Global and Efficient Self-Similarity

Feature Extractor and Descriptor

1 Introduction

The comprehension of semantics of an image demands in-depth analysis of that particular image. To detect or classify objects in a picture, the image is preprocessed so that one gets a better representation of the information of interest. The two major steps are the extraction of image features, which describe certain characteristics of the image information, and the computation of an image descriptor, which encodes these features in an easy to handle, fixed-size mathematical object; e.g. a matrix or a vector.

Self-similarity is one such characteristic, which depicts similarity of the mathematical object within itself. In image analysis this means re-occurrence of specific patterns in an image over and over again. [Shechtman and Irani, 2007] investigated how self-similarity can be locally computed. That is, self-similarity as a feature is extracted on a patch-level and the geometric arrangement of similar patches are compared thereafter. Figure 1 shows the example of several objects with similar shape. Such similarities are not shared in other image properties, e.g. colors or texture, but can be found by in the similar alignment of locally self-similar patches.

Figure 1: Several with similar geometric arrangement of similar local patterns, from [Shechtman and Irani, 2007].

Extracting self-similar features globally, on the overall image, is computational expensive. [Deselaers and Ferrari, 2010] analyzed how this can be tackled in a feasible way. They depict an algorithm to extract the global self-similarity features (GSS) and define the self-similarity hypercube (SSH) descriptor to describe these features.

This report gives a detailed analysis of Deselaers and Ferrari's work and a re-implementation of their feature extractor and descriptor in C++ using the OpenCV framework. Sections 2 and 3 contain the analysis, Section 4 gives some details on the implementation, and Section 5 shows experiments to find out what this new implementation is capable of.

2 Self-Similarity Features

Extracting self-similarity information of an image is finding all regions within this image that "look alike." A similarity measurement between patches must be defined and an retrieval algorithm on the image must be specified.

[Shechtman and Irani, 2007]'s definition of local self-similarity (LSS) on a patch-level forms the the basis of computing self-similarities. On this basis, [Deselaers and Ferrari, 2010] designed their global algorithm.

2.1 Local Self-Similarity Feature

Taking the patches around two pixels, we can measure their similarity by calculating the sum of squared distances of their pixel values.

Definition 1. SSD. *For two pixels p and p' of the image I, let t_p and $t_{p'}$ denote the $w \times w$-patches centered around p respectively p'. The* ***Sum of Squared Distances*** *(SSD) of patch t_p and $t_{p'}$ is calculated on each pixel value $t_{p,i}$ and $t_{p',i}$ as*

$$\mathrm{SSD}(t_p, t_{p'}) \quad = \quad \sum_i (t_{p,i} - t_{p',i})^2. \tag{1}$$

A low SSD value between two patches indicates strong similarity. However, the correlation of two patches is depicted by a correlation value between 0 and 1. A high value reflects strong correlation, where the maximum value 1 states equality. The correlation of the two patches is calculated on their SSD value.

Definition 2. Correlation. *Let t_p be a patch centered around pixel p and t_x a patch centered around pixel x. The correlation of two t_p and t_x is reflected by a correlation value $\mathcal{C}_p(x) \in (0; 1]$. $\mathcal{C}_p(x)$ is the negative exponent of normalized sum of squared distances; i.e.,*

$$\mathcal{C}_p(x) \quad = \quad \exp\left(-\frac{\mathrm{SSD}(t_p, t_x)}{\sigma}\right). \tag{2}$$

In the original publication of LSS a pixels patch t_p is compared to each others pixels patch t_x within a larger region R_p around p; i.e. $x \in R_p$. All correlation values $\mathcal{C}_p(x)$ of the region R_p together form a correlation surface $\mathcal{C}_p$, which is of the same size as R_p.

Definition 3. Local Self-Similarity Feature. *Let p be an arbitrary pixel, t_p the patch centered around p, and R_p a larger region around p. The Local Self-Similarity Feature is the correlation surface C_p of p to each $x \in R_p$.*

$$\mathcal{C}_p = \mathcal{C}_p(x), \forall x \in R_p. \tag{3}$$

Figure 2: Correlation of two example patches within the global image, from [Deselaers and Ferrari, 2010].

2.2 Global Self-Similarity Feature

The naive way to calculate global self-similarities is to directly apply the LSS feature to not just to a restricted region R_p around a pixel, but to the whole image I. This yield in total $H \times W$ correlation surfaces C_p, one for each pixel $p \in I$. For example, figure 2 shows two patches 1 and 2 and their correlation surface. Brighter pixels in $\mathcal{C}_p$ indicate a stronger correlation of to the particular patches.

Definition 4. Direct Global Self-Similarity Feature. *The direct global self-similarity tensor $\mathcal{S}_I$ of the image I consists of all the correlation surfaces $\mathcal{C}_p$ of each pixel p in the image I.*

$$S_I(p, p') \quad = \quad C_p(p'), \forall p, p' \in I. \tag{4}$$

Restricting the correlation surface to a small fixed-size region R_p around pixel p in the LSS features was to maintain feasibility. But not just the computational cost of the direct GSS tensor is enormous, also $\mathcal{S}_I$ is a 4D matrix that requires memory of size $H \times W \times H \times W$ in terms of the input image size $H \times W$.

The definition of an efficient GSS tensor, which is fast to calculate and which requires just as much memory as the input image itself, is the contribution in Deselaers and Ferrari's work. The idea is, not to correlate each pixel patch t_p to each other patch of I, but only to a small selection of significant prototype patches.

Definition 5. Prototype Codebook. *Let θ be a prototype patch of size $w \times w$ that defines a image basic pattern. The set Θ, which contains all the prototype patches θ, is the **codebook** of the efficient GSS tensor.*

Denote that this definition does not cover the meaning of "significant prototypes." As the detailed specification of how to find the prototypes is not important for the definition of the GSS feature, their description is delayed to section 2.3.

The correlation of each pixel p to the prototypes $\theta \in \Theta$ is calculated just as before by replacing the pixel patches $t_x \in R_p$ with the prototypes $\theta \in \Theta$; i.e., the correlation is calculated as $\mathcal{C}_p(\theta), \forall \theta \in \Theta$. For the assignment of a prototype to a pixel, it is not necessary to keep each of these correlations. In fact, Deselaers and Ferrari required that each pixel must be assigned to exactly one prototype. The assignment is computed with the prototype assignment function $\mathcal{M}_I$.

Definition 6. Prototype Assignment Function. *Let $p \in I$ be a pixel of the input image and Θ a prototype codebook. Function $\mathcal{M}_I : I \to \Theta$ assigns each pixel in I to exactly that prototype in Θ that has the strongest correlation to p.*

$$\mathcal{M}_I(p) \quad = \quad \arg\max_{\theta \in \Theta} \{\mathcal{C}_\theta(p)\}\,. \tag{5}$$

Calculating the prototype assignment for each pixel of the input image will yield a assignment map $\mathcal{M}_I$. It has the same size as the input image.

Definition 7. Efficient Global Self-Similarity Feature. *Given image I, the prototype assignment map $\mathcal{M}_I$ is defined by*

$$\mathcal{M}_I \quad = \quad \mathcal{M}_I(p),\, \forall p \in I. \tag{6}$$

This defines the ***efficient global self-similarity feature****.*

Example 1. Figure 3 shows an image of a few pyramids from the Caltech 101 database [Fei-Fei et al., 2004] and an prototype assignment map of that image. For each pixel in I a $w \times w$ sized patch has been extracted and the set of patches has been clustered with k-means into 400 distinct prototypes. This yields a so-called image specific prototype codebook, as it will be explained in the next section. Figure 4 contains eight example prototypes of this codebook. Then, each pixel p in the image is assigned to the closest cluster regarding SSD. That is, the prototype θ for which the correlation $\mathcal{C}_\theta(p)$ is maximal. The cluster assignments itself are just numbers in the assignment map; to display them, they have been mapped to HSV color values using the OpenCV function.

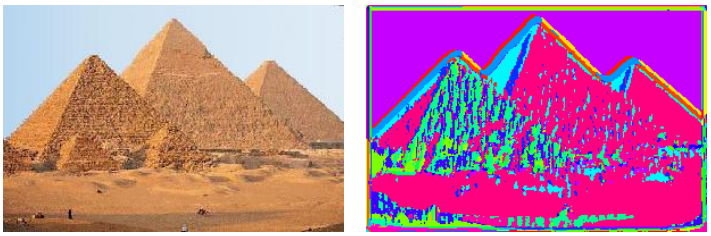


Figure 3: Example image I of pyramids and its prototype assignment map.

Figure 4: Eight example prototypes from the codebook of image I.

2.3 Discussion on Significant Prototypes

As denoted, the Definition 5 neither specifies the meaning of "significant prototypes" nor how the codebook Θ is ought to be collected. Deselaers and Ferrari present three different types of codebooks: the generic codebook Θ_{DCT}, an database specific codebook Θ_{DB}, and the image specific codebook Θ_I. See figure 5 for examples.

Generic Codebook. For this codebook, the discrete cosine transformation (DCT) is computed on an image patch. It disassembles the patches in terms of 2D cosine functions with different basis frequencies. Each of the basis frequencies can be interpreted as a prototype. The respective prototype patch contains discretized values of the 2D basis function as pixel values. An image patch is the linear combination of multiple basis functions, i.e. prototype patches. When applying the DCT for the prototype assignment, then the patch is assigned to the basis function prototype with the highest coefficient.

In practice the codebook Θ_{DCT} is not collected, but the DCT is applied to the image patches directly. It is applied to each of the patches color channels independently and then the assignment $\mathcal{M}_I(p)$ is "the composition of the prototypes of the three channels," [Deselaers and Ferrari, 2010][1]. Furthermore, for practical reasons the constant frequency prototype is discarded.

Database-specific Codebook. The database or image set specific codebook is generated by collecting all pixel patches from the whole set of images and then selecting the $k = |\Theta|$ "most significant" ones. Deselaers and Ferrari propose to apply k-means clustering on all patches and choose the cluster centers as prototypes. However, this collection of patches might become very large and on the same time numerous patches are alike as the patches are extracted for each pixel of the images. The proposed compromise is to randomly sample a subset of all patches and then cluster the prototypes from this subset in a shorter time.

Image-specific Codebook. The generation of the image specific codebook follows the same procedure as the generation of the data specific prototypes, besides that prototypes for each assignment map $\mathcal{M}_I$ are cluster only from the single image I.

Figure 5: Examples of Θ_{CDT}, Θ_{DB}, and Θ_I codebooks, from [Deselaers and Ferrari, 2010].

[1] We interpret "composition" as: We are to apply the DCT on each color channel independently and to select the prototype assignment of the color channel with the highest response. We took a few samples of patches and discovered that in all cases the prototype assignment of each channel were the same.

3 Self-Similarity Descriptors

Once the features are computed they must be converted into a comparable fixed-size form, the descriptor. It contains the important information about the image features.

3.1 Local Self-Similarity Descriptor

To describe the correlation surface $\mathcal{C}_p$ in a fixed-size descriptor, Shechtman and Irani proposed to overlay the surface with an log-polar grid that centered around p. Each bin of the grid is assigned to the maximal value of the corresponding grid cell in the correlation surface. Figure 6 shows an example of the LSS descriptors on 3 different patches.

Definition 8. Local Self-Similarity Descriptor. *The descriptor $\mathcal{L}_p$ of the correlation surface $\mathcal{C}_p$ is calculated by overlaying $\mathcal{C}_p$ with a (ρ, θ)-log-polar grid, where ρ are radial and θ angular coordinates for the bins. For each bin $x \in BIN(\rho, \theta)$ the maximum correlation value of $\mathcal{C}_p$ is assigned to $\mathcal{L}_p$; i.e.,*

$$\mathcal{L}_p(\rho, \theta) = \max_{x \in BIN(\rho,\theta)} \{C_p(x)\}. \tag{7}$$

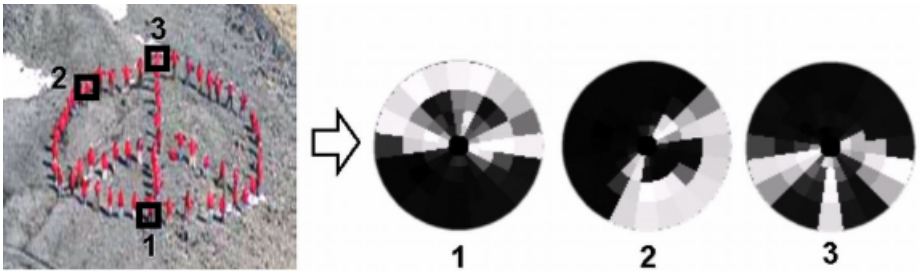


Figure 6: Example LSS descriptor, from [Shechtman and Irani, 2007].

3.2 Efficient Global Self-Similarity Descriptor

Deselaers and Ferrari adopted the idea of using an overlay grid for their GSS descriptor and partitioned the assignment map $\mathcal{M}_I$ into $D_1 \times D_2$ grid cells. However, an assignment map does not contain any correlation values of which they could take the maximum. Instead, to find out how similar two grid cells are, they count the sum of how many pixels in the one cell have the same assignment as the pixels in the other cells using the function $\mathcal{H}_I$.

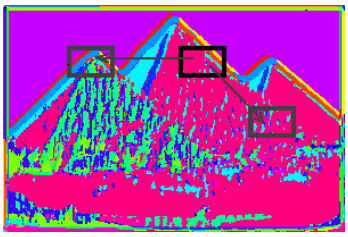
Definition 9. Similarity of Grid Cells. *Let J and J' be two overlay grid cells of the assignment map $\mathcal{M}_I$. The* ***similarity of the two grid cells*** *is measured by the number pixels that share the same prototype assignments:*

$$\mathcal{H}_I(J, J') = \sum_{p \in J} \sum_{p' \in J'} \delta(\mathcal{M}_I(p) = \mathcal{M}_I(p')). \tag{8}$$

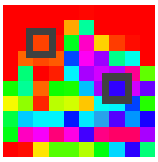
Comparing each of the $D_1 \times D_2$ grid cells with, again, each of the $D_1 \times D_2$ grid cells results in a 4D self-similarity hypercube (SSH) descriptor $\mathcal{H}_I$ for the image I.

Definition 10. Global Self-Similarity Hypercube. *The self-similarity hypercube $\mathcal{H}_I$, which contains the similarity values of each of the grid cell pairs (J, J'), is the descriptor of the efficient global self-similarity features $\mathcal{M}_I$.*

Example 2. We resume with example 1 and show how the self-similarity hypercube descriptor is created on the pyramid's assignment map. For this example, let $D_1 = 10$ and $D_2 = 10$. In figure 7a the assignment map is shown with the exemplary grid cell J, which is marked by the black frame, and two other arbitrarily selected cells K' and K'' are framed. For each cell K in the grid the similarity value $\mathcal{H}_I(J, K)$ is computed. All those values $\mathcal{H}_I(J, \cdot)$ are stored in a, for J specific, slice of the hypercube. Figure 7b shows the slice for the exemplary cell J, its values are mapped into HSV colors space, too. In this slice, the two values $\mathcal{H}_I(J, K')$ and $\mathcal{H}_I(J, K'')$ are marked by a gray frame. The correlation of cell J to the left cell K' is rather low. That is indicated by the red-orange color. The correlation of J to the lower right cell K'' is far stronger, indicated by the blue color. In general, the red and orange colors in HSV space map to correlation values that are close to zero; the blue and purple colors map to values close to 1. By that, one can see that the selected cell J strongly correlates to all those cell that show the front of the middle pyramid and those that show much desert sand, but not to those cells that show the sky.



(a) Framed cell J is compared to two other cells K' and K''.



(b) Hypercube slice $\mathcal{H}_I(J, \cdot)$ with frames around the values for $\mathcal{H}_I(J, K')$ and $\mathcal{H}_I(J, K'')$.

Figure 7: Assignment map $\mathcal{M}_I$ with an example overlay cells and the slice in the GSS descriptor for the selected cell.

Such slice must be calculated for any cell in the grid, which yields the hypercube descriptor. Figure 8 depicts the overall hypercube, where the slices for all cells are colored and placed next to each other. The previously denoted slice for cell J in figure 7b is marked a black frame in the overall hypercube. In addition, two further examples of cells and their slices can be found in figure 17 in the Appendix A.

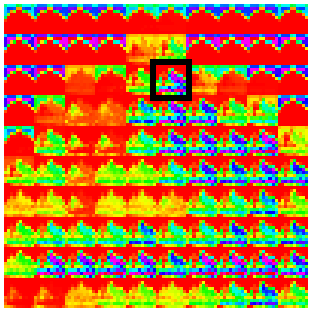


Figure 8: Complete GSS descriptor $\mathcal{H}_I$ for the pyramids.

4 Implementation

The C++ implementation of the efficient global self-similarity feature and descriptor follows the design of the Matlab code given by Deselaers and Ferrari[2]. The class `FastSelfSimilarity` encapsulates the creation of the prototype codebooks Θ, the generation of the GSS assignment map $\mathcal{M}_I$, and calculations for the hypercube descriptors $\mathcal{H}_I$.

4.1 The `FastSelfSimilarity` Object

Constructor. The object constructor takes the following parameters:

- `n_clusters`, the number overall of clusters $k = |\Theta|$, where 400 clusters is the default.
- `method`, identifies the way how the prototypes are to be clustered. It is:

  0: if $k-$means should be applied on all patches,

  1: if the patches that are used in the $k-$means clustering are just a random subset of all patches,

  2: if the prototypes are gathered only by sampling; i.e., no clustering,

  3: if the DCT prototypes are to be used; i.e., no clustering.
- `n_patches`, which gives the maximal number of patches that are sampled before they are clustered. This is only relevant for `method=1`.
- `d1`, the number of overlay grid cell rows (height) to be used for the GSS hypercube.
- `d2`, the number of overlay grid cell columns (width).

The `FastSelfSimilarity` object provides the following functionality:

[2] `http://calvin.inf.ed.ac.uk/software/global-and-efficient-self-similarity/`

`imagePatcher`. This function takes an image as input and then gathers all patches from this image. The extracted patches are of size $w \times w$, where $w = 2 \cdot \texttt{scale} + 1$, and they are centered around the pixels $p \in I$.

`getClustersFromPixels`. Given image or an image set, this function gathers their patches using `imagePatcher` and then finds the set of prototypes Θ according to the selected clustering `method`. This function uses OpenCV's $k-$means implementation, if method was set to 0 or 1.

`quantiseSSD`. Given the set of prototypes has been found, this function takes a vector of patches of image I and computes the prototype assignment map $\mathcal{M}_I$ according to the minimal sum of squared distances between the patches and the prototypes.

`quantiseDCT`. To find the assignment map that is based on the general prototypes, it is not necessary to build the clusters of prototype patches. The DCT is directly applied on each pixel patch and the prototype assignment map $\mathcal{M}_I$ is created by choosing the DCT base frequency with the maximal absolute coefficient. Denote, the constant prototype, which has no zero, is omitted.

`quantise`. As a generalization, this function delegates calls to either `quantiseSSD` or `quantiseDCT` according to the `method` value.

`getHistogram`. It returns the histogram of prototype assignments of an assignment map or a display window of it.

`getOneSSH`. The self-similarity hypercube $\mathcal{H}_I$ is computed for a region of interest (ROI) of the assignment map $\mathcal{M}_I$. The correlation of each grid cell of the $D_1 \times D_2$ overlay grid is calculated by comparing the number of equal prototype assignments in each two grid cells. At first, a histogram vector of prototype assignments of each cell is created with the `getHistorgram` function. Then, the number of equal prototypes of two cells is the dot product of their histograms.

4.2 Example Algorithm

The algorithm 1 depicts how the `FastSelfSimilarity` object can be used to create a set of hypercube descriptors for a set of input images.

At first, it needs clustering cluster the images into the prototype set Θ. Therefore, `getClustersFromPixels` takes the whole set of input images and generates the database-specific prototype set. Calling `getClustersFromPixels` within each iteration of the `for`-loop creates the image-specific prototypes. Figure 9 shows an example image and 8 randomly selected cluster patches of the codebook. The codebook itself is stored and maintained internally in the `fss` object and shall not be accessed from the outside.

The prototype assignment maps $\mathcal{M}_I$ are computed on the set of pixel patches of the image I, which are collected with the `imagePatcher` function. `quantise` then forwards the call to `quantiseSSD` and returns the database-specific assignment. If the `fss` object

Algorithm 1 Using the `FastSelfSimilarity` object.

```
std::vector<cv::Mat> images = loadImages();
std::vector<cv::Mat> descriptors;

// Create the fast self-similarity object.
int method = 2;
FastSelfSimilarity fss(n_clusters, method, n_patches, d1, d2);

// Gather the protoypes from the image set.
fss.getClustersFromPixels(images);

// For each image ...
for (cv::Mat image_i : images) {

        // gather the patches of image i, ...
        std::vector<cv::Mat> patches_i = fss.imagePatcher(image_i);

        // create the protoype assignment map M_i, ...
        cv::Mat assignment_i = fss.quantise(patches_i,hight,width);

        // define a target region in the assignment map, ...
        cv::Rect roi = getRegionOfInterest();

        // and compute the descriptors for image i in the ROI.
        cv::Mat SSH_i = fss.getOneSSH(assignment_i, roi);

        // Save the SSH descriptor for further processing.
        descriptors.push_back(SSH_i);
}
```

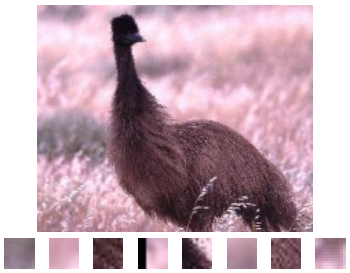


Figure 9: Example of an input image I (Caltech 101) and 8 random clusters.

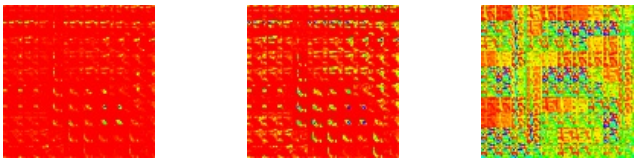
had been created with `method=3`, `quantise` would forward the call to `quantiseDCT` instead. In this case, the call `to getClustersFromPixels` would not do anything, but exit early. Figure 10 depicts three assignment maps, which are generated based on the three different types of prototypes. For the Θ_{DB}-example assignment map we have taken all emu images of the Caltech 101 database for the clustering.

The SSH descriptors can be calculated on any specific region of the assignment map. This is important for object localization and it is used in classification of the display

(a) Θ_I prototypes. (b) Θ_{DB} prototypes. (c) Θ_{DCT} prototypes.

Figure 10: Assignment maps $\mathcal{M}_I$ from fig. 9, created with the different prototype sets.

windows in section 5.3. Calling `getOneSSH` without a region of interest results in calculating a hypercube for the whole assignment map. To illustrate the image descriptors, each slice of the 4D self-similarity hypercube in figure 11 is colored and placed side by side.



(a) Θ_I prototypes. (b) Θ_{DB} prototypes. (c) Θ_{DCT} prototypes.

Figure 11: SSH tensors $\mathcal{S}_I$ of the assignment maps in fig. 10.

5 Experiments

The implementation is analyzed on the Caltech 101 image benchmark [Fei-Fei et al., 2004]. To find out, what it is capable of, we will first evaluate it on image classification; once with images of emus and flamingos and once with emus and cars. A training set, which contains 25% of the images of both categories, is taken to train a linear SVM. The remaining images are used for validation. The second test is to show the robustness of the GSS descriptors to scale and shift changes. We validate different display windows gathered with a sliding window on three emu images. The SVM is then trained with the remaining images of either of the two categories.

In the following, section 5.1 contains a more detailed description of the image categories, section 5.2 the results of the classification of the categories, and section 5.3 depicts the analysis with the sliding display windows.

5.1 Image Categories

The Caltech 101 benchmark dataset is a collection of images gathered in the Internet and sorted into 101 different categories. It contains images of objects and animals like butterflies, cars, laptops, pigeon, and the like. However, in our analysis we have chosen only three of these categories that are the emus, flamingos, and cars categories. The category emu contain 53 colored images, in the flamingo's are 67 colored images, and there are 123 gray scale images of side views on cars. The restriction on the number of only three categories, which have 243 images in total, was made due to the long runtime of the experiments, as you will see in the following sections. The selection of those three categories was made, because the discrimination of emus against flamingos was assumed to be a rather complicated test and the discrimination of emus and cars a rather easy experiment. We would like to see, how the descriptors work with these two different settings. In figure 12, three example images of each of the selected category are shown.

Figure 12: Three images of the Caltech 101 categories emus, flamingos and cars.

5.2 Classification of Categories

The first test is object classification. As denoted, these tests are evaluated once on the image sets 1, emu vs flamingo, and once on set 2 that contains emus and cars. In every test case, we use the OpenCV implementation of a linear SVM and take every 4^{th} input image to train the SVM. The remaining images are used for verification.

Variable Parameters. For the clustering we always set `method=1`, which is sampling at most $n_{patches}$ patches and then clustering that sampled set with $k-$means into $k = |\Theta|$ prototypes. We choose three different parameter sets to run the clustering, which are parameters 1: $n_{patches} = 10,000$ and $k = 400$, parameters 2: $n_{patches} = 50,000$ and $k = 300$, and at last parameters 3: $n_{patches} = 500,000$ and $k = 1,000$. Due to the limits of our memory, we can not cluster from more than $500,000$ patches, which also means that we can not run the tests with `method=0`. Denote, that the parameters no. 3 are only used with the Θ_{DB} codebooks, since there are no more than $50,000$ pixels in a single image, which is relevant for Θ_I. Also, none of the cluster parameters influence the tests with the generic prototypes Θ_{DCT}, since there is no clustering for the DCT codebook.

Image Set 1	Images	Verified on	Errors with Θ_I		Errors with Θ_{DB}			with Θ_{DCT}
			pars. 1	pars. 2	pars. 1	pars. 2	pars. 3	
Emu	53	39	20	27	28	23	27	20
Flamingo	67	51	28	26	14	19	15	24
$\sum$	120	90	48 (53.3%)	53 (58.9%)	42 (46.7%)	42 (46.7%)	42 (46.7%)	44 (48.9%)

Table 1: Classification Emus vs Flamingos.

Image Set 2	Images	Verified on	Errors with Θ_I		Errors with Θ_{DB}			with Θ_{DCT}
			pars. 1	pars. 2	pars. 1	pars. 2	pars. 3	
Emu	53	39	8	7	18	16	19	5
Car	123	93	5	7	2	4	1	11
$\sum$	176	132	13 (9.8%)	14 (10.6%)	20 (15.2%)	20 (15.2%)	20 (15.2%)	16 (12.1%)

Table 2: Classification Emus vs Cars.

Results. Tables 1 and 2 show the results of the tests. The average error on the image set 1 is 50.20%, which is basically the result of a random guess. The results with Θ_I codebook and parameters 2 yield the greatest distance to the random chance, though they also have the most misclassified images. We do not conclude any advantages or disadvantages from this, but only that the chosen categories are hard to separate with these self-similarity features for any codebook.

On image set 2 the average error is 13.02%. Using the Θ_I codebook yield the lowest error, Θ_{DB} the worst, and Θ_{DCT} performed slightly better than average. This test shows slightly superiority of Θ_I prototypes over the others. Our results with the DCT codebook do not coincide with Deselaers and Ferrari's findings, which is that Θ_{DCT} performs poorly. However, our implementation might differ from theirs.[3] In addition, we find that the errors are more or less stable with the different cluster parameters $n_{patches}$ and k. The variance of the error might stem from the randomness of the sampling. Only the the results with the Θ_{DCT} codebook are stable, as they do not incorporate any randomness.

Runtime. The tests ran on a 2006's Intel Celeron 575 processor; the runtime of the experiments is listed in table 3. The clustering and the quantization takes most of the time. Especially with the image-specific codebook, as it needs clustering in each iteration. In contrast, the database-specific codebook runs $k-$means just once. This is an advantage, as higher expenditures on clustering does not even guarantee better results. The quick DCT alternative does not needs either clustering or quantization and its runtime performance depicts this.

[3]The DCT codebooks are not implemented in the provided Matlab code. In their experiments section, it reads as if they do not apply DCT directly, but only on a very few (5 or 10) DCT-like patches. Also their "composition" of the color channel prototypes might different from ours.

The classification results do not indicate that there is any advantage in taking a huge set of pixel patches to cluster nor do they advocate a large number of clusters. The Θ_{DB} codebook yields an almost constant error rate. In stead of running the experiments with parameters 2 and 3, we could have gotten better results with repeating only the experiments with parameters no. 1 multiple times, or even using smaller number of clusters and patches.

The time it takes to create the SSH descriptor is negligible compared to the quantization and clustering. Once a good set of clusters have been found and the assignment maps are computed the assignment maps could be stored with the images, if computing the descriptors needs to be done over and over again.

	Time on Θ_I		Time on Θ_{DB}			Time on Θ_{DCT}
	pars. 1	pars. 2	pars. 1	pars. 2	pars.3	
a) Emu vs Flamingo (120 images):						
$\sum$ Clustering	2.47 h	5.76 h	78.10 s	4.64 min	2.92 h	-
$\sum$ Quantization $\mathcal{M}_I$	1.51 h	1.13 h	1.50 h	1.13 h	4.09 h	2.48 min
$\sum$ Computing $\mathcal{H}_I$	2.93 s	2.31 s	2.92 s	2.30 s	6.77 s	1.77 s
b) Emu vs Cars (176 images):						
$\sum$ Clustering	3.62 h	8.26 h	79.68 s	5.72 min	2.7 h	-
$\sum$ Quantization $\mathcal{M}_I$	2.00 h	1.55 h	2.00 h	1.58 h	5.1 h	3.48 min
$\sum$ Computing $\mathcal{H}_I$	4.25 s	3.48 s	4.28 s	3.51 s	9.7 s	2.53 s
c) average time per image:						
$\sum$ Clustering	74.09 s	2.84 min	-	-	-	-
$\sum$ Quantization $\mathcal{M}_I$	42.59 s	32.57 s	42.56 s	32.87 s	1.46 min	1.17 s
$\sum$ Computing $\mathcal{H}_I$	24.24 ms	19.53 ms	24.32 ms	19.64 ms	2.05 min	14.52 ms

Table 3: Runtime of the classification a) Emu vs Flamingo, b) Emu vs Car, and c) image average.

5.3 Classification of Display Windows

The second test is to find how the GSS descriptor performs when the input image is shifted and scaled. We take some images for verification, extract display windows following the sliding window approach, and verify them using a linear SVM. As before, the evaluation image sets are emus vs flamingos and emus vs cars. The set of verification images contains only the first three images of the emu category. The SVM was trained each with the remaining images of the datasets. However, the results on both images sets are exactly the same, so the sets are not discriminated in the reported results.

Results. In the following, the tables list the numbers of detected emus and misclassified display windows for each image and on each type of prototype assignment maps. For each depth level the size of the respective region of interest (ROI) is given, as well as the number of correct classified and misclassified regions.

Image no. 1 (table 4) was split into 164 display windows. In most of them, the emu is detected when evaluating the windows on the Θ_{DB} assignment maps, but misclassified in the evaluations on the Θ_{DCT} and Θ_I maps. The tests with the Θ_I and Θ_{DCT} return a clear result only on the first two levels. Below that the results become rather heterogeneous. For the test with Θ_{DB} this is also true on depth level 1 and below.

Depth Level	ROI Size	Θ_I		Θ_{DB}		Θ_{DCT}	
		#Emus	#Errors	#Emus	#Errors	#Emus	#Errors
0	200×165	0	1	1	0	1	0
1	180×148	0	6	3	3	5	1
2	162×133	6	10	5	11	10	6
3	145×119	14	16	17	13	9	21
4	130×107	23	25	38	10	10	38
5	117×96	29	34	52	11	7	56
$\sum$	-	72	92	116	48	42	122

Table 4: Results of the emu 1 image.

The results on the second image (table 5) are very clear for the database specific codebook. This is in contrast to the results of the generic and the image specific codebooks, where the test failed to detect the emu in the whole image and it becomes rather diverse on the lower depth levels just as in image 1.

Depth Level	ROI Size	Θ_I		Θ_{DB}		Θ_{DCT}	
		#Emus	#Errors	#Emus	#Errors	#Emus	#Errors
0	200×147	0	1	1	0	0	1
1	180×132	3	3	6	0	3	3
2	162×118	3	9	12	0	3	9
3	145×106	8	22	30	0	9	21
4	130×95	17	31	48	0	18	30
5	117×85	24	39	62	1	21	42
$\sum$	-	55	105	159	1	54	106

Table 5: Results of the emu 2 image.

At last, the results for the third image in (table 6). Unlike the images 1 and 2, the image 3 was always misclassified in the classification tests of section 5.2 on the database specific assignment maps. By that, it can be explained that the validation is failing on the Θ_{DB} assignment maps. Other than the different classification, the results are similar to the ones on image 2.

Depth Level	ROI Size	Θ_I		Θ_{DB}		Θ_{DCT}	
		#Emus	#Errors	#Emus	#Errors	#Emus	#Errors
0	181×200	0	1	0	1	0	1
1	162×180	3	3	0	6	2	4
2	145×162	8	8	0	16	10	6
3	130×145	16	20	1	35	23	13
4	117×130	24	32	7	49	37	19
5	105×117	30	42	15	57	51	21
$\sum$	-	81	106	23	164	123	64

Table 6: Results of the emu 3 image.

Discussion. To explain the heterogeneousness of the results, we give some insights on the image windows and their descriptors on the exemplar image of Emu 3. Figure 13 depicts the classification result of the display windows at depth level 2, which were evaluated on the generic prototype assignment map. This figure shows only the first two rows of display windows, the remaining windows can be found in the Appendix B. The frames show the different windows from which the SSH descriptor was created; their color indicate the classification result. That is, a green frame is classified as showing an emu," the red frame is misclassified. However, the location of the displayed frames seem not to give a clear relation to their classification.

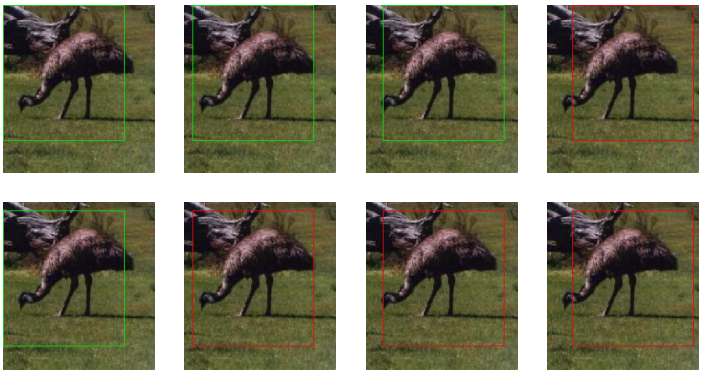


Figure 13: Emu 3, sliding windows at depth level depth 2 (starting at level 0) and the classification of the generic prototype assignment map.

The results in section 5.2 indicate that the discrimination of emus from flamingos is rather complicated and the discrimination of emus from cars must be easier, but this is not reflected by this sections results. The following descriptors where created on three different images, one for each of the categories, to understand these results. Figure 14 shows one image specific and one generic assignment map for each image. Observe the descriptors of the overall assignment map and the ones extracted with the display window at level 2 of the image pyramid, to spot changes in the descriptors when the

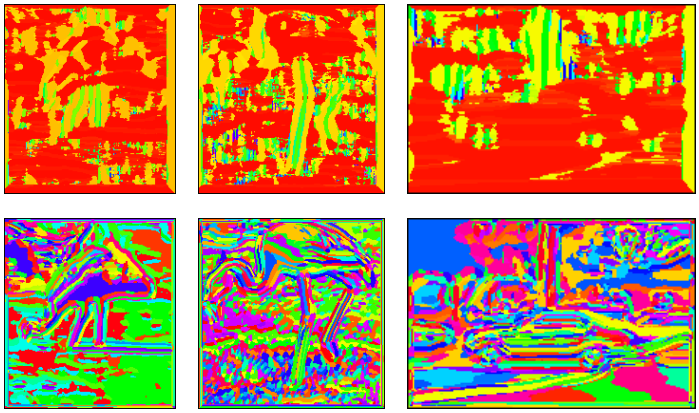


Figure 14: Generic and image specific assignment maps of an emu, a flamingo, and a car.

display window gets scaled or shifted. At the 2nd depth level, the window has 0.81 times the size of the original image and it is moved by 10 pixels in each step. See figure 15 for the descriptors of the previous assignment maps. The descriptors from the display windows at level 2 are depicted in figure 16; to maintain clarity it contains only the top row of sifted descriptors. Shifting the display window affects the descriptors, as their overlay grid shifts with the window. This leads to a shift of the rows or columns in the hypercube. This moves the representation in the vector space of the SVM and leads to such diverse classification results. To re-establish the link between the window descriptors and the target object description, the shifted descriptors need further correlating before running a classification test.

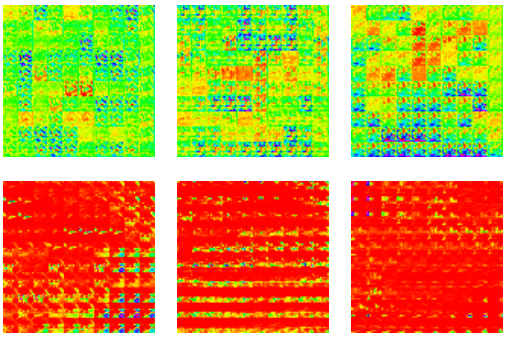
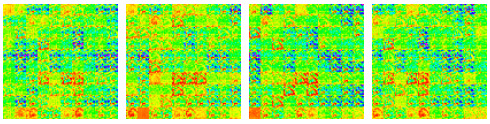
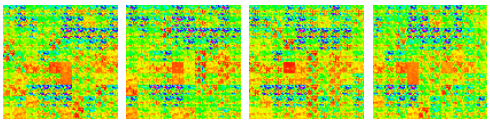


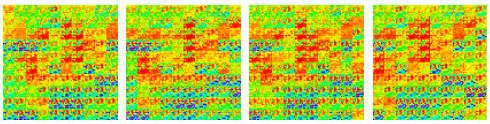
Figure 15: Hypercube descriptors of the assignment maps in figure 14.



(a) 1st row level 2 sliding window of the emu.



(b) 1st row level 2 sliding window of the flamingo.



(c) 1st row level 2 sliding window of the flamingo.

Figure 16: SSH descriptors on the first row of sliding windows at depth level 2 on the generic assignment maps of figure 14.

Besides the effects that occur when the window is shifted, the descriptors also change when we scale into the image. The overlay grid on the assignment map scales with the scaling of the display window and so does the resolution of the descriptors. In the given figures, one can find similar structures of descriptors on the lower levels in the descriptors of the overall image. However, these relations are more vague than those between shifted descriptors on the same level.

These results indicate that the hypercube descriptors alone are not robust to scale and shift changes. A detection of an object in the overall image needs subsequent correlation of the window descriptor to the descriptor of the target object.

6 Conclusion

This report gives a survey of Deselaers and Ferrari's global self-similarity features and descriptors, a re-implementation of these using C++ and OpenCV, and some basic experiments on applying these features in image classification.

The results on the object recognition task are not outstanding. A discrimination of the categories emus and flamingos of the Caltech 101 database failed. Also the tests did not indicate clear superiority of any particular codebook over the other codebooks, as it has been the case in Deselaers and Ferrari's experiments. In addition, the computation of the features and descriptors take quite long in general. At last, it shows that the descriptor is not quite robust to view changes, that is scale or shift of the view onto the object. Slightly changes might change the description of the target completely.

To deal with these problems, one might pick a more specific representation of the target object or class, as the self-similarity of an image also contains the similarity of

the background within this image. Furthermore, a subsequent correlation of the target description to the description of an image or image section is needed to re-detect the object in the image.

In their publication, Deselaers and Ferrari also reported on the possibility of enhancing other image features by the use of self-similarity features. This has not been tested in this work. Also further experiments on different image sets and a comparison to state-of-the-art features are in need to get a better impression on the potential of these features.

References

Thomas Deselaers and Vittorio Ferrari. Global and efficient self-similarity for object classification and detection. *IEEE CVPR 2010*, CVPR 2010, 2010.

L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. *IEEE CVPR 2004, Workshop on Generative-Model Based Vision*, Workshop on Generative-Model Based Vision, 2004.

Eli Shechtman and Michal Irani. Matching local self-similarities across images and videos. *IEEE CVPR 2007*, 2007.

Appendix A

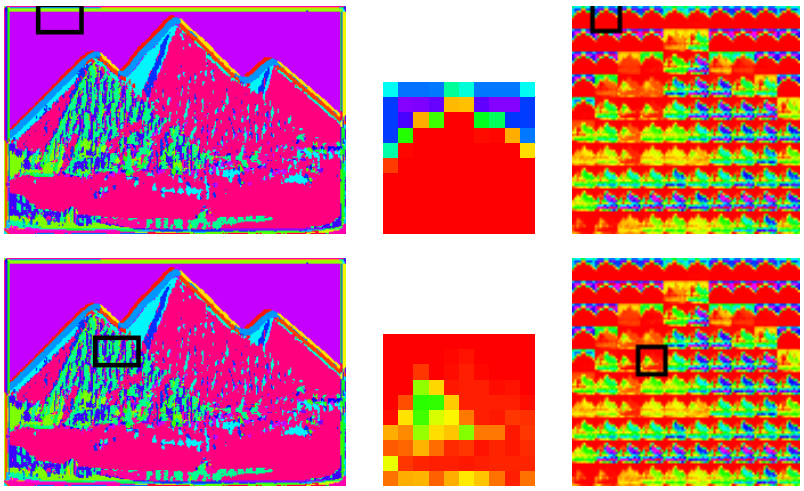


Figure 17: Assignment map $\mathcal{M}_I$ with two further exemplary overlay cells and their hypercube slice.

Appendix B

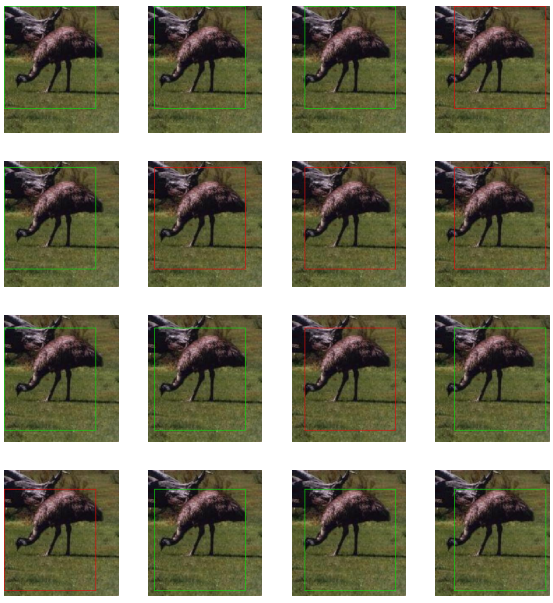
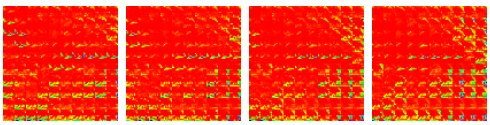
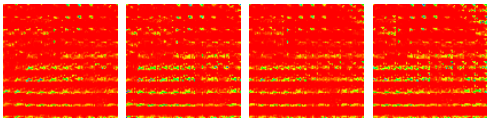


Figure 18: Emu 3, sliding windows at depth level depth 2 (starting at level 0) and the classification of the generic prototype assignment map.

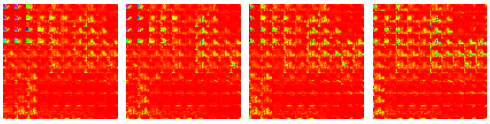
Appendix C



(a) 1st row level 2 sliding window of the emu.



(b) 1st row level 2 sliding window of the flamingo.



(c) 1st row level 2 sliding window of the flamingo.

Figure 19: SSH descriptors on the first row of sliding windows at depth level 2 on the image specific assignment maps of figure 14.