SIFTpack: a compact representation for efficient SIFT matching

Alexandra Gilinsky
Technion
Israel Institute of Technology
sashkagil@gmail.com

Lihi Zelnik-Manor
Technion
Israel Institute of Technology
lihi@ee.technion.ac.il

Abstract

Computing distances between large sets of SIFT descriptors is a basic step in numerous algorithms in computer vision. When the number of descriptors is large, as is often the case, computing these distances can be extremely time consuming. In this paper we propose the SIFTpack: a compact way of storing SIFT descriptors, which enables significantly faster calculations between sets of SIFTs than the current solutions. SIFTpack can be used to represent SIFTs densely extracted from a single image or sparsely from multiple different images. We show that the SIFTpack representation saves both storage space and run time, for both finding nearest neighbors and for computing all distances between all descriptors. The usefulness of SIFTpack is also demonstrated as an alternative implementation for K-means dictionaries of visual words.

1. Introduction

In numerous applications in computer vision a basic building block is the computation of distances between sets of SIFT descriptors [16]. Some applications require finding the nearest match for each descriptor, e.g., image alignment, scene classification and object recognition [9, 14, 19, 25]. In other cases one needs to compute all the distances between all the SIFTs, e.g., when constructing affinity matrices for image segmentation [3], co-segmentation [11, 12] or self-similarity [23]. As the number of descriptors increases, the computation time of these distances becomes a major consideration in the applicability of the algorithm.

Previous solutions to reduce the run-time, can be categorized into three different approaches. The first approach reduces the dimensionality of the descriptors, thus decreasing the computation time of each distance calculation [27]. This comes at the cost of loss of accuracy due to the reduced dimension. The second approach reduces the number of descriptors by using sparse sampling of the image [26, 29]. This could compromise accuracy, as it was repeatedly shown that dense-sampling yields better results in recognition [9, 14, 19]. Furthermore, in many applications, the obtained set of descriptors could still be very large, e.g., when reconstructing 3D scenes using thousands of images. Hence, efficient methods for computing the distances are still required. Finally, the third approach is based on approximate solutions [2, 8]. These reduce run-time significantly, however, they are relevant only for the nearest-neighbor case and become inefficient when all distances need to be computed. Algorithms for approximate matching across images are also highly efficient [5, 10, 13, 20], but are limited to dense matching across a pair of images and most of them cannot be applied for matching SIFTs.

In this paper we propose the SIFTpack: a compact form for storing a set of SIFT descriptors that reduces both storage space and run-time when comparing sets of SIFTs. Our key idea is to exploit redundancies between descriptors to store them efficiently in an image-like structure. Redundancies can occur, for example, when two descriptors are computed over overlapping image regions. In this case the descriptors have a shared part that doesn't need to be stored twice. The SIFTpack construction identifies such redundancies and hence saves in storage space. We show how SIFTpack can be used to compactly represent descriptors extracted densely from a single image. We further suggest an algorithm for constructing SIFTpacks for descriptors extracted sparsely from one or many images. Such SIFTpacks can serve as an efficient alternative to SIFT dictionaries.

The key contribution of this work is a significant reduction in run-time when computing distances between sets of SIFTs. The speed-up is due to two reasons. First, we avoid repeated calculations of the same distances. Second, since the SIFTs are stored in an image-like form, we can utilize existing efficient algorithms for fast matching between images. We suggest such solutions for both nearest-neighbor matching and all-distances calculation. The proposed solutions are shown to be useful for bag-of-words models.

The rest of the paper is organized as follows. We start by describing the SIFTpack construction for dense-SIFT and for an arbitrary set of SIFTs in Section 2. Efficient algorithms for matching sets of SIFTs, based on SIFTpack,

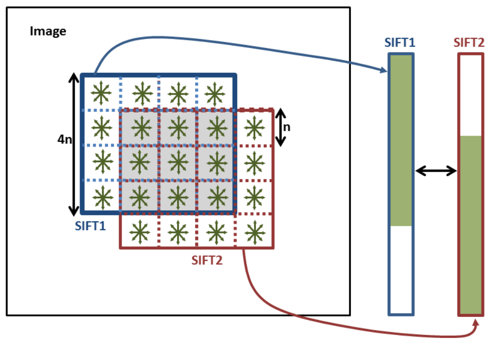


Figure 1. **Redundancy in overlapping SIFTs:** When two SIFT descriptors are computed over overlapping image regions the shared region (marked in gray) could result in common SIFT values in the descriptors (marked in green). The joint values will appear at different entries of the vectors and will be stored twice by the standard approach of keeping SIFTs in an array.

are presented in Section 3. The usefulness of SIFTpack as an efficient alternative to Bag-of-words is presented in Section 4. Finally, we conclude in Section 5.

2. The SIFTpack representation

The SIFTpack construction that we propose is based on two observations. The first is that two different SIFT descriptors could still have shared components. This could occur, for example, when the regions over which they were computed have a highly similar appearance. Storing these shared components twice is redundant and inefficient. The second observation we make is that highly efficient solutions have been proposed for computing nearest-neighbor fields across images. This motivates us to seek an image-like representation for a set of SIFTs. In what follows we suggest a representation that builds upon these two observations. We first describe how SIFTpack is constructed for dense-SIFT and then continue to present an algorithm for constructing a SIFTpack for an arbitrary set of SIFTs.

2.1. SIFTpack for dense-SIFT

We start by briefly reviewing the structure of the SIFT descriptor [16]. The SIFT at a pixel is based on the gradients within a neighborhood around it. The neighborhood is divided into $4 \times 4 = 16$ sub-regions and an 8-bin histogram of weighted gradients is computed for each sub-region. The weights are set according to the distance from the center pixel. The 16 histograms are then concatenated into a 128-dimensional vector. Typically, SIFTs are stored in a $128 \times M$ array, where M is the number of descriptors.

To efficiently store dense-SIFT descriptors [28] we start by computing the SIFTs on a grid with an n pixel gap. We

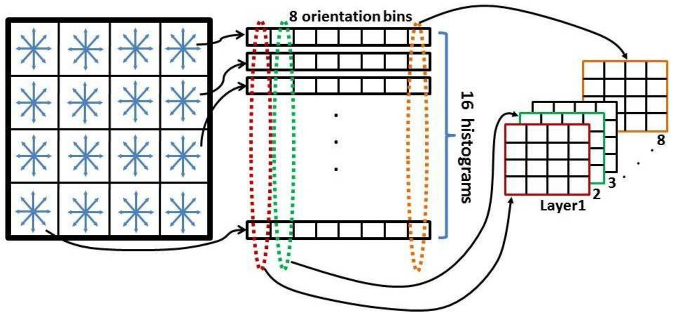


Figure 2. **Reshaping SIFT:** SIFTpack requires reshaping the 128 dimensional SIFT descriptor into 8 layers of 4×4 pixels each.

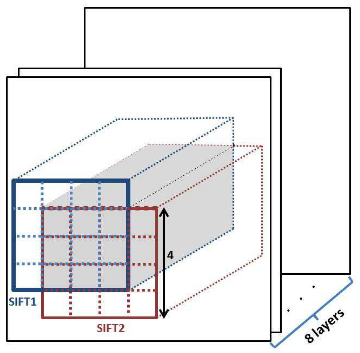


Figure 3. **SIFTpack:** is an 8-layer image, where each 4×4 patch corresponds to a single SIFT descriptor. Histograms that are shared between descriptors are stored only once in this construction, e.g., the gray area is mutual to the blue and red SIFTs.

set the scale such that each descriptor is computed over a region of $4n \times 4n$ pixels, i.e., each 8-bin histogram is computed over a sub-region of size $n \times n$ pixels. As illustrated in Figure 1, in this construction the spatial regions of neighboring descriptors overlap and the sub-regions are aligned. To simplify the presentation we will assume for the time being that no weighting is applied to the pixels when constructing the descriptors. Later on we will remove this assumption. Without the weighting, two descriptors with overlapping regions will have at least one shared histogram. In fact, all the descriptors including a certain sub-region will share its corresponding gradient histogram. While the standard approach stores this histogram multiple times, once for each descriptor, we wish to store it only once.

To enable a compact and image-like representation we start by reshaping the extracted SIFT descriptors. As illustrated in Figure 2, we reshape each 128 dimensional SIFT vector into a 3D array of dimensions $4 \times 4 \times 8$. We think of this 3D array as an image with 4×4 pixels and 8 layers. Each layer corresponds to one of the 8 bins of the histograms, i.e., each layer captures one of the eight gradient orientations considered by SIFT. Each "pixel" corresponds to one of the 16 histograms used to construct the descriptor.

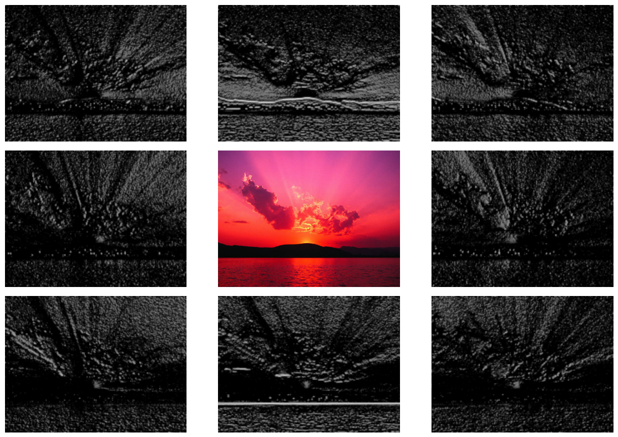


Figure 4. **Visualizing SIFTpack:** The 8 gray scale images are the layers of the SIFTpack of the center image. As expected, they capture gradients at different orientations. The SIFTpack layers are rescaled to the image size for visibility but their original size is n times smaller in each axis.

The key idea behind our construction is that after the reshape step, the shared histograms of two neighboring descriptors correspond to shared "pixels". Therefore, to store the descriptors compactly, we simply place them next to each other, with overlaps, according to their original position in the image. This is illustrated in Figure 3. The SIFTpack storage space is 16 times smaller than that of the conventional approach of saving all SIFTs in an array. The construction time is linear in the number of descriptors.

To complete our construction, we next remove the temporary assumption of no pixel weighting. In practice, Gaussian weighting is applied to the region over which the descriptor is computed, thus two overlapping SIFTs do not share the exact same entry values. Therefore, we average the values of all SIFTs that overlap i.e. we store only one, averaged version, of all SIFT entries that describe the same spatial bin. Figure 4 displays the layers of a SIFTpack constructed from dense-SIFT.

To examine the implications of averaging the SIFT values, we performed the following experiment. We used SIFT-flow [15] to compute the flow-field between the pairs of images of the Middlebury data-set [4] and on the "light" data-set of "Mikolajczyk" **[18]**. We performed this twice, once with the original SIFT descriptors, and once while averaging overlapping descriptors. As shown in Table 1, not only does the averaging not interfere, in fact it slightly improves the correspondence accuracy and reduces errors. This is probably due to the enhanced robustness to noise when smoothing. For this experiment we used normalized values of SIFT. While the averaging described above alters the normaliztion a bit, this is not significant as matching accuracy is not harmed.

To summarize, our method for constructing SIFTpack for dense-SIFT is outlined in Algorithm 1.

Algorithm 1 SIFTpack for dense-SIFT

```
**Input:** Image
Compute dense-SIFT, each over a $4n \times 4n$ neighborhood, on a grid with $n$ pixels spacing.
**for all** SIFTs **do**
    – Reshape into a $4 \times 4 \times 8$ array.
    – Place SIFT of pixel $i, j$ at SIFTpack location $i/n, j/n$ (these are integer values due to the grid).
    – Average all overlapping values.
**end for**
**Output:** SIFTpack
```

2.2. SIFTpack for a set of SIFTs

Next, we suggest a similar construction for the more popular case where SIFTs are extracted at isolated locations (typically interest points), possibly from multiple different images, of different scales and orientations. When the number of descriptors is large we expect to find many similarities between part of them, e.g., when multiple descriptors include sub-regions of similar appearance. However, unlike the dense case, here we cannot tell a-priori which descriptors have corresponding components.

To identify these repetitions and enable the construction of a compact SIFTpack we build upon the dictionary optimization framework of [1]. Given a set of M SIFT descriptors y_i, $i = 1, \dots, M$ we wish to find a SIFTpack S, of size $m \times m \times 8$, that forms a sparsifying dictionary for them. S is obtained by optimizing the following objective:

$$\begin{aligned} \min_{S,x} \quad & \sum_{i=1}^{M} \left\| y_i - \sum_{k=1}^{m}\sum_{l=1}^{m} x_{i[k,l]} C_{[k,l]} \hat{S} \right\|_F^2 \\ \text{s.t.} \quad & \|x_i\|_0 \le \theta, \; i = 1, \dots, M \end{aligned} \tag{1}$$

where $\hat{S}$ is a vector obtained by column-stacking the SIFTpack S, $C_{[k,l]}$ is an indicator matrix that extracts the 128-dimensional SIFT descriptor in location $[k, l]$ in S and $x_{i[k,l]}$ are the corresponding sparse coding coefficients.

Problem (1) is not convex with respect to both S and x, however, when fixing either one of them it is convex with respect to the other. Therefore, to find a local minimum of Problem (1) we iterate between optimizing for the sparse coefficients x via Orthogonal Matching Pursuit (OMP) [21, 17] and optimizing for the SIFTpack S. In all our experiments we set the sparsity $\theta = 1$ and $x_{i[k,l]} = 1$ for only a single SIFTpack location $[k, l]$ and all other values are 0. In practice, this implies that the OMP finds for each original SIFT y_i the closest SIFT within the current SIFTpack. The SIFTpack update step is:

$$\hat{S} = \mathbf{R}^{-1} p, \tag{2}$$

	without averaging	with averaging (SIFTpack)
name	error (pixels)	error (pixels)
Dimetrodon	1.6104	**1.6089**
Grove2	1.7314	**1.7297**
Grove3	1.9034	**1.8883**
Hydrangea	0.6690	**0.6446**
RubberWhale	1.2247	***1.2211***
Urban2	1.5638	***1.5302***
Urban3	141.1214	***22.5310***
Venus	1.2661	***1.2435***

(a)

	without averaging	with averaging (SIFTpack)
num	error (pixels)	error (pixels)
1	1.5397	**1.5239**
2	1.9062	**1.8774**
3	1.9281	**1.8884**
4	1.2238	**1.1979**
5	2.0961	**2.0459**

(b)

Table 1. **Flow-field accuracy on Middlebury (a) and Mikolajczyk data-set (b):** As can be seen from both tables, the flow-fields computed by SIFT-flow [15] are consistently more accurate when applied to descriptors packed in SIFTpack, than the original SIFTs (see supplementary for more detailed results). This suggests that averaging overlapping SIFTs reduces noise and hence matching accuracy is improved.

where:

$$\mathbf{R} = \sum_{i=1}^{M} \left(\sum_{k=1}^{m} \sum_{l=1}^{m} x_{i[k,l]} C_{[k,l]} \right)^T \left(\sum_{k=1}^{m} \sum_{l=1}^{m} x_{i[k,l]} C_{[k,l]} \right)$$

$$p = \sum_{i=1}^{M} \left(\sum_{k=1}^{m} \sum_{l=1}^{m} x_{i[k,l]} C_{[k,l]} \right)^T y_i$$

It can be shown that since we use sparsity $\theta = 1$ and set $x_{i[k,l]} = 1$, this is equivalent to the averaging of all SIFTs that were assigned to the same location in S.

The above optimization requires initialization. When the SIFTpack represents a set of SIFTs extracted at interest points of multiple different images we have found empirically that best results were obtained when initializing with a SIFTpack constructed from a dense-SIFT of a randomly chosen image, as described in the previous section. Note that the initial image should be textured, otherwise the results will be inferior.

Our method for constructing SIFTpack for any set of SIFTs is summarized in Algorithm 2.

Figure 5.(a) visualizes a SIFTpack constructed by Algorithm 2. The final SIFTpack is of size $m \times m \times 8$, where m is a user defined parameter. The smaller m is, the more storage space is saved. However, this comes at the cost of lower accuracy, as illustrated in Figure 5.(b). When the SIFTs are taken from multiple images, and $m \ll M$, a SIFTpack constructed by Algorithm 2 can be viewed as a dictionary of visual words. It differs from the standard dictionaries in exploiting redundancies between visual words to store the dictionary more compactly. Note the resemblance between Algorithm 2 and the K-means algorithm, as their complexity is the same.

Since the proposed optimization framework is applicable to any arbitrary set of SIFTs it is also applicable to dense-SIFT, leading to further reduction in the required storage space, as compared to that obtained by Algorithm 1. Inspired by [24], when the input is dense-SIFT we iteratively reduce the size of the SIFTpack to 95% of its previous dimension, initializing each iteration with a resized version of the previous one. The iterations continue until the desired size is reached. The advantage of the gradual resizing is a more compact SIFTpack for the same representation error. On the down side, applying Algorithm 2 with gradual resizing is slower, and hence is not always preferable.

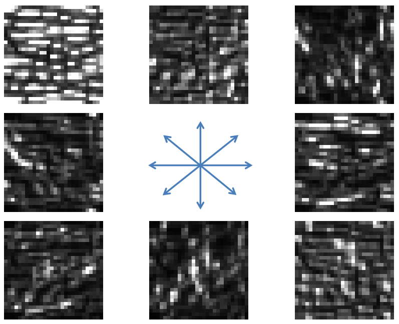
Algorithm 2 SIFTpack for a set of SIFTs

```
Input:
– A set of $M$ SIFTs
– The desired SIFTpack size $m$
Initialization
– Construct a SIFTpack of size $m \times m \times 8$ using Algorithm 1 on an arbitrary image.
repeat
    – Assign each SIFT in the input set to its most similar SIFT in the current SIFTpack.
    – Update the SIFTpack by averaging SIFTs assigned to overlapping SIFTpack entries.
until Objective of Eq. (1) converges.
Output: SIFTpack
```

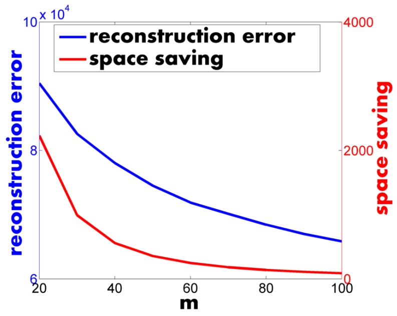
It is important to note, that while Algorithm 2 is described for the L_2 distance between SIFTs, our framework is generic and can be applied to many other distance metrics. Exchanging the distance metric amounts to replacing the update step of Equation (2) by an appropriate calculation for averaging overlapping SIFTs and exchanging stage 1 with regards to the new metric.

3. Efficient matching solutions

So far we have described algorithms for constructing a space-efficient representation for multiple SIFTs, extracted



(a)



(b)

Figure 5. **SIFTpack for a set of SIFTs:** (a) The 8 layers of a SIFTpack constructed of $\sim 50,000$ SIFTs extracted at interest points from different images with different scales. (b) The average representation error (blue) and saving in storage space (red) as a function of the SIFTpack size m. The smaller the SIFTpack, the more space we save, at the price of a larger representation error.

from either one or multiple images. While saving in storage space is a desirable property, our main goal is to obtain a significant reduction in computation time as well. A main advantage of the SIFTpack is that it can be viewed as an 8-layer image, therefore, one can employ existing algorithms for efficient matching across images.

3.1. Computing all distances

In applications such as image segmentation, co-segmentation and self-similarity estimation one needs to compute all (or multiple) distances between all (or multiple) pairs of descriptors within a given set or across two sets of SIFTs. When the number of descriptors is large, computing all distances naively is highly time consuming: $O(M^2)$, where M is the number of descriptors.

Storing the descriptors in a SIFTpack enables a more efficient computation since we avoid redundant calculations. As described in Section 2, the SIFTpack stores joint descriptor-parts only once, hence, they are used only once when computing distances. We next present an efficient algorithm, which makes use of the special image-like structure of the SIFTpack to avoid redundant calculations.

Let $S1$ and $S2$ denote the SIFTpacks constructed for two set of SIFTs. In applications where distances are to be computed between the descriptors of a single set, we assign $S2 = S1$. We further denote by $S_{i,j}$ the SIFT descriptor in location i, j in S, and define $S^{[k,l]}$ as the shift of S by k, l pixels in x, y directions, respectively.

Our approach for efficiently computing all distances between the descriptors of $S1$ and $S2$ is adopted from [20], where it was used to compare image patches. We use the Integral Image [6] to compute the distance between all pairs of descriptors in $S1$ and $S2$ that have the same location i, j. To compute distances between descriptors at different locations we loop through all shifts k, l of $S2$. Our algorithm can be summarized as follows:

Algorithm 3 All distances between SIFTs

```
**Input:** SIFTpacks $S1$ and $S2$
**for all** shifts $k, l$ **do**
    – Compute the element-wise square difference
    $\Delta = (S1 - S2^{[k,l]})^2$
    – Compute the Integral Image $F(\Delta)$ , summing the 8 layers.
    – The distance between $S1_{i,j}$ and $S2^{k,l}_{i,j}$ is equal to:
    $F(i, j) + F(i + 3, j + 3) - F(i + 3, j) - F(i, j + 3)$
**end for**
**Output:** Distances between all SIFTs
```

Algorithm 3 loops through all possible shifts k, l, similarly to the naive solution. However, it computes the distances between all pairs of descriptors with a location shift k, l, faster than the standard solution. The speedup is due to the integral image approach, that is possible here since our SIFTpack has an image-like structure.

Figure 6 ascertains this via empirical comparison between the run-time of the naive approach and Algorithm 3. For this experiment we first extract the dense-SIFT [28] descriptors of images of varying sizes. We then compute the distances between all pairs of SIFTs within a a radius R of each other, using the naive approach. Next, we construct a SIFTpack for each image, using Algorithm 1, and compute the distances between the same pairs of SIFTs using Algorithm 3. We use the “RetargetMe” data-set [22] together with some images collected by us, that together consist of 90 images. We used different sizes for each image by applying resizing. For each image we repeated the distances computation 100 times and averaged the results. As can be seen, for all image sizes and for all radii R the reduction in run-time of the SIFTpack approach varies between one and two orders of magnitude. This experiment, as well as all others presented in this paper, were performed on an i7 2.53GHz Linux machine with 16Gb RAM, using one core. Similar results were obtained on i7 3.4GHz, 16Gb Windows machine.

The above experiment shows the speed-up that can be obtained for applications that require computing distances

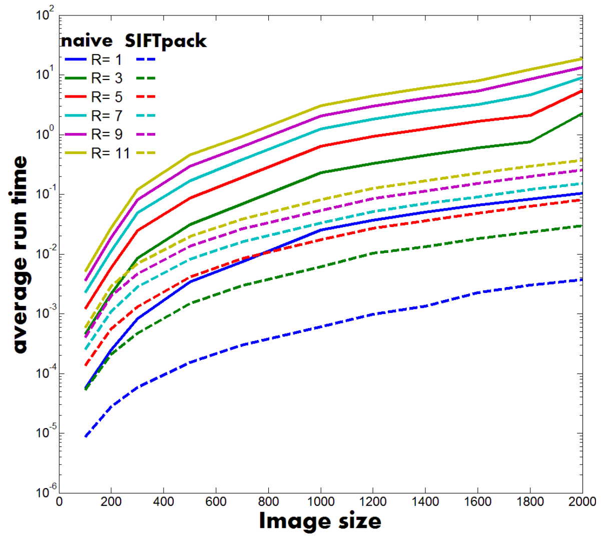


Figure 6. **Run-time saving by SIFTpack when computing multiple distances:** The curves represent the average run-time for computing distances between all pairs of SIFTs of an image, within a radius R of each other. Solid curves correspond to the naive approach and dotted lines correspond to Algorithm 3. As can be seen, the standard approach is an order of magnitude slower than using SIFTpack and Algorithm 3. This holds for varying image sizes and different R values.

between dense descriptors, e.g., image segmentation and self-similarity. When one needs to compute all distances between two arbitrary sets of SIFTs, the SIFTpack construction needs to be performed via Algorithm 2, which is time consuming on its own. In such cases using our approach makes sense when the SIFTpack can be computed and stored a-priori.

3.2. Exact Nearest-Neighbor matching

When a single nearest-neighbor is needed, the naive solution is to compute all distances and take the minimum. Alternatively, one could use more efficient tree-based methods such as that proposed in [2]. We propose to use SIFTpacks and Algorithm 3, with the slight modification that only the nearest neighbor is stored for each descriptor. We performed an experiment on the "VidPairs" data-set [13] which consists of 133 pairs of images. We used resized versions of each pair to test on images of different sizes. We computed the exact NN field between each pair of different sizes. Figure 7 shows that our solution is much faster than that of [2].

3.3. Approximate Nearest-Neighbor matching

Due to the high computational demands of finding the exact nearest-neighbor, more efficient approaches have been proposed that settle for finding the Approximate-Nearest-Neighbor (ANN) [2, 8]. These methods provide

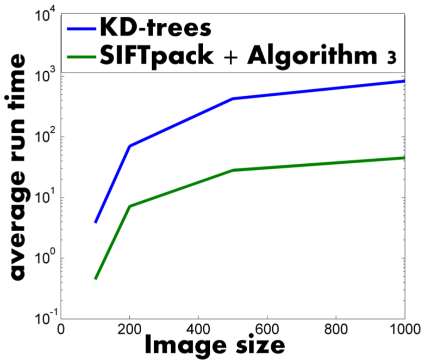


Figure 7. **Run-time saving by SIFTpack for Exact-Nearest-Neighbor:** Computing the exact NN using SIFTpacks and Algorithm 3 is significantly faster than using Kd-trees [2].

significant run-time reduction at the price of loss in accuracy. More recently it has been shown that even faster algorithms can be developed when computing ANN densely across images [5, 10, 13, 20]. These algorithms utilize the coherency between near-by image patches to speed-up the matching. Since the SIFTpack can be viewed as an 8-layer image, where each patch represents a SIFT descriptor, we can apply these algorithms to find ANN between SIFTs.

Figure 8 presents an empirical evaluation of the benefits of SIFTpack for computing ANN. Given a random pair of images and their corresponding dense-SIFT descriptors, we find for each descriptor in one image its ANN in the second image, using four methods: (i) SIFT-flow [15], (ii) Kd-trees [2] computed over the set of SIFTs, (iii) PatchMatch [5] applied to standard dense-SIFT and (iv) TreeCANN [20] applied to the SIFTpack. TreeCANN is one of the fastest algorithms for ANN that can be applied only for finding ANN between image patches. Note, that the standard approaches to date are (i)–(iii), while our proposed SIFTpack, being an image, enables method (iv). Similarly to the exact NN experiment, we test the performances on the "VidPairs" data-set [13], repeating 100 times each NN field calculation and averaging. As can be seen, SIFTpack+TreeCANN significantly outperforms both Kd-Trees and PatchMatch in both accuracy and run-time. We have omitted the results of SIFT-flow as their quality was too low, probably since this approach is designed only for pairs of highly similar images.

4. SIFTpack as a Bag-of-Words

Another advantage of our construction is that a SIFTpack built from SIFTs extracted from multiple different images, using Algorithm 2, can be viewed as an alternative for the highly popular dictionaries of visual words [7], which are typically constructed by K-means clustering. The dictionary construction process is performed off-line, hence, its computation time is of lesser interest. Our main objectives

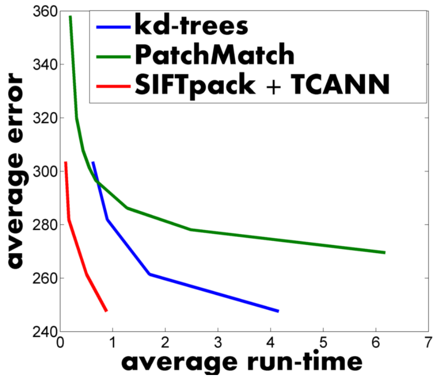


Figure 8. **Run-time saving by SIFTpack for ANN:** Computing the ANN using SIFTpack and TreeCANN leads to significantly lower errors and faster run time than both Kd-trees and PatchMatch. These graphs are for images of size 800×800, but similar results were obtained for other image sizes as well.

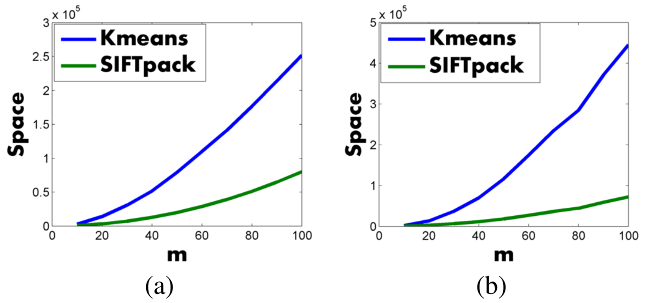


Figure 9. **Space saving by SIFTpack for BoW:** The plots present the required storage space of SIFTpacks of varying representation errors, constructed with Algorithm 2, and of corresponding k-means dictionaries (with the same representation error). (a) The results when representing a set of SIFTs from multiple different images. (b) The results when representing dense-SIFT of a single image. In all cases SIFTpack is significantly more space-efficient than the standard k-means dictionary.

are thus two-fold. First, we wish to evaluate the benefits in storage space and representation error of SIFTpack in comparison with the standard K-means dictionary. Second, we wish to examine the contribution in run-time when using the SIFTpack to compute the histogram of word frequencies. The histogram is obtained by finding for each given SIFT the most similar SIFT word in the SIFTpack/dictionary.

Storage space saving: To assess the benefits of SIFTpack in terms of storage space we performed the following experiment. Given a set of SIFTs we use Algorithm 2 to construct the corresponding SIFTpack, of size $m \times m \times 8$. We then compute the representation error of the SIFTpack as the average L_2 difference between each original SIFT and its nearest-neighbor in the SIFTpack. In addition, we also construct a standard dictionary using K-means, setting K such that the same (as much as possible) representation error is obtained. Finally, we compare the array size of the SIFTpack and of the K-means dictionary. We have repeated this experiment for varying representation errors.

Figure 9 presents the obtained results for two setups. Figure 9.(a) shows the results when the set of SIFTs was extracted at interest points of $\sim$ 1700 images from the scene classification database [9] of 6 different scenes: "kitchen", "bedroom", "MITcoast", "MIThighway", "MITmountain" and "MITstreet". Figure 9.(b) shows the results when representing dense-SIFT of a single image. The results are averaged over 80 repetitions of the experiment. We used $m = [10, 20, 30 \ldots 100]$. The units of space are the number of entries ($m \times m \times 8$ for SIFTpack and $K \times 128$ for K-means dictionary). As can be seen, the saving in storage space is tremendous for both setups. When packing dense-SIFTs the space-reduction is more significant since the SIFTpack is more compact due to the gradual resizing during its construction.

Run-time saving: Next, we assess the run-time benefits of SIFTpack when constructing the bag-of-words representation for an image. The evaluation is done via the following experiment. We use as test-bed the SIFTpack and K-means dictionaries constructed in the previous experiment from $\sim$ 1700 images of 6 different scenes (corresponding pairs with the same representation error). Given an arbitrary input image we first extract dense-SIFT descriptors and construct the corresponding SIFTpack using Algorithm 1. Next, we find for each descriptor its nearest-neighbor in the k-means dictionary and in the SIFTpack. We use the common naive approach as well as the kd-tree algorithm [2] for searching the K-means dictionary. We apply Algorithm 3 to find the exact nearest neighbor within the SIFTpack.

Figure 10 presents the obtained results, averaged over multiple experiments. As can be seen, using SIFTpack is significantly faster than using the popular k-means dictionary, even when kd-trees are used. We should note that as far as exact NN are concerned, the kd-tree algorithm doesn't outperform the naive one significantly.

5. Conclusions

This paper suggested an approach for compactly storing sets of SIFT descriptors. We have shown that the proposed SIFTpack saves not only in storage space, but more importantly, it enables huge reductions in run-time for matching between SIFTs. While our key idea is very simple, its implications could be highly significant as it can speed-up many algorithms. In particular, we have shown empirically the benefits of using SIFTpack instead of the traditional Bag-of-Words dictionary and as an alternative image signature for retrieval purposes. We believe that storing SIFTs as an image could open the gate to using other algorithms on SIFTs that were restricted to only images before. In addi-

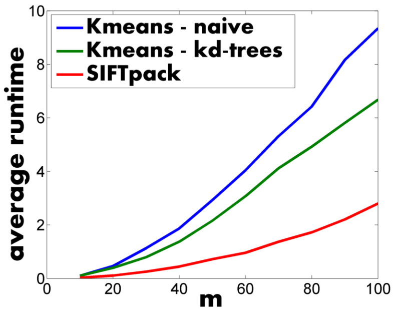


Figure 10. **Run-time saving by SIFTpack for BoW:** Computing the bag-of-words model (constructing the histogram of frequencies) is significantly faster when using SIFTpack to represent the dictionary, instead of the standard k-means dictionary with the same representation error.

tion, our framework could be easily extended to other descriptors whose spatial properties are similar to SIFT.

Acknowledgements This research was supported in part by the Ollendorf foundation and by the Israel Ministry of Science. We'd also like to thank Prof. Michael Elad for useful conversations and good ideas.

References

[1] M. Aharon and M. Elad. Sparse and redundant modeling of image content using an image-signature-dictionary. *SIAM J. Imaging Sciences*, 1(3):228–247, 2008.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[3] S. Bagon, O. Boiman, and M. Irani. What is a good image segment? a unified approach to segment extraction. In *ECCV*, pages 30–44, 2008.

[4] S. Baker, D. Scharstein, J. Lewis, S. Roth, M. Black, and R. Szeliski. A database and evaluation methodology for optical flow. *IJCV*, 92(1):1–31, 2001.

[5] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. Patchmatch: a randomized correspondence algorithm for structural image editing. *SIGGRAPH*, 28(3):24:1–24:11, July 2009.

[6] F. C. Crow. Summed-area tables for texture mapping. *SIGGRAPH*, 18(3):207–212, Jan. 1984.

[7] G. Csurka, C. Dance, L. Fan, J. Willamowski, and C. Bray. Visual categorization with bags of keypoints. In *ECCV*, volume 1, pages 1–22, 2004.

[8] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, pages 253–262, 2004.

[9] L. Fei-Fei and P. Perona. A bayesian hierarchical model for learning natural scene categories. In *CVPR*, pages 524–531, 2005.

[10] K. He and J. Sun. Computing nearest-neighbor fields via propagation-assisted kd-trees. In *CVPR*, pages 111–118, 2012.

[11] A. Joulin, F. Bach, and J. Ponce. Discriminative clustering for image co-segmentation. In *CVPR*, pages 1943–1950, 2010.

[12] A. Joulin, F. Bach, and J. Ponce. Multi-class cosegmentation. In *CVPR*, pages 542–549, 2012.

[13] S. Korman and S. Avidan. Coherency sensitive hashing. In *ICCV*, pages 1607–1614, 2011.

[14] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *CVPR*, pages 2169–2178, 2006.

[15] C. Liu, J. Yuen, and A. Torralba. Sift flow: Dense correspondence across scenes and its applications. *TPAMI*, 33(5):978–994, 2011.

[16] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2):91–110, 2004.

[17] S. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing,*, 41(12):3397 –3415, 1993.

[18] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *TPAMI*, 27(10):1615–1630, 2005.

[19] E. Nowak, F. Jurie, and B. Triggs. Sampling strategies for bag-of-features image classification. In *ECCV*, pages 490–503, 2006.

[20] I. Olonetsky and S. Avidan. Treecann - k-d tree coherence approximate nearest neighbor algorithm. In *ECCV*, pages 602–615, 2012.

[21] Y. C. Pati, R. Rezaiifar, Y. C. P. R. Rezaiifar, and P. S. Krishnaprasad. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proceedings of the 27 th Annual Asilomar Conference on Signals, Systems, and Computers*, pages 40–44, 1993.

[22] M. Rubinstein, D. Gutierrez, O. Sorkine, and A. Shamir. A comparative study of image retargeting. *SIGGRAPH*, 29(5):160:1–160:10, 2010.

[23] E. Shechtman and M. Irani. Matching local self-similarities across images and videos. In *CVPR*, pages 1–8, 2007.

[24] D. Simakov, Y. Caspi, E. Shechtman, and M. Irani. Summarizing visual data using bidirectional similarity. In *CVPR*, pages 1–8, 2008.

[25] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *CVPR*, pages 1470–1477, 2003.

[26] C. Strecha. Dense matching of multiple wide-baseline views. In *ICCV*, pages 1194–1201, 2003.

[27] E. Tola, V. Lepetit, and P. Fua. A fast local descriptor for dense matching. In *CVPR*, pages 1–8, 2008.

[28] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms, http://www.vlfeat.org/, 2008.

[29] J. Yao and W. kuen Cham. 3d modeling and rendering from multiple wide-baseline images by match propagation. *Sig. Proc.: Image Comm.*, 21(6):506–518, 2006.