Homework2 - Machine Learning

Longo Valerio

December 2019

Contents

1 Pre-processing the data **2**
1.1 Management of the input 2
1.2 The Pre-vectorization 2
1.3 The effect of the resolution 3

2 The model **4**
2.1 Part 1 - The custom CNN 5
2.2 Part 2 - Transfer Learning using Fine Tuning 6

3 Testing the best models **8**

4 Output analysis **10**

Overview

The goals of this homework were:

- define a CNN and train it from scratch
- apply transfer learning and fine tuning from a pre-trained model.

The report is divided in several parts. The first one will illustrate the preprocessing phase: the strategy that i used to manipulate the data and the reasons behind that. Moreover i tested different shapes for the input in order to see how this affected the performances.

The second part will show the models used to reach the goals of the homework reporting the performances of various networks, in order to choose one best model for each goal. In this part, the evaluations are made using half MWI-Dataset as training and half *TestSet_Weather* as test set.

The third part will compare the best model built from scratch and the best one built with transfer learning in order to decree which one is the best to test the blind set. Here i used all the MWI-Dataset and all the *TestSet_Weather*.

The last part will evaluate the best model found in the previous section to the blind set and will be shown the results.

1 Pre-processing the data

All the reads and manipulations of the dataset are done using *OpenCV* library. Due to the fact that the images must be reshaped, i had to choose between storing all of them in primary or in secondary memory. I decided the first one because i tested the nets with different reshape methods in order to get the best one.

1.1 Management of the input

The images are grouped in four folders. So, at first i build *DB_path* and *ClassArray* with the labels.

```
DB_path = '/content/drive/MyDrive/DatasetHW2/MWI-Dataset'
ClassArray = [ '/HAZE', '/RAINY', '/SNOWY', '/SUNNY']
```

1.2 The Pre-vectorization

After the extraction of the zip files, i built *X_path* containing the paths of all the images, and *Y_all* containing the respective labels.

```
X_path = []
Y_all = []

num_classes = len(ClassArray)
```

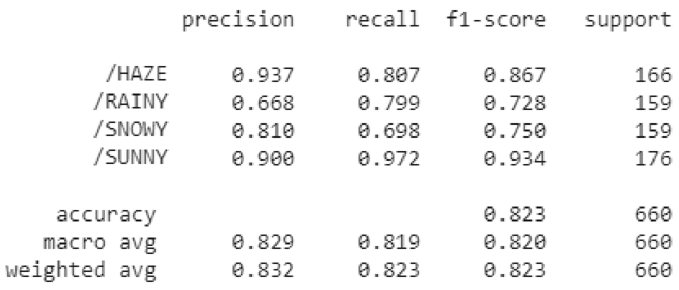
```
    for i in range(len(ClassArray)):
      for j in os.listdir(DB_path+ClassArray[i]):
        Xtrain = [DB_path+ClassArray[i]+'/{}'.format(j)]
        X_path.append(Xtrain)
        Y_all.append([i])
    X_path = np.array(X_path).flatten()
```

Then, as said before, i stored all the images in a single variable.

```
X_all = []

for imges in X_path:
  imges = cv2.imread(imges, cv2.IMREAD_UNCHANGED)
  if len(imges[0][0]) > 3:
        imges = cv2.cvtColor(imges, cv2.COLOR_BGRA2BGR) #normalizing to 3 channel

  X_all.append(cv2.resize(
        imges, (img_resized_res,img_resized_res), interpolation = cv2.INTER_AREA))

print("done ",len(X_all)," images")
```

1.3 The effect of the resolution

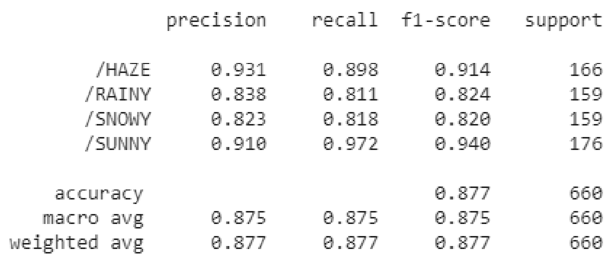
The first hyperparameter that i tried to change was the size of the images (*width x high x channel*). All these experiments have been done using the simple custom cnn without any regularization terms, a splitted training/validation set to build the model in order to get the classification report, 64 epochs and 32 as batch size. I started using 28x28x3 as *img_resized_res*, obtaining:



```
              precision    recall  f1-score   support

       /HAZE      0.937     0.807     0.867       166
      /RAINY      0.668     0.799     0.728       159
      /SNOWY      0.810     0.698     0.750       159
      /SUNNY      0.900     0.972     0.934       176

    accuracy                          0.823       660
   macro avg      0.829     0.819     0.820       660
weighted avg      0.832     0.823     0.823       660
```

Figure 1: classification report with 28x28x3

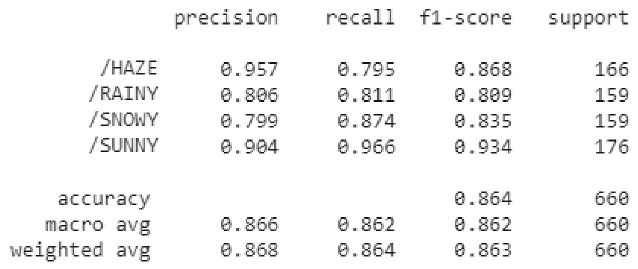
Then i continued using 150x150x3 as *img_resized_res*, obtaining:



```
              precision    recall  f1-score   support

       /HAZE      0.931     0.898     0.914       166
      /RAINY      0.838     0.811     0.824       159
      /SNOWY      0.823     0.818     0.820       159
      /SUNNY      0.910     0.972     0.940       176

    accuracy                          0.877       660
   macro avg      0.875     0.875     0.875       660
weighted avg      0.877     0.877     0.877       660
```

Figure 2: classification report with 150x150x3

As third attempt, i used 240x240x3 as *img_resized_res*:



```
              precision    recall  f1-score   support

       /HAZE      0.957     0.795     0.868       166
      /RAINY      0.806     0.811     0.809       159
      /SNOWY      0.799     0.874     0.835       159
      /SUNNY      0.904     0.966     0.934       176

    accuracy                          0.864       660
   macro avg      0.866     0.862     0.862       660
weighted avg      0.868     0.864     0.863       660
```

Figure 3: classification report with 240x240x3

As last experiment, i used a resolution of 480x480x3 but the system crashed due to the fact that too much memory was used.

From these initial results we can conclude that changing the resolution is not so indifferent with respect to the general results (we have a boost of about 5% in accuracy in 150x150x3 case with respect to the 28x28x3 experiment). Moreover, the usage of huge resolutions produced a big memory effort and lots of computational time than we can avoid considering that the accuracy is lightly changed (beeing even worse). So, for this reasons, i used **as default the 150x150x3 resolution**.

2 The model

To choose the right model in part 1 and part 2 i considered 64 epochs, 32 as size of batches half of the dataset MWI and half of the SMART-I dataset. After the simple vectorization made as follows,

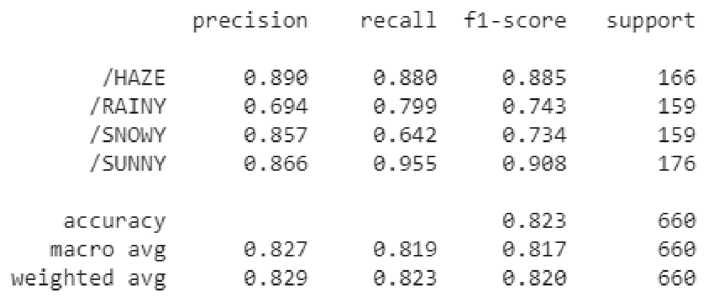
```
X_all = np.array(X_all)
Y_all = keras.utils.to_categorical(Y_all, 4)
input_shape =  (img_resized_res, img_resized_res, 3)
```

2.1 Part 1 - The custom CNN

The neural network built from scratch has this skeleton:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                input_shape=input_shape, padding = 'same'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',padding = 'same'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',padding = 'same'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',padding = 'same'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(num_classes, activation='sigmoid'))
model.compile(loss=keras.losses.categorical_crossentropy,
                optimizer='adam',
                metrics=['accuracy'])
```

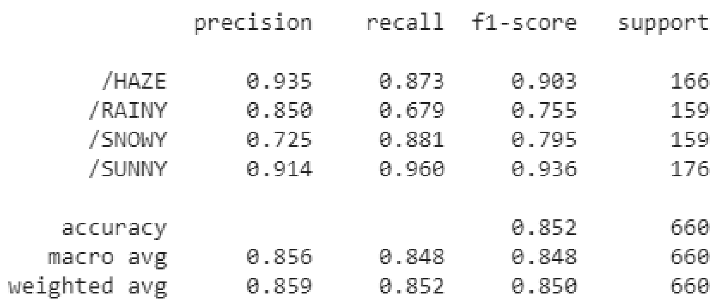
From the *Figure 2* of the previous section we can see that the accuracy is 0.87. So now let's try to change some hyperparameters. Adding some dropout on the last layer and changing to 5x5 kernels doesn't move too much the accuracy.



```
              precision    recall  f1-score   support

       /HAZE      0.890     0.880     0.885       166
      /RAINY      0.694     0.799     0.743       159
      /SNOWY      0.857     0.642     0.734       159
      /SUNNY      0.866     0.955     0.908       176

    accuracy                          0.823       660
   macro avg      0.827     0.819     0.817       660
weighted avg      0.829     0.823     0.820       660
```

Figure 4: Classification report with dropout and different kernels

Changing activation function of the hidden layers in *tanh* and Average pooling i got:



```
              precision    recall  f1-score   support

       /HAZE      0.935     0.873     0.903       166
      /RAINY      0.850     0.679     0.755       159
      /SNOWY      0.725     0.881     0.795       159
      /SUNNY      0.914     0.960     0.936       176

    accuracy                          0.852       660
   macro avg      0.856     0.848     0.848       660
weighted avg      0.859     0.852     0.850       660
```

Figure 5: Classification report with tanh and average pooling

So we conclude this part taking as best model the first one.

2.2 Part 2 - Transfer Learning using Fine Tuning

This is the skeleton of the model used with transfer learning:

```
#example with VGG16
prior = keras.applications.VGG16(include_top=False,
                                 weights='imagenet',input_shape=input_shape)

for layer in prior.layers[:-9]:
    layer.trainable = False

model = Sequential()
model.add(prior)
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(num_classes, activation='sigmoid'))

model.compile(loss=keras.losses.categorical_crossentropy,
            optimizer=Adam(lr=0.00001), metrics=['accuracy'])
```

Basically i take a "famous" network, i freeze its last layers and attach them to the last part of the custom cnn built by myself. due to the fact that transfer learning requires a low learning rate, i put it. I tested various networks. First i used VGG16 and got these results.

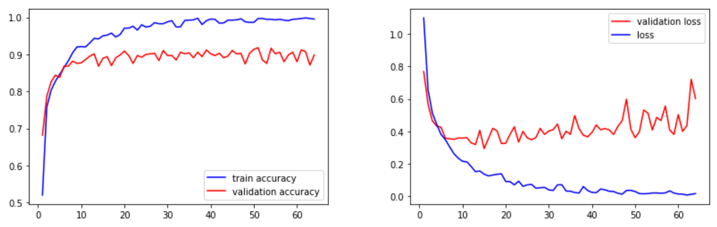
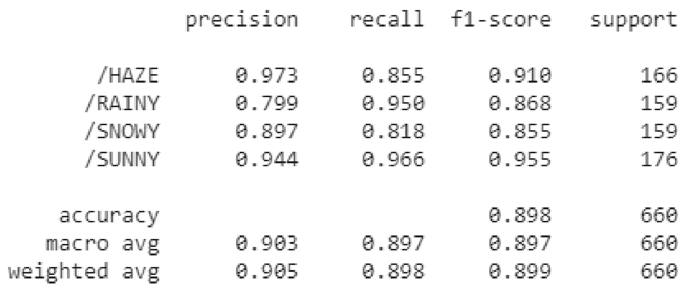


Figure 6: history of VGG16



```
              precision    recall  f1-score   support

       /HAZE      0.973     0.855     0.910       166
      /RAINY      0.799     0.950     0.868       159
      /SNOWY      0.897     0.818     0.855       159
      /SUNNY      0.944     0.966     0.955       176

    accuracy                          0.898       660
   macro avg      0.903     0.897     0.897       660
weighted avg      0.905     0.898     0.899       660
```

Figure 7: Classification report of VGG16

Then, i decided to try with the recent VGG19, with these results:

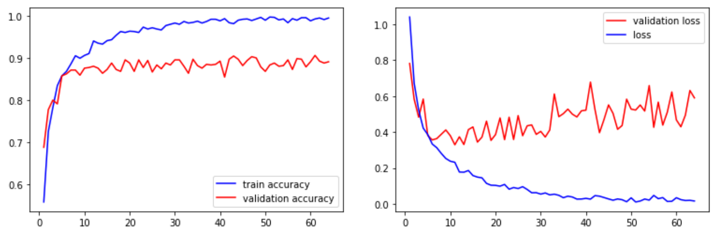
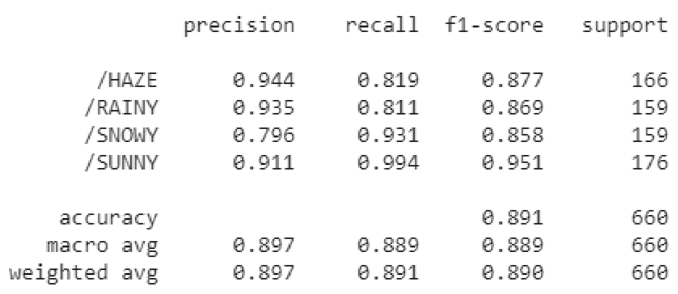


Figure 8: history of VGG19



```
              precision    recall  f1-score   support

       /HAZE      0.944     0.819     0.877       166
      /RAINY      0.935     0.811     0.869       159
      /SNOWY      0.796     0.931     0.858       159
      /SUNNY      0.911     0.994     0.951       176

    accuracy                          0.891       660
   macro avg      0.897     0.889     0.889       660
weighted avg      0.897     0.891     0.890       660
```

Figure 9: Classification report of VGG19

Given these results, i decided to use VGG16 as best model for this second part

3 Testing the best models

from the previous section we can say that the best two networks are the simple custom cnn and the VGG16. So, now that i got the best models for each part, i fitted them with: 64 epochs, 32 as batch size, all the *MWI-dataset* plus the *TestSet_Weather*. these are the results. Network built from scratch:

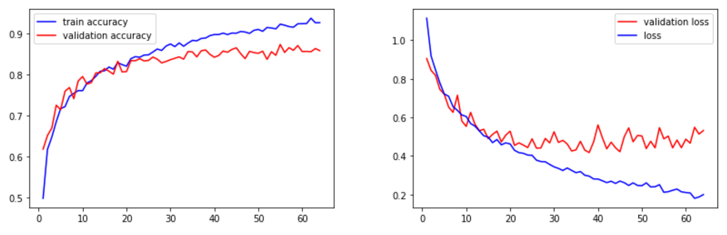
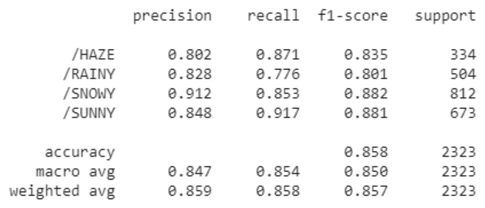


Figure 10: history of the custom CNN with the entire dataset



```
              precision    recall  f1-score   support

       /HAZE      0.802     0.871     0.835       334
      /RAINY      0.828     0.776     0.801       504
      /SNOWY      0.912     0.853     0.882       812
      /SUNNY      0.848     0.917     0.881       673

    accuracy                          0.858      2323
   macro avg      0.847     0.854     0.850      2323
weighted avg      0.859     0.858     0.857      2323
```

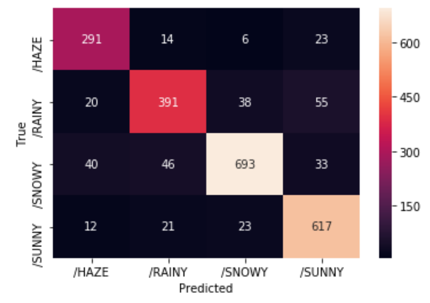


Figure 11: Classification report and confusion matrix of the custom CNN with the entire dataset

Transfer learning from VGG16 with fine tuning:

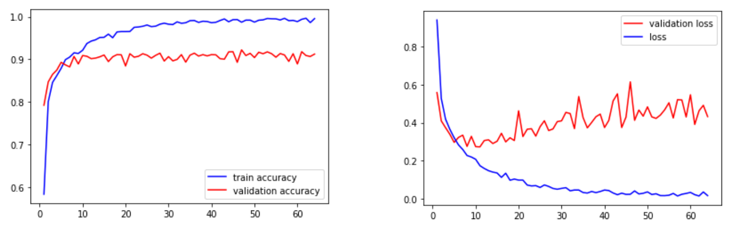
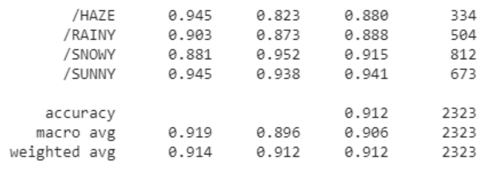


Figure 12: history of the VGG16 with the entire dataset



```
       /HAZE      0.945     0.823     0.880       334
      /RAINY      0.903     0.873     0.888       504
      /SNOWY      0.881     0.952     0.915       812
      /SUNNY      0.945     0.938     0.941       673

    accuracy                          0.912      2323
   macro avg      0.919     0.896     0.906      2323
weighted avg      0.914     0.912     0.912      2323
```

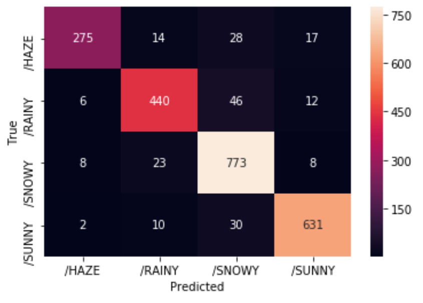


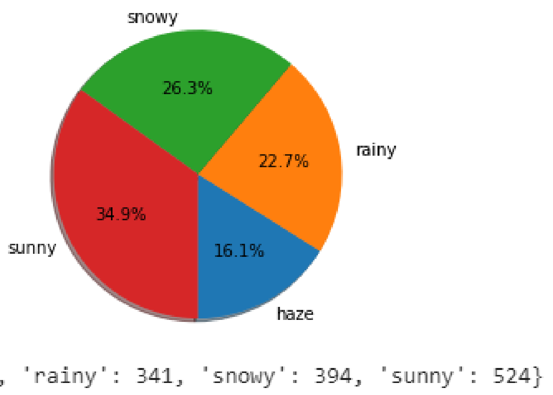
Figure 13: Classification report and confusion matrix of the VGG16 with the entire dataset

We can see that even if previously they had more or less the same accuracy, with the double of the data, **VGG16 model outperformed the one made from scratch. So i used it to test the blind set**

4 Output analysis

Now that we found the best model between the one tested, let's see the results from the blind set. Here the summary:

- **241 elements labeled as haze**
- **341 elements labeled as rainy**
- **394 elements labeled as snowy**
- **524 elements labeled as sunny**



```
{'haze': 241, 'rainy': 341, 'snowy': 394, 'sunny': 524}
```

Figure 14: Analysis of the blind set using my best model

So, as final conclusion we can say that the blind dataset is a little unbalanced, with less haze images with respect to sunny but with the same number of rainy and snowy photos.