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

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
5. Describing each of the $N \times M \times P \times S$ pieces by a set of learning features

Thus, from a database of $M$ images, we end up with a feature matrix with $M \times S$ objects described by a number of variables function of $N \times P$.

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.

### 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 substracting 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.

**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 substracting 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 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

$$\begin{aligned} J &= I * F \\ J_1 &= I * F_1 \\ J_2 &= I * F_2 \end{aligned}$$

We will show by counterexample that $\| F - F_1 \| \geq \| F - F_2 \| \not\Longrightarrow \| J - J_1 \| \geq \| J - J_2 \|$. 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 $\| e_1 \| \geq \| e_2 \| \not\Longrightarrow \| I * e_1 \| \geq \| I * e_2 \|$. 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 $y = x - frac1n\mathbf{1}^T x\mathbf{1}$. 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.

### 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.

**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.

**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 ...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 domainfree 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 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 vast. 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.

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 substract 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 $T(r, c) = (rH - r^2 + r) \times (cW - c^2 + c)$.

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 reinterpolation 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 ().

### 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.

Each filtered and spatial 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. 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 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.

#### 4.1.5.1 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. 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.

#### 4.1.5.2 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 totatlly 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 (SVM).

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 developping the tree (pre-pruning) or artifically 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 method 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.

## 4.3 Implementation

In this section, we dive into implementation details and issues. We will not go over all the elements in detail 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. Not all the code is brand new, however. The ExtraTrees implementation comes from the scikit-learn library (Pedregosa et al. (2011)), version 0.15. We also use scikit-learn for SVM classification,

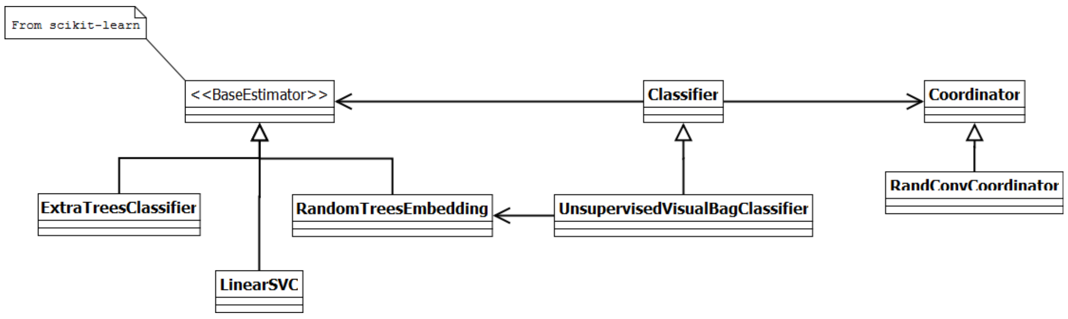

Figure 4.1: UML representation of the `Classifier` class and its major components

although it actually consists of a wrapper to the liblinear library (). As for the subwindow extraction, it is a reorganization of the Pixit implementation (). 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 method. Between the preprocessing and the classification, we use the `RandomTreesEmbedding`, a totally randomized trees implementation, to build the histogram we will fed to the `BaseEstimator`, supposed to be a `LinearSVC` instance. This is summarized by figure 4.1.

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`, the 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 paragraph. 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 to 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. All this is summarized by the figure 4.2.

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

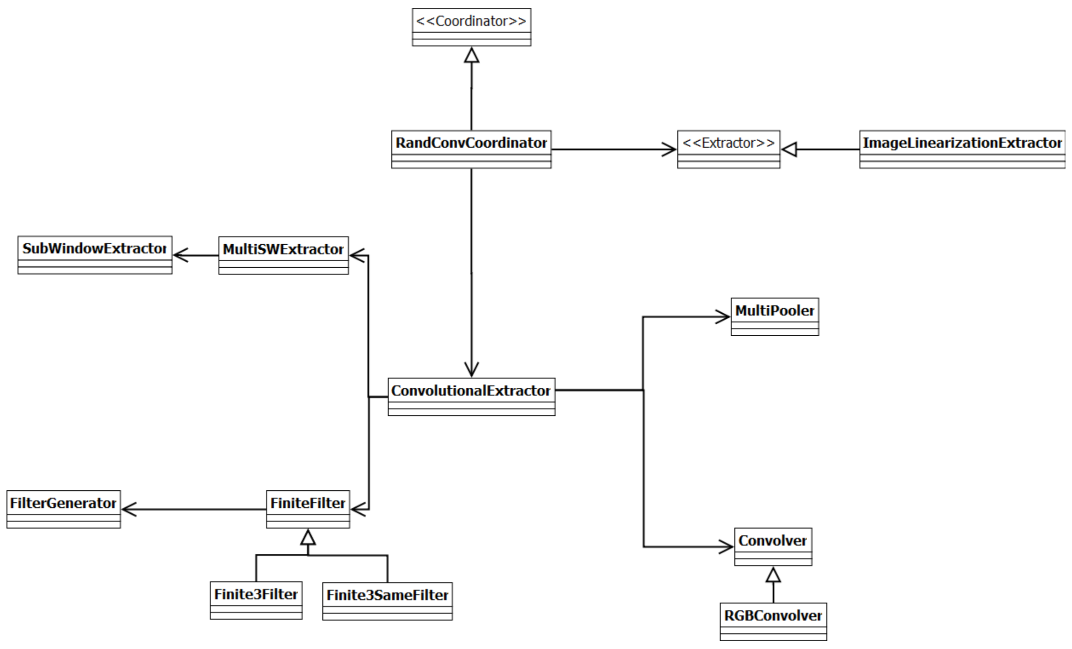

Figure 4.2: UML representation of the `RandConvCoordinator` and its major components

uses the base class of `NumberGenerator`. As figure 4.3 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 4.4 depicts, our two groups of spatial poolings are presents.

### 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 ), 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.

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.

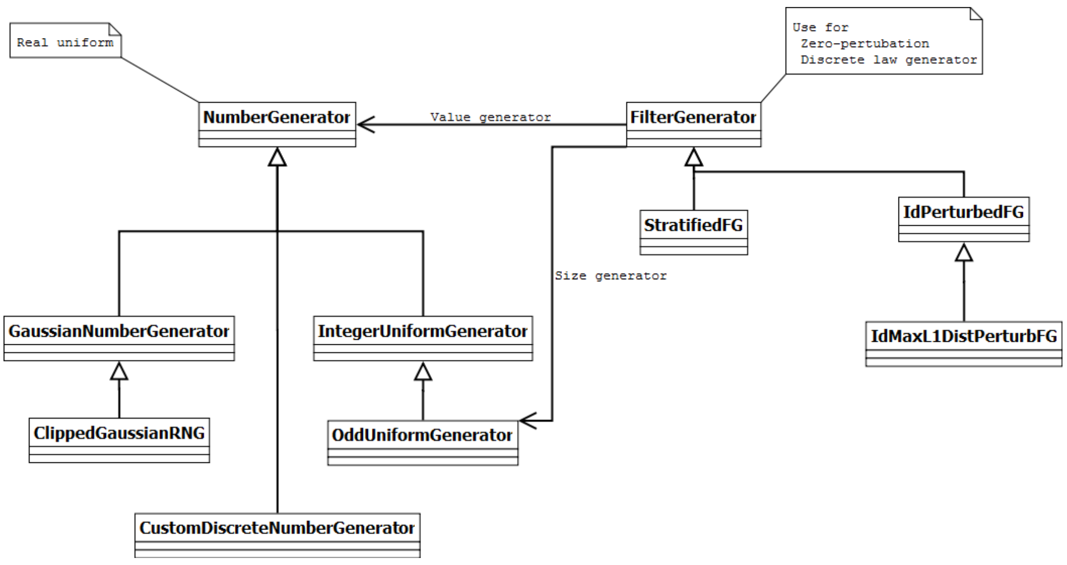

Figure 4.3: UML representation of the `FilterGenerators` and `NumberGenerators`

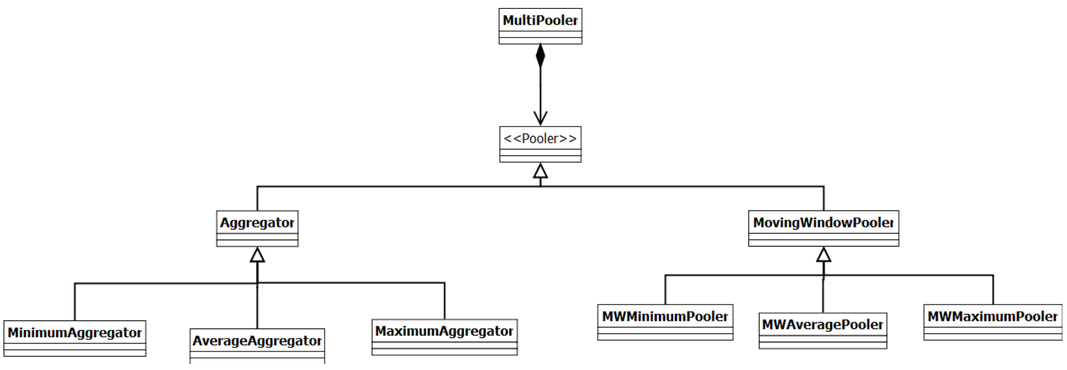

Figure 4.4: UML representaiton of the spatial poolings

## 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
  - 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 : 30)
  - 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.

| | |
|---|---|
| test11 | test12 |
| test21 | 22 |

Table 4.1: Default values for hyper-parameters

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 . 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.

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, 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, while the actual classification is undergone by a support vector machine.

## 5.1 Direct classification scheme

## 5.2 Feature learning scheme

Chapter 6

# Conclusion and perspective

# Bibliography

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

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.

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.

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.