



CDAC CHENNAI

Tidel Park, 8th Floor,
'D' Block (North & South),
No.4 Rajiv Gandhi Salai, Tiramani,
Chennai- 600113, Tamilnadu (India)

Project Report

On

**Weather Image Classification Using Deep
Learning**

Submitted By

Shivraj Dinkar Shinde (220960825043)

Suhas Suryakant Yadav (220960825044)

Project Guide -: Mr. P. Prasanna

In The Partial Fulfilment of
Post Graduate Diploma in Big Data Analysis
September 2022 – March 2023

TABLE OF CONTENTS

Chapter 1: Introduction

- 1.1 About the Project
- 1.2 Project Objective
- 1.3 Problem Statement
- 1.4 Scope of the project
- 1.5 Limitations of the project

Chapter 2: Literature Survey

Chapter 3: Feature Engineering

- 3.1 Data Collection
- 3.2 Data Preprocessing
- 3.3 Exploratory Data Analysis (EDA)

Chapter 4: Development and Coding

- 4.1 Technology Used
- 4.2. Architecture and Training of Models

Chapter 5: Visualization and Testing

- 5.1 Comparison of Models through Visualization
- 5.2 User Interface and Testing

Chapter 6: Conclusion

Chapter 1: Introduction

1.1) About the Project:

The project is about creating trained CNN models from the given data and use them to classify weather image given by the user in one out of 11 different weather conditions.

1.2) Project Objective:

The main objective of the project is to classify the given images according to its condition. We will first create pre-trained all four CNN models which we are going to use to classify images. We are using four popular CNN models ResNet50, MobileNet, VGG16, DenseNet201. We also comparing these CNN models with accuracy and losses they produce.

1.3) Problem Statement:

Weather image classification using Conventional Neural Network Models.

1.4) Scope of the Project:

Image Processing extracts information from images and integrates it for several applications. There are several fields in which image processing applications is used. In our project, we are classifying weather images for weather condition monitoring, road condition monitoring, transportation, Automation in agriculture and forestry management, and the detection of the natural environment and keeping records in time to time manner.

1.5) Limitations of the Project:

Large number of images required to train the model to get the better classification accuracy. More images Better the Accuracy, Less Losses.

Chapter 2: Literature Survey

1. Weather Classification with Deep Convolutional Neural Networks: Mohamed Elhoseiny, Sheng Huang, Ahmed Elgammal
(https://www.researchgate.net/publication/280218749_Weather_classification_with_deep_convolutional_neural_networks).

In this work, they analyzed the recognition performance for the layers of both pretrained ImageNet-CNN and Weather-trained CNN. They also studied the performance drop under spatial distortion for the layers. They concluded our work by Weather classification results outperforming the state-of-the-art by a huge margin (82.2% compared with 53.1%).

2. Classification of Weather Phenomenon from Images by Using Deep Convolutional Neural Network: Haixia Xiao, Feng Zhang, Zhongping Shen, Kun Wu, Jinglin Zhang
(<https://agupubs.onlinelibrary.wiley.com/doi/full/10.1029/2020EA001604>)

In this paper, they have established a new representative database of weather phenomena images under the meteorological criterion. This database contains 6,877 images with 11 weather phenomena, which has more types than the previous database and can provide a research basis for future weather publicity research. Meanwhile, they proposed a weather phenomenon classification model, MeteCNN, which is a deep CNN model. The MeteCNN model can learn the features of weather phenomena well. Extensive experiments have shown that the proposed MeteCNN model is effective for weather phenomena classification and can avoid the mistakes caused by subjective error, making it superior to traditional methods. However, the MeteCNN model confuses some categories of weather phenomena, which may be due to the similarity and complexity of the images. Overall, the classification accuracy of

the MeteCNN model is as high as 92.68%, and the proposed model has a competitive classification performance among some mainstream models (e.g., Vgg16, Resnet34, Efficientnet-B7) on our data set. Therefore, the proposed model can be widely applied to the daily observation of weather phenomenon images and also can provide weather guidance for environmental monitoring, agriculture, and transportation, especially pertaining to weather change and forecasting.

3. Multi-Class Weather Classification Using ResNet-18 CNN for Autonomous IoT and CPS Applications: Qasem-Abu-Al-Haija, Mahmoud A. Smadi, Saleh Zein-Sabato.

<https://american-cse.org/sites/csci2020proc/pdfs/CSCI2020-6SccvdzjqC7bKupZxFmCoA/762400b586/762400b586.pdf>

A reliable auto-recognition deep-learning model to classify the weather condition images with high-level of classification accuracy, precision, and recall. To enhance the performance of feature extraction and learning, we have utilized the power of transfer learning technique with finetuning of the recognized deep ResNet-18 CNN pretrained on ImageNet dataset. The developed model uses the multi-class weather recognition dataset with 75% of the images used for training and 25% used for testing. Actually, the proposed work provides an inclusive framework model for multi-class image classification applications from input layer to the output layer. Finally, based on the comparison with other related research in the field, the obtained results outperform the results of existing automated classification models for weather conditions images.

Chapter 3: Feature Engineering

Feature engineering is a machine learning technique that leverages data to create new variables that are not in the training set. It can produce new features for both supervised and unsupervised learning, with the goal of simplifying and speeding up data transformations while also enhancing model accuracy. Feature engineering is required when working with machine learning models. Regardless of the data or architecture, a terrible feature will have a direct impact on your model.

3.1) Data Collection:

This dataset contains 6862 images of different types of weather, it can be used to implement weather classification based on the photo. The pictures are divided into 11 classes: dew, fog/smog, frost, glaze, hail, lightning, rain, rainbow, rime, sandstorm and snow.



Fig. 1 – Dataset

3.2) Data Pre-processing:

Data pre-processing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model.

When creating a machine learning project, it is not always a case that we come across the clean and formatted data. And while doing any operation with data, it is mandatory to clean it and put in a formatted way. So, for this, we use data pre-processing task.

A real-world data generally contains noises, missing values, and maybe in an unusable format which cannot be directly used for machine learning models. Data pre-processing is required tasks for cleaning the data and making it suitable for a machine learning model which also increases the accuracy and efficiency of a machine learning model.

It involves below steps:

- Getting the dataset
- Importing libraries
- Importing datasets
- Finding Missing Data
- Encoding Categorical Data
- Splitting dataset into training and test set
- Feature scaling

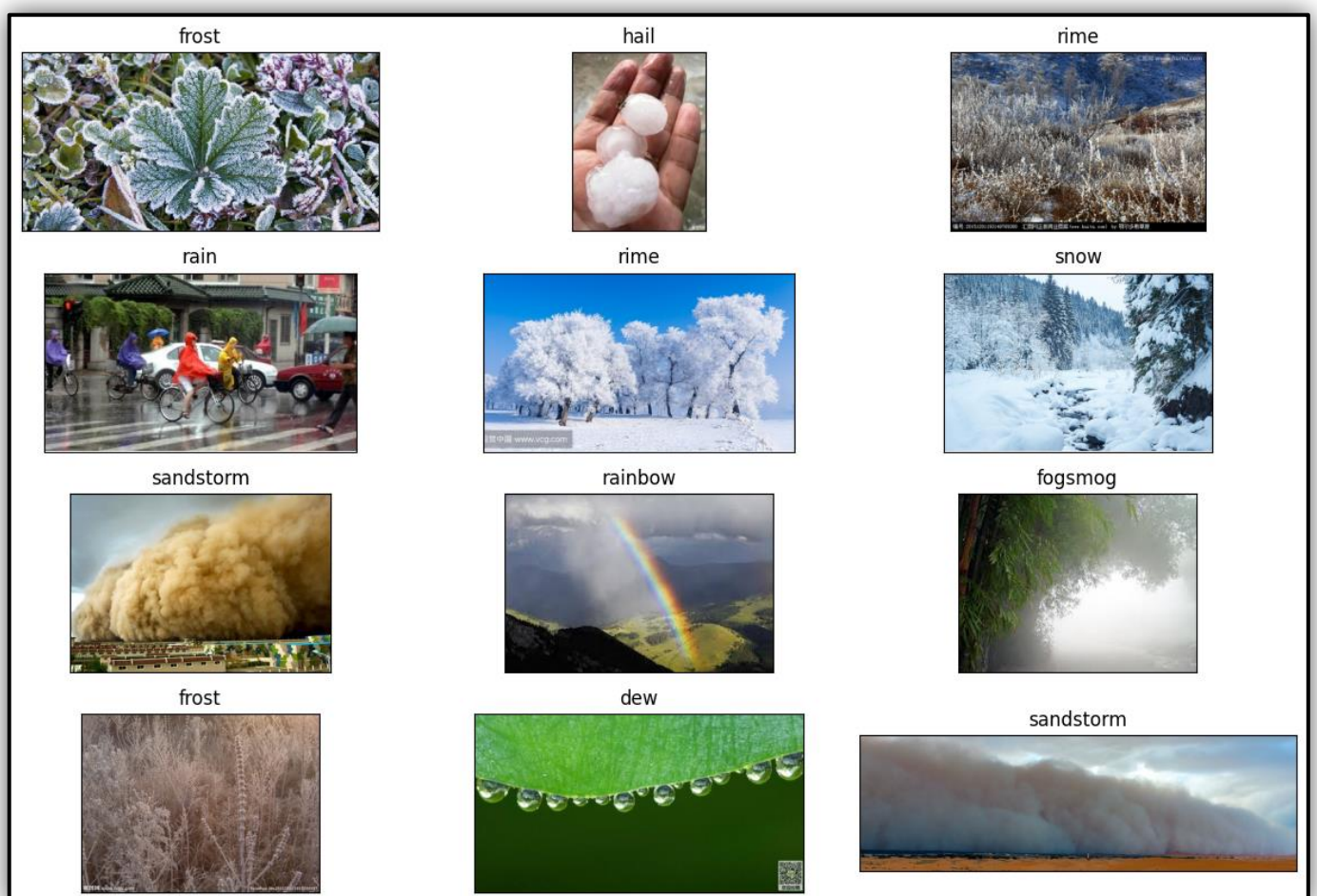
3.3) Exploratory Data Analysis (EDA):

Basically, Exploratory Data Analysis involves generating overall statistics for given data in the dataset and creating various graphical representations to visualize and understand the data in better way. In our project, by doing EDA we have learned that it helps us to determine how best to manipulate data sources to get the answer you need.

To extract some useful information, we have built some plots that are as follows:

1)

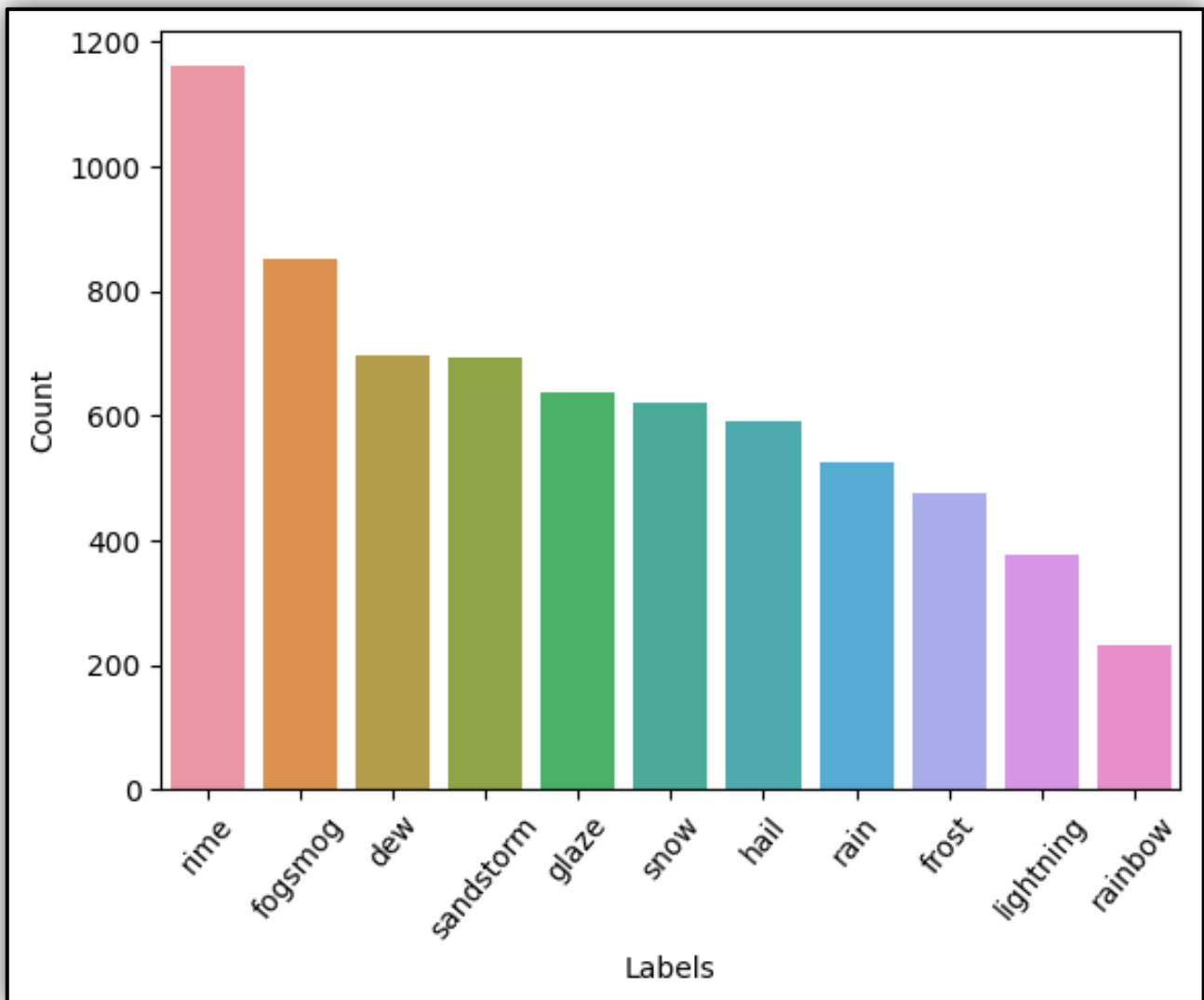
```
fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(12, 8),
                        subplot_kw={'xticks': [], 'yticks': []})
for i, ax in enumerate(axes.flat):
    ax.imshow(plt.imread(data.File_Path[i]))
    ax.set_title(data.Labels[i])
plt.tight_layout()
plt.show()
```



Explanation: From the above image we can analyze that there is total 12 different image which contains several types example: dew, snow, frost, hail etc.

2)

```
counts = data.Labels.value_counts()  
sns.barplot(x=counts.index, y=counts)  
plt.xlabel('Labels')  
plt.ylabel('Count')  
plt.xticks(rotation=50);
```



Explanation: In our dataset there are total 11 weather images type and in above image, we have use bar plot to show total number of images in our dataset contains.

Chapter 4: Development and Coding

4.1) Technology Used:

In our project we have created an individual python file, which contains various functions such as data augmentations, functions required for model compilation, plotting function and test result function which returns the test loss and accuracy of each model. We have imported required pre-trained models and libraries according to requirements of the project.

1) Importing Libraries –

In machine learning libraries plays an important role, similarly in our project we have added some important libraries which are as follows:

```
import os
import glob
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import Callback, EarlyStopping
from sklearn.metrics import confusion_matrix, classification_report
```

This code imports necessary libraries and modules, including:

- **os**: a module that provides a portable way of using operating system dependent functionality like reading or writing to the file system.
- **glob**: a module that finds all the pathnames matching a specified pattern according to the rules used by the Unix shell.
- **numpy** (as **np**): a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- **pandas** (as **pd**): a library for the Python programming language,

providing tools for data manipulation and analysis.

- **seaborn**: a visualization library based on Matplotlib, providing a high-level interface for creating informative and attractive statistical graphics.
- **matplotlib.pyplot** (as **plt**): a plotting library for the Python programming language, providing a wide variety of visualization tools and techniques.
- **train_test_split** from **sklearn.model_selection**: a function to split arrays or matrices into random train and test subsets.
- **ImageDataGenerator** from **tensorflow.keras.preprocessing.image**: a class for real-time data augmentation on image data.
- **Dense** from **tensorflow.keras.layers**: a class representing a densely connected neural network layer.
- **Model** from **tensorflow.keras.models**: a class for constructing deep learning models.
- **Callback** from **tensorflow.keras.callbacks**: a base class for defining custom callbacks.
- **EarlyStopping** from **tensorflow.keras.callbacks**: a callback that stops training when a monitored quantity has stopped improving.
- **confusion_matrix** from **sklearn.metrics**: a function that computes the confusion matrix from ground truth (correct) target values and predicted targets.
- **classification_report** from **sklearn.metrics**: a function that builds a text report showing the main classification metrics.

2) Data Augmentation –

Data augmentation helps to reduce overfitting when training a deep neural network. In this technique we are creating variation of the images which improve the ability of the fit models to generalize what they have learned to new images. Following are the code that we are Using in our project for data augmentation:

```
def augment(pre,train,test):  
    train_augment = ImageDataGenerator(preprocessing_function=pre, validation_split=0.2)  
    test_augment = ImageDataGenerator(preprocessing_function=pre)
```

This is a function called **augment** that takes in three parameters:

- **pre**: a preprocessing function to be applied to the image data.
- **train**: a training dataset.
- **test**: a testing dataset.

The function uses the **ImageDataGenerator** class from the **tensorflow.keras.preprocessing.image** module to perform real-time data augmentation on the training and testing datasets.

The **train_augment** object is initialized with a **validation_split** of 0.2, which means that 20% of the training data will be used for validation during training.

The **preprocessing_function** parameter is set to the **pre** function passed as an argument to the **augment** function. This function is applied to each image in the dataset before it is passed through the network.

The **test_augment** object is also initialized with the **preprocessing_function** parameter set to the **pre** function, but it does not use any validation split.

```

train_gen = train_augment.flow_from_dataframe(
    dataframe=train,
    x_col='File_Path',
    y_col='Labels',
    target_size=(224,224),
    class_mode='categorical',
    batch_size=64,
    shuffle=True,
    seed=0,
    subset='training',
    rotation_range=30,
    zoom_range=0.15,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.15,
    horizontal_flip=True,
    fill_mode="nearest")

```

This code creates a generator object called **train_gen** for the training dataset, using the **flow_from_dataframe** method of the **train_augment** object created earlier.

The **flow_from_dataframe** method takes several arguments:

- **dataframe**: the training dataset as a Pandas DataFrame.
- **x_col**: the column name of the file paths for the images in the DataFrame.
- **y_col**: the column name of the labels for the images in the DataFrame.
- **target_size**: the size to which the images are resized.
- **class_mode**: the type of classification problem, set to 'categorical' for multiclass classification.
- **batch_size**: the number of images per batch.
- **shuffle**: whether to shuffle the order of the images.
- **seed**: the random seed used for shuffling the images.
- **subset**: whether the generator should be used for training or validation. In this case, it is set to 'training'.
- **rotation_range**: the range of rotation (in degrees) for random rotations of the images.
- **zoom_range**: the range of zoom for random zooming of the images.

- **width_shift_range**: the range of horizontal shift for random horizontal shifting of the images.
- **height_shift_range**: the range of vertical shift for random vertical shifting of the images.
- **shear_range**: the range of shear for random shearing of the images.
- **horizontal_flip**: whether to randomly flip the images horizontally.
- **fill_mode**: the method used for filling in newly created pixels during transformations.

The **flow_from_dataframe** method creates a generator that yields batches of augmented image data and their corresponding labels during training.

```
valid_gen = train_augment.flow_from_dataframe(
    dataframe=train,
    x_col='File_Path',
    y_col='Labels',
    target_size=(224,224),
    class_mode='categorical',
    batch_size=32,
    shuffle=False,
    seed=0,
    subset='validation',
    rotation_range=30,
    zoom_range=0.15,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.15,
    horizontal_flip=True,
    fill_mode="nearest")
```

This code creates another generator object called **valid_gen** for the validation dataset, also using the **flow_from_dataframe** method of the **train_augment** object.

The arguments for **flow_from_dataframe** are similar to those for **train_gen**, with a few exceptions:

- **subset**: set to 'validation' to use the validation split defined earlier in **train_augment**.
- **shuffle**: set to False to prevent shuffling the images in the validation set.

- **batch_size**: set to 32 for smaller batch sizes during validation.
- Like **train_gen**, **valid_gen** yields batches of augmented image data and their corresponding labels during training and validation, respectively.

```
test_gen = test_augment.flow_from_dataframe(
    dataframe=test,
    x_col='File_Path',
    y_col='Labels',
    target_size=(224,224),
    color_mode='rgb',
    seed=0,
    class_mode='categorical',
    batch_size=64,
    verbose=0,
    shuffle=False)
return train_gen, valid_gen, test_gen
```

This code creates a generator object called **test_gen** for the testing dataset, using the **flow_from_dataframe** method of the **test_augment** object created earlier.

The arguments for **flow_from_dataframe** are similar to those for **train_gen** and **valid_gen**, with a few exceptions:

- **color_mode**: set to 'rgb' to indicate that the images are in RGB colour format.
- **shuffle**: set to False to prevent shuffling the images in the testing set.
- **verbose**: set to 0 to disable printing of progress messages during testing.

The **test_gen** generator yields batches of augmented image data and their corresponding labels during testing. The final line of the function returns the three generator objects: **train_gen**, **valid_gen**, and **test_gen**.

3) Run Model –

```
def run_model(mod_name):
    pre_model = mod_name(input_shape=(224,224, 3),
                          include_top=False,
                          weights='imagenet',
                          pooling='avg')

    pre_model.trainable = False
    inputs = pre_model.input
    x = Dense(100, activation='relu')(pre_model.output)
    x = Dense(100, activation='relu')(x)
    outputs = Dense(11, activation='softmax')(x)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(loss = 'categorical_crossentropy', optimizer=tensorflow.optimizers.Adam(5e-5), metrics=['accuracy'])
    early_stop = [EarlyStopping(monitor='val_loss',
                                min_delta=0,
                                patience=3,
                                mode='min')]

    return model, early_stop
#https://keras.io/api/callbacks/early_stopping/
```

This line defines a function called **run_model** that takes in a single argument **mod_name**. This creates a pre-trained model by calling **mod_name** with the following arguments:

- **input_shape=(224,224,3)**: This sets the shape of the input tensor to be 224x224 pixels with 3 colour channels (RGB).
- **include_top=False**: This removes the top layer of the pre-trained model, which is typically a fully connected layer used for classification. We will add our own classification layer later.
- **weights='imagenet'**: This specifies that we want to use the pre-trained weights from the ImageNet dataset.
- **pooling='avg'**: This sets the final pooling layer of the pre-trained model to global average pooling.

freezes the weights of the pre-trained model so that they are not updated during training. We will only train the weights of the classification layer that we add later.

This adds our own classification layer on top of the pre-trained model. We first define the input tensor (**inputs**) to be the same as the input tensor of the pre-trained model. Then we add two fully connected layers with 100 nodes each and ReLU activation functions (**Dense (100, activation='relu')**). The first layer takes the output of the pre-trained model (**pre_model.output**) as input (**(pre_model.output)**). The output of the first layer (**x**) is then fed as input to the second layer. Finally, we

add an output layer with 11 nodes (one for each class in our dataset) and a softmax activation function.

Creates the final model by defining the inputs and outputs. compiles the model with the categorical cross-entropy loss function, the Adam optimizer with a learning rate of 5e-5, and accuracy as the evaluation metric.

Sets up early stopping to monitor validation loss (**monitor='val_loss'**), stop training if there is no improvement (**min_delta=0**) for 3 consecutive epochs (**patience=3**), and use the minimum validation loss as the stopping criteria (**mode='min'**). Returns the **model** and **early_stop** objects as a tuple.

Overall, this function creates a new model by adding a few fully connected layers on top of a pre-trained model, and sets up the model for training with the Adam optimizer and early stopping. The function returns the model and early stopping objects so that we can use them for training.

4) Plotting Function –

We have implemented plotting function to plot the graph of val_accuracy and val_loss, conf_matrix, classification report and we get the prediction of actual vs predicted.

```
def plotting(history, test_gen, train_gen, model, testLabel, testFilePath):
    # Plotting Accuracy, val_accuracy, loss, val_loss
    fig, ax = plt.subplots(1, 2, figsize=(10, 3))
    ax = ax.ravel()

    for i, parameter in enumerate(['accuracy', 'loss']):
        ax[i].plot(history.history[parameter])
        ax[i].plot(history.history['val_' + parameter])
        ax[i].set_title(f'Model {parameter}')
        ax[i].set_xlabel('epochs')
        ax[i].set_ylabel(parameter)
        ax[i].legend(['Train', 'Validation'])

    # Predict Data Test
    pred = model.predict(test_gen)
    pred = np.argmax(pred, axis=1)
    labels = (train_gen.class_indices)
    labels = dict((v, k) for k, v in labels.items())
    pred = [labels[k] for k in pred]
```

The **ploting** function is responsible for plotting the accuracy, loss, validation accuracy, and validation loss of the trained model during training. It takes in several parameters as inputs:

- **history**: The history object returned by the **fit** method of the Keras model. It contains information about the training process, such as the loss and accuracy values for each epoch.
- **test_gen**: The test data generator object that was created using **ImageDataGenerator.flow_from_dataframe** method.
- **train_gen**: The train data generator object that was created using **ImageDataGenerator.flow_from_dataframe** method.
- **model**: The trained Keras model that is being evaluated.
- **testLabel**: The true labels of the test set.
- **testFilePath**: The file paths of the images in the test set.

The function first creates a figure object with two subplots, one for the accuracy and validation accuracy, and another for the loss and validation loss. It then loops over the list **['accuracy', 'loss']** to plot the training accuracy/loss and validation accuracy/loss for each epoch.

After plotting the training metrics, the function uses the trained model to predict the labels of the test set using the **predict** method of the Keras model. It then converts the predicted labels from one-hot encoding to their corresponding class names using the **train_gen.class_indices** dictionary. Finally, it plots a confusion matrix of the true labels versus the predicted labels using the **confusion_matrix** function from **scikit-learn**.

```

# Classification report
cm=confusion_matrix(testLabel,pred)
# Creating a dataframe for a array-formatted Confusion matrix,so it will be easy for plotting.
cm_df = pd.DataFrame(cm,
                      index = ['dew', 'fogsmog','frost','glaze','hail','lightning','rain','rainbow','rime','sandstorm','snow'],
                      columns = ['dew', 'fogsmog','frost','glaze','hail','lightning','rain','rainbow','rime','sandstorm','snow'])

#Plotting the confusion matrix
plt.figure(figsize=(20,15))
sns.heatmap(cm_df, annot=True)
plt.title('Confusion Matrix')
plt.ylabel('Actual Values')
plt.xlabel('Predicted Values')
plt.show()
clr = classification_report(testLabel, pred)
#print(cm)
print(clr)
# Display 3 picture of the dataset with their labels
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(12, 8),
                        subplot_kw={'xticks': [], 'yticks': []})

for i, ax in enumerate(axes.flat):
    ax.imshow(plt.imread(testFilePath.iloc[i+1]))
    ax.set_title(f'Actual: {testLabel.iloc[i+1]}\nPredicted: {pred[i+1]}')
plt.tight_layout()
plt.show()

return history

```

The function then creates a confusion matrix using the **confusion_matrix ()** function from scikit-learn. It also creates a heatmap of the confusion matrix using seaborn's **heatmap ()** function. The confusion matrix is a table that shows how many samples were classified correctly and incorrectly for each class. The rows represent the true labels and the columns represent the predicted labels. The diagonal elements of the matrix represent the number of correctly classified samples for each class, while the off-diagonal elements represent the number of misclassified samples. The heatmap makes it easier to visualize the matrix by assigning different colors to different values.

The function then creates a classification report using scikit-learn's **classification_report ()** function. The report shows several metrics such as precision, recall, and f1-score for each class. These metrics are commonly used to evaluate the performance of a multi-class classification model.

Finally, the function displays three random images from the test set along with their true and predicted labels using matplotlib's **imshow ()** function. The images are displayed in a row of three subplots.

The function returns the training history as output.

5) Result Test Function –

This `result_test` function gives us the value of `Test_loss` and `Test_accuracy`.

```
def result_test(test,trained_model):  
    results = trained_model.evaluate(test, verbose=1)  
  
    print("    Test Loss: {:.5f}".format(results[0]))  
    print("Test Accuracy: {:.2f}%".format(results[1] * 100))  
  
    return results
```

This function takes in the **test** set and a **trained_model** as inputs and returns the evaluation results of the model on the test set.

First, it calls the **evaluate ()** method of the model with the **test** set as input and stores the results in the **results** variable. The **verbose** parameter is set to 1, which means progress bar will be displayed during evaluation.

Then, it prints the test loss and test accuracy, formatted to display the results with 5 decimal places for loss and 2 decimal places for accuracy. The test accuracy is multiplied by 100 to display the percentage value. Finally, the function returns the **results** variable.

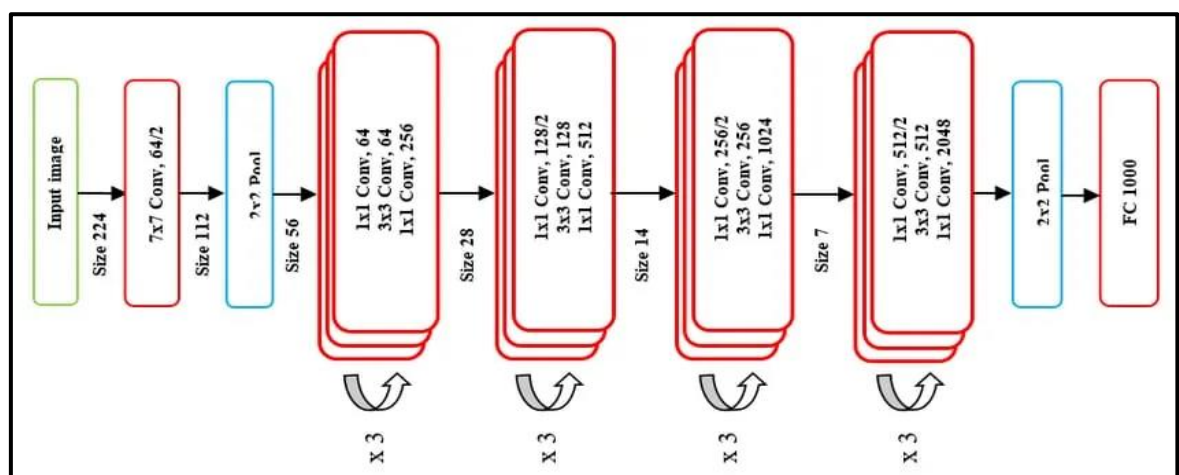
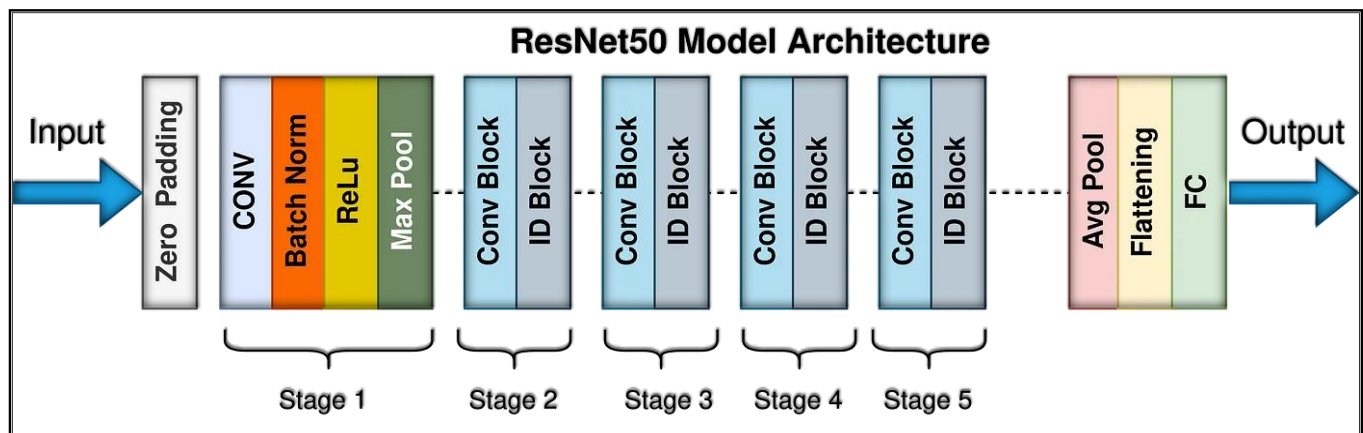
4.2) Architecture and Training of Models:

1) ResNet50 –

ResNet-50 is a convolutional neural network that is 50 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network

has an image input size of 224-by-224.

<https://www.mathworks.com/help/deeplearning/ref/resnet50.html>



The 50-layer ResNet architecture includes the following elements, as shown in the table below:

- **A 7×7 kernel convolution** alongside 64 other kernels with a 2-sized stride.
- **A max pooling layer** with a 2-sized stride.
- **9 more layers**—3×3,64 kernel convolution, another with 1×1,64 kernels, and a third with 1×1,256 kernels. These 3 layers are repeated 3 times.
- **12 more layers** with 1×1,128 kernels, 3×3,128 kernels, and 1×1,512 kernels, iterated 4 times.

- **18 more layers** with $1 \times 1, 256$ cores, and 2 cores $3 \times 3, 256$ and $1 \times 1, 1024$, iterated 6 times.
- **9 more layers** with $1 \times 1, 512$ cores, $3 \times 3, 512$ cores, and $1 \times 1, 2048$ cores iterated 3 times.

(Up to this point the network has 50 layers)

- **Average pooling**, followed by a fully connected layer with 1000 nodes, using the softmax activation function.

<https://towardsdatascience.com/the-annotated-resnet-50-a6c536034758>

<https://datagen.tech/guides/computer-vision/resnet-50/>

```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
ResNet_pre_input=preprocess_input
train_gen_ResNet, valid_gen_ResNet, test_gen_ResNet = fp.augment(ResNet_pre_input,train_it,test_it)
ResNet_model, stop_early=fp.run_model(ResNet50)
history = ResNet_model.fit(
    train_gen_ResNet,
    validation_data=valid_gen_ResNet,
    epochs=15,
    callbacks=stop_early
)
history_ResNet= fp.plotting(history,test_gen_ResNet,train_gen_ResNet, ResNet_model,testLabels,testFilePath)
result_ResNet = fp.result_test(test_gen_ResNet,ResNet_model)
```

- The first two lines of code imports the ResNet50 model from the Keras library. ResNet50 is a pre-trained convolutional neural network that has been trained on millions of images from the ImageNet dataset. The second line imports the preprocess_input function from the ResNet50 module. This function is used to preprocess input images to the ResNet50 model.
- Next line of code creates a variable called ResNet_pre_input and assigns the preprocess_input function to it. This variable is later used as an argument in the augment function.
- Next line calls the augment function from a custom module called fp. This function takes in the ResNet_pre_input function, as well as the training and testing generators (train_it and test_it). The function returns three generators: one for the augmented training data, one for the augmented validation data, and one for the

augmented test data.

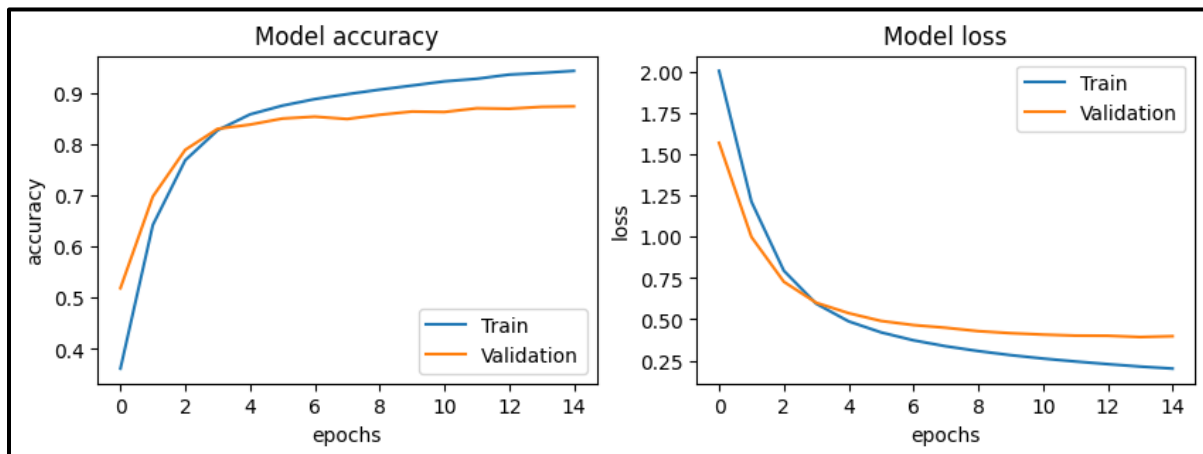
- After this we call the `run_model` function from the `fp` module. This function compiles the ResNet50 model and sets up an early stopping callback to prevent overfitting. It returns two values: the compiled ResNet50 model, and the early stopping callback.
- Code from next line fits the compiled ResNet50 model to the augmented training data and validates it with the augmented validation data for 15 epochs, using the early stopping callback to prevent overfitting. The fit function returns a history object that contains the training and validation loss and accuracy for each epoch.
- After fitting the module, we call the plotting function from the `fp` module. This function takes the history object, the augmented test data generator, the augmented training data generator, the ResNet50 model, as well as the test labels and file path as arguments. It plots the training and validation loss and accuracy curves, and evaluates the model performance on the test data.
- The last line of code calls the `result_test` function from the `fp` module. This function takes the augmented test data generator and the ResNet50 model as arguments, and evaluates the model performance on the test data. It returns the test loss, accuracy, precision, recall, and F1-score.

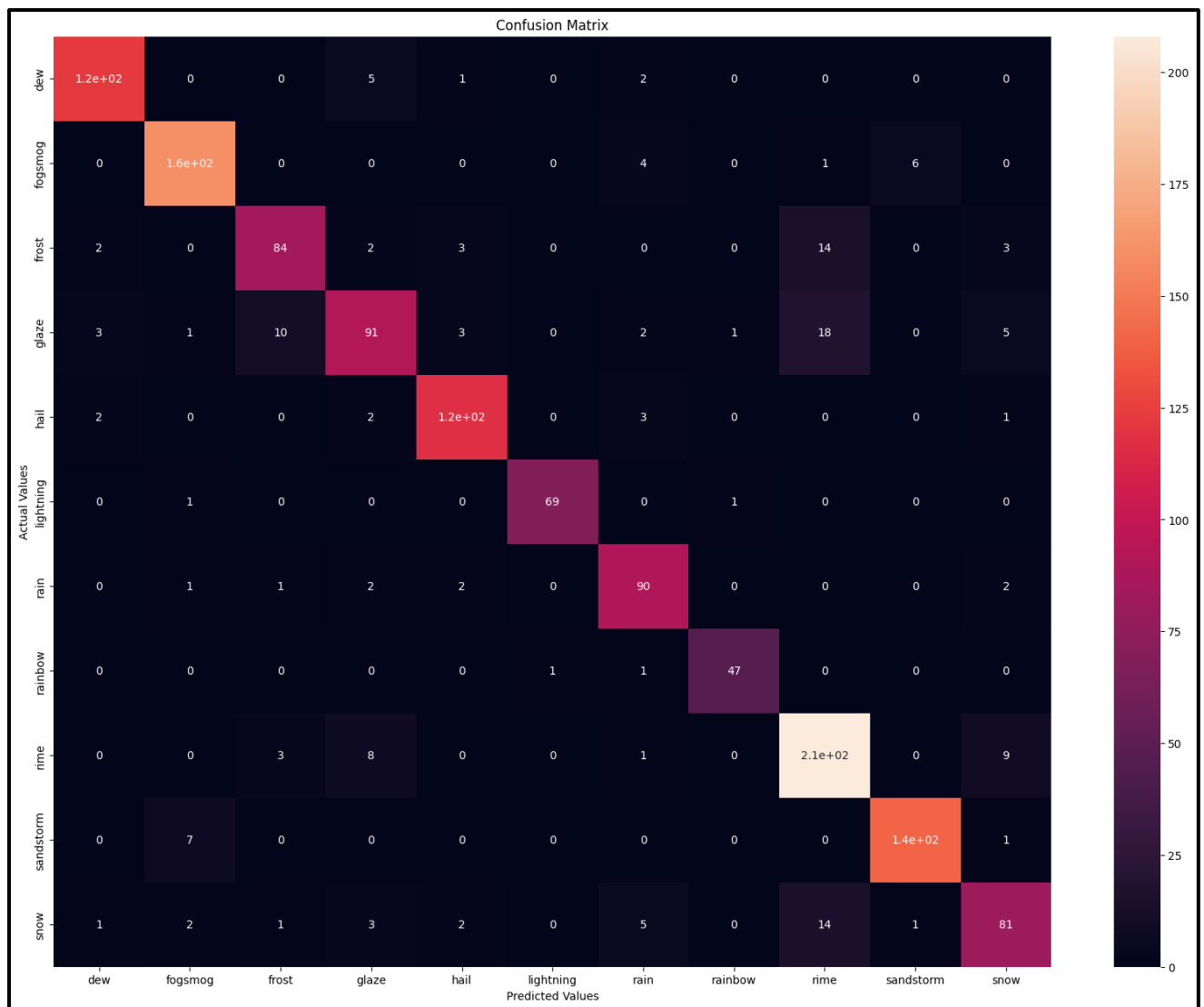
Output –

```

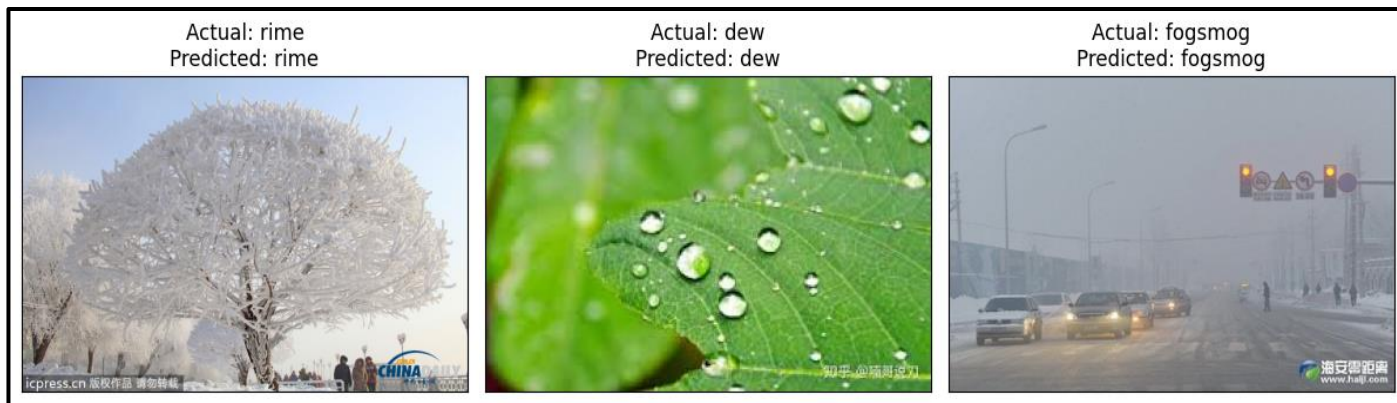
Found 4392 validated image filenames belonging to 11 classes.
Found 1097 validated image filenames belonging to 11 classes.
Found 1373 validated image filenames belonging to 11 classes.
Epoch 1/15
69/69 [=====] - 468s 7s/step - loss: 2.0073 - accuracy: 0.3618 - val_loss: 1.5705 - val_accuracy: 0.5187
Epoch 2/15
69/69 [=====] - 454s 7s/step - loss: 1.2137 - accuracy: 0.6418 - val_loss: 0.9995 - val_accuracy: 0.6974
Epoch 3/15
69/69 [=====] - 465s 7s/step - loss: 0.7950 - accuracy: 0.7682 - val_loss: 0.7281 - val_accuracy: 0.7885
Epoch 4/15
69/69 [=====] - 455s 7s/step - loss: 0.5948 - accuracy: 0.8267 - val_loss: 0.6007 - val_accuracy: 0.8295
Epoch 5/15
69/69 [=====] - 464s 7s/step - loss: 0.4878 - accuracy: 0.8577 - val_loss: 0.5376 - val_accuracy: 0.8377
Epoch 6/15
69/69 [=====] - 473s 7s/step - loss: 0.4210 - accuracy: 0.8748 - val_loss: 0.4903 - val_accuracy: 0.8496
Epoch 7/15
69/69 [=====] - 465s 7s/step - loss: 0.3728 - accuracy: 0.8875 - val_loss: 0.4653 - val_accuracy: 0.8532
Epoch 8/15
69/69 [=====] - 454s 7s/step - loss: 0.3368 - accuracy: 0.8971 - val_loss: 0.4489 - val_accuracy: 0.8487
Epoch 9/15
69/69 [=====] - 467s 7s/step - loss: 0.3076 - accuracy: 0.9060 - val_loss: 0.4284 - val_accuracy: 0.8569
Epoch 10/15
69/69 [=====] - 459s 7s/step - loss: 0.2829 - accuracy: 0.9139 - val_loss: 0.4162 - val_accuracy: 0.8633
Epoch 11/15
69/69 [=====] - 463s 7s/step - loss: 0.2622 - accuracy: 0.9221 - val_loss: 0.4075 - val_accuracy: 0.8624
...
69/69 [=====] - 456s 7s/step - loss: 0.2135 - accuracy: 0.9385 - val_loss: 0.3926 - val_accuracy: 0.8724
Epoch 15/15
69/69 [=====] - 464s 7s/step - loss: 0.2013 - accuracy: 0.9426 - val_loss: 0.3976 - val_accuracy: 0.8733
22/22 [=====] - 115s 5s/step

```





	precision	recall	f1-score	support
dew	0.94	0.94	0.94	130
fogsmog	0.93	0.94	0.93	171
frost	0.85	0.78	0.81	108
glaze	0.81	0.68	0.74	134
hail	0.91	0.94	0.92	125
lightning	0.99	0.97	0.98	71
rain	0.83	0.92	0.87	98
rainbow	0.96	0.96	0.96	49
rime	0.82	0.91	0.86	229
sandstorm	0.95	0.95	0.95	148
snow	0.79	0.74	0.76	110
accuracy			0.88	1373
macro avg	0.89	0.88	0.88	1373
weighted avg	0.88	0.88	0.88	1373



```
22/22 [=====] - 113s 5s/step - loss: 0.3674 - accuracy: 0.8806
Test Loss: 0.36739
Test Accuracy: 88.06%
```

```
ResNet_model.save('C://Users/MrBot/Documents/Weather-Image-Classification-using-Deep-Learning-main/Interfaces/model_ResNet_1.h5')
```

[21]

Python

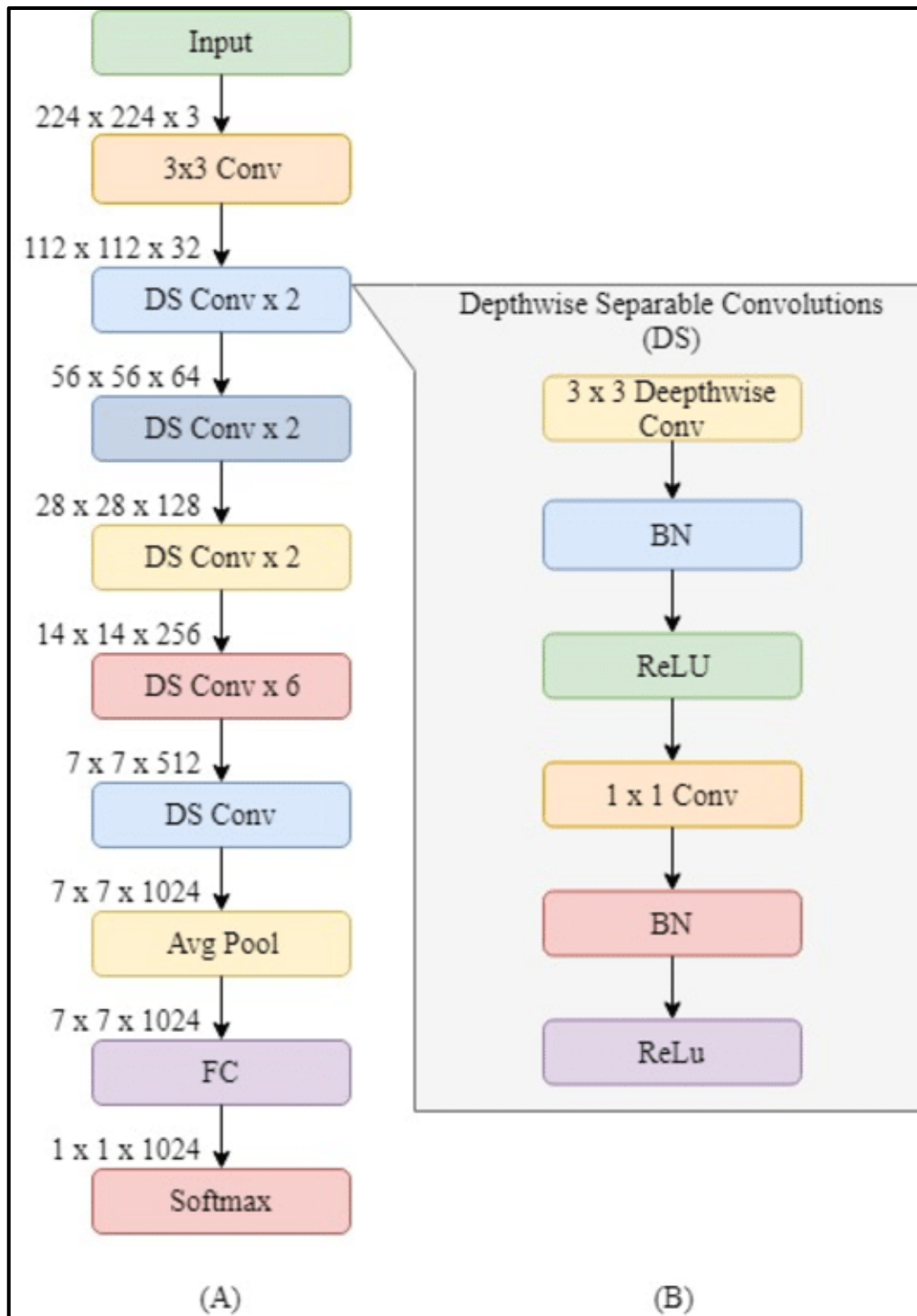
At the end we save the model in '.h5' file to use it in app to classify the images.

2) MobileNet –

MobileNet-v2 is a convolutional neural network that is 53 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224.

<https://www.mathworks.com/help/deeplearning/ref/mobilenetv2.html>

MobileNet is a streamlined architecture that uses depth wise separable convolutions to construct lightweight deep convolutional neural networks and provides an efficient model for mobile and embedded vision applications. The structure of MobileNet is based on depth wise separable filters, as shown in Fig.



Depthwise separable convolution filters are composed of depthwise convolution filters and point convolution filters. The depthwise convolution filter performs a single convolution on each input channel, and the point convolution filter combines the output of depthwise convolution linearly with 1×1 convolutions.

<https://wikidocs.net/165429>

<https://medium.com/@godeep48/an-overview-on-mobilenet-an-efficient-mobile-vision-cnn-f301141db94d>

```

from tensorflow.keras.applications import MobileNet
from tensorflow.keras.applications.mobilenet import preprocess_input
MobileNet_pre_input=preprocess_input
train_gen_MobileNet, valid_gen_MobileNet, test_gen_MobileNet = fp.augment(MobileNet_pre_input,train_it,test_it)
MobileNet_model, stop_early=fp.run_model(MobileNet)
history = MobileNet_model.fit(
    train_gen_MobileNet,
    validation_data=valid_gen_MobileNet,
    epochs=10,
    callbacks=stop_early,
    verbose=1
)
history_MobileNet = fp.plotting(history,test_gen_MobileNet,train_gen_MobileNet, MobileNet_model,testLabels,testFilePath)
result_MobileNet = fp.result_test(test_gen_MobileNet,MobileNet_model)

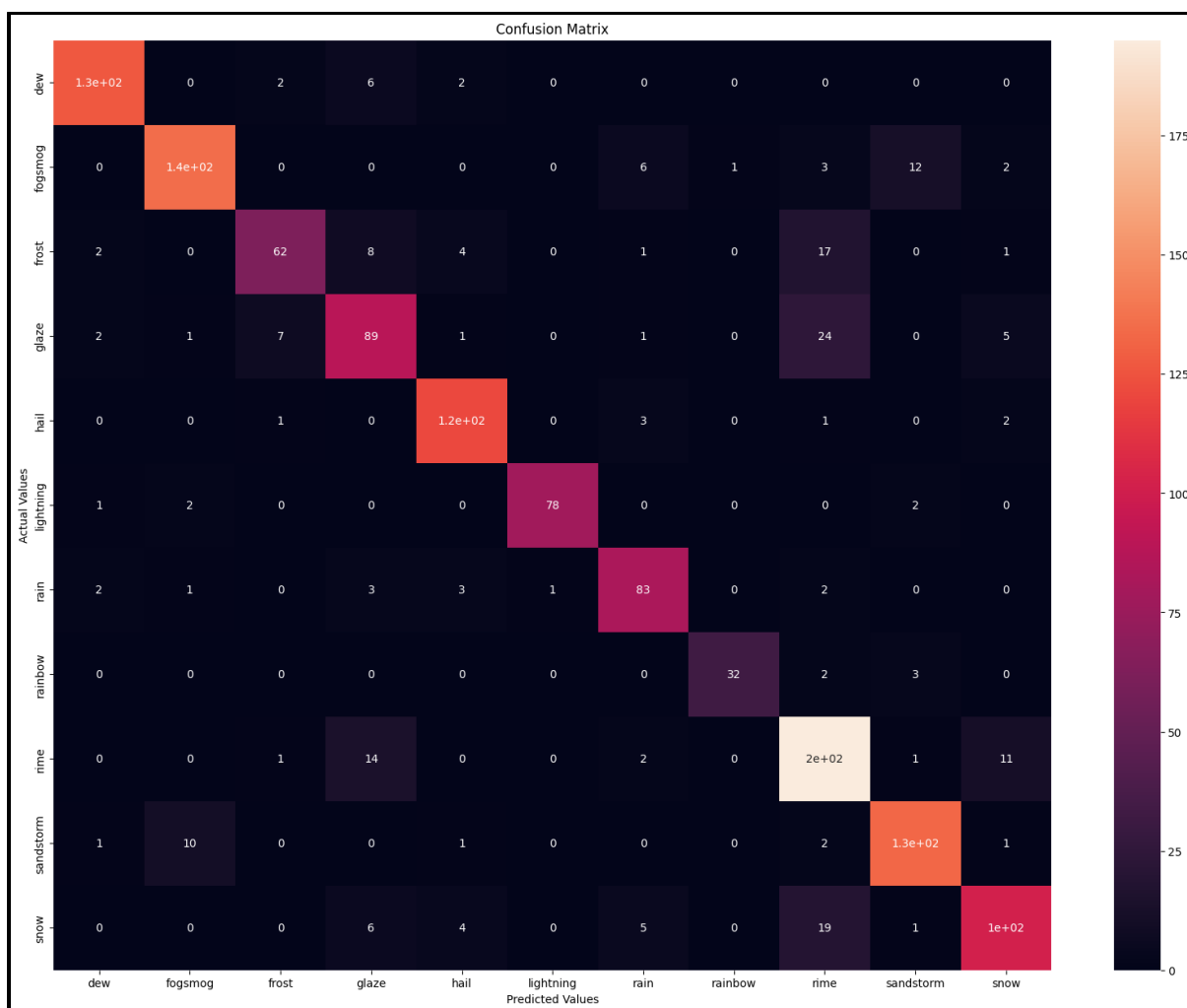
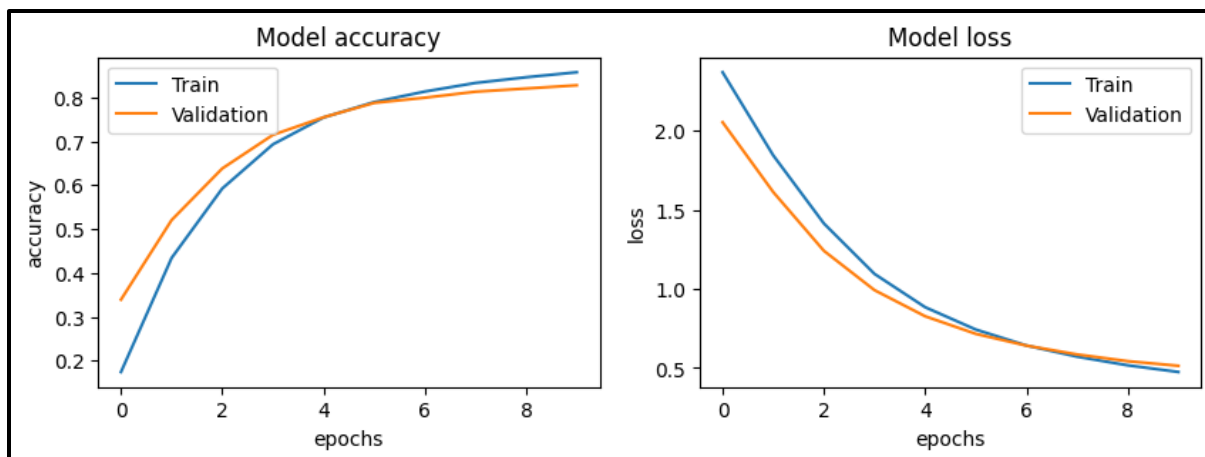
```

- First, we are importing the MobileNet pre-trained model from the TensorFlow Keras applications module. Next, we are importing the preprocess_input function from the MobileNet module.
- Creating a variable called MobileNet_pre_input that references the preprocess_input function. We are calling a function called augment from a module called fp.
- This function takes the MobileNet_pre_input function, a training data iterator called train_it, and a test data iterator called test_it as inputs, and generates augmented images for the training set using various data augmentation techniques such as random rotations, flips, and zooms. The augmented data is returned as three separate data generators: train_gen_MobileNet, valid_gen_MobileNet, and test_gen_MobileNet, which are used for training, validation, and testing, respectively.
- After that, we are defining two variables called MobileNet_model and stop_early using a function called run_model from the fp module.
- This function takes the MobileNet model as input and creates a new model for fine-tuning on the new dataset. Specifically, it freezes all the layers in the MobileNet model except for the last few layers, which are replaced by new layers that are trained on the new dataset. The stop_early variable is a callback function that is used to monitor the validation loss during training and stop

the training process early if the loss stops improving.

- Next, we are training the MobileNet_model on the augmented data using the fit function. This function trains the MobileNet_model on the augmented data for a fixed number of epochs (in this case, 10), and uses the validation set to monitor the model's performance and prevent overfitting. The stop_early callback is passed to the fit function to stop the training process early if the validation loss stops improving. The training history is stored in the history variable, which can be used later for plotting and evaluation.
- After training, we are using the plotting function from the fp module to plot the training and validation loss curves, and to evaluate the model's performance on the test set. This function takes the training history, the test data generator, the train data generator, the trained model, the test labels, and the test file path as inputs, and generates plots of the training and validation loss.

```
Found 4392 validated image filenames belonging to 11 classes.
Found 1097 validated image filenames belonging to 11 classes.
Found 1373 validated image filenames belonging to 11 classes.
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet/mobilenet\_1\_0\_224\_tf\_no\_top.h5
17225924/17225924 [=====] - 3s 0us/step
Epoch 1/10
69/69 [=====] - 122s 2s/step - loss: 2.3711 - accuracy: 0.1742 - val_loss: 2.0548 - val_accuracy: 0.3391
Epoch 2/10
69/69 [=====] - 161s 2s/step - loss: 1.8451 - accuracy: 0.4342 - val_loss: 1.6130 - val_accuracy: 0.5205
Epoch 3/10
69/69 [=====] - 151s 2s/step - loss: 1.4144 - accuracy: 0.5924 - val_loss: 1.2414 - val_accuracy: 0.6381
Epoch 4/10
69/69 [=====] - 151s 2s/step - loss: 1.0951 - accuracy: 0.6931 - val_loss: 0.9923 - val_accuracy: 0.7147
Epoch 5/10
69/69 [=====] - 159s 2s/step - loss: 0.8844 - accuracy: 0.7543 - val_loss: 0.8271 - val_accuracy: 0.7548
Epoch 6/10
69/69 [=====] - 135s 2s/step - loss: 0.7431 - accuracy: 0.7896 - val_loss: 0.7157 - val_accuracy: 0.7876
Epoch 7/10
69/69 [=====] - 138s 2s/step - loss: 0.6420 - accuracy: 0.8135 - val_loss: 0.6414 - val_accuracy: 0.7995
Epoch 8/10
69/69 [=====] - 122s 2s/step - loss: 0.5715 - accuracy: 0.8331 - val_loss: 0.5855 - val_accuracy: 0.8131
Epoch 9/10
69/69 [=====] - 120s 2s/step - loss: 0.5167 - accuracy: 0.8461 - val_loss: 0.5429 - val_accuracy: 0.8204
Epoch 10/10
69/69 [=====] - 121s 2s/step - loss: 0.4746 - accuracy: 0.8575 - val_loss: 0.5136 - val_accuracy: 0.8277
22/22 [=====] - 31s 1s/step
```

	precision	recall	f1-score	support
dew	0.94	0.93	0.93	137
fogsmog	0.91	0.85	0.88	159
frost	0.85	0.65	0.74	95
glaze	0.71	0.68	0.70	130
hail	0.89	0.95	0.92	128
lightning	0.99	0.94	0.96	83
rain	0.82	0.87	0.85	95
rainbow	0.97	0.86	0.91	37
rime	0.74	0.87	0.80	224
sandstorm	0.88	0.90	0.89	148
snow	0.82	0.74	0.78	137
accuracy			0.84	1373
macro avg	0.86	0.84	0.85	1373
weighted avg	0.85	0.84	0.84	1373



```
22/22 [=====] - 30s 1s/step - loss: 0.5256 - accuracy: 0.8427
Test Loss: 0.52561
Test Accuracy: 84.27%
```

```
MobileNet_model.save('C://Users/MrBot/Documents/Weather-Image-Classification-using-Deep-Learning-main/Interfaces/model_MobileNet_1.h5')
```

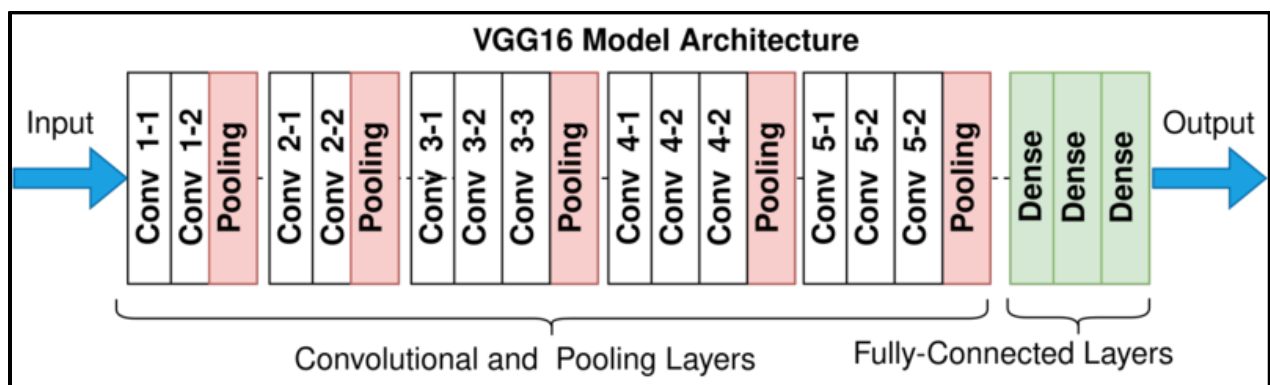
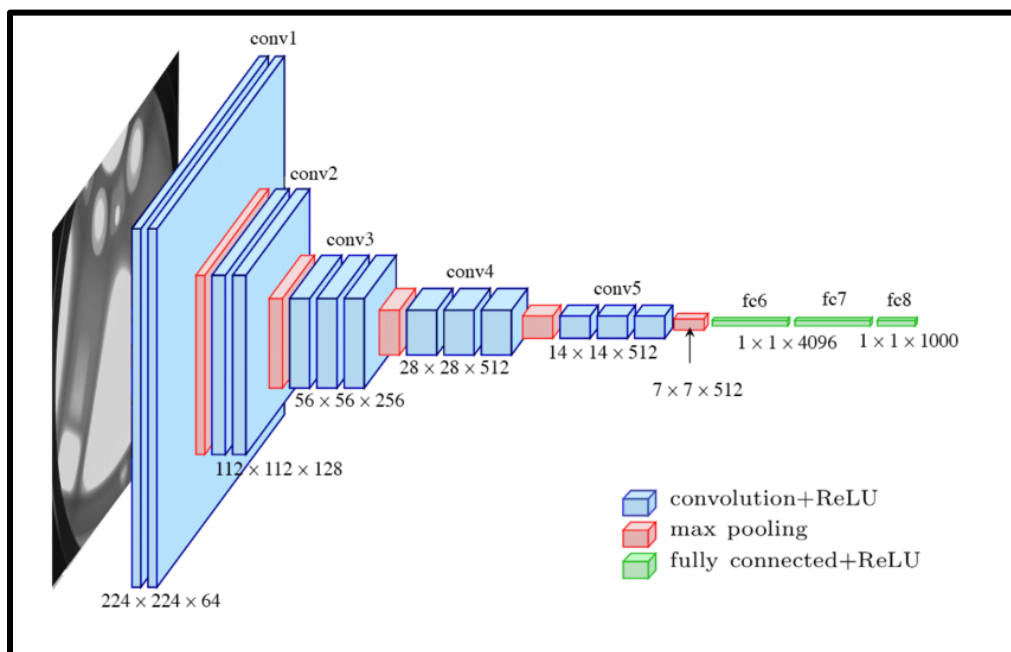
Python

At the end we save the model in '.h5' file to use it in app to classify the images.

3) VGG16 –

VGG-16 is a convolutional neural network that is 16 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224.

<https://www.mathworks.com/help/deeplearning/ref/vgg16.html>



A VGG network consists of small convolution filters. VGG16 has three fully connected layers and 13 convolutional layers.

Here is a quick outline of the VGG architecture:

- **Input**—VGGNet receives a 224×224 image input. In the ImageNet competition, the model's creators kept the image input size constant by cropping a 224×224 section from the centre of each image.
- **Convolutional layers**—the convolutional filters of VGG use the smallest possible receptive field of 3×3 . VGG also uses a 1×1 convolution filter as the input's linear transformation.
- **ReLU activation**—next is the Rectified Linear Unit Activation Function (ReLU) component, AlexNet's major innovation for reducing training time. ReLU is a linear function that provides a matching output for positive inputs and outputs zero for negative inputs. VGG has a set convolution stride of 1 pixel to preserve the spatial resolution after convolution (the stride value reflects how many pixels the filter “moves” to cover the entire space of the image).
- **Hidden layers**—all the VGG network's hidden layers use ReLU instead of Local Response Normalization like AlexNet. The latter increases training time and memory consumption with little improvement to overall accuracy.
- **Pooling layers**—A pooling layer follows several convolutional layers—this helps reduce the dimensionality and the number of parameters of the feature maps created by each convolution step. Pooling is crucial given the rapid growth of the number of available filters from 64 to 128, 256, and eventually 512 in the final layers.
- **Fully connected layers**—VGGNet includes three fully connected layers. The first two layers each have 4096 channels, and the third layer has 1000 channels, one for every class.

<https://datagen.tech/guides/computer-vision/vgg16/>

```

from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input

vgg_pre_input=preprocess_input
train_gen_VGG, valid_gen_VGG, test_gen_VGG = fp.augment(vgg_pre_input,train_it,test_it)
model_VGG16, stop_early=fp.run_model(VGG16)
history = model_VGG16.fit(
    train_gen_VGG,
    validation_data=valid_gen_VGG,
    epochs=10,
    callbacks=stop_early,
    verbose=1
)

```

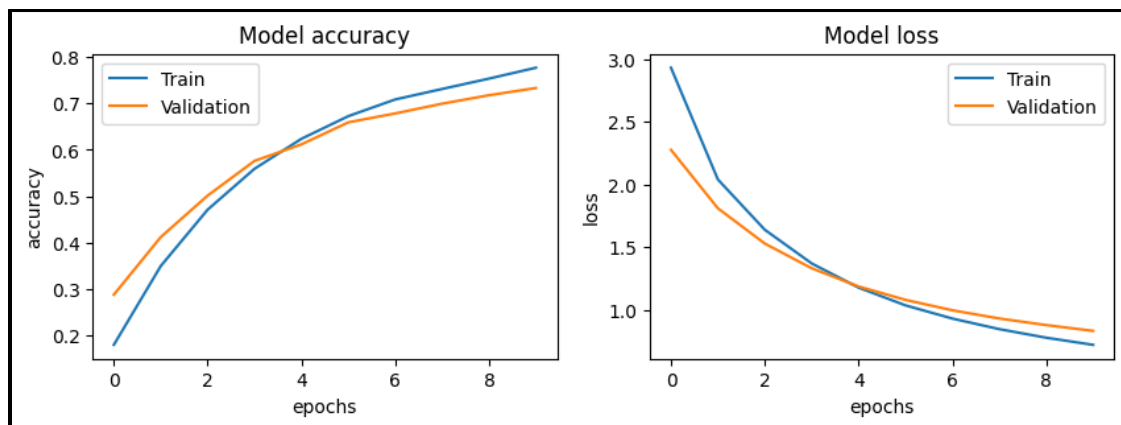
- First, we are importing the VGG16 pre-trained model from the TensorFlow Keras applications module. Next, you are importing the preprocess_input function from the VGG16 module.
- This function performs the preprocessing steps required by the VGG16 model before making predictions on an image. Specifically, it scales the pixel values to be in the range [-1, 1], subtracts the mean RGB values, and reverses the order of the channels from RGB to BGR. We are then creating a variable called vgg_pre_input that references the preprocess_input function.
- This variable will be used later to preprocess the input images for the VGG16 model. Next, we are calling the augment function from the fp module.
- This function takes the vgg_pre_input function, a training data iterator called train_it, and a test data iterator called test_it as inputs, and generates augmented images for the training set using various data augmentation techniques such as random rotations, flips, and zooms. The augmented data is returned as three separate data generators: train_gen_VGG, valid_gen_VGG, and test_gen_VGG, which are used for training, validation, and testing, respectively.
- After that, we are defining two variables called model_VGG16 and stop_early using the run_model function from the fp module.
- This function takes the VGG16 model as input and creates a new

model for fine-tuning on the new dataset. Specifically, it freezes all the layers in the VGG16 model except for the last few layers, which are replaced by new layers that are trained on the new dataset. The `stop_early` variable is a callback function that is used to monitor the validation loss during training and stop the training process early if the loss stops improving.

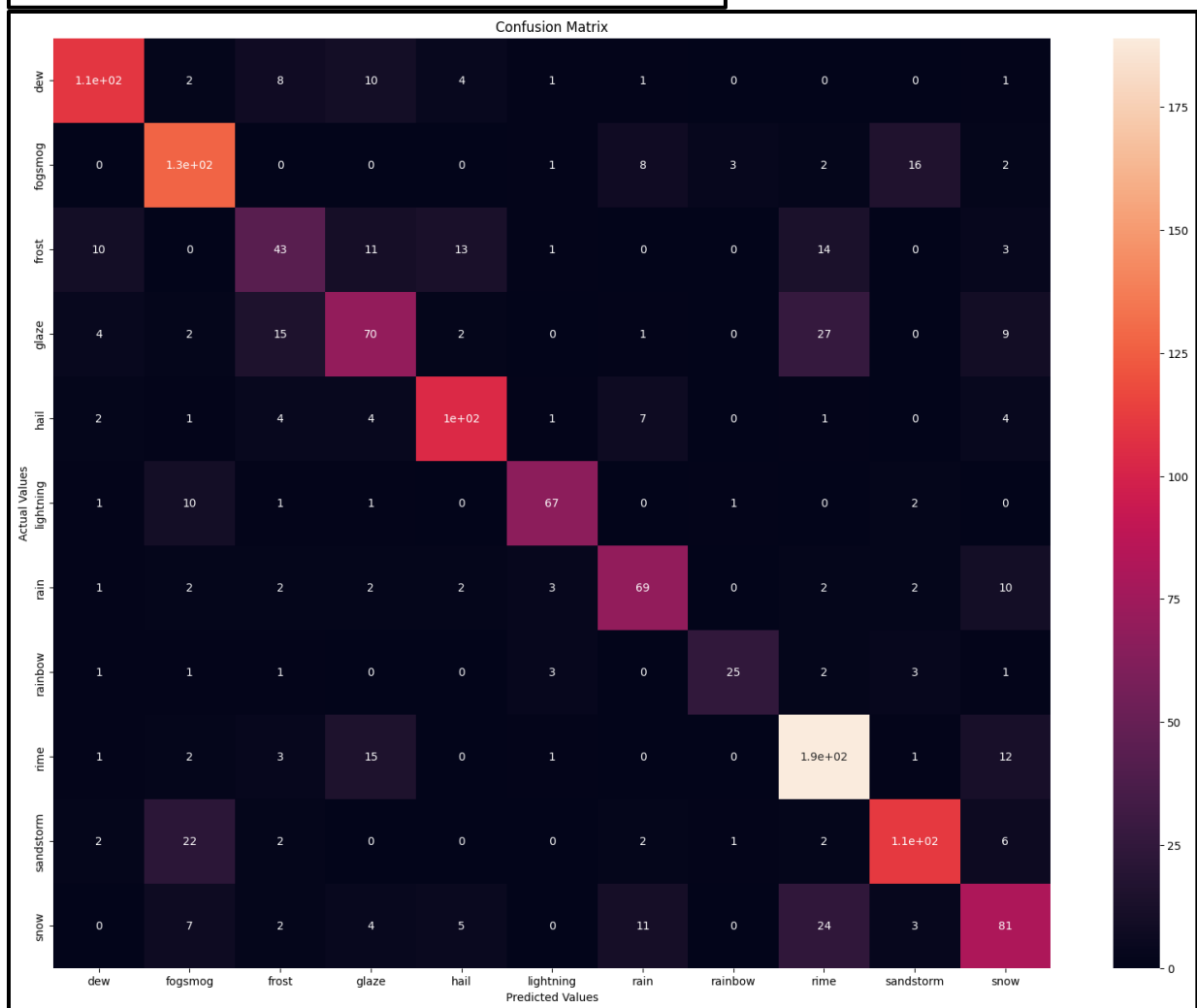
- Next, we are training the `model_VGG16` on the augmented data using the `fit` function. This function trains the `model_VGG16` on the augmented data for a fixed number of epochs (in this case, 10), and uses the validation set to monitor the model's performance and prevent overfitting. The `stop_early` callback is passed to the `fit` function to stop the training process early if the validation loss stops improving. The training history is stored in the `history` variable, which can be used later for plotting and evaluation.
- We are calling the plotting function from the `fp` module. This function takes the training history (i.e., the `history` variable returned by the `fit` function), the test data generator (`test_gen_VGG`), the training data generator (`train_gen_VGG`), the trained VGG16 model (`model_VGG16`), the test labels (`testLabels`), and the path to the test images (`testFilePath`) as inputs. It then plots the training and validation loss and accuracy over each epoch, and also generates a confusion matrix and classification report to evaluate the model's performance on the test set. The `history_VGG` variable stores the output of this function, which includes the training and validation loss and accuracy plots, the confusion matrix, and the classification report.
- Finally, we are calling the `result_test` function from the `fp` module. This function takes the test data generator (`test_gen_VGG`) and the trained VGG16 model (`model_VGG16`) as inputs, and evaluates the model's performance on the test set by computing the accuracy, precision, recall, and F1-score. The

output of this function is stored in the result_VGG16 variable, which includes the accuracy, precision, recall, and F1-score values.


```
Found 4392 validated image filenames belonging to 11 classes.
Found 1097 validated image filenames belonging to 11 classes.
Found 1373 validated image filenames belonging to 11 classes.
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5
58889256/58889256 [=====] - 10s 0us/step
Epoch 1/10
69/69 [=====] - 1304s 19s/step - loss: 2.9329 - accuracy: 0.1801 - val_loss: 2.2765 - val_accuracy: 0.2881
Epoch 2/10
69/69 [=====] - 1314s 19s/step - loss: 2.0398 - accuracy: 0.3497 - val_loss: 1.8095 - val_accuracy: 0.4120
Epoch 3/10
69/69 [=====] - 1304s 19s/step - loss: 1.6380 - accuracy: 0.4711 - val_loss: 1.5277 - val_accuracy: 0.5014
Epoch 4/10
69/69 [=====] - 1314s 19s/step - loss: 1.3690 - accuracy: 0.5592 - val_loss: 1.3302 - val_accuracy: 0.5761
Epoch 5/10
69/69 [=====] - 1317s 19s/step - loss: 1.1758 - accuracy: 0.6236 - val_loss: 1.1848 - val_accuracy: 0.6117
Epoch 6/10
69/69 [=====] - 1574s 23s/step - loss: 1.0338 - accuracy: 0.6724 - val_loss: 1.0779 - val_accuracy: 0.6591
Epoch 7/10
69/69 [=====] - 1395s 20s/step - loss: 0.9283 - accuracy: 0.7083 - val_loss: 0.9945 - val_accuracy: 0.6782
Epoch 8/10
69/69 [=====] - 1302s 19s/step - loss: 0.8439 - accuracy: 0.7309 - val_loss: 0.9288 - val_accuracy: 0.6992
Epoch 9/10
69/69 [=====] - 1293s 19s/step - loss: 0.7753 - accuracy: 0.7532 - val_loss: 0.8763 - val_accuracy: 0.7174
Epoch 10/10
69/69 [=====] - 1303s 19s/step - loss: 0.7184 - accuracy: 0.7766 - val_loss: 0.8300 - val_accuracy: 0.7329
```




	precision	recall	f1-score	support
dew	0.83	0.80	0.82	137
fogsmog	0.72	0.80	0.76	159
frost	0.53	0.45	0.49	95
glaze	0.60	0.54	0.57	130
hail	0.80	0.81	0.81	128
lightning	0.86	0.81	0.83	83
rain	0.70	0.73	0.71	95
rainbow	0.83	0.68	0.75	37
rime	0.72	0.84	0.78	224
sandstorm	0.80	0.75	0.78	148
snow	0.63	0.59	0.61	137
accuracy			0.73	1373
macro avg	0.73	0.71	0.72	1373
weighted avg	0.72	0.73	0.72	1373




Actual: snow
Predicted: rain



Actual: dew
Predicted: frost



Actual: rime
Predicted: rime



22/22 [=====] - 332s 15s/step - loss: 0.8668 - accuracy: 0.7254
 Test Loss: 0.86682
 Test Accuracy: 72.54%

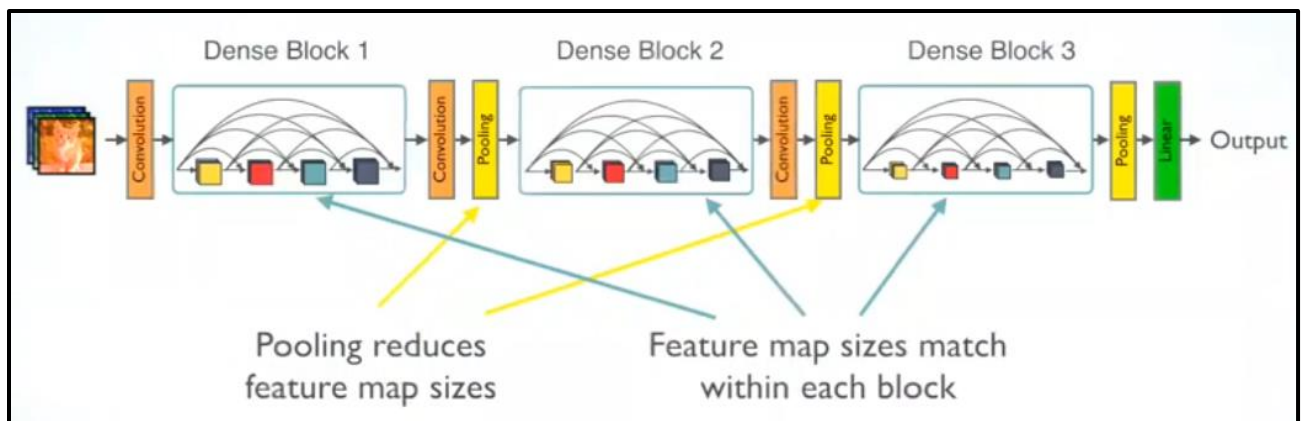
```
model_VGG16.save('C://Users/MrBot/Documents/Weather-Image-Classification-using-Deep-Learning-main/Interfaces/model_VGG16_1.h5')
```

At the end we save the model in '.h5' file to use it in app to classify the images.

4) DenseNet201 –

DenseNet-201 is a convolutional neural network that is 201 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224.

<https://www.mathworks.com/help/deeplearning/ref/densenet201.html>



Dense net is densely connected-convolutional networks. It is very similar to a ResNet with some-fundamental differences. ResNet is using an additive method that means they take a previous output as an input for a future layer, & in DenseNet takes all previous output as an input for a future layer as shown in the above image. The output from that particular dense block is given to what is called a transition layer and this layer is like one-by-one convolution followed by Max pooling to reduce the size of the feature maps. So, the transition layer allows for Max pooling, which typically leads to a reduction in the size of your feature maps.

<https://www.analyticsvidhya.com/blog/2022/03/introduction-to-densenets-dense-cnn/>

```
from tensorflow.keras.applications import DenseNet201
from tensorflow.keras.applications.densenet import preprocess_input

DenseNet201_pre_input=preprocess_input
train_gen_DenseNet201, valid_gen_DenseNet201 = fp.augment(DenseNet201_pre_input,train_it,test_it)
model_DenseNet201, stop_early=fp.run_model(DenseNet201)
history = model_DenseNet201.fit(
    train_gen_DenseNet201,
    validation_data=valid_gen_DenseNet201,
    epochs=8,
    callbacks=stop_early,
    verbose=1
)
history=fp.plotting(history,test_gen_DenseNet201,train_gen_DenseNet201, model_DenseNet201,testLabels,testFilePath)
result_DenseNet201 = fp.result_test(test_gen_DenseNet201,model_DenseNet201)
```

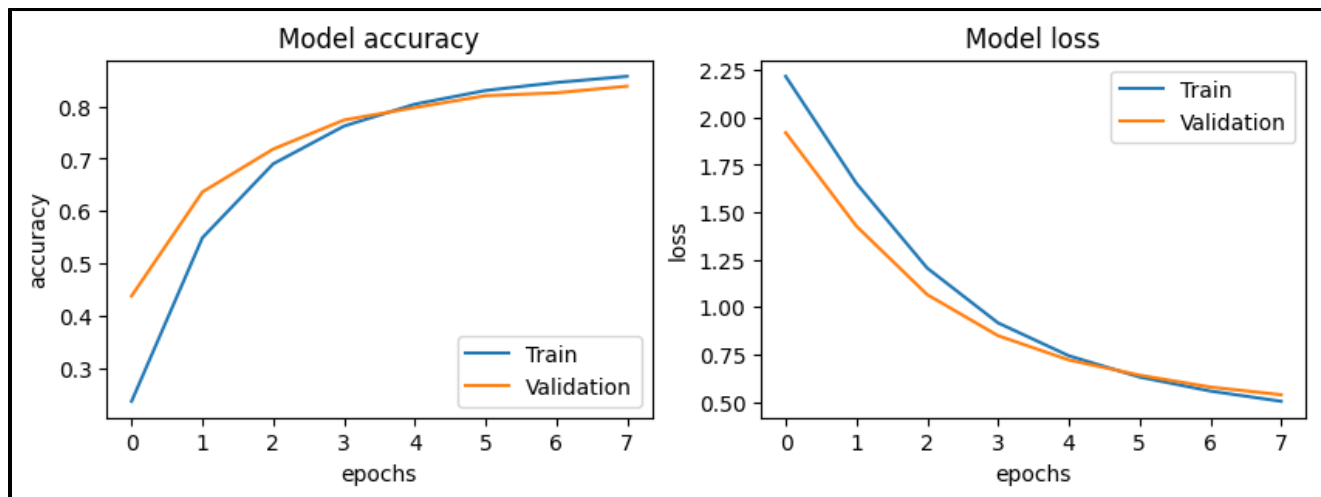
- First, you are importing the DenseNet201 model and its preprocess_input function from Keras, and defining the preprocess_input function as DenseNet201_pre_input.
- Next, we are using the augment function from the fp module to

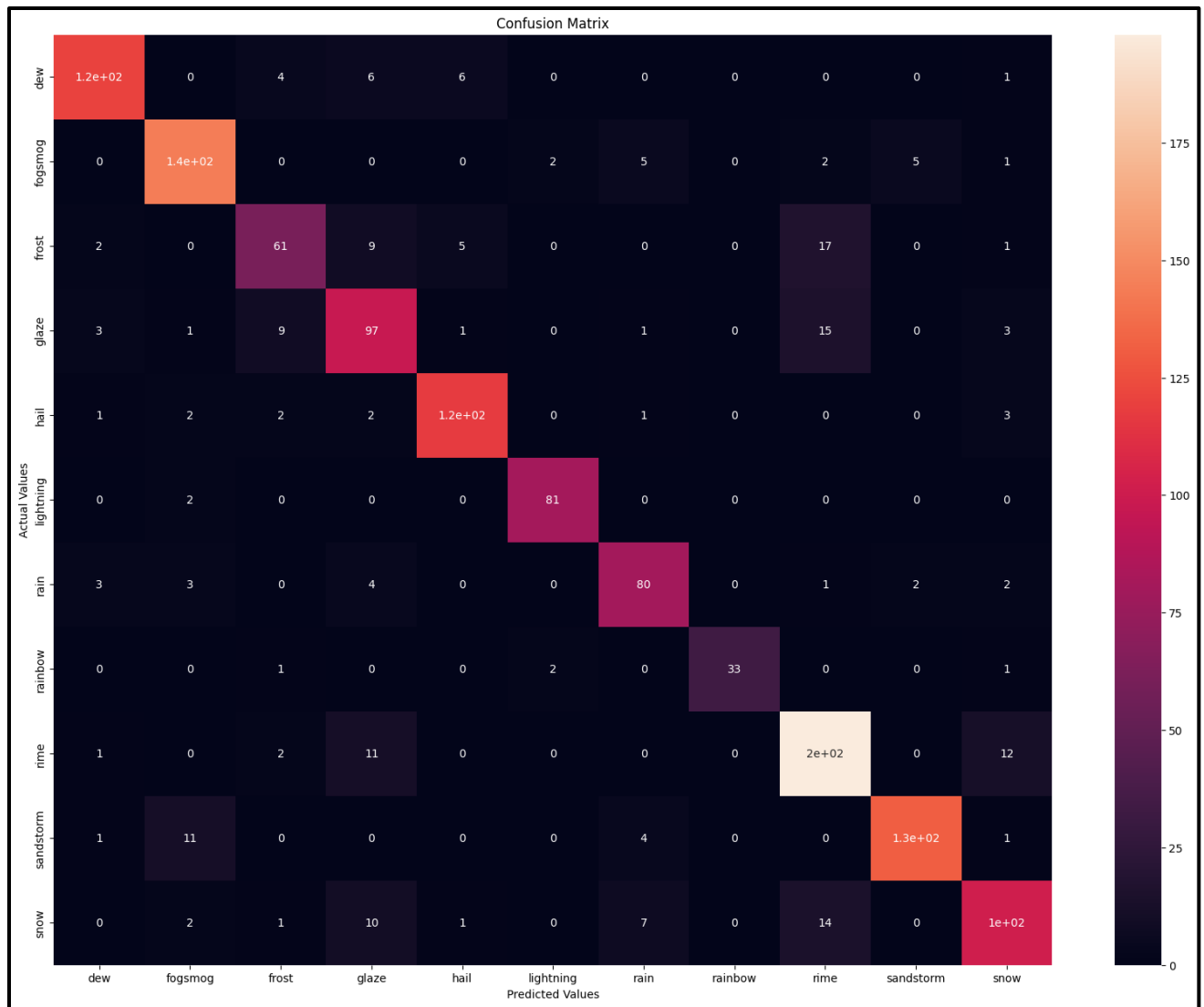
generate augmented training, validation, and test data generators using the `DenseNet201_pre_input` function. Then, we are using the `run_model` function from the `fp` module to initialize and compile a `DenseNet201` model, and to set up an Early Stopping callback.

- After that, we are fitting the model on the training data using the `fit` method, and storing the history of the training process in the `history` variable. You are using the same arguments as before: the training and validation data generators, the number of epochs, the `EarlyStopping` callback, and `verbose=1` to print the training progress to the console.
- Next, we are using the plotting function from the `fp` module to visualize the training history and evaluate the performance of the model on the test set.
- This function takes the training history (i.e., the `history` variable returned by the `fit` function), the test data generator (`test_gen_DenseNet201`), the training data generator (`train_gen_DenseNet201`), the trained `DenseNet201` model (`model_DenseNet201`), the test labels (`testLabels`), and the path to the test images (`testFilePath`) as inputs. It then plots the training and validation loss and accuracy over each epoch, and also generates a confusion matrix and classification report to evaluate the model's performance on the test set. The `history_DenseNet201` variable stores the output of this function, which includes the training and validation loss and accuracy plots, the confusion matrix, and the classification report.
- Finally, we are using the `result_test` function from the `fp` module to compute the accuracy, precision, recall, and F1-score of the model on the test set. This function takes the test data generator (`test_gen_DenseNet201`) and the trained `DenseNet201` model (`model_DenseNet201`) as inputs, and computes the accuracy, precision, recall, and F1-score of the model on the test set. The

output of this function is stored in the result_DenseNet201 variable.

```
Found 4392 validated image filenames belonging to 11 classes.
Found 1097 validated image filenames belonging to 11 classes.
Found 1373 validated image filenames belonging to 11 classes.
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/densenet/densenet201\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5
74836368/74836368 [=====] - 12s 0us/step
Epoch 1/8
69/69 [=====] - 692s 10s/step - loss: 2.2176 - accuracy: 0.2361 - val_loss: 1.9204 - val_accuracy: 0.4376
Epoch 2/8
69/69 [=====] - 680s 10s/step - loss: 1.6526 - accuracy: 0.5487 - val_loss: 1.4269 - val_accuracy: 0.6363
Epoch 3/8
69/69 [=====] - 687s 10s/step - loss: 1.2056 - accuracy: 0.6901 - val_loss: 1.0657 - val_accuracy: 0.7183
Epoch 4/8
69/69 [=====] - 688s 10s/step - loss: 0.9165 - accuracy: 0.7623 - val_loss: 0.8491 - val_accuracy: 0.7739
Epoch 5/8
69/69 [=====] - 686s 10s/step - loss: 0.7430 - accuracy: 0.8042 - val_loss: 0.7208 - val_accuracy: 0.7976
Epoch 6/8
69/69 [=====] - 679s 10s/step - loss: 0.6305 - accuracy: 0.8304 - val_loss: 0.6406 - val_accuracy: 0.8204
Epoch 7/8
69/69 [=====] - 694s 10s/step - loss: 0.5567 - accuracy: 0.8456 - val_loss: 0.5778 - val_accuracy: 0.8259
Epoch 8/8
69/69 [=====] - 691s 10s/step - loss: 0.5027 - accuracy: 0.8575 - val_loss: 0.5375 - val_accuracy: 0.8387
22/22 [=====] - 175s 8s/step
```





	precision	recall	f1-score	support
dew	0.92	0.88	0.90	137
fogsmog	0.87	0.91	0.89	159
frost	0.76	0.64	0.70	95
glaze	0.70	0.75	0.72	130
hail	0.90	0.91	0.91	128
lightning	0.95	0.98	0.96	83
rain	0.82	0.84	0.83	95
rainbow	1.00	0.89	0.94	37
rime	0.80	0.88	0.84	224
sandstorm	0.95	0.89	0.92	148
snow	0.80	0.74	0.77	137
accuracy			0.85	1373
macro avg	0.86	0.85	0.85	1373
weighted avg	0.85	0.85	0.85	1373



```
22/22 [=====] - 172s 8s/step - loss: 0.5340 - accuracy: 0.8478  
Test Loss: 0.53396  
Test Accuracy: 84.78%
```

```
model_DenseNet201.save('C://Users/MrBot/Documents/Weather-Image-Classification-using-Deep-Learning-main/Interfaces/model_DenseNet201.h5')
```

At the end we save the model in '.h5' file to use it in app to classify the images

Chapter 5: Visualization and Testing

5.1) Comparison of Models Through Visualization

Objective of the model comparison is better performance of machine learning solution. Above we have explained all models that we are using in our project. We are comparing these 4 models on the basis of val_loss and val_accuracy parameters. In the following image we can see that how we have compared all the models:

On Basis of Losses –

```
output = pd.DataFrame({'Model': ['MobileNet', 'ResNet50', 'VGG16', 'DenseNet201'],
                      'loss': [result_MobileNet[0], result_ResNet[0], result_VGG16[0],
                               result_DenseNet201[0]]})

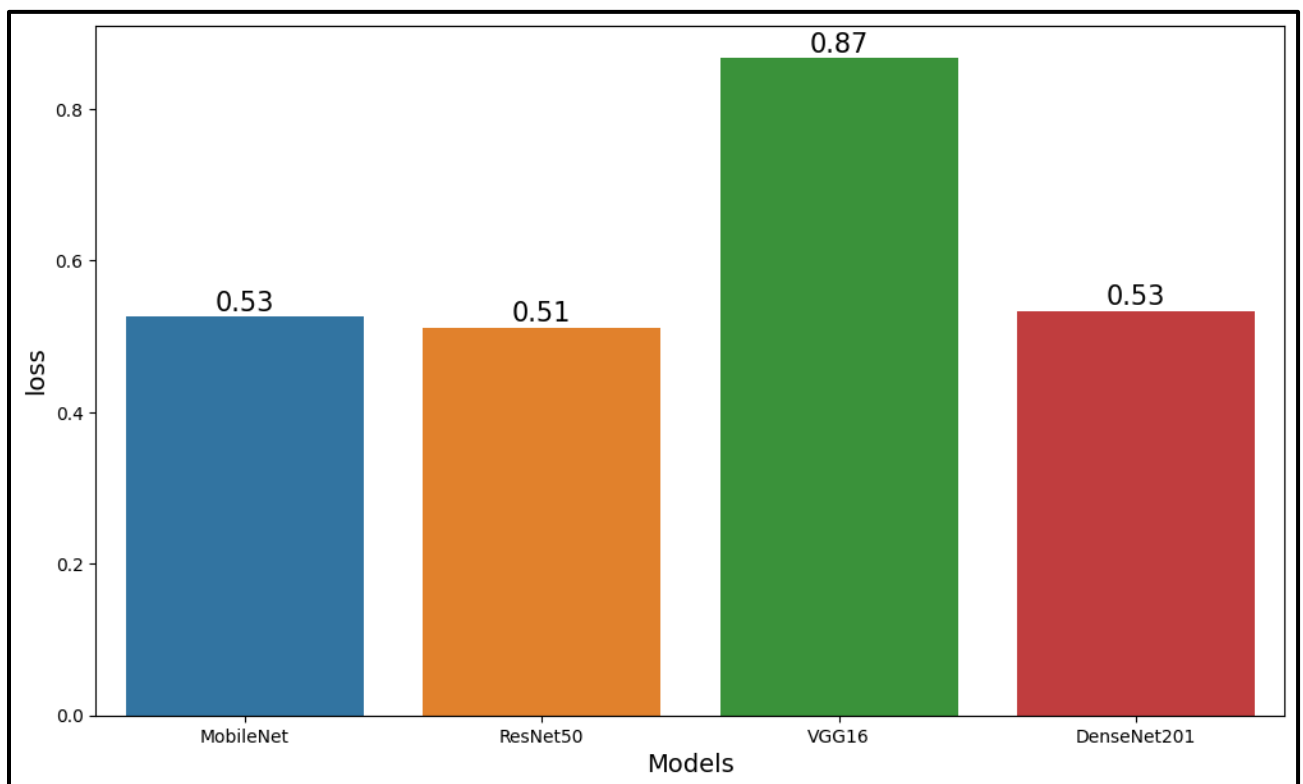
plt.figure(figsize=(12, 7))
plots = sns.barplot(x='Model', y='loss', data=output)
for bar in plots.patches:
    plots.annotate(format(bar.get_height(), '.2f'),
                  (bar.get_x() + bar.get_width() / 2,
                   bar.get_height()), ha='center', va='center',
                  size=15, xytext=(0, 8),
                  textcoords='offset points')

plt.xlabel("Models", size=14)
plt.ylabel("loss", size=14)
plt.show()
```

- First code creates a pandas DataFrame object named "output" with four columns: "Model", "Loss", and the names of four deep learning models: MobileNet, ResNet50, VGG16, and DenseNet201. The "Loss" column contains the corresponding accuracy scores for each model, which are obtained from four different variables: "result_MobileNet [0]", "result_ResNet [0]", "result_VGG16[0]", and "result_DenseNet201[0]".
- The figure size of the plot is set to 12 inches by 7 inches using the "figure (figsize= (12, 7))" function call. This code creates a bar plot using the seaborn library with the x-axis representing the different models (MobileNet, ResNet50, VGG16, and DenseNet201) and the y-axis representing their loss scores.
- The bar plot is created using the DataFrame object named

"output" which should have two columns, "Model" and "loss", where "Model" holds the name of the models and "loss" holds their respective loss values.

- The "sns.barplot" function is used to create the actual bar plot, where the "x" parameter specifies the "Model" column of the "output" DataFrame, and the "y" parameter specifies the "loss" column of the same DataFrame.
- The for loop is used to annotate the height of each bar with its corresponding loss value using the "annotate" function of matplotlib. The height of each bar is obtained using the "get_height" method, and the center of each bar is obtained using the "get_x" and "get_width" methods. The loss value is displayed with two decimal points using the "format" function.
- Finally, the x-axis and y-axis labels are set using the "xlabel" and "ylabel" functions, respectively, and the plot is displayed using the "show" function of matplotlib.



Losses in ResNet50 is comparatively lower than others where as in VGG16 those are the most.

On Basis of Accuracy –

```

output = pd.DataFrame({'Model':['MobileNet','ResNet50','VGG16','DenseNet201'],
                      'Accuracy':[result_MobileNet[1],result_ResNet[1],result_VGG16[1],
                                  result_DenseNet201[1]]})

plt.figure(figsize=(12, 7))
plots = sns.barplot(x='Model', y='Accuracy', data=output)
for bar in plots.patches:
    plots.annotate(format(bar.get_height(), '.2f'),
                  (bar.get_x() + bar.get_width() / 2,
                   bar.get_height()), ha='center', va='center',
                  size=15, xytext=(0, 8),
                  textcoords='offset points')

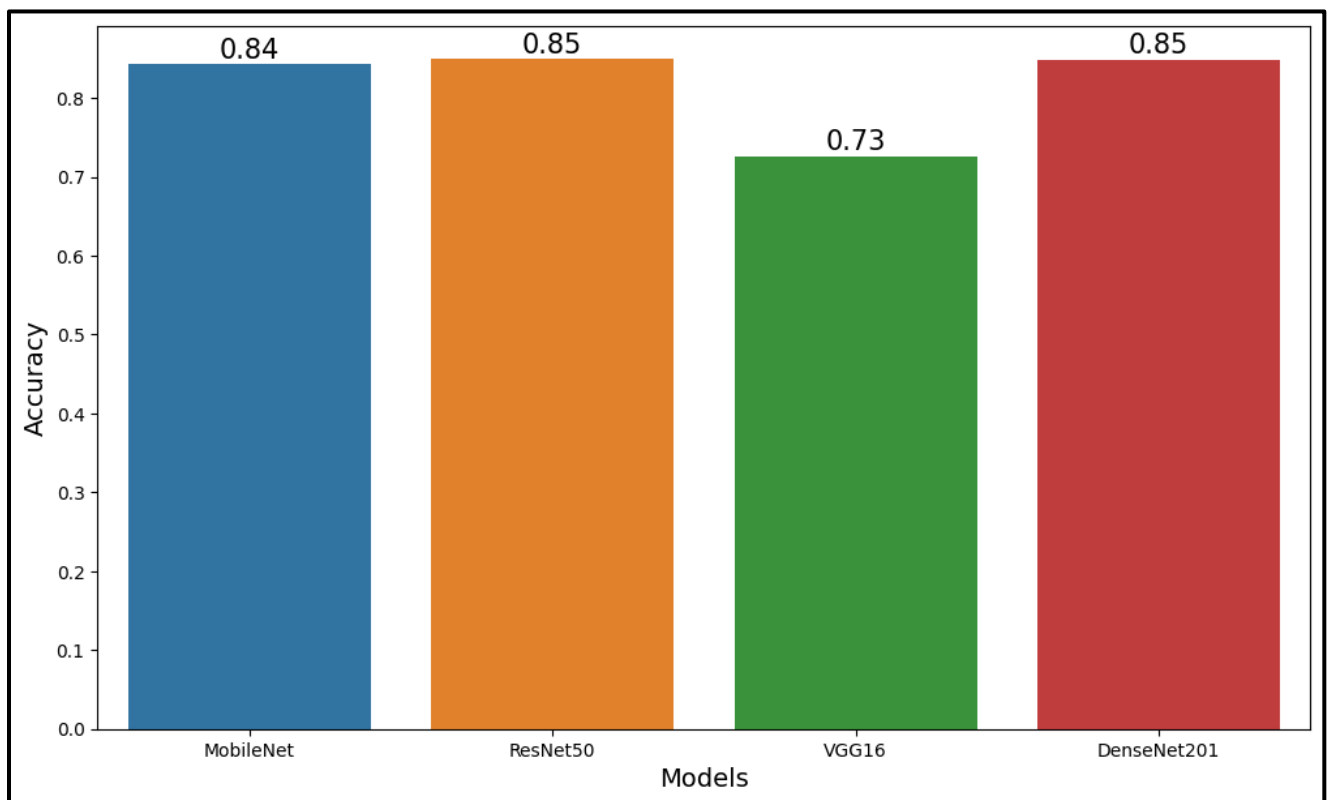
plt.xlabel("Models", size=14)
plt.ylabel("Accuracy", size=14)
plt.show()

```

- This code creates a pandas DataFrame object named "output" with four columns: "Model", "Accuracy", and the names of four deep learning models: MobileNet, ResNet50, VGG16, and DenseNet201. The "Accuracy" column contains the corresponding accuracy scores for each model, which are obtained from four different variables: "result_MobileNet [1]", "result_ResNet [1]", "result_VGG16[1]", and "result_DenseNet201[1]".
- The figure size of the plot is set to 12 inches by 7 inches using the "figure (figsize= (12, 7))" function call. This code creates a bar plot using the seaborn library with the x-axis representing the different models (MobileNet, ResNet50, VGG16, and DenseNet201) and the y-axis representing their accuracy scores.
- The bar plot is created using the DataFrame object named "output" which should have two columns, "Model" and "Accuracy", where "Model" holds the name of the models and "Accuracy" holds their respective accuracy values.
- The "sns.barplot" function is used to create the actual bar plot, where the "x" parameter specifies the "Model" column of the "output" DataFrame, and the "y" parameter specifies the "Accuracy" column of the same DataFrame.
- The for loop is used to annotate the height of each bar with its corresponding accuracy value using the "annotate" function of

matplotlib. The height of each bar is obtained using the "get_height" method, and the center of each bar is obtained using the "get_x" and "get_width" methods. The accuracy value is displayed with two decimal points using the "format" function.

- Finally, the x-axis and y-axis labels are set using the "xlabel" and "ylabel" functions, respectively, and the plot is displayed using the "show" function of matplotlib.



The Accuracy of ResNet50 and Densnet201 is almost identical and more compare to others where as accuracy of VGG16 is at lowest of all.

5.2) User Interface and Testing –

After comparing the model, we have built the user interface. User Interface which we are using in our project is Tkinter. With the help of UI, we can get the access of model and we can classify weather images. Code which required for UI are as follows:

```
import tkinter as tk
from PIL import ImageTk, Image
from tkinter import filedialog
import numpy as np
from tensorflow.keras.preprocessing.image import img_to_array

path='model_ResNet_1.h5'

#loading the model to be used
from keras.models import load_model
# Load model
model = load_model(path)
```

- This code imports the required modules and loads a pre-trained Keras model from the specified file path.
- The tkinter module is imported as "tk" to create a graphical user interface (GUI) for selecting an image file to be classified using the loaded model. The PIL (Python Imaging Library) module is imported to handle image loading and manipulation. The filedialog submodule is imported from tkinter to create a file dialog box to choose an image file.
- The path variable specifies the file path of the pre-trained Keras model to be used. The Keras model is loaded using the "load_model" function from the "keras.models" module.

```
def load_img():
    global img, image_data
    for img_display in frame.winfo_children():
        img_display.destroy()

    image_data = filedialog.askopenfilename(initialdir="/", title="Choose an image",
                                           filetype= (("all files", "*..*"), ("png files", "*.png")))

    basewidth = 300 # Processing image for displaying
    img = Image.open(image_data)
    wpercent = (basewidth / float(img.size[0]))
    hsize = int((float(img.size[1]) * float(wpercent)))
    img = img.resize((basewidth, hsize), Image.ANTIALIAS)
    img = ImageTk.PhotoImage(img)
    file_name = image_data.split('/')
    panel = tk.Label(frame, text= str(file_name[len(file_name)-1]).upper()).pack()
    panel_image = tk.Label(frame, image=img).pack()
```

- This code defines a function `load_img ()` that loads an image file from the user's file system and displays it in a tkinter window. Here's what the code does: The function first declares that it will be using global variables `img` and `image_data`. This means that the variables will be accessed and updated within the function.
- The function then deletes any existing child widgets of the tkinter frame where the image will be displayed. This ensures that only one image is displayed at a time.
- The user is prompted to select an image file using the `filedialog.askopenfilename()` method. The method displays a file dialog window that allows the user to browse their file system and select an image file.
- The code then processes the selected image to display it with a maximum width of 300 pixels. This is done to ensure that the image is not too large to fit in the tkinter window. The `Image.open()` method is used to open the selected image file, and the `Image.resize()` method resizes the image to the desired width while maintaining its aspect ratio. The `ImageTk.PhotoImage()` method is then used to create a tkinter-compatible image object.
- The code extracts the file name from the full file path and displays it as a label in the tkinter window. Finally, the tkinter-compatible image object is displayed in a label widget in the tkinter window.

```
def classify():
    original = Image.open(image_data)
    original = original.resize((224, 224), Image.ANTIALIAS)
    numpy_image = img_to_array(original)
    image_batch = np.expand_dims(numpy_image, axis=0)

    label = model.predict(image_batch)
    table = tk.Label(frame, text="").pack()

    l={"dew":label[0][0],"fogsmog":label[0][1],"frost":label[0][2],"glaze":label[0][3],"hail":label[0][4],"lightning":label[0][5],
    "rain":label[0][6],"rainbow":label[0][7],"rime":label[0][8],"sandstorm":label[0][9],"snow":label[0][10]}
    val=label.max()

    #def get_key(val):
    for key, value in l.items():
        if val == value:
            result = tk.Label(frame,text=str(key),font=("", 30)).pack()
```

- This code defines a function `classify ()` that uses a pre-trained machine learning model to classify the image loaded by the `load_img ()` function. Here's what the code does:
- The function first opens the original image file using the `Image.open()` method and resizes it to 224x224 pixels using the `Image.resize()` method. This is done to ensure that the image has the same dimensions as the images that the pre-trained model was trained on.
- The `img_to_array()` function from the `tensorflow.keras.preprocessing.image` module is then used to convert the image to a numpy array. This is necessary because the pre-trained model expects numpy arrays as input.
- The numpy array is then converted into a batch of one image using the `np.expand_dims ()` method. This is necessary because the pre-trained model expects a batch of images as input, even if there's only one image.
- The pre-trained model is then used to predict the label of the input image using the `model.predict()` method. The output of this method is a numpy array of probabilities for each class label.
- The function then creates an empty label widget in the tkinter window to create space for the output. A dictionary `l` is defined that maps each class label to its corresponding probability value in the output numpy array. The dictionary is used to map the maximum probability value to its corresponding class label.
- The maximum probability value `val` is obtained by calling the `max ()` method on the output numpy array. The code then loops through the dictionary `l` to find the class label that corresponds to the maximum probability value `val`.
- The class label is then displayed in a label widget in the tkinter window using the `tk.Label()` method with a font size of 30.


```

root = tk.Tk()
root.title('Weather Image Classifier')
root.resizable(False, False)
tit = tk.Label(root, text="Weather Image Recognition", padx=25, pady=6, font=("", 30), bg='white').pack()
canvas = tk.Canvas(root, height=500, width=500, bg='grey')
canvas.pack()
frame = tk.Frame(root, bg='white')
frame.place(relwidth=0.8, relheight=0.8, relx=0.1, rely=0.1)
chose_image = tk.Button(root, text='Choose Image',
                        padx=35, pady=10,
                        fg="white", bg="black", command=load_img)
chose_image.pack(side=tk.LEFT)
class_image = tk.Button(root, text='Classify Image',
                        padx=35, pady=10,
                        fg="white", bg="black", command=classify)
class_image.pack(side=tk.RIGHT)
root.mainloop()

```

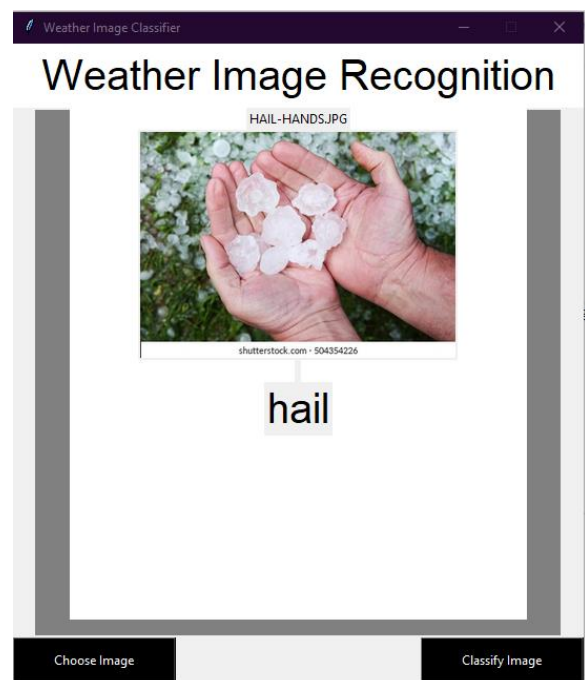
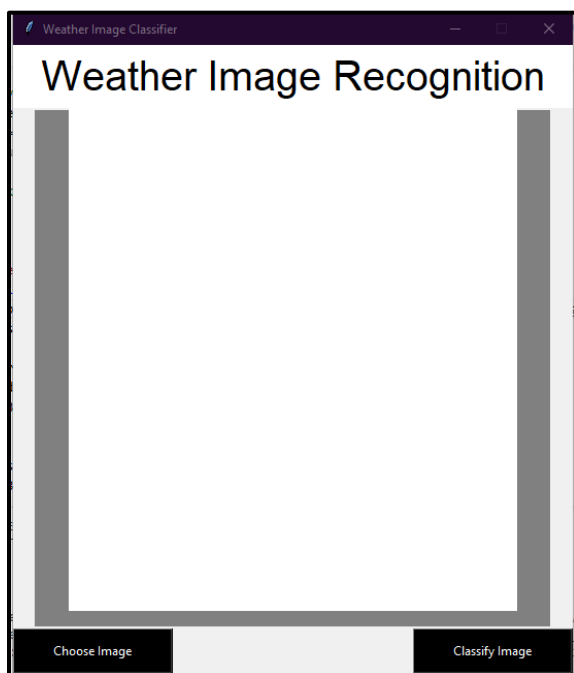
- This code creates a tkinter GUI application that allows the user to load an image and classify it using a pre-trained machine learning model. Here's what the code does. The code first imports the tkinter module and creates a new tkinter window using the tk.Tk() method. The title of the window is set to "Weather Image Classifier" using the title() method.
- The window is set to be non-resizable using the resizable() method with arguments False and False. A label widget tit is created using the tk.Label() method with text "Weather Image Recognition", padding of 25 pixels on the x-axis and 6 pixels on the y-axis, and font size of 30. The pack() method is then called on the label widget to add it to the tkinter window.
- A canvas widget canvas is created using the tk.Canvas() method with height and width of 500 pixels and background color of "grey". The pack() method is then called on the canvas widget to add it to the tkinter window.
- A frame widget frame is created using the tk.Frame() method with background color of "white". The place() method is then called on the frame widget to set its size and position relative to the tkinter window.
- Two button widgets chose_image and class_image are created using the tk.Button() method with text "Choose Image" and "Classify Image", padding of 35 pixels on the x-axis and 10 pixels on the y-axis, and foreground color of "white" and background color of "black". The command argument of the tk.Button()

method is set to the `load_img()` function for `chose_image` and to the `classify()` function for `class_image`.

- The `pack()` method is then called on the two button widgets to add them to the tkinter window, with `chose_image` placed on the left side of the window and `class_image` placed on the right side of the window.
- The `pack()` method is then called on the two button widgets to add them to the tkinter window, with `chose_image` placed on the left side of the window and `class_image` placed on the right side of the window.
-

User Interface and Test Result –

From the below image we can see that how our model classifies images given by user as input.



Chapter 6: Conclusion

- Data Augmentation helps to train the models to achieve more accuracy with less losses
- Callback and early stopping helps to achieve consistency in performance of the models
- Accuracy of ResNet50 and DenseNet201 is More Compare to other models.
- Losses are more in VGG16 and Very much less in ResNet50 compare to others.