Machine Learning **Emanuele Giona**
A.Y. 2019/2020

Homework 2: Multiclass weather classification

1 Introduction

In the scope of Smart Cities and Industry 4.0, computer vision and deep learning tools are applied in order to process videos or pictures of urban environments for several purposes.
This task is commonly referred to as urban scene analysis; a specific problem that may be useful to many Smart City applications is tied to weather classification. Instances of this problem consist of labelling pictures with the weather condition depicted in them, and it can be formalized as learning the following target function:

$$f : X \to W = \{w_1, ..., w_k\} \tag{1}$$

with $\forall i \in [1, k]$ w_i refers to a weather condition label.
The function has to be learnt from a dataset that can be formalized as per the following equation:

$$D = \{(x_i, w_i)_{i=1}^{N}\} \tag{2}$$

with $\forall i \in [1, N]$ $x_i \in X, w_i \in W$; X represents to the input space for samples, and W represents the set of weather condition labels, as defined in Equation 1.

2 Dataset

Two datasets have been provided, namely Multi-class Weather Image Dataset (MWI)[1] and Smart-I[2], consisting of:

- MWI: 20 000 images gathered from the Internet, album covers, TV shows, etc., depicting four weather conditions: HAZE, RAINY, SNOWY, and SUNNY;
- Smart-I_{total}: 5 000 images gathered from Smart-Interaction's own sensors and CCTV cameras, depicting three weather conditions: RAINY, SNOWY, and SUNNY.

Actually, subsets of both MWI and Smart-I_{total} have been directly provided in this homework assignment, and the subsets have been the following ones:

- MWI-400: 400 images picked from the MWI dataset, balanced over the four weather classes (100 images per class);
- MWI-2000: 2 000 images picked from the MWI dataset (superset of MWI-400), balanced over the four weather classes (500 images per class);
- MWI-4000: 4 000 images picked from the MWI dataset (superset of MWI-2000), balanced over the four weather classes (1 000 images per class);
- Smart-I: 3 038 images picked from the Smart-I_{total} dataset, unbalanced over the three weather classes;

[1] URL: https://mwidataset.weebly.com
[2] Not publicly available, property of Smart-Interaction (URL: http://www.smart-interaction.com)

- BlindTestSet: 1 500 images with no weather class labels.

Images in all the previously defined dataset are RGB pictures, with average and median image sizes[3] being:

- MWI-400: average 613x784, median 517x719;
- MWI-2000: average 705x878, median 599x720;
- Smart-I: average 495x712, median 480x640.

3 Pre-processing

Two different pre-processing settings have been used: one for the parameter tuning phase and the other for the actual training phase. Both pre-processing phases have been implemented using instances of the `ImageDataGenerator` class provided in the Keras Python library[4].
The parameter tuning pre-processing only consisted of rescaling images with a rescale factor equal to 1/255.
The training pre-processing, additionally to rescaling in the same way of the parameter tuning settings, also applies zoom and rotation variations, width and height shifts, and horizontal flips[5].
They both share the same target image size to which images are scaled to, which is 48x48: this value has been chosen in order to satisfy the minimum input size required by pre-trained models (see Section 4.2 for more details). The differences in pre-processing techniques is due to the fact that the parameter tuning phase should only consider the effectiveness of different models, while robustness (and therefore the need for such data augmentation techniques) can be added to models once parameters are deemed stable.

4 Models

For solving this homework, the models that have been used are the following: a fully custom convolutional neural network (CNN) called `EmaNet`, and a neural network based on transfer learning called `TransferNet`.
Both these models rely on the concept of convolutional neural network: a CNN is a neural network in which at least one layer is a convolutional layer. A convolutional layer is a standard neural network layer in which techniques like sparse connectivity and parameter sharing are implemented; sparse connectivity consists of each unit in the layer receiving a limited number of inputs from the output of the previous layer, while parameter sharing consists of assigning the same weights to all the units in the layer. A convolutional layer learns a *kernel*, which efficiently represents the combination of both of these techniques.
The output of a convolutional layer can be formalized as follows:

$$(I * K)(i, j) = \sum m \sum n I(i + m, j + n) \cdot K(m, n) \tag{3}$$

with I being the input image (or *feature map* in case it's the output of a previous convolutional layer), and K being this layer's kernel.
This value is still a linear transformation of the input, and therefore it is passed through a non-linear activation function; the resulting value is then fed as input to a pooling layer (which can

[3] All values have been rounded to the nearest integer; see function `show_stats()` in the `preprocess.py` file
[4] URL: https://keras.io
[5] See function `full_training()` in the `train.py` file

be optional). Pooling layers are also defined in terms of kernel, but it has a different meaning to the one used in convolutional layers: this kind of kernel only defines a region of the input image to which the pooling function has to be applied to; pooling functions are usually `max()` or `average()`, and are used in order to increase the robustness to local translations.

`EmaNet` and `TransferNet` have been implemented via the functional API of the Keras Python library[6], version 2.2.4, running on top of a GPU-enabled TensorFlow backend[7], version 1.13.

4.1 EmaNet

This fully custom CNN architecture makes use of two separate stacks of convolutional and deconvolutional layers (see Figure 1).

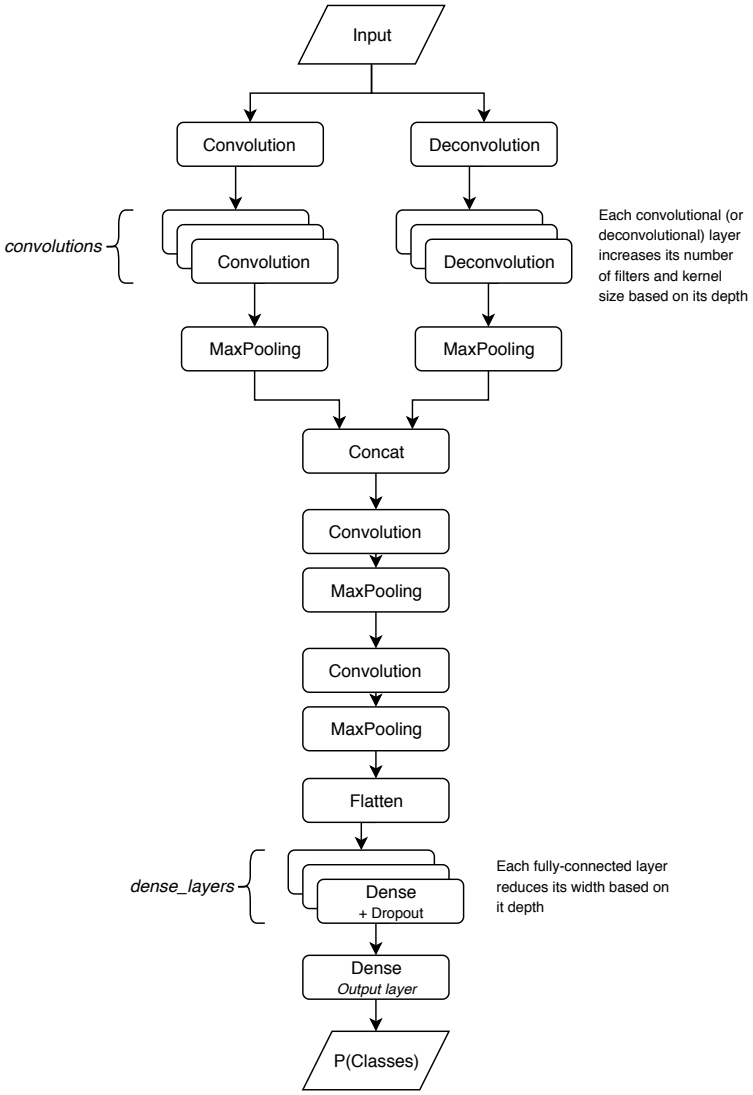


Figure 1: EmaNet model architecture

The stacks contain the same number of layers per operation, applying padding and without using any pooling layer in between them. The idea is that both the convolution and deconvolution operations try to focus on important features of the image, but using a different approach

[6]URL: https://keras.io/models/model
[7]URL: https://www.tensorflow.org/versions/r1.13/api_docs/python/tf

that might be complementary.
Moreover, going through each stack of layers, each subsequent layer uses an increasingly bigger kernel size and number of filters, as per the following formulas:

$$filters_i = filters * \sqrt{i} \tag{4}$$

$$kernel\_size_i = (1 + i, 1 + i) \tag{5}$$

with $filters_i$ and $kernel\_size_i$ respectively being the number of filters and the kernel size used at the i-th layer in the stack $\forall i \in [1, convolutions)$, and $convolutions$ being the size of the stacks used in this model.

Outputs coming from the two stacks are passed through a max pooling layer each, and then concatenated; the resulting vector is fed as input to two additional convolutional layers, each of them using max pooling as well. Finally, the flattened output is fed as input to a fully-connected neural network, in which each layer's width shrinks until the output layer is reached. The shrinkage follows the following formula:

$$units_i = \frac{filters * 5}{i} \tag{6}$$

with $units_i$ being the width of the i-th layer $\forall i \in [1, dense\_layers]$, and $dense\_layers$ being the depth of the fully-connected neural network at the end of the model, before the output layer.

In Equations 4, 5, and 6: $filters$, $convolutions$, and $dense\_layers$ are hyper-parameters of the `EmaNet` model[8].
The model is trained using categorical cross-entropy as loss function and using the Adam optimizer with an initial learning rate equal to 0.1.

4.2 TransferNet

This custom CNN architecture exploits a pre-trained CNN model as feature extractor, then feeding the resulting features into a fully-connected neural network (see Figure 2).

`TransferNet` exploits the concept of transfer learning in the feature extractor setting: the pre-trained model is not fine tuned on the new instances; its fully-connected layers are removed and the rest of the layers are excluded from training. The selected models suitable to be used as feature extractors are provided from the Keras `application` package, and are the following: `DenseNet`[9] and `NASNet`[10], both pre-trained on `ImageNet`[11].
An input image is therefore passed through the feature extractor, obtaining the feature map from its last layer. Its flattened representation is then considered as the input to the fine-grained dense layers. As in the `EmaNet` model (Section 4.1), each dense layer reduces its width until the output layer is reached, following the same shrinking rule (Equation 6), but the instead of having $filters * 5$ as the first layer's width, it is fixed to 128.

The model is trained using categorical cross-entropy as loss function and using the Adam

[8]See function `EmaNet()` in the `models.py` file
[9]URL: https://keras.io/applications/#densenet
[10]URL: https://keras.io/applications/#nasnet
[11]URL: http://www.image-net.org

optimizer with an initial learning rate equal to 0.1; the feature extractor and, as in `EmaNet`, *dense_layers* are hyper-parameters of the `TransferNet` model[12].

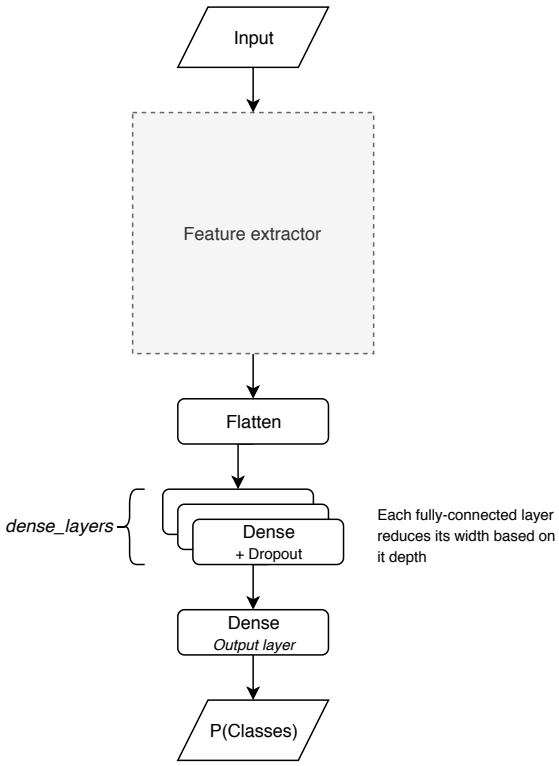


Figure 2: TransferNet model architecture

5 Parameter tuning

The parameter tuning phase has been carried out on the MWI-400 dataset in the following way: for each set of parameters, the model has undergone K-fold cross-validation[13]. For each of the K iterations, the model is initialized with the same weights (after the execution of the `compile()` function) and trained for a maximum of number of epochs, using the `fit()` function with the following settings:

- training set: 80% of MWI-400;
- validation set: the remaining 20% of MWI-400;
- test set: Smart-I;
- custom early stopping[14]: monitor accuracy on the validation set for 5 epochs, only after having trained for at least 1/4 of the number of epochs.

The 80-20 split of MWI-400 is provided by the `train_test_split()` function from the SciKit-Learn Python package[15]; at the end of the training (either having exhausted the maximum epochs or due to early stopping), the accuracy returned by the K-fold cross-validation is the arithmetic mean of the ones achieved on the provided test set over the K iterations.

[12]See function `TransferNet()` in the `models.py` file
[13]See function `kfold_cross_validation()` in the `train.py` file
[14]See class `CustomEarlyStopping` in the `train.py` file
[15]URL: https://scikit-learn.org

The intuition is that this kind of evaluation should resemble the real deployment of the model: it is trained on MWI, for which we have the ground truth labels; the Smart-I dataset, although being very different in terms of image characteristics, it is close to what the BlindTestSet consists of, but also having the labels can be used to understand how the model would behave on the latter.

For the `EmaNet` model the following values have been explored:

Parameter	**Set of values**
convolutions	2, 3
dense_layers	3, 5
filters	32
dropout rate	0.2, 0.3

Table 1: Parameter tuning for the `EmaNet` model

For the `TransferNet` model, the the following values have been explored:

Parameter	**Set of values**
feature extractor	Dense201, NASNetMobile
dense_layers	3, 5
dropout rate	0.2, 0.3

Table 2: Parameter tuning for the `TransferNet` model

Both models have undergone the parameter tuning phase via grid parameter search[16]; as previously stated, for each set of parameters, K-fold cross-validation is performed on the models. The grid parameter search has leveraged a 5-fold CV with each iteration using 100 epochs as maximum training epochs.

The best performing `EmaNet` model has achieved 37.5% mean accuracy on Smart-I using 2 convolutional layers, 3 dense layers, and 0.2 as dropout rate.
The best performing `TransferNet` has achieved 33% mean accuracy on Smart-I using `NASNetMobile` as feature extractor, 5 dense layers, and 0.3 as dropout rate.

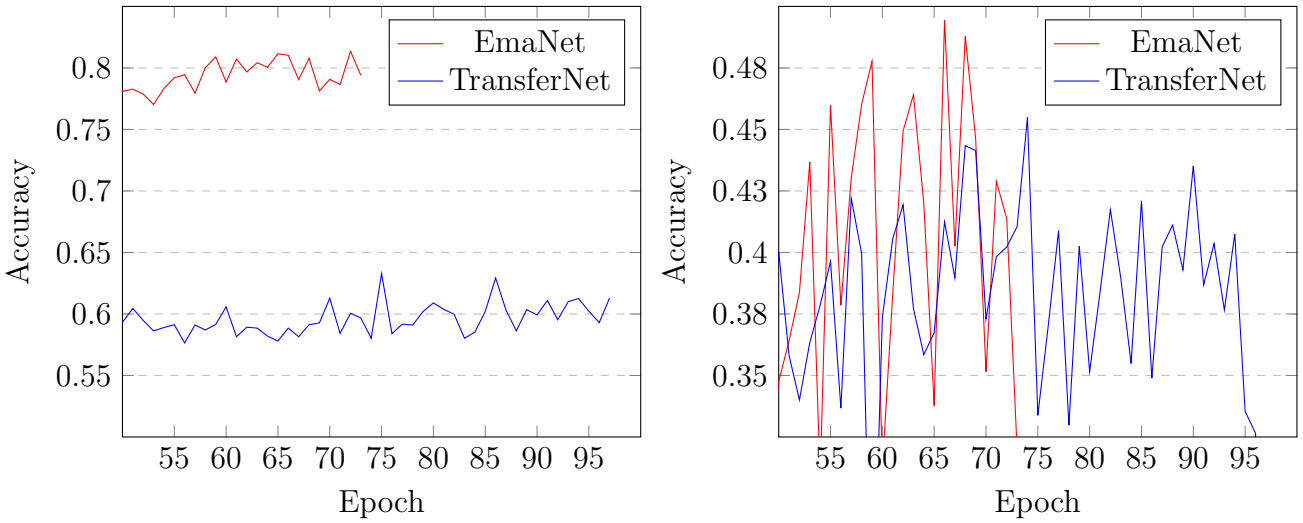
The simple `EmaNet` architecture has consistently outperformed `TransferNet` models using `DenseNet` as feature extractor, which ranks amongst the lowest scoring models; indeed 3/4 `TransferNet` models exploiting `NASNetMobile` perform better than its `DenseNet` counterparts, while still being part of the top-6 models. This might point out that `DenseNet`-based architectures should be more complex, but per the homework's target (embedded sensors), they might be too much.

Therefore, the training phase has been carried out the `EmaNet` and `TransferNet` models using their best performing parameters.

[16]See function `grid_search()` in the `train.py` file

6 Training

The chosen models have been trained on the MWI-4000 dataset with data augmentation techniques (as stated in Section 3) for a maximum of 150 epochs, using Smart-I as validation set and implementing a custom early stopping strategy monitoring the validation set accuracy: 10-epoch patience, but only after having trained for at least 1/3 of the maximum number of epochs.
In case of training end due to early stopping, the weights achieving the highest validation set accuracy are restored into the model.



(a) Training set (MWI-4000)

(b) Validation set (Smart-I)

Figure 3: Training accuracy over the training epochs.

As depicted in Figure 3, `EmaNet` reaches around 80% accuracy over MWI-4000 in just 74 epochs, while `TransferNet` stops training in the 98-th epoch with a 61% accuracy.
When evaluating the models over the validation set (Figure 3b), `EmaNet` proves to be the best performing model among the two, beating `TransferNet`'s 42% accuracy by more than 3 percent points, reaching 48.7% accuracy.

7 Performance evaluation

The two models' performances were further inspected on the Smart-I dataset by looking at the actual predictions instead of just considering the mean accuracy[17]. The following results have been collected after the training phases had stopped by the early stopping strategy and the models' best weights being loaded back into them.
Confusion matrices have been computed using SciKit-Learn's `confusion_matrix()` function[18].

7.1 EmaNet

The `EmaNet` model found the most problems when classifying RAINY images, mistakenly predicting SUNNY or SNOWY instead; while the rest of the classes have been correctly classified most of the times, an interesting misclassification happens for SUNNY images being predicted

[17] See function `static_evaluation()` in the `predict.py` file
[18] URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

as HAZE ones. More details in Table 3: values in **bold** are the number of correctly classified images, values in *italic* are the number of interesting misclassified images.
`EmaNet`'s overall accuracy is 46%.

Ground truth \ Predictions	HAZE	RAINY	SNOWY	SUNNY
HAZE	**0**	0	0	0
RAINY	81	**122**	124	*194*
SNOWY	90	232	**794**	*305*
SUNNY	*211*	193	198	**494**

Table 3: Confusion matrix for the `EmaNet` model

7.2 TransferNet

The `TransferNet` model found SUNNY images to be the most challenging, mistakenly predicting RAINY for them. It also has severe issues when presented SNOWY pictures, for which it predicted RAINY in too many cases. It performed better than `EmaNet` on RAINY images, and in general it didn't wrongly predict HAZE as many times as the other model, probably due to the pre-trained nature of the feature extractor: while `EmaNet` expected Smart-I to be balanced as well as MWI-4000 is, the `ImageNet` training of `NASNetMobile` hints as making `TransferNet` more robust to unbalanced test sets than `EmaNet` is. More details in Table 4 (same meaning for values in **bold** and *italic* as before).
`TransferNet`'s overall accuracy is 40%.

Ground truth \ Predictions	HAZE	RAINY	SNOWY	SUNNY
HAZE	**0**	0	0	0
RAINY	38	**285**	108	90
SNOWY	124	*477*	**591**	229
SUNNY	25	*547*	178	**346**

Table 4: Confusion matrix for the `TransferNet` model

8 Conclusions

The `EmaNet` model proved to be the best performing model both on MWI-4000 and, most importantly, on Smart-I, despite being the simplest of the two models and not leveraging on the supposed benefits of transfer learning.
Further improvements could be done by using one or multiple feature extractors in it, expanding the architecture with enough parallel stacks when the input image is fed into the model, and then joining feature maps right before the fully-connected layers.