Université de Liège
Faculté des sciences appliquées

Generic image classification : random and convolutional approaches

Master thesis

Thesis directors
Pierre Geurts
Raphaël Marée

Author
Jean-Michel Begon

Année académique 2013-2014

Contents

Chapter 1

Introduction

Chapter 2

State of the art

The early days of computer vision have seen the development of myriads of domain specific methods, notably in the field of image classification. Image classification is the process of assigning the correct label from a predefined set to an image. The main drawback of the domain specificity is that solutions are not necessarily transposable to other domains. The supervised machine learning framework sidestep this limitation by making no assumptions about the particular domain of application. Rather, it proposes a representation general enough for most problem to fit in, albeit with some preprocessing. Overcoming the domain specificity is achieved by letting the computer *learn* the discrimination scheme instead of supplying it.

This chapter is divided in two sections. In the first one, we review the supervised learning framework in general. In the second part, we focus on supervised learning image classification.

2.1 Supervised learning.

At the framework core are objects, also called individuals. The objects differ from each other by their features, also called variables. Among them is one which holds a special status : the output variable. The ultimate goal of a supervised learning algorithm is to produce a model mapping a previously unseen object's regular features to the output variable. Thus, the learning algorithm needs a space of candidate models. In order to provide the most adequate model out of the candidates, the algorithm needs two additional elements : a quality measure and an optimization strategy. It is sometimes more convenient to use the inverse of the quality measure : the loss function. We talk about *learning* because the algorithm optimizes the quality measure using a set of objects, called the learning set.

More formally, the learning set $LS = \{(\boldsymbol{x}_i, y_i) | i = 1, ..., N\}$ is composed of N objects. Each object i is described by a tuple of features : $\boldsymbol{x}_i \in \mathcal{X}$ are the regular features and $y_i \in \mathcal{Y}$ is the output variable. We represent the learning algorithm hypothesis space by $\mathcal{H}$ and its quality measure by $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. The learning algorithm maps the learning set to a function $f \in \mathcal{H} : \mathcal{X} \to \mathcal{Y}$ so as to try to minimize some expectation over the loss function. Depending on the structure of $\mathcal{Y}$, the model either performs a regression or a classification.

The most usual quality measures are the square error $\ell(y, \hat{y}) = (y - \hat{y})^2$ for regression problems and the classification error

$$\mathbf{1}_{\neq}(y, \hat{y}) = \begin{cases} 1 \text{ if } y \neq \hat{y} \\ 0 \text{ otherwise} \end{cases}$$

for classification problems. Based on those loss functions, we can define the expected error by $E_{\boldsymbol{x},y}\{\ell(f(\boldsymbol{x}), y)\}$. If we dispose of a testing set $TS = \{(\boldsymbol{x}_i, y_i) | i = 1, ..., M\}$ with a sufficiently large M, we can assess the expected error by $\frac{1}{M} \sum_{\boldsymbol{x},y \in TS} \ell(f(\boldsymbol{x}), y)$. If $TS = LS$, we call this

the resubstitution error. However, we are generally interested in the case where both sets are different while coming from the same distribution. We then talk about generalization error. This is the error we are trying to minimize.

The example of decision trees.

Decision tree (Breiman et al. (1984)) is a good example of classification algorithm. More accurately, the decision tree is the model produced and the learning algorithm is the growing algorithm. The decision tree is a binary tree where each interior node represent a dichotomic choice regarding one regular feature and each leaf is labeled by a class. Classifying an object consist of moving from one node to another according to the node's test. Once the image reaches a leaf, its class is associated to the image.

Growing a good tree, in the sense of the quality measure, is done in a top-down fashion by a greedy heuristic. We first need to define an impurity measure which indicates how much diversity for the variable y there is in a sample. We start at the root and pick up the dichotomic choice, also called splitting criterion, which accomplishes the maximum expected reduction of impurity. Then the learning sample is split into a left and a right branch according to the test result. We can now develop both nodes recursively up till there is only objects of the same class in a node, making it a leaf. With this mechanism, the resubstition error is null, thus proving that the method minimizes the loss function. However, the generalization can be quite large. That is why other mechanism, such as other stopping criteria, are implemented.

A widely-used impurity measure is Shannon entropy. In that case, the construction algorithm has a nice interpretation. The reduction of impurity is the information gain. At each node, we choose the test which brings the most information to the classification variable.

The learning algorithm dependence on the learning set is clearer in the regression case, where we have the following bias-variance decomposition :

$$\begin{aligned} E_{LS}\{E_{\boldsymbol{x},y}\{(y - f(\boldsymbol{x})^2)\}\} = \; & E_{\boldsymbol{x},y}\{(y - E_{y|\boldsymbol{x}}\{y\})^2\} \\ & + E_{\boldsymbol{x}}\{(E_{y|\boldsymbol{x}}\{y\} - E_{LS}\{f(\boldsymbol{x})\})^2\} \\ & + E_{\boldsymbol{x}}\{(f(\boldsymbol{x}) - E_{LS}\{f(\boldsymbol{x})\})^2\} \end{aligned}$$

The first term, characterize the intrinsic difficulty of the regression task. It quantifies the average deviation of the best possible model $E_{y|\boldsymbol{x}}$, the Bayes model, from the ground truth. The second term is the expected square bias. It quantifies the average error between the Bayes model and the average model, $E_{LS}\{f(\boldsymbol{x})\}$. The third term is the average variance of the model. It quantifies how much the model varies from one learning set to another. Although there is no such analytical decomposition for classification, the concepts of Bayes model, bias and model variance still abide as conceptual tools.

The space of candidate model $\mathcal{H}$ is composed of elements of varying complexity. For instance, a decision tree complexity is assessed by its number of nodes. A more complex model will fit the learning set better, thus reducing the resubstitution error. However, fitting this set too well will cause overfitting : the algorithm incorporate set-specific details. This is reflected by the model variance term of the learning algorithm error : slight changes in the learning set will produce very different models. The bias decrease will, at first, overcompensate that increase, yielding a better generalization error. The compensation will work up to the point where the average is of the same order of complexity as the Bayes model. From that point on, the bias decrease will slow down while the model variability still increases. Therefore, the generalization error starts to increase, as well. The ability to control the complexity is an important characteristic of a learning algorithm. Another way around this problem is to resort to ensemble methods. Among such methods are the averaging methods which combine several models by averaging their predictions. The models

are usually drawn from the same candidate space by either using different learning set or introducing some randomization in the learning algorithm, sometimes both. Ensemble methods rely on the averaging to reduce variability and thus can enjoy more complex models. Ensemble of decision trees form classification forests.

A learning algorithm is usually dependent of some parameters, which influence the optimization strategy. They are called hyper-parameters so as not to confuse them with the other parameters on which the optimization is performed.

2.2 Image classification

Traditional learning algorithms are not able to works with "structured data" such as texts, images and graphs. Indeed, they expect the individual features to be scalar. Image classification is therefore much more about bridging the two realities than about developing brand new learning algorithm. We will now review the main such techniques.

Chapter 3

Objectives

The hypothesis at the core of the present master thesis can be stated as follows :

> It is possible to combine the advantages of the classification forests, namely computational cost, feature importance evaluation and ease of use, with those of convolutional networks, primarily the accuracy.

The feature importance evaluation capability is one of the nicest features of the classification forests. The importance of a given feature is computed as the total reduction of impurity brought by that feature, normalize so that the feature importances sum to one. The most notable use of this measure is feature selection.

The ease of use of the forests is particularly obvious in comparison of the neural networks. With the former, the number of hyper-parameters is quite small and well understood. Therefore, tuning the method is easy and can, usually, be undertaken manually with good results. On the other hand, neural networks tuning is much more complex, as even the structure has to be adapted for each problem. Evidence of this complexity is the amount of work dedicated to this subject in the literature.

Lastly, let us mention an interesting characteristic of convolutional networks we did not pursue but which has a important impact on scalability : online learning. Indeed, classification forests require to have the total amount of data right away which will be a limitation of our method.

Validating this hypothesis constitutes our main objective. To achieve this, we developed a method based on classification forests which incorporates some convolutional networks mechanics. More specifically, random linear filters are applied to the image database, followed by one or several spatial poolings. Then, several random subwindows are extracted from each transformed image. Each subwindow is described by the row pixel values. The in-depth description of this "RandConv" method is the main subject of chapter 4. This method builds on previous works. The idea of applying predefined convolutional filters followed by several spatial poolings before extracting subwindows has already been done in . It constituted a generalization of their generic image classification scheme. The contribution of the current paper is two fold. Firstly, the RandConv framework proposes several extensions of that method, the most noticeable of which being the ability to generate the filters. This approach resembles much more the convolution networks', where the filters are actually learned. Secondly, whereas the aforementioned work was more like a proof of concept, the present study aim at analyzing more deeply this method. Indeed, proving the hypothesis is not our only goal. We also want to study closely the behavior of our classification method so as to understand its strength and limitations.

Chapter 4

Methodology

This chapter is divided into three sections. The first one aims at fully describing our classification method. The second section details the experimental condition in which our method will be evaluated. Finally, the last one highlights implementation details and technical issues.

4.1 The RandConv framework

This section is dedicated to an in-depth description of our classification method : RandConv. It stands for "Random and convolutional". The "random" part refer to both the filter generation and subwindow extraction. While the "convolutional" adjective refers to the application of the linear filters.

So as to bridge between classification forests and convolutional networks, we started from the former and added characteristics of the latter. Those characteristics are the convolutional filtering followed by spatial pooling. The RandConv method is divided into the following parts :

1. Generating the N linear filters
2. Applying the N filters to the M images of the databases
3. Applying the P spatial poolings to the $N \times M$ filtered images
4. Extracting S subwindows from each of the $N \times M \times P$ pooled and filtered images and resizing them
5. Describing each of the $N \times M \times P \times S$ pieces by a set of F learning features each
6. Reorganizing the data to create a $(M \times S) \times (F \times (N \times P))$ learning matrix

The pseudo code is presented by algorithm 1. The algorithm can be parallelized easily by subdividing the image dataset in several pieces and reassembling the learning submatrices accordingly. It is also intended to be able to use the original image as if the first filter was the identity filter. The poolings, however, are still applied to it.

Although, the method has been designed with the use of classification forest in mind, the RandConv method, *per se*, is actually a feature extraction method. Its goal is to transform a set of images into a set of corresponding feature vectors. The actual classification could be carried out by any traditional learning algorithm. Nevertheless, regarding our primary objective and some other attractive properties of the trees, which will be developed in subsection 4.1.1, we will stick with classification forests in one way or another.

Algorithm 1 RandConv extraction algorithm

```
1:  procedure PROCESS(RandConvInstance, images)
2:      rci ← RandConvInstance
3:      N ← rci.nbFilters
4:      P ← rci.nbPoolings
5:      S ← rci.nbSubwindows
6:      F ← rci.nbFeaturesPerSubwindows
7:      M ← images.length
8:      learningMatrix[M × S][F × (N × P)]
9:      row ← 0
10:     colMin : colMax ← 0 : F
11:     for image ∈ images do
12:         cropboxes ← rci.generateSubwindowLocations()
13:         for filter ∈ rci.filters do
14:             for pooling ∈ rci.poolings do
15:                 filtered ← rci.applyFilter(filter, image)
16:                 pooled ← rci.applyPooling(pooling, filtered)
17:                 for cropbox ∈ cropboxes do
18:                     subwindow ← rci.extractSubwindow(cropbox, pooled)
19:                     learningMatrix[row][colMin : colMax] = rci.describe(subwindow)
20:                     colMin : colMax ← colMax : colMax + F
        return learningMatrix
```

4.1.1 Filter Generation and application

Mimicking the convolutional filtering is carried out by generating random linear, spatially invariant filters. More precisely, we generate the 2D finite impulse response matrices. First, the filter dimensions and then the filter coefficients are randomly drawn. This means that, contrary to the ConvNet, the coefficients are not directly learned. The coefficient learning is simulated by generating a vast number of filters and letting the learning algorithm choose the ones to emphasis.

This calls for an important remark : decision tree-based solutions are ideal classifier candidates. Firstly, their construction technique allow them to emphasis easily the interesting filters. Secondly, they deal well enough with numerous, possibly irrelevant, features. Indeed, the major impact is a reduction of the model effective complexity. The resulting accuracy drop is much less tremendous than with some other classifiers. Besides, this reduction of complexity can be balanced by the number of subwindows extracted from each image. Augmenting the dataset produces deeper trees ; more complex model. Lastly, they scale well enough due to their relatively low computational cost, especially the extremely randomized tree variant.

4.1.1.1 Drawing mechanism

How to draw the filters is one of the RandConv framework cornerstone. The drawing mechanism should meet two prerequisites. Firstly, it should be able to produce unlimited, or at the least great, number of different filters. Secondly, the filters should be of some value by themselves but also together. Intuitively, a valuable filter should highlight "information" not directly accessible from the original image by the learning algorithm. We will call this characteristic the individual usefulness or simply usefulness. As for having value together, two different filters should not uncover the exact same "information". For instance, producing twice the same filter is useless. We will call this the group usefulness or co-usefulness.

Several drawing mechanisms have been developed with different characteristics in mind :

↬ Custom filters

- Discrete law generator
- Zero-perturbation generator
- Identity-perturbation generator
- Maximum-distance-from-identity generator
- Stratified perturbed generator

The first one is a special case. It consists of a set of 38 well known filter, among which the Sobel and Prewitt filters, several Laplacian filters of different sizes, the compass gradient filters, some low and high pass filters and other line detection filters. Being a small set, it violates the first prerequisite. However, this pseudo filter generator will be useful as a comparison basis : the filter are the same ones as in . Besides, these filters have practical application cases which random filters might not share. It is thus a reference point to see whether the generated filters highlight interesting "information".

The other mechanism draw randomly the filters. Before generating the coefficients of a filter, its dimensions must first be determined. The widths and heights of the impulse response matrices are drawn from an bounded set of odd, positive integers. Although we limited our tests to square matrices, this is not a strict requirement. We mainly worked with a uniform distribution of sizes, playing somewhat with the set bounds. Once again, this is not a limitation as other distribution can easily be used. For example, it is possible to create a distribution biasing towards small sizes. As for the bounds, a minimum seems to be 3. The maximum size should not be greater than twice the image size but needs probably not be greater than half this size. Indeed, greater filters might incorporate mostly non-local information. Conceptually, for a given maximum size, say $n = h \times w$, it is easy to build a bijection between the filter matrices space and $\mathbb{R}^n$. This representation will help us visualize the drawing mechanisms.

Discrete law generator. Once the size is fixed, every coefficient is drawn for a predefined discrete law. Even though the number of such filters is bounded for a given maximum size, this filter space is still vast enough so as to meet the first generator prerequisite. We tested the following law : -1 with a probability of .3, 0 with a probability of .4 and 1 with a probability of .3. This generator was motivated by the spatial interpretation of the convolution. It accounts for summing and subtracting neighboring pixel together.

Zero-perturbation generator. Once the size is fixed, every coefficients are drawn from the same continuous probability law. Although there is no restriction on the probability law, we expect it to be symmetrical and zero-centered, hence the generator name. We used two such laws. The first one is the uniform law over reals bounded with -1 and 1. In this respect, the generator space is mappable to an hypercube centered on the origin. The second law was a Gaussian so that the probability of being outside the range [-1, 1] is equal to a given threshold. The isoprobabilities thus form hyperspheres. The points lying outside of the range can be forced to the boundary so that the generator space becomes the same hypercube as with the uniform law. Zero-centered generator were motivated by the examination of common filters which portray the same characteristic.

Identity-perturbation generator. Identity-perturbation generator work in the same fashion as its zero-perturbation counterpart. The only difference is that the hyper-structures are centered around the identity filter instead of the origin. The motivation behind this generator was to produce filtered images resembling the original while being different enough so as to be of value.

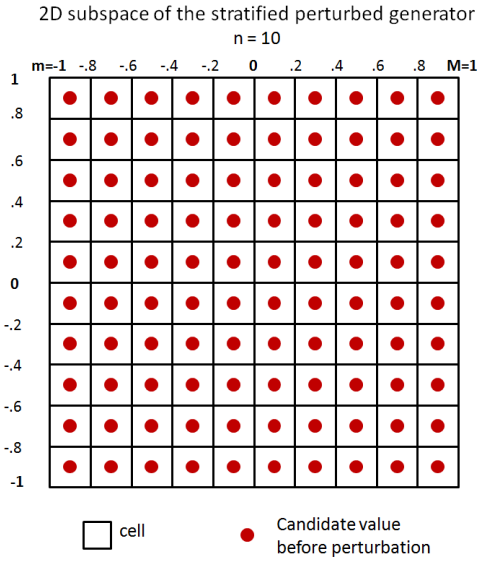


Figure 4.1: 2D cut in the stratified generator space (10 subdivision between -1 and 1, no perturbation)

Maximum-distance-from-identity generator. This kind of generators fulfills the same purpose as the previous one. The generator space is also centered on the identity filter but its shape is different since we decided to work with the Manhattan distance. Concretely, the generator is parametrized by a maximum distance, independent of the filter sizes. The coefficients are processed in a random order. A random perturbation from the range [-maximum distance, maximum distance], expectedly from a uniform distribution, is applied to the first coefficient. Before processing the next coefficient, the maximum distance is updated by subtracting the absolute value of the perturbation.

Stratified perturbed generator This last class of generators are parametrized by a minimum value m, a maximum value M and a subdivision number n. For each coefficient, a value v from the set $\{m + \frac{k+1}{2} \times \frac{(M-m)}{n} | k \in \mathbb{Z}, k < n\}$ is chosen randomly. This value is then randomly perturbed before being assigned to the coefficient. The perturbation is not mandatory and should stay in the range $[-\frac{(M-m)}{2n}, \frac{(M-m)}{2n}]$. Expected perturbation law are Gaussian and uniform. This is illustrated by figure 4.1

This generator class was motivated by the idea to produce as dissimilar filters as possible so as to meet our second requirement about co-usefulness. Disregarding the perturbation, the filter space is finite but still huge. For instance, the space for a subdivision number of 10 with only the smallest filters (3x3) would still mean 10^9 filters. Whereas, 2^9 filters, *i.e.* a subdivision number of 2, is manageable, the other generators are able to produce filters as dissimilar. Furthermore, the following non-monotonicity property suggests that a dissimilar approach in the filter space might not be the best way to produce sets of co-useful filters. Indeed, we can use the distance to measure co-usefulness : if two filtered image are close, they probably highlight the same "information".

Non-monotonicity property. We will show that closeness in the filter space does not necessarily imply closeness of the filtering results. Closeness is to be understood as distance from a reference. Let I be an image and F, F_1, F_2 be three linear, spatially invariant filters

of possibly different sizes. Let also

$$J = I * F$$
$$J_1 = I * F_1$$
$$J_2 = I * F_2$$

We will show by counterexample that $\| F - F_1 \| \geq \| F - F_2 \| \not\Rightarrow \| J - J_1 \| \geq \| J - J_2 \|$. First let us name $e_1 = F - F_1$ and $e_2 = F - F_2$. By linearity of the convolution, we have :

$$J_1 = I * F_1 = I * (F - e_1) = (I * F) - (I * e_1) = J - (I * e_1) \iff J - J_1 = I * e_1$$

In these terms, we have to show that $\| e_1 \| \geq \| e_2 \| \not\Rightarrow \| I * e_1 \| \geq \| I * e_2 \|$. Let us take e_1 such that the coefficients sum up to zero but with a great dispersion (a Sobel filter, for example) and e_2 such that the sum of the coefficients is strictly greater than zero but with a smaller dispersion than e_1 (the 3x3 average filter, for instance). Thus, we have $\| e_1 \| \geq \| e_2 \|$. Moreover, let us consider the case of a image I with constant value $c > 0$. In this setting, $\| I * e_1 \| = 0$ while $\| I * e_2 \| = c \times k > \| I * e_1 \|$.

Therefore, playing with closeness or dissimilarities in the filter space yield no warranty about the same metrics with the filtered images. However, using the distance as measure of co-usefulness is arguably a poor choice, since close images might still highlight different aspects of the images. Considering this remark, the main shortcoming of the stratified generator is probably that, with respect to the number of generated filters we will use, it does not produce any significant advantage over other generators.

4.1.1.2 Normalization

All the generators we discussed in the previous section are able to perform a post-processing normalization of the filter. There are four normalizations :

- No normalization : the post-processing normalization is skipped.
- Zero mean : the mean value of the filter coefficients is null.
- Unit variance : the coefficients have a unit variance.
- Zero mean and unit variance : both the previous. First the zero mean then the unit variance.
- Unit sum : the coefficients sum to one.

The introduction of the zero mean and unit variance normalizations was primarily motivated by supplying support for learning algorithms other than classification forests. Indeed, their effect is to impose a common dynamics to all the filters. While trees can cope easily with variables of different dynamics, some classification schemes are not applicable is that setting or suffer greatly from it. As for the unit sum normalization, applied in conjunction with a generator producing positive coefficients, it produces "convex combination filters" in the following sense : for each step of the convolution, the output pixel value is a convex combination of the minimum and maximum of the neighboring original pixels (where the neighborhood is defined by the filter size). We will now look at the implication of the normalizations on the generator space and the filtering in both the spacial and frequency spaces. We will reuse the filter representation in $\mathbb{R}^n$ and will denote by $\mathbf{1}$ the vector whose coefficients are all 1.

Zero mean normalization. In $\mathbb{R}^n$, the zero mean filters form the hyperplane $\{x \in \mathbb{R}^n | \mathbf{1}^T x = 0\}$. The normalization is a projection onto that hyperplane. The resulting filter y is computed as $y = x - \frac{1}{n}\mathbf{1}^T x \mathbf{1}$. This operation can produce a filter which is outside of the original filter space. Since this operation is linear, the impact of the filtering are

straightforwardly identifiable. Denoting I a given image, x_f a given filter, whose mean m form the constant filter m_f, y_f the normalization $y_f = x_f - m_f$ and $\mathbf{1}_f$ the constant filter with only ones as coefficients, we have :

$$I * y_f = (I * x_f) - (I * m_f) = (I * x_f) - m \times (I * \mathbf{1}_f)$$

The $(I * \mathbf{1}_f)$ correction part is independent of the filter and proportional to the mean coefficient value. The practical impact is clearer in the frequency space. Let us denote by $\rightleftharpoons_{\mathcal{F}}$ the Fourier transform :

$$I \rightleftharpoons_{\mathcal{F}} Ux_f \qquad \rightleftharpoons_{\mathcal{F}} H_x m_f \rightleftharpoons_{\mathcal{F}} H_m y_f \qquad \rightleftharpoons_{\mathcal{F}} H_y = H_x - H_m$$

$$I * y_f \rightleftharpoons W = U \times H_y = U \times (H_x - H_m) = (U \times H_x) - (U \times H_m) = Y - (U \times H_m)$$

Since m_f is a constant signal, the transfer function H_m is null everywhere except at the origin. Thus, the overall frequency response is only marginally modified and both filter achieve the same results. Therefore, the normalization does not restrict the class of filters.

Unit variance normalization. In $\mathbb{R}^n$, the unit variance filters form the hypersphere $\{x \in \mathbb{R}^n | x^T x = 1\}$. However, in practice, the normalization works with the current filter size and not the maximum filter size. Thus, there are several hyperspheres to consider, one per possible size. In the filter space, the normalization equals to scaling the filter so as to meet the appropriate hypersphere. The impact on filtering is immediate :

$$I * y_f = I * (\frac{1}{\sigma_x} x_f) = \frac{1}{\sigma_x} (I * x_f)$$

The whole result is scaled by the same factor. Therefor, the normalization does not restrict the class of filters.

Unit sum normalization The reasoning is identical to the zero mean normalization except for the hyperplane : $\{x \in \mathbb{R}^n | \mathbf{1}^T x = 1\}$. This normalization does not restrict the class of filters either.

4.1.1.3 Filter application

In this subsection, we cover two topics about the filter application. The first one concerns working with colors. The second one is about how the filter are actually applied.

Handling colors can be done in three ways. The first one, is to realize a 3D convolution. This results in a single output value per original pixel. However, there are two drawbacks to this approach. Firstly, in the spatial space, it means combining values of different colors together. This would work but lacks of physical interpretation. Indeed, the RGB space is only a convention. The second drawbacks has to do with the frequency space. It feels awkward to put on a same level spatial frequencies and color frequency, whatever it might mean.

The second method to handle colors is to use separate 2D filters on each channel. This produces three values per original pixels. As for the last and simplest method, is to use the same 2D filters on each color. This also produces three values per original pixels. Because the last approach seems more natural than the first one and is simpler to interpret, it is the one we adopted.

Now that we know how to handle colors, let us investigate the filter application. The convolution is carried out in the frequency space by multiplying the Fourier transform of the original image by the transfer function of the filter. The output is of the same size as the original image. The borders are handled by padding the original image with zeros.

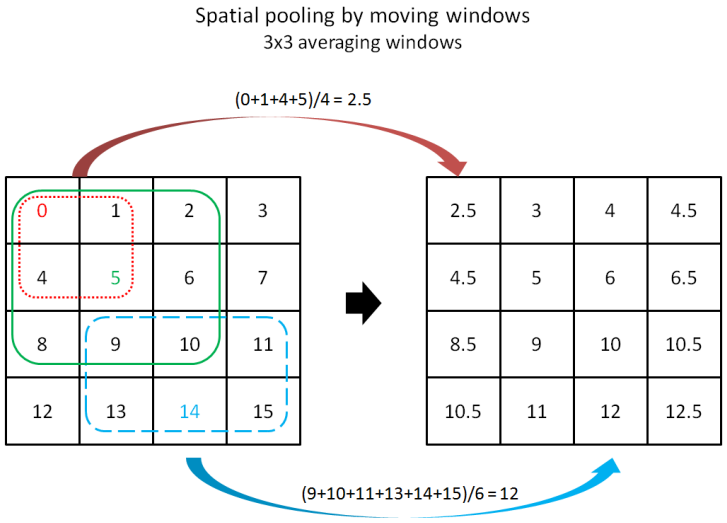


Figure 4.2: Spatial pooling by moving windows

4.1.2 Pooling goals and strategies

Now that we have fully covered the filter generation and application mechanisms, we can move on to the next part concerning the spatial pooling.

Moving windows. In the case of spatial pooling by moving windows, two elements are needed : the moving window size and the pooling function. Windows are supposed to have odd width and height. The center of the window moves to match every pixel of image. The pooling function is computed on the overlapping part of the window and the image. The resulting image as the same size as the original image. This is illustrated by figure 4.2

Aggregations. In the case of spatial pooling by aggregation, the neighborhood windows do not overlap. The image is divided into several non overlapping neighborhood such that each neighborhood has the appropriate size. The pooling function is then applied on each cell of this neighborhood grid. Thus, contrary to moving windows, the resulting image is smaller and correspond to the neighborhood grid layout. This is illustrated by figure 4.3

Pooling functions. The pooling function box is comprise of the minimum, maximum and average functions. In the case of the average function with moving window, we are close to defining a composition of two linear filters. The difference comes from the way the border are handled. In the case of the application of the linear filter, the outside element are replaced by zero while they are ignored in the pooling case.

4.1.3 Subwindows extraction

Once the generated filters and the spatial poolings have been applied, it is time to extract subwindows from the images. Extracting subwindows has several advantages. Firstly, it expands the number of learning objects. Depending on the feature descriptor extraction mechanism, there may be numerous features describing a given image. Using less features but more objects performs usually better. As we mentioned earlier, this is especially true with trees, at least for the "more objects" part, where a greater database means a greater tree and therefore a more complex model. Secondly, this approach is more robust towards scaling and occlusions (Marée et al. (2013)). Lastly, it can be used as a zone of interest detection system, see for example Marée et al. (2006). This is an computationally interesting and domain-free alternative to other techniques. Although this is not the focus of the present work, the RandConv method retains this capability as well. While expanding the number of

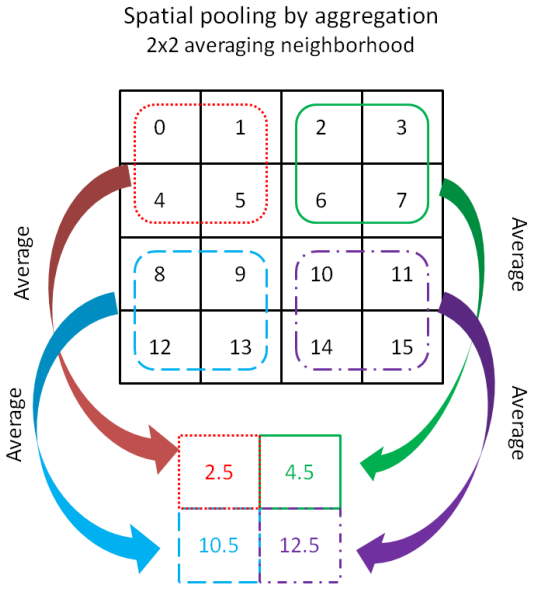


Figure 4.3: Spatial pooling by aggregation

learning objects, we also need to to expand the class label accordingly. Consequently, each subwindows will be described by the label of its original image.

The number of possible subwindows is quite large. First, let us notice that the number of subwindows of size $a \times b$ ($N(a,b)$) factorizes into the product of the number of subwindows along each axis : $N_v(a) \times N_h(b)$. These can be computed easily as $N_v(a) = H - a + 1$, where H represents the height of the image and similarly for $N_h(b)$, which depends on the width W. Indeed, for a column of size H, there is H origins of 1 pixel subwindows. If we take subwindows of size 2, we can take all the same origins as previously except the last one. Subwindows of size 3 cannot take the last two compare to 1 pixel subwindows, and so on. Therefore, the total number of subwindows N is :

$$N = \sum_{a=1}^{H}\sum_{b=1}^{W} N(a,b) = \sum_{a=1}^{H}\sum_{b=1}^{W}(H-a+1)(W-b+1) = \frac{1}{4}(H^2W^2 + H^2W + HW^2 + HW)$$

For 32x32 images, this yields 278,784 subwindows !

As we can see, there are numerous subwindows. Nevertheless, not all are of interests. The small size subwindows do not bring much information. For instance, taking only one pixel is not interesting. Delimiting a good threshold on the size is problem dependent, however. Despite focusing on big enough subwindows, there may still be too many of them. For instance, on 32x32 images, there are still 2025 subwindows of sizes ranging from 24x24 to 32x32. Nevertheless, this entails a redundancy level which is not needed. Since it would be difficult to establish a general heuristic to choose good candidates, we resort to drawing them randomly. At this point we have to be careful. Since we want to describe together all, *i.e.* within the same feature vector, the filtered images in a coherent fashion, we need to extract the same subwindows on all the filtered image belonging to the same original one. The filtering and pooling aim is to better describe a subwindow. However, for two different original images, we may choose two different sets of subwindows. By hypothesis, the chance of drawing twice the same subwindow for a given original image is small. We start by drawing the size uniformly in the affordable range. Then the upper left position of the subwindow is drawn from the possible position considering the subwindow size.

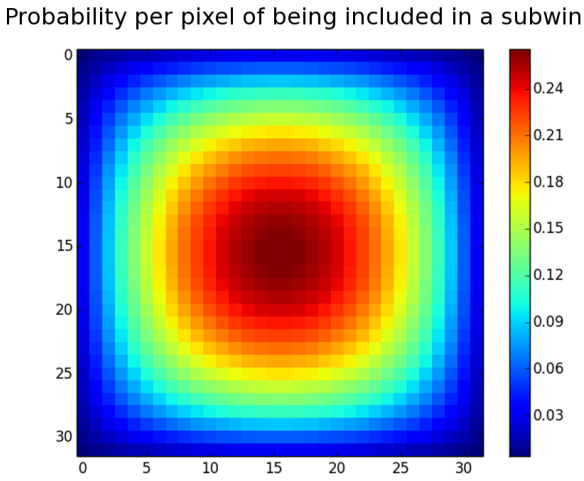


Figure 4.4: Probability associated to each pixel of being picked up

Since the subwindows are chosen uniformly, we can compute the probability of a given pixel belonging to a subwindow by counting the number of subwindows containing that pixel and dividing it by the total number of subwindows. Once again, we will use the fact that we can factorize the numbering for each axis. We will proceed by recurrence. Let us take a column of height H and index the element starting at 1 for the top element and ending at H for the bottom one. We will denote by $T(i)$ the number of subwindows encompassing the ith element. It is immediate that $T(1) = H$: only one subwindow of each length can contain the first element. By symmetry this is also the case for the last element : $T(H) = H$. The second element is encompassed by all the subwindows of the first one but for the one-pixel subwindow. Besides this, we have to add the $H - 1$ subwindows starting at this element. Therefore, $T(2) = T(1) - 1 + (H - 1)$. The reasoning is similar for the next one, the only difference being that now we have to subtract two previous windows from $T(2)$: the monopixel one starting at element 2 and the bipixel one starting at element 1 (its monopixel has already been removed). Thus, $T(3) = T(2) - 2 + (H - 2)$. Expanding the reasoning we get the general formula $T(n) = T(n-1) - (n-1) + H - (n-1) = T(n-1) + H - 2(n-1)$. Resolving the recurrence yields $T(n) = nH - n(n+1) + 2n = nH - n^2 + n$, which verifies $T(1) = T(H) = H$. We just need to pay attention to the fact that we have started numbering at 1, which is not the convention. Coming back to the 2D case, we have that the number of subwindows encompassing a pixel $(r+1, c+1)$ is $T(r,c) = (rH - r^2 + r) \times (cW - c^2 + c)$. Figure 4.4 shows the probability associated to each pixel of being picked up.

Although the subwindows can have different sizes, the feature vector describing a particular subwindows cannot. More specifically, a column of the learning matrix must correspond to a well identified variable. Thus, we need to rescale all the subwindows to a common size. Ideally, the size should be chosen so as to minimize the re-interpolation error. The interpolation algorithms are nearest neighbor, bilinear and bicubic. We will focus on the nearest neighbor because it is faster and it was found to be comparable in term of classification accuracy to the others in most cases (Marée et al. (2014)).

4.1.4 Feature descriptions

Starting from the original database of M images, a set of N filters and P spatial poolings, we produced $N \times P$ images for each original ones. From each of those M sets, we extracted S subwindows on each of the $N \times P$ filter images. We now have $M \times S$ learning objects described by $N \times P$ complex structures. We will describe each structure by a feature vector and concatenate all the feature vectors corresponding to the same subwindow.

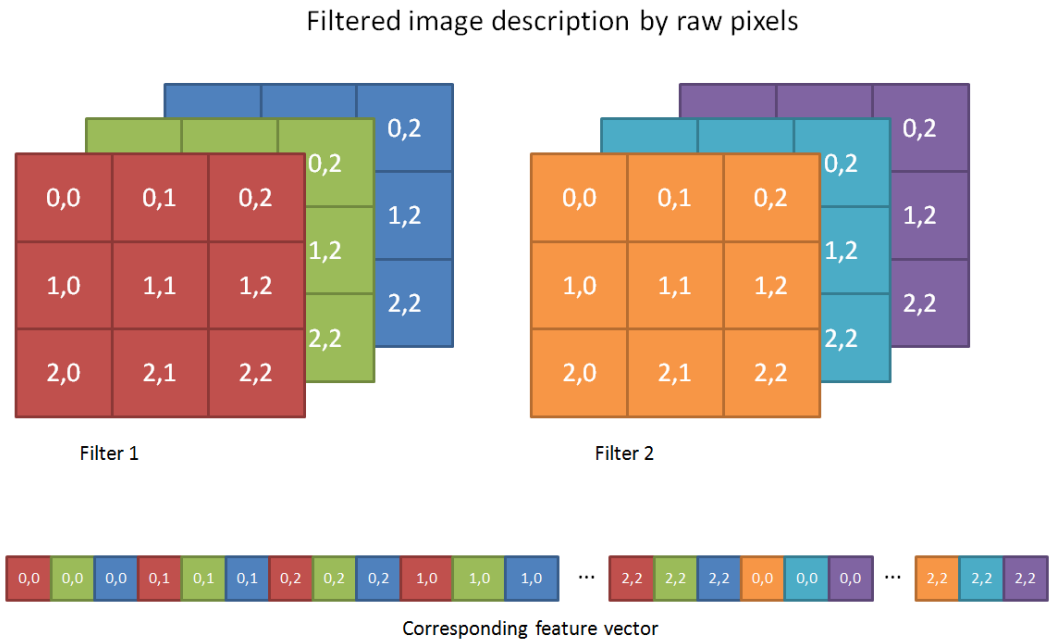


Figure 4.5: Filtered and spatially pooled image description

Each filtered and spatially pooled image will be described by its raw pixels in a last-dimension-first fashion. Therefore, we start by the color dimension and group the three color values of the top left pixel together. Then we append the second pixel (top row, second pixel from the left) and so on for all pixels of the first row. After that, we append the second line in the same fashion and so on for all the image. This is resumed by figure 4.5. Thus, there are $3(h \times w) \times (N \times P)$ features per subwindows. For instance, 100 filters with 1 pooling on 16x16 subwindows yields feature vector of $3(16 \times 16) \times 100 = 76,800$ variables.

4.1.5 Compression layer

An additional so-called compression layer was envisioned for two reasons. Its goal is to reduce the number of features require to describe a subwindow. The first reason was to limit memory usage and the second one was to reduce the relative number of features associated to a given filter compare to the original image so as to put more emphasis on it. Two mechanisms were implemented. The first one is an additional layer in the feature descriptions part and only of subsets of the feature vectors. The second compression layer is situated after the traditional RandConv and works on the whole learning matrix.

Feature subvector compression. In this variant, we treat all the feature subvector separately. A subvector corresponds to the contiguous part of a subwindow feature vector related to one filter and pooling. The idea was to reduce the number of variables of such subvectors while losing as little information as possible. In order to accomplish that, we reduce the number of variable by selecting only one or two colors by pixel. The variables are chosen so that neighbor pixels of the same row have not the same color missing. In that way, we rely on spatial redundancy to limit the loss of information. This is illustrated by figure 4.6. The compression rate is of $\frac{3}{2}$ for a low compression keeping two colors by pixel or $\frac{3}{1}$ for a high compression keeping only one color by pixel. Let us note that this variant does not allow for a separate treatment of the original image.

Learning matrix compression. In this variant, we work on the whole learning matrix by reducing the number of variable associated to each combination of filter and pooling

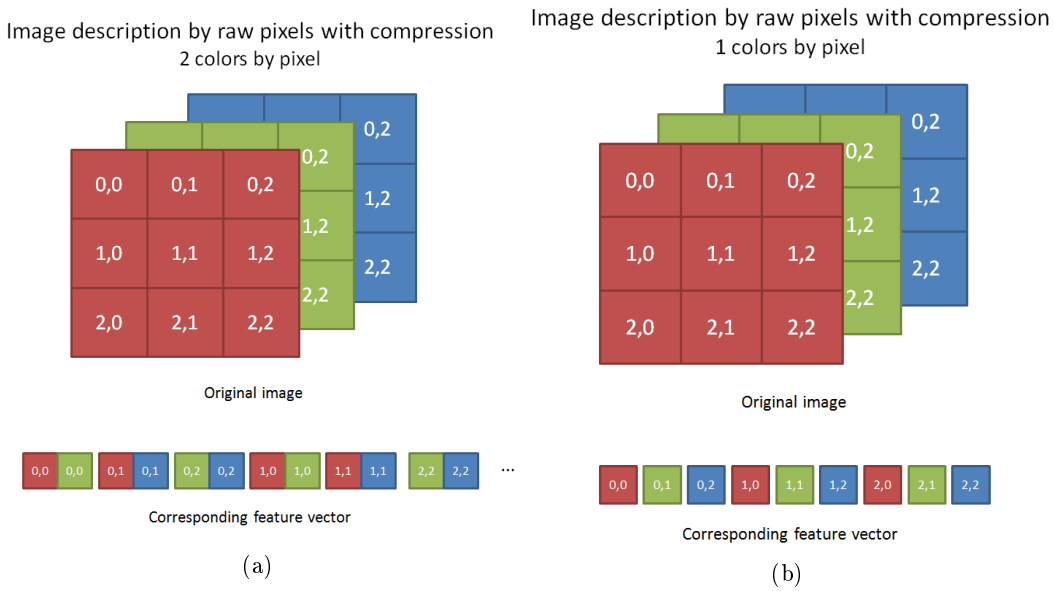


Figure 4.6: Feature subvector compression

with the knowledge of all the learning objects. The method can treat the original image independently and the processing of the other group of features can be parallelized. Using the whole learning matrix allow for more sophisticated compression techniques, which can look for structure in the submatrices. This component works like a learning algorithm as well in the sense that it must fit itself on the learning matrix and then remember its choices for the prediction part. The layer is generic meaning that the processing can be parametrized.

We implemented two such processings. The first one, was simply to draw randomly a subset of different features for each filter. The second one is a bit more elaborated and complex. It computes the principal component analysis (PCA) on all the features of a given filter and selects the first components. Therefore, the new features are linear combination of the original ones. In both cases, the compression rate can be tuned. Although both mechanisms are unsupervised, it is possible to exploit the labels as well.

4.1.6 Classification schemes

The previous subsections cover the preprocessing steps to go from a database of images to an actual learning matrix. Now is time to delve into the actual classification mechanism. We will use two approaches described in the following.

4.1.6.1 Direct classification

In this variant, classification will be undertaken by a special kind of classification forest : ensemble of extremely randomized trees, also known as ExtraTrees.

Ensemble of extremely randomized trees (ExtraTrees)

They were introduce in Geurts et al. (2006) and resemble random forest (Breiman (2001)). In both kind of forests, only a subset of the features are examined at each node to determine the splitting criterion. This approach is called local random subspace and was introduced in Ho (1998). They differ in the following way : while random forests use bagging as an extra mechanism to introduce randomness, the ExtraTrees determine the splitting thresholds randomly. Bagging, for bootstrap aggregating, is the fact of drawing with replacement several learning samples from the original one (boostrap) and combining their predictions (aggregating). The advantage of ExtraTrees over random

forests is threefold. Firstly, the bagging introduces an effective reduction of the learning size of 36% for each tree. This is not the case of the ExtraTrees, where all trees can learn from the whole learning sample. Secondly, they are much faster to build since we do not need to pick the optimal splitting thresholds of all the variables examined in a node. Lastly, they tend to perform better than their counterpart (Geurts et al. (2006)).

In this variant, also called ET-DIC (ExtraTrees for direct image classification), the classification is undertaken by the ExtraTrees directly. Let us note that this scheme allows not only for individual feature importance evaluation but also for filter relevance evaluation by aggregating the importance of all the features corresponding to a given filter. This metric is surely a good candidate for filter co-usefulness evaluation.

4.1.6.2 Feature learning scheme

Also called ET-FL (ExtraTrees for feature learning) or ERC-forests (Extremely randomized clustering forests), this variant was introduce in Moosmann et al. (2008). The ExtraTrees are not used as classifier any longer. They form a preprocessing step whose output will constitute the learning matrix of the actual classifier. The idea is to use the trees to form visual dictionary. Specifically, ExtraTrees are used in an unsupervised way by selecting randomly the splitting criterion of each node, a variant called totally randomized trees. The dictionary words are composed of T parts of different sizes, where T is the number of trees. Each part encodes the index of the leave where a given object ends up in a one-hot fashion. Thus, the part corresponding to a tree with L leaves will be a binary word with $L-1$ zeros and 1 one. In the learning phase, the totally randomized trees are grown from the learning sample, then the same learning sample is propagated down the tree to form the new learning matrix, which is quite sparse. Since we are using several subwindows per original image, we can aggregate by summing bitwisely the words corresponding to the original image to form a new word : an histogram word. The actual classification is then undertaken by a Support Machine Vector.

Support Machine Vector (SVM)

Support machine vector take their roots in the work of Vapnik (1963) and were originally designed as linear classifier for linearly separable tasks. A dataset with binary classes is said to be linearly separable if it is possible to draw an hyperplane in the space of regular features which "separates" the two subsets of data. More formally, the hyperplane $H_{w,b} = \{x \in \mathbb{R}^n | w^T x = b\}$ with $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ separates two points x_1, $x_2 \in \mathbb{R}^n$ if and only if

$$(w^T x_1 - b)(w^T x_2 - b) < 0$$

Let us note that, with this definition, if two sets are separable, there is an infinity of separation hyperplanes. The goal of the SVM is to choose the one which produces the biggest margin : to maximize the minimum distance from any point of the dataset to the hyperplane.

So far, the SVM suffers from three limitations. Firstly, it can only tackle linear problems. Secondly, it is restricted to the class of separable problems. Lastly, it is further confined to binary classification problems.

The first problem can be overcome either by deriving new features through functions of the original ones or via what is known as the "kernel trick". All the computations involving the data are actually inner products. Therefore, one can use any such product instead of the standard scalar one.

Overcoming the linear separability is done by introducing slack variables that allows for points to be inside or even cross the margin, behavior which is penalized in the objective function. This soft margin formulation was proposed by Cortes and Vapnik (1995).

> As for the last limitation, several ways around have been proposed and discussed in the literature (Duan and Keerthi (2005)). A common scheme is the one-versus-all. It consists on training a SVM for each class. The ith model is learned by considering two classes : the ith original class one the hand and all the other classes on the other hand. The prediction is carried out by assigning the class furthest from its margin.

Two remarks are in order. Firstly, we have lost the ability the evaluate the feature importances. Secondly, the learning size, and consequently the number of subwindows, is now crucially important. Indeed, of all the parameters, it is one of the most influent parameters on the trees depth, and consequently the number of leaves. The second most important parameter is the pruning parameter. Pruning is the mechanism of stopping the tree creation before its completion. It can be applied either while developing the tree (pre-pruning) or artificially after the tree creation by cutting of some branches. It obviously also impact dearly the number of leaves. The main goal of pruning is to reduce the model complexity and consequently the overfitting. In the ET-DIC variant, pruning is not necessary because the voting takes care of reducing the overfitting. However, on the data fed to the SVM, there is no automatic overfitting control mechanism and the SVM will suffer from it. Therefore, pruning is also very important for this variant. Last but obviously not least, the number of trees multiply the number of leaves. Therefore, this hyper-parameter is even more sensitive than in the ET-DIC approach.

An extensive comparison of both methods was carried out in . They show that, in most cases, the ET-FL approach performs slightly better (3-4% in average). Nevertheless, ET-FL is more difficult to tune.

4.2 Dataset and environment

We worked with the CIFAR-10 database (Krizhevsky and Hinton (2009)). This dataset is composed of a learning set of 50,000 images and a testing set of 10,000 images. They are grouped into 10 mutually exclusive classes : airplane, automobile, bird, cat, deer, dog, frog, horse, ship and trucks. Both subsets contain an equal number of images from each class. The images have all the same size, 32x32, and are RGB.

This database has been chosen because it was the same one on which the precursory method from was tested. In turn, they chose this dataset because their traditional solution had difficulties with it. Their best solution with the ET-DIC method was an accuracy rate of 53.67%. The corresponding hyper-parameters were the following : 10 fully grown trees, 20 subwindows of 75-100% of the original size reshaped by nearest neighbor interpolation to 16x16 image described by raw pixel values and inspecting all the features on each node. As for the ET-FL variant, the best result was 50.07% of accuracy with 10 almost fully grown trees, 20 subwindows of 75-100% of the original size reshaped in the same manner as before and using $k = \sqrt{M}$ of inspecting features at each node, where M is the total number of features : 16x16x3. The trees are not totally grown : the minimum number of samples to split a node is fix to 10. Contrary to the majority of cases, the best ET-DIC is better than the best ET-FL. Using the aforementioned precursory method, they were able to obtain a significant raise of accuracy, attaining a new record of 74.31% with 750 trees totally randomized trees and 20 subwindows.

The best result on this database is an accuracy of 91.2% and is hold by a convolutional network (Lin et al. (2013)). The top ten best results are above 80%. Most are neural network solutions and none are based on classification forests.

The learning and testing were carried out on a 64 bits 30-core 2.1 GHz computer with 288 GB of RAM.

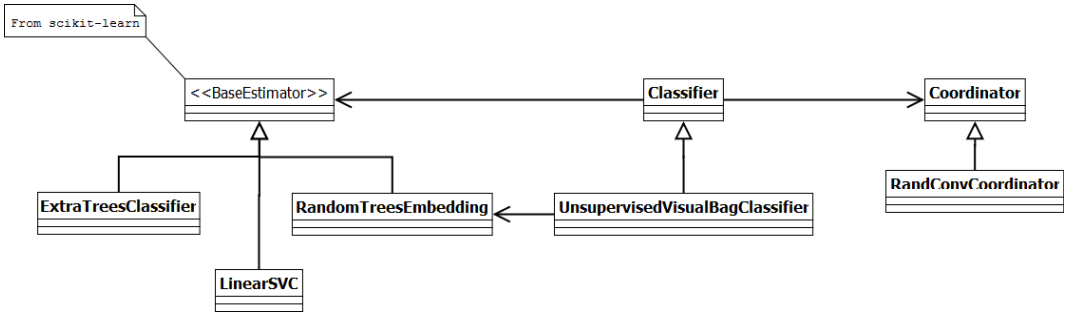


Figure 4.7: UML representation of the `Classifier` class and its major components

4.3 Implementation

In this section, we dive into implementation details. We will not go over all the elements in depth again but rather see how the discussions of the previous sections translate into code. In a second subsection we will also develop technical limitations.

4.3.1 Software architecture

A major concern of the design was to allow as much room as possible for flexibility, so as to be able to develop extensions and variants of the method rapidly. Consequently, the code was split into many classes. A drawback of this is that assembling all the pieces together might be difficult. Factory methods are provided to help with this issue but care must be taken to build up a coherent classifier. We will go back on this in a short while. The code is written in Python 2.7 and relies and the 0.13.3 version of Scipy (Jones et al. (01)), including Numpy 1.8.0. Not all the code is brand new. The ExtraTrees implementation comes from the scikit-learn library (Pedregosa et al. (2011)), version 0.15. We also use scikit-learn for SVM classification, although it actually consists of a wrapper to the liblinear library (Marée et al. (2014)). The SVM computes the optimal linear soft margin and manages multiclass by one-versus-all. As for the subwindow extraction, it is a reorganization of the Pixit implementation (Marée et al. (2014)). We will examine the code in a top-down manner and focus on the important classes.

At the top, we find the `Classifier` class. Its aim is to supervise all the parts. The base class correspond to the ET-DIC variant. The `RandConvCoordinator` is responsible for all the preprocessing : filtering, pooling, subwindow extraction, feature description. After this step, the `Classifier` instance delegates the actual classification to a `BaseEstimator` instance from scikit-learn. In this case, the actual classifier is supposed to be an instance of `ExtraTreesClassifier`. The `UnsupervisedVisualBagClassifier` corresponds to ET-FL mode. Between the preprocessing and the classification, we use the `RandomTreesEmbedding`, a scikit-learn totally randomized trees implementation, to build the histogram which we will then fed to the `BaseEstimator`, supposed to be a `LinearSVC` instance. This is summarized by figure 4.7.

We now go back to the `RandConvCoordinator`. Its responsibility is to transform the subwindows into feature vectors. It proceeds by subdividing the dataset to parallelize the transformation. Then, the `ConvolutionalExtractor` process each image. This entails filtering the image by each element of the `FiniteFilter` thanks to the `Convolver`, then applying all the spatial poolings contained in the `MultiPooler` and finally extracting several subwindows via the `MultiSWExtractor`. Once all this is done, each filtered and pooled subwindow is passed through the `Extractor` and reassembled to form a coherent learning submatrix.

The `FiniteFilter` objects are containers for filters. They pre-generate a finite number of filters thanks to the `FilterGenerator`. We will come back to those in the next para-

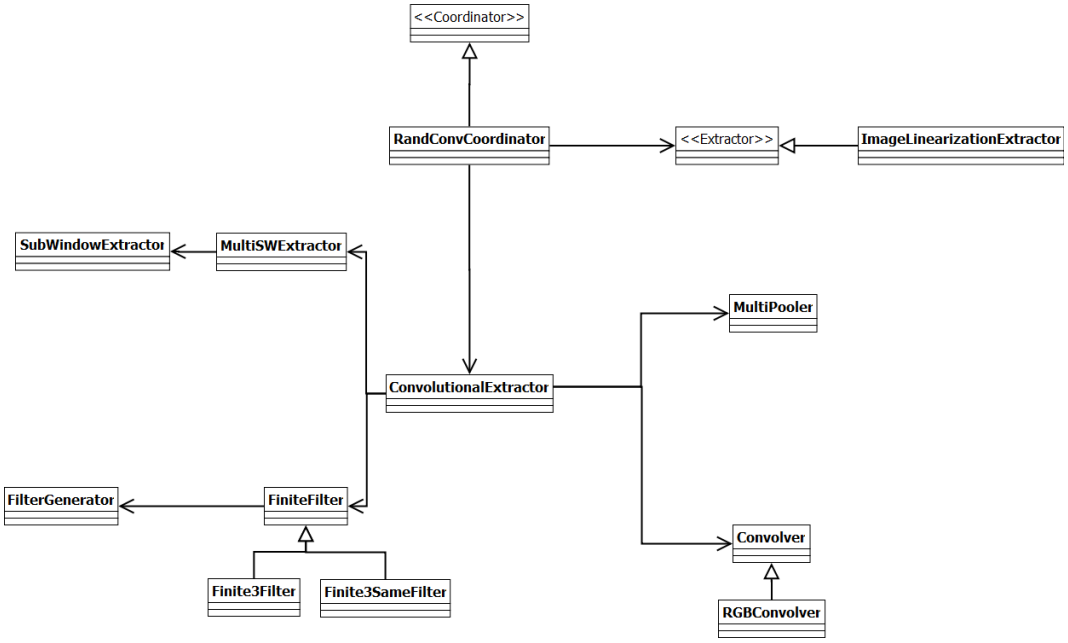


Figure 4.8: UML representation of the `RandConvCoordinator` and its major components

graph. If we are working with RGB images, we need to use either a `Finite3Filter` or a `Finite3SameFilter`. The former produces a different filter per color component while the latter uses the same filter on each color. Also, we need to use an appropriate `Convolver`, namely the `RGBConvolver`.

The subwindow extraction is carried out by the `MultiSWExtractor` whose sole purpose is to keep track of the subwindows to extract for the set of filtered and pooled images belonging to the same original image. The actual subwindow generation and extraction are delegated to the `SubWindowExtractor`.

The transformation from subwindows to feature vector is the responsibility of the `Extractor` instance. In this case, a `ImageLinearizationExtractor` object. However, other mechanism could be implemented, such as extracting descriptive statistics. All this is summarized by the figure 4.8. Let us note that the images are only loaded at the time they are processed so as not to drain useless memory. Python's garbage collector is responsible for freeing them as soon as needed after they were treated.

We now explore the `FilterGenerator`. They need two random number generators. One for drawing the values, either directly or not, and one for drawing the size. The base class is used for two of the generation methods : the discrete law generator and the zero-perturbation generator. The former is made by using a `CustomDiscreteNumberGenerator` while the later uses the base class of `NumberGenerator`. As figure 4.9 displays, there is a class dedicated to each of the other generators.

The `GaussianNumberGenerator` works by specifying a lower bound, an upper bound and the probability of being outside of that range. The `ClippedGaussianRNG` works similarly but in addition forces the values outside of the range to the appropriate bound.

The `MultiPooler` class involved in the `RandConvCoordination` is a container of spatial poolings. As the figure 4.10 depicts, our two groups of spatial poolings are presents.

Other classes are also present to take care of lesser chores, such as loading the data to the right format, logging the progress and so on.

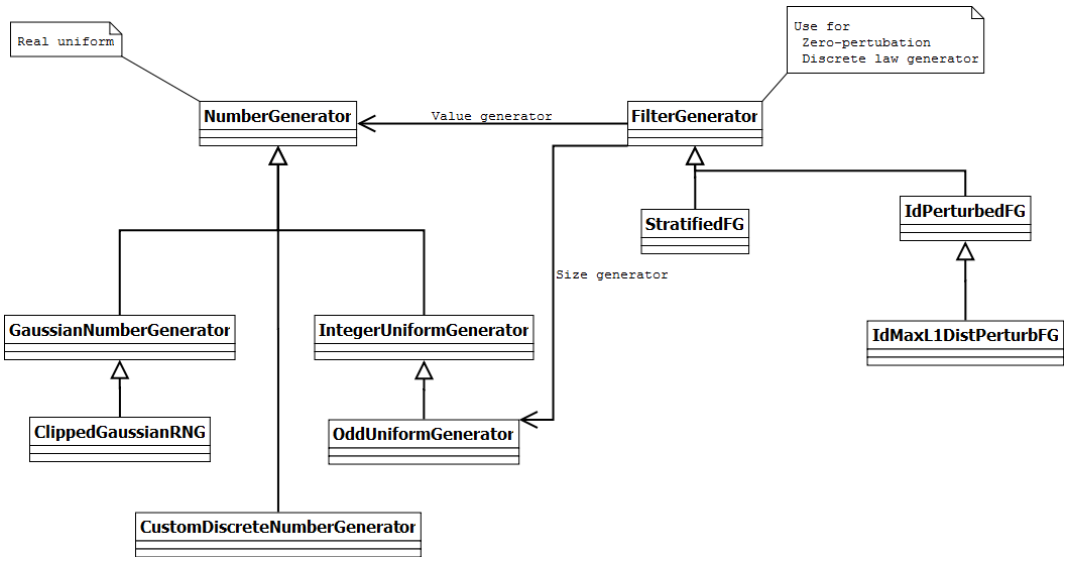


Figure 4.9: UML representation of the `FilterGenerators` and `NumberGenerators`

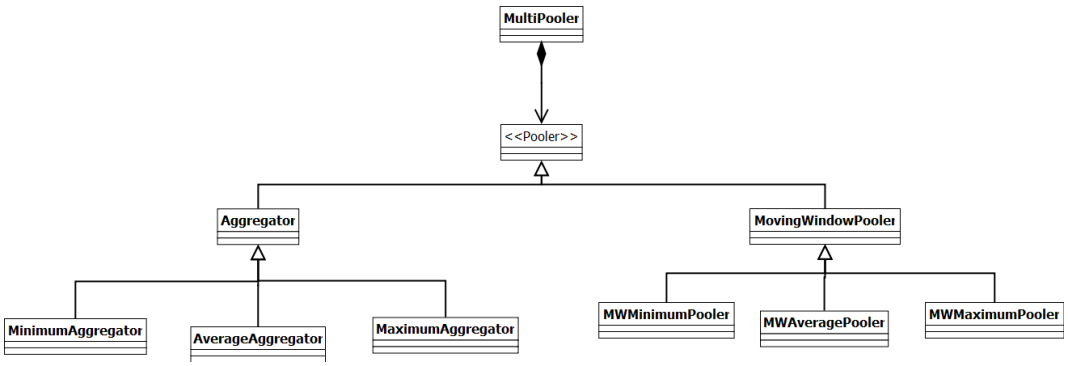


Figure 4.10: UML representaiton of the spatial poolings

4.3.2 Technical issues

The main limitation we will face is memory. The ExtraTrees implementation require 32-bits floats and the SVM, 64-bits floats. However, in the case of the ET-FL, the matrix is mostly sparse on therefore the 64-bits floats requirement of the SVM will not be troublesome. Thus, the space cost bottleneck is the input of the ExtraTrees, which require to hold all the data in memory. Considering 100 filters, 1 spatial pooling, subwindows resized in 16x16, 3 colors, 10 subwindows and the whole learning set (50,000 images) the RandConv method will produce 153.6 GB of data. For 39 filters (the custom filters plus the original image) and 20 subwindows with 9 spatial poolings (the configuration of the best results of Marée et al. (2014)), 1,1 TB would be required. Since we are limited to 288 GB, we will not be able to reproduce such a configuration.

One way of sidestepping this limitation is to build several forests with a different subset of the features, a variant which might be called "global random subset" (GRS) of features. In the case of the RandConv this can easily be done by choosing a subset of filters for each forest.

More generally, a great care has been devoted to limiting memory consumption as much as possible, such as loading the images only on request and avoiding useless copy as much as possible.

To a lesser extend, the time complexity will be an hindrance. It will not actually prevent any computation but we will have to plan carefully the experiment to carry out. For instance, our first example, which produces 153.6 GB of data, takes between 5h and 12h depending on the machine load.

4.4 Hyper-parameters summary

Before elaborating on the results, we will rapidly summarize all the hyper-parameters involved with the RandConv framework.

- Filter generation
  - Size range
  - Whether or not to produce squared filters
  - Value range
  - Filter generator
  - Random law
  - Other filter generator specific parameters (maximum distance, number of subdivisions,...)
  - Filter normalization
  - whether or not to include the original image
- Spatial poolings
  - Number and type of poolings
    - Aggregation or moving window
    - Pooling function : identity, minimum, average or maximum
  - Size of the neighborhood
- Subwindow extraction
  - Number of subwindows
  - Subwindows cropping size
  - Subwindows rescaling size

- ∝ Subwindows rescaling interpolation

↬ ExtraTrees

- ∝ Number of trees (default : 10)
- ∝ Number of features of the local random subspace : k (default : square root of the total number of features)
- ∝ Maximum depth (default : no maximum depth)
- ∝ Minimum sample to split : n_{min} (default : 2)
- ∝ Minimum sample per leaf (default: 1)
- ∝ Whether or not to use bootstrap (default : no bootstrap)

As we can see, the number of hyper-parameters is already large. We can divide them in two categories. On the one hand, we have structural parameters and on the other, traditional hyper-parameters. Structural parameters have a more profound impact than traditional ones. The filter generator and its random law and the number and type of spatial poolings form the structural parameters. Conceptually at least, changing one of them is closer to changing the classification method than only one of its hyper-parameters.

Three of the ExtraTrees hyper-parameters are used to control the pruning : the maximum depth, the minimum sample split and the minimum sample per leaf. The difference between the last two is that the first one does not attempt to split a node whose number of samples are under the given threshold, while the second does the split but rollback if any of the children are under the threshold.

Several of the parameters can be fixed. The value range of the filters will be set as in the filter generator descriptions. Considering the size of the images, we will mostly focus on 3x3 to 9x9 filters. Since we are working with trees, we will use no normalization of the filters. Concerning the spatial poolings, we will also restrict ourselves to small neighborhoods. For comparative purposes, we will resize subwindows to have sizes of 16x16 with nearest neighbor interpolation, as in Marée et al. (2014). Moreover, we will extract subwindows of 24x24 to 32x32 pixels. For the ET-DIC variant, we will use the default trees parameters, which means no pruning. We will also use 30 trees (one per core). However, in the ET-FL approach, we will have to prune the trees and use more of them. The n_{min} parameter will be set to 500 with 750 trees. Table 4.1 displays the fix or default values of the hyper-parameters. Unless stated otherwise, these values are to be assumed in the next chapter.

Hyper-parameter	default value
Filter sizes	3 to 9, both included
Square sizes	True
Filter value range	[-1, 1]
Filter generator	`FilterGenerator` (Zero-perturbed generator)
Random law	Real uniform
Filter normalization	None
Include original image	True
Number of spatial poolings	1
Type of spatial poolings	Average moving window
Size of the neighborhood	3x3 (moving window), 2x2 (aggregation)
Number of subwindows	10
Cropping size	24x24 to 32x32
Rescaling size	16x16
Interpolation	Nearest neighbor
Number of trees	30 (ET-DIC), 750 (ET-FL)
k	Square root of the total number of features
Maximum depth	None
Minimum sample to split	2 (ET-DIC), 500 (ET-FL)
Minimum sample per leaf	1
Bootstrap	False

Table 4.1: Default values for hyper-parameters

Chapter 5

Result analysis

In this chapter, we describe the experiments conducted with our new classification method and analyze their results. The chapter is divided into two sections. The first one tackles the direct classification scheme (ET-DIC), where the forest of extremely randomized trees serves as classificator. The second section describes the other variant where the extremely randomized trees are used to create a visual dictionary (ET-FL), while the actual classification is undergone by a support vector machine.

When the hyper-parameters value are not explicit, the default values are to be assumed (table 4.1 of page 24).

5.1 Direct classification scheme

5.1.1 Accuracy as a function of the learning set size

Our first experiment will be to measure how the accuracies of the different methods evolve as a function of the learning set size. We will test the RandConv method with both types of pooling mechanisms (aggregation and moving window). The aggregation pooling uses a neighborhood of 2x2 while the moving windows are of size 3x3. Both poolings uses an averaging function. We will limit ourselves to the zero-perturbed filter generator for now. In both cases, the same 100 filter are drawn. The 10 extracted subwindows are resized to 16x16. Consequently there are $(16 \times 16 \times 100 \text{ filters} \times 1 \text{ pooling}) \times 3 \text{ colors} = 76,800$ features. The accuracy is measured on the whole testing set composed of 10,000 images. The`pixit` method will serve as reference.

We measure the accuracy for the learning set sizes of 500, 5,000, 10,000, 20,000, 30,000, 40,000 and 50,000. The result of this experiment is depicted by figure 5.1. The conclusion is clear : the aggregation pooling mechanism is not working (0.38 of accuracy). Although being of the smallest interesting size, the non-overlapping neighborhood windows already lose too much of the information. One might argue that using 3x3 moving subwindows allow for that variant to capture more of the spatial structure than 2x2 neighborhood. However, this does not seem to be the case as using 3x3 neighborhoods yields a lower accuracy of 0.35 for the whole learning set. Therefore, the bad performances of the aggregation mechanism must indeed comes from its abusive spatial compression. This is unfortunate because those poolings suffered less from the rescaling of the extracted subwindows. Considering the gap of accuracy between both methods, the following experiments will focus on the moving window poolings.

Other observations are worth mentioning. Firstly, the "good" RandConv method performs well. It is consistently better than the `pixit`, although the difference might not be worth the trouble. The RandConv methods takes several hours whereas the `pixit` requires a couple of minutes at most. Secondly, the graph shows the impact of using only a subset

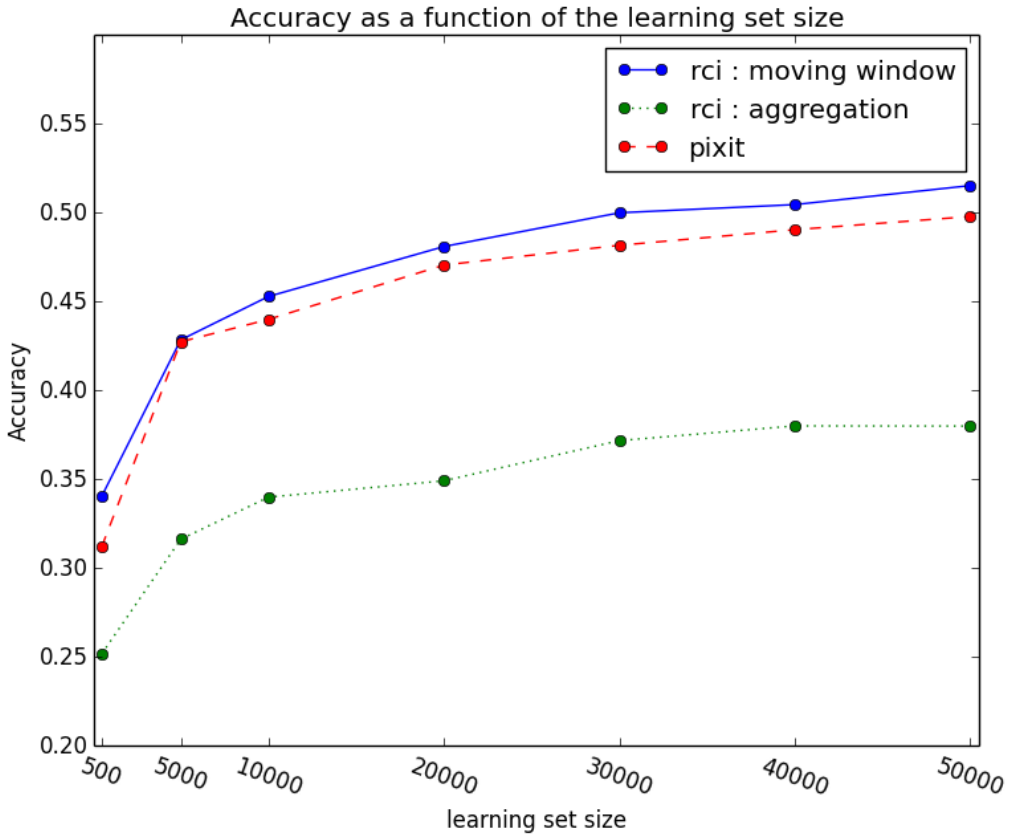


Figure 5.1: Accuracy as a function of the learning set size (30 trees, $k = 277$)

of the data. This information is important in case of hyper-parameter optimization, for instance, where some of the data must be set aside as validation set.

Our best result so far is an accuracy of 0.515 due to the RandConv method with moving windows for spatial pooling. The `pixit` methods with the same parameters yields an accuracy of 0.50, while its best result is 0.53 for 20 subwindows, 10 trees and a minimum sample to split of 10. We will now analyze the variability of the method and how the method varies with some its hyper-parameters.

5.1.2 Variability

We now inspect the variability of the accuracy and of the filter importances. We will first look at the situation where the learning matrix is fixed and the variability can only come from the ExtraTrees. It is important to differentiate both types of variability. The one from the ExtraTrees would mainly be due to the high dimensionality of the problem. Indeed, the size of the local random subspace ($k = \sqrt{76,800} \simeq 277$) is relatively small compare to the total number of features : $\frac{277}{76,800} \simeq 3.6 \times 10^{-3}$. Therefore, different trees will end up making different choices. This is, of course, exactly the behavior we are looking for. However, with as few trees as 30, we may expect that different forests will have a high variability on the filter importances and consequently on the accuracy.

We have run the ExtraTrees 12 times with the same hyper-parameters and the same learning matrix. Table 5.1a holds the accuracies and table 5.1b holds the first line of the correlation matrix of the filter importances between the tests. Figure 5.2 depicts the filter importances for the two firsts tests. The test number 0 corresponds to the results given

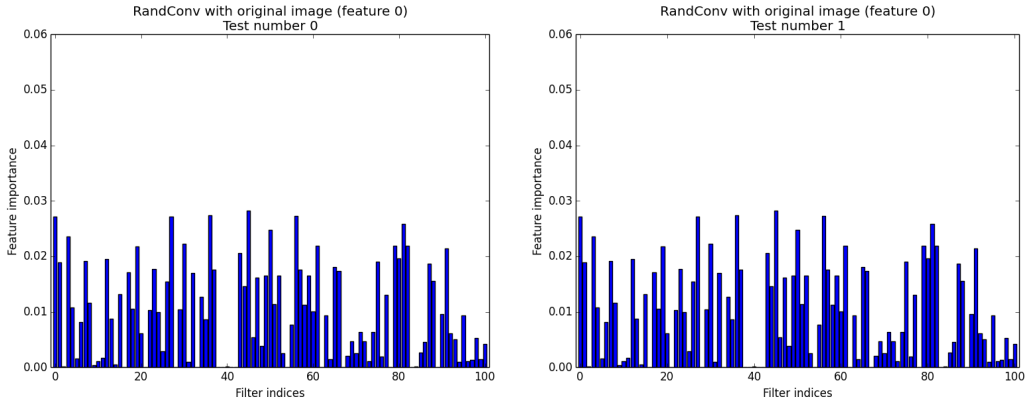
Test number	Accuracy
0	0.5151
1	0.5119
2	0.5118
3	0.5136
4	0.5135
5	0.5141
6	0.5154
7	0.5114
8	0.5145
9	0.5156
10	0.5148
11	0.5118

(a) Accuracy variability

Test number	Correlation with test 0
0	1
1	0.998
2	0.997
3	0.999
4	0.997
5	0.998
6	0.998
7	0.998
8	0.997
9	0.998
10	0.998
11	0.997

(b) Correlation vector of the filter importances with test number 0

Table 5.1: Variability induced by the tree growing algorithm (same learning matrix)



(a) Test 0

(b) Test 1

Figure 5.2: Filter importances for two different forests with the same parameters

in the previous section. As we can see, the accuracy variability is very small. More tests would be needed to establish a rigorous confidence interval but the conclusion seems clear nonetheless : ExtraTrees variability has no impact on the accuracy. This is also backed up by the evident stability of the filter importances. Such stability is due to the smoothing effect of aggregating all the feature importances of a given filter.

A look at the confusion matrices exposes more variability, notably on the diagonal. Despite the accuracy stability, the models are not exactly equivalent. This suggests that combining them or resorting to more trees will allow for some accuracy gain.

Let us digress on the filter importances. From figure 5.2, we can see that several filters have a high importance ; of the same order of magnitude than the original image (filter 0). On the other hand, some filters bring little to no information in the classification process. Either these filters are not (co-)useful or the trees are too small to incorporate their usefulness. In regard to the correlations between tests, however, the co-usefulness and complexity hypotheses are less likely. If it were one of these problems, the low-rated filters would have developed on different trees and therefore the filter importance profiles would be less similar. Therefore, the most likely hypothesis is that those filters are simply not useful. Inspecting them in parallel of the best ones might reveal information about the classification task. In any case, removing them to make room for more interesting filters can only improve accuracy. Sadly, the gain might not be as substantial as we may hope. Indeed, the current accuracy gain compare to the `pixit` is marginal, even though we already have many filters whose importance rivals the original image.

The overall conclusion is the following. For a given learning matrix, increasing the number of trees should not impact the filter importances because they are already quite stable. On the other hand, it should increase the accuracy because the models are not equivalent. The same conclusions should hold for the local random subspace size k, at least for the filter importances. In both cases, a decrease of the parameters may affect both the accuracy and filter profile. This variability analysis bring less insight to the minimum number of samples to split a node, n_{min}. Finally, as far as the accuracy and filter importances are concerned, the forest variability is negligible.

We now turn to the variability produced by the RandConv with the other parameters fixed to their default values. There are three sources of variability :

- ↬ The filter generated
- ↬ The subwindows extracted
- ↬ The trees built

As we have just seen, the last source of variability is negligible, therefore, we will mainly assess the first two ones. We ran 10 more tests whose accuracies are shown in table 5.2. Test 0 corresponds to our previous test. Once again, more tests would be required to establish a proper confidence interval but the variability, although more important than previously, seems to be quite low. Tests number 0 and number 1 are particularly good. Nevertheless, this has to be put in perspective : such low range of accuracies, although always welcome for a practical application, is not really significant in our context.

Since we are using different filters, the importance profiles will be different from one another. Figure 5.3 illustrates this for the tests number 1, 5, 6 and 10. We expect the ideal situation to be when all the filters have the roughly same importances. Nevertheless, how the filter importances influence the accuracy in other circumstances is unclear and cannot be determined from the importance profile. This is made clearer by comparing the cumulative importances of several tests. Figure 5.4 depicts the situation for tests number 0, 1 and 6. For the 30 most influential filters, test 6 is between the other two curves, therefore we cannot predict anything about the accuracy. After that point, test 6 depicts a greater cumulative filter importance. This situation also implies that there are more lowly important filter in test 6 than test 0 or 1. Once again, we cannot deduce anything from it because test 1 is also greater than test 0 in term of both the cumulative filter importance and accuracy.

Test number	Accuracy
0	0.5151
1	0.5183
2	0.5117
3	0.5137
4	0.5095
5	0.5094
6	0.5093
7	0.5117
8	0.5129
9	0.5116
10	0.5124

Table 5.2: Accuracy of several tests

Besides, the cumulative profile of test 5 is nearly identical to the one of test 0. Therefore, it is impossible to assess finely the accuracy from the importance profile. This conclusion is also backed up by the areas under the curves.

So far, we have learned that the RandConv method was surprisingly stable. Fortunately, this will allow us to draw conclusions from few experiments, a welcome surprise considering their running time. We will now inspect further the influence of hyper-parameters with a same set of filters if not a same learning matrix. We will roughly proceeds in a bottom-up fashion : starting from tree-related hyper-parameters and going to purely RandConv parameters.

5.1.3 Influence of the number of trees

The accuracy is usually an increasing function of the number of trees. On the one hand, voting the prediction reduces model variability due to overfitting the data. On the other hand, each tree is unique and aggregating their prediction allow to encompass much more tests than what would be possible with a smaller forest. In this subsection, we will see how the improvement behaves for our method and we will look at the filter importances as well. In section 5.1.2 about the variability, we foresaw that increasing the number of trees would have little influence on the filter importances. Now is the time to validate this thesis. Once again, the other parameters will be set to the default values of table 4.1 on page 24 and the learning matrix is the same in all the tests. Consequently, the only source of variability is due to the forest creation.

In this experiment, we measure the accuracy for the following number of trees : 10, 30, 50, 100, 300 and 500. The result of this experiment is depicted by figure 5.5. As expected, we observe an increasing function of the number of trees. The increase slows down a bit at 100 trees with an accuracy of 0.526. After that, the increase rate drops considerably. Two remarks are in order. Firstly, we observe the classical increasing pattern, which means that the number of trees can be chosen solely with respect to the allowable computational budget. However, for a given learning matrix, the accuracy is upper bounded, as we can see from the rate decrease. Secondly, the 30-trees accuracy is closer to the 500-trees' than to the 10-trees'. Adding our observation about variability, we may conclude that using 30 trees allow us a good assessment of the method characteristics.

We now turn to the filter importances in the classification process. Table 5.3 displays the filter importance correlation between the test with 10 trees and the other tests. As we had predicted, the number of trees does not impact the filter importances. Therefore, using 30 trees is well enough to assess filter usefulness as well.

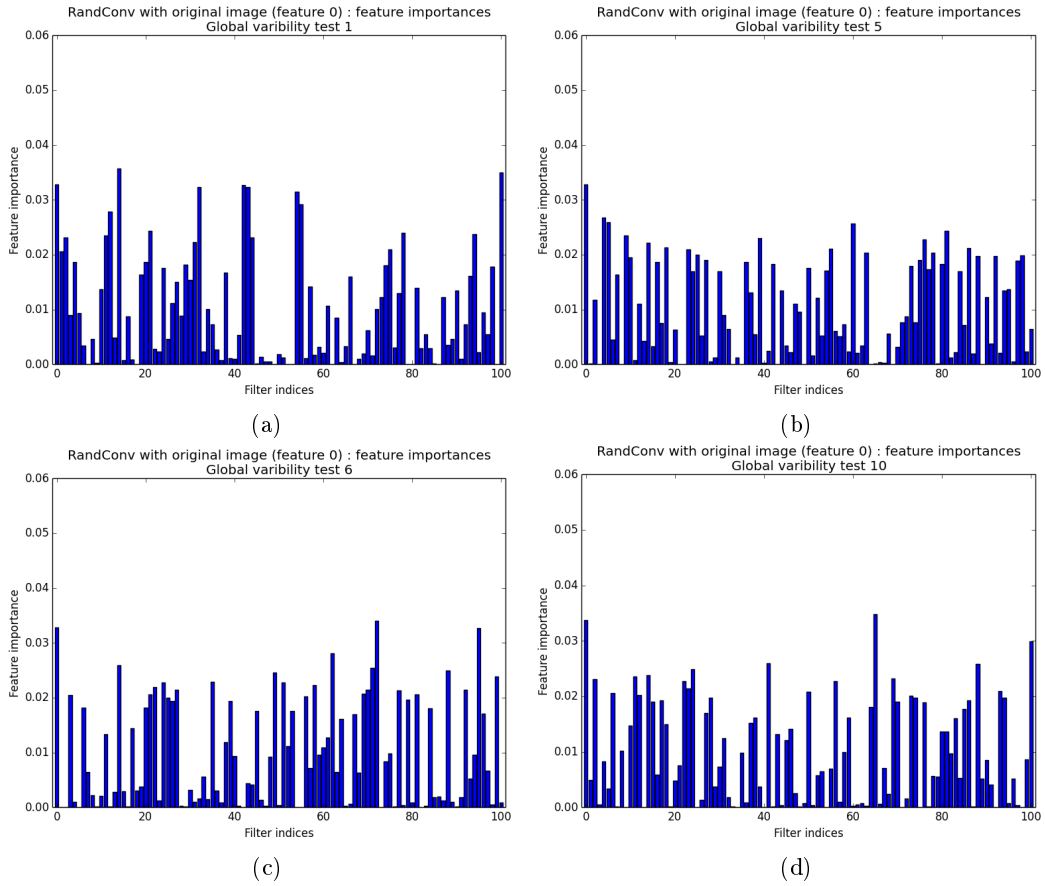


Figure 5.3: Filter importances with global variability

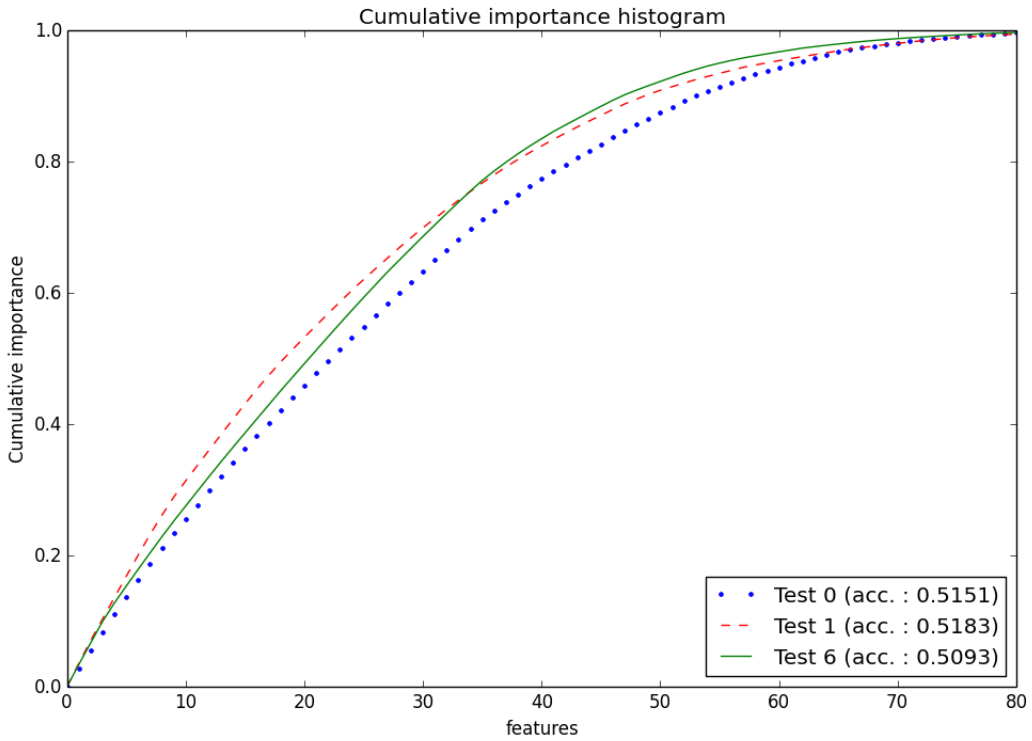


Figure 5.4: Cumulative filter importance (first 80 filters)

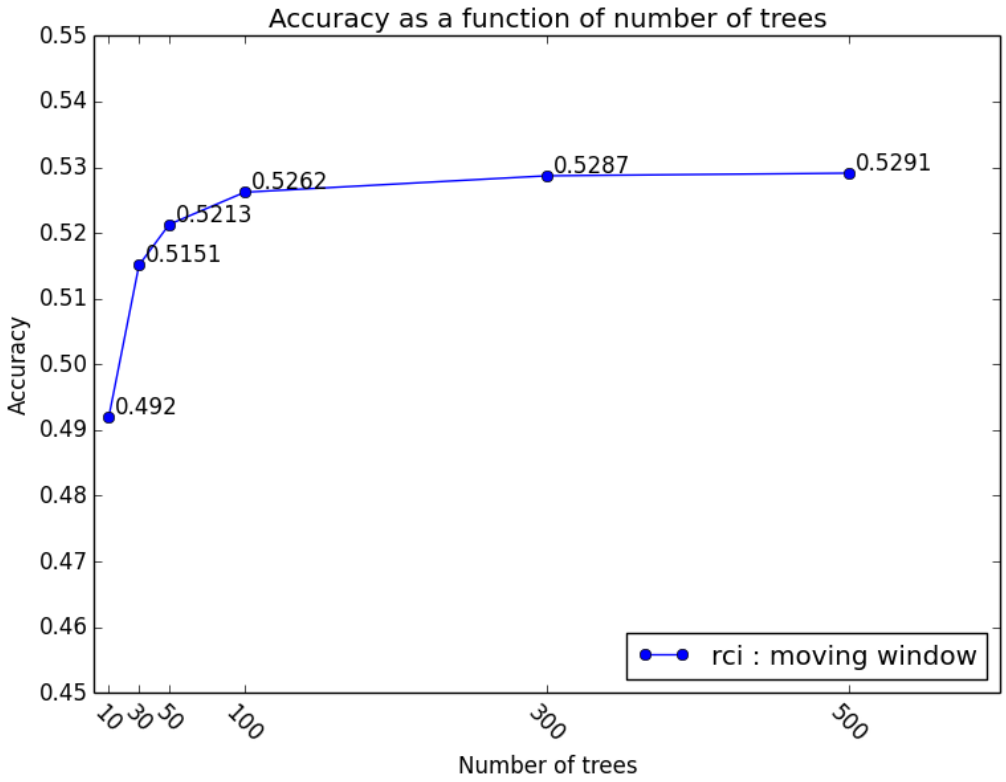


Figure 5.5: Accuracy as a function of the number of trees

Number of trees	Correlation
10	1
30	0.996
50	0.998
100	0.998
300	0.998
500	0.998

Table 5.3: Filter importance correlation with 10 trees

Value of k	Accuracy
1	0.4467
50	0.4969
100	0.5057
277	0.5151
350	0.5172
500	0.5223
10,000	0.5276

Table 5.4: Influence of local random subspace size on the accuracy

5.1.4 Influence of local random subspace size

The parameter k affects the individual tree variability and performances. Too low a value and trees are mostly random, ignoring the data at hand and consequently degrading accuracy. On the other hand, a high value might lead to a large redundancy between the trees, even though this is not the only randomization mechanism. Thus ending up with a lesser accuracy improvement from the forest's voting mechanism. We ran the experiment using the following values of k : 1 (totally randomized trees), 50, 100, 277 (the default value), 350, 500, 10,000. The accuracies are presented at table 5.4. As expected, the low values of k decrease the accuracy. A value of 50 does already quite well (closer to 10,000 than 1 in term of accuracy). It seems that going up to 10,000 (13% of the total number of features) does not yet reduces the accuracy. However, the increase may not be worth the time required to get it (more than twelve hours). Once again, the default value allows for capturing the essence of the method.

The interesting part on the filter importances is that the profile correlation between the totally randomized trees and the default forest is still very high : 0.977. Therefore, it really means that the importances reflect the filter usefulnesses.

5.1.5 Influence of the minimum number of samples to split

This parameter control the model complexity. In the previous experiments, the trees were fully grown ($n_{min} = 2$) and variability reduction was achieve through the smoothing effect of the forests prediction aggregation. Still, the model might be too complex to begin with, thus requiring either some pruning or more trees. Adding more trees did improve the model accuracy but, considering this has several impacts, we do not yet know whether we produce too complex models. So as to dissipate doubts, we tested the following values for this hyper-parameter : 2, 10, 50, 500. The corresponding accuracies are : 0.515, 0.512, 0.498, 0.453. Once again we see that the default value performs well. In other words, Besides, pruning too much the trees is harmful : the models are not complex enough any longer.

The influence on the filter importances is more interesting. From figure 5.6, we clearly can identify two groups of filters. The first one is composed of the filters whose importance increases. This increase implies a decrease of the filter importances in the second group. It is thus possible to see where the filter is used in the trees. The first group holds the top of the tree. This was already obvious for the filters whose importance was either large or low. However, this approach sheds more light on the average filters, where we can observe divergent behaviors. Average filters of the second group are purely co-useful filters : they bring little information on their own but are helpful with the other filters. This characteristic resembles the "V-structures" of the graphical probabilistic models. On a specific classification task, these particular filters may be worth investigating. It might be difficult to identify the filters on which they depend, however.

5.1.6 Influence of the number of subwindows

As a reminder, the number of subwindows enlarges the learning set and therefore allows for more complex models. In this experiment, we tested how accuracy varied with the number

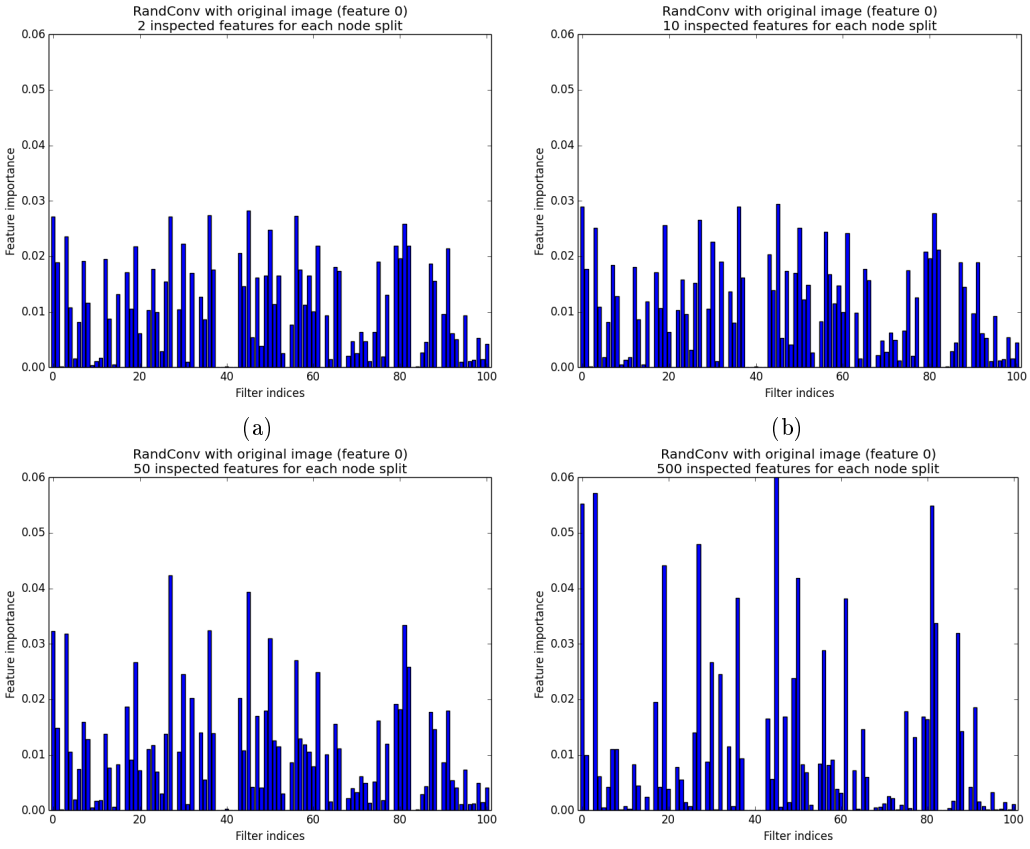


Figure 5.6: Filter importances for different values of n_{min}

Number of subwindows	Accuracy
1 (whole image)	0.44
1 (subwindow)	0.40
5	0.48
10	0.515
14	0.514
18	0.521

Table 5.5: Accuracy as a function of the number of subwindows

Number of filters	Accuracy
10	0.4831
38	0.5087
100	0.5151
120	0.5112
180	0.5150

Table 5.6: Influence of the number of filters on the accuracy

of extracted subwindows. We used 1, 5, 10, 14 and 18 subwindows. The memory limited us to that threshold in order to keep the other parameters to their default values (table 4.1, page 24). We also tested the method where no subwindows are extracted but rather the whole image is used directly. The results are reported at table 5.5. The filters were the same in all the tests.

As we might have suspected, using the whole image yields better accuracy than using only one random subwindows. However, a few of them already accounts for a better accuracy. Judging by the accuracy jump between 5 and 10 subwindows compare to the stagnation between 10 and 14 subwindows, we may conclude that 10 subwindows allows for complex-enough models.

Once again, the filter importances correlation is high. In particular, it is above 0.997 between 5 and 10 subwindows.

5.1.7 Influence of the number of filters

Our expectation regarding the number of filters is that, with few of them, the accuracy might be better or worse than the `pixit`'s depending on the filter usefulnesses. Too many useless filters would hinder the process. With many filters, however, there should be enough usefulness altogether to be able to beat the `pixit` systematically. The accuracy should increase accordingly with the number of filters as long as the learning set size allows for complex-enough model. However, from the comment on the variability (subsection 5.1.2), we suspect that we will reach an upper bound quite fast.

From a more practical point of view, we will have to limit ourselves to 180 filters, which should not be a problem if we were right about the upper bound. Table 5.6 presents the accuracies for 10, 38 (the number of filters in the custom generator),100, 120, 180 filters. As we can see, from 100 filters on, there is no accuracy gain and the filter and subwindow variabilities are responsible for the slight differences of accuracy. We would need a more profound analysis to determine whether or not the accuracy obtained with 38 filters is statically inferior to the 100 filter case's. In any case, it is closer to the accuracy of the 100 filters than of the 10 filters.

5.1.8 Influence of the filter sizes

So far, we restricted the filter sizes to the range 3x3-9x9, implicitly focusing on local characteristics. The idea behind this was that larger filter would incorporate too global information and thus produce more useless redundancy. Besides this, larger filter also means more bound-

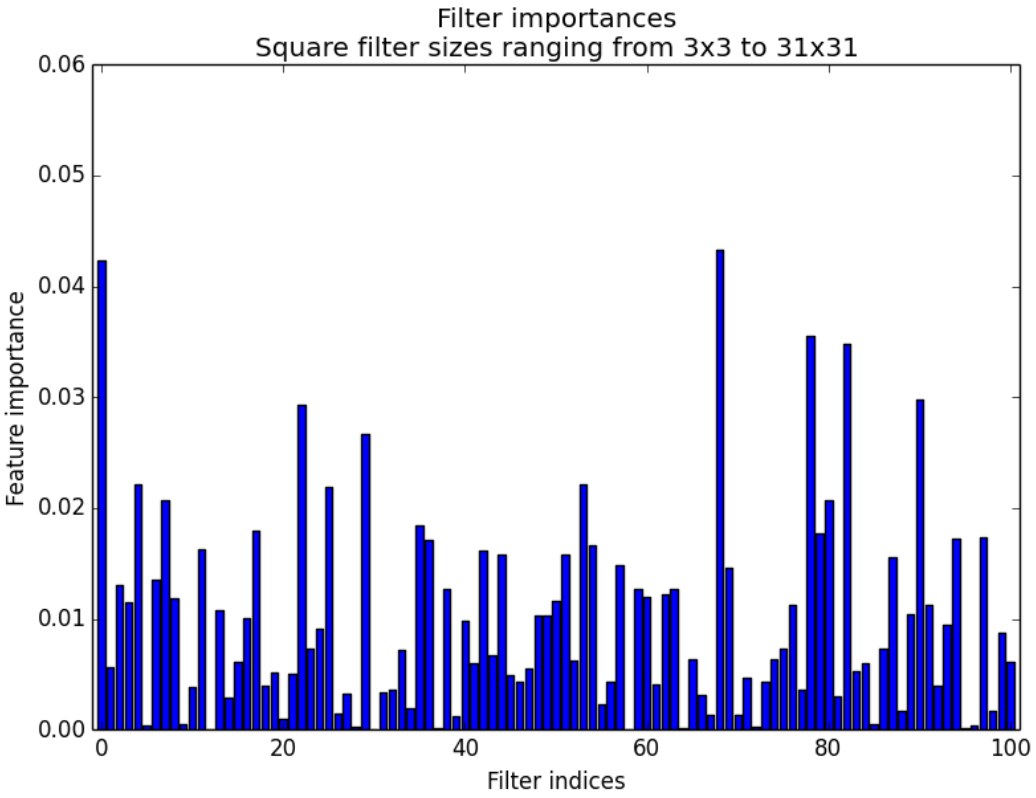


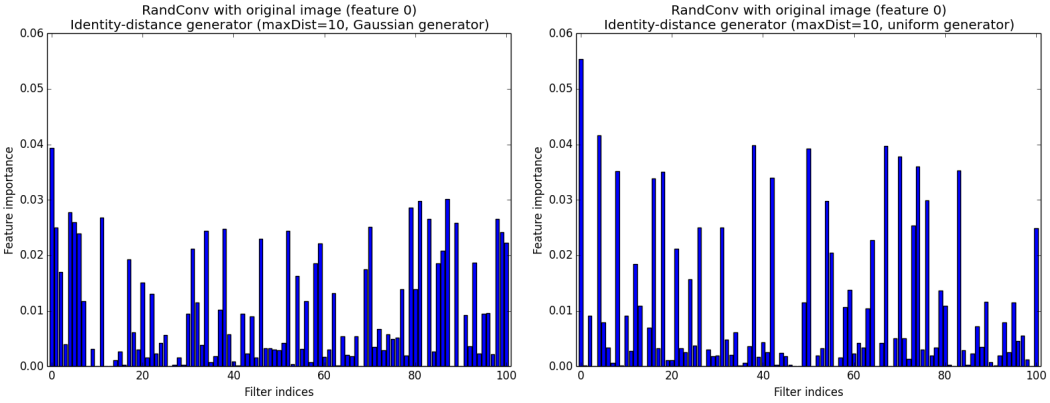
Figure 5.7: Filter importances for filter sizes ranging from 3x3 to 31x31

ary problems. Nevertheless, the best sizes are problem and image sizes dependent. We ran an experiment where we used sizes ranging from 3x3 to 31x31, while other parameters are left to their default value. The accuracy was of 0.4996, slightly less than for the standard case but not small enough to conclude much from it. Inspecting the feature importances brought more insight to the situation. Figure 5.7 depicts the situation. As we can see, the situation is more contrasted than usual. Linking the importances to the sizes, we see that, in the top ten filter importance-wise, all 3x3 filters are present and the remaining three are 5x5. Furthermore, the other five 5x5 filters are in the top twenty, with 7x7 and 9x9 filters, mostly.

5.1.9 Influence of the filter generator

In this section, we analyze the influence of the filter generator. We ran our experiments with the defaults values (table 4.1, page 24), except for the parameters linked to the filter generators. We made the following experiments :

- Custom filters (accuracy : 0.5057)
  - Only 38 filters + the original image
- Zero-perturbed generator with a discrete random law (accuracy : 0.509) :
  - -1 with a probability of 0.3
  - 0 with a probability of 0.4
  - 1 with a probability of 0.3
- Identity-perturbed generator with a real uniform law on $[-1, 1]$ (accuracies : 0.5085 and 0.5154)



(a) Maximum distance of 10 and Gaussian law (b) Maximum distance of 10 and real uniform law

Figure 5.8: Filter importances of the identity-distance generators

- ↬ Identity-perturbed generator with a Gaussian law of 0 mean and 0.51 standard deviation (accuracies : 0.5149 and 0.5135)
- ↬ Identity-perturbed generator with a Gaussian law of 0 mean and 0.36 standard deviation (accuracies : 0.5151 and 0.5119)
- ↬ Identity-distance generator with a maximum distance of 5 and a real uniform law (accuracies : 0.5068 and 0.5029)
- ↬ Identity-distance generator with a maximum distance of 10 and a real uniform law (accuracy : 0.4999)
- ↬ Identity-distance generator with a maximum distance of 5 and a Gaussian law of 0 mean and a probability of being outside of the range $[-maxDist, maxDist]$ of 0.005 (accuracies : 0.5032 and 0.5116)
- ↬ Identity-distance generator with a maximum distance of 10 and a Gaussian law of 0 mean and a probability of being outside of the range $[-maxDist, maxDist]$ of 0.005 (accuracies : 0.5049 and 0.5079)
- ↬ Stratified generator with a subdivision in 10 cells and a Gaussian law of 0 mean and 0.021 standard deviation (accuracies : 0.5088 and 0.5063)
- ↬ Stratified generator with a subdivision in 10 cells and a Gaussian law of 0 mean and 0.036 standard deviation (accuracy : 0.5109)

The standard deviation value of 0.51 is chosen so that the generated values have a probability of being outside the range $[-1, 1]$ of 0.05. The value of 0.36 reduces this probability to 0.005. As for the value of 0.021 and 0.036, they were chosen so that there is only a probability of 0.001 or 0.05, respectively, to be outside the range $[-0.07, 0.07]$. Let us remember that, in the case of the identity-distance generator, the drawing interval is not constant but decreases after each draw.

Judging from those experiments, we can see that identity-distance generator do worse than our usual generator. This is also visible from the filter profiles (figure 5.8), where we can see much more variability than what we were accustomed. Considering the variability we have observed previously, the situation is unclear for the other generators, although the identity-perturbed suites tends to do well. In any case, the filter generator is not the most influential parameter. This is an important, disturbing and revealing result. Our initial expectation laid in the opposite direction : the filter generators were the cornerstone of the method. A good choice (or combination) of generators producing many filters should

have helped unveil useful information for the classification. Sadly, this did not turn out as expected. Our last hope to get significant accuracy improvements now resides in the non-linear layer of spatial pooling. The bright side is that we do not have to worry much about optimizing those parameters ; the zero-perturbed or identity-perturb generators will do just fine.

5.1.10 Influence of spatial poolings

In this subsection, we tackle the influence of the spatial poolings with a couple of experiments. We will resort to the 39 custom filters so as to be able to use several poolings (the original image and the actual 38 filters). The only meaningful source of randomness is the drawing of subwindows. We tested the following scenarios :

- No pooling (accuracy : 0.4816)
- 1 pooling : 3x3 moving windows assigning the minimum (accuracy : 0.5151)
- 1 pooling : 3x3 moving windows assigning the mean (accuracy : 0.5057)
- 1 pooling : 3x3 moving windows assigning the maximum(accuracy : 0.5476)
- 3 poolings : 3x3 moving windows assigning the minimum, the mean and the maximum respectively (accuracy : 0.5454)
- 3 poolings : 3x3, 5x5, 7x7 moving windows assigning the minimum (accuracy : 0.5294)
- 3 poolings : 3x3, 5x5, 7x7 moving windows assigning the mean (accuracy : 0.5173)
- 3 poolings : 3x3, 5x5, 7x7 moving windows assigning the maximum (accuracy : 0.5741)
- 1 pooling : 7x7 moving windows assigning the maximum (accuracy : 0.5675)

We can clearly see that the poolings have a beneficial effect on the accuracy, in particular the maximum. Choosing the best pooling must either be done by cross-validation or not done : using several poolings should not reduce the accuracy dramatically. The main drawback of this last option is that the memory might be better used in another fashion. Using the same pooling at different spatial scales may or may not be interesting.

We now turn to the filter importances. Figure 5.9 depicts the profiles for the single pooling cases. Some filters always have a high or low importance. For other filters, however, the importances vary. Concerning the profiles, if not the accuracies, the no pooling and average strategies are close, with a correlation of 0.954 (see the correlation matrix on table 5.7). The major difference comes from the number 20, which is a high pass filter. The other pieces of information given by the correlation matrix are as one might expect. More interesting, we can observe that the profiles can be importantly influenced by the pooling, which means that it does bring information and increase some filter usefulnesses. In the occasion of filter 28 (the west compass gradient mask), the maximum pooling even goes as far as giving it more importance than other filters appreciated by the remaining poolings. Interpreting the cumulative importance diagram (figure 5.9) is, once more, complicated and we will limit ourselves to the following observation. The maximum pooling is the closet example to a straight line (the ideal situation where every filter has the same importance) we have seen so far and also scores the best accuracy for now. Despite the closeness of the no pooling strategy, however, it does not score a high accuracy.

The observation of single pooling strategies bode well for their combine application. Yet, it may happen that they shade each other in a co-utilization scenario. In turns out this is a empty threat. The correlation between importances in single pooling strategies and their corresponding importances in multipooling strategies are above 0.96 for all of them (no pooling included). This is visually illustrated by figure 5.11. Each group of four bars belong to the same linear filter.

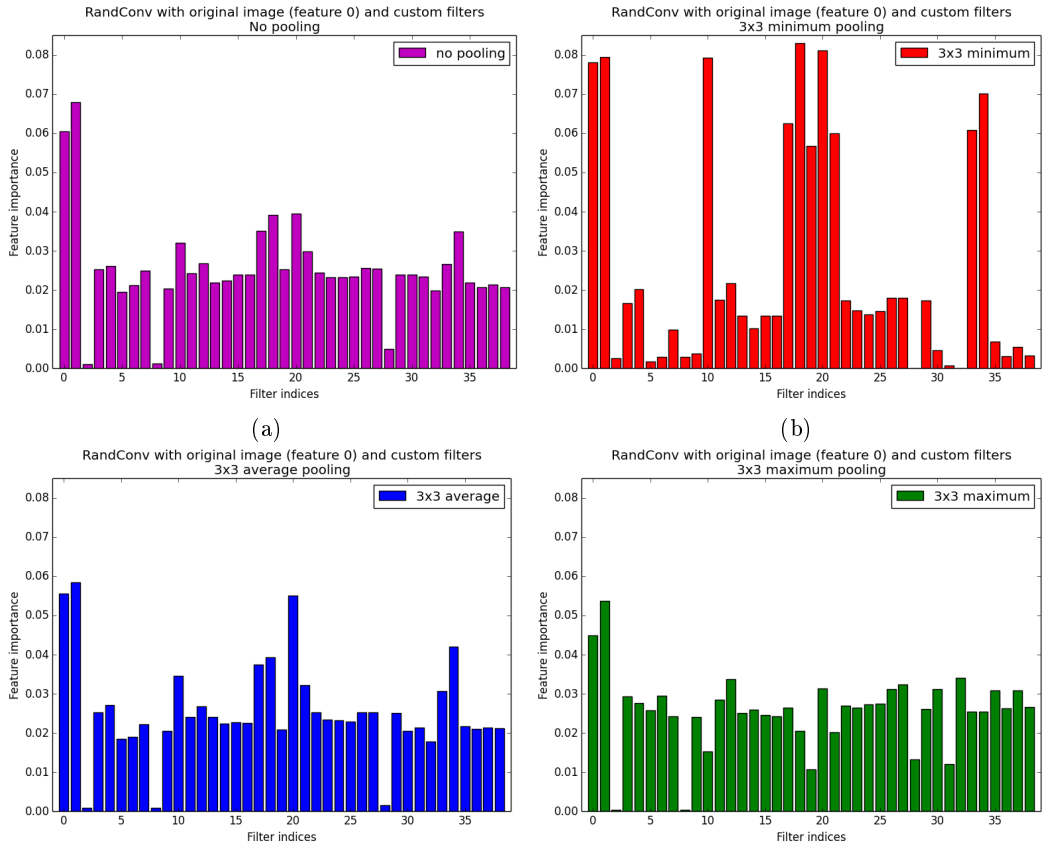


Figure 5.9: Filter importances with different types of poolings

Pooling	no pooling	3x3 minimum	3x3 average	3x3 maximum
no pooling	1			
3x3 minimum	0.762	1		
3x3 average	0.954	0.826	1	
3x3 maximum	0.723	0.226	0.661	1

Table 5.7: Correlation matrix of the filter importances between the poolings

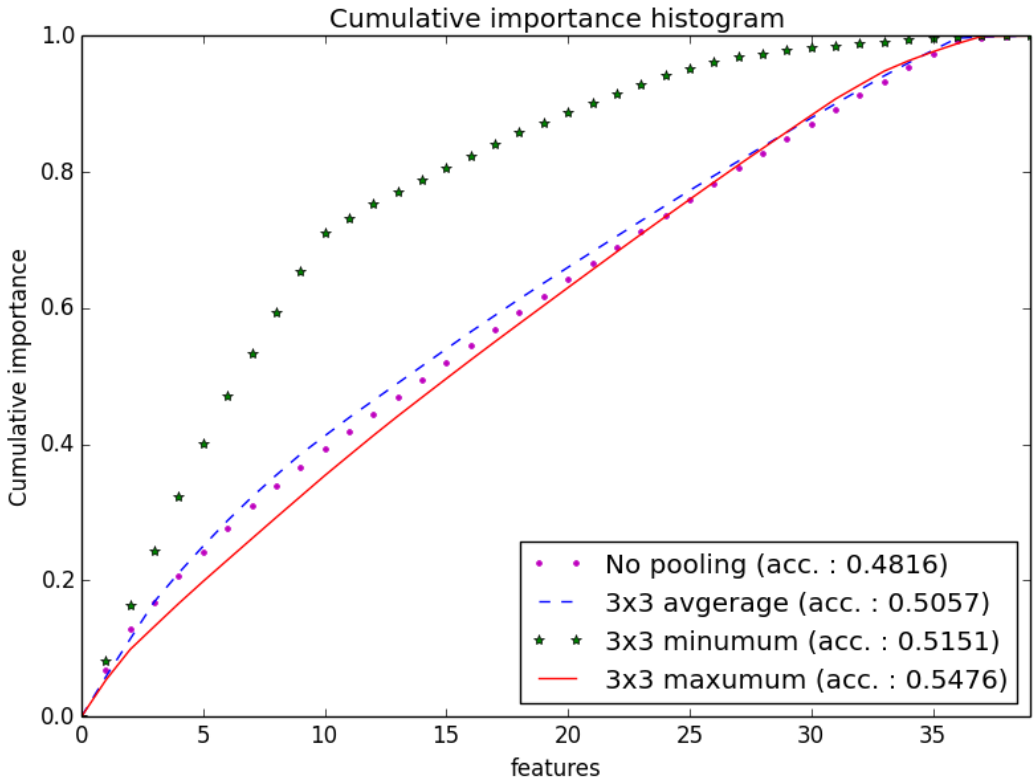


Figure 5.10: Cumulative filter importances with different types of poolings

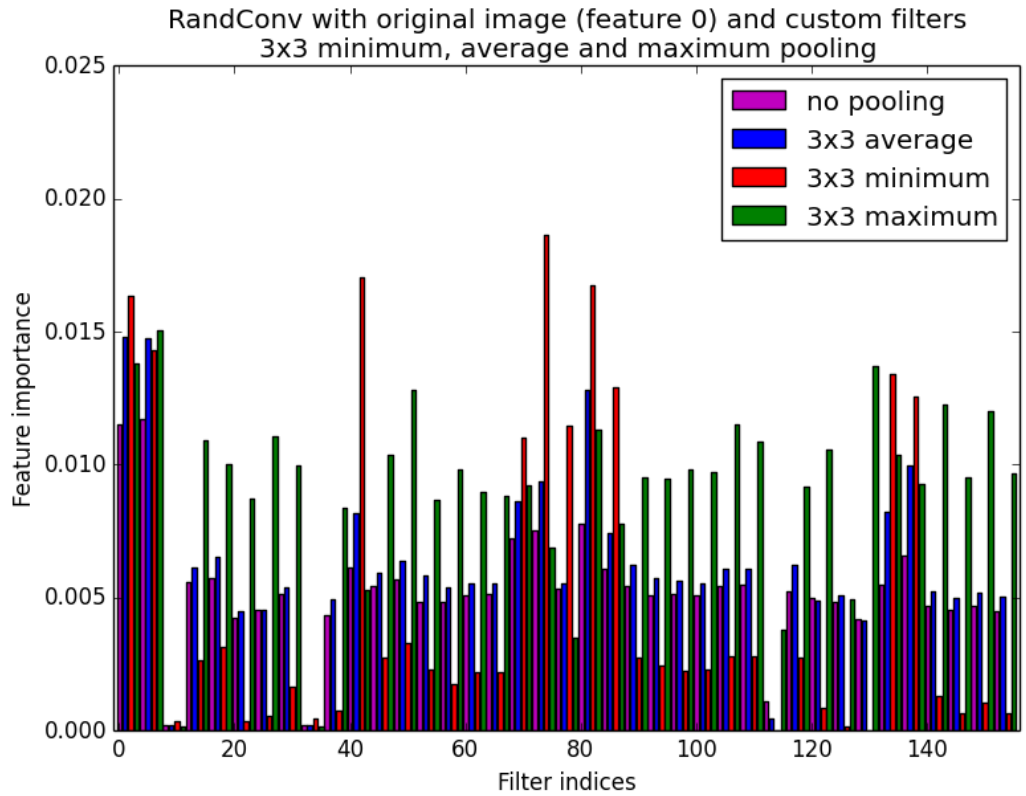
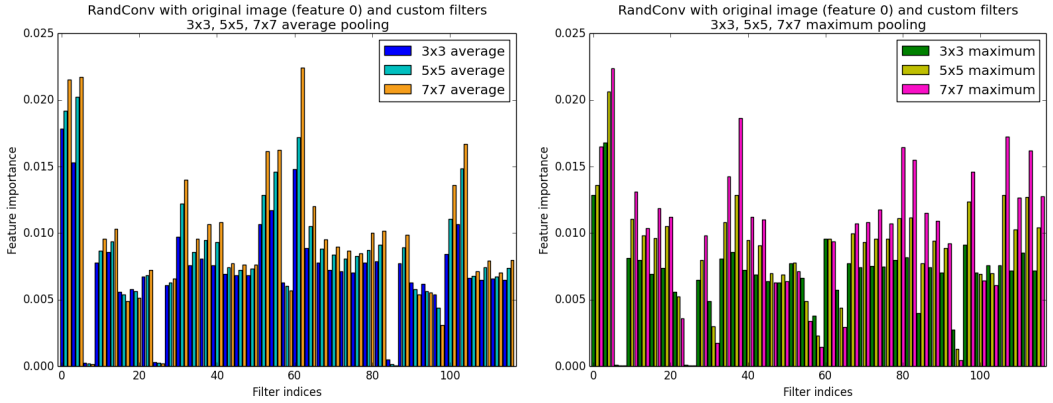


Figure 5.11: Filter importances with several poolings



(a) (b)

Figure 5.12: Filter importances with several pooling scales

Figure 5.12 displays the filter profile for several poolings only differing by the moving window size. Most filters seems to favor larger windows. This is also the case with the minimum pooling. This (non-)locality preference should be problem dependent.

Using a more appropriate spatial pooling is what has brought us the best improvement so far. Since poolings do not interfere with each other, an alternative to picking the best one is simply to use several. In doubt, a combination of the minimum and maximum pooling should do. If the best one is known, however, it might be worth investing the extra free memory elsewhere. We basically have two choices : either increasing the number of filters or the number of subwindows. The latter seems more promising from our recent observations.

5.1.11 Influence of the compression

In this subsection, we tackle the subvector compression. We ran the RandConv method with the default value except for the feature extraction which uses color compression. The accuracy results for keeping only one and keeping two colors per pixel are respectively 0.5041 and 0.5112. In both cases the confusion matrix seems to be in accordance with those of the previous experiments. Other experiments tends to produce the same kind of results : one color per pixel is usually slightly inferior to two colors per pixel. In any case, the influence of the compression is marginal. This can be exploited to reallocate the freed memory to more interesting parameters.

The conclusion on filter importances is as usual : compression does not influence much the filter profiles.

5.1.12 Empirical upper bound on the accuracy

We have now had a tour on the influence of the hyper-parameters. Clearly, the most important is the spatial pooling. Then comes the number of trees, the parameters k and the number of subwindows. Sadly, the number of filters and the filter generator play a less important role in the process. Based on this observation we can now test an optimal set of parameters. The accuracy we will obtain will be subject to some bias because we did not determine this set of parameters on a independent database. However, the bias should be small. Figure 5.1 about the influence of the database size on accuracy suggests that a database of 40,000 images is representative enough. Furthermore, our observation were either confirming our intuitions or too clear to be dependent on the learning samples only. Finally, we will consider this result for what it is : an upper bound on the accuracy.

We chose to use the custom filters with two poolings, a 7x7 minimum and maximum. We impose compression so as to have 2 colors per original pixel. We use the remaining memory to extract 30 subwindows per images. We built 500 trees with a value of k of 500

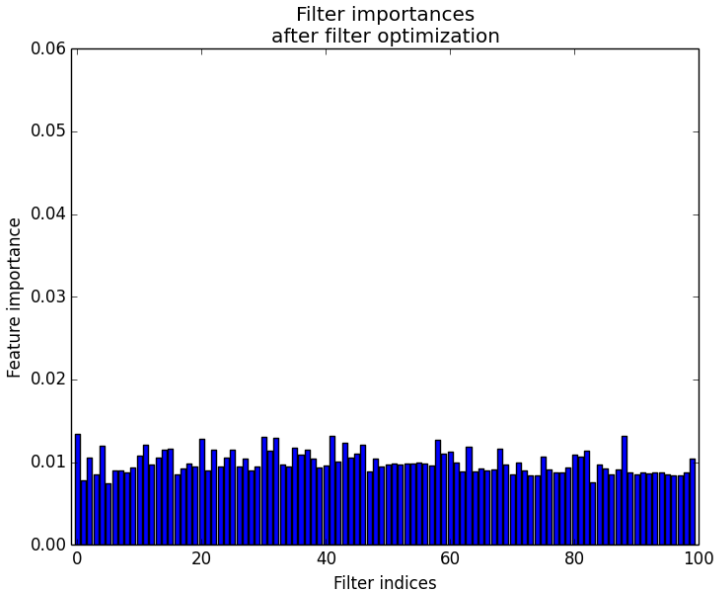


Figure 5.13: Filter importances after optimization of the filter usefulness

as well. The final accuracy scores **0.613**. Of course, this made for a long learning phase (approximately 1 day).

5.1.13 Improvements

We thought of two improvements for the ET-DIC methods.

Useful filter selection The first one was to use the trees to assess the filter usefulness, keep only the best of them and proceeds in an iterative manner to build a stock a "good" filters. This selection can be conducted in an unsupervised way and therefore would not suffer from selection bias. The good news brought by the previous section is that we do not need much resources to assess the filter usefulness : the filter profiles are quite stable and can be assessed with totally randomized trees using only 5 subwindows and compression so as to quicken the process and/or allow for more filters. We need to use the same spatial poolings, however. The very bad news is that the filter generation mechanism do not seem to matter as much as we had originally hoped.

We ran the following experiment : the default classification with a preprocessing for choosing the 100 best filters out of 1000 random ones. The preprocessing is conducted as proposed by totally randomized trees with only 5 subwindows per images and a compression of 1 color per pixel. Figure 5.13 depicts the filter profile. As we can see, we have a very regular profile, much more than what we ever have encountered before. Besides, there is no single useless filter, which is also a first. The accuracy is, however a mild 0.5137 ; not even the best for the default classification. Once again, this highlights the fact that the filters do not play a major role in the process.

Global random subspace The second improvement consists of growing forests independently with different sets of filters, as was mentioned in subsection 4.3.2 about technical issues. The results are aggregated by averaging the class probabilities of a given image. In other words, we vote the class over the several forests. Considering the independence of the filters of two different forests and our discussion about the number of filters, we foresee that the main accuracy gain will actually comes from the augmentation of the total number of trees.

We made our experiment using the following filter generators (unmentioned parameters are left to their default values) :

- ↬ Identity-perturbed generator with a real uniform law on $[-1, 1]$ (accuracy : 0.5128)
- ↬ Identity-distance generator with a maximum distance of 10 and a real uniform law (accuracy : 0.4999)
- ↬ Stratified generator with a subdivision in 10 cells and a Gaussian law of 0 mean and 0.021 standard deviation (accuracy : 0.5063)

The accuracy was of 0.5241, which is indeed more than the individual accuracies. The total number of trees is of 90. From our previous study on the topic, we can say that the major part of the accuracy gain comes indeed from the increase in the number of trees.

We can also add our best result of 0.613 with the custom filters and 500 trees of subsection 5.1.12. However, we need to weight appropriately the model predictions, for example by the number of trees. This yields a slight accuracy gain, rising to 0.6137, whereas we would obtain 0.569 without weights.

5.2 Feature learning scheme

The previous section focused on the ET-DIC pros and cons. We now turn to the ET-FL mode. In this mode, we use totally randomized trees to build a visual dictionary and the actual classification is carried out by a SVM.

This mode is less prone to a deep analysis of the hyper-parameters' influences. The parameter k is fixed to 1, and the number of subwindows, the number of trees and the minimum number of samples to split all influence the number of leaves in a well-understood fashion. An increase in the first two will produce an increase of the number of leaves and ultimately of the discriminant power. The number of subwindows will be severely limited by the memory requirement while the number of trees limitation will be the computational time, although to a lesser extent. A good tradeoff must then be found between the number of subwindows and number of filters and spatial poolings. From the information we gathered in the last section, our expectations are that the number of subwindows will be much more important than the number of filters. The compression can be reused in order to free some extra memory if need be. Since we are not averaging the results any longer, we have lost the ability to reduce overfitting and must resort to pruning in order to limit the complexity. Usually, the optimal pruning should be obtained by cross-validation. However, for our database, a good value of $n_{min} = 500$ has already been established (Marée et al. (2014)).

5.2.1 Accuracy

Chapter 6

Conclusion and perspective

Bibliography

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. (1984). *Classification and regression trees*. CRC press.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

Duan, K.-B. and Keerthi, S. S. (2005). Which is the best multiclass svm method? an empirical study. In *Multiple Classifier Systems*, pages 278–285. Springer.

Geurts, P., Ernst, D., and Wehenkel, L. (2006). Extremely randomized trees. *Machine learning*, 63(1):3–42.

Ho, T. K. (1998). The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844.

Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python.

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep.*

Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *CoRR*, abs/1312.4400.

Marée, R., Geurts, P., and Wehenkel, L. (2006). Biological image classification with random subwindows and extra-trees.

Marée, R., Geurts, P., and Wehenkel, L. (2014). Towards generic image classification. *Submitted*.

Marée, R., Wehenkel, L., and Geurts, P. (2013). Extremely randomized trees and random subwindows for image classification, annotation, and retrieval. In *Decision Forests for Computer Vision and Medical Image Analysis*, pages 125–141. Springer.

Moosmann, F., Nowak, E., and Jurie, F. (2008). Randomized clustering forests for image classification. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(9):1632–1646.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Vapnik, V. (1963). Pattern recognition using generalized portrait method. *Automation and remote control*, 24:774–780.