University of Rome "La Sapienza"

Department of Ingegneria Informatica, Automatica e Gestionale

Machine Learning
Homework 2

Image classification problem

Tedesco Giancarlo

Matricola:
2057231

CHAPTER 1: Introduction

The task of this report is to solve an image classification, that is, the subject of the second homework.
The main topic of this report is to explain solutions and reasoning behind the approaches I used to accomplish the classification task

1.1 Domain problem:

Search for a dataset that must satisfy some conditions:

- At least 10 classes
- At least 150 images per class
- Excluded educational datasets

Find a solution for the following problem on the chosen dataset:

- an image classification problem

1.2 Approach to the problems

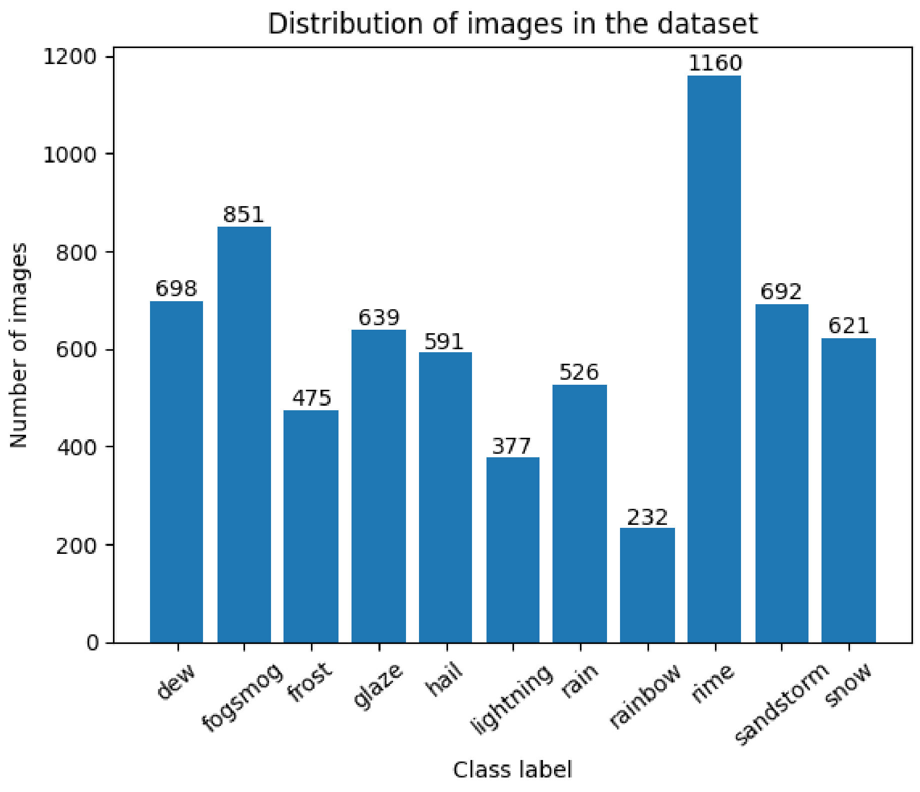
To accomplish that task, the work has been based on a try-and-compare approach: models have been trained and tested, with different configurations of the same, and the results have been compared. This report analyses and compares in details different architectures for the two models: a pre-trained model and a custom model, built from scratch.

CHAPTER 2: The dataset

2.1 Dataset description

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

In the table below, is represented the image distribution among the classes:



For a total of 6862 images:

- 1160 images belong to the class “rime” (17%)
- 851 images belong to the class “fog/smog” (12%)
- 698 images belong to the class “dew” (10%)
- 692 images belong to the class “sandstorm” (10%)
- 639 images belong to the class “glaze” (9%)
- 621 images belong to the class “snow” (9%)
- 591 images belong to the class “hail” (9%)
- 526 images belong to the class “rain” (8%)
- 475 images belong to the class “frost” (7%)
- 377 images belong to the class “lightning” (6%)
- 232 images belong to the class “rainbow” (3%)

This plot provides an important information: the dataset is imbalanced with a ratio of 1:5 between the largest and the smallest class.

2.2 Data-subset generation

On the first try, the method used to create training and validation sets was image_dataset_from_directory(). Unfortunately, in the validation step, the predicted labels tensor of validation subset had its elements shuffled leading to a mismatch with the true label's tensor. The results generated, inevitably, were false.

The method used to generate training and validation sets is the flow_from_directory() applied to ImageDataGenerator(). The latter method allows for real-time data augmentation.
The following parameters were used for the subset training:

- **rescale = 1. / 255**: rescale the input data by dividing it by 255.
- **rotation_range=20**: randomly rotate the input images by a random angle within the given range
- **width_shift_range=0.2**: randomly shift the input images horizontally and vertically by a fraction of the total width or height
- **height_shift_range=0.2**: randomly shift the input images horizontally and vertically by a fraction of the total width or height
- **horizontal_flip=True**: randomly flip the input images horizontally
- **validation_split=0.3**: specifies the number of elements to be used in validation set

The parameters used for validation subset are the following:

- **rescale = 1. / 255**: rescale the input data by dividing it by 255
- **validation_split=0.3**: specifies the number of elements to be used in validation set

2.2.1 Hyperparameter for the subsets

In order to generate the training and validation subsets, are used some hyperparameters for both splits that will be used for the neural networks also.

The first hyperparameter set is the "batch_size": due to lack of memory and computational power, the size of batches is fixed to a value of 8.

The second hyperparameter modified refers to the image dimension: the image height is reduced to a standard value of 225

The image width parameter is the third hyperparameter modified: also, this parameter is for image resizing and set the width to 317

The last hyperparameter is the "seed" parameter used to specify a fixed seed for the data shuffling that occurs before each epoch. The value chosen is equal to 123.

Here the summary of hyperparameters used for splitting:

- **batch_size = 8**
- **img_height = 225**
- **img_width = 317**
- **seed=123**

2.3 Integrity check of splits

With the aim to exclude any possibility of introducing bias, the generated subsets are been examined to verify the goodness of the previous steps.

The first control was made to detect overlapping of the same image between the two sets. This because using an image used in training set to evaluate the network can introduce undue bias.
How is shown in the code, this problem does not occur:

```
No overlaps between the training and validation sets.
```

After, a second control was made to check dimension of input, the correct loading of classes and images in each subset and some useful information. Here the following results obtained matches the desired values:

```
Image input: (225, 317, 3)
Classes: ['dew', 'fogsmog', 'frost', 'glaze', 'hail', 'lightning', 'rain', 'rainbow', 'rime', 'sandstorm', 'snow']
Loaded 4808 training samples from 11 classes.
Loaded 2054 test samples from 11 classes.
```

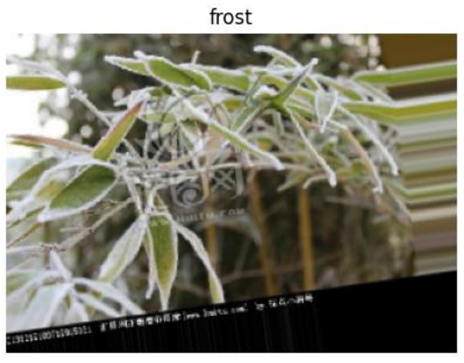
Moreover, a third check was made to retrieve information on the batches that will be the input for the networks. Also, this control performed as desired:

```
(8, 225, 317, 3)
(8, 11)
```

The first line represents the shape of the image batch, the second corresponds to the shape of the labels batch. Both results match the values given in the beginning and demonstrate that the dataset has been perfectly manipulated and it is ready to be fed into the networks.

2.4 Random samples

Just out of curiosity, some images have been extracted from the set of train. As you can see from the table below, the images have been pre-processed correctly and moreover, the last image returns a very important information: some images have a watermark. Therefore, the initial dataset is also affected by noise that could somehow affect the performance of the neural networks.



CHAPTER 3: Preparing the network

3.1 Configuration of models learning process

To configure the learning process of the models are been used the following parameters:
as optimizer, it is been used the "**Adam**" optimizer which uses gradient information to update the model weights in an iterative manner; since the classes are mutually exclusive and data are categorical, the loss used is the "**CategoricalCrossentropy**". The metric that will be used to evaluate the model during training and testing is the "**accuracy**" metrics. This metrics will be computed on the training and validation data and reported at the end of each epoch.

3.2 Callback definition

Some callback functions are been implemented in order to perform certain actions at the end of each epoch. Functions have been defined for save the model weights, adjus the learning rate and early stopping

3.2.1 ModelCheckpoint callback

The ModelCheckpoint callback is used for tracking the performance of the model on a validation dataset, saving the model weights with the best performance on the validation set and using them to make comparison later on. I set some parameters to save and get access to a model with the following goals:

- **monitor='val_accuracy'**: choosen metrics to evaluate
- **mode='max'**: the model will be saved if the monitored metric has a higher value at the end of the epoch than the previous best
- **save_best_only=True**: save only the model with the best metrics value
- **save_weights_only= False**: to save the entire model (mainly to access to history after loading of a saved model)

3.2.2 ReduceLROnPlateau callback

The ReduceLROnPlateau callback is used to reduce the learning rate of the optimizer when the performance on a given metric stops improving. It is useful for fine-tuning the learning rate during training and avoiding getting stuck in local minima. The function was set as reported below:

- **monitor='val_loss'**: the metric to be monitored for reducing the learning rate
- **patience=3**: the number of epochs to wait before reducing the learning rate
- **factor=0.1**: factor by which the learning rate will be reduced
- **min_lr=1e-6**: lower bound of the learning rate
- **max_lr=1e-2**: upper bound of the learning rate

3.2.3 EarlyStopping callback

The EarlyStopping callback is used to stop the training process when the performance on a selected metric stops improving. It is very useful for avoiding overfitting and reducing the risk of training for too long unnecessarily. The function is defined with the following parameters:

- **monitor='val_loss'**: the metric to be monitored for early stopping
- **patience=6**: the number of epochs to wait before stopping the training when the metric stops improving
- **mode = 'min'**: the mode in which to monitor the metric (minimum value of the metric)
- **min_delta=0.006**: the minimum change in the monitored metric to qualify as an improvement
- **restore_best_weights=True**: whether to restore the model weights from the epoch with the best performance on the monitored metric

3.3 Training phase definition

All the models are been trained with the same training configuration. Given the datasets generated as descripted in the previous chapters, the number of steps per epoch and the validation steps are based on the number of elements and batch size of these sets. Also, al the models were trainend for a total of 30 epochs, obviously taking into account the various callback actions.

From the history object generated by the fit() method, **'loss'**, **'accuracy'**, **'val_loss'**, **'val_accuracy'** metrics were saved to make comparison at the end.

3.4 Evaluation and comparison metrics

To evaluate the models, the metrics I have chosen are the loss and the accuracy. I based the choise of a model respect another one during training phase, on these two parameters. After identifying the best model, I analyzed it in detail through the *f1 score* and the *confusion matrix* in order to have a more or less complete view of the model's capabilities.
The f1 score is a useful metric because it combines precision and recall into a single metric; confusion matrix show how a sample is classified or missclassified, and it is very useful for understanding performances. It is especially useful for imbalanced datasets, where one class is underrepresented, as it allows to see how the model is performing on each class.

CHAPTER 4: DenseNet121 model

4.1 Model description

The DenseNet121 model is characterized by its 121-layer network, which is made up of a series of dense blocks. Each dense block contains a number of convolutional layers, with each layer receiving input from all preceding layers in the block. This allows the model to learn more robust and discriminative features, as the information from the earlier layers is directly passed on to the later layers.
The DenseNet121 model also includes skip connections, which allow the model to bypass a layer and directly connect the input to the output. This helps to alleviate the vanishing gradient problem, where the gradients of the earlier layers become very small as they are backpropagated through the network, making it difficult to update their weights.

The DenseNet model is a network pre-trained on the ImageNet dataset.
The ImageNet dataset consists of more than 14 million images that have been labelled with one of 1000 different object categories. The images in the dataset are organized into a hierarchy of categories, with more general categories at the top and more specific categories at the bottom. Some of the main categories in the dataset are: clothing, vehicle, sport, electronics, animals, food, buildings, ecc…
The ImageNet dataset includes images of various weather conditions. The dataset includes photographs of outdoor scenes that may depict different weather conditions, such as sunshine, clouds, rain, snow, and fog. However, the ImageNet dataset is primarily focused on objects and scenes, and it does not include a specific class for different weather conditions. Instead, the images in the dataset are labelled with more general classes that may include images with different weather conditions. For example, an image of a mountain landscape in the snow might be labelled with the class "mountain," rather than a specific class for snow or cold weather.

Hence, the dataset used for the classification is quite different from the one used for the pre-training.

The model used is a slightly modified DenseNet121 network: the top of the network has been replaced with 10 custom layers to perform the final classification by reducing the number of neurons in each fully-connected layer. The last 10 custom layers added to the DenseNet model are the following in the exactly same flow:

- flatten = Flatten(name='Start_of_the_customTop')(output_extractor)
- flatten_norm = BatchNormalization()(flatten)
- dense1 = Dropout(0.4)(flatten_norm)
- dense1 = Dense(200, activation='relu')(dense1)
- dense1 = BatchNormalization()(dense1)
- dense2 = Dropout(0.4)(dense1)
- dense2 = Dense(100, activation='relu')(dense2)
- dense2 = BatchNormalization()(dense2)
- dense3 = BatchNormalization()(dense2)
- dense3 = Dense(num_classes, activation='softmax')(dense3)

In the list above, are mentioned also the hyperparameters of the layers used in most tests. However, below we can see the summary of the custom DenseNet model:

Total params: 21,683,235
Trainable params: 20,650,755
Non-trainable params: 1,032,480

4.2 DenseNet121 without fine-tuning

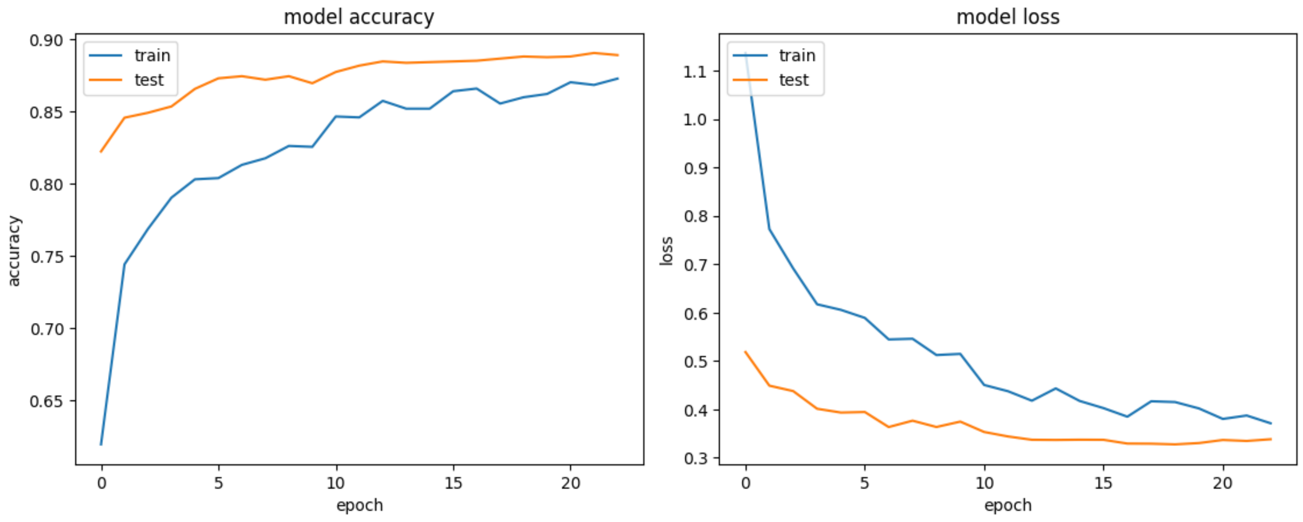
The first classification test is made on the model shown before, with all layers not trainable (except for the custom top): the weights of these layers cannot be adjusted during the training process.
The training process has stopped at the end of epoch 22 thanks to the *EarlyStopping* callback. The model started its training process with the following results at the end of the first epoch:

loss: 1.1367 - accuracy: 0.6196 - val_loss: 0.5186 - val_accuracy: 0.8223 - lr: 0.0010

and achieved at epoch 22 the following results:

loss: 0.3714 - accuracy: 0.8727 - val_loss: 0.3383 - val_accuracy: 0.8890 - lr: 1.0000e-05

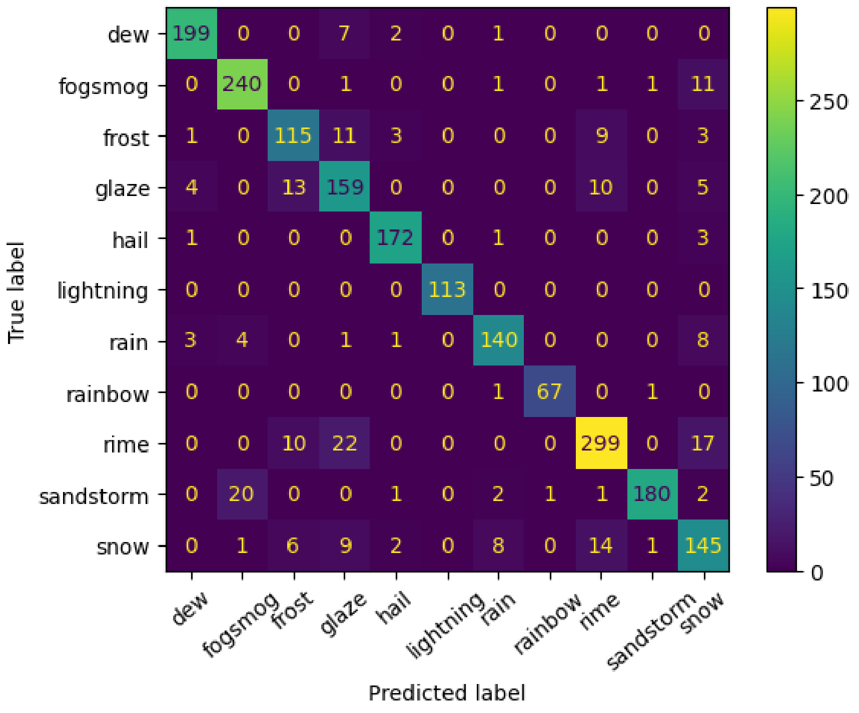
We can observe that the learning rate was decreased, according to the *ReduceLROnPlateau* callback. Furthermore, this model coincides with the one with the best performances.
In the images below, it is represented the accuracy and loss behavior over the time.



For the best model, the results on validation set are reported in the below left table. On the right, we can observe the confusion matrix computed from the model on validation set:

accuracy	loss
0.890	0.335

	f1-score
Dew	0.954
Fogsmog	0.923
Frost	0.804
Glaze	0.793
Hail	0.961
Lightning	1.000
Rain	0.900
Rainbow	0.978
Rime	0.877
Sandstorm	0.923
snow	0.763



	dew	fogsmog	frost	glaze	hail	lightning	rain	rainbow	rime	sandstorm	snow
dew	199	0	0	7	2	0	1	0	0	0	0
fogsmog	0	240	0	1	0	0	1	0	1	1	11
frost	1	0	115	11	3	0	0	0	9	0	3
glaze	4	0	13	159	0	0	0	0	10	0	5
hail	1	0	0	0	172	0	1	0	0	0	3
lightning	0	0	0	0	0	113	0	0	0	0	0
rain	3	4	0	1	1	0	140	0	0	0	8
rainbow	0	0	0	0	0	0	1	67	0	1	0
rime	0	0	10	22	0	0	0	0	299	0	17
sandstorm	0	20	0	0	1	0	2	1	1	180	2
snow	0	1	6	9	2	0	8	0	14	1	145

The results on this first model were very high, but with some errors. Anyway, the diagonal of the confusion matrix is well defined and it is a good result.
The above results can be proved loading the "no-fine_model_ep22.h5" in the code.

4.3 DenseNet121 with fine-tuning

The second test was made on the previous standard custom model of DenseNet, but now with the last 31 layers trainable. This is the fine-tuned version of the model.
The training process has stopped at the end of epoch 23 thanks to the *EarlyStopping* callback. The model started its training process with the following results at the end of the first epoch:

loss: 1.2943 - accuracy: 0.5849 - val_loss: 1.2380 - val_accuracy: 0.6986 - lr: 0.0010

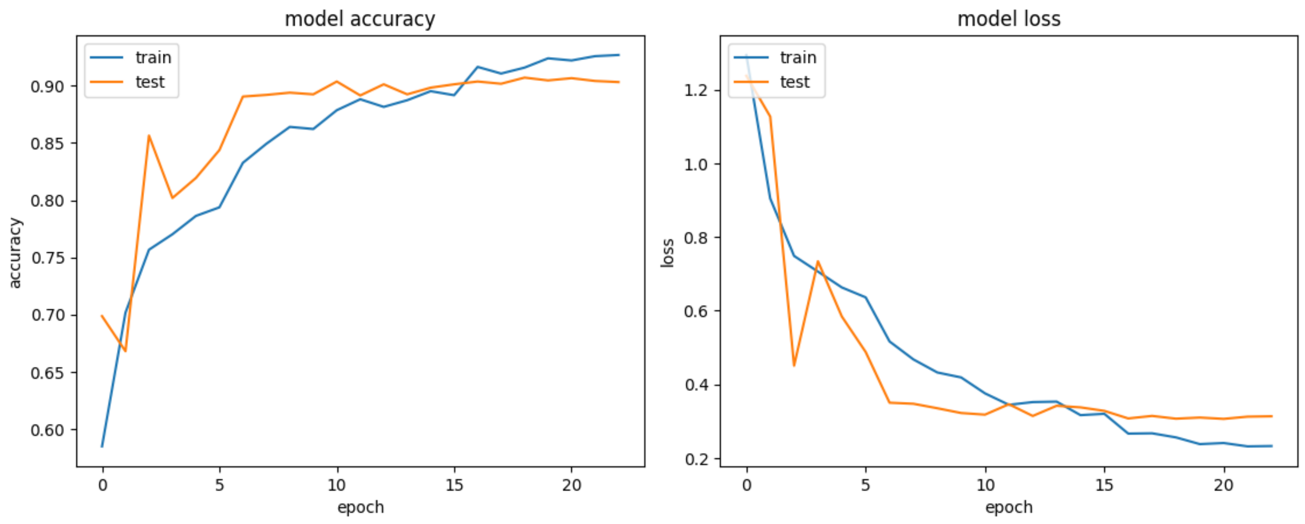
and achieved at epoch 23 the following results:

loss: 0.2327 - accuracy: 0.9268 - val_loss: 0.3135 - val_accuracy: 0.9031 - lr: 1.0000e-05

We can observe that the learning rate was decreased also in this case, according to the *ReduceLROnPlateau* callback. Moreover, the *ModelCheckpoint* callback indicated as the best model, the one at the end of the epoch 19 as indicated below:

loss: 0.2562 - accuracy: 0.9158 - val_loss: 0.3070 - val_accuracy: 0.9070 - lr: 1.0000e-05

On this model will be done the evaluation phase.
In the following image is shown the accuracy and loss behavior over time:

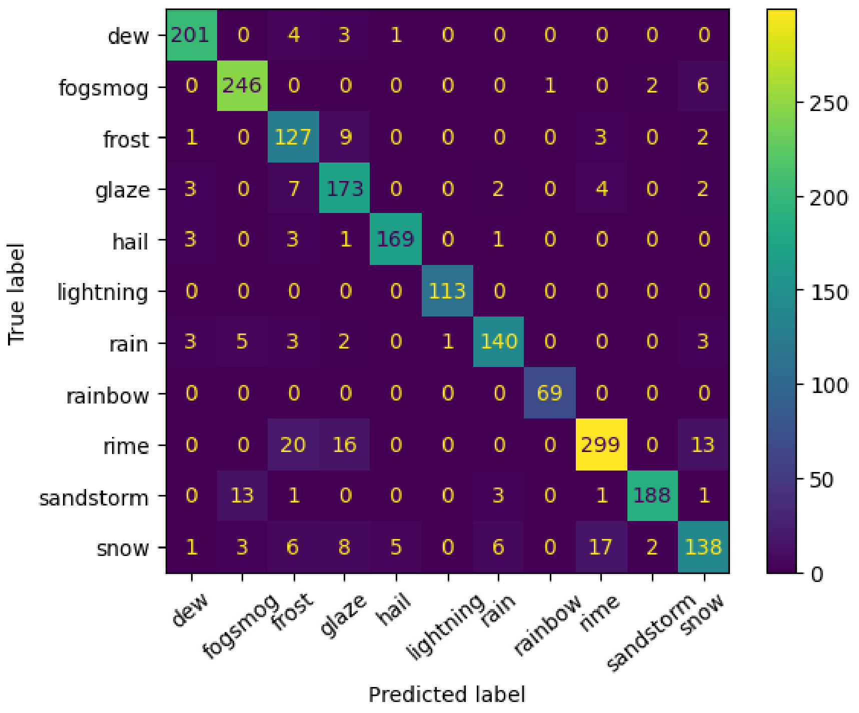


With reference to the same graphs, this second model has peaks and the trend is less linear than the first model. Also, both the methods tend to overfit after about 18-20 epochs.
Anyway the fine tuned version of the model achieved the highest performances

For the best model, the results on validation set are reported in the below left table. On the right, we can observe the confusion matrix computed from the model on validation set:

accuracy	**loss**
0.907	0.307

	f1-score
Dew	0.955
Fogsmog	0.943
Frost	0.812
Glaze	0.859
Hail	0.960
Lightning	0.996
Rain	0.906
Rainbow	0.993
Rime	0.890
Sandstorm	0.942
snow	0.786



Now, with all these information it is possible to compare the results of the two versions of the models.
As you can imagine, the fine-tuned model was able to achieve better results by reducing the number of errors in the classification. This second model lost abilities in predicting images belonging to “hail” and snow”, but improved classification of the other classes. At the end, we can notice how almost all f1 scores for each class in fine-tuned model improved regarding the results obtained for the first model.

Alse these results can be examined, loading the model named: “fine_tuned_model_ep19.h5”.

4.4 Some extra experiments on DenseNet121

The model was also trained with all layers unfreezed. The architecture at the end, overfitted with an average *accuracy* value of about 0.70.

Other configurations with regularizers top layers were also tested, but results were strangely worse than the other two models illustrated previously.

The code of these last models is not provided so as not to weigh down the folder and the work.

CHAPTER 5: Custom model based on convolutional layers

5.1 Model description

This model has three convolutional layers, each followed by a max pooling layer and a dropout layer to reduce overfitting. The output of the convolutional layers is flattened and passed through a fully connected layer with 512 nodes, followed by another dropout layer. The final layer is an output layer with 11 nodes, one for each class, and uses the softmax activation function to produce probabilities for each class. Here, the architecture of the model (the layers on which the regularizers are applied have been marked):

input_8 (InputLayer)
conv2d_18 (Conv2D) *[REGULARIZER LAYER]*
max_pooling2d_18 (MaxPooling2D)
dropout_34 (Dropout)
conv2d_19 (Conv2D) *[REGULARIZER LAYER]*
max_pooling2d_19 (MaxPooling2D)
dropout_35 (Dropout)
conv2d_20 (Conv2D) *[REGULARIZER LAYER]*
max_pooling2d_20 (MaxPooling2D)
dropout_36 (Dropout)
flatten_6 (Flatten)
dense_23 (Dense)
dropout_37 (Dropout)
dense_24 (Dense)

From the summary() method, the prospect on the parameters is the following:

Total params: 63,145,035
Trainable params: 63,145,035
Non-trainable params: 0

5.2 Custom model with regularizers

The following model is a basic model built to perform the classification. The network has regularizers on each convolutional layer: kerner regularizer on first (L2 penaly with a factor of 0.01), bias regularizer on second and third layers (also L2 penaly with a factor of 0.01).
It performed as expected: quite good but not very well. After 27 epochs, the *EarlyStopping* callback stopped training.
The model started its training process with the following results at the end of the first epoch:

loss: 2.5714 - accuracy: 0.3819 - val_loss: 1.9802 - val_accuracy: 0.4815 - lr: 0.0010

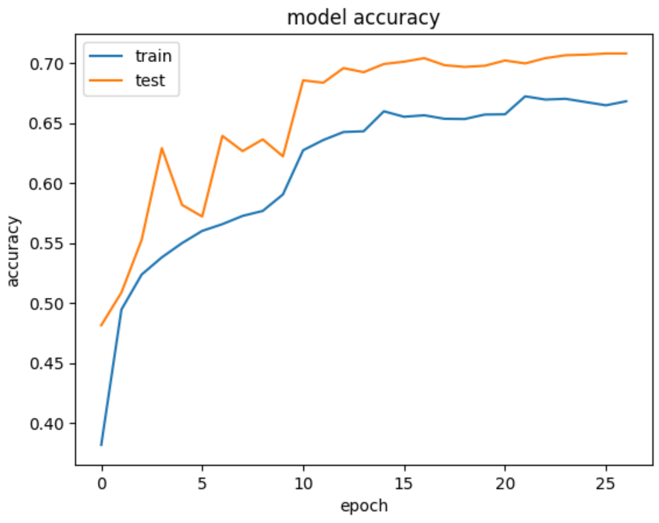
and achieved at epoch 27 the following results:

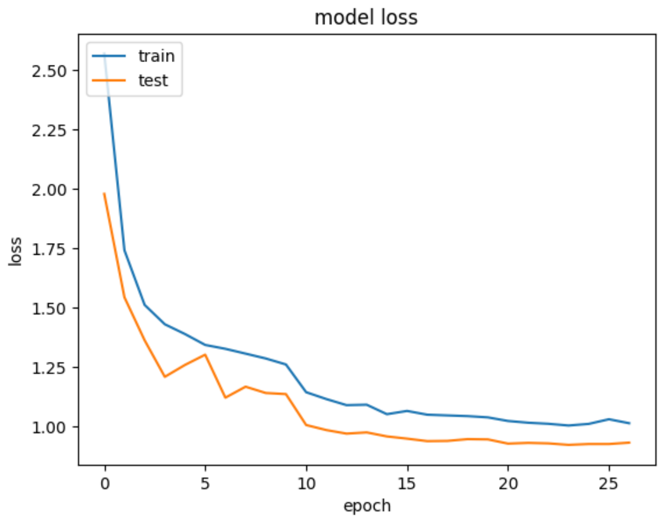
loss: 1.0144 - accuracy: 0.6681 - val_loss: 0.9327 - val_accuracy: 0.7079 - lr: 1.0000e-05

We can observe that the learning rate was decreased also in this case, according to the *ReduceLROnPlateau* callback.
The model achieved the best accuracy at epoch 26 with the following values:

loss: 1.0311 - accuracy: 0.6647 - val_loss: 0.9268 - val_accuracy: 0.7079 - lr: 1.0000e-05

In the images below, it is represented the accuracy and loss behavior over the time:

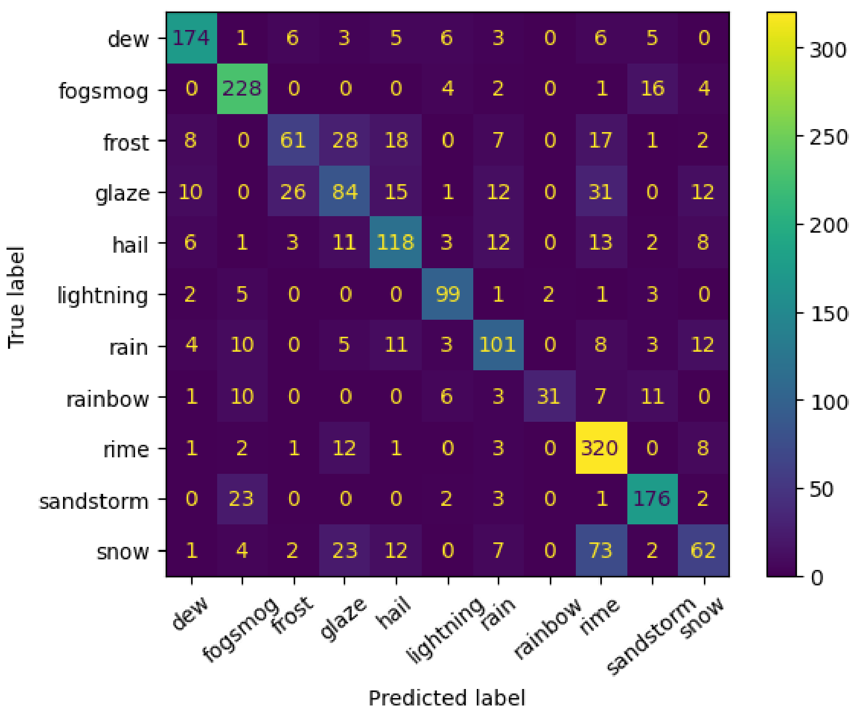




For the best model, the results on validation set are reported in the below left table. On the right, we can observe the confusion matrix computed from the model on validation set:

accuracy	**loss**
0.708	0.927

	f1-score
Dew	0.837
Fogsmog	0.846
Frost	0.506
Glaze	0.471
Hail	0.661
Lightning	0.835
Rain	0.650
Rainbow	0.608
Rime	0.775
Sandstorm	0.826
snow	0.419



	dew	fogsmog	frost	glaze	hail	lightning	rain	rainbow	rime	sandstorm	snow
dew	174	1	6	3	5	6	3	0	6	5	0
fogsmog	0	228	0	0	0	4	2	0	1	16	4
frost	8	0	61	28	18	0	7	0	17	1	2
glaze	10	0	26	84	15	1	12	0	31	0	12
hail	6	1	3	11	118	3	12	0	13	2	8
lightning	2	5	0	0	0	99	1	2	1	3	0
rain	4	10	0	5	11	3	101	0	8	3	12
rainbow	1	10	0	0	0	6	3	31	7	11	0
rime	1	2	1	12	1	0	3	0	320	0	8
sandstorm	0	23	0	0	0	2	3	0	1	176	2
snow	1	4	2	23	12	0	7	0	73	2	62

The results obtained from this model are not excellent, in some classes quite high values are obtained, in other classes the results are poor. The confusion matrix shows many errors in the classification: for example, the "snow" class gives many mismatches.

The model is saved with the name " conv_reg_customModel_ep26.h5 " and is available for validating what written above

5.2.1 Variants of custom model with regularizers

The following variation of the model has kernel regularizer on each convolutional layer: L2 penaly with a factor of 0.001. Achieved the best results on validation set at epoch 28 as below:

loss: 0.8406 - accuracy: 0.7235

The code of this model is not provided to not make heavy and redundant of not useful information

The variation of the model with bias regularizer (L2 penaly with a factor of 0.001) on each convolutional layer, achieved its best performances at epoch 26 as following:

loss: 0.9219 - accuracy: 0.6857 - val_loss: 0.8436 - val_accuracy: 0.7191 - lr: 1.0000e-04

The code of this model is not provided to not make heavy and redundant of not useful information

The model with a fourth convolutional branch, unfortunately, stopped training due to earlystop callback condition after 23 epochs with the best following values at epoch 18:

loss: 1.1133 - accuracy: 0.6292 - val_loss: 1.0089 - val_accuracy: 0.6665 - lr: 1.0000e-04

The code of this model is not provided to not make heavy and redundant of not useful information

5.3 Custom model without regularizers

This is the model without regularizers. It has the exact same architecture of the previous model but does not have any regularization parameter set up. After 26 epochs, the *EarlyStopping* callback stopped training.
The model started its training process with the following results at the end of the first epoch:

loss: 2.1715 - accuracy: 0.3151 - val_loss: 1.6739 - val_accuracy: 0.4601 - lr: 0.0010

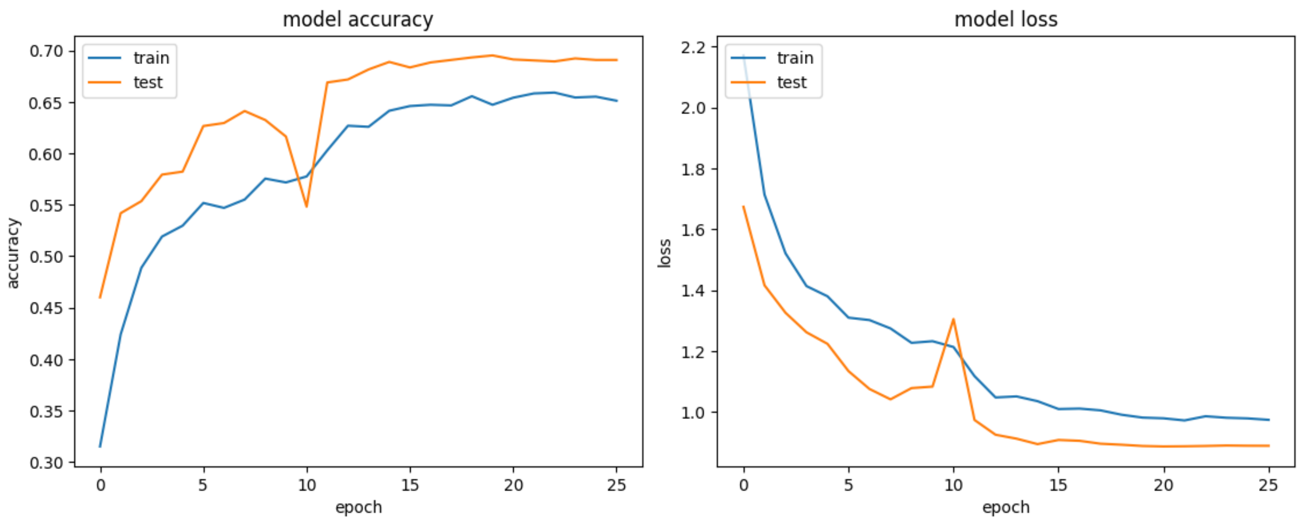
and achieved at epoch 26 the following results:

loss: 0.9739 - accuracy: 0.6512 - val_loss: 0.8886 - val_accuracy: 0.6908 - lr: 1.0000e-06

We can observe that the learning rate was decreased and for the first time it fell to the minimum value, according to the *ReduceLROnPlateau* callback.
The model achieved the best accuracy at epoch 20 with the following values:

loss: 0.9813 - accuracy: 0.6473 - val_loss: 0.8880 - val_accuracy: 0.6952 - lr: 1.0000e-05

The performance of the best model achieved are more or less similar to those of the previous model with regularizers: this model is slightly less accurate, but has a smaller loss value.
In the following image is shown the accuracy and loss behavior over time:

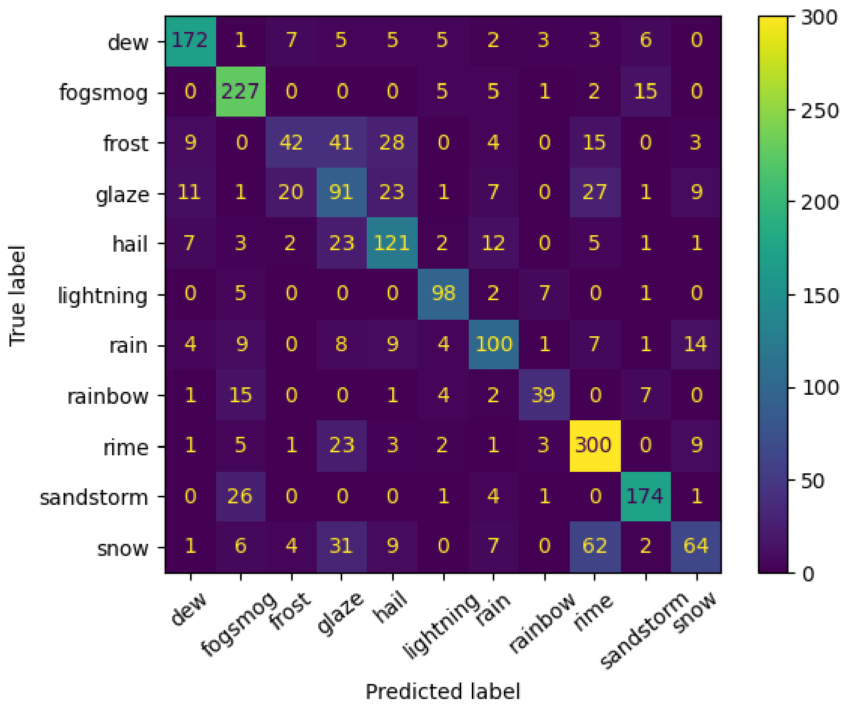


The trend is quite good: after a first phase of learning, there was a small phase in which model has degraded the learning. However, it was able to continue the learning improving accuracy and lowering the loss.

For the best model, the results on validation set are reported in the below left table. On the right, we can observe the confusion matrix computed from the model on validation set:

accuracy	loss
0.695	0.8880

	f1-score
Dew	0.829
Fogsmog	0.821
Frost	0.385
Glaze	0.441
Hail	0.644
Lightning	0.834
Rain	0.660
Rainbow	0.629
Rime	0.780
Sandstorm	0.839
snow	0.446



The results obtained from this model are not excellent as told before, quite high values are obtained in some classes, in other classes the results were poor. The confusion matrix shows many errors in the classification: for example, the “frost”,”glaze”,“hail” classes raises many mismatches.

The model is saved with the name " conv_no_reg_customModel_ep20.h5" and is available for validating what written above

5.4 Comparison between custom models

We now have enough information to compare the models. The metrics used is the “accuracy” on validation and the models are only the best for each architecture. In the following table there is a summary about accuracies of all the previous models:

	accuracy
Custom model with kernel&bias regularizers	0.7080
Custom model with kernel regularizers (0.001)	0.7235
Custom model with bias regularizers	0.7191
Custom model with 4 conv. branches	0.6655
Custom model without regularizers	0.6952

From the table we can see how the models have more or less similar values. However, the best model is the Custom model with kernel regularizers with a penalty value of 0.001 on each convolutional layer.

The comparison with the pre-trained model is quite useless as there is a huge difference in architectures and in the weights of the nodes in the various layers