UNIVERSITÀ DEGLI STUDI DI TRIESTE

DIPARTIMENTO DI MATEMATICA E GEOSCIENZE

MASTER DEGREE IN

DATA SCIENCE AND SCIENTIFIC COMPUTING

MASTER DEGREE THESIS

Learning-based automatic classification of lichens from images

---

Author:
Alberto PRESTA

Supervisor:
Dr. Felice Andrea
PELLEGRINO

ANNO ACCADEMICO 2019-2020

Università degli studi di Trieste

Abstract

Biomonitoring plays a crucial role in air quality control, as it allows to measure pollution, by estimating deviation from normal conditions of components of a specific ecosystem. In this framework, lichens are often used as bioindicators, due to their capability to adapt to the most disparate environments.

In this work we focus our attention on the task of lichens classification: even if our ultimate goal was to create a detection model, able to count the number of different lichens present in an image, due to the lack of data we organize our process as *patch classification*, meaning that given an image, we divide it in non-overlapping patches, and we associate each of them to a specific category. In order to perform this task, we extract image descriptors, following three different procedures: we first use handcrafted descriptors, that are based on predefined feature extractor algorithms, then we exploit convolutional neural networks to obtain a second type of descriptors, called deepnet, and in the end we merge these two approaches by employing scattering networks, which are structures based on cascading convolutions with particular wave functions, called wavelets. For each of these methods, we use the obtained descriptors as inputs to a classification algorithm.

We evaluate our process in terms of classification accuracy, empirically determining the most appropriate parameters for our problem. We show that, in our dataset, best results are obtained with handcrafted descriptors, as they overcome better the problem of lack of data.

Università degli studi di Trieste

Sommario

Il biomonitoraggio gioca un ruolo cruciale nel controllo della qualità dell'aria, poiché consente di misurare l'inquinamento stimando la deviazione dalle condizioni normali dei componenti di uno specifico ecosistema. In questo contesto, i licheni sono spesso utilizzati come bioindicatori, grazie alla loro capacità di adattarsi agli ambienti più disparati.

In questa tesi, ci focalizziamo sulla classificazione automatica di licheni: infatti, anche se il nostro obiettivo originale era quello di sviluppare un modello di rilevamento, abile quindi a contare quante specie diverse sono presenti in un'immagine, a causa della mancanza di dati organizziamo il nostro processo come un problema di *patch classification*, ossia data un'immagine, la dividiamo in celle non sovrapposte, e associamo ciascuna di esse ad una specifica categoria di licheni. Per conseguire questo obiettivo, da ogni immagine estraiamo dei descrittori, seguendo tre procedure distinte: inizialmente sfruttiamo descrittori handcrafted, ossia basati su algoritmi predefiniti, poi utilizziamo delle reti neurali convoluzionali per ottenere un secondo tipo di descrittori, chiamati deepnet, mentre alla fine uniamo questi due approcci utilizzando le scattering networks, le quali sono delle reti basate su convoluzioni a cascata con particolari funzioni d'onda, chiamate wavelets. I descrittori ottenuti vengono poi utilizzati per allenare un algoritmo di classificazione.

Valutiamo ognuno di questi metodi rispetto alla loro accuratezza di classificazione, determinando empiricamente i parametri più adatti per il nostro problema, e mostrando come i risultati migliori provengano dai descrittori handcrafted, poiché soccombono meglio al problema della mancanza di dati.

Acknowledgements

First of all I want to thank my parents Elena and Paolo, my brother Federico, my sister Margherita, and my grandparents Mario and Vittoria. My family has supported me with great love throughout my studies, and because of this I will be forever grateful to them.

I want to thank my friends Luigi, Lorenzo C., Andrea, Nicole, Aldana, Aronne, Lorenzo S., and Alberto who shared with me the years of my youth, making them fantastic. I want to thank also all my fellows from Cacaopoli, because you made the years of the master special (especially thanks to the dinners).

Regarding the realization of this project, I would like to thank my supervisor Felice Andrea Pellegrino for his patience, attention and willingness to help me both during the experimental phase and during the writing of the thesis. I also thank Professor Stefano Martellos for introducing me to the problem and kindly providing the dataset.

Contents

Chapter 1

Introduction

Computer vision, also known as CV, is defined as the field that develops techniques to understand the content of both digital images and videos. In recent years, thanks also to the increasing amount of data and the growing available computational power, CV became a fundamental technology in many fields, such as industrial automation, medicine, automotive, and biology. The reason why CV is so important is that it is able to solve different kinds of problems: it can track a specific object in a video, or it can discriminate a benign tumor from a malignant one. In particular, we are interested in *image classification*, that faces the problem of categorizing different objects of a certain class in digital images. Our ultimate goal is to build a process, based on CV techniques, finalized to recognize different species of lichens, in order to be able to compute the so called *lichen biodiversity index*, or IBL [1]: because of the lack of data, in this thesis we focus on solving a problem of patch classification, meaning that given an image, we divide it in non-overlapping patches, and we associate each of them to a specific category. We are not the first to face the problem of lichens classification: in [2], they used a convolutional neural network to classify 12 different species of lichens, obtaining an average accuracy of ~ 0.61. However, they treated the whole images as training data, while in this work we exploit single patches to train our model, extracted from available dataset. In Section 1.1, we give a brief definition of lichen, just to understand the nature of the object considered during this project, and why it is important to compute IBL, while in Section 1.2 we present the pipeline of the thesis, exploiting the main reasons of our choices.

1.1 Lichens

Lichens, as explained in [1], are *symbiotic organisms* composed of a fungus (mycobiont), and a photosynthetic partner (a green alga, or a cyanobacteria). The fungus benefits of the organic substances produced by photosynthetic partner,which in return receives protection, water, and mineral. During time, the coevolution of the partners led the formation of a complex and original biological structure. Thus, lichens are able to colonize the most disparate environments, at any latitude and longitude, managing to survive in extreme environments, such as the rocky outcrops of the Antarctica. Based on the characteristics of their body (called *thallus*), lichens have been divided in different morphological groups, which are:

1. *Crustose lichens*: thallus is very adherent to substrate.
2. *Squamulose lichens*: variant of Crustose lichens, whose thallus consists of ascending scales.
3. *Foliose lichens*: the lobes of thallus are flattened, while loosely attached to substrate.

FIGURE 1.1: Image of the lichen *Evernia prunastri*.

4. *Fruticose lichens*: thalli are tridimensional, and tend to branch in various directions. The overall look is bushy.

A detailed description of these different morphological groups is out of the scope of this thesis; it is enough to understand that lichens can come in many shapes, colors, structures, and sizes. Morphological characters, together with spores, and fruiting bodies are the bases for the identification for lichen taxa. On Figure 1.2 there are four examples of lichens from the above mentioned morphological group.

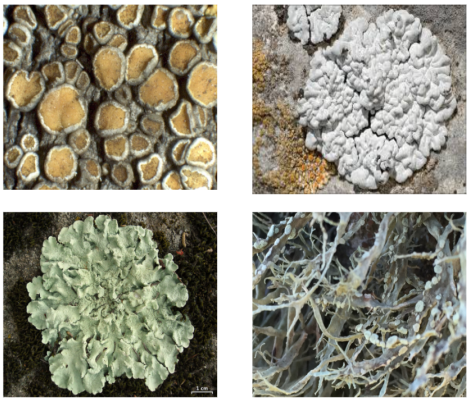


FIGURE 1.2: From top-left to bottom-right in clockwise order: *Caloplaca Cerina* (Crustose lichens), *Toninia Candida* (Squamulose lichens), *Flavoparmelia Caperata* (Foliose lichens), and *Ramalina Farinacea* (Fruticose lichens).

1.1.1 Lichen biodiversity index

The monitoring of air quality is one of the most tricky issues in environmental protection: even if pollution, in terms of concentrations measured instrumentally, has an operational definition, its monitoring is complex for several reasons, such as:

- The concentrations of pollutants vary in space and time following non linear patterns.
- The necessary equipment is often very expensive, both in terms of acquisition and maintenance.
- It is difficult to measure the effects of pollution on the environment, and on ecosystem services.

The use of organisms as biomonitors allows to estimate the biological effects of pollutant, in the sense that it does not output some estimates of air quality, but it measures deviations from normal conditions of components of an ecosystems. In this sense, the biodiversity of epiphytic lichens (i.e. those that grow on trees) has proved to be an excellent indicator of air quality alteration. In fact, lichens respond quite quickly (often by dying, or by evidencing necrosis of portions of the thallus) to the presence in the air of phytotoxic gases, which are associated to anthropic pressure (especially burning fuels in domestic heating, and traffic). These gases are often emitted in association with other substances, which are dangerous to humans, and other organisms. The decrease of lichen diversity has thus been associated with human diseases, such as lung cancer [3]. Thus, in highly polluted areas there is a reduction of the total number of species, as well as a decrease in the number of individuals for each species: the *lichen biodiversity index*, or IBL, which is a standardized approach for estimating air quality [1], is calculated by measuring the average number of lichen species present per surface unit.

1.2 Pipeline of the thesis

In this section we present the pipeline of the thesis, explaining, chapter by chapter, the reasons of our choices.

1. In Chapter 2, we describe the dataset and its limitation, especially for what concerns the number of images for each class. We also present all the steps we perform to make this dataset useful for our goal, and in particular we explain why we focus on a *patch classification* problem.

2. In Chapter 3, we try to recognize patterns able to discriminate different lichen species, by using handcrafted descriptors, that exploit predefined algorithm to extract features from images. The main goal here is to build a process that is not totally data dependent, but is it based on robust and well studied feature extractors, as we have at our disposal a small dataset.

3. In Chapter 4, we take advantage of the power of *convolutional neural networks*, in order to create deepnet descriptors: indeed, in last years they represent a real turning point in CV, and in general they outperform handcrafted descriptors in accuracy prediction. Because of this, we exploit them for the classification of lichens patches: however, in general they need lots of data to perform well, so in this chapter we try to adapt them to be effective on the available dataset.

4. In Chapter 5, we present a structure, called *scattering network*, that combines the handcrafted and the deepnet descriptors: on one hand, it is based on convolution with fixed wavelets, so there is no a learning phase through the data, but in the other hand it resembles the structure of convolutional neural networks. Scattering networks should be able to build a signal representation of an image, that is invariant to translation and stable to deformation.

5. In Chapter 6, we present the conclusions of the thesis, taking stock of all the results obtained with the different approaches, and assuming possible future developments.

Chapter 2

The Dataset

This chapter is entirely dedicated to the description of the dataset. In particular, in Section 2.1 we move toward the original one, describing its content in terms of number of species and number of images per species. Finally, in Section 2.2 we explain all the steps we took to modify the dataset, in order to make it useful for our purpose, and we explain why we prefer to solve a patch classification problem than an object detection one.

2.1 Description of the Dataset

The original dataset consists of over 50 different species, each of them having few labelled images (from a minimum of 1 to a maximum of 10). First of all, we did not take into consideration all different classes, but we preferred to narrow the field, by choosing the most representative ones[1]: in this way we focused on finding a reliable process to find generalized patterns, rather than working with too many different species. In particular, we chose 20 different classes, shown on Figure 2.1.

2.2 Patch classification: dataset management and image processing

In summary, we are dealing with a dataset composed of 20 different classes, each of which contains a number of images ranging from 4 to 10. In this context, we have been faced with two important problems. The first one is that *the dataset is too small to develop a lichens recognition process*: indeed, we need to have enough data both to train and test any procedure we decide to build. In this situation, we need to determine how many images, in general, are needed to implement an object detection algorithm. Even if a minimum number has never been set, it is crucial that a large amount of representative images is available per class, where the word *representative* stands for the fact that they should correspond with the range of scenario in which the model will be used. Considering the number of species and the variability present within the individual classes, we believe that at least $60 \sim 80$ representative images per category must be present, to be divided into train and test set. It is clear that the original dataset does not meet this requirement. Moreover, the second problem is that *there is no open-source database related to lichens*. This situation led us to change our approach, in order to make the most of the data we have: we moved to a *patch classification* problem, meaning that, given an image, we divide it into pieces, and we classify each of them individually. This allowed us to use a more extensive

[1] Choices have been made by considering both the importance of lichens and the quality of the images.

dataset, no longer formed by the original images, but by the patches that compose them. In practice, for each image we performed the following steps:

1. We resized the image to 1000×1000 pixels, in order to deal with data of same size. In doing this, we did not preserve aspect ratio: this could seem an error, but in practice images are all similar in size, so this change doesn't lead to exaggerated distortions that could compromise the process.
2. We removed the image border, since the lichens were mostly concentrated in the center. We obtained image of dimension 800×800.
3. From this new image, we extracted 16 non-overlapping patches each of which of size 200×200.
4. We normalized each channel of the obtained patch individually.
5. New dataset was composed by patches extracted from all images.

Samples of new dataset are shown on Figure 2.2. In this way, we augmented dataset, making it usable for our purposes. During the experiments, we divided the dataset into a train part, that represents 70% of the total, and a test part, that instead covers the remaining 30%. However, in making this simplification we ignored *background detection*. In fact, when we deal with a test image with more than one lichen, patch classification approach limits itself to follow these steps:

1. It resizes image to 1000×1000 pixels, in the same way as before.
2. It divides the new image in non-overlapping patches, each of them of dimension 200×200.
3. It classifies each patch individually.

Therefore, it does not consider the presence of background. Although this represents a problem that sooner or later will have to be addressed, in our opinion patch classification is the only possible approach we can use, at least in this first experimental phase.

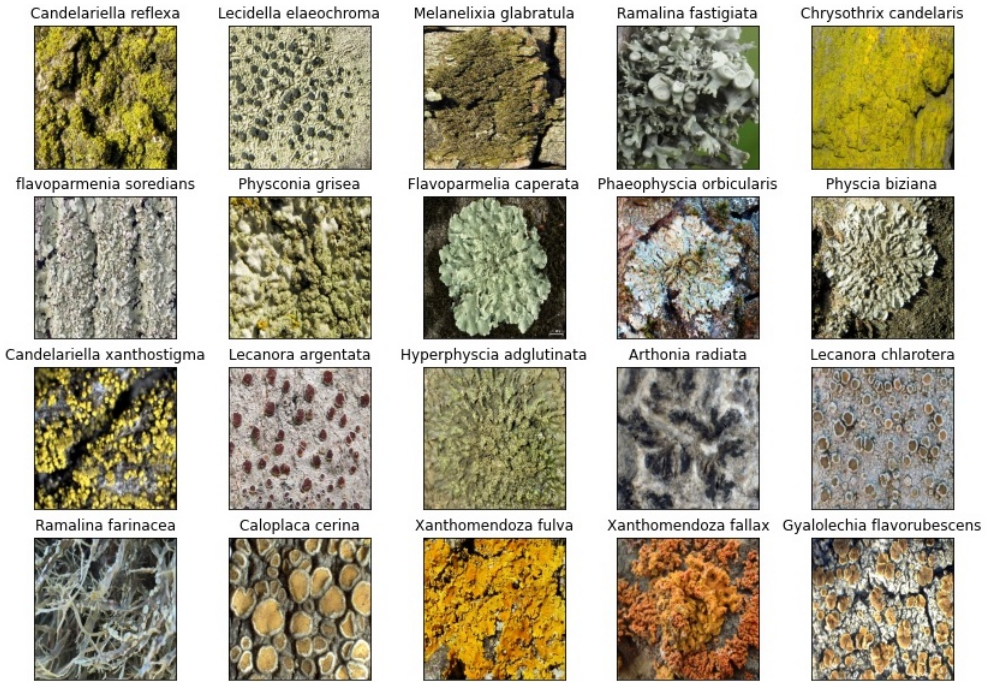


FIGURE 2.1: Images of the lichens taken into consideration during the project: for each class, we extract 1 sample from the original dataset.

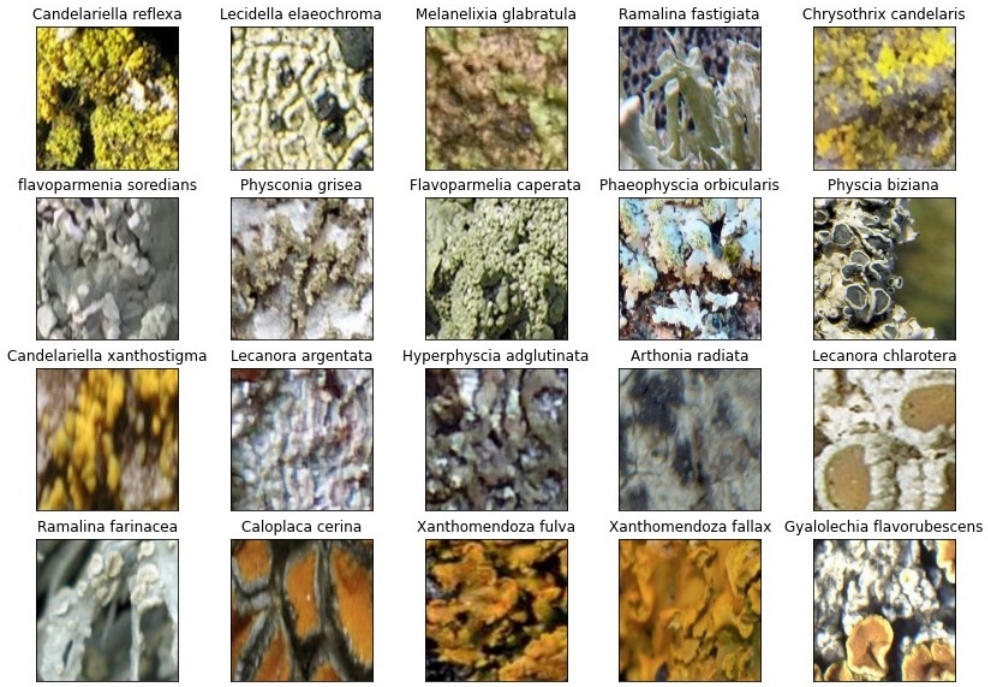


FIGURE 2.2: Patches of the lichens taken into consideration during the project: for each class, we extract 1 sample from the final dataset.

Chapter 3

Handcrafted descriptors for patch classification

In this chapter we exploit handcrafted descriptors, in order to perform classification of lichens images. This procedure is basically composed of three main steps:

1. Feature extraction using specific descriptors.
2. If necessary, creation of a dictionary using Bag of Visual Words technique.
3. Training of a classification model for lichens recognition.

This process has been widely used in the past and it has been proved to be quite effective. Rassen and Khoo [4] performed colored-image classification using combinations of SIFT descriptors on three different image channels. For classification, they used SVM equipped with a chi-square kernel function. However, they did not take into account scale invariance, which is important in our task. In this framework we focus our attention to handcrafted features, which are computed using a predefined algorithm that aims to extract specific characteristics from the image. In particular, we implement *local descriptors*, which describe neighborhoods around properly chosen points. Due to the high diversity present in our images, we have considered several descriptors, in order to find the one that has the most generalization power. In particular, we performed experiments with the following:

1. Local binary patterns (LBPs) [5].
2. Dense SIFT [4].
3. SIFT [6].
4. Gabor filters based descriptors [7].
5. Binary Gabor Patterns [8].

3.1 Descriptors

3.1.1 Local binary patterns (LBPs)

Local binary patterns, or *LBPs*, are very popular thanks to their computational simplicity and good performance. LBPs compute local descriptors of textures, by comparing each pixel with its surrounding neighborhood. For each pixel of the image (or for those pixels which have been chosen to be described, using for example a dense grid), we take its neighborhood, represented by a number P of points, all at a fixed distance r, called radius (see Figure 3.1). The value r allows to account for

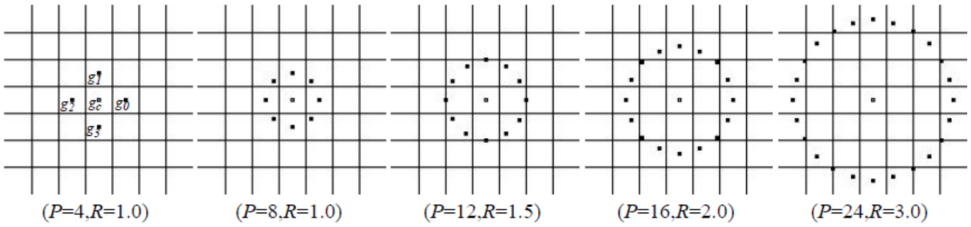


FIGURE 3.1: Circular neighbour sets, for different number of points P and radius r.

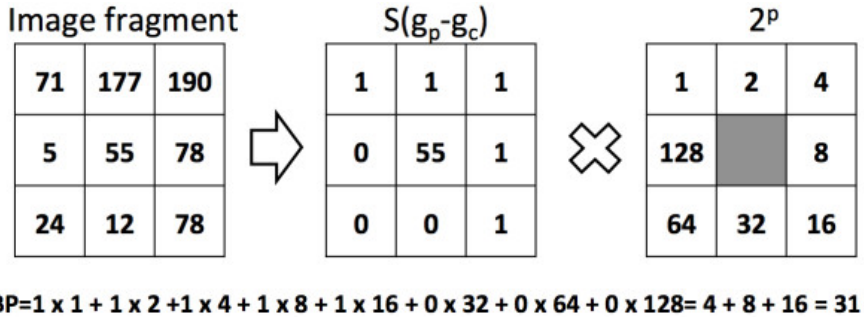


FIGURE 3.2: Computation of LBP. In this case, we start computing from upper-left nearby pixel in clockwise order.

different scales. Then, a binary mask is obtained by thresholding this neighborhood with the value of the center pixel: a nearby pixel becomes 1 if it is greater or equal than the center, 0 otherwise. Once we have the mask, we compute the LBP value with the following formula:

$$LBP\left(x_c, y_c\right) = \sum_p 2^p S\left(I_n\left(p\right) - I\left(x_c, y_c\right)\right) \tag{3.1}$$

where (x_c, y_c) is the pixel center, I_n are the values of neighboring pixels, p is their index, and S is the thresholding function. Figure 3.2 gives a visual representation of LBP computation for pixel (x_c, y_c). Finally, we extract the descriptor by calculating the frequency histogram of *LBPs*. We notice that dimension of the final vector depends on the parameters P: in general, we have that

$$\bar{v} \in \mathbb{R}^k \quad \text{where } k = \sum_{i=0}^{P-1} 2^i$$

where $\bar{v}$ is the final descriptor. For example, if we set $P = 8$, we have that $\bar{v} \in \mathbb{R}^{256}$, as values of *LBPs* ranges from 0 to 255.

Scale Invariance

LBPs are not scale invariant, since the radius r remains fixed during computation. A simple approach to partially overcome this problem is to take an image and extract *LBPs* using different r values, and considering all the resulting descriptors separately.

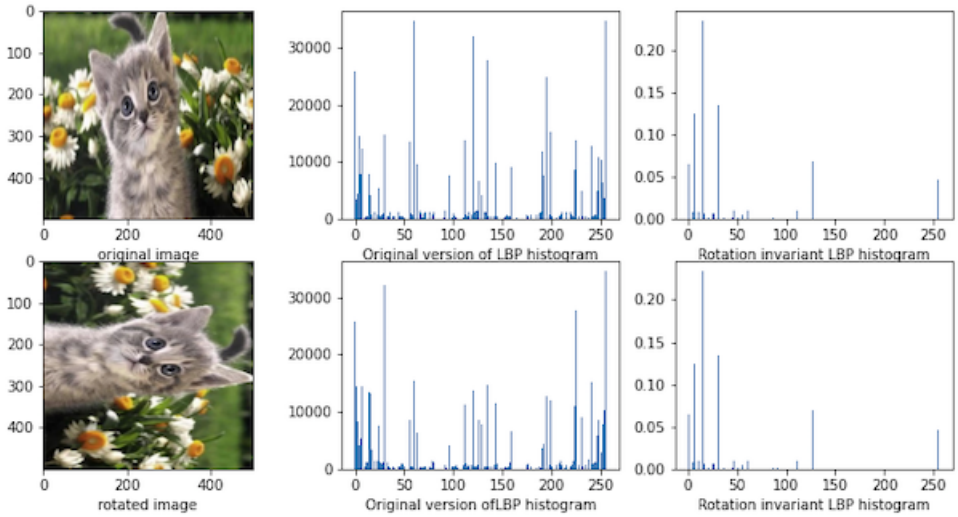


FIGURE 3.3: Comparison between two version of the LBP histogram. On the left columns we have an image and its 90° rotated version. On the center column we have the non rotation invariant frequency LBP histogram. On the right column we have the rotation invariant version of LBP frequency histogram.

Rotation Invariance

LBP operator is not able to obtain rotation invariance, as when we rotate the image, the values inside the sum in eq. (3.1) will change, resulting on a different descriptor. The above does not apply to the binary mask obtained during LBP computation: in fact, the neighborhood's member of a point remain constant at all rotation angles. To cancel the effect of rotation, we apply the function $ROR(x, i)$, which performs circular bit-wise shift on the binary number x, extracted from the binary mask, exactly i times. Let's clarify how this function works with an example: on Figure 3.2, we obtained from the mask the binary number $x = 11111000$. If we apply the ROR function with $i = 2$, we obtain:

$$ROR(x, 2) = 00111110$$

basically, we calculate LBP for every possible sequence, and then we take the minimum value: this process is described by the following formula:

$$LBP^{ri}_{P,R} = \min\{ROR(LBP_{P,R}, i) | i = 0, 1, \ldots, P-1\}$$

On Figure 3.3 we can observe the difference between the two distinct versions of the descriptor: in particular, the rotation invariant descriptor remained unchanged with respect to the two images.

3.1.2 SIFT

SIFT (Scale invariant feature transform) is one of the most famous existing keypoints detector and descriptors. Proposed by Lowe [6], it has been proved to be very effective for image matching, especially because it is capable of reaching both *scale and rotation invariance*.
In general, SIFT is divided in 3 steps:

1. Keypoint Localization.

2. Orientation assignment.
3. Feature point descriptors calculation.

Keypoint localization

Difference of Gaussians (DoGs)

The main goal of this stage is to find keypoints, i.e. points that are easy to describe and recognize both from different point of view and different scales: only for the latter the calculation of the descriptors is foreseen. How does the algorithm find these points? by detecting extrema of a continuous function known as *scale-space* [9].

Definition 3.1.1 *The scale-space of an image is defined as follows:*

$$L(x, y, \sigma) = G(x, y, \sigma) \star I(x, y) \tag{3.2}$$

where I is the input image, $G(x, y, \sigma)$ is the Gaussian filter of parameter σ, and $\star$ denotes the convolution product.

Instead of using directly the scale-space of an image, we approximate Laplacian of Gaussian with Difference of Gaussians (DoGs), defined as

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) \star I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma)$$

We use $D(x, y, \sigma)$ for the following reasons:

1. It is particular efficient to compute, as the smooth images L need to be compute in any case, and D can be computed by image subtraction.
2. It provides a close approximation to the scale-normalized Laplacian of Gaussians $\sigma^2\nabla^2 G$. Indeed, we have that

   $$\sigma\nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x, y, k\sigma) - G(x, y, \sigma)}{k\sigma - \sigma}$$

   and thus we have,

   $$G(x, y, k\sigma) - G(x, y, \sigma) \approx (k - 1)\sigma^2\nabla^2 G$$

   so it already includes the scale normalization terms. Note that the factor $(k-1)$ is a constant, so it does not affect the extrema location.

To construct $D(x, y, \sigma)$, SIFT combines the pyramid representation of an image: initially, a scale space representation of the image is created by convolving with Gaussians with different σ's, separated by a constant factor k in scale space. Each octave of scale-space is divided into a number s of intervals, so $k = 2^{\frac{1}{s}}$. Producing $s + 3$ images for each octave to be stacked, the final extrema detection covers a complete octave. Adjacent images are subtracted to produce *DoGs*, as it is shown on Figure 3.4. Once a complete octave has been processed, we downsample by a factor of 2 the Gaussian image, and the process is repeated.

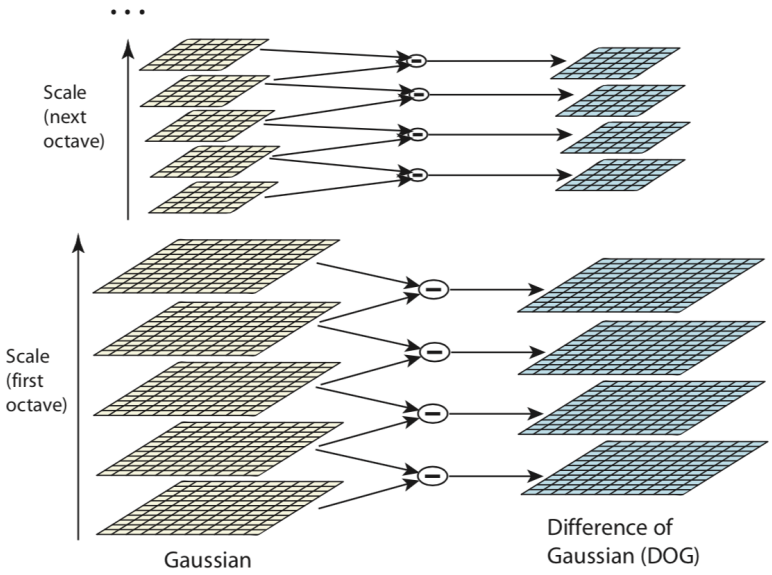


FIGURE 3.4: Calculation of DoGs with smoothed Gaussian images.

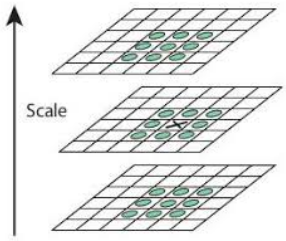


FIGURE 3.5: Search for extrema along 3-dimensional neighbour in DoGs.

Local extrema detection

Once we have built $D(x, y, \sigma)$, to find local maxima or minima all the sample points are compared to their $3 \times 3 \times 3$ neighbourhood, for a total of 26 comparisons: a point is selected only if it is larger or smaller than all of its nearby points, as we can see in figure 3.5.

Keypoints location refinement

Once a keypoint candidate has been detected, the SIFT procedure provides a localization at sub-pixel and sub-scale level by fitting a 3D quadratic function to the local points, in order to determine the interpolated location of the extrema. This approach exploits Taylor expansion of $D(x, y, \sigma)$, shifted so that the origin is at the sample point:

$$D(\mathbf{z}_0 + \mathbf{z}) \approx D(\mathbf{z}_0) + \left(\frac{\partial D}{\partial \mathbf{z}} \bigg|_{\mathbf{z}_0} \right)^T \mathbf{z} + \frac{1}{2} \mathbf{z}^T \left(\frac{\partial^2 D}{\partial \mathbf{z}^2} \bigg|_{\mathbf{z}_0} \right)^T \mathbf{z} \tag{3.3}$$

where $\mathbf{z}_0 = [x_0, y_0, \sigma_0]^T$ is the proposed point and $\mathbf{z} = [\delta x, \delta y, \delta \sigma]^T$ is the offset from this point. The location of the extremum $\hat{\mathbf{z}}$ is obtained by setting to 0 the gradient of

eq. (3.3)

$$\hat{\mathbf{z}} = -\left(\frac{\partial^2 D}{\partial \mathbf{z}^2}\Big|_{\mathbf{z}_0}\right)^{-1}\left(\frac{\partial D}{\partial \mathbf{z}}\Big|_{\mathbf{z}_0}\right) \tag{3.4}$$

The final offset $\hat{\mathbf{z}}$ is added to the location of its sample point, obtaining in this way the interpolated minimum or maximum. $D(\mathbf{z}_0 + \hat{\mathbf{z}})$ can also be used to discard unstable solution: indeed, if $|D(\mathbf{z}_0 + \hat{\mathbf{z}})|$ is below a certain threshold (decided a priori), the point is rejected, and classified as *low contrast point*.

Eliminating edge response

DoGs function has strong response along edges, but these points suffer from ambiguity, concerning the matching purpose, and moreover they are also unstable to small amounts of noise. A way to discard these problematic points is through the Hessian matrix H:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy}. \end{bmatrix}$$

It is well-known that a *well defined peak will have a large principle curvature in both directions*. At first, the Hessian matrix is calculated at the feature point, and then we take the *ratio of the two eigenvalues* λ_1 and λ_2 (principal curvatures): only points that respect next formula

$$\frac{Tr^2(H)}{\det(H)} = \frac{(\lambda_1 + \lambda_2)^2}{(\lambda_1 \lambda_2)} < \frac{(r+1)^2}{r} \tag{3.5}$$

are accepted, where r is a threshold fixed a priori.

Orientation assignment

SIFT algorithm assigns to each keypoint a dominant orientation based on local image properties: indeed, the descriptors will be extracted according to this orientation, thus obtaining invariance by rotation. The scale $\tilde{\sigma}$ of the keypoint is used to select the specific Gaussian smoothed image $L(x, y, \tilde{\sigma})$: once we have obtained it, the algorithm proceeds in the following way:

1. Take a 16×16 window around the sample point.
2. Compute the gradient around each pixel of the window.
3. Build a 36 angular bins histogram, where the contribution to each bin is weighted by the gradient magnitude and a Gaussian centered in the window.
4. The dominant orientation is the mode of the histogram.
5. The mode is estimated by quadratic interpolation with neighbourhoods.
6. Any other mode that is within 80% of the highest mode is used to create another descriptor with that orientation.

Feature point descriptor

Finally, we are able to extract the descriptor, which is basically a histogram of gradients and it is computed in the following way:

1. Take A 16×16 window around the point, rotated according to the dominant orientation.
2. For each pixel calculate a Gaussian-weighted gradient.
3. The window is divided into 16 sub-windows, each of size 4×4.
4. For each sub-windows build an 8 bin histogram of orientations, weighted by gradient magnitude.
5. Flatten all values in a 128 dimensional vector.
6. Normalize the vector.
7. Values in the vector below a certain threshold are shrunk to 0, and normalize again the vector.

On Figure 3.6 we can see a visual representation of this procedure.

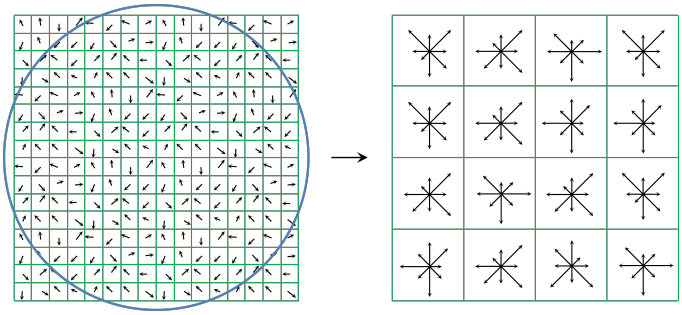


FIGURE 3.6: Representation of SIFT: on the left part of the image, we have gradients of the 16×16 window around the feature point. On the right part we have the 8 bin histograms of the sub-windows.

SIFT for color images

In order to extend SIFT to color images, we consider each channel of an image individually: for each of them,we find the keypoints and we extract the descriptors. All of these output vectors are considered in computing the final frequency histogram.

3.1.3 Dense SIFT

As we have seen, SIFT describes only keypoints, as they are easily traceable, and it obtains both scale and rotation invariance: it has been shown with many application that it performs well with feature matching. However, such a sparse descriptors distribution over the image could bring poor results in term of image classification, as it would be necessary to have lots of descriptors, extracted uniformly from images of interest. Because of this, we introduce *Dense SIFT*: it differs from the previous version only for the fact that *it extracts descriptors from a dense uniform grid instead of taking only keypoints*, without changing the remainder of the process. This allows us to have more descriptors, and furthermore to have the same number of them for each image, achieving better results in terms of classification accuracy.

3.1.4 Scale invariance in dense framework

The main problem in this dense scenario is the management of scale changes: indeed, for non-important points (which are the majority in Dense SIFT) a stable scale is difficult to determine: this makes impossible to calculate descriptors at their maximum scale, forcing us to compute them at the same scale, losing any kind of invariance. This can be a problem if we have to recognize images at different scales. In order to overcome this problem, we can follow two different approaches, explained below:

1. We extract descriptors at different scales, as if they came from different images. This method can be seen as some kind of data augmentation, and it helps us to achieve invariance to small change of scale. The idea behind this approach is that, in absence of any priori information about what scales to choose, we take the image and we consider it at multiple scales.
2. We choose specific scale of a point based on the characteristics of the nearby keypoints.

In the following sections we analyze better the second approach, as it has two possible ways to compute the scale of a fixed point.

Propagation of scales

In the next few lines we explain how scale can be propagated from keypoints to all the other points in an image. These methods have been introduced in [10]. First of all, we give the following definition:

Definition 3.1.2 *Scale-map*
We define Scale-map $S_I(\mathbf{p})$, for pixel $\mathbf{p}$ of image I *as*

$$S_I(\mathbf{p}) = \sigma_{\mathbf{p}}.$$

where $\sigma_{\mathbf{p}}$ is the assigned scale to the point $\mathbf{p}$.

Our purpose is to define S_I for all pixels of the image. To do so, we define the following function, which has to be minimized

$$C(S_I) = \sum_i \left(S_I(\mathbf{p}) - \sum_{\mathbf{q} \in N(\mathbf{q})} \left(\mathbf{w_{pq}} S_I(\mathbf{q}) \right) \right)^2. \tag{3.6}$$

The above cost function expresses the wish that the scale assigned to pixel $\mathbf{p}$ is as similar as possible to a weighted average of all the scales of nearby keypoints, represented by $N(\mathbf{q})$. The important part here is to define weights $\mathbf{w_{pq}}$, which reflect how keypoints influence other points. We see two possible approaches:

1. *Geometric scale propagation*: In this framework, we assume that neighbouring pixels should be assigned with similar scales: this assumption is represented as weights of this function

   $$\mathbf{w_{pq}} = \frac{1}{|N|}$$

   where $|N|$ is the number of spatial neighbors for each pixel. Here, we propagate scales using only the geometry of the feature point selections, ignoring image intensities as valuable factor.

2. *Image-aware scale propagation*: here, the assumption is that neighboring pixels with similar intensities, should be assigned with similar scales. Following this premise, we can assign weights based on normalized cross-correlation of the intensities of the two pixels, so we have

$$\mathbf{w_{pq}} = 1 + \frac{1}{\sigma_{\mathbf{p}}^2} \left(\left(I\left(\mathbf{p}\right) - \mu_{\mathbf{p}} \right) \left(I\left(\mathbf{q}\right) - \mu_{\mathbf{p}} \right) \right)$$

where $\mu_{\mathbf{p}}$ and $\sigma_{\mathbf{p}}$ are the mean and variance of the intensities in the neighborhood of pixel $\mathbf{p}$.

3.1.5 Dense SIFT for color images

Dense SIFT is thought for grayscale images, as it deals with only 3 dimensions: two spatial dimensions and one scale dimension. However, our dataset is composed by color images and, moreover, colors are a fundamental characteristic for the recognition of different species. So, how can we deal with such a dataset? of course we must take into account the responses at 3 channels of the image. In order to extend Dense SIFT we can follow two strategies, borrowed from [4], explained below:

1. **RGB-SIFT**: it is obtained by simply computing the dense SIFT descriptor over all 3 channels of the RGB color space independently, and then concatenate the results together, obtaining a 384-dimensional vector.

2. **Opponent-SIFT**: it is obtained by extracting descriptors using opponent color space channels, described by the following equations

$$O_1 = \frac{R - G}{\sqrt{2}} \tag{3.7}$$

$$O_2 = \frac{R + G - 2B}{\sqrt{6}} \tag{3.8}$$

$$O_3 = \frac{R + G + B}{\sqrt{3}} \tag{3.9}$$

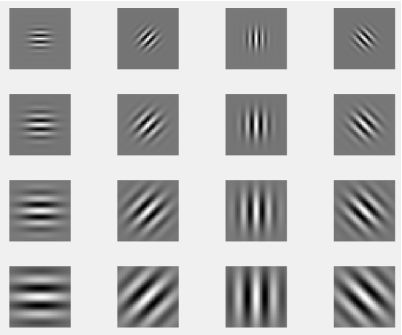
O_1 and O_2 represent the color information, while O_3 represents the intensity information. Also in this case the final vector is of dimension 384.

3.1.6 Gabor filter based descriptors

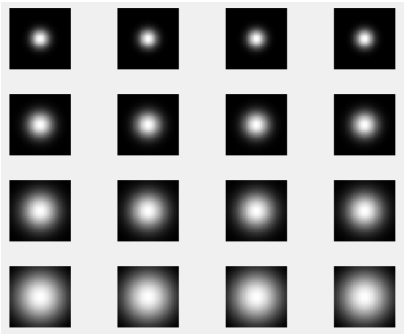
In this section we introduce how we exploit Gabor filters to perform patch classification. Gabor filter is a linear band-pass filter widely used in computer vision tasks such edge detection, feature extraction, and texture classification. These filters can approximate the characteristics of certain cells in the virtual cortex of some mammals. Formally, we can give a mathematical definition of Gabor filters:

Definition 3.1.3 *A 2D Gabor filter is a sinusoidal signal of a specific frequency and orientation, weighted by a Gaussian wave. Being a complex filter, it has a real and a complex component. Denoting by (x, y) the image coordinates, we have the following equation to represent the complex Gabor filter:*

$$g(x, y) = \left(\frac{1}{2\pi\Omega} \right) \exp\left(-\frac{1}{2} \left(\alpha x'^2 + \beta y'^2 \right) \right) \exp\left(2\pi i \Omega \right) \tag{3.10}$$



(A) Real part of Gabor filters when we change orientation and scale.



(B) Magnitude of real part of Gabor filters when we change orientation and scale.

with

$$x' = x\cos\theta + y\sin\theta \qquad y' = -x\sin\theta + y\cos\theta$$

where

1. Ω *is the frequency of Gabor function.*
2. $\alpha = \frac{1}{\sigma_x^2}$ *and* $\beta = \frac{1}{\sigma_y^2}$ *are equal to the inverse of directional standard deviations of the Gaussian envelope, which determine its bandwidth.*

Figure 3.7a and 3.7b show a bunch of Gabor filters and their magnitude, where we vary both orientation and scale along horizontal and vertical axis, respectively.

Gabor descriptor

We want to extract a valid descriptor from Gabor filters. First of all, for a given image $I(x, y)$ of dimension $P \times Q$, we define

$$G_{mn} = I \star g_{mn} = \sum_s \sum_t I(x, y) g^*_{mn}(x - s, y - t)$$

the convolution of the image with a specific Gabor filter g with scale m and orientation n,where the $*$ denotes the complex conjugation. In order to represent regions, we use the mean and the standard deviation of G_{mn}[1]

$$\mu_{mn} = \frac{1}{PQ} \sum_x \sum_y |G_{mn}(x, y)|$$

$$\sigma_{mn} = \sqrt{\frac{\sum_x \sum_y (|G_{mn}(x, y)| - \mu_{mn})^2}{PQ}}.$$

We define the Gabor descriptors as follow

$$\mathbf{x} = [\mu_{00}, \sigma_{00}, ..., \mu_{(S-1)(K-1)}, \sigma_{(S-1)(K-1)}] \tag{3.11}$$

where S is the total number of scales and K is the total number of orientations. As in LBPs and SIFT case, Gabor filters work with grayscale images: to adapt to our dataset, as usual, we limit ourselves to chaining the descriptors of the three different channels:

$$\mathbf{x} = [\mathbf{x_r}, \mathbf{x_g}, \mathbf{x_b}]$$

[1] We assume that regions have homogeneous textures.

3.1.7 Binary Gabor Patterns (BGPs)

In this last section we try to combine Gabor filters and LBPs, with the so called *binary Gabor patterns*, or BGPs [8]. In simple words, we take a patch around a fixed point $\mathbf{x}$ of radius R, and from that we extract the LBP value as we did in Section 3.1.1, but in this case we first apply a Gabor filter to this patch.

Rotational invariant descriptor

Let's express Gabor filters as real-part and imaginary-part, separately. We have that

$$g_r(x,y) = exp\left(-\frac{1}{2}\left(\frac{x'^2}{\sigma^2}+\frac{y'^2}{\sigma^2}\right)\right)\cos\left(2\pi\Omega x'\right) \tag{3.12}$$

$$g_{im}(x,y) = exp\left(-\frac{1}{2}\left(\frac{x'^2}{\sigma^2}+\frac{y'^2}{\sigma^2}\right)\right)\sin\left(2\pi\Omega x'\right) \tag{3.13}$$

Consider an image patch p with linear size R centered at $\mathbf{x}$: taking a Gabor filters bank $g_0 \sim g_{J-1}$ (real or imaginary) in which all parameters are shared except for the orientation one (θ), each value is calculated as follows:

1. We apply $g_0 \sim g_{J-1}$ to the patch p, by multiplying them in point-wise manner and then summing up all together. By doing this we obtain a vector $r = \{r_j | j = 0, \ldots, J-1\}$
2. By binarizing r, we get a binary vector $b = \{b_j | j = 0, \ldots, J-1\}$ of $0's$ (if the element is negative) and $1's$ (if the element is positive).
3. As in LBPs, we transform b into a unique value using the following formula

   $$BGP = \sum_j b_j \cdot 2^j$$

   BGP can produce 2^j different output values.
4. In order to achieve rotation invariance, we define BGP_{ri} as

   $$BGP_{ri} = \max\{ROR(BGP, j) | j = 0, 1, \ldots, J-1\},$$

   where $ROR(x, j)$ is defined as in Section 3.1.1.

The whole process is represented in Figure 3.8. Once we have obtained rotation invariance, we can focus on the problem of scales. It is quite easy to achieve this goal, since Gabor filter is an excellent tool for multi-resolution analysis: all we need is to simply varying parameters λ and σ, obtaining Gabor filters at different resolutions. For all these resolutions, we can extract specific BGP_{ri} operator, that can be concatenate together.

Frequency histogram of BGPs

Considering s different resolutions, we have $2s$ *BGPs* operators (one for real part and one for imaginary part). With these operators, we construct a normalized histogram response over the whole image, obtaining $2s$ different histograms. The final descriptor is obtained by concatenating all of them. The dimension of each histogram depends only on J (number of orientations): setting this value to 8, we have that

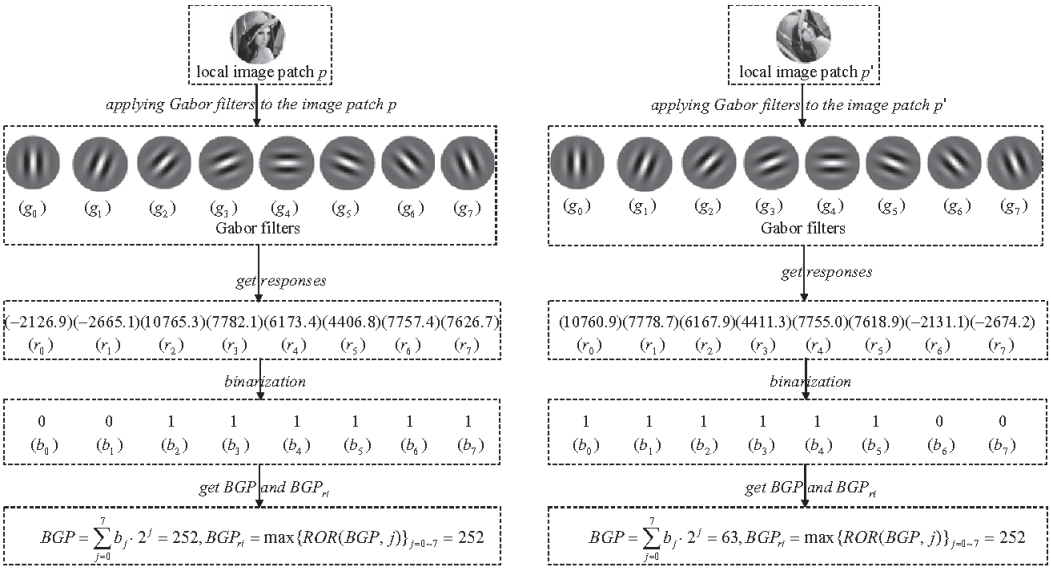


FIGURE 3.8: BGP_{ri} calculation's steps from an image patch. As we can see, the value for rotated version of the same path remains unaltered. Image take from [8].

each histogram has got 36 bins. Putting all this together, the histogram consists in a vector of dimension $36 \times s$. Moreover, on order to extend the descriptor to RGB images, we concatenate descriptors from the 3 different channels.

3.2 Bag of visual words (BOVW)

Descriptors like SIFT (both dense and sparse version) extract an N-dimensional vector for each chosen point, so each image has a huge number of vectors, all with the same size. The best way to proceed in this situation is to treat the image as a *document*, where each descriptor is a specific *word*. This technique, borrowed from text retrieval framework, is called *Bag of visual words* (BOVW). It is a very common technique in computer vision and it is based on two sequential phases:

1. Visual words vocabulary construction.
2. Creation of frequency histograms through visual words.

Once we have performed these 2 steps, we have a dataset composed by frequency histograms (one for each image): we can use them to train a classifier, or to apply the nearest neighbors procedure.

3.2.1 Visual vocabulary construction

In this first stage, we take each image and we extract descriptors, following a specific procedure (SIFT for example). Once this is done, we take all these descriptors and we group them using a clustering algorithm, for example k-means. Finally, we simply define the centroids of the clusters as the words in our vocabulary, also called *visual words*.

3.2.2 Frequency histograms of images

The last step is to create, for each image, a frequency histogram, where we count, for each visual word, how many times the latter is present in the image. Those histograms are our bag of visual words. Most of the time we must employ with normalized histogram, because algorithms like SIFT can extract a different number of descriptors for different images.

3.3 Classification model

Once we have translated images in a set of frequency histograms, we use them to train a model able to perform supervised classification: to this task, we have chosen to use two different models:

1. Support vector machine (SVM).
2. *K*-nearest neighbor (KNN).

3.3.1 Support vector machine (SVM)

Maximal margin classifier

Support vector machine (SVM) is one of the most popular machine learning classifier, that is widely used for classification and regression problem. Most of the following explanation is from [11]. This method is based on the idea of finding the hyperplane that best divides a dataset, as shown in Figure 3.9. The name is due to the presence of *support vectors*, which are the data points nearest to the hyperplane and that alter its position: indeed, one of the main features of this algorithm is that it depends on few data, so it is possible to achieve quite good results with few sample points. Consider the case in which we have 2 different classes: so we have a $n \times m$ data matrix $\mathbf{X}$ with n training data of dimension m

$$\mathbf{x_1} = \begin{pmatrix} x_{11} \\ x_{12} \\ \dots \\ x_{1m} \end{pmatrix}, \dots, \mathbf{x_m} = \begin{pmatrix} x_{m1} \\ x_{12} \\ \dots \\ x_{mn} \end{pmatrix}$$

and $y_i \in (-1, 1)$. The goal here is to find the hyperplane which divides this two classes with the maximum possible distance. Supposing that it is possible to construct such a hyperplane (called *maximal margin hyperplane*), we have the following property:

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_p x_{ip}) > M \qquad M > 0 \qquad \forall i = 1, \dots, n \tag{3.14}$$

If $\beta_0, \beta_1, \dots, \beta_p$ are the coefficients of the maximal margin hyperplane, then a test observation $\mathbf{x}^*$ is classified in the following way

$$y^* = \begin{cases} 1 & (\beta_0 + \beta_1 x_1^* + \dots + \beta_m x_m^* > 0) \\ -1 & (\beta_0 + \beta_1 x_1^* + \dots + \beta_m x_m^* < 0) \end{cases}$$

Figure 3.9 shows an example of maximal margin classifier in the case of 2-dimensional training data.

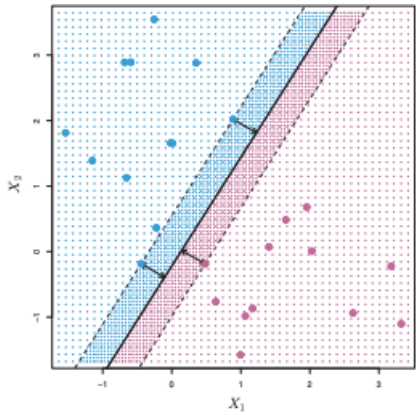


FIGURE 3.9: Representation of maximal margin classifier which divides the 2 classes, represented here with different colors. The margin is the distance from the hyperplane (solid line) to the support vectors.

Optimization problem to obtain the maximal margin classifier

To construct this kind of hyperplane, what we have to do is to make sure that M is as large as possible: this is translated in the following optimization problem

$$\max_{\beta_0,\beta_1,\dots,\beta_m} M \tag{3.15}$$

$$\text{subject to} \quad \sum_j \beta_j^2 = 1 \tag{3.16}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}) > M \qquad \text{M} > 0 \qquad \forall \text{i} = 1,\dots,\text{n} \tag{3.17}$$

Equation (3.17) is the constraint that forces each observation to be on the correct side of the hyperplane (M is positive). Moreover, eq. (3.16) is not a really constraint, since it allows to take a unique representation of the maximal margin classifier, without redundancy.

Non separable case

Maximal margin classifier is a very effective way to perform classification: however, in many practical cases such a margin classifier does not exist, and so we have to permit the presence on errors. The goal now becomes to find the hyperplane that best classifies training data, and minimizes misclassification errors. Such a classifier is called *support vector classifier*. Also in this case, we translate this situation to an optimization problem

$$\max_{\beta_0,\beta_1,\dots,\beta_m,\epsilon_1,\dots,\epsilon_n} M$$

subject to

$$\sum_j \beta_j^2 = 0 \tag{3.18}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + . \cdots + \beta_p x_{ip}) \geq M(1 - \epsilon_i) \tag{3.19}$$

$$\epsilon_i \geq 0, \qquad \sum_i \epsilon_i \leq C \tag{3.20}$$

where C is a non-negative parameter which represents how much error are we able to bear. When C is large, there is a high tolerance for errors and, because of this, the

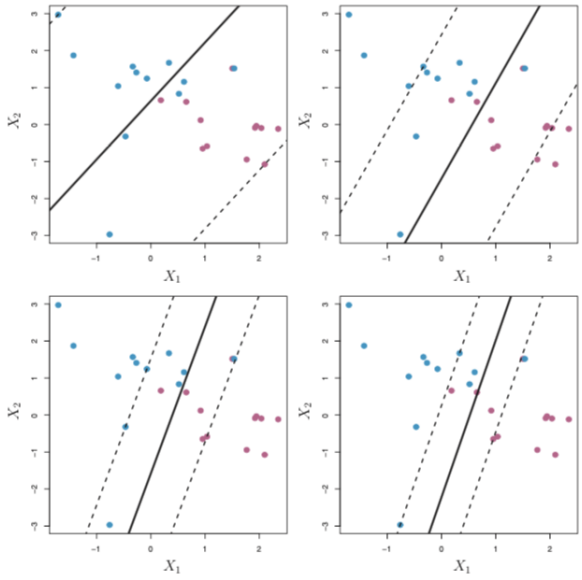


FIGURE 3.10: Support vector classifiers for different values of C. Largest value is used in top left panel, and smaller values are used in remaining. If C is large, we allow a lot of errors as we have high tolerance, so the margin will be large.

margin will be large: otherwise the margin will be small. $\epsilon_1, \ldots, \epsilon_n$ are called *slack variables* and allow single observations to be misclassified. As we mentioned before, observations which strictly lies on the correct side do not affect the classifier, which is determined by only the closest vectors. Fitting process is shown in Figure 3.10.

Introducing Non-linearity

For a lot of problems, a hyperplane is not enough to reach good results: sometimes the intrinsic nature of the data is not linear, so it is necessary to extend our classification model to these cases. A first possible approach is to move in a larger space, in which data are linearly separable, as we can see in Figure 3.11. The 2 steps in this frameworks are:

1. Non-linear transformation through a feature mapping.
2. Linear classification in the new space.

The problem in this context is that the feature space can be very large, becoming computationally intractable. Because of this, it is common to apply the so called *kernel trick*. It can be shown that the maximal margin classifier can be represented as

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i \langle x_i, x \rangle \tag{3.21}$$

where S denotes the set of support points and $\langle \cdot, \cdot \rangle$ is the scalar product. The core of the kernel trick is to replace the inner product with a generalization of it

$$K(x_i, x_{i'})$$

where K is called *kernel* and determines the similarity between two observations. Examples of kernels are:

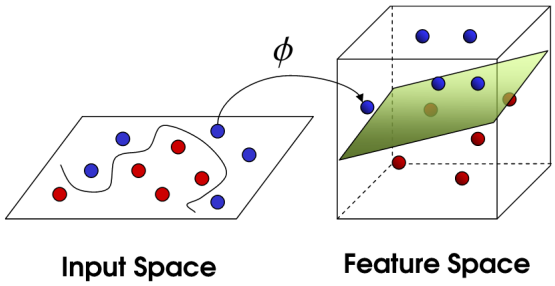


FIGURE 3.11: Graphical representation of the non linear application with which we move in a larger space, where it is possible to build a linear classifier.

1. Polynomial kernel of degree d

$$K(x_i, x) = \left(1 + \sum_j x_{ij} x_{i'j}\right)^d$$

2. Radial kernel

$$K(x_i, x_{i'}) = \exp(-\gamma \sum_j (x_{ij} - x_{i'j})^2)$$

Once a kernel has been decided, 3.21 is combined with it obtaining the final form

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x, x_i) \tag{3.22}$$

The main advantage in this approach is that we do not work explicitly in the enlarged space, but we simply compute $K(x_i, x_{i'})$ for all $\binom{n}{2}$ couples that can be formed with the training set. Figure 3.12 shows SVM models with the 2 different kernels mentioned before.

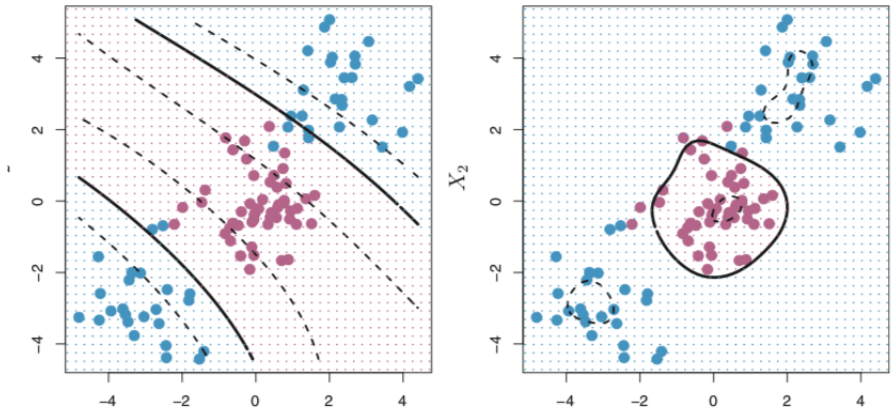


FIGURE 3.12: Left: SVM with a polynomial kernel of degree 3. Right: SVM with Gaussian kernel.

One-vs-Rest classifier

SVM is a classifier which takes into account only 2 different classes. However, in our problem we deal with 20 different species of lichens, so we must enlarge the method for our purpose. Setting T as the number of classes, we train T different $SVMs$ model, each time considering one class against the remaining ones. Each model t produces a set of parameters $\beta_{0t}, \beta_{1t}, \ldots, \beta_{mt}$. Taking a test observation x^*, we assign it to the class for which

$$\beta_{0t} + \beta_{1t}x_1^* + \cdots + \beta_{mt}x_m^*$$

is the largest.

3.3.2 K-NEAREST NEIGHBORS (KNN)

Another possible approach is K-nearest neighbors, or KNN. This algorithm assumes that instances belonging to the same class are near to each other in the input space. Fixed a test observation x^*, the core of the process is to look at its k nearest points in the training set, and use them to estimate the probability for the point to be in a class j.

$$Pr(\text{class} = j | X = x^*) = \frac{1}{K} \sum I(\text{class} = j) \tag{3.23}$$

In practice, for each x^* the algorithm follows these steps:

1. Initialize K.
2. Take the K closest points and order them based on their distance to x^*.
3. Get the labels of these K points.
4. Return the mode of the K labels as the class of x^*.

3.3.3 Pipeline of the process

In previous sections we introduced 4 descriptors, with which we try to describe different lichens. As we can notice, these descriptors are different to each other, and in particular we recognize two categories:

1. SIFT and Dense SIFT, that extract a vector from each point, so each image is described by a collection of descriptors. In this framework, it is necessary to synthesize all the descriptors relating to a single image in a frequency histogram, by applying BOVW. Once we have the histograms, we can use them for the last classification step, in which we train a SVM model, or we perform KNN classification.
2. LBPs, Gabor features and BGPs, in which a single number is calculated for each point, and the image is described by the frequency histogram of these values: because of this it is not necessary to apply BOVW. So, the descriptors are directly used either to train the classification model or to perform KNN classification.

Pipelines of the entire process is represented in Figure 3.13 and Figure 3.14.

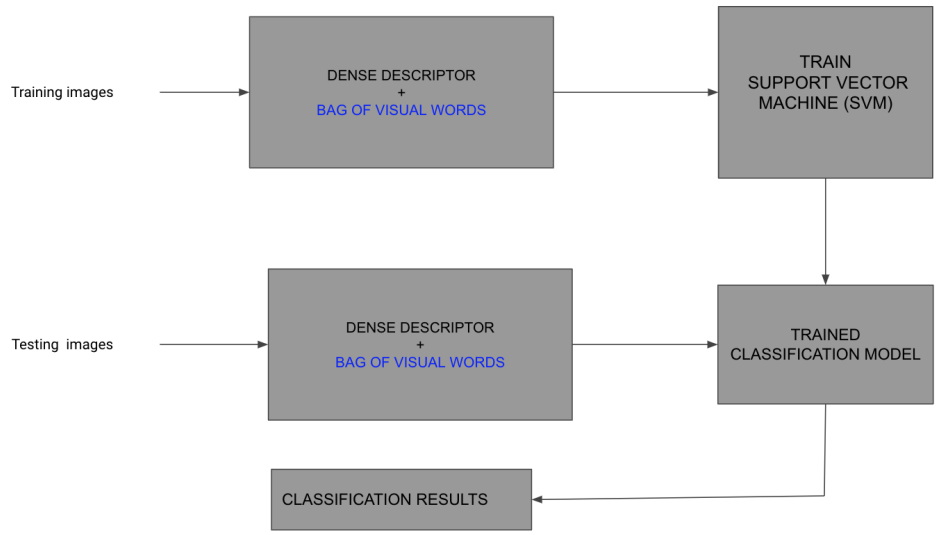


FIGURE 3.13: Classification procedure using SVM algorithm. BOVW step is in blue because it is not always necessary.

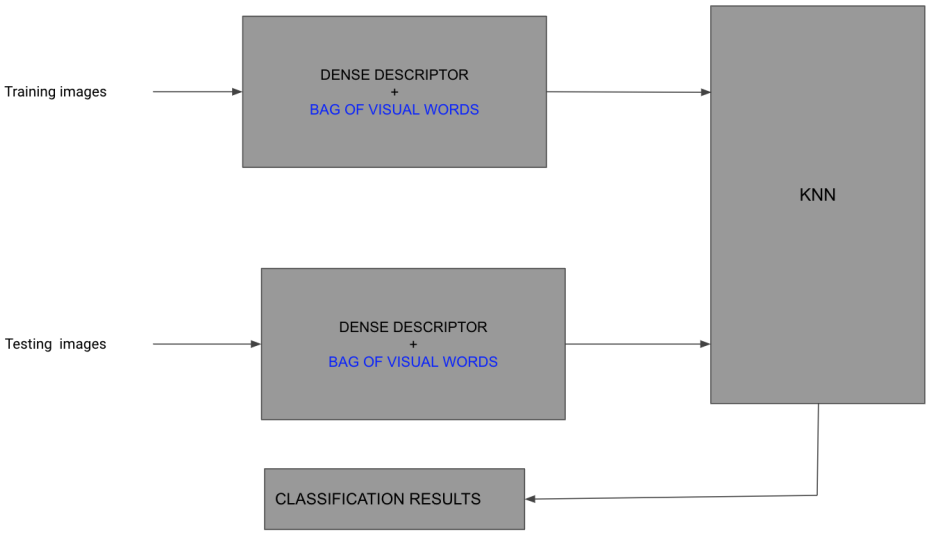


FIGURE 3.14: Classification procedure using KNN algorithm. BOVW step is in blue because it is not always necessary.

3.4 Experimental Results

In this section we experimented handcrafted descriptors to classify lichens. We treated our dataset as described in Chapter 2. We analyzed all descriptors mentioned in previous section one by one, and we interpreted results in terms of accuracy achieved, which can range from 0 (worst situation, where no images in the test set are classified correctly) to 1 (best situation, where all images in the test set are classified correctly). Furthermore, in this section we analyzed some specific model in details, evaluating both the *precision*, that measures what is the proportion of positive identifications that are actually correct , and *recall*, that measures what is the proportion of actual positive identification that are classified correctly. We often

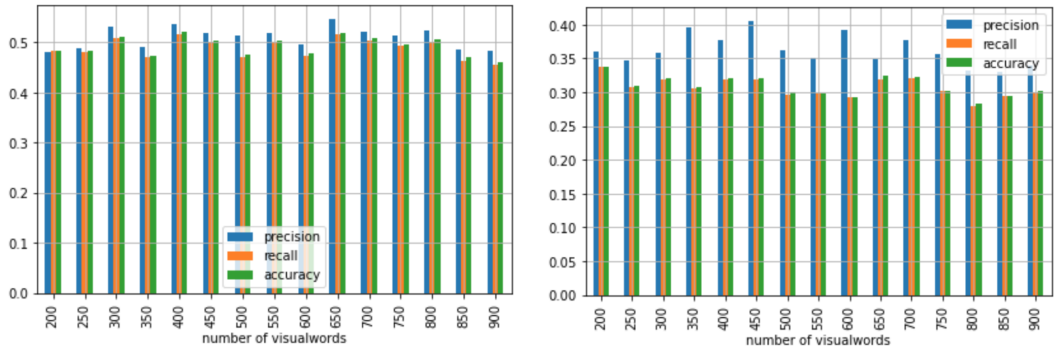
showed F1-score as well, that is defined as the harmonic mean of precision and recall. In summary, we have that:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN} \qquad F1 - score = 2\frac{precision \times recall}{precision + recall}$$

where TP stands for *true positive*, FP for *false positive*, and FN for *false negative*.

3.4.1 Classic SIFT

The first descriptor we studied is the classic version of SIFT, where only keypoints of an image are taken into account. The number of visual words varied from 200 to 900. Once we had the SIFT descriptors, we fed them into a classification model, which can be either a Gaussian SVM (whose parameters have been tuned experimentally) or a KNN model, with $k = 1$: we compared results of these two different classifiers. On Figure 3.15a and Figure 3.15b we can observe how accuracy, precision and recall vary with respect to the number of visual words with SVM and KNN model, respectively. We observe that in both cases, especially with the KNN model, the precision is higher than both recall and accuracy. We achieved best results with SVM model and 650 visual words, where we reached an accuracy ~ 0.52: despite this, we notice that the performances are similar when we change the number of visual words, meaning that it is not such a fundamental parameter to choose. Results are significantly lower when using the KNN model, achieving a maximum accuracy of ~ 0.32.



(A) Classic SIFT with SVM model.

(B) Classic SIFT with 1NN model.

FIGURE 3.15: Results in terms of Accuracy, precision and recall with Classic SIFT descriptors.

Take a specific model

As an example, we considered a specific model, in order to evaluate how single species are described by classic SIFT. We put ourselves in the best possible situation, that is the one with 650 visual words and the Gaussian SVM classifier. Figure 3.16 represents how well different species are classified by this algorithm: indeed, accuracy, precision, and recall are plotted. We can observe how the results change a lot from species to species: some are well classified, such as *Lecidella elaochroma*, while others show some problems, such as *Flavoparmenia soredians*. We can also note that for most species there is a lot of difference between precision and recall; this demonstrates an imbalance in the accuracy of predictions. This high variability in the results is not a good thing, and shows that there are some species that need a

more accurate description than what the classic SIFT offers. This fact is also testified by the confusion matrix shown in Figure 3.17: we can see that it works as a heatmap, where lighter colors indicate values close to 0, while darker ones indicate values close to 1.

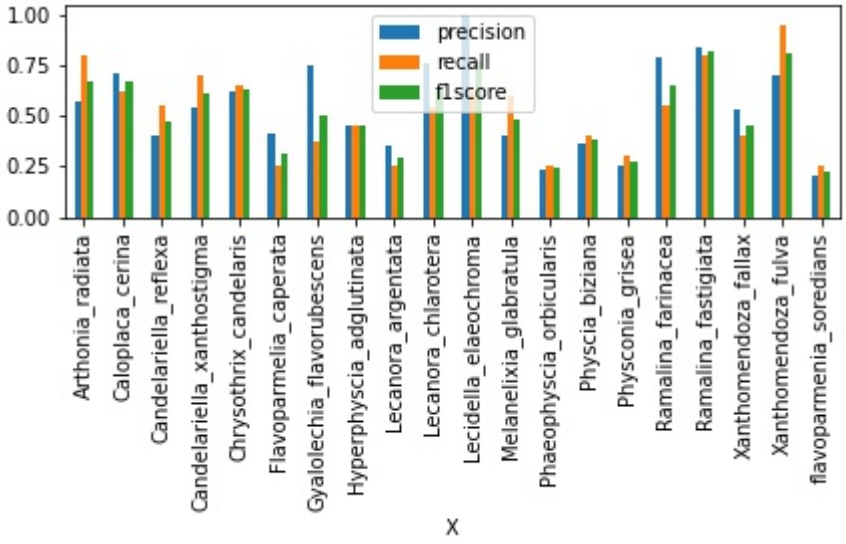


FIGURE 3.16: Variation of accuracy, precision, and recall within classes, using classic SIFT with Gaussian SVM model and 650 visual words.

3.4.2 Dense SIFT

Regarding Dense SIFT, on each image of the training set we applied an uniform grid to extract descriptors. The size of the grid is a very important parameter called *gridsize*: the smaller is this value, the greater will be the total number of descriptors for each image. In our experimentation, we empirically decided that *gridsize* $\in \{4, 8, 12, 16, 32, 64\}$. As we said in Section 3.1.4, Dense SIFT loses its scale invariance, so we decided to simply calculate descriptors at different scales, by taking into consideration convolution with Gaussian with different σ's. We set empirically that $\sigma \in \{0.53, 1.06, 1.6, 2.13, 2.7\}$. As mentioned before, Each descriptor is a vector, so each image is a collection of vectors: to aggregate them, we applied *BOVW*, varying visual words from 200 to 900. For what concerns the classifier, we applied a SVM, equipped with Gaussian kernel.[2] In this first part, we used the opponent color space introduced by eq. (3.7), eq. (3.8), and eq. (3.9). Descriptors have been calculated using Matlab package *vl_feat* [12].

Varying the gridsize

In Figure 3.18 we can observe how *gridsize* impacts on the overall performance of the process: taking larger value of this parameter, we considered sparser descriptors, and so we lost image's details. Best results are obtained with the minimum possible value of *gridsize*: we achieved an accuracy of ~ 0.89 with 600 visual words. Precision and Recall followed the same behaviour of accuracy. We can notice that considerable decreasing of performance starts when we employed *gridstep*> 8: when this

[2]Parameters of the model have been chosen empirically.

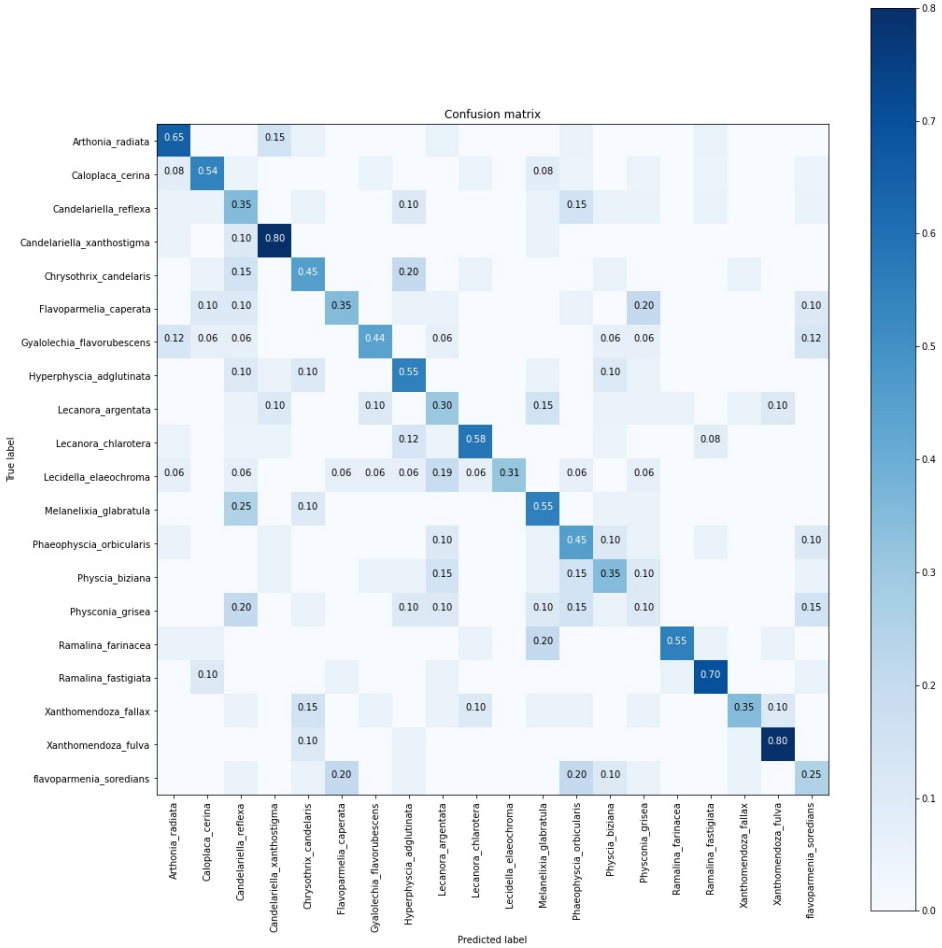


FIGURE 3.17: Confusion matrix obtained using classic SIFT with Gaussian SVM model and 650 visual words.

fact happens, descriptors are no longer overlapping, in the sense that they describe disconnected patches of an image, thus losing information.

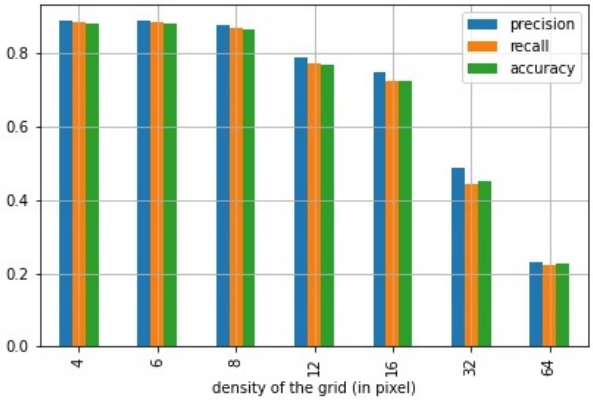


FIGURE 3.18: Variation of accuracy, precision and recall vary with gridsize. We used a Gaussian SVM and 600 visual words.

Varying the number of visual words

We checked if the number of words considered in BOVW is an important parameter, in terms of accuracy: to do so, we fixed the *gridstep* to 4, and we built a model using different values of visual words. The trend of the results is shown in Figure 3.19: as we can see, there is not a specific direction of improvement, and there are no major differences in performance. This fact suggests that it is not such an important aspect to take into consideration. Knowing that best results are obtained with *gridstep*$=4$, we can conclude that we achieved best results with 600 visual words, obtaining an accuracy of ~ 0.89.

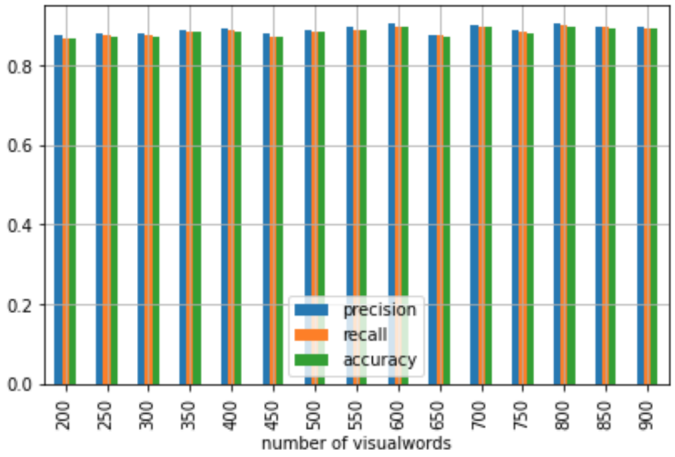


FIGURE 3.19: Variation of accuracy, precision and recall w.r.t. number of visual words.

Take a specific model

In this paragraph we described in details a specific model, in order to understand if all lichen's species are well recognized. We took the best model in terms of accuracy, fixing the parameters in the following way:

1. *gridsize* equal to 4.
2. Number of visual words equal to 600.
3. SVM equipped with Gaussian kernel.

Figure 3.22 represents the confusion matrix of this model: while most of the species are perfectly recognized, there is some confusion between *Candelariella reflexa* and *Chrysonthrix candelaris*. The probable reason of this uncertainty is that these two classes are very similar, both for color and texture shape, so it turns out to be very difficult to discriminate them. This difficulties is reflected also in Figure 3.21, in which we analyzed all the different categories: for each of them, we plotted F1-score, precision, and recall, in order to study how much our model is able to manage the positive and false negative. However, this model achieved an average accuracy of $\sim$ 0.89, that represents the best results of this thesis. We also replaced SVM model with a KNN one, to check if this approach improves results. Figure 3.20 what happens when we change the parameter k: we can observe that best results are achieved with only one neighbor, and performance decreases as k increases. This reflects that, even if we manage to reach good results, data are not perfectly grouped in the N-dimensional (where N is the dimension of the feature vectors) space and so, given

a point, it can have neighbors from different classes. By setting $k = 1$, we got a classification accuracy of ~ 0.83, which is a good result, but it did not exceed that obtained using a SVM model.

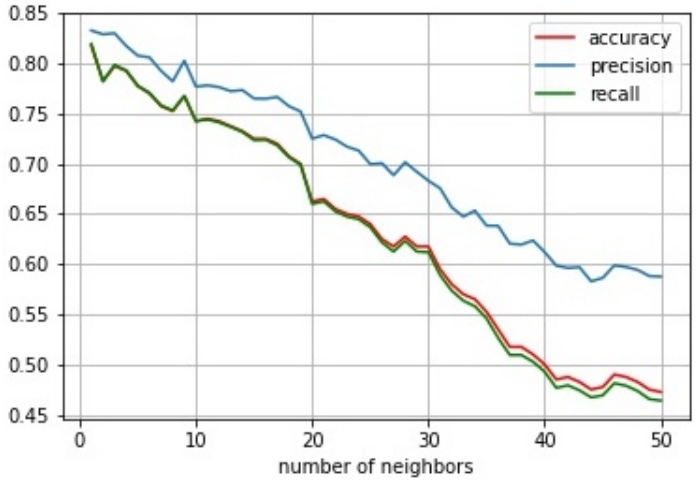


FIGURE 3.20: Variation of accuracy, precision, and recall using KNN with different numbers of neighbors.

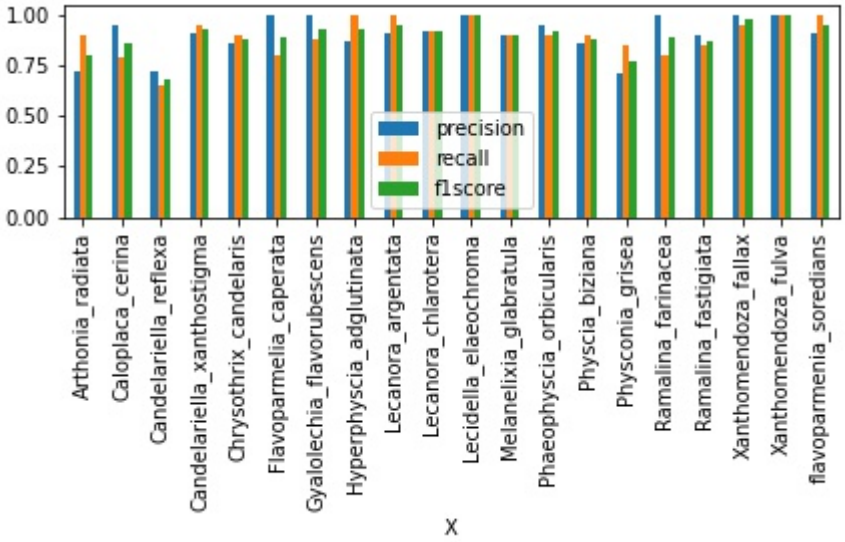


FIGURE 3.21: Variation of precision, recall and F1-score of the same model of figure 3.22

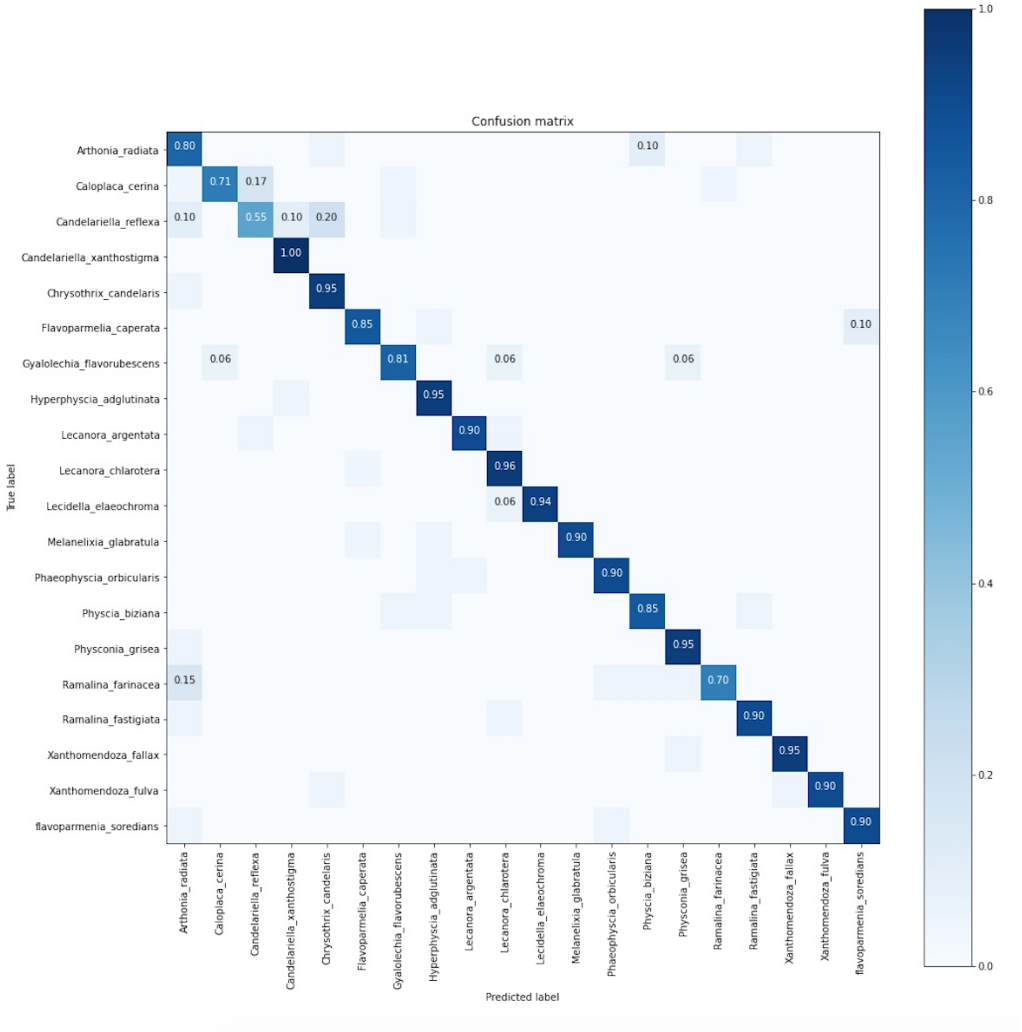


FIGURE 3.22: Confusion matrix of the model with *gridsize* = 4, visual words = 600, and Gaussian SVM.

Varying the color channels space

Until now we have always considered the opponent color space, in order to isolate the luminance component: the next step is to change this factor and use RGB images. We fixed the number of visual words to 600, since we have ascertained that it is not such an important parameter for the accuracy of the process, and we changed the *gridsize* parameter. Figure 3.23 represents how accuracy, precision, and recall change with *gridsize*, using RGB color space. We observe that even if results are quite good, achieving ~ 0.79 of accuracy when *gridsize* is equal to 4, they are worse than those obtained with the opponent color space: RGB indeed does not take into account luminance variation. Opponent color space, on the other hand, provides some brightness invariance, which is very important, especially in real situations.

Varying Scales of descriptors

Until now we have always extracted descriptors at different scales as if they came from different images; in this section we applied the *scale propagation* technique introduced in Section 3.1.4. We exploited both the *geometric* and the *image-aware* approach. As we have already verified, the number of visual words does not affect

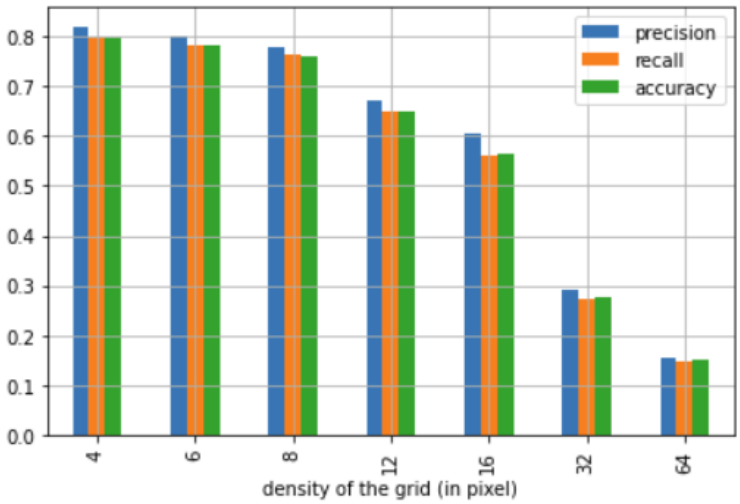
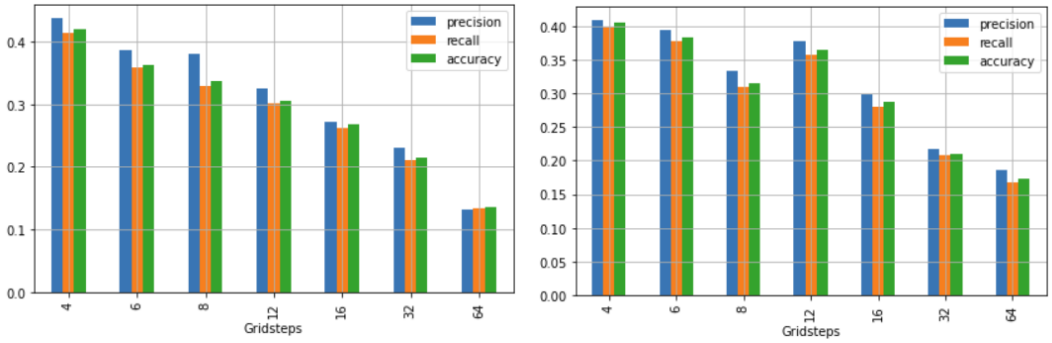


FIGURE 3.23: Variation of accuracy, precision and recall with *gridsize* when we used the RGB color space, with a Gaussian SVM model.

accuracy, so we fixed this value to 500, while we changed the *gridstep* in the same way we did in Section 3.4.2. For classification, we used a SVM model, equipped with Gaussian kernel. Figure 3.24a and Figure 3.24b show results using geometric and image-aware scale propagation, respectively. First of all, in both cases best results were achieved when $gridstep = 4$; this fact is not a surprise, since it has already happened in previous experiments. Moreover, there is not all this difference between the two techniques. Obtained results are far worse than those obtained previously, with a maximum accuracy of ~ 0.42. This fact demonstrates two thing:

1. Our way of propagating the scale from key points is not effective for our dataset. The worsening of performance demonstrates how information is lost with these methods.

2. In order to achieve good results, it is necessary to consider lichens at different scales, as they were different images.



(A) Model with Geometric scale propagation. (B) Model with Image-aware scale propagation.

FIGURE 3.24: Variation of precision, recall and accuracy using scale propagation techniques, when we change the gridsize. As classification model, we used a Gaussian SVM.

3.4.3 LBPs

LBPs have 2 parameters to handle: the *number of points P* and *the radius r*: the first determines how many points we consider to describe a neighborhood, while the second represents the scale. We empirically fixed the first value to 8, while we changed r in order to extract the best scale for our problem. We also considered the rotation invariant version of *LBPs*. For classification, we adopted a SVM equipped with Gaussian kernel. It is quite straightforward that as r increases, the description's capability of *LBPs* decreases, as we considered points with lower spatial correlation with the center of the descriptor. Figure 3.25 represents well what we just explained. We can notice that the maximum accuracy achieved is ~ 0.54, which is less compared to the results of Dense SIFT, despite the fact that *LBPs* are one of the most used descriptors for texture classification: probably this is because our dataset is too complex to be described with *LBPs*. A possible approach to reach better results is to consider more values of r. In practice, we extracted different feature vectors with different r, and we concatenated them: in this way we considered points at various distance from the center, hoping to capture more details of a batch. Unfortunately, this attempt did not provide better results (as we can see in Table 3.1), probably because, as r increases, the distribution of points becomes more scattered, decreasing the predictive capacity of *LBPs*.

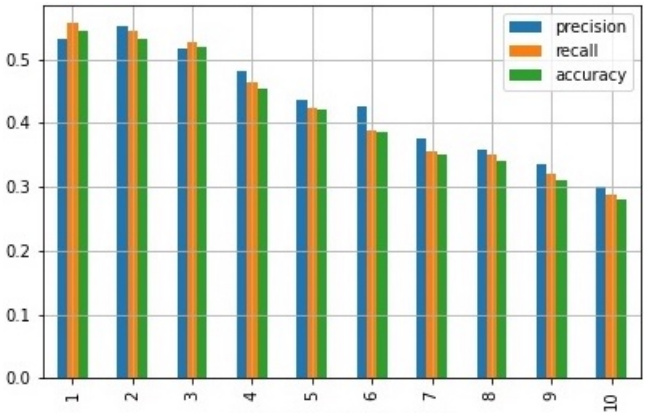


FIGURE 3.25: Variation of precision, recall and accuracy using *LBPs*, when we change the radius. As classification model we used a Gaussian SVM. On the x-axes there is the value of the radius r.

radius set	accuracy
(1,2,3)	0.46
(1,2,3,4)	0.43
(1,2,3,4,5)	0.39

TABLE 3.1: Variation of accuracy with respect to sets of r's used for descriptors.

3.4.4 Gabor features

Our filter bank has been constructed by changing m orientations and n scales. In particular, different orientation θ are calculated in the following way:

$$\theta = \frac{j}{m}\pi \qquad j = 0, \ldots, m-1$$

while scales are calculated as follows:

$$\Omega = \frac{0.25}{(\sqrt{2})^i} \qquad i = 0, \ldots, n-1$$

$$\alpha = \beta = \frac{\Omega}{\sqrt{2}}$$

In the first model we set $m = 8$ and $n = 5$, as in [7]: this is a standard setting for texture classification problems. For classification, we adopted SVM model, equipped with Gaussian kernel. Results of this model are shown in Figure 3.26 and Figure 3.27: the accuracy is ~ 0.806, which is quite close to the performance achieved with dense SIFT. There are some lichens which are almost perfectly discriminated, like *Xanthomendoza fulva* and *Chrysothrix candelaris*, while for others, the results are not as good as expected: this variability can be seen both in the confusion matrix on Figure 3.26, and in Figure 3.27, where precision, recall, and F1-score are plotted for each different class of lichens. We also tried to change both orientations and scale parameters, in order to find the best combination of them, and to verify if they affect performance. We changed both m and n from 1 to 8, and also we tried different type of classifiers, which can be a Gaussian SVM or a KNN with $k = 1$. Results are shown in Table 3.2 and Table 3.3: the first one represents accuracy of the model obtained with Gaussian SVM, while the second shows kNN-based model results. In the first scenario, we can see that best results are obtained with $m = 8$ and $n = 3$. However, we can notice that, in general, when we increase m, the accuracy always improves, while this is not always true for n. Slightly more random is the trend in the nearest neighbor case, where best result is achieved with $m = 7$ and $n = 7$.

m \ n	1	2	3	4	5	6	7	8
1	0.52	0.57	0.63	0.67	0.66	0.68	0.70	0.71
2	0.57	0.68	0.74	0.75	0.77	0.74	0.77	0.76
3	0.64	0.67	0.75	0.76	0.75	0.72	0.74	0.74
4	0.66	0.68	0.76	0.78	0.78	0.76	0.76	0.77
5	0.70	0.71	0.8	0.79	0.81	0.79	0.8	0.8
6	0.69	0.71	0.82	0.80	0.81	0.80	0.81	0.8
7	0.69	0.73	0.82	0.81	0.81	0.80	0.8	0.8
8	0.71	0.72	**0.83**	0.81	0.81	0.79	0.79	0.79

TABLE 3.2: Table with accuracy of the model described in 3.4.4 when we change orientations m and scales n: all models are based on Gaussian SVM.

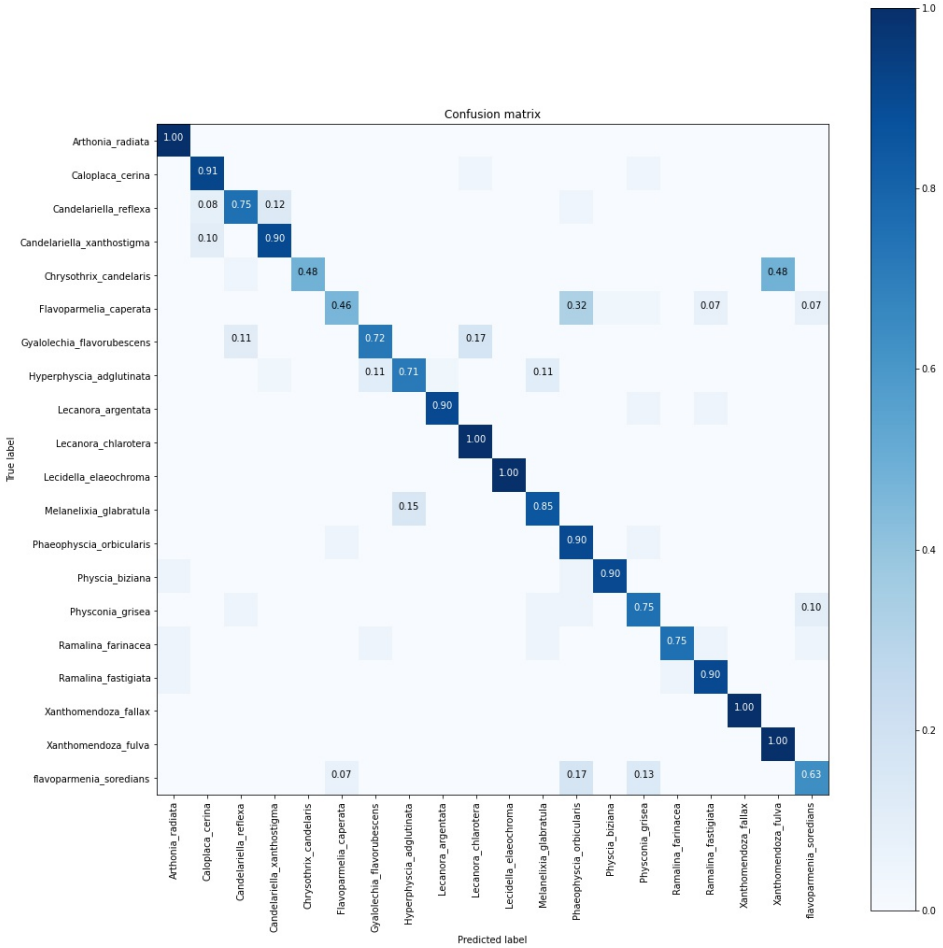


FIGURE 3.26: Confusion matrix of the model defined in section 3.4.4.

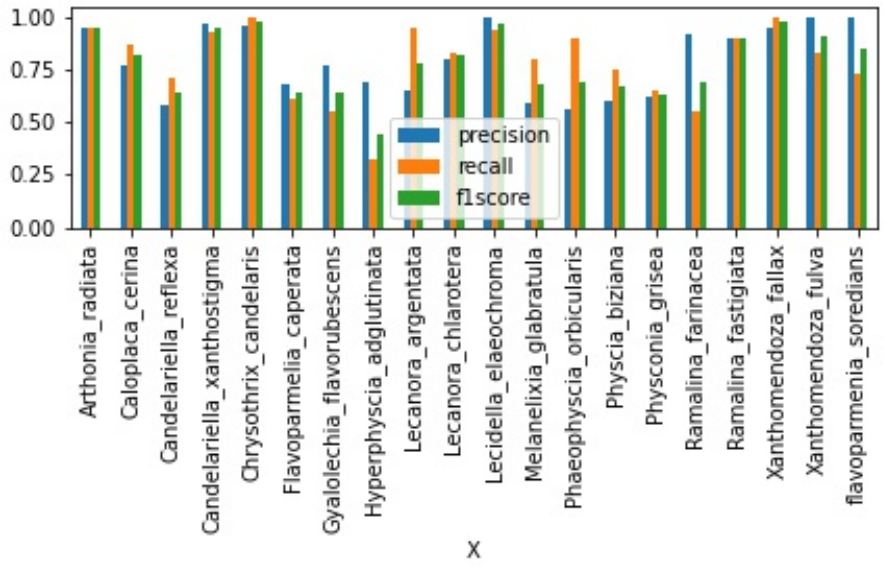


FIGURE 3.27: Distribution of precision, recall and f1 score for the model described in 3.4.4.

m \ n	1	2	3	4	5	6	7	8
1	0.43	0.47	0.58	0.57	0.62	0.59	0.62	0.59
2	0.48	0.54	0.61	0.62	0.69	0.64	0.67	0.65
3	0.50	0.55	0.62	0.64	0.70	0.65	0.70	0.68
4	0.57	0.59	0.67	0.68	0.73	0.70	0.74	0.73
5	0.57	0.62	0.71	0.70	0.73	0.72	0.73	0.74
6	0.59	0.62	0.70	0.73	0.74	0.75	0.74	0.75
7	0.60	0.65	0.73	0.73	0.75	0.75	**0.76**	0.70
8	0.57	0.65	0.73	0.74	0.74	0.73	0.74	0.74

TABLE 3.3: Table with accuracy of the model described in 3.4.4, when we change orientations m and scales n: all models are based on 1NN.

3.4.5 BGPs

Last descriptors that we used is some kind of mixture of *Gabor filtering* and *local binary pattern*. We decided to adopt the same parameters used in [8], where they exploited the same descriptors on *CUReT* dataset. In particular, we used Gabor filters of size 8 at 3 different resolutions $\{(\lambda_i, \sigma_i) : |i = 0, 1, 2\}$, which are:

1. $(\lambda_i, \Omega_i) = (1.3, 1.42)$.
2. $(\lambda_i, \Omega_i) = (5.2, 0.4)$.
3. $(\lambda_i, \Omega_i) = (22, 0.22)$.

For each resolution, we took 8 different orientations:

$$\theta = \{\theta_j = j\pi/8 : |j = 0, 1, \ldots, 7\}$$

In this way, the resulting normalized histogram has 36 bins. At each resolution, 2 frequency histograms were created (one from real part Gabor filter and one from imaginary part Gabor filter): Thus we had 6 histograms to be concatenated, obtaining a large histogram of 216 bins for each color channel. Since we worked with RGB images, we ended up with a big frequency histogram of 648 bins. For classification, we used a SVM model, equipped with Gaussian kernel. We obtained a test accuracy of ~ 0.81, and we can observe the corresponding confusion matrix on Figure 3.29. As we can see, most of lichens are well recognized, even if there are some exceptions, i.e *Flavoparmenia Caperata* is confused with *Phaeophyscia Orbicularis*. This fact can be seen also in Figure 3.28, where we can notice how recall for *Flavoparmenia Caperata* is lower respect other species: this is because a lot of images belonging to this class have been misclassified.

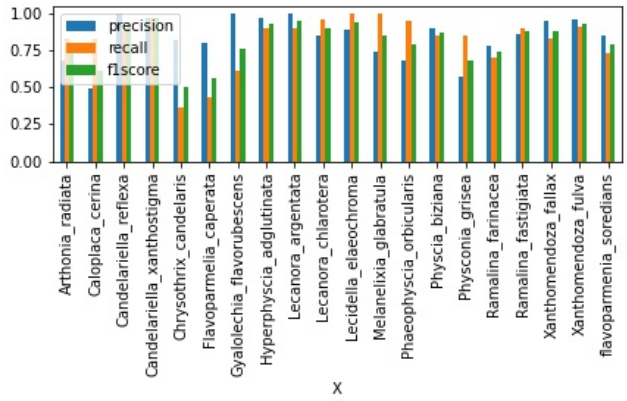


FIGURE 3.28: precision, recall and F1-score of the model described in 3.4.5.

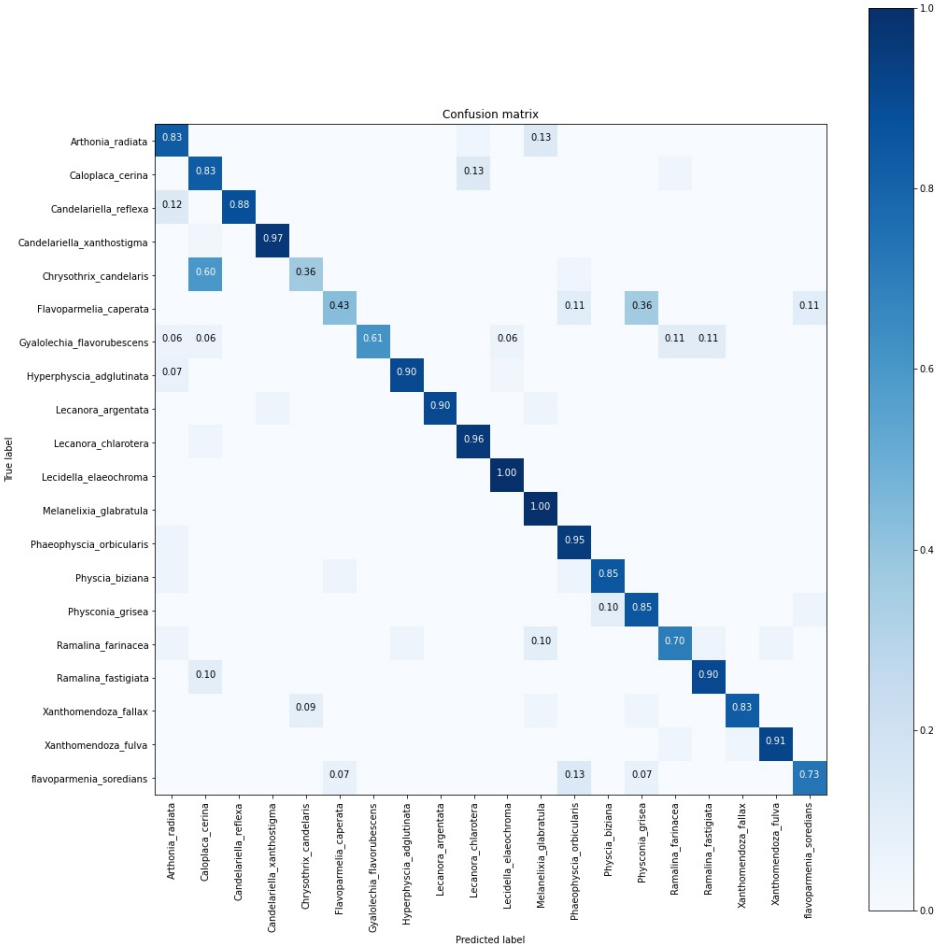


FIGURE 3.29: Confusion matrix of the model described in Section 3.4.5

3.4.6 Comparison between descriptors and final consideration

We explored different descriptors in order to capture the main discriminatory characteristics of lichens, and perform classification. Table 3.4 synthesizes the main results we obtained, showing that a dense computation like Dense SIFT outperforms a sparse one: this fact is due to the necessity to describe almost every part of an image, especially when the dataset is not so rich. Even if results may appear satisfactory, we focused on classification, which is a simplified version of our original problem. In practice, we could use process described in Section 3.3.3 in the following way: we take a test image $\mathbf{x}$, we divide it in patches with a grid, and we classify each single cell: in this way, with a good model, we could recognize different lichens in an image. The whole process is shown in Figure 3.30, and Figure 3.31 shows an example where we applied it. In this case, we created an artificial image formed by 4 different lichens: *Caloplaca Cerina*, *Lecanora Chlarotera*, *Xanthomendoza Fulza*, and *Ramalina Farinacea*. We adopted dense SIFT as descriptors, with *gridstep*= 4, visual words= 600, and for classification we trained a Gaussian SVM. It is clear that this represents a very simplified problem, as we dealt with only 4 different lichens which are very separated from each other, and moreover we did not consider presence of background.

Descriptors	Best accuracy	Process details
LBPs	0.54	Gaussian SVM, r=1
Classic SIFT	0.52	Gaussian SVM, visual words=500
Multi-scale dense SIFT	**0.89**	Gaussian SVM, visual words=600, gridsize = 4
No scale dense SIFT	0.70	Gaussian SVM, visual words = 600, gridsize = 4
Gabor-based	0.83	Gaussian SVM, m = 8, n = 3
BGPs	0.81	Gaussian SVM

TABLE 3.4: Table that represents best results obtained with the different descriptors explored in this thesis.

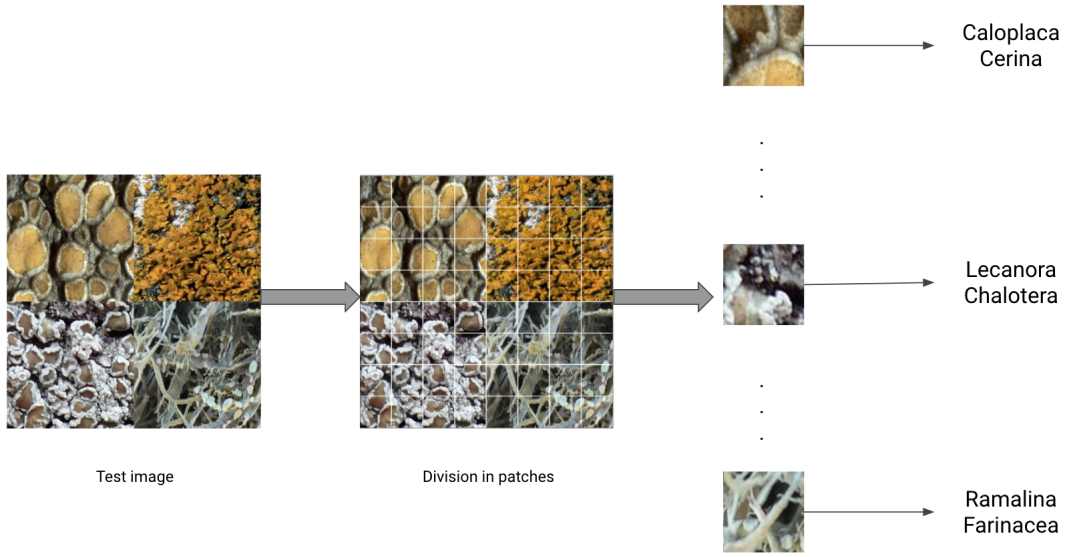


FIGURE 3.30: Pipeline of the process to classify a test data $\mathbf{x}$: for each image's patch, we perform classification.

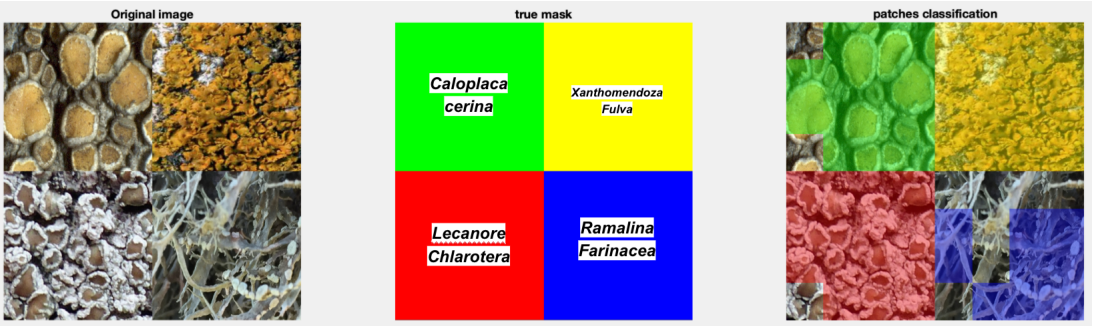


FIGURE 3.31: On the left: the artificial image composed by 4 different lichens. On the center: the true mask which delimits distinct classes. On the right: prediction of our process for each patch.

Chapter 4

Deepnet descriptors for patch classification

In the last few years convolutional neural networks, or CNNs, have represented a real turning point in computer vision. Thanks to the increasing amount of data and the growing available computational power, CNNs achieved incredible results in terms of accuracy, and surpassed classic methods in many classification problems: in 2012, for example, *AlexNet* outperformed classic SIFT descriptors in classifying Imagenet dataset[1]. However, this advanced approach typically needs a lot of data. When the dataset is not large enough, CNNs often break down due to *overfitting*: the process is not able to generalize, and therefore to make predictions on new data. To overcome this problem, different solutions have been proposed during years, i.e. data augmentation, transfer learning, and few-shot learning. in Section 4.1 and 4.2, we first introduce theory of CNNs, with main focus on the mentioned above technique; then, in Section 4.3, we present all the models we trained to classify lichens patches, emphasizing both the benefits and the limitations of this approach.

4.1 Introduction to CNNs

A CNN is a special type of artificial neural network, or ANN, that receives images as inputs. This structure has been widely used for problems like image classification, object detection, and image segmentation. CNN is organized in feed-forward layers, that are mainly part of the following categories

1. Convolutional layers.
2. Pooling layers.
3. Fully connected layers (FCL).

while the last typology is the same that we have in feed-forward ANN, we spend few lines to introduce the first two types of layers, that represent the core of the CNNs.

4.1.1 Convolutional layer

Each convolutional layer contains a set of filters, called *convolutional kernels*, whose parameters are learnable with data. Each filter can be small (usually it goes from 3×3 to 11×11), but extends through the full depth of the input. In each layer, input is convolved with these kernels, and results are passed through a non linear activation function. The main characteristics of convolutional layers are:

[1] Imagenet is a renowned database with thousands of images for each class.

1. *Local connectivity*: Each neuron in a specific layer is connected only to a subset of neurons of the preceding one. This leads to have a smaller amount of connections compared to dense layers. Moreover, spatial arrangement is preserved.

2. *Weight sharing*: Connections share weights. This can be interpreted with the fact that weights form spatial filters, that are applied to different regions of the input.

3. *Multiple feature maps*: To extract more features, each layer acts as a bank of filters.

Figure 4.1 represents the different approach of a convolutional layer, compared to a dense one.

4.1.2 Pooling layer

Between two subsequent convolutional layers, it is a common thing to reduce the spatial resolution of the inputs, in order to cut down the amount of network's parameters without losing information: this is performed by the so-called *pooling layer*. Figure 4.2 shows how this layer works. Usually, Pooling layer exploits a 2×2 filters with stride 2: in this way, we halve the size of the input by taking the maximum (or the average) of sub-regions. Another particular type of pooling is called *adaptive pooling*: here, instead of choosing explicitly the size and the stride of the filters, we impose the size of the output, and the stride and kernel-size are automatically selected to adapt to the needs.

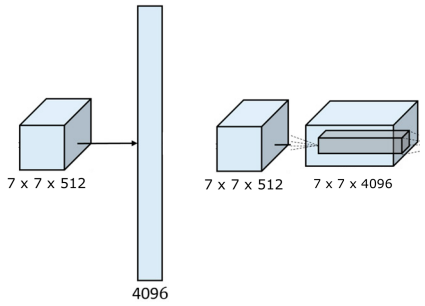


FIGURE 4.1: Comparison between FCL (on the left) and convolutional layer (on the right), when we deal with the same $7 \times 7 \times 512$ input.

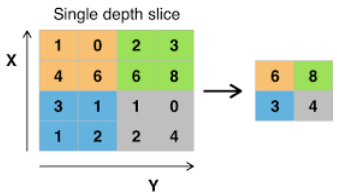


FIGURE 4.2: Example of max-pooling with a 2×2 filter and stride equal to 2. For each crop, we take only a single value, that can be the maximum or the mean (in this case, the maximum). In this way we reduce the entire crops to a single number, scaling down the size of the input.

4.2 CNN with few data

In practice, CNNs have millions of parameters to learn, and this is performed through *minimization of a specific loss function with the backpropagation algorithm*[2]. If we have a small dataset, running a large number of minimizing iteration can result in overfitting, that indicates when *the model fits too much training data, but it is not able to predict any future unseen observation*: this situation is represented in Figure 4.3. In order to reduce this problem, several approaches can be adopted: we may work on the dataset with *data augmentation* (DA), on the model structure by adding specific layers specialized to reduce overfitting, or on the overall process by applying techniques like *transfer learning* and *few-shot learning*.

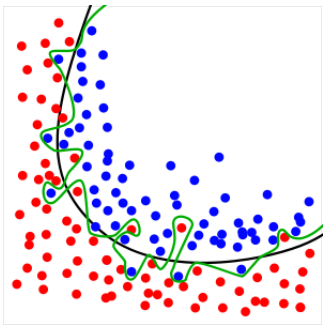


FIGURE 4.3: The green line represents the overfitted version of the model, that follows to much the training data. Black line represents a regularized model, that instead is more capable of predicting new unseen input.

4.2.1 Working on the dataset: data augmentation (DA)

With *data augmentation*, or DA, we mean the act of increasing the size of the dataset by applying some transformation: in this way, we obtain slightly modified images, that reduce overfitting. DA is based on the assumption that slightly transformed images belong to the same class of the original. Moreover, DA can help to achieve specific invariances, using transformations like rotation, cropping, or flipping. In choosing the appropriate transformations, we must pay attention that they preserve the data labels. For our problem, we performed following transformation:

1. *Rotation* augmentations are performed by rotating an image right or left, on an axis between 1° and 359°: it is a totally safe type of augmentation, since lichens have a shape that does not depend on rotation. An example is shown in Figure 4.4.

2. *Flipping* an image means reversing the rows or columns of pixels, in the case of a vertical or horizontal flip, respectively. For the same reasons as before, it is a quite safe type of transformation. An example is shown in Figure 4.5.

3. *Shearing* (or *Skewing*) transformation slants an image. You can apply shear along x-axes (X-shear), or along y-axes (Y-shear): in both cases only one coordinate is changed. This transformation simulates different points of view of the same object; this can be a good thing for our dataset, even if we have to pay attention not to distort images too much. An example is shown in Figure 4.6.

[2]Explanation of this process is out of the scope of this thesis.

4. *Scaling* transformation consists on zooming in or zooming out images. This is a very important invariance, because it allows us to recognize lichens at different scales. An example is shown in Figure 4.7.

FIGURE 4.4: On the left we have the original image, on the right the 90 degree clock-wise rotation of it.

FIGURE 4.5: From left to right: the original image, horizontal flipped image, vertical flipped image, and both vertical and horizontal flipped image.

FIGURE 4.6: On the left we have the original image. On the right we have the shear version of it, with angle equal to 0.2 in radians.

FIGURE 4.7: On the left we have the original image. On the right we have zoomed version of it.

4.2.2 Working on the model: Dropout layer

Another possible way to improve generalization capability of the process is to act directly on the structure of CNNs, by adding some specific layers that help to reduce overfitting. In particular, we focus our attention to *Dropout layers*. The word Dropout refers simply to ignore some random neurons during training, by applying a random mask to all of the input and hidden units in the network: ignored neurons are not considered in the training process. Specifically, starting from a base network, at each training step nodes are dropped out with probability $1 - p$ (p is a

fixed parameter): basically, what we obtain is an ensemble of different subnetworks. A graphical representation of dropout is on Figure 4.8. At test time, we approximate the prediction of all the subnetwork by using the complete one, with weights scaled by a factor of p. Most important benefits of dropout layers are:

1. For each training step, we reduce number of trainable parameters.
2. We force the net to learn more robust features.
3. Dropout reduces the training time of a single epoch.

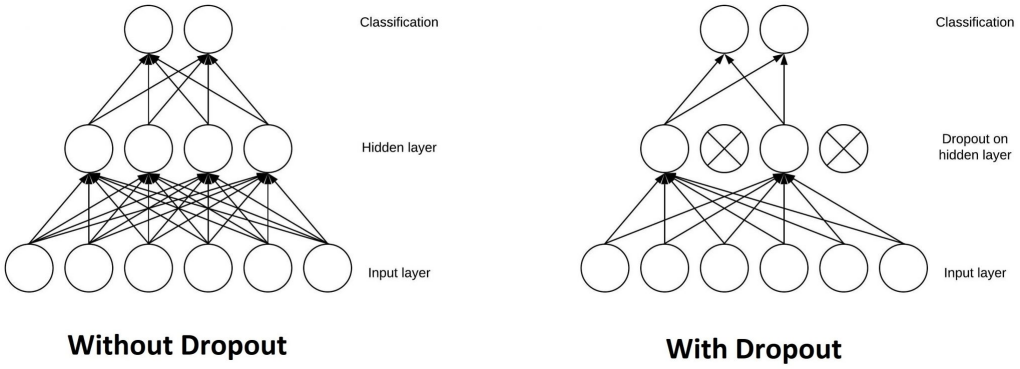


FIGURE 4.8: On the left we have the entire network with all neurons. On the right we have a possible sub-net after applying dropout: Units with the cross have been removed from the training process.

4.2.3 Working on the process: transfer learning (TL)

When we have to deal with a small dataset, it is almost impossible to build a model from scratch: in this context, *transfer learning*, or TL, comes to the rescue. The idea behind TL is that some learned features are important regardless of the specific task: so one can solve a specific job by exploiting a huge model, pre-trained for a related task with a greater dataset. TL is not a proper deep learning technique, but it is a generic way of thinking, in which previous knowledge from trained models is used to tune newer ones, in order to solve specific problems (see Figure 4.9). In our framework, the idea is to take a big network, trained on a huge open-source dataset called Imagenet. At this point, we follow three different ways:

1. Fix all the weights of the net, and treat it as a simple *feature extractor*, that will be fed to a SVM classifier. This approach is used especially with very small dataset.
2. Fix all the weights of the net, and treat it as a simple *feature extractor*, which will be the input of an ANN. This approach is used with medium-sized dataset, as we need some data to train the classification net.
3. Don't fix the weights of the net, and attach to it some dense layers for classification.

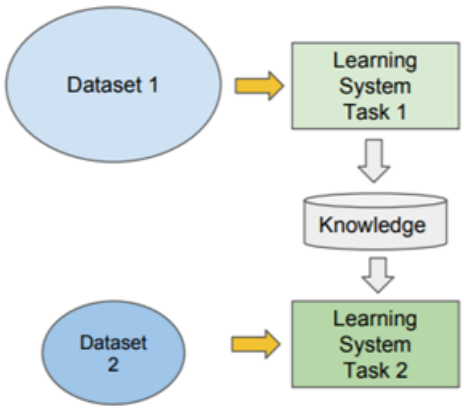


FIGURE 4.9: TL framework.

VGG16

VGG16 is one of the most famous CNN, introduced by Simonyan and Zisserman [13], in 2015. Its goal was to improve classification's performance on *ImageNet*, and it outperformed Alexnet replacing larger kernel filters with multiple 3×3 kernel-sized filters. The architecture of this net is depicted in Figure 4.10. We use VGG16 structure as base network for TL.

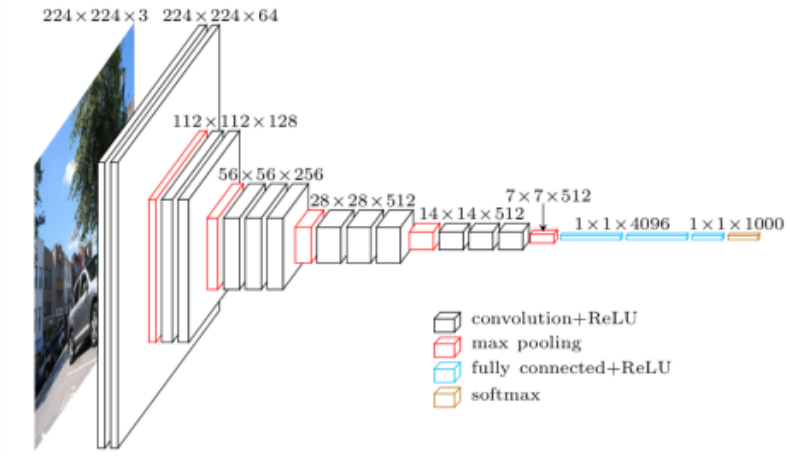


FIGURE 4.10: Architecture of VGG16.

4.2.4 Few-shot learning (FSL)

Few-shot learning, or FSL, refers to the ability for an algorithm to generalize well with few examples: this skill, which is really common for humans, is not so easy to achieve when we are dealing with computers. First of all, let's give the following definition:

Definition 4.2.1 *Given:*

1. *A support set consisting of:*
   (a) *N class labels.*
   (b) *K labelled images for each class (usually no more than* 10*).*

2. *Q query images.*

An N-way K-shot classification problem is when we want to classify Q query images among N classes, with K samples for each class in the support set.

There are several approaches to this problem: we focus on the *metric-learning* one, that basically means to learn a distance function over objects. In general, these algorithms classify query samples based on their similarity to the support samples. In practice, a specific CNN is trained to output an image embedding vector, that is later compared to other embeddings, in order to predict the right class.

Siamese neural network

Siamese neural network, or SNN, is a particular structure used to naturally compute similarity between 2 inputs, introduced by Koch, Zemel, and Salakhutdinov [14]. We can think about SNN as a net that can discriminate between the class-identity of image pairs, so it learns to identify input pairs according to the probability that they belong to the same class or not. In practice, it receives 2 images as input, and it produces a *similarity score*, that ranges from 0 (no similarity) to 1 (perfect similarity), as output. SNN is composed by 2 identical CNNs, joined by an energy function at the top: however, they accept 2 distinct inputs. The parameters of the 2 networks are tied together, to guarantee that similar images produce similar embedding vectors in the feature space. Another important thing is that these 2 neural networks are symmetric, so that it is the same thing if we present the same couple of images, but in the opposite order. Specifically, each net of a SNN consists of a CNN with N layers, each with T_n units: we set that

1. $\mathbf{h}_{1,n}$ is the hidden vector in layer n for the first twin net.
2. $\mathbf{h}_{2,n}$ is the hidden vector in layer n for the second twin net.

The features extraction part of the model consists of a sequence of convolutional layers. After each of them, we apply a ReLU activation function to the output of feature maps, and a max-pooling layer with both filter stride and size equal to 2. The *Kth* filter can be described in the following way

$$a^k_{1,n} = maxpool(\max(0, \mathbf{W}_{n-1,n} * \mathbf{h}_{1,n-1} + \mathbf{b}_n), 2)$$

$$a^k_{2,n} = maxpool(\max(0, \mathbf{W}_{n-1,n} * \mathbf{h}_{2,n-1} + \mathbf{b}_n), 2)$$

where $\mathbf{W}_{n-1,n}$ represents the weights matrix of feature maps for layer n, and $*$ is the convolutional product. In the original paper, after the convolutional part, the weighted L1 distance between the extracted flattened features is computed. However, in our problem this structure is not effective, so we changed the architecture by replacing the L1 distance part with a dense ANN. Overview of the architecture is on Figure 4.11.

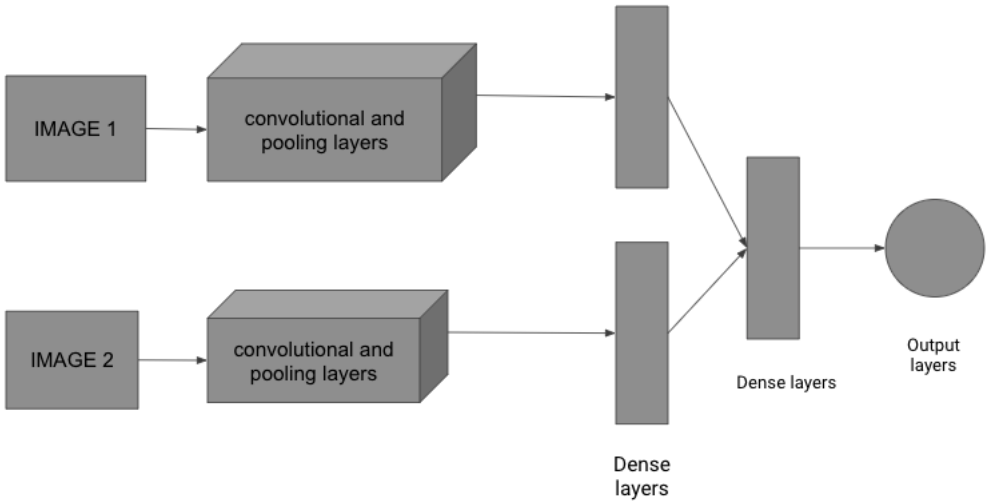


FIGURE 4.11: Scheme of SNN: it receives 2 images, extract features with same filter, pass through dense layer and return the similarity score.

4.3 Experiments and results

In this section we tested techniques based on CNNs to classify lichens. The dataset was treated as described in Section 2.2, apart from the fact that we downsampled images to $100 \times 100 \times 3$, preserving the aspect ratio. Experiments were organized in three different parts:

1. *Scratch net*: We built different networks from scratch, in order to try to obtain valid results without pre-training.
2. *TL network*: We exploited theory of TL, to adapt pre-trained network to our purpose.
3. *Siamese networks*: we took a look to SNN, to extract similarities between patches.

4.3.1 Models from scratch

We implemented three different networks, that are deeper and deeper in order to facilitate the learning of more abstract features. In particular, the last one had the same structure of VGG16. We performed DA, explained in Section 4.2.1, to enlarge dataset.

Model A

First model is shown in Figure 4.13a. It receives input of dimension $(BS, 100, 100, 3)$, where BS is the batch-size, and it is formed by:

1. A first block with:
   (a) A convolutional layer of 8 filters, with kernel size equal to 3 and ReLU activation function.
   (b) A max-pooling layer with strides equal to 2.
2. A second block with:

(a) A convolutional layer with 16 filters, with kernel size equal to 8 and ReLU activation function.

(b) A max-pooling layer with strides equal to 2.

3. A dropout layer with $p = 0.3$.
4. A convolutional layer with 32 filters, with kernel size equal to 8 and ReLU activation function.
5. A flattening layer.
6. A dense layer with 128 units and ReLU activation function.
7. A batch-normalization layer.
8. An Output dense layer with 20 units and Softmax activation function.

We used categorical-crossentropy as loss function, and Adam as optimizer, with learning rate equal to 10^{-4}. We trained the net for at maximum of 50 epochs, using early stopping to fight overfitting. Accuracy and loss are shown in Figure 4.12: we can observe that train loss always decreases, while test loss stops improving considerably after $\sim$ 30 epochs, and then it starts to oscillate: this fact can be seen also in the accuracy plot, where test trend is always smaller that train one, and starts to flatten after $\sim$ 10 epochs. Moreover, we did not achieve same results as in Section 3.4.2, both for the train and test set. These performance suggested that this model did not manage neither to fight overfitting, nor to find suitable filters to obtain satisfactory results.

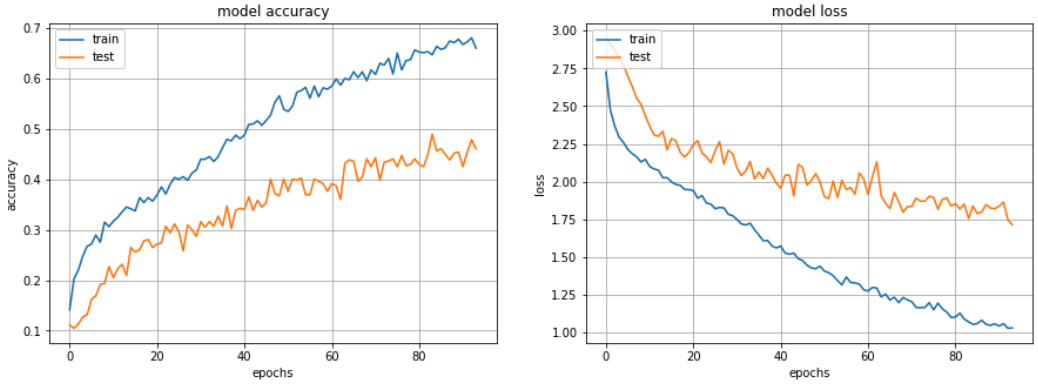


FIGURE 4.12: On the left: train (blue line) and test (orange line) accuracy. On the right: train (blue line) and test (orange line) loss.

Model B

The second model we built was not very different from model A: we only added one more block before flattening, formed by a convolutional layer with 512 filters of size 3 and ReLU activation function. For what concerns the other convolutional blocks, we just increased number of filters: in particular we had

1. 128 filters in the first convolutional layer, instead of 8.
2. 256 filters in the second convolutional layer, instead of 32.
3. 256 filters in the third convolutional layer, instead of 32.

Before flattening, we applied batch-normalization. Finally, the dense part was formed by

1. A dense layer with 128 units and ReLU activation function.
2. A dropout layer with $p = 0.5$.
3. An Output dense layer with 20 units and Softmax activation function.

We maintained the same optimizer and loss function as model A. Model B is represented in Figure 4.13b, while in Figure 4.14 we can observe accuracy and loss trend through epochs. For what concerns efficacy, situation is slightly improved compared to model A, achieving an accuracy of ~ 0.60. Nevertheless, there is a lot of oscillation in the test loss, and training performance is always better than test one, which is a symptom of overfitting.

Model: "sequential_8"

Layer (type)	Output Shape	Param #
conv2d_24 (Conv2D)	(None, 98, 98, 8)	224
max_pooling2d_16 (MaxPooling	(None, 49, 49, 8)	0
conv2d_25 (Conv2D)	(None, 24, 24, 16)	1168
max_pooling2d_17 (MaxPooling	(None, 12, 12, 16)	0
dropout_13 (Dropout)	(None, 12, 12, 16)	0
conv2d_26 (Conv2D)	(None, 10, 10, 32)	4640
flatten_8 (Flatten)	(None, 3200)	0
dense_16 (Dense)	(None, 128)	409728
batch_normalization_18 (Batc	(None, 128)	512
dense_17 (Dense)	(None, 20)	2580

Total params: 418,852
Trainable params: 418,596
Non-trainable params: 256

(A) Structure of the model A.

Layer (type)	Output Shape	Param #
conv2d_39 (Conv2D)	(None, 100, 100, 128)	3584
max_pooling2d_30 (MaxPooling	(None, 50, 50, 128)	0
conv2d_40 (Conv2D)	(None, 50, 50, 256)	295168
max_pooling2d_31 (MaxPooling	(None, 25, 25, 256)	0
conv2d_41 (Conv2D)	(None, 25, 25, 256)	590080
max_pooling2d_32 (MaxPooling	(None, 13, 13, 256)	0
conv2d_42 (Conv2D)	(None, 13, 13, 512)	1180160
max_pooling2d_33 (MaxPooling	(None, 7, 7, 512)	0
batch_normalization_30 (Batc	(None, 7, 7, 512)	2048
flatten_14 (Flatten)	(None, 25088)	0
dense_30 (Dense)	(None, 128)	3211392
dropout_31 (Dropout)	(None, 128)	0
dense_31 (Dense)	(None, 20)	2580

Total params: 5,285,012
Trainable params: 5,283,988
Non-trainable params: 1,024

(B) Structure of model B.

FIGURE 4.13: Summary of the 2 CNNs used for classification.

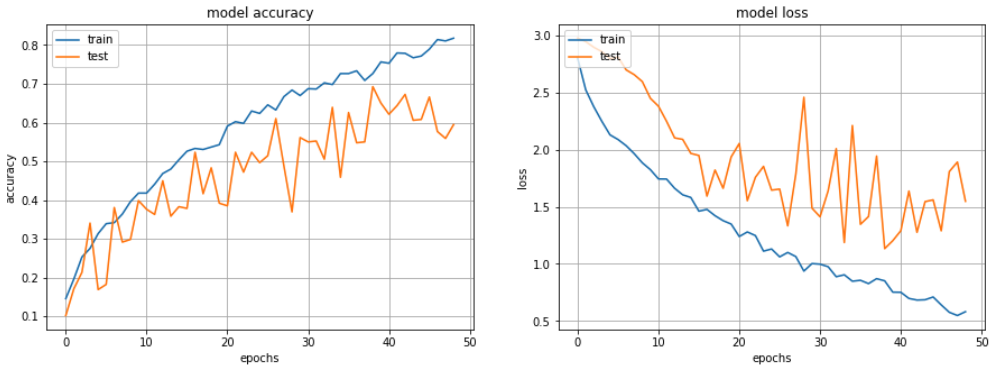


FIGURE 4.14: On the left: train (blue line) and test (orange line) accuracy. On the right: train loss (blue line) and test loss (orange line).

VGG16 based model

The last model we built had the same structure of VGG16, as far as is concerned the convolutional part. In addition to that, we concatenated following layers:

1. A flattening layer, to transform features into embedding vectors.
2. A dense layer with 512 units and ReLU activation function.
3. A dropout layer with $p = 0.5$.
4. A dense layer with 128 units and ReLU activation function.
5. An output layer with 20 units and Softmax activation function.

We did not use pre-trained weights. Structure is represented in figure 4.15. We adopted the same loss function and the same optimizer as the previous models, in order to be able to compare results. Figure 4.16 testifies how this model outperforms previous two: test accuracy is higher and keeps increasing during epochs, even if there are small oscillation: it achieved a maximum value of ~ 0.66. In general, we observe that, compared to previous models, test trend is more similar to the train one, both for accuracy and loss: this reflects that this model achieved an acceptable level of generalization, with few overfitting.

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 3, 3, 512)	14714688
flatten_5 (Flatten)	(None, 4608)	0
dense_11 (Dense)	(None, 512)	2359808
dropout_8 (Dropout)	(None, 512)	0
dense_12 (Dense)	(None, 128)	65664
dense_13 (Dense)	(None, 20)	2580

Total params: 17,142,740
Trainable params: 17,142,740
Non-trainable params: 0

FIGURE 4.15: Structure of the VGG16-based model.

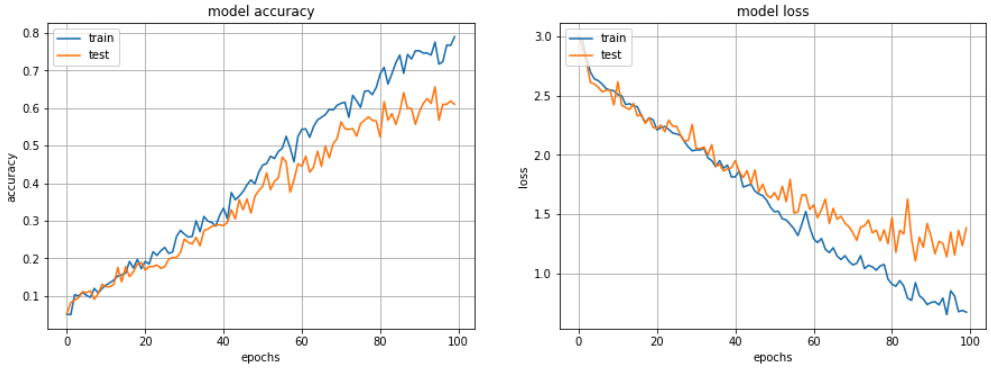


FIGURE 4.16: On the left: train (blue line) and test (orange line) accuracy. On the right: train loss (blue line) and test loss (orange line).

4.3.2 TL models

We experienced that, with scratch models, at a certain point it is not possible to learn better features, and test accuracy stops to increase. In the following, we verifies if TL can be a valid alternative.

VGG16 features and SVM

Our first approach was to use pre-trained VGG16 simply as a features extractor, that then were used in a SVM for classification. In other words, we did not train the CNN with our dataset, but we simply used it as it is. We tried three different kernels for the SVM model: polynomial (with degree ranging from 2 to 9), linear, and Gaussian kernel. We forced the VGG16 to receive images of size $100 \times 100 \times 3$ as input: output was therefore of size $3 \times 3 \times 512$, so the flattened features lived in a 4608-dimensional space. Table 4.1 shows mean accuracy for different kernels. Results are similar if we change kernels, with a slight improvement with a 8 degree polynomial one. Figure 4.17 represents confusion matrix for this specific model. We can observe that some classes are well recognized, while there are species where uncertainty reigns: an example is *Flavoparmenia Soredians*, that achieved an accuracy of just 0.37.

kernel	accuracy	precision	recall
linear	0.6525	0.6579	0.6620
rbf	0.5946	0.6270	0.6080
poly of degree 2	0.64	0.648	0.639
poly of degree 3	0.66	0.648	0.639
poly of degree 4	0.62138	0.6536	0.6350
poly of degree 5	0.6414	0.6637	0.6543
poly of degree 6	0.6481	0.6696	0.6598
poly of degree 7	0.6548	0.6873	0.6677
poly of degree 8	**0.6592**	**0.6873**	**0.6717**

TABLE 4.1: Results of SVM on VGG16 features for different kernels.

TL and dense networks

In this section we built a CNN with following structure, depicted in Figure 4.18:

1. A convolutional part represented by pre-trained VGG16.
2. A dense part represented by the subsequent ANN:
   (i) A flattening layer.
   (ii) A batch-normalization layer.
   (iii) A dense layer with 512 units and ReLU activation function.
   (iv) A dropout layer with $p = 0.5$.
   (v) A dense layer with 128 units and ReLU activation function.
   (vi) A dropout layer with $p = 0.3$.
   (vii) An output layer with 20 units and Softmax activation function.

For the training phase, we followed two possible ways:

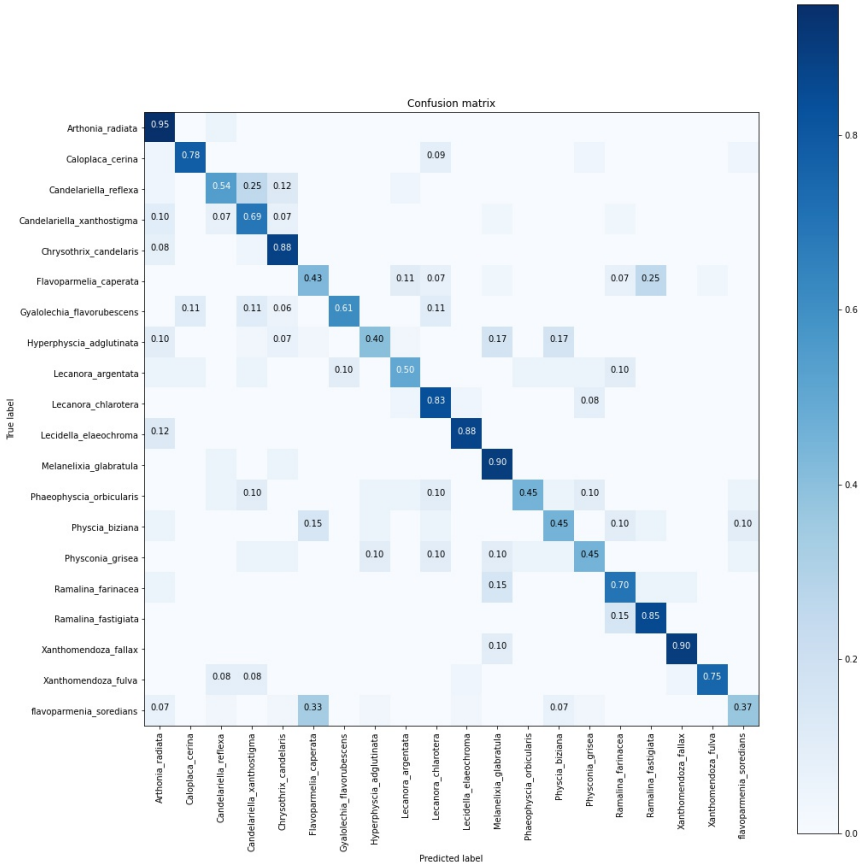


FIGURE 4.17: Confusion matrix with polynomial kernel of degree 8.

a) Train the whole CNN, including the VGG16 part, for a total of 17 151 956 trainable parameters .

b) Fix weights of VGG16 and fine-tune only the dense net, for a total of 2 437 268 trainable parameters.

We used categorical-crossentropy as loss function, and Adam as optimizer: we set to 200 the maximum number of epochs, with early stopping to fight overfitting. Results are shown in Figure 4.19 and Figure 4.20 for case a) and b), respectively: in particular, we achieved best results when we trained the whole net on our dataset, obtaining an accuracy of ~ 0.80. On the other hand, when we tuned only the dense part of our CNN, we achieved a correctness of $\sim$ 0.70, but train accuracy never outperformed test one (this is also valid for loss function), meaning that we did not obtain an overfitted model. Figure 4.21 represents an example of application of case a) on sample images.

4.3.3 SNN model

In this section we changed our point of view: instead of building a model that performs classification directly, we exploited SNN to calculate similarities between 2 images. For what concerns the shared convolutional layers, we adopted the same structure used in [15]. Our model was organized in 4 blocks, each of them was formed by a convolutional layer with 64 filters of size 3 and ReLU activation function, a batch-normalization layer, and a max-pooling layer with both stride and size

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 3, 3, 512)	14714688
flatten_1 (Flatten)	(None, 4608)	0
batch_normalization (BatchNo	(None, 4608)	18432
dense_8 (Dense)	(None, 512)	2359808
dropout_6 (Dropout)	(None, 512)	0
dense_9 (Dense)	(None, 128)	65664
dropout_7 (Dropout)	(None, 128)	0
dense_10 (Dense)	(None, 20)	2580

FIGURE 4.18: Structure of TL model based on VGG16.

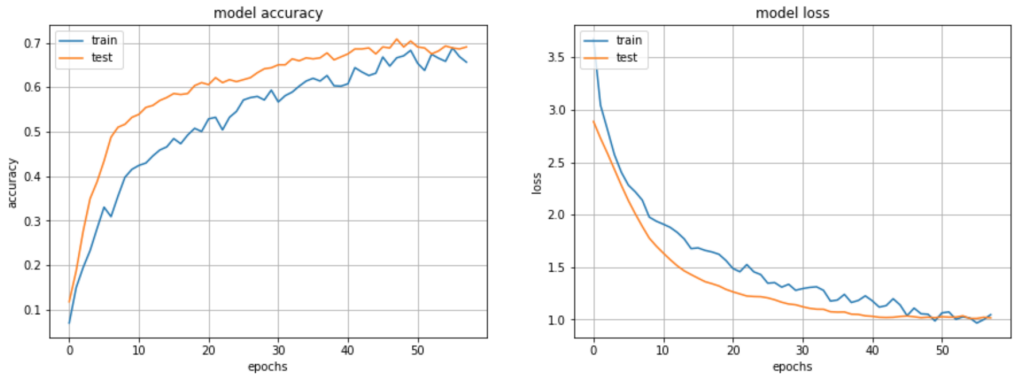


FIGURE 4.19: Accuracy and loss trend of TL model when we tuned only the dense part.

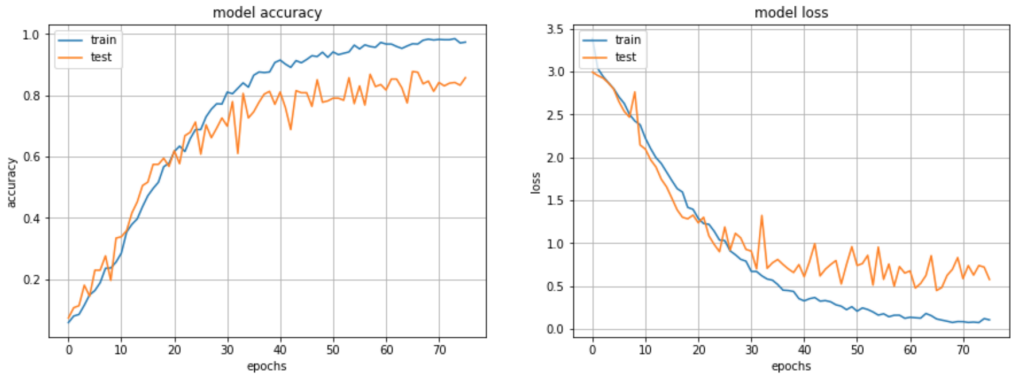


FIGURE 4.20: Accuracy and loss trend of TL model when we trained the whole net. We can observe the presence of overfitting.

equal to 2. After flattening the features, and before concatenating the vectors of the 2 images, we passed them through a dense layer with 32 units and ReLU activation function. Once we had the 2 separate features vector, we concatenated them, and we passed the resulting vector to an ANN formed in this way:

1. A dense layer with 16 units and ReLU activation function.
2. A batch-normalization layer.
3. A dense layer with 4 units and ReLU activation function.
4. A batch-normalization layer.

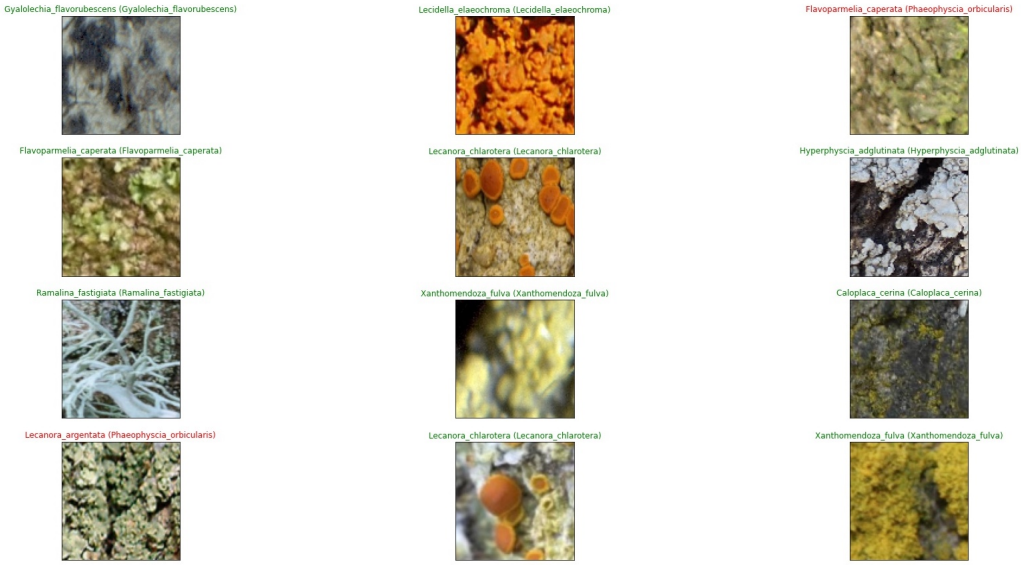


FIGURE 4.21: Application of the TL model on 12 unseen images. We can see that it performs quite well, with only 2 mistakes. Outside the brackets there is the predicted lichen, while inside the brackets there is the real species.

5. An output layer with 1 unit and Sigmoid function: this output is a single value, that represents the similarity between the 2 input images. In particular, it ranges from 0 (no similarity) to 1 (perfect similarity).

We used binary crossentropy as loss function and mean accuracy error (*mae*) to calculate quality of predictions. We adopted Adam as optimizer, with predefined parameters set by Keras implementation. Figure 4.22 represents our model. We trained model for 50 epochs using batch size of 32 couples, 16 with images from same class and 16 from different classes.

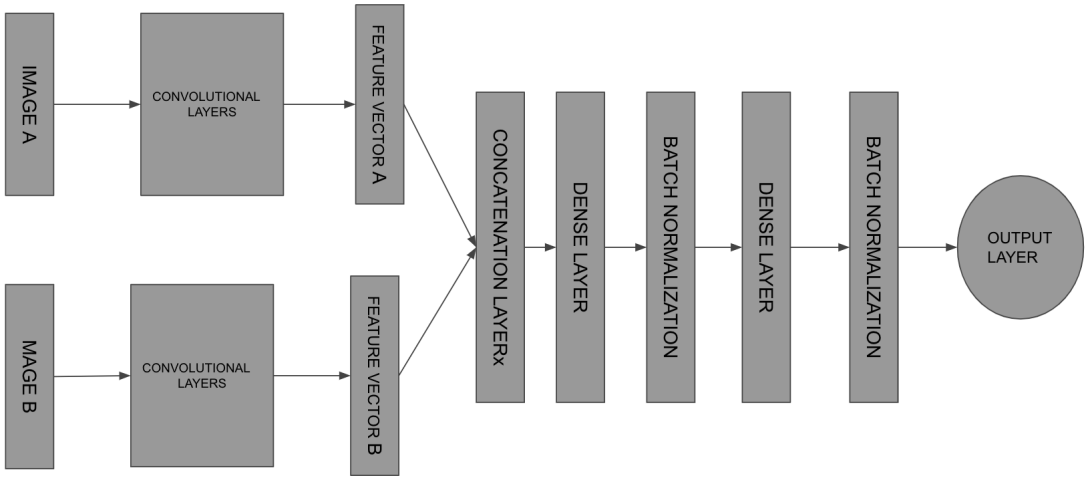


FIGURE 4.22: Structure of SNN: 2 images pass through convolutional layers with shared weights, which produce 2 feature vectors. The latter are then concatenated, and pass through a dense net, producing the similarity output.

Results on Image Similarity

We evaluated SNN on its original task: calculating similarity between images. Figure 4.23 shows how *mae* and loss vary with epochs: test loss starts early to oscillate, while test accuracy becomes worse than train accuracy after less than 10 epochs, even if it keeps decreasing with small oscillation. For what concerns *mae*, we achieved a value of around 0.25, meaning that average similarity is around 0.75 between images from same class, otherwise it is around 0.25. We used similarity score to predict if a test couple is composed by images belonging to the same class or not. Figure 4.24a represents how accuracy varies when we changed similarity threshold.[3] Best result is achieved with 0.1, while the worst case scenario is when the threshold is 0.9: in between the results fluctuate, but remain substantially constant. Figure 4.24b shows how error changes with respect to threshold, and especially how it is divided. We can notice that, apart when the threshold is 0.9, *false diversity* (when the model considers two identical images to be different) is always greater than *false equality* (when the model considers two different images to be identical), meaning that the SNN struggled to recognize images from the same class. This problem is due to the fact that there is high variability within the classes of our dataset. Figure 4.25 represents some couples analyzed by our net.

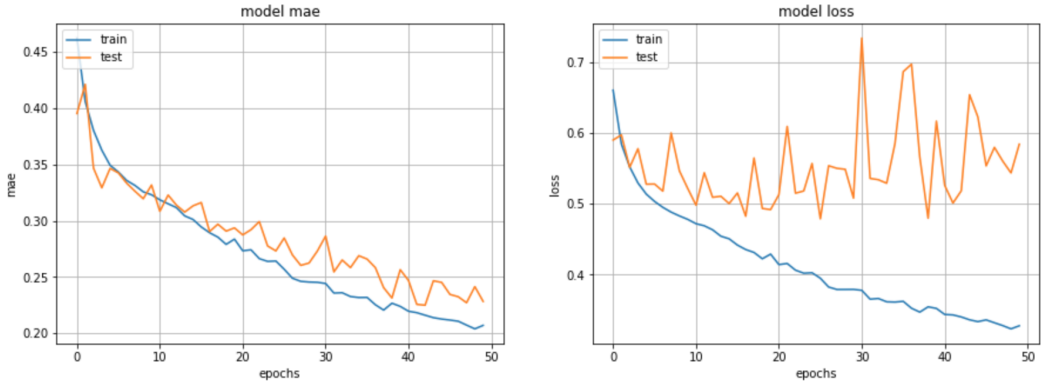
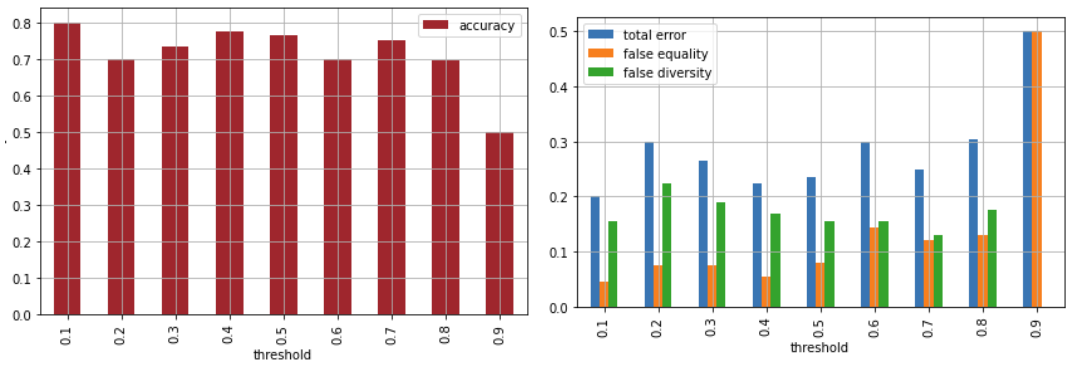


FIGURE 4.23: On the left: comparison between train and test *mae*. On the right: comparison between train and test loss.



(A) Variation of the performance.

(B) Variation of the error.

FIGURE 4.24: Performance of SNN model with respect to similarity threshold, both in terms of accuracy and error.

[3] Value that represents the edge between couple of images classified to belong to same class or not.

With a model capable of discerning whether two images belong to the same class or not, we could think to classify a test data by comparing it with available images, and associating the most similar species with it. To obtain acceptable results, our process should be very effective in terms of image comparison: Figure 4.26 shows that, unfortunately, this is not the case. This plot represents average similarity of test images, and it was calculated in the following way: for 2 different classes of lichens, we took all possible couples and we passed them to the net, obtaining similarities coefficient for all of them. Once we had all the results, we took the mean similarity. We can observe that, even if in the diagonal the values are quite high, for most of classes there is a lot of confusion: this makes difficult to use our algorithm for classification.

FIGURE 4.25: Validation couples used to test SNN. First three couples are formed by same class images, the rest couples are formed by different class images. Value predicted is the similarity score computed by the net.

Results of few shot learning

In this last section we focused on *few-shot classification* problem. In practice, we took 5 unseen species, and 10 images for each of them: according to Definition 4.2.1 we resolved a *5-way 10-shot classification problem*. As new species, we chose *Lecanora Caripinea*, *Candelaria Concolor*, *Melanelixia Subaurifera*, *Xanthoria Parietina*, and *Amandinea Punctata*: Figure 4.27 shows a sample for each of these new classes. For a specific test image $\mathbf{x}$, we calculated similarity scores with all support images, using the model trained in the previous section, and we took the average of each class: the greatest value represented the predicted species. We emphasize the fact that the SNN was not trained on this data, but we used model trained with the dataset described in Section 2.1. We utilized 50 images as support set (10 for each class), and 200 images as query set, uniformly distributed among the 5 classes. Figure 4.28 and 4.29 show an example of application of this process on 2 samples: we can observe that, even if the right class has a high similarity score, a clear separation with other species is not present. Figure 4.30a represents the similarity scores between different classes: we can see that, even if in the diagonal values are not irrelevant, classes are not well discriminated, as happened in previous section. This fact is reflected in the classification problem, where we achieved a mean accuracy of ~ 0.43. The main problem here is that there is a lot of difference in performance between different classes. In particular, for *Candelaria concolor* no images were classified correctly, how we can see

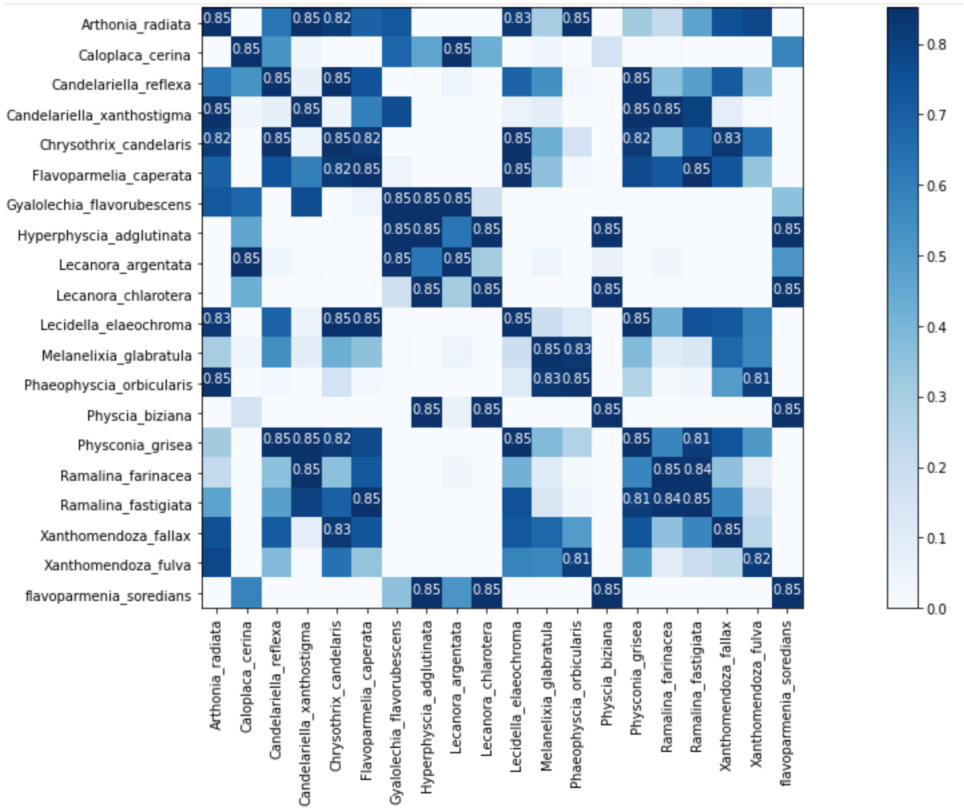


FIGURE 4.26: Similarity matrix: each value represents the mean similarity between two species.

in Figure 4.30b, and on the contrary *Xanthoria Parietina* turns to be an absorbing class, in the sense that a lot of images ended up in this category. To conclude, we experienced that, even with a simpler problem (we considered just 5 classes, instead of 20), we did not manage to build a model capable of discerning perfectly among different species, with few data available. In fact, despite our model was able to give a likely value of similarity between two images, it never found a clear separation between the classes, making the classification, as we have set it up, not very effective.

FIGURE 4.27: From left to right: *Amandinea Punctata*, *Candelaria Concolor*, *Lecanora Caripinea*, *Melanelixia Subaurifera*, and *Xanthoria Parietina*.

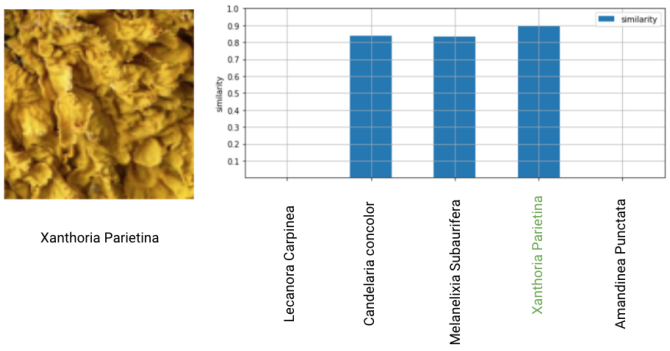


FIGURE 4.28: On the left: test sample classified correctly by the SNN base process. On the right: mean similarity score from each possible class. As we can see the right species has the highest score.

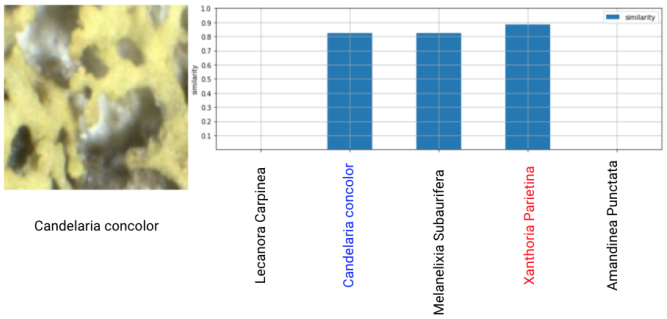
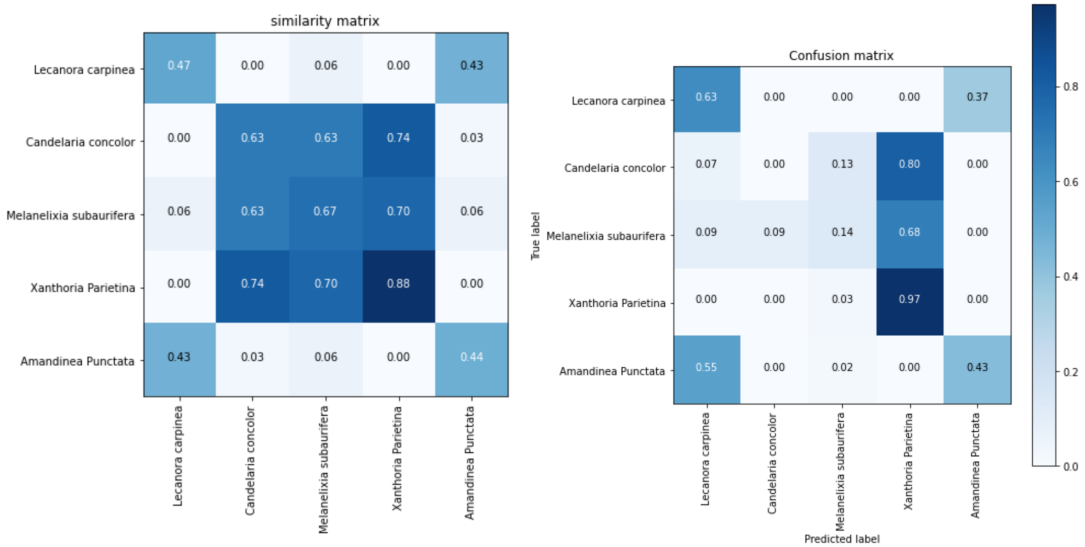


FIGURE 4.29: On the left: test sample. On the right: mean similarity score from each possible class. In this case the right species (blue) is different from the predicted species (red).



(A) similarity matrix.

(B) confusion matrix.

FIGURE 4.30: On the left: similarity matrix, where each value is the mean similarity between 2 species . On the right: confusion matrix, where each value represents prediction confusion between 2 classes.

Chapter 5

Scattering networks

One of the most critical points in image classification is to find a representation $\Phi(x)$, where x represent a specific image, with which to be able to measure similarity, using Eucledian distance. Section 4.3.3 indicated the considerable variability within the classes, that makes the search for such a representation complicated; this diversity is partially due to deformations like rigid translations, rotations, or scaling. Because of this, in this section our goal is to build a process that can handle them. In particular, we want to achieve both *invariance to translations* and *stability to deformations*, so a small deformation in the data would bring small change in distance between representations of images. We analyze a structure based on cascade of convolutions with localized waveforms (known as *wavelets*). The chapter is organized as follows: in Section 5.1, we give some preparatory definitions, in Section 5.2 we introduce theory of Scattering networks, and finally in Section 5.3 we explain our experiments based on this structure.

5.1 Stable representations

Let $\mathcal{I} \in L^2(\mathbb{R}^2)$ be the set of input images[1], and $\Phi \in L^2(\mathbb{R}^2)$ a transformation that, for $x \in \mathcal{I}$, outputs a representation Φx. From now on, the symbol $||x||$ represents the norm of an image x, defined as

$$||x|| = \int |x(u)|^2 du$$

First of all, we want Φ to be translation invariant, which means that it has to respect the following definition:

Definition 5.1.1 *Translation invariant representations*
$\forall x \in \mathcal{I}$ *and* $\forall c \in \mathbb{R}^2$ *a translated image w.r.t. c can be represented as:*

$$\forall u \in \mathbb{R}^2, \quad x_c(u) = x(u - c)$$

A transformation Φ *is said to be translation-invariant if*

$$\forall c \in \mathbb{R}^2, \Phi x_c = \Phi x$$

Moreover, to preserve stability to deformations, we want Φ to be non expansive:

$$\forall (f, g) \in L^2(\mathbb{R}^2)^2 \qquad ||\Phi(f) - \Phi(g)|| \leq ||f - h||$$

In order to reach this property, it is enough to have Lipschitz continuity relatively to the action of small diffeomorphism [16]. Let $\tau : \mathbb{R}^2 \to \mathbb{R}^2$ be a diffeomorphism, with

[1] $L^2(\mathbb{R}^2)$ is the space of measurable functions such that L^2-norm is finite.

$\tau \in \mathcal{C}^2$ and $|\nabla\tau(u)| < 1$. $\forall x \in \mathcal{I}$, we define

$$x_\tau(u) = x(u - \tau(u))$$

the deformed version of x, representing the action of $\mathbf{1} - \tau$ on x. Lipschitz stability means that $||\Phi x_\tau - \Phi x||$ is limited by the 'size' of τ, up to a multiplicative constant multiplied by $||x||$. This is synthetized by the following definition, taken from [16], that has to be valid:

Definition 5.1.2 *Lipschitz continuity to the action of $\mathcal{C}^2$ diffeomorphism*

A translation invariant operator Φ is said to be Lipschitz continuous to the action of $\mathcal{C}^2$ diffeomorphisms if there exists C such that $\forall x \in \mathcal{I}$ and $\forall \tau \in \mathcal{C}^2(\mathbb{R}^2)$

$$||\Phi x_\tau - \Phi x|| \leq C||x|| \left(\sup_u |\nabla\tau(c)| + \sup_u |H\tau(x)| \right)$$

where $\nabla\tau$ is the gradient, $H\tau$ is the Hessian matrix, and $||x||$ is the norm of x.

The reason why we want stability to both deformations is that often this variability is uninformative, and should be eradicated. However, with total invariance, performance would decrease irreparably: indeed, beyond a certain level of noise, any couple of images would become indistinguishable.

5.2 Scattering Networks

A *wavelet* ψ (small wave) is a localized waveform stable to deformation, with an amplitude that begins at zero, increases, and then decreases back to zero. A crucial property of wavelets is that they have zero mean:

$$\int \int_{\mathbb{R}^2} \psi(v) dv = 0 \tag{5.1}$$

An example of wavelets are the *complex Morlet Wavelets*, that are defined by the following equation:

$$\psi(u) = (e^{iu\cdot\xi} - K_1)e^{-|u|^2/(2\sigma^2)} \tag{5.2}$$

where K_1 is put to respect eq. (5.1). Figure 5.1a and Figure 5.1b represent real and imaginary part of Morlet wavelets, with $\sigma = 0.85$ and $\xi = \frac{3\pi}{4}$.

From eq. (5.2), we can obtain a family of rotated and scaled version of ψ, thanks to the following definition:

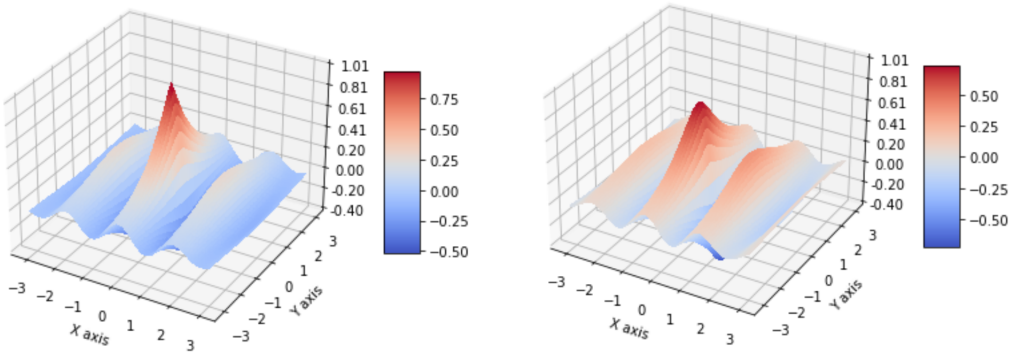
Definition 5.2.1 *Family of wavelets*

Let G be a rotations group in $\mathbb{R}^2$. $\forall\ \lambda = 2^j r$ where $j \in \mathbb{Z}$ (scaling factor) and $r \in G$, we define the family of rotated and scaled version of ψ

$$\psi_\lambda(u) = 2^{2j}\psi(2^j r^{-1} u)$$

We denote with $\mathcal{P} = \{\lambda = 2^{-j}r | r \in G, j \in \mathbb{Z}\}$ the set of parameters (scaling and rotation) that we use to create filter bank.

Figure 5.2 shows a family of Morlet wavelets, with 4 dilations factor and 8 rotation factors.



(A) Real part of Morlet Wavelet.

(B) Imaginary part of Morlet wavelet.

FIGURE 5.1: Representation of Complex Morlet wavelet, with $\sigma = 0.85$ and $\xi = \frac{3\pi}{4}$.

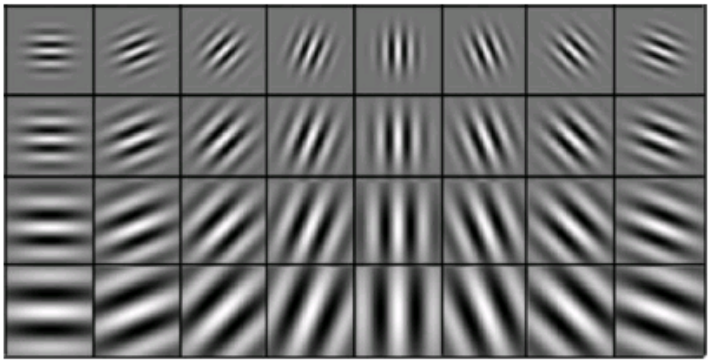


FIGURE 5.2: family of dilated and rotated Morlet wavelets, with 4 dilation factor J (across rows) and 8 rotation factor K (across columns).

5.2.1 Continuous wavelet transform

Once we have the wavelet family $(\psi_\lambda)_{\lambda \in \mathcal{P}}$, and given an image $x \in \mathcal{I}$, we can calculate a family of *wavelet transform* $\Phi_\lambda x$ in the following way

$$\forall u \in \mathbb{R}^2, \quad \Phi_\lambda x(u) = (x \star \psi_\lambda)(u) = \int \int_{\mathbb{R}^2} x(v) \psi_\lambda^*(u - v) dv \tag{5.3}$$

where $*$ represents the complex conjugate. For each point, the output of eq. (5.3) is influenced only by its neighborhood, as wavelets shrink quickly to zero, so wavelet transforms commute with translations, but are not translation invariant. However, we know that, if we have an operator R that commutes with translations $(R(L_c x) = L_c(Rx))$, then the integral $\int Rx(u)du$ is translation invariant. So, the idea is to take an operator M, apply it to the family of wavelet transform, and take the integral; because of eq. (5.1), we have to choose a non-linear operator, otherwise the integral would go to 0. In summary, we are looking for an operator $Q = M \circ \Phi_\lambda$ such that

1. $Q = M \circ \Phi_\lambda$ commutes with translations.
2. Integral of Q is stable to both deformation and additive noise.

One possible choice is the *modulus operator*, so we have:

$$\forall x \in \mathbb{R}, u \in \mathbb{R}^2 \quad Qx(u) = (M \circ \Phi_\lambda)x(u) = M(x \star \psi_\lambda)(u) = |x \star \psi_\lambda(u)| \tag{5.4}$$

the modulus is a point-wise operator, because $Mx(u)$ depends only on value of $x(u)$. Finally, we have that the resulting translation invariant coefficient is the L^1-norm:

$$||x \star \psi_\lambda||_1 = \int |x \star \psi_\lambda| du \tag{5.5}$$

This value is stable to deformations, besides being translation invariant. However, by integrating $\{|x \star \psi_\lambda|\}_\lambda$ we lose some information, as we remove all non-zero frequency components. This information can be recovered by calculating *wavelets coefficients of the wavelets coefficient* $\{|x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}(u)\}$: this notion represents the starting point for the construction of the *scattering convolutional networks*.

5.2.2 Scattering convolutional networks

We can now build a structure composed by a cascade of convolutions, followed by non-linear operators. The "first layer" is represented by the application of eq. (5.4) and eq. (5.5).

$$x \longrightarrow |x \star \psi_{\lambda_1}| \longrightarrow |||x \star \psi_{\lambda_1}||_1$$

We can proceed with calculations, by taking another parameter $\lambda_2 \in \mathcal{P}$, $\lambda_2 \neq \lambda_1$, and compute:

$$|||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}||_1 = \int ||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}| du \tag{5.6}$$

Recursively, we can consider how many different parameters we want. In this way, we build a structure called *scattering convolutional network*, or more simply *scattering network* (SN). In general, given a *path* of parameters $p = (\lambda_1, \lambda_2, \ldots, \lambda_n)$ we define the following operator

$$U[p]x = U[\lambda_m] \ldots U[\lambda_2] U[\lambda_1] x = |\, ||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}| \ldots | \star \psi_{\lambda_n}| \tag{5.7}$$

Note that for $n = 0$, we have $U[\emptyset]x = x$. Finally, for each p, we can give the following definition:

Definition 5.2.2 *Scattering coefficients*

$$Sx(p) = \mu_p^{-1} \int U[p]x(u) du \quad with \quad \mu_p = \int U[p]\delta(u) du$$

In practice, a SN performs better if the coefficients are localized at predefined scale: in this way they become invariant to small translations, but keep spatial variability at higher scales. Given $J > 0$, we have that 2^J represents both the maximum scaling factor of wavelet family and the scale at which the scattering coefficients lose their translational invariance property. Because of this, instead of the coefficients defined in definition 5.2.2, we prefer to use more localized coefficients, called *localized scattering coefficients* (LSCs), defined as follow:

Definition 5.2.3 *Localized scattering coefficients*
Given $J > 0$, we define a scaled spatial window

$$\phi_{2^J} = 2^{-2J} \phi(2^{-J} u)$$

With this, we define the localized scattering coefficients (LSCs) in the neighborhood of u as

$$S[p]x(u) = U[p]x \star \phi_{2^J}(u) = \int U[p]x(v) \phi_{2^J}(u - v) dv$$

and hence

$$S_J[p]x(u) = ||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}|...| \star \psi_{\lambda_n}|| \star \phi_{2^J}(u) \quad with \quad S_J[\emptyset]x = x \star \phi_{2^J}(u)$$

We set ϕ to be a Gaussian density.

In summary, setting $\mathcal{P}^m$ the collection of all paths of length $m \in \mathbb{N}$, a SN works as follow: it starts from the original signal x, and it outputs (at $m = 0$) the first LSC, that is equal to $S[\emptyset] = x \star \phi_{2^J}$. Then, at $m = 1$, for any $\lambda_i \in \mathcal{P}^i$, it first computes $|x \star \psi_{\lambda_1}|$ and subsequently it outputs LSCs $S[\lambda_i] = ||x \star \psi_{\lambda_1}| \star \phi_{2^J}|$. The algorithm proceeds until we reach the maximum level $\bar{m}$, how we can see in Algorithm 1, while Figure 5.3 gives a visual representation of a SN.

Algorithm 1 SN computation

```
procedure
    for m = 1 to m̄ do
        for all p ∈ P^{m-1} do
            Output S[p]x = U[p]x ⋆ φ_{2^J}
        for all p + λ_m ∈ P^m do
            U[p + λ_m]x = |U[p] ⋆ ψ_{λ_m}(2^{j_m})|
    for all p ∈ P^{m̄} do
        Output S[p]x = U[p]x ⋆ φ_{2^J}
```

Comparison with CNNs

As previously mentioned, there are some similarities with CNNs. In particular, we have that:

1. As CNNs, main operations are the convolution with filters and the application of a non-linear operator (the modulus).
2. This algorithm outputs LSCs, which can be seen as features.
3. It is a cascade-based structure, in the sense that for any path p and any relevant index $\lambda \in \mathcal{P}$, a new wavelet is computed and send to next layer.

However, there are some differences compared to CNNs, that can be synthesized whit the following points:

1. Output LSCs are produces at each layer, and not only after the last layer, as in CNNs. We can observe that in Figure 5.3.
2. Filters are fixed, since they are based on convolution with wavelets, so there is no training through backpropagation algorithm.

In summary, SNs can be seen as some kind of mixture between handcrafted and deepnet descriptors: on one hand we do not train any filters with data, but on the other hand we emulate the structure of a CNN.

Dimension of scattering Coefficient

In this section we just clarify the dimension of LSCs. Let N be the number of pixel of an image. The fact that ϕ_{2^J} is a low-pass filter scaled by 2^J implies that $S[p]x(u) =$

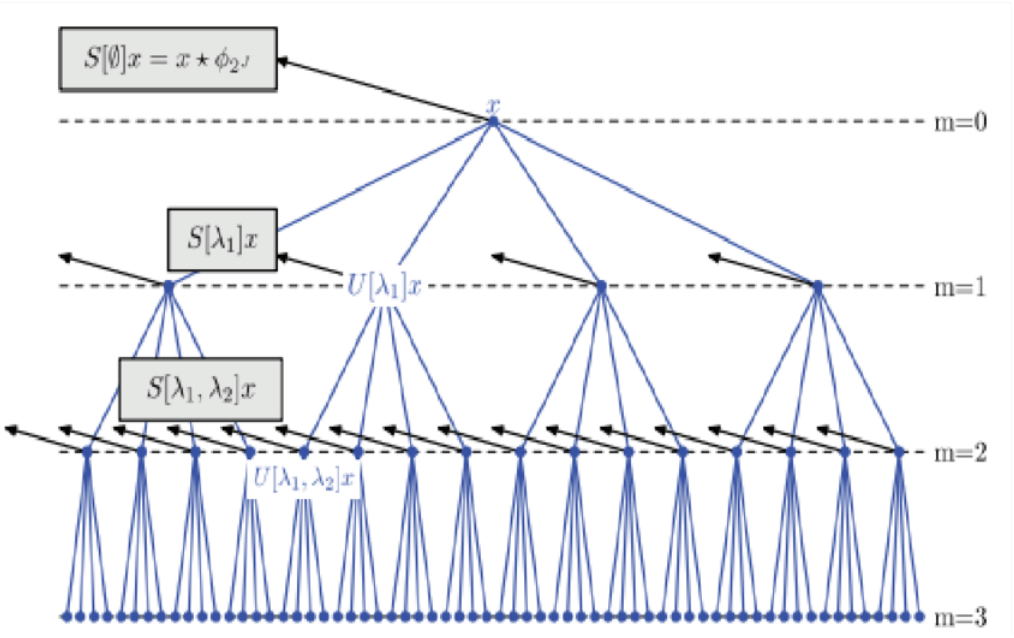


FIGURE 5.3: SN structure. We see that, at each level, the network outputs LSCs for every possible path.

$U[p]x \star \phi_{2^J}(u)$ is uniformly sampled at intervals 2^J. So, each scattering coefficient is an image itself[2], with an amount of $2^{-2J}N$ coefficients. In practice, we reshape LSCs, in order to have spatial dimension equal to $\left(\frac{N_x}{2^J}, \frac{N_y}{2^J}\right)$, where N_x and N_y are the horizontal and vertical dimension of the input image, respectively.

Scattering Properties

SNs exploit two important properties, that make them interesting; we limit ourselves to state them, with no proof, since it would be out of the scope of this project (more details can be found in [16] and [17]). If we consider the *scatter transform* $Sx = \{S[p]x\}_{p \in \mathcal{P}^\infty}$, with the signal energy equal to $||Sx||^2 = \sum_{p \in \mathcal{P}_\infty} ||S[p]x||^2$, where $\mathcal{P}_\infty$ stands for the set of all possible paths, the two properties are the following:

1. Non-expansiveness of the Scatter transform, meaning that:

$$||Sx - Sy|| \leq ||x - y||$$

   This property ensures that Sx is stable to deformations.

2. Decrease of $||S[p]x||$ as length of p increases. Because of this, we can approximate the signal energy $||Sx||^2$ using few levels, without loosing too much energy.

5.2.3 Wavelet scattering computation

One of the main problems of this structure is the time needed to compute all coefficients. The goal is to find a fast way to implement SNs, without losing too much

[2]Integration is computed component-wise.

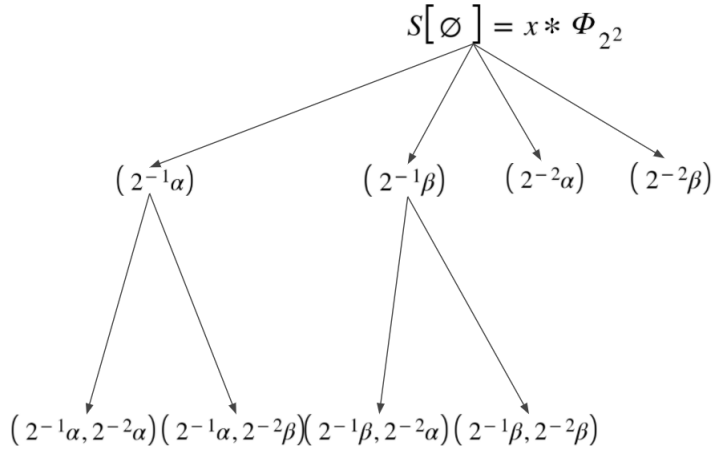


FIGURE 5.4: Outputs of the scattering net with $J = 2$, $m = 1$, and $K = 2$.

information. The idea is to calculate the LSCs only over a path of decreasing frequencies, represented below:

$$p = (2^{-j_1} r_1, ..., 2^{-j_n} r_n) \quad \text{s.t.} \quad 0 < j_k \leq j_{k+1} \leq J$$

It can be shown that on these coefficients is concentrated almost all of the scattering energy. We denote with $\mathcal{P}_{\downarrow}^{m}$ the set containing paths of decreasing frequencies of length m. In general, with K and J fixed, we have that $\#\left(\mathcal{P}_{\downarrow}^{m}\right) = K^m \binom{J}{m}$, where $\#$ represents the cardinality. Let's clarify this situation with an example. We set $m = 2$, $K = 2$ (we call the angles α and β), and $J = 2$, the network will output 9 scattering coefficients, given by:

1. $2^0\binom{2}{0} = 1$ for paths of length 0: $\{\varnothing\}$.
2. $2^1\binom{3}{1} = 4$ for paths of length 1:$\{(2^{-1}\alpha), (2^{-2}\alpha), (2^{-1}\beta), (2^{-2}\beta)\}$.
3. $2^2\binom{2}{2} = 4$ for paths of length 2: $\{(2^{-1}\alpha, 2^{-2}\alpha), (2^{-1}\alpha, 2^{-2}\beta), (2^{-1}\beta, 2^{-2}\alpha), (2^{-1}\beta, 2^{-2}\beta)\}$.

Figure 5.4 shows the structure of a SN with the example setting. Considering that each scattering coefficient has $2^{-2J}N$ coefficients (where N is the number of image's pixels), we have that the total number of LSCs in a SN is given by the following formula:

$$\#(LSCs) = 2^{-2J} N \sum_{m=0}^{\bar{m}} K^m \binom{J}{m} \tag{5.8}$$

Parameters of this process

Before starting with the experiments, we just resume what are the parameters that influence the characteristics of our network. In particular, we have:

1. J is the maximal scale of spatial variability, and size of spatial windows ϕ_{2^J}.
2. K: is the number of considered rotations in $G = \{r = \frac{k\pi}{K} | 0 \leq k < K\}$.
3. $\bar{m}$ is the maximal depth of the net.

5.3 Experiments and results

In this section we exploited LSCs as image descriptors, in order to obtain acceptable results for lichen's classification. For what concerns the dataset, we followed procedure described in Section 2.2, and furthermore we downsampled images to $100 \times 100 \times 3$, preserving the aspect ratio. As usual, we used LSCs in two ways:

1. As features to be fed into a SVM model.
2. As inputs to a CNN.

5.3.1 LSCs and SVM

Our first approach was to use LSCs simply as features, that then were fed in a SVM for classification. First of all, we dealt with LSCs dimension: from Section 5.2.3 we know that each image produces exactly $K^m\binom{J}{m}$ coefficients, each of them is an image itself of dimension $\left(\frac{N_x}{2^J}, \frac{N_y}{2^J}\right)$, where N_x and N_y are the horizontal and vertical shape of the input data, respectively. In order to work with colors, we simply considered each image's channel separately, and concatenated results: for example, if we take an input image of dimension $(100, 100, 3)$, and fix $J = 2$, $K = 4$, and $\bar{m} = 2$, we obtained features of dimension $(75, 25, 25)$ (see eq. (5.8)), where the first value refers to the number of LSCs, while the second and the third are the spatial dimensions. In order to flatten LSCs, we followed two ways:

1. For each LSCs, we calculated the sum of all elements.
2. For each LSCs, we calculated the mean of all elements.

In both cases we obtained a final vector of dimension $3 \times \sum_{i=0}^{\bar{m}} K^i\binom{J}{i}$. Finally, we trained a SVM model for classification, trying different kernels, in order to find the best one. The entire process is represented in Figure 5.5. We fixed $\bar{m} = 2$, while J ranges from 2 to 5, and K from 1 (no rotations) to 8. Table 5.1 and Table 5.2 display accuracy score for all possible configurations, using sum and mean approach, respectively: we can see how these two approaches bring quite similar results, even if there is a slight improvement with the mean operator. The variable K, instead, plays a more important role: indeed, we can notice that, when $K = 1$ (no orientation), accuracy drops dramatically, testifying to the fact that, in our process, we have to consider more orientations. Figure 5.7 displays the confusion matrix, while Figure 5.6 represents how accuracy, recall, and F1-score vary withing classes: both images refer to the particular case where $K = 5$, $J = 4$, and with mean operator (best case scenario). These images testifies that, even if the overall performance achieved an accuracy of ~ 0.77, there is a lot of variability between different lichens: for example, *Arthonia Radiata* and *Lecanora Argentata* had an accuracy of 0.95 and 0.50, respectively. Moreover, we performed some experiments changing the kernel of the SVM. We fixed $J = 4$, $K = 5$, $\bar{m} = 2$, with the mean operator. Table 5.3 displays performance variation (these values represent an average between the different species) with respect to the kernel of SVM: We can notice that, even if we reach quite similar results, the best kernel remains the Gaussian one.

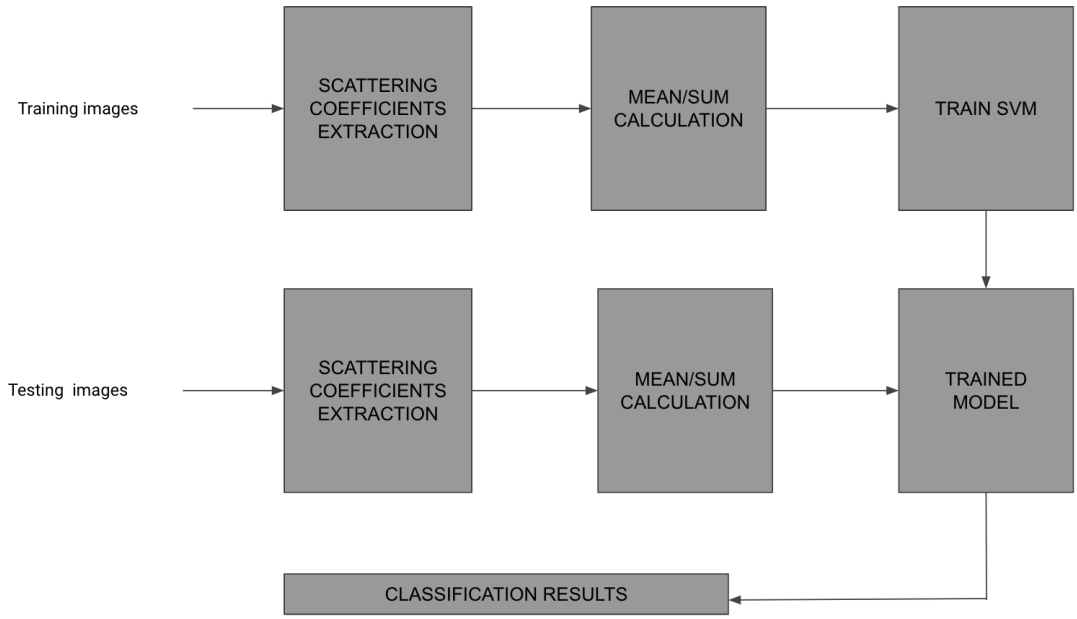


FIGURE 5.5: Classification process using LSCs and SVM. This process includes both the mean and sum operator.

K \ J	2	3	4	5
1	0.5812	0.6525	0.6770	0.6703
2	0.6570	0.7394	0.7461	0.7371
3	0.6770	0.7527	0.7550	0.7282
4	0.6726	0.7438	0.7616	0.7371
5	0.6726	0.7438	**0.7661**	0.7416
6	0.6859	0.7349	0.7594	0.7394
7	0.6837	0.7260	0.7327	0.7438
8	0.6770	0.7216	0.7550	0.7287

TABLE 5.1: Accuracy results of the model based on LSCs, changing orientations K (across rows) and scale value J (across columns), using sum operator.

K \ J	2	3	4	5
1	0.4988	0.6503	0.6726	0.6837
2	0.5768	0.7216	0.7260	0.7416
3	0.6124	0.7193	0.7572	0.7416
4	0.6347	0.7193	0.7616	0.7483
5	0.6416	0.7171	**0.7706**	0.7527
6	0.6481	0.7260	0.7639	0.7461
7	0.6458	0.7238	0.7594	0.7371
8	0.6547	0.7282	0.7522	0.7438

TABLE 5.2: Accuracy results of the model based on LSCs, changing orientations K (across rows) and scale value (across columns), using mean operator.

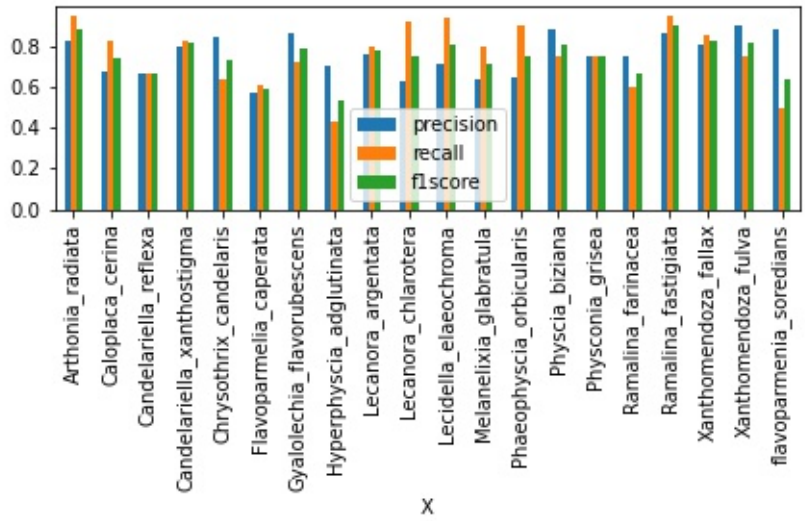


FIGURE 5.6: Variation of precision, recall and F1-score for different lichens species in the specific case of $J = 4$, $K = 5$ and sum operator. We notice that some species are well recognized by the model (*Ramalina fastigiata*), while some other classes are not (*Flavoparmelia caperata*).

kernel	accuracy	precision	recall
linear	0.7216	0.7461	0.7369
poly of degree 2	0.7555	0.7760	0.7674
poly of degree 3	0.7210	0.7518	0.7342
poly of degree 4	0.7203	0.7181	0.7328
poly of degree 5	0.7104	0.7380	0.7220
poly of degree 6	0.6926	0.7294	0.7057
Gaussian	**0.7706**	**0.7867**	**0.7737**

TABLE 5.3: Table which shows results of SVM on model based on LSCs.

Classification with few training data

Until now, we have always dedicated 70% of our dataset to the training phase, and the remaining 30% to test our process: in this section we instead used only a little fraction of data (20%) to train the SVM model, in order to check how SNs are able to generalize well with few images. As in the previous section, we set $J = 4$, $K = 5$, $\bar{m} = 2$, and the mean operator to create feature vectors. The results obtained are quite positive: indeed, we managed to achieve an accuracy of ~ 0.6, and this expresses how the SNs do not need large amounts of data. In Figure 5.9 and Figure 5.8 we can observe confusion matrix of the model trained with 20% of the dataset, and how precision, recall, and F1-score vary for different lichens, respectively: from these two plots we can observe that, even if overall accuracy is acceptable, there are a lot of variability between different classes. Figure 5.10 displays how overall performance changes when we decrease size of the training set: we notice that even if we used only the 15% of the dataset to train the model, we achieved above 0.50 of accuracy for classification task. In summary, we experienced that a large amount of data is not needed to get good results with SNs; despite this, even with a greater amount of images, the situation does not improve too much.

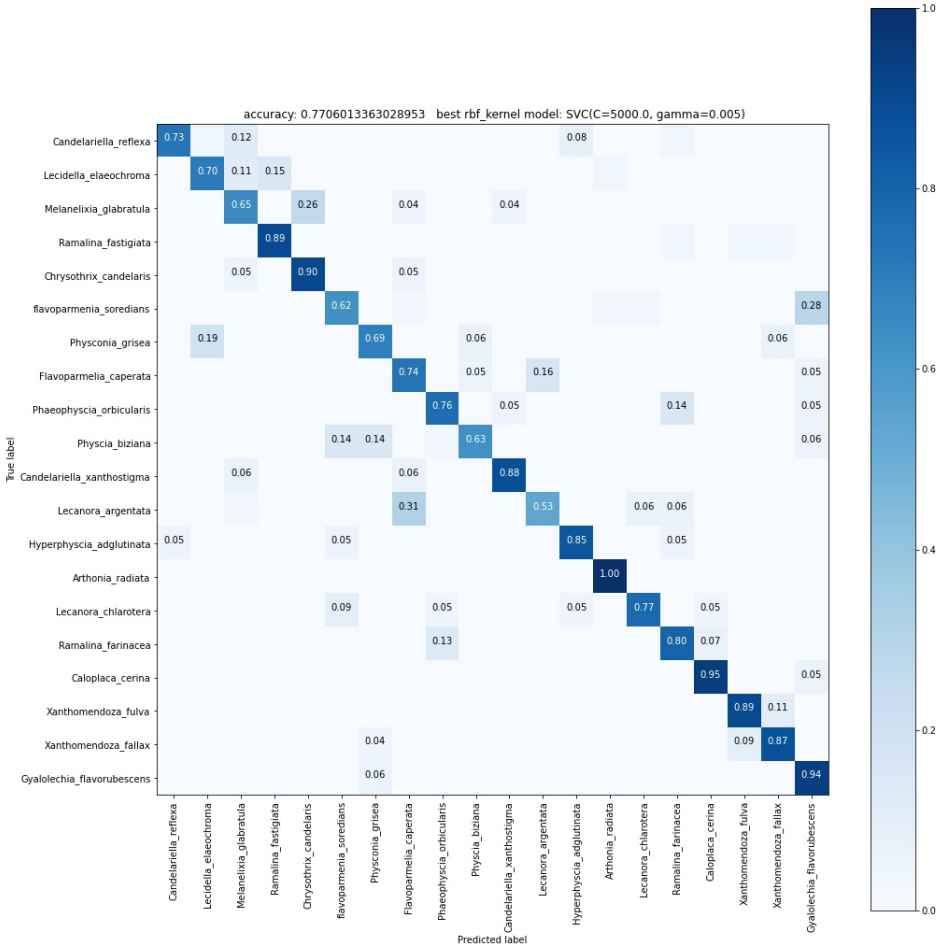


FIGURE 5.7: Confusion matrix in the case of $J = 4$, $K = 5$ and mean operator. We trained a SVM model, equipped with Gaussian kernel.

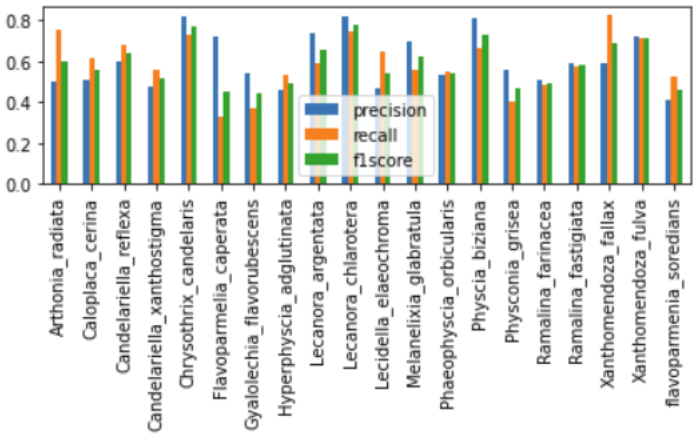


FIGURE 5.8: Variation of precision, recall, and F1-score with 20% of training data, using the LSCs based model. We set $J = 4$, $K = 5$, $\bar{m} = 2$, and the mean operator.

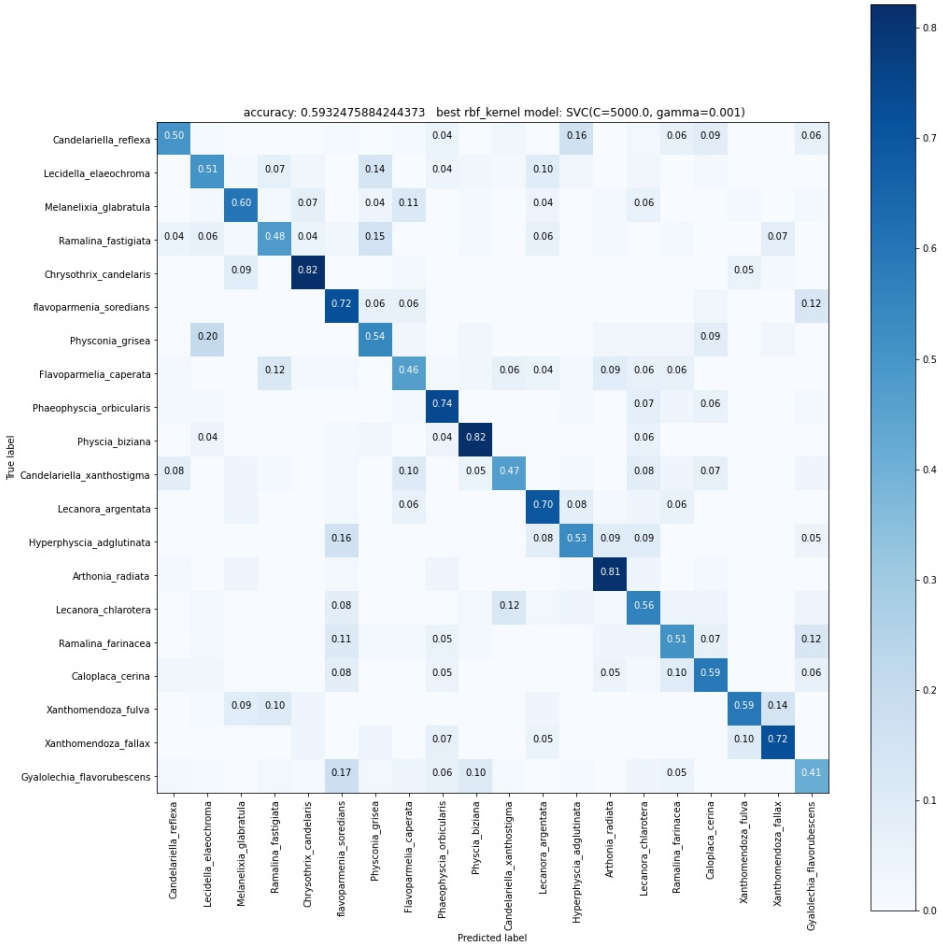


FIGURE 5.9: Confusion matrix obtained with 20% of training data, using the LSCs based model. We set $J = 4$, $K = 5$, $\bar{m} = 2$, and the mean operator.

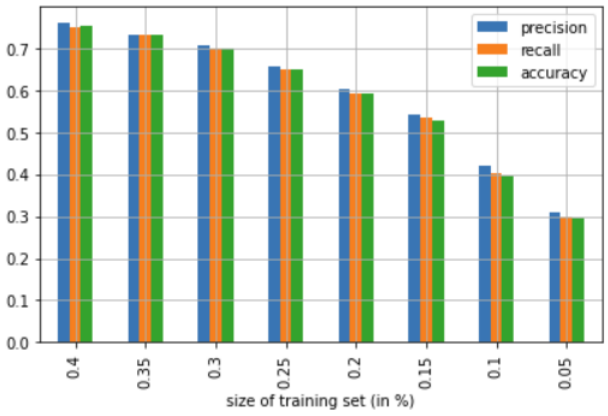


FIGURE 5.10: Variation of precision, recall, and accuracy w.r.t size of training set. We set $J = 4$, $K = 5$, $\bar{m} = 2$, and the mean operator.

5.3.2 LSCs and CNNs

In this last section we merged together SNs and CNNs. The idea here is to use LSCs as inputs of a CNN that, after some convolutional layers, performs classification. In

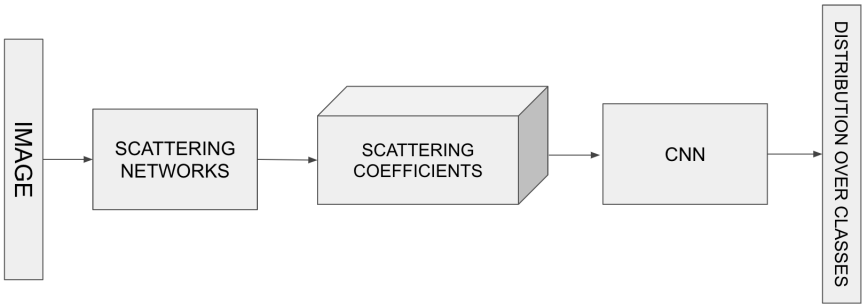


FIGURE 5.11: Structure that joins SNs and CNNs.

practice, we obtained a deep CNN, the first part of that has fixed weights (consisting of LSCs), while the second part is a trainable CNN. General structure is represented in Figure 5.11. For what concerns the SN, we fixed $J = 2$, $K = 4$, and $\bar{m} = 2$: in this way we obtained LSCs of dimension $(75, 25, 25)$. Instead of flattening these features, we treated them as inputs of a CNN, built in the following way:

1. A first block formed by a convolutional layer with 256 filters of size 3, followed by a batch-normalization layer and ReLU activation function.
2. A max-pooling layer with both stride and size equal to 2.
3. 2 blocks, each of them formed by a convolutional layer with 512 filters of size 3, followed by a batch-normalization layer and ReLU activation function.
4. An adaptive pooling layer (explained in section 4.1.2), that reduces dimension of filters to $512 \times 2 \times 2$.
5. A flattening layer.
6. A final Output layer with Log-Softmax activation function.

As optimizer, we adopted Stochastic gradient descent, with momentum equal to 0.9, and initial learning rate equal to 0.01 (it decreases by a factor of 0.2 every 20 epochs). We used cross-entropy as loss function, and we trained for a maximum of 100 epochs, using early stopping to fight overfitting. As we did in Chapter 4, we applied data augmentation, exploiting the same techniques described in Section 4.2.1. Figure 5.12 shows how loss and accuracy vary during epochs: we can observe how the presence of overfitting is minimal, since, for what concerns accuracy, the difference between the train and test data is not that huge. In terms of test accuracy, we obtained a results of ~ 0.82; moreover, we can observe in the confusion matrix, represented in Figure 5.14, that almost all different classes are well discriminated, apart from some exception. For example, there is a lot of confusion with *Gyalolechia Flavorubescens*, that is predicted only the 33% in the right way; this species has a very high percentage of *false negative*, and this turns to a very low *recall* value, as we can see in Figure 5.13. As we did in Section 5.3.1, we tried to change values of both K (from 1 to 8) and J (from 2 to 4), while $\bar{m} = 2$ remained fixed. We did not change the main structure, but we just adapted the input size of the CNN, so that it accepts inputs (LSCs) of the right size. Accuracy results are exhibited in Table 5.4: we can see that they are similar to each other, with some small fluctuation, also due to the fact we only changed the input size of the first convolutional layer, while the remaining part of the structure remained unaltered. Best results are achieved when $K = 5$

and $J = 2$, with an average accuracy of ~ 0.83, even if we emphasize the fact that these are average evaluations, therefore subject to possible stochastic variations. To conclude, we have created a structure capable of achieving results similar to those obtained in the previous chapters, by exploiting the characteristics of SNs.

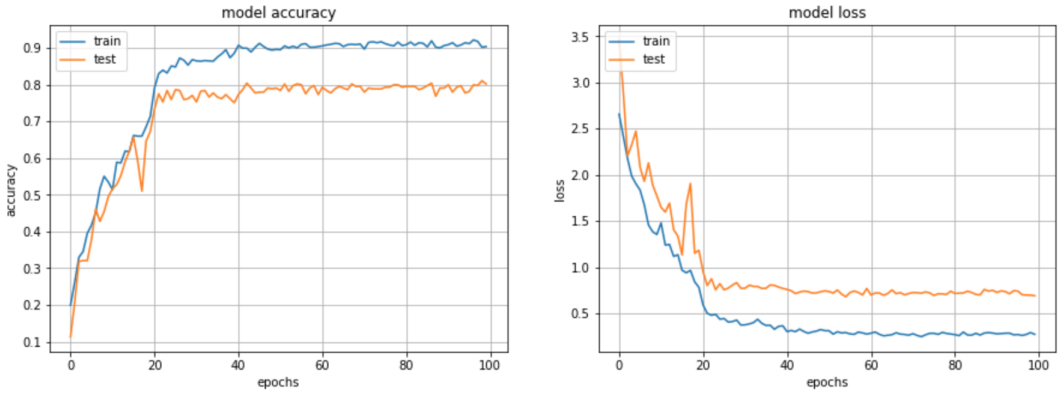


FIGURE 5.12: On the left: variation of test and train accuracy during epochs. On the right: variation of test and train loss during epochs.

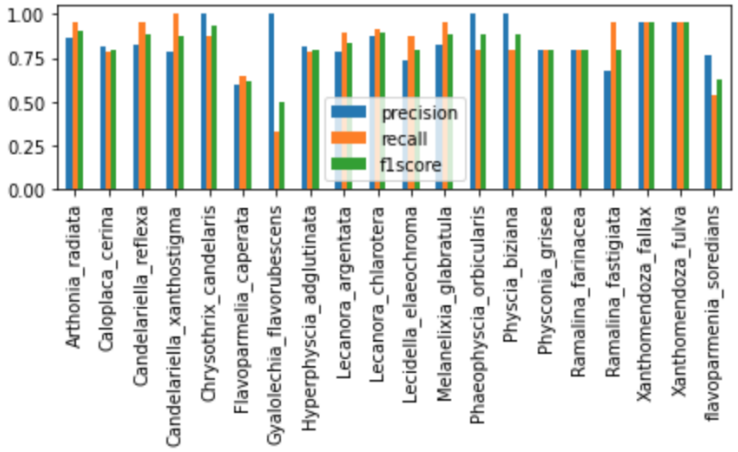


FIGURE 5.13: Variation of precision, recall and accuracy of different species: the model is the one described in section 5.3.2.

K \ J	1	2	3	4
1	0.7766	0.736	0.74	0.6726
2	0.8028	0.8084	0.8330	0.8329
3	0.8140	0.8151	0.8173	0.8218
4	0.8140	0.8260	0.8351	0.8040
5	0.8107	**0.8351**	0.8173	0.8062
6	0.8010	0.8215	0.8195	0.7839
7	0.8190	0.8218	0.8106	0.7772
8	0.8218	0.8151	0.7839	0.7928

TABLE 5.4: Average accuracy of network described in 5.3.2, when we change both J and K.

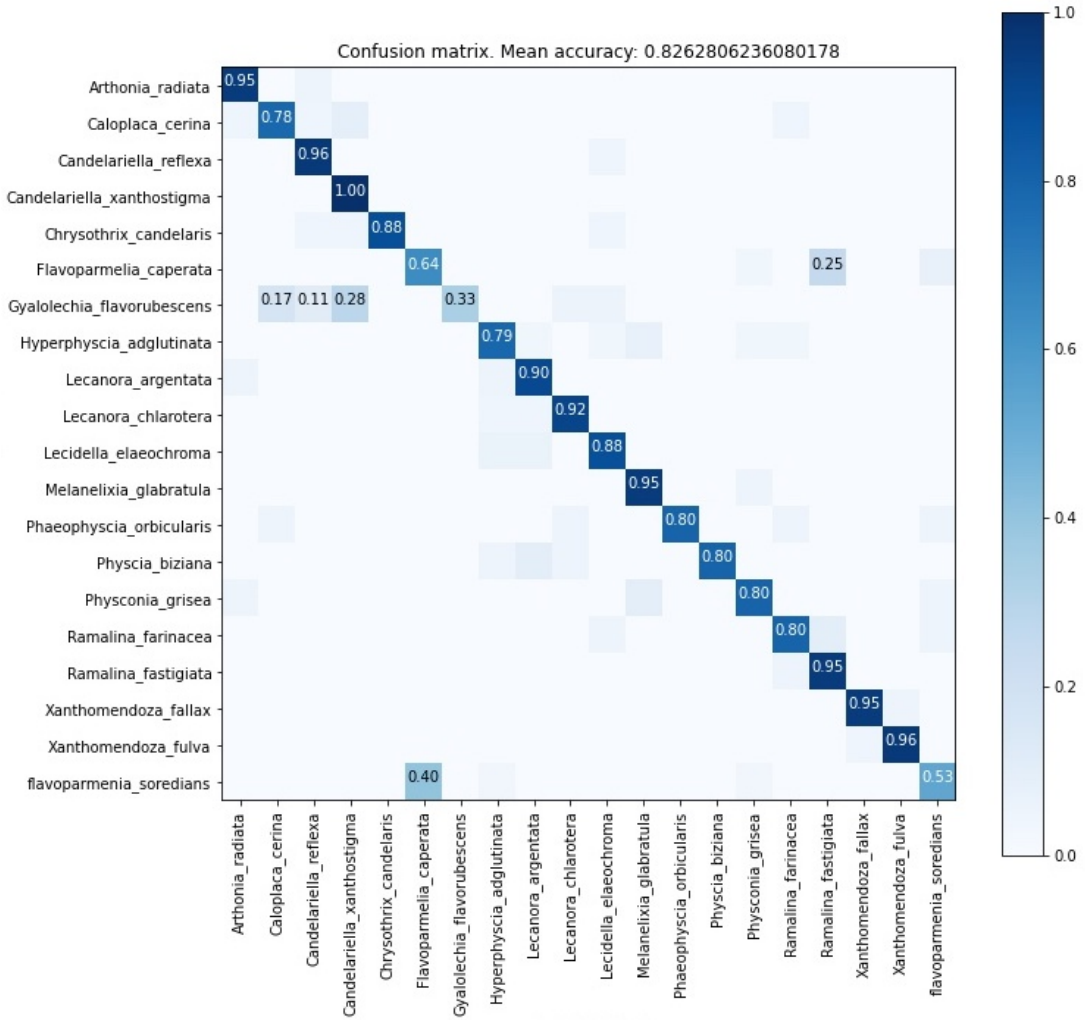


FIGURE 5.14: Confusion matrix on the test set of the model described in section 5.3.2.

Chapter 6

CONCLUSION

In this work, we studied the problem of classification of lichen's images. In particular, we organized our project as *patch classification*, meaning that we divided an image in non-overlapping cells, and we classified each of them individually. We developed different possible solutions, rather than staying anchored to just one, in order to find what is the most suitable process for the available dataset.

In Chapter 3, we exploited several descriptors, in order to recognize discriminating patterns. We used them as training data for a classification algorithm, trying to find both the best parameters and structure. Even if these methods could seem antiquated, there represented a viable option for our project, as our biggest problem was lack of data: indeed, in terms of accuracy, they brought the best results.

In Chapter 4, we took advantage from the power of *convolutional neural networks*. We both implemented model from scratch and exploited pre-trained nets with transfer learning, concluding that the latter is the best solution for our problem. In the end of this chapter, we also exploited *siamese neural network*, a structure that compares two different images, discerning if they belong to the same class or not. This above model could be a good way for possible future works, because it would make possible to compare new lichens species not seen before; however, in the way we set the model, we did not get the desired results.

In Chapter 5, we explored the *scattering networks*, that can be seen as a mixture of the two previous approaches: even if they are based on convolution with fixed wavelets, they reminded structure of convolutional neural networks. Scattering networks allow to build a signal representation of an image, that is invariant to translation and stable to deformation. We used them in two ways: In the first one, we exploited them simply as input of a support vector machine, while in the second one we modeled a huge structure formed by the concatenation of a scattering network and a convolutional neural network. Results are quite promising, even if they did not outperform the ones obtained in Chapter 3.

Compared to [2], we obtained better results in terms of accuracy, probably because we approached to the problem in a different way, classifying singular patches rather than the entire image, making the problem easier to solve. In conclusion, our experiments lead us to say that for now, with the current available dataset, best approach in terms of effectiveness is still the one based on dense handcrafted descriptors, as they succumb better to the lack of data. We must say that we resolved a simplified version of the original problem, as we ignored glitches like *background detection* or *lichens precise segmentation*: however, we are confident that this work can be a good starting point for other projects, that can improve performance and make the process suitable for real application.

Bibliography

[1] Gioia Bini, Massimo Bonannini, and Roberta Ferrarese. "I.B.L Indice di Biodiversità lichenica." In: *Manuale ANPA, Dipartimento di Stato dell'Ambiente, controlli e sistemi informativi.* (2001).

[2] Deep convolutional neural network for preliminary in-field classification of lichen species. "Agnieszka Galanty and Tomasz Danel and Michal Wegrzyn and Irma Podolak and Igor Podolak." In: *Elsevier Ltd* (2021).

[3] Cislaghi C. and Nimis P. "Air pollution and lung cancer". In: *Nature 387, 463-464* (1997).

[4] Taha H. Rassen and Bee Ee Khoo. "Object Class Recognition using Combination of Color SIFT Descriptors". In: *IEEE International Conference on Imaging Systems and Techniques* (2011).

[5] Timo Ojala, Matti Pietikäinen, and Topi Mäenpää. "Multiresolution Gray Scale and Rotation Invariant Texture Classification with Local Binary Patterns". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2002).

[6] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision 60, 91–110* (2004).

[7] Sultan Aljahdali and Aasift Ansari. "Classification of Image Database using SVM with Gabor Magnitude". In: *International Conference on Multimedia Computing and Systems, Tangiers, Morocco.* (2012).

[8] Lin Zhangm Zhiqiang Zhou and Hogyu Li. "Binary Gabor Pattern: an efficient and robust descriptor for texture classification". In: *2012 19th IEEE International Conference on Image Processing, Orlando, FL, USA.* (2012).

[9] Tony Lindeberg. "Scale-space". In: *School of Computer Science and Communication, Royal Institute of technology, SE-100 44 Stockholm, Sweden* (2008).

[10] Moria Tau and Tal Hassner. "Dense Correspondences across Scene and Scales." In: *IEEE Trans. on Pattern Analysis and Machine Intelligence* X (2016).

[11] Gareth James, Daniela Witten, and Trevor Tibshirani. "An introduction to Statistical learning". In: *Springer Texts in Statistics* (2013).

[12] Andrea Vedaldi and Brian Fulkerson. "VLFEAT - An open and portable library of computer vision algorithms". In: (2010).

[13] Karen Simonyan and Andrew Zisserman. "Very large convolutional neural networks for large-scale Image recognition". In: *CoRR* (2015).

[14] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. "Siamese Neural Networks for One-shot Image recognition". In: (2015).

[15] Jake Snell, Kevin Swerky, and Richard Zemel. "Prototypical Networks for Few-shot Learning". In: *Advances in Neural Information Processing Systems* (2017).

[16] Stéphane Mallat. "Group Invariant Scattering". In: *Pure and applied mathematics, volume 65* (2012).

[17] Joan Bruna and Stéphane Mallat. "Invariant Scattering Convolution Network". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2012).

[18] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep learning". In: *Journal of big Data 6,60* (2019).

[19] Paolo Napoletano. "Hand-Crafted vs Learned Descriptors for Color Texture Classification". In: *dipartimento di Informatica, Sistematica e Comunicazione, Università degli studi di Milano-Bicocca, Italy* (2017).

[20] Laurent Sifre et al. "ScatNet: a MATLAB Toolbox for Scattering Networks". In: *CMAP, Ecole Polytechnique, Palaiseau, France* (2012).