Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Computer Vision Course

Weather Classification - Final Project

Student: Merlo Simone

Matricola: 2076747

Contents

Introduction

The aim of this project is to classify a set of images representing different weather conditions. We did it by exploiting images characteristics, such as color histograms, gradient, luminance and sharpness. In particular we created an algorithm that first extracts the necessary features from the images and then trains and tests a neural network (Multi Layer Perceptron classifier) based on these features.
Several dataset were provided and the proposed algorithm works with all of them, both independently and on some combinations.

Chapter 1

Algorithm, Code and Motivations

In this chapter we describe the basic structure of the code, we focus on the algorithm by deeply describing the feature extractor/descriptor components and eventually we explain design choices and motivations of the features composing the image descriptors. The goal of the proposed procedure is to classify a set of images corresponding to different weather conditions. The datasets on which the algorithm has been tested are:

- Multi-class Weather Dataset (MWD).
- ACDC.
- UAVid.
- SYNDRONE.

Some combinations of these datasets have also been used.

1.1 Code structure

The *featureextractor.py* file contains the class **Loader**, whose methods are used to compute an array of features for each image in the provided dataset. In particular, the class takes as input in the constructor the path and name of the dataset to consider and some other parameters (number of bins and some flags) that can be used to tune the image descriptors (see Section 1.2.1 to understand the behaviour of these parameters). Furthermore, the most important method is the *loadImages* method, that computes the actual features vector and returns a multidimensional array together with a one-dimensional

array: the first containing, on each row, an array describing a single image of the dataset and the second containing the class for each image.

The *weatherclassifier.py* file contains the class **WeatherClassifier**, whose methods are used classify the images. In particular, it takes as input in the constructor a multidmensional numpy array corresponding to the training set, a one-dimensional array corresponding to the classes of the training examples and some other parameters to set the random seeds (to achieve a deterministic behaviour accross different executions) and the maximum number of iterations of the Multi Layer Perceptron (MLP) classifier. Furthermore, the most important methods are the *trainClassifier* and the *score* methods: the first used to fit the MLP Classifier to the training data, the second used to test the classifier based on the dataset and labels provided as parameters.

The *weatherclassification.ipynb* file represents the python notebook containing the exeution of the algorithm. In particular, it tries to classify the images by evaluating different parameters and datasets combinations.

1.2 Algorithm

The proposed algorithm can be partitioned in three parts:

- Image description: converts each image into an array of features (see Section 1.2.1).
- Weather classification: it takes a set of image descriptors together with the corresponding labels and trains (and then tests) a MLP classifier (see Section 1.2.2).
- Execution and Testing: loads the datasets and tests the algorithm on different dataset combinations (see Section 1.2.3).

Each of these three parts corresponds to a different file (see Section 1.1 for more details).

The general pipeline of the proposed procedure is the following: the algorithm first loads the training images by computing an image descriptor and the corresponding label for each of them, then it loads in the same way also the test images. After that it trains a MLP Classifier using the training image descriptors (and labels). Eventually, it tests the classifier, obtaining a score, using the test images descriptors (and labels).

1.2.1 Image Description

To compute the descriptors the algorithm first load the images and then processes them by computing the histograms on each channel and, based on the flags, the average sharpness, the average gradient and the average luminance of the image. The histograms (that correspond to three arrays) and the other computed values are then concatenated to form a single array.

The image description is performed thanks to a class called **Loader**. In particular this class takes as parameter in the constructor the path and name of the dataset to consider, the number of bins and three boolean flags called use_gradient, use_sharpness and use_luminance.
The first two parameters are used to select the class labels corresponding to the dataset and to know the path of the source folder of the images.
The class has several relevant methods used to compute the descriptors:

- *getClass*
- *computeHistograms*
- *computeSharpness*
- *computeGradient*
- *computeLuminance*
- *getFeatures*
- *loadMWD*
- *loadOther*
- *loadImages*

The method *getClass* exploits the name of the image file (provided as an argument) and an array of strings computed in the constructor to return the class label of an image. In particular each element on the array of strings corresponds to a piece of text that must be contained in the image file's name for the image to belong to a specific class.

The method *computeHistograms* takes as input a multidimensional array representing a color image, computes the color histograms of the image quantizing the levels axis in a number of levels specified by the bin_num (number of bins) parmeter, which is provided in the constructor. The quantization can be performed for two main reasons:

- To reduce the number of features and thus the length of the image descriptors. This is useful also because we consider only 3 features other than the ones related to the histograms and reducing the number of histograms features allows to give more importance to those three features.

- To group color levels together, since images corresponding to the same weather condition tend to have similar but not perfectly matching colors.

The histograms are computed for each of the three RGB channels separately and then they are concatenated in a single array.

The method *computeSharpness* takes as input a multidimensional array representing a color image, converts the image to a gray-scale image, computes the Laplacian of the image (that provides a measure of the sharpness) and returns the mean value of the laplacian of the image.

The method *computeGradient* takes as input a multidimensional array representing a color image, converts the image to a gray-scale image, computes the derivatives dx and dy of the image along the x and y axis by exploiting the Sobel masks, obtains the gradient magnitude from the derivatives using the formula $\sqrt{dx^2 + dy^2}$ and returns the mean value of the gradient module of the image.

The method *computeLuminance* takes as input a multidimensional array representing a color image, converts the image to the CIELAB color space, considers the luminance (L) channel of the image and returns the mean value of the luminance channel of the image.

The method *getFeatures* takes as input a multidimensional array representing a color image, calls the method *computeHistograms* obtaining an array of histogram features, it then calls (or not), based on the use_gradient, use_sharpness and use_luminance flags, the *computeGradient*, *computeSharpness* and *computeLuminance* methods obtaining a single value from each call, it appends the gradient/sharpness/luminance value to the array representing the histograms values and eventually it returns the array.

The methods *loadMWD* and *loadOther* load every image in the provided paths and for each of them compute the descriptors by exploiting the method *getFeatures* and the class label by using the method *getClass*. They then concatenate all the descriptors in a multidimensional numpy array (that has each descriptor as a row of the matrix) and they append all the lables in a list (the $i-th$ label correspond to the image descriptor in the $i-th$ row)

The method *loadImages* simply calls the sub-methods *loadMWD* or *loadOther* based on the fact that the dataset used is the MWD dataset or a different

one. This is needed because the folder structure of the MWD dataset is different from the others (it has multiple subfolders).

The final size of an image descriptor will correspond to $(3 * bin\_num) + num\_param$ where $3 * bin\_num$ are the features related to the histograms (since we compute the histograms on 3 channels and for each histogram we get bin_num values, where bin_num is the number of bins provided in the constructor) and num_params is equal to 3 if both sharpness, gradient and luminance are used, otherwise it corresponds to the number of these parameters used.

1.2.2 Weather Classification

To classify the images we first train a MLP classifier on the images descriptors and labels. In particular, the images classification is performed thanks to a class called **WeatherClassifier** that provides also some method to test the trained MLP classifier. This class takes as parameters in the constructor a multidimensional numpy matrix that must contain in each row the descriptor of an image, an array representing the images labels (the $i-th$ label correspond to the image descriptor in the $i-th$ row of the descriptor matrix) and two integer parameters called random_seed and max_iter that are used to set the random seed (to achieve a de- terministic behaviour accross different executions) and the maximum number of iterations needed to train the MLP classifier. Note that the data passed as parameter in the constructor must correspond to the training data.
The class has several relevant methods used to classify the images:

- *permute*
- *scale*
- *trainClassifier*
- *score*
- *gridSearch*

The method *permute* simply takes as input a multidimensional numpy array representing a dataset and an array representing the corresponding labels, computes a permutation of the indexes, applies the permutation to both the dataset and the labels in the same way and returns the permuted dataset and labels. This method is used to add some randomness in the ordering of the examples in a dataset.

The method *scale* simply takes as input a multidimensional numpy array representing a dataset and scales the features of the descriptors (rows of the multidimensional array) to balance the impact of different features having different ranges of values (note that the scaler used for this procedure is instantiated in the constructor and fitted on the training data).

The method *trainClassifier* creates a basic MLP classifier (with 100 neurons in the only hidden layer) based on the parameter passed in the constructor (random seed and number of iterations), it then permutes and scales the training data provided in the constructor, it fits the MLP classifier on the training data and it returns the trained classifier.

The method *scale* simply takes as input a multidimensional numpy array representing a test dataset and an array representing the corresponding labels, it scales the test dataset, scores the test dataset on the previously trained MLP classifier and it returns the score.

The method *gridSearch* works in the same way as the *trainClassifier* method but instead of using the standard MLP classifier it performs a grid search, testing the lbfgs and adam solvers and different hidden layers-neurons combinations (in particular we compare solutions with: single layer with 100 neurons, single layer with 20 neurons, single layer with 10 neurons, double layer with 100 neurons each, double layer with 10 neurons each). This method returns the best estimator found and the corresponding best score.

1.2.3 Execution and Testing

The execution and the testing of the proposed algorithm is performed in the python notebook *weatherclassification.ipynb* and exploits the class *Loader* (see Section 1.2.1) to compute the images descriptors and the class *WeatherClassifier* (see Section 1.2.2) to perform the image classification. In particular, we test the performances of the proposed algorithm with different parameters combinations and on different dataset combination. The basic pipeline is the same for each of the tests/executions and it is the following:

1. Create an instance of class *Loader* (one for each dataset to load), passing as parameters the appropriate data paths and flags (Note that the training and the corresponding test datasets must be loaded using the same parameter configuration, except for the data path and the dataset parameter).

2. Load the images of the training dataset(s) calling the method *loadImages* of the *Loader* instance.

3. Combine the loaded datasets (if needed).

4. Create an instance of class *WeatherClassifier* passing as argument the training dataset, the training set labels and the other needed parameters.

5. Train the classifier calling the method *trainClassifier* (or *gridSearch*) of the *WeatherClassifier* instance.

6. Load the images of the testing dataset(s) calling the method *loadImages* of the *Loader* instance.

7. Combine the loaded datasets (if needed).

8. Score the performances of the classifier calling the method *score* on the *WeatherClassifier* instance, passing the test dataset and the corresponding labels as parameters

9. Print results.

We performed 7 different tests by changing the parameters and/or dataset combinations (performances are reported in Chapter 2).
The first test is considered as the base case, in fact, we perform the classification (using the MLP classifier) for each of the datasets (separately) by considering only the features related to the color (all the 256 values of the histograms for each of the RGB color planes).
The second test is the same as the first but this time we considered also the features related to both the gradient, the sharpness and the luminance (since, after testing multiple configurations, we noticed an increase in performance when using all of them).
The third test works in the same way as the second but instead of considering all the 256 levels of the histograms they are quantized by considering only 100 bins (see Section 1.3 to understand this choice). This represents the best configuration of parameters found. We used this configuration in all of the following tests.
The fourth test uses the same images descriptors as the third test but it is performed on a combination of the SYNDRONE and the UAVid datasets; in particular, we merged the training set of the two datasets to obtain a single training set and we combined the two test set of the two datasets to obtain a single test set.
The fith test uses the same images descriptors as the third test but combines all the datasets; in particular, we merged all the training sets in a single training set and all the test sets in a single test set.
The sixth test is the same as the fifth test but we merged the two classes "shine" (corresponding to sunshine) and "clear" (corresponding to clear weather)

in the class "clear".
The seventh test is equivalent to the third test but performs grid search instead of using the standard MLP classifier.

1.3 Motivations & Desing choices

The image descriptors contain many features related to 4 main image characteristics: color, sharpness, gradient and luminance.
The image histograms have been used to try to extract some color features from the images, since weather conditions are usually characterized by a color, for example the sunset images will have colors close to orange and yellow, the snowy images will have color close to white, the rainy images will have colors close to gray etc.. The proposed algorithm allows also to quantize the histogram levels and this is due to two main facts:

- It reduces the number of features related to the histograms (color) and thus the length of the image descriptors. This is useful also because we consider only 3 features other than the ones related to the histograms and reducing the number of histograms features allows to give more importance to those three features.

- It allows to group color levels together, since images corresponding to the same weather condition tend to have similar but not perfectly matching colors.

The sharpness and the gradient have been used to try to capture the smoothing (or not) of the image. In particular, the fog and the rain, if compared to clean weather situations, usually tend to make the images smoother and to hide the edges.
The luminance has been used to try to distinguish very bright situations (usually corresponding to clear weather) from darker ones (usually corresponding to foggy or cloudy weather).
We decided to not use semantic segmentation to extract the sky region and compute some features based on those because, in many images, the sky corresponded only to a very little portion of the image or it was totally absent (e.g. SYNDRONE dataset's images don't have the sky).

Chapter 2

Performances

In this chapter we report the performances and some considerations about all the tests performed (see Section 1.2.3 to understand the tests behaviour).

The performances of the first test, that has been considered as the base case, since it classifies the images based only on the color features, are reported in table 2.1. As it can be seen the method performs very well in the UAVid and SYNDRONE dataset. This happens because the UAVid dataset has a small number of images and the SYNDRONE dataset's images have very different colors between images belonging to different classes.

Table 2.1: First test performances (base case).

Dataset	Correctly classified images (%)
MWD	83,45%
ACDC	86,8%
UAVid	100%
SYNDRONE	99,5%

The performances of the second test are reported in table 2.2. As can be seen from the table by using the additional parameters instead of only the color features the performances increase.

Table 2.2: Second test performances (all parameters).

Dataset	Correctly classified images (%)
MWD	88,84%
ACDC	87,8%
UAVid	100%
SYNDRONE	99,5%

The performances of the third test, that is considered the best test without combining the datasets, are reported in table 2.3. As can be seen from the table, by reducing the number of bins considered in the histograms we are able to increase the performances. This could be due to the fact that we merge similar color levels and images corresponding to similar weather conditions that have similar, but not equal, color.

Table 2.3: Third test performances (best parameter configuration on single datasets).

Dataset	Correctly classified images (%)
MWD	89,21%
ACDC	89,2%
UAVid	100%
SYNDRONE	99,5%

The fourth test, that combines the SYNDRONE and the UAVid datasets, is able to correctly classify the 98,64% of the images.

The fifth test, that combines all the datasets, has a correct classification rate of 92,86%.

The sixth test, that combines all the datasets and merges the "shine" and "clear" classes, correctly classifies the 92,28% of the images. By merging the classes we slightly reduce the performances of the algorithm; this could be due to the fact that the images classified as "shine" in the fifth test are all belonging to the MWD dataset and have a very bright and defined color if compared to the images classified as "clear" in the previous test.

The performances of the seventh test are reported in table 2.4. As it can be seen from the table, the grid search does not affect the performances.

Table 2.4: Seventh test performances (best parameter configuration on single datasets with grid search).

Dataset	Correctly classified images (%)
MWD	89,21%
ACDC	89,2%
UAVid	100%
SYNDRONE	99,5%

By comparing the performances of all the tests, we can conclude that the third test exploits the best parameter configuration by considering both the color, sharpness, luminance and gradient features and quantizing the histograms color levels. Furthermore, comparing the third and seventh test we can notice that the standard MLP configuration already has noticeable performances while requiring much less time than grid search. Eventually

the fifth and sixth test highlight that the proposed algorithm works well even with heterogeneous datasets.

Conclusions

The proposed algorithm represents a valuable method to classify a set of images based on the weather conditions. It trains a neural network exploiting some image descriptors that can be computed in a reasonable amount of time. The procedure is able to correctly classify more than the 90% of the images in the provided datasets reducing the memory and time consumption if compared to more complex methodologies that use CNNs.