Machine Learning Homework 2
Report

Andrea Caciolai
1762906

December 15, 2019

Contents

1 Introduction

For the second homework of the Machine Learning course, we have been tasked with the problem of classifying weather conditions in images by means of Convolutional Neural Networks (*CNN*s in the following).

In particular, the problem of classification is to learn a function

$$f : Images \to \{Haze, Rainy, Snowy, Sunny\} \tag{1}$$

where $Images$ is the set of tensors representing images: each image is represented as a matrix n by m of pixels, each pixel consisting of three channels, each channel consisting of an integer number between 1 and 255, so a tensor with dimension $(n, m, 3)$, in which each element lies in $\{1, \dots, 255\}$.

To learn this function we had to use *CNN*s, which are a particular type of neural network, nonlinear function approximators which have proved to be especially good at solving classification problems involving images.

Since we are working in a supervised learning setting, we had at our disposal a labeled dataset

$$D = \{(Image_i, t_i)_{i=1}^{n}\} \tag{2}$$

which was the "*Multi Weather Image Dataset*" containing four thousands images (so $n = 4 \cdot 10^3$) obtained from different sources, of different resolutions, featuring one of the four classes of weather conditions.

In solving this homework, I mainly relied on the Python libraries *Tensorflow* and *Keras*, similar to the exercises shown in class.

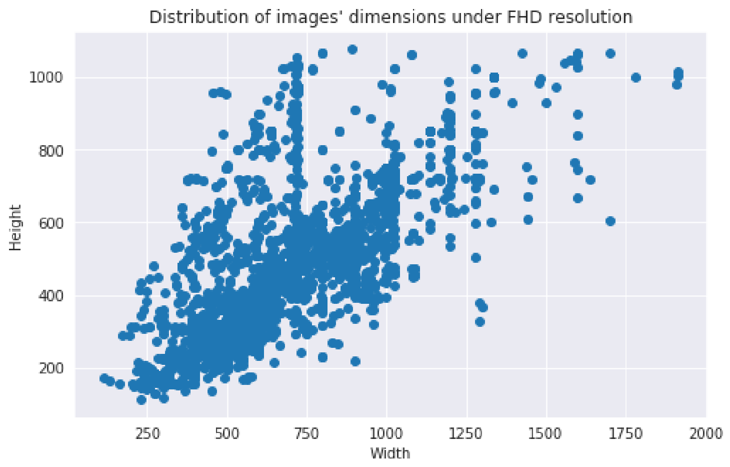
2 Input representation

One of the first choices to be made has been how to represent images of different dimensions. This problem arises from the fact that the images all have different shape, while the network expects inputs of consistent shape.

The *ImageDataGenerator* class from *Keras* comes into handy, since it offers the possibility to reshape every image in a given dataset into a target shape. It is also very useful since it can apply small transformations to the images of the dataset, like a small rotation, zoom, and so on. This is a nice trick that allows the network to be trained on a more diverse dataset, allowing it to recognize features in the image even if they are "placed" in slightly different parts of it, decreasing the overfitting on the specific images.

Furthermore, it does so without having to load all the images of the dataset into the memory at once, but instead by loading and applying the transformations to batches of images at a time.

So to choose the target shape, I studied the distribution of the shapes of the images in the *MWI* dataset, which I report below.



From the distribution above it can be seen that the mean of the ratio $\frac{width}{heigth}$ is approximately 1.31. Since the generator will "fill" the images in order to make them fit into the desired shape, keeping this ratio will result in the least amount of "filling", keeping the images untouched (except for the resizing, of course) as much as possible.

I kept the value 224 for the width as in exercise *MLEx10*, and by keeping the aforementioned ratio I got a value for the height of 172, so the target shape for the generator (and thus the input shape to the networks I engineered) is $(172, 224)$.

3 Custom CNN

In designing my own *CNN* I drew inspiration from the implementation of the *CNN* called "*AlexNet*" we saw in class. For lack of a better name, I called this network "*AndreaNet*".

3.1 Design

I took the following features of *AlexNet* as they are:

- the optimizer: I kept *Adam* with learning rate 10^{-4};
- the type of padding of the kernels: I kept the *valid* padding;
- the activation functions: I kept *ReLU* activation functions for the convolutional layers and the dense layers alike, with *Softmax* as the activation function of the last output layer, since we are interested in a probability distribution;
- the *l2* regularization term of 10^{-4} for the dense layers, in an attempt to reduce overfitting.

I made the following changes to *AlexNet* in order to build *AndreaNet*:

- the structure: I added a convolutional layer, so my network has 6 convolutional layers followed by 3 dense layers and the output layer;
- the size of kernels: I adopted only $(1, 1)$ and $(3, 3)$ kernels, with the former used in the first and last convolutional layer, and the latter used in the middle ones;
- the number of kernels: I organized the six layers in a specular and increasing fashion, which means the first three layers have $256, 512, 1024$ kernels, respectively, and the last three layers have the same, but in the reverse order;
- the pooling: I used MaxPooling after every convolutional layer and before every activation function, with the exception of the last convolutional layer, after which I placed an AveragePooling layer
- the number of units of the dense layers: I chose to have $4096, 2048, 1024$ units for the three dense layers, respectively;
- the dropout: I increased the dropout rate from 0.4 to 0.5, in an attempt to further decrease overfitting. I found this value to be the optimal one.

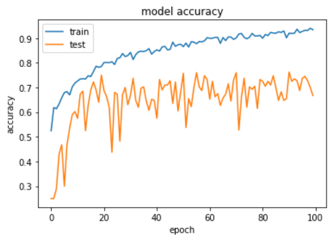
This architecture resulted in a network with $32, 805, 124$ trainable parameters.

3.2 Evaluation

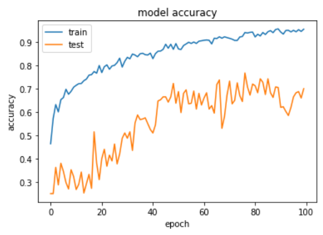
This has been the first attempt for the homework, and I decided to tread carefully, so I trained the network first on a smaller dataset, as suggested by Professor Iocchi, then used the whole *MWI* dataset.

I evaluated the network performance after training using cross validation, so using yet another feature of the *ImageDataGenerator* class, which lets you split the input data into *training* and *validation* subsets. I chose a ratio of 0.2 for the validation split.

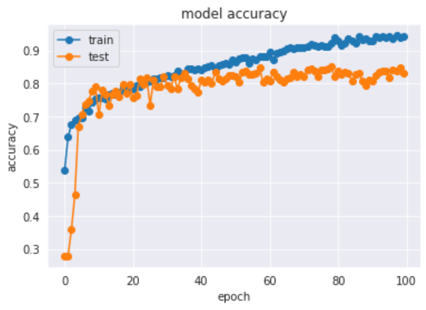
In the following I report the results of the evaluation of the network in three instances: after being trained for 100 epochs on the *MWI-1.1* subset, after being trained for 100 epochs on the *MWI-1.2* subset, and fter being trained yet again for 100 epochs on the whole *MWI* dataset.



(a) MWI-1.1



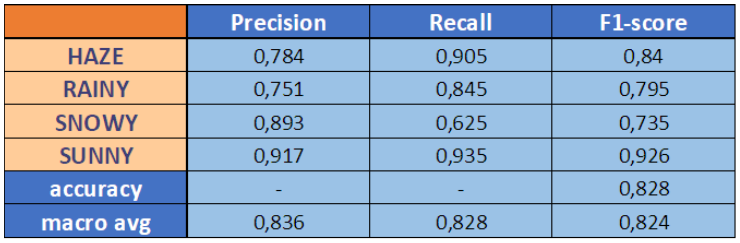
(b) MWI-1.2



(c) Whole MWI

Figure 1: Training history of the network, with the curves representing the model's performance in terms of accuracy of prediction, over both the training and validation set

We can see a sharp increase in the validation accuracy in the bottom graph, which I explained by considering the fact that having more samples to be trained on allows the network to generalize more.



	Precision	Recall	F1-score
HAZE	0,784	0,905	0,84
RAINY	0,751	0,845	0,795
SNOWY	0,893	0,625	0,735
SUNNY	0,917	0,935	0,926
accuracy	-	-	0,828
macro avg	0,836	0,828	0,824

Figure 2: Classification report

From the classification report above we can see that the model, trained over all the *MWI* dataset, scored a 82.8% of accuracy, with a precision, recall, and f1-score macro average of respectively 83.6%, 82.8%, 82.4%.

The class being classified correctly most often is *Sunny*, with a performance of 91.7%, 93.5%, 92.6% for the three metrics. This does not come as a surprise, since sunny images feature generally brighter and more vivid colours, while the distinctions among the other three is not so clear. The worst class from this perspective is *Snowy*, with a performance of 89.3%, 62.5%, 73.5%.

The really low recall is visible also from the confusion report, which I am not reporting here for brevity, which says that *Snowy* is the most confused class, being misclassifed as *Rainy* 5.50% of the time and as *Haze* 3.38% of the time.

4 Transfer Learning

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second, related, task.

The intuition behind transfer learning is that if a model is trained on a large and general enough dataset, this model will effectively serve as a generic model, since it will learn generally useful and informative feature maps, not only for the specific initial task. You can then take advantage of these learned feature maps without having to start from scratch training a large model.

In this case, the large dataset is the *ImageNet* dataset, featuring $14M$ images; the source task is unknown since I have used pre-trained networks available from the *keras.applications* library; the destination task is, of course, classifying weather conditions.

In particular, I decided to try out two very different networks: *VGG16*, with $138,357,544$ parameters organized in 23 layers; and *InceptionResNetV2* (*ResNet* in the following), featuring $55,873,736$ organized in a stunning 572 layers.[1]

There are two ways in which one can make use of the aforementioned feature maps that the base, pre-trained model supposedly has learned already, and both involve removing the last, dense layers of the network since those contain parameters specialized in the specific classification problem at hand during the original training of the base model.

In the following, I will describe both approaches, since I experimented with both.

4.1 Feature extraction for fully connected network

This approach consists in attaching to the convolutional part of the base model a fully connected part, so a certain number of dense layers whose purpose is to classify the incoming samples based on the features the previous convolutional part "extracts" from the samples.

4.1.1 Design

At first, I experimented with shallow networks, which means a network with just an output layer (with 4 units, of course) right after the convolutional part of the pre-trained network. So I built models with this architecture, starting from both *VGG16* and *ResNet*. They turned out to be surprisingly good, with few parameters to be trained ($4 \cdot \#$features extracted by the base *CNN*).

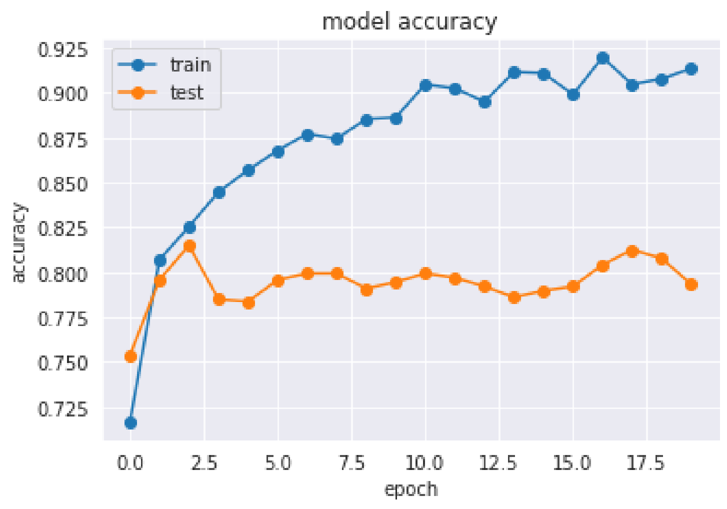
Then, I tried adding a couple of dense layers, with a moderate number of units, otherwise, the whole transfer learning process would be pointless in terms of decrease of learning effort, resulting in models with more parameters to be learned than the respective base ones had to begin with. In particular, I added two dense layers with 256 and 128 units, respectively, before the output layer.

In the following, I report the results obtained for both networks, with both approaches, for a total of four different experiments.

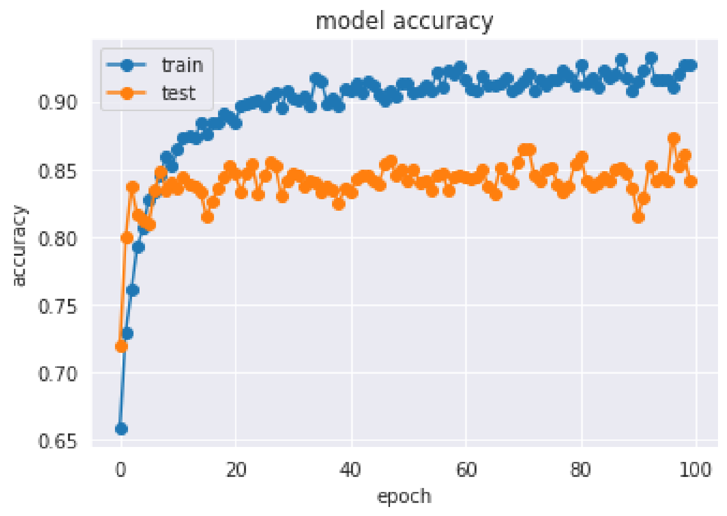
[1]Data taken directly from Keras documentation at `https://keras.io/applications/`

4.1.2 Evaluation

VGG16 In the following graphs, we see the training history for the models resulting from the two different approaches described above, applied to the pre-trained network *VGG16*.



(a) Shallow model



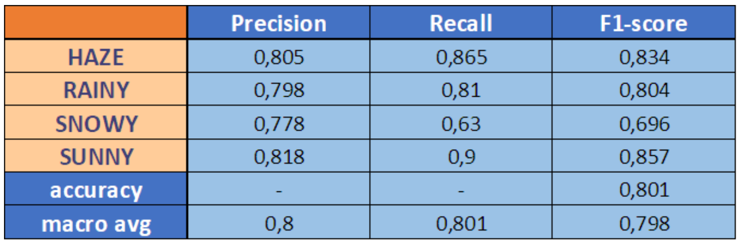
(b) Deeper model

Figure 3: Training history of models based on VGG16

For the former, I limited the training at 20 epochs, since there were only 71, 684 parameters to be trained (resulting from having 4 neurons in the output layer and an incoming flattened feature vector of dimension $17,920$, so $4 \cdot 17,920$ for the linear weights and 4 parameters for the bias terms).

For the latter, I let the training phase proceed for the full 100 epochs, as with my own *CNN*, as described in Section 3. The resulting model had $4,657,796$ trainable parameters, and both the former and the latter model had $14,751,296$ non-trainable parameters, resulting from keeping the weights of the convolutional part of *VGG16* as they are.

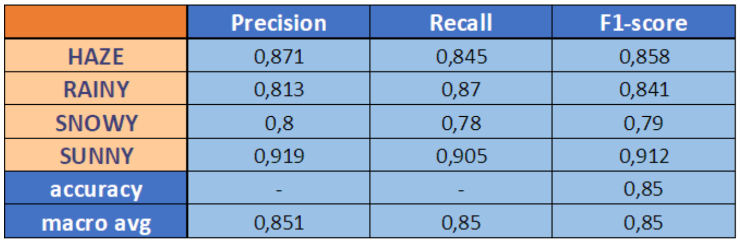
In both cases, it can be seen from the validation curves that I could have stopped the training much earlier, since they start oscillating around their definitive validation accuracy pretty soon, compared to the horizon of training.



	Precision	Recall	F1-score
HAZE	0,805	0,865	0,834
RAINY	0,798	0,81	0,804
SNOWY	0,778	0,63	0,696
SUNNY	0,818	0,9	0,857
accuracy	-	-	0,801
macro avg	0,8	0,801	0,798

Figure 4: Classification report of shallow network based on *VGG16*

The first model scored a decent 80.0%, 80.1%, 79.8% of macro average in the three evaluation metrics when evaluated with cross-validation over the *MWI* dataset, which has been used for the training too.



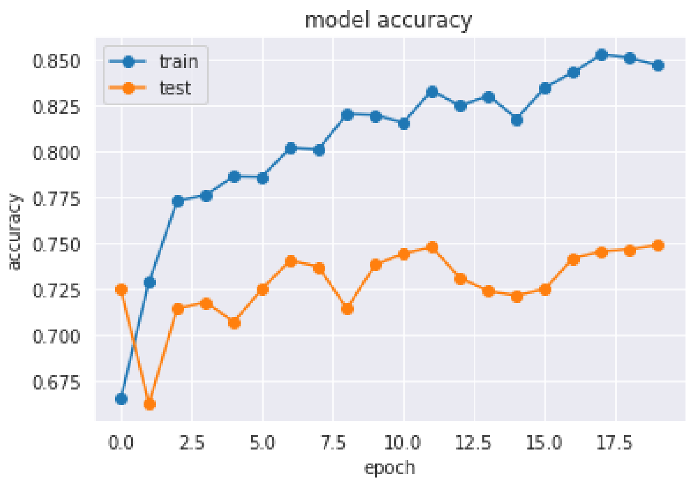
	Precision	Recall	F1-score
HAZE	0,871	0,845	0,858
RAINY	0,813	0,87	0,841
SNOWY	0,8	0,78	0,79
SUNNY	0,919	0,905	0,912
accuracy	-	-	0,85
macro avg	0,851	0,85	0,85

Figure 5: Classification report of deeper network based on *VGG16*

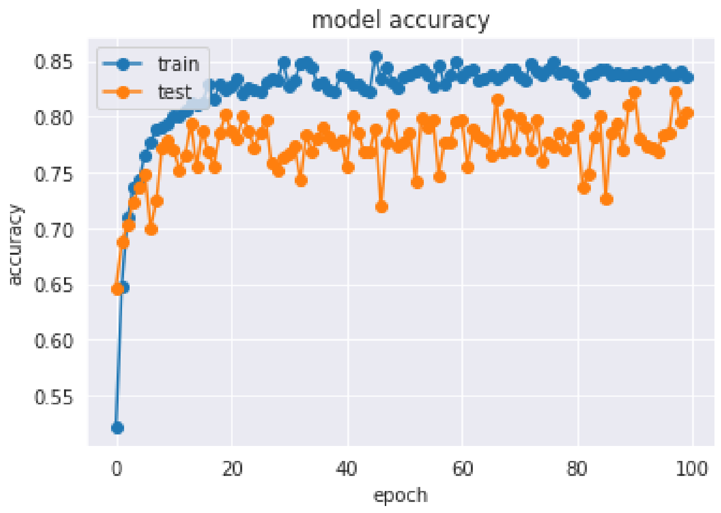
The second model scored a very good 85.1%, 85.0%, 85.0% in the same metrics with the same evaluation procedure.

Both models confirmed the misclassification trends that manifested in the evaluation of my custom *CNN* (see Section 3.2), with *Sunny* being the most correctly classified class, with 85.7% and 91.2% in f1-score, respectively, and also *Snowy* being the least correctly classified class, with 69.6% and 79.0% in f1-score, respectively.

ResNet In the following graphs, we see the training history for the models resulting from the two different approaches described above, applied to the pre-trained network *ResNet*.



(a) Shallow model



(b) Deeper model

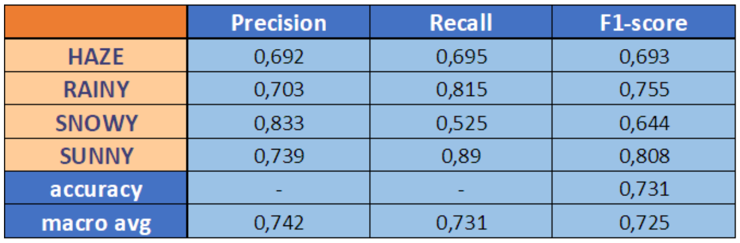
Figure 6: Training history of models based on ResNet

Again, in both cases, if I had implemented early stopping the training would have not kept on going for this long, especially for the latter.

For the former, I limited the training at 20 epochs, since keeping on with training would have yielded no further increase in validation accuracy. In this case, there were 122, 884 parameters to be trained, so close to double the parameters of the shallow network based on *VGG16*.

Also again, for the latter, I let the training phase proceed for the full 100 epochs. The resulting model had 7, 960, 196 trainable parameters, so again something less than double the number of parameters of the deeper network based on *VGG16*.

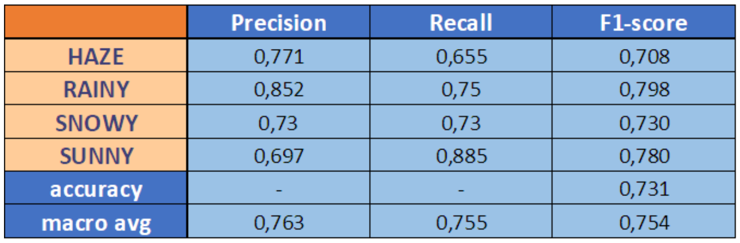
Both the former and the latter model had 54, 398, 944 non-trainable parameters, resulting from keeping the weights of the convolutional part of *ResNet* as they are, almost four times as many as the ones of the previous *VGG16*-based models. The big complexity of the base model, in this case, resulted in a disadvantage more than an advantage, as the results show.



	Precision	Recall	F1-score
HAZE	0,692	0,695	0,693
RAINY	0,703	0,815	0,755
SNOWY	0,833	0,525	0,644
SUNNY	0,739	0,89	0,808
accuracy	-	-	0,731
macro avg	0,742	0,731	0,725

Figure 7: Classification report of shallow network based on *ResNet*

The first model scored a disappointing 74.2%, 73.1%, 72.5% in the three macro average of the evaluation metrics when evaluated with cross-validation over the *MWI* dataset, which has been used for the training too.



	Precision	Recall	F1-score
HAZE	0,771	0,655	0,708
RAINY	0,852	0,75	0,798
SNOWY	0,73	0,73	0,730
SUNNY	0,697	0,885	0,780
accuracy	-	-	0,731
macro avg	0,763	0,755	0,754

Figure 8: Classification report of deeper network based on *ResNet*

The second model scored a still disappointing 76.3%, 75.5%, 75.4% in the same metrics with the same evaluation procedure.

Basically, there has been no point in choosing a more complex base model, because the shallow network based on the “simple” *VGG16* outperformed the deeper one based on the very “complex” and deep *ResNet*. My explanation to this result is that the fully connected part is not “powerful” enough to capitalize on the fine feature extraction that *ResNet* has done on the samples.

Interestingly enough, The misclassification trends changed in the second model, while remaining the same for the first. For the second model it was

Rainy the most correctly classified class, with 79.8% of f1-score, stealing the throne of *Sunny*, stuck at 78.0%. The very low precision of *Sunny* emerges also in the confusion report, which says that *Sunny* is the first and third most confused for class. Also the throne for least correctly classified class was different for the second model, with *Haze* stealing it from *Snowy*, with the former having an f1-score of 70.8% compared to the one of the latter being 73.0%.

4.2 Feature extraction for linear classifier

This second approach consists in exploiting the convolutional part of the base pre-trained model as a literal feature extractor black box, that given an image will output a vector of features that it "extracted" from the image.

We hope that those features will be relevant and helpful in classifying correctly the sample.

4.2.1 Design

The strategy has been to use the *ImageDataGenerators* over the whole *MWI* dataset, applying no transformation whatsoever apart from the rescaling (normalizing each element of the image tensor in $[0, 1]$), to obtain the feature vectors of all its samples in order to build a "transformed" dataset.

This dataset, split into training and testing set as per usual, has then been used to train linear classifiers of the kind we covered in the first part of the course; in particular *Support Vector Machine* and *Logistic Regression*.

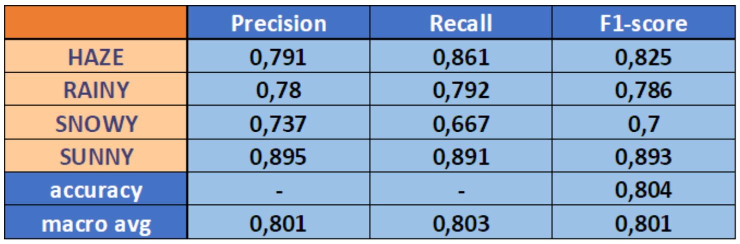
This has been done for both the pre-trained *CNN*s I have been using in this transfer learning experimentation: *VGG16* and *ResNet*.

The classifiers have been implemented through the library *Scikit-learn*, and for the training I used the *GridSearchCV* method, so k-fold cross-validation (with $k = 4$) and with default parameters of $C = 1$ and linear kernel. So for each approach, 4 different models have been fit, and this required more time than I expected (approximately half an hour for each of the models, so I would say ≈ 8 minutes for each fit). I think this is normal though, since the feature vectors that *VGG16* outputs are of dimension $21,504$, while the ones outputted by *ResNet* are even larger, with dimension of $36,864$, both way larger than the ones I have been used to so far when working with linear classifiers (like in Homework 1).

4.2.2 Evaluation

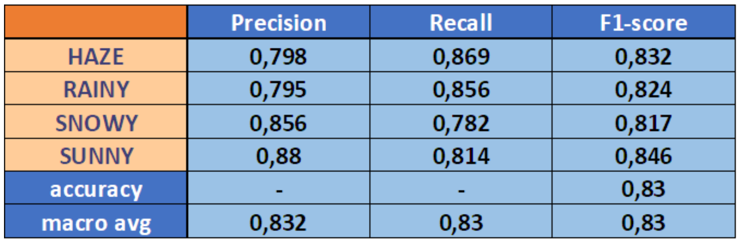
In the following, I will present the results obtained from the evaluation of both linear classifiers on both the networks, thus of the four different combinations. The results are evaluated on the test set, through the standard *precision, recall, f1-score, accuracy* metrics.

Here are the performances of the *SVM* classifiers.



	Precision	Recall	F1-score
HAZE	0,791	0,861	0,825
RAINY	0,78	0,792	0,786
SNOWY	0,737	0,667	0,7
SUNNY	0,895	0,891	0,893
accuracy	-	-	0,804
macro avg	0,801	0,803	0,801

(a) *VGG16*

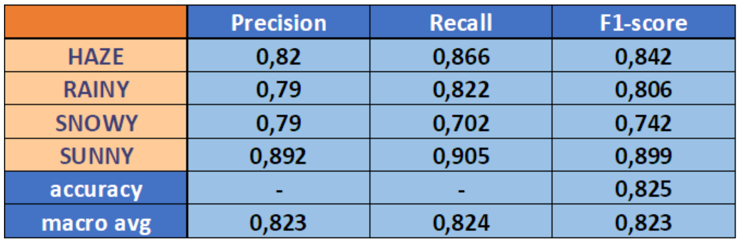


	Precision	Recall	F1-score
HAZE	0,798	0,869	0,832
RAINY	0,795	0,856	0,824
SNOWY	0,856	0,782	0,817
SUNNY	0,88	0,814	0,846
accuracy	-	-	0,83
macro avg	0,832	0,83	0,83

(b) *ResNet*

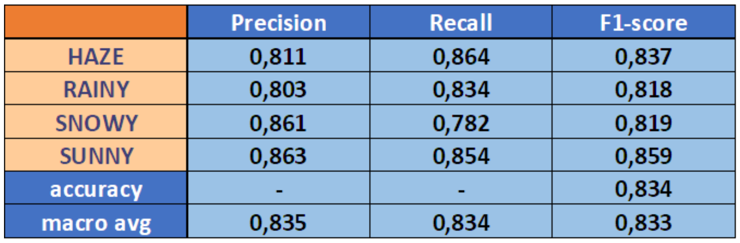
Figure 9: Classification report of SVM classifier on dataset extracted by the two pre-trained *CNN*s

Here are the performances of the *Logistic Regression* classifiers.



	Precision	Recall	F1-score
HAZE	0,82	0,866	0,842
RAINY	0,79	0,822	0,806
SNOWY	0,79	0,702	0,742
SUNNY	0,892	0,905	0,899
accuracy	-	-	0,825
macro avg	0,823	0,824	0,823

(a) *VGG16*



	Precision	Recall	F1-score
HAZE	0,811	0,864	0,837
RAINY	0,803	0,834	0,818
SNOWY	0,861	0,782	0,819
SUNNY	0,863	0,854	0,859
accuracy	-	-	0,834
macro avg	0,835	0,834	0,833

(b) *ResNet*

Figure 10: Classification report of Logistic Regression classifier on dataset extracted by the two pre-trained *CNN*s

I think several things can be noticed in these results.
The first is that they are indeed very good results, which I did not expect; that must mean that the base models are doing a good job at extracting useful, (almost) linearly separable data points.

Also, in this case, the classifiers based on *ResNet* outperformed the ones based on *VGG16* as opposed to what happened for the transfer learning with dense layers (as I explained in Section 4.1).

Then the fact that the Logistic Regression classifiers in both cases scored higher than SVM classifiers, which is again something I did not anticipate, since to my (very limited) experience the latter tend to perform better than the former; maybe doing some fine-tuning would have helped, but I believe that goes beyond the scope of this homework.

A more subtle but very interesting fact is that there is much less variance among the classes in terms of misclassified samples: we can see $f1$-scores very similar among the 4 classes, for both base models and for both linear classifiers, as opposed to all classifiers with an architecture based exclusively on neural networks. So these models will predict *Sunny* correctly less often but also predict *Snowy* correctly more often.

5 Conclusions

Among all the approaches that I tried, the one that yielded the best results, as explained above, has been transfer learning with *VGG16* and two dense layers. I did not expect transfer learning to behave so well in general, compared to a network trained entirely for the specific classification task at hand.

It is worth noting that every single model I experimented with performed extremely poorly on the other dataset we have been provided by *SMART-I*. My interpretation is that this is to be attributed to the very diverse nature of the images in the two datasets: the *MWI* dataset contains a very diverse collection of images, featuring weather displayed in natural landscapes, city roads, and so on; on the other hand, the *SMART-I* contains only snapshots of videos recorded by *CCTV*s on roads, capturing mainly the street with a very particular angle that the network has not seen in the training set, and also capturing a very little portion of the sky. In addition to that, I have noticed that a lot of pictures are consecutive frames of the same recording, having thus the same label; this means that a wrong prediction on one of those samples almost certainly leads to wrong predictions on all the others too. Last but not least, since it only contains three classes (*Haze* was missing) all the samples predicted as *Haze* were mistakes. I tried to dampen this effect by "dropping" all the probabilities the models assigned to the *Haze* class and rescaling the predictions vector, achieving higher but still very poor accuracy ($\approx 50\%$).

I tried (just for "fun") to train a couple of models on the *SMART-I* dataset, and evaluating them again with cross-validation led to accuracies comparable to the ones the same models scored when trained on *MWI* dataset. Unfortunately, while it is possible to "drop" a prediction for the *Haze* class, it is not possible to "create" one out of nowhere, so it was not possible to test the opposite of what I did, that is evaluating the models trained on the *SMART-I* dataset on the *WMI* dataset. I could have dropped all the samples labelled as *Haze* and then do the evaluation, but then I thought this "just for fun" experimentation was going a little too far, and decided to let it be.

Going back on topic, given the very good performance of the transfer learning models, I chose the best performing model among those, which is the *VGG16*-based one as my "best" model, and also as the one to use to compute the predictions on the blind test set.

So much more could have been tried for this homework, like fine-tuning the several hyperparameters of the "best" model, or experimenting with more pre-trained models, or with more linear classifiers, or with more articulate custom-made *CNN*s, but still, I feel that this work has been very good for me to familiarize with the practical aspects of problem-solving via convolutional neural networks.