training and validation loss at the end of each epoch.

The function is used to visualize the training and validation loss during training. It is useful for monitoring the progress of the model during training and identifying overfitting.

The `ModelCheckpoint()` callback saves the best weights of the model during training. The `EarlyStopping()` callback stops training when a monitored metric has stopped improving.

Train the model

In Keras, the `compile()` method is used to configure the model for training. Compile the `dnn_model` model that we created.

The `optimizer` parameter specifies the optimizer to use during training. In this case, the Adam optimizer is used with a learning rate of 0.001.

The `loss` parameter specifies the loss function to use during training. In this case, the sparse categorical crossentropy loss function is used.

The `metrics` parameter specifies the metrics to evaluate during training. In this case, accuracy is used as the metric.

The `fit()` method is used to train the model on the training data. The `train_data` parameter specifies the training data to use during training. The `validation_data` parameter specifies the validation data to use during training. The `epochs` parameter specifies the number of epochs to train the model for .

For the histroyresnet, The `traingen` parameter specifies the training data generator to use during training. The `validation_data` parameter specifies the validation data generator to use during training. The `epochs` parameter specifies the number of epochs to train the model for. The `historyresnet` variable is used to store the history of the training process.

Plotting the Accuracy and Loss

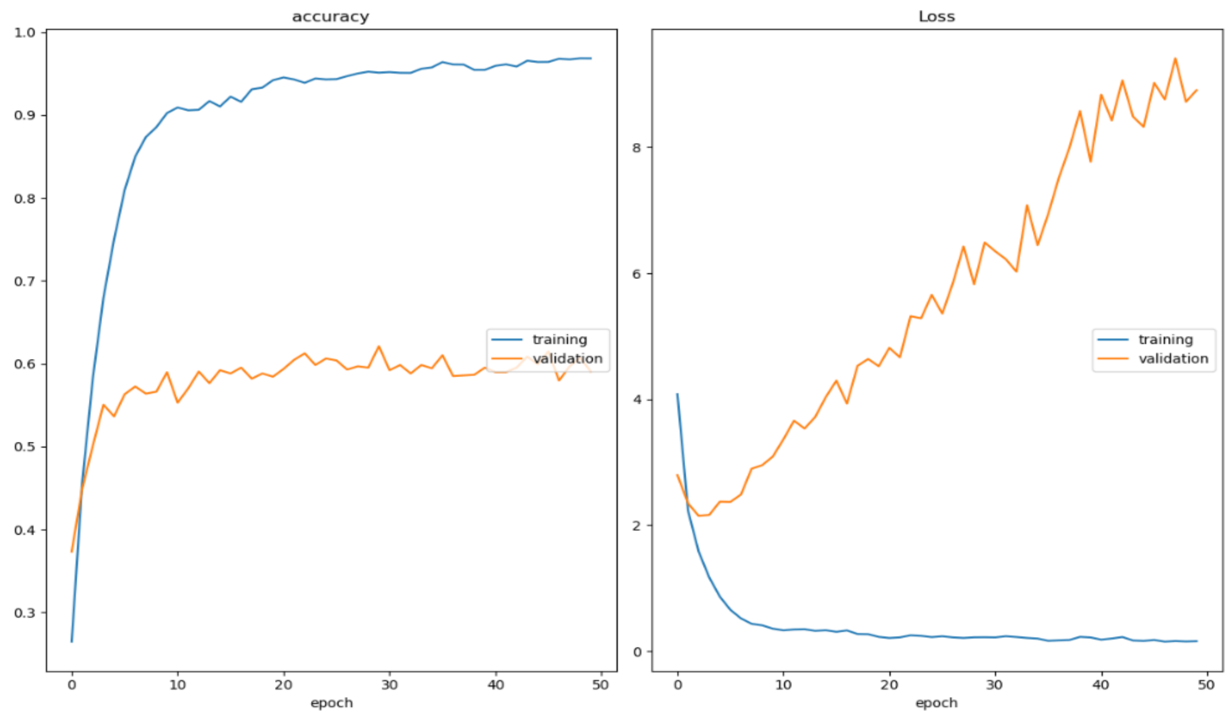
Plot the training and validation accuracy & loss of the model over the epochs before data preprocessing

The training loss is the error of the model on the training data, while the validation loss is the error of the model on the validation data.

The plot shows how well the model is learning over time. If the training loss decreases but the validation loss increases, it means that the model is overfitting to the training data and not generalizing well to new data.

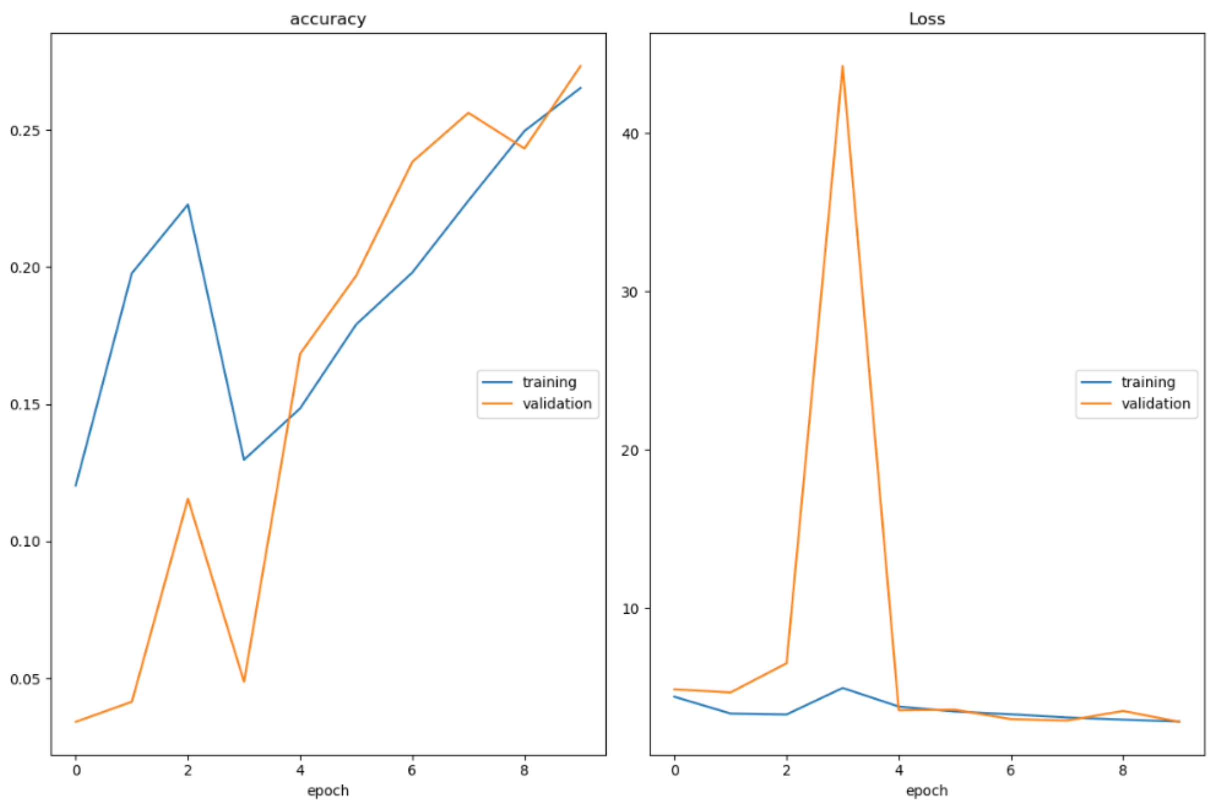
The training accuracy is the accuracy of the model on the training data, while the validation accuracy is the accuracy of the model on the validation data.

The plot shows how well the model is learning over time. If the training accuracy increases but the validation accuracy decreases, it means that the model is overfitting to the training data and not generalizing well to new data.



```
accuracy
        training                  (min:    0.265, max:    0.968, cur:    0.968)
        validation                (min:    0.373, max:    0.621, cur:    0.591)
Loss
        training                  (min:    0.150, max:    4.081, cur:    0.157)
        validation                (min:    2.148, max:    9.414, cur:    8.909)
359/359 [==============================] - 31s 85ms/step - loss: 0.1572 - accuracy: 0.9684 - val_loss: 8.9091 - val_accuracy: 0.5906
CPU times: user 34min 28s, sys: 3min 9s, total: 37min 37s
Wall time: 33min 29s
```

Plot the training and validation accuracy & loss of the model over the epochs after data preprocessing



```
accuracy
        training                 (min:    0.120, max:    0.265, cur:    0.265)
        validation               (min:    0.034, max:    0.273, cur:    0.273)
Loss
        training                 (min:    2.815, max:    4.938, cur:    2.815)
        validation               (min:    2.801, max:   44.262, cur:    2.801)
361/361 [==============================] - 183s 506ms/step - loss: 2.8150 - accuracy: 0.2654 - val_loss: 2.8012 - val_accuracy: 0.2734
CPU times: user 38min 9s, sys: 31.7 s, total: 38min 41s
Wall time: 32min 13s
```

Generate predictions for ResNet50 architecture by loading the best trained weights for the model.

Generates predictions for the test dataset using the trained model. Then returns the predicted class for each input in the test dataset.

```
116/116 [==============================] - 42s 358ms/step
```

Calculate the accuracy and F1 score for the ResNet50.

```
Resnet50 Model Accuracy with Fine-Tuning: 4.58%
From Resnet50 Model F1 Score with Fine-Tuning: 0.50%
```

Model Evaluation

Test Set

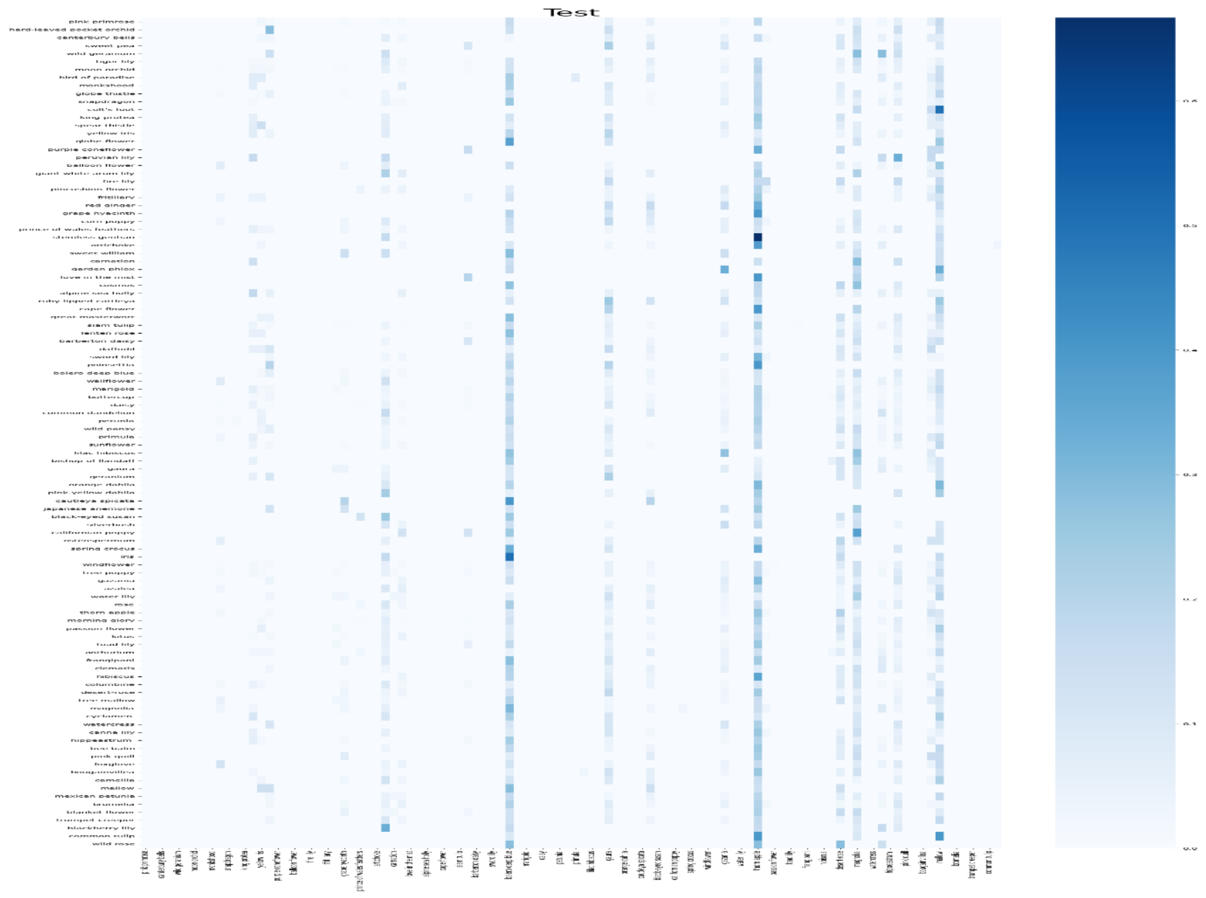
CNN

A confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. It allows visualization of the performance of an algorithm. The matrix compares the predicted values with the actual values and shows how many predictions were correct and how many were incorrect.

Generate a confusion matrix using the `confusion_matrix()` function from scikit-learn library. The confusion matrix is then normalized by dividing each row by the sum of that row. This gives us the percentage of correct predictions for each class.

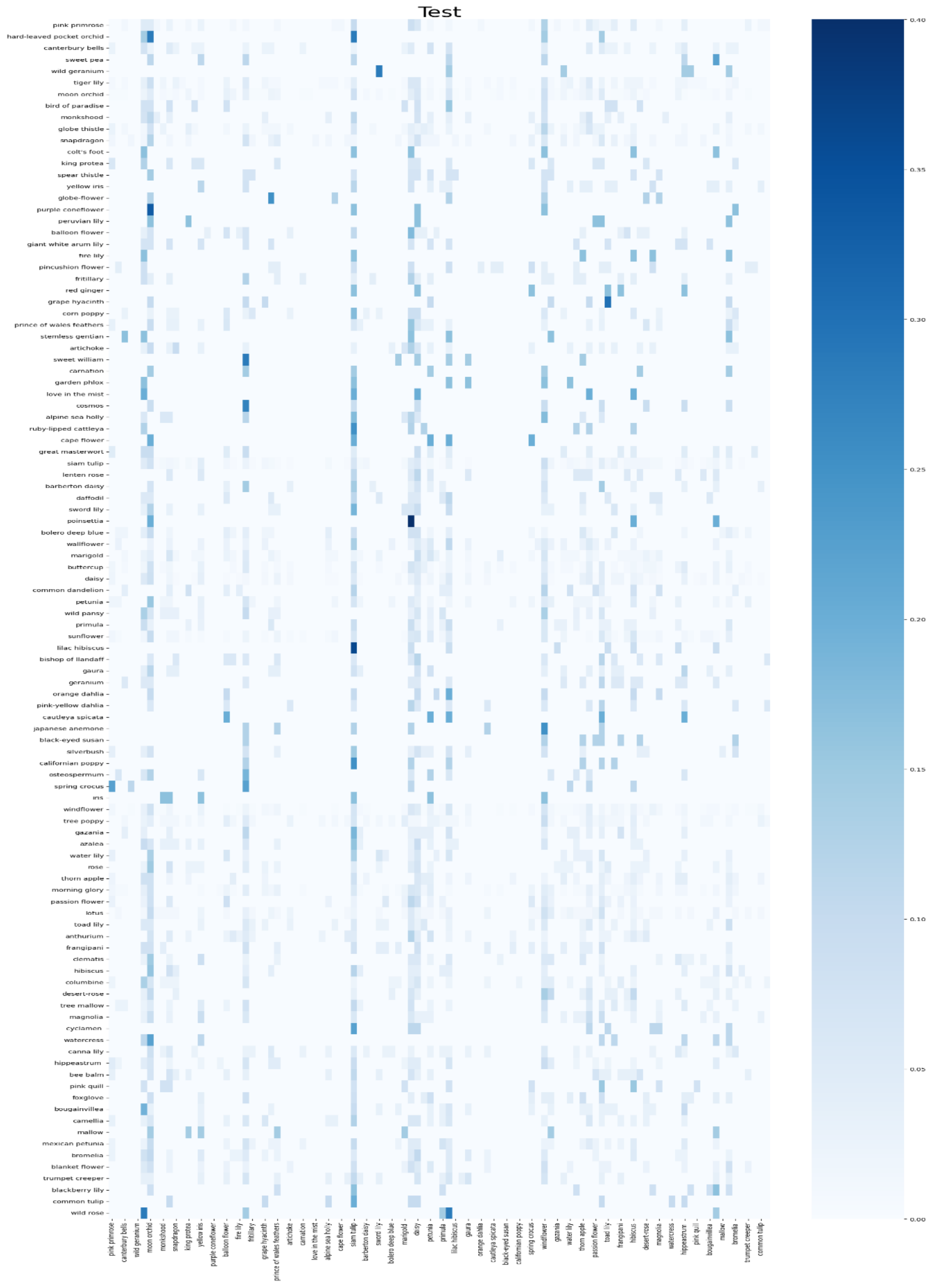
The `confusion_matrix()` function takes two arguments: `true_classes` and `cnn_pred_classes`. The `true_classes` are the actual classes of the test data, while `cnn_pred_classes` are the predicted classes generated by your model. The function returns a matrix that shows how many times each class was predicted correctly or incorrectly.

The resulting confusion matrix is then plotted using `seaborn` library's heatmap function. The heatmap shows how well your model performed on each class.



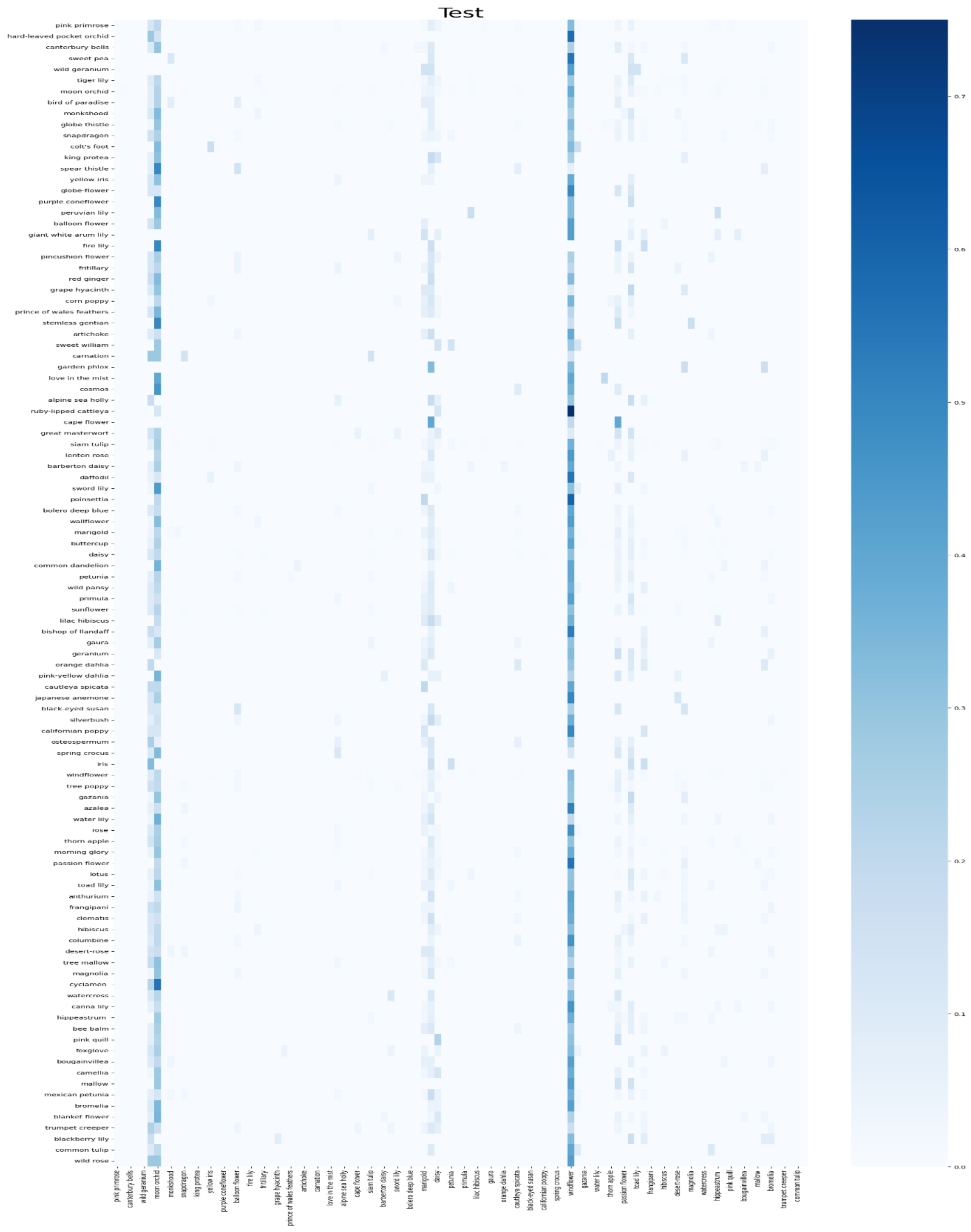
VGG16

The same as CNN but it takes the predicted classes generated by VGG16



ResNet50

The same as CNN but it takes the predicted classes generated by ResNet50

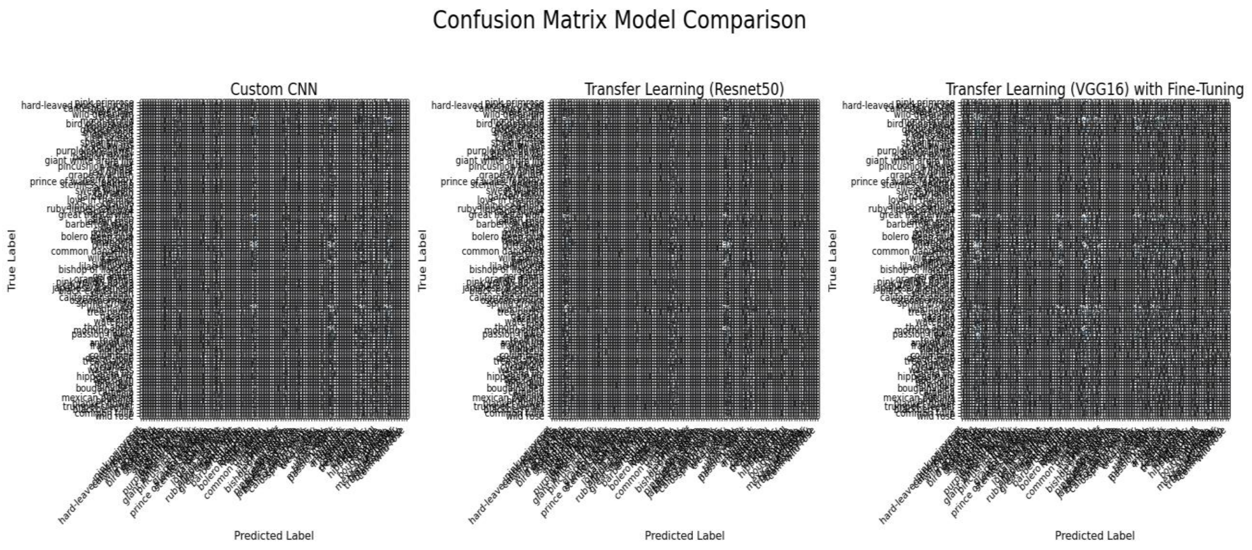


Confusion Matrix

Define a function called `plot_heatmap`. The function takes four arguments: `y_true`, `y_pred`, `class_names`, and `ax`. The purpose of this function is to plot a heatmap that shows the confusion matrix for the true and predicted labels of a classification model.

The function uses the `confusion_matrix` function from the Scikit-learn library to calculate the confusion matrix for the true and predicted labels. It then uses the Seaborn library to plot the heatmap using the `sns.heatmap` function. The heatmap shows the number of true positives, false positives, true negatives, and false negatives for each class in the classification model.

The code then creates three subplots using the Matplotlib library and calls the `plot_heatmap` function for each of them. The subplots show the confusion matrix for three different models: a custom CNN, transfer learning with VGG16 and fine-tuning, and transfer learning with Resnet50.v\



Ensemble (VGG16+ResNet50)

Build an ensemble of models and training them using image data augmentation.

`ensemble=[]`: This creates an empty list called `ensemble` which will be used to store the models. Then appends a model called `vgg_model_ft` to the `ensemble` list and appends another model called `dnn_model` to the `ensemble` list.

Calculates the number of steps needed to iterate through the training data generator (`traingen`) with a batch size of 64. It divides the total number of samples in the training set by 64 and discards the remainder.

Calculates the number of steps needed to iterate through the validation data generator (`valgen`) with a batch size of 64. Then sets the number of epochs for training.

Creates an instance of `ImageDataGenerator` with various augmentation parameters such as rotation, zooming, and shifting.

Creates an empty list called `models` which will be used to store the trained models.

For loop iterates over the models in the `ensemble` list: trains the `i`-th model in the ensemble. It uses the `fit_generator` function to train the model using data generated by the `datagen` generator. It specifies the number of epochs, validation data, steps per epoch, validation steps, and a callback for reducing learning rate on plateaus.

`models.append(model[i])`: After training each model, it appends the trained model to the `models` list.

This part essentially trains each model in the `ensemble` list using data augmentation techniques provided by the `ImageDataGenerator`. The trained models are stored in the `models` list for later use.