AnnArbor: Approximate Nearest Neighbors Using Arborescence Coding

Artem Babenko
Yandex, Moscow
National Research University
Higher School of Economics, Moscow
artem.babenko@phystech.edu

Victor Lempitsky
Skolkovo Insitute of Science and Technology
(Skoltech), Moscow
lempitsky@skoltech.ru

Abstract

To compress large datasets of high-dimensional descriptors, modern quantization schemes learn multiple codebooks and then represent individual descriptors as combinations of codewords. Once the codebooks are learned, these schemes encode descriptors independently. In contrast to that, we present a new coding scheme that arranges dataset descriptors into a set of arborescence graphs, and then encodes non-root descriptors by quantizing their displacements with respect to their parent nodes. By optimizing the structure of arborescences, our coding scheme can decrease the quantization error considerably, while incurring only minimal overhead on the memory footprint and the speed of nearest neighbor search in the compressed dataset compared to the independent quantization. The advantage of the proposed scheme is demonstrated in a series of experiments with datasets of SIFT and deep descriptors.

1. Introduction

Visual search and other computer vision applications are routinely dealing with million or billion-scale datasets of visual descriptors corresponding to images and/or image parts. Lossy compression of such descriptor datasets that reduce their memory footprint and increase the search speed have therefore become an active area of research. Currently, approaches based on (non-binary) quantizations [14, 12, 18, 3, 22, 5] achieve the best compression error-compression ratio trade-off, while also permitting efficient computation of scalar products and squared distances between uncompressed queries and compressed descriptor sets using look-up tables. For million-scale datasets, the look-up tables allow fast exhaustive search that scans through entire datasets in a matter of milliseconds.

Existing quantization approaches represent dataset descriptors as combinations of codeword vectors that come from different codebooks. The codebooks are invariably adapted to the dataset (or its hold-out part) through the optimization process, so that statistical regularities of the descriptor distribution can be exploited for better coding accuracy. Importantly, once the codebooks have been learned, existing quantization schemes apply *independent* coding to each of the dataset descriptors.

In this work, we propose the approach that brings further improvement in terms of coding accuracy on top of the existing descriptor coding techniques, while incurring small memory and search time overheads. The improvement comes as we consider *joint* coding of descriptors in the dataset that goes beyond codebook learning. Our approach (*arborescence coding*) avoids direct coding of the majority of the dataset vectors, and instead focuses on coding *relative* displacements between nearby vectors. A similar idea underlies predictive coding [11], however arborescences coding goes beyond predictive coding by selecting sets of parent-children pairs that are most suitable for predictive quantization.

The main idea of our approach is simple (Figure 1). During coding, the optimization process splits the dataset into a set of arborescence graphs (i.e. directed trees), with individual descriptors being the vertices of those arborescences. For each arborescence, the topology (structure), the absolute position of the root, and the relative displacements along the arcs are encoded and stored using quantization techniques. When computing the scalar product with the query vector, the scalar product between the query and the root vector is computed first, and the scalar products with other vectors are computed in a breadth-first manner. Such breadth-first scan process can then benefit from the standard look-up table tricks used by quantization methods [14] thanks to the additivity of the scalar product. Arborescence coding therefore does not incur significant reductions in search speed compared to the base quantization algorithm.

Crucially, arborescence coding makes the topology (including the number) of the arborescences part of the optimization process during the encoding of the dataset. In the process of optimization, individual descriptors are free

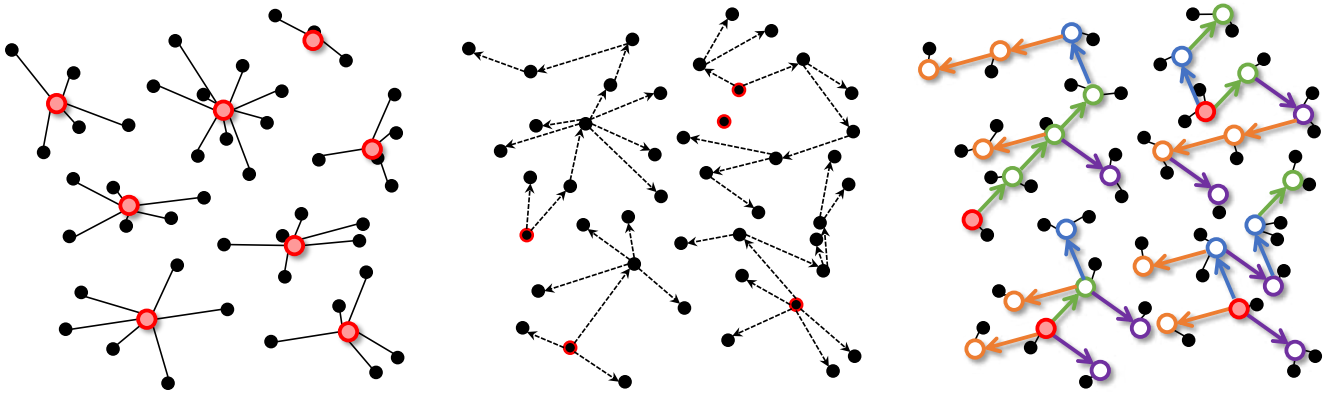

Figure 1: **Independent (traditional) coding vs. Arborescence coding.** *Left* – a set of descriptors (black dots) are encoded using the codebook of eight codewords (red). Each descriptor gets assigned to the closest codeword. This is a standard, independent (given the codebook) coding approach resulting in large coding errors (thin solid lines). *Middle* – arborescence coding splits the dataset into a set of arborescence graphs (the roots of arborescences are highlighted in red). *Right* – following the structure of these arborescences, the coding uses one four-word codebook to encode root descriptors (red circles), and another four-word codebook to encode displacements along arcs in the arborescences (the codewords are shown as green, magenta, orange, blue vectors). By coordinating the choice of arborescence topology and the codebooks, arborescence coding creates reconstructions (circles) that result in much lower coding errors (thin lines) than independent coding, while still using one codeword per descriptor and eight different codewords for the dataset. The cost of storing arborescence topology is small and independent of space dimensionality. *Note:* in this 2D toy example, single codebook quantizations are used. In high dimensions, both plain coding and arborescence coding can benefit from multi-codebook quantization methods (e.g. product quantization [14]) .

to choose whether to become roots (and to be encoded directly) or to become non-root nodes (and to be coded relative to some *parent* descriptor). The choice of the parent is driven by the (greedy) desire to minimize the coding error. Importantly, our approach can be used on top of almost any existing quantization scheme [14, 10, 12, 18, 3, 22, 5] or, in fact, any other vector compression schemes such as generative binary hashing [20]. The optimization process in arborescence coding can be initialized with the "trivial" state where each desriptor forms a separate arborescence and is therefore coded independently. As the subsequent optimization is guaranteed not to increase the coding errors of individual vectors, our approach is guaranteed to achieve same or lower compression error compared to the base coding algorithm.

Alongside the full-fledged version of our approach (*arborescence coding*), we also consider the simpler version of the approach where all arborescences are restriced to be star-shaped (*star quantization*). In a number of experiments on datasets of various nature (SIFTs [17] and deep descriptors), we show that both arborescence coding and star coding bring consistent improvements in terms of coding error (which directly translates into the accuracy of nearest neighbor search) over the base quantization scheme, which in our experiments is optimized product quantization (OPQ) [12, 18].

Below, we cover the related work in Section 2. We then introduce the general principles of arborescence coding in Section 3. In Section 4, we discuss the specific instantiation of arborescence coding on top of optimized product quantization. We conclude with the experimental validation in Section 5 and the discussion in Section 6.

2. Related Work

The plethora of recently proposed quantization methods include product quantization [14], residual vector quantization [10], optimized product quantization [12, 18], additive quantization [3], composite quantization [22], tree quantization [5]. All of these approaches can be used as base methods for arborescence coding or star coding. In our experiments, we use optimized product quantization [12, 18] because of its appealing balance between the speed and the accuracy of the encoding process.

Arborescence coding and in particular star coding are in some ways reminiscent of the bi-layer coding approach used in the IVFADC [14] and Multi-D-ADC systems [2]. Both systems are motivated by the indexing task for very large scale datasets, as they split the descriptor space into disjoint cells, and encode the *displacements* of individual points w.r.t. cell centroids. The Multi-D-ADC system thus uses two separate codebook sets, one to encode the centroids and one to encode the displacements (while the IVFADC system stores the centroids directly). Arborescence coding also maintains difference codebooks for root encoding and descriptor encoding. However, unlike IVFADC and Multi-D-ADC that pick centroids in a separate optimiza-

tion process and automatically assign each descriptor to the nearest centroid, arborescence coding makes the optimization over possible arborescence topology part of the encoding process. In the experiments, we compare arborescence coding with the Multi-D-ADC system in a comparable setting, and find such optimization advantageous. Star coding is also related to the locally-optimized product quantization system that also uses multiple OPQ codebooks [16].

Another method that perform non-independent compression of descriptor sets is [1] that is targeted towards very strong compression of low to medium-dimensional data, and is not competitive with quantization-based approaches for the code lengths that we consider (e.g. eight bytes per vector or more).

Compression of unordered sets is also studied in the data compression community. Many of such studies are focused on sets of simple objects such as integers [21, 19, 13, 8] or real numbers [21], whereas we are interested in compression of sets of high-dimensional descriptors. A popular idea for set coding is seeking optimal permutation of the entries (re-ordering) that permits efficient predictive coding. Re-oredering has been applied to e.g. image pixels [7] and binary strings [15]. Finding optimal order involves (approximate) solution to the travelling salesman problem. The resulting Hamiltonian path can be regarded as a very large arborescence spanning the entire dataset, and therefore arborescence coding can be regarded as a generalization of re-ordering.

3. Arborescence coding

In this section, we introduce arborescence coding and its variant, star coding, in their general form. The next section then discusses the particular instantiation of arborescence coding on top of optimized product quantization [12, 18]. The variants of arborescence coding on top of other quantization schemes can then be derived analogously.

Assume that a dataset $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ of D-dimensional vectors is given. **Arborescence coding** is a lossy compression scheme that for each vector $\mathbf{x}_i$ in the dataset encodes either its absolute position in the descriptor space, or its relative displacement w.r.t. a certain other descriptor. We denote the code stored for the i-th descriptor t_i and we denote with p_i the index of its parent. Parent-child relations are constrained to form a set of arborescences (i.e. directed tree graphs). If the i-th descriptor is the root of its arborescence, then we use the convention $p_i{=}0$. Arborescences that consist of single descriptors are allowed.

We also consider a particular variant of arborescence coding (**star coding**), which corresponds to the case when all arborescences have depth at most one, i.e. no non-root descriptors are allowed to have children. Below, 'arborescence coding' refers to all variants including star coding, unless stated otherwise.

Arborescence coding requires that two decoding operations are defined that allow to recover the reconstruction $\mathbf{y}_i$ for every descriptor $\mathbf{x}_i$. The first decoder d^0 with parameters θ^0 reconstructs the roots of the arborescences:

$$\mathbf{y}_i = d^0(t_i; \theta^0), \text{ if } p_i = 0\,. \tag{1}$$

The coding process then picks the code t_i and the parameters of the encoder θ^0 to ensure that $\mathbf{y}_i \approx \mathbf{x}_i$ for root descriptors.

The second decoder d^Δ with parameters θ^Δ reconstructs the displacements $\mathbf{z}_i$ from the reconstructions of parents to their children:

$$\mathbf{z}_i = \mathbf{y}_i - \mathbf{y}_{p_i} = d^\Delta(t_i; \theta^\Delta), \text{ s.t. } p_i > 0\,. \tag{2}$$

The coding process then picks the code t_i and the parameters of the encoder θ^Δ so that $\mathbf{z}_i \approx \mathbf{x}_i - \mathbf{y}_{p_i}$.

Encoding process. The dataset can be encoded by minimizing the overall squared reconstruction error. This optimization task has the following formulation:

$$\begin{aligned} \underset{\theta^0,\theta^\Delta,P,T,\mathbf{y}}{\text{minimize}} \quad & \sum_{i:\, p_i=0} \|d^0(t_i;\theta^0) - \mathbf{x}_i\|^2 + \\ & \sum_{i:\, p_i>0} \|\mathbf{y}_{p_i} + d^\Delta(t_i;\theta^\Delta) - \mathbf{x}_i\|^2 \end{aligned} \tag{3}$$

$$\text{subject to } \mathbf{y}_i = \begin{cases} d^0(t_i;\theta^0), & \text{if } p_i = 0 \\ \mathbf{y}_{p_i} + d^\Delta(t_i;\theta^\Delta), & \text{if } p_i > 0 \end{cases}.$$

In (3) P denotes the set (vector) $\{p_1, p_2, \ldots, p_N\}$, which defines the topology of arborescences, T denotes the set of codes $\{t_1, t_2, \ldots, t_N\}$. Performing optimization (3) process is usually hard, only an approximate (local) minimum can be found, and a reasonable initialization procedure is usually required. In Section 4.1, we discuss the optimization for arborescence coding based on optimized product quantization scheme.

Decoding process. The approximations to the original dataset can be recovered by applying formulas (1) and (2). To decode all descriptors from a certain arborescence, we first decode the root descriptor using (1). We then proceed along the arborescence. For the ith descriptor, we take the reconstruction $\mathbf{y}_{p_i}$ of its parent, recover the displacement vector $\mathbf{z}_i$ using (2), and get the descriptor reconstruction as $\mathbf{y}_i = \mathbf{y}_{p_i} + \mathbf{z}_i$.

Fast search. For the majority of quantization schemes, search for descriptors with high scalar product or low squared distance to a certain query descriptor $\mathbf{q}$ does not require explicit decoding (1)-(2). Instead, the quantization schemes usually provide the way to quickly estimate the scalar product $\Pi_i^0(\mathbf{q}, t_i; \theta^0) = \langle \mathbf{q}, d^0(t_i;\theta^0)\rangle = \langle \mathbf{q}, \mathbf{y}_i\rangle$ using look-up tables precomputed once for the given $\mathbf{q}$ and the parameters θ^0 [14]. Likewise, the scalar product $\Pi_i^\Delta(\mathbf{q}, t_i; \theta^\Delta) = \langle \mathbf{q}, d^\Delta(t_i;\theta^\Delta)\rangle = \langle \mathbf{q}, \mathbf{z}_i\rangle$ between the query and the encoded displacement $\mathbf{z}_i$ can be estimated

without the explicit reconstruction of $\mathbf{z}_i$. The scalar product between the query and the encoded vectors can then be quickly evaluated by traversing an arborescence from the root to the leaves:

$$\langle \mathbf{q}, \mathbf{y}_i \rangle = \begin{cases} \Pi_i^0(\mathbf{q}, t_i; \theta^0), & \text{if } p_i = 0 \\ \langle \mathbf{q}, \mathbf{y}_{p_i} \rangle + \Pi_i^\Delta(\mathbf{q}, t_i; \theta^\Delta), & \text{if } p_i > 0 \end{cases} . \quad (4)$$

The squared Euclidean difference can then be estimated using the formula

$$\|\mathbf{q} - \mathbf{y}_i\|^2 = \|\mathbf{q}\|^2 + \|\mathbf{y}_i\|^2 - 2\langle \mathbf{q}, \mathbf{y}_i \rangle . \quad (5)$$

While the third term in (5) can be estimated using (4), the scalar term $\|\mathbf{y}_i\|^2$ is trickier to handle. In practice, we store a coarse estimate of $\|\mathbf{y}_i\|^2$ (one-byte quantization) along with (inside) every descriptor code t_i. Note that some of the quantization schemas (such as additive quantization [3], tree quantization [5]) have to store such estimate anyways (if fast nearest neighbor search is desired), so that the requirement to store the quantized squared norm does not incur additional memory costs over these schemes.

Memory overhead. In general, the memory footprint of arborescence coding consists of the parameters θ^0 and θ^Δ, the codes t_i, and the topology information, which is needed to infer the parents p_i during decoding or fast search. The memory spent on the topology information thus constitutes overhead over the base quantization scheme. If stored directly, the indices p_i take $\lceil \log_2 n \rceil$ bits each, which is significant for many interesting scenarios.

Fortunately, simple tricks can allow to store arborescence topology at a much lower cost. For that, the descriptors can be re-ordered, so that arborescences are stored sequentially (descriptors forming the same arborescence are stored contiguously). Furthermore, within each arborescence, the descriptors can be reordered following breadth-first order. Then to recover the topology it is sufficient to store the number of children with every descriptor. These numbers follow a very low-entropy distribution (upto 2-3 bits in all our experiments), which is a very low overhead compared to reasonable code sizes for most practical purposes.

For star coding, the overhead can be made negligible as follows. We store stars contiguously, further ordering stars by the number of elements in them. Then, the only information that needs to be stored in order to recover the topology is the maximal star size and the number of stars of each size, which is at most few hundred bytes per dataset for any reasonable dataset.

4. Arborescence coding using OPQ

We now discuss a particular instantiation of arborescence coding when the decoders (1) and (2) use *optimized product quantization* (OPQ) [12, 18].

We first recap OPQ using the notation introduced above. In the OPQ scheme, a vector is encoded as a rotated concatenation of M codewords coming from M different codebooks. The parameters for the decoders can thus be written as: $\theta^0 = \{R^0, C_1^0, C_2^0, \dots, C_M^0\}$ and $\theta^\Delta = \{R^\Delta, C_1^\Delta, C_2^\Delta, \dots, C_M^\Delta\}$, where R^0 and R^Δ are the $D \times D$ orthogonal matrices, and each of the codebooks C_j^0, C_j^Δ contains K codeword D/M-dimensional vectors. We denote $\mathbf{c}_{j,k}^0$ the k-th codeword in the j-th codebook for the first decoder. In our implementation, we keep the two rotation matrices the same: $R = R^0 = R^\Delta$. Using two different rotation matrices is possible, but leads to more complex encoding process.

The code t_i for each vector is then an M-tuple of codeword indices $t_i^1, t_i^2, \dots, t_i^M$ in the respective codebooks, each of them being an integer between 1 and K. The decoding of root descriptors (1) then takes the form:

$$\mathbf{y}_i = R\,[\mathbf{c}_{1,t_i^1}^0, \mathbf{c}_{2,t_i^2}^0, \dots, \mathbf{c}_{M,t_i^M}^0], \text{ if } p_i = 0\,, \quad (6)$$

where square brackets denote concatenation. Likewise, the decoding of displacements (2) then takes the form:

$$\mathbf{z}_i = R\,[\mathbf{c}_{1,t_i^1}^\Delta, \mathbf{c}_{2,t_i^2}^\Delta, \dots, \mathbf{c}_{M,t_i^M}^\Delta], \text{ if } p_i > 0\,, \quad (7)$$

The fast search procedure discussed above then uses the look-up tables $L^0(j,k) = \langle R^T\mathbf{q}, \mathbf{c}_{j,k}^0 \rangle$ and $L^\Delta(j,k) = \langle R^T\mathbf{q}, \mathbf{c}_{j,k}^\Delta \rangle$ that are precomputed for a given query (after rotating it by $R^T{=}R^{-1}$). Using these tables, the scalar product of the query $\mathbf{q}$ with $\mathbf{y}_i$ (for root nodes) or $\mathbf{z}_i$ (for non-root nodes) can be evaluated by M look-ups in the tables and $M-1$ scalar additions.

4.1. Optimizing the encoding

We now discuss the encoding process (3) in the case of OPQ. The following groups of variables are part of the optimization: the arborescence topology P, the code tuples T, the codebooks C, and the rotation matrix R. As common in the quantization schemes, optimization proceeds by alternations (group-coordinate descent). At each update stage, one group of variables is updated, while others are kept fixed. We now go through the update stages.

Updating rotation. The updates for rotation can be performed by finding optimal rotation δR of the dataset X that minimizes the squared distance (3) and applying the update $R := \delta R^T R$ to the current rotation. The update δR can be found using Procrustes analysis as detailed in [12, 18]. In the remaining updates,we can remove matrix R from consideration by applying the rotation R^T to the dataset X, effectively reducing our quantizers to unoptimized product quantizers [14]. We apply this trick to simplify the derivations below.

Updating codebooks. When all other variables are fixed, $\mathbf{y}_i$ can be expressed as a linear function of the codeword entries. Consider the f-th dimension in the recon-

struction $\mathbf{y}_i$. Let us assume that it corresponds to the l-th dimension in the m-th chunk of dimensions that OPQ-splits the D dimensions into. In other words, let $f = l + (m-1)\frac{D}{M}$. Then the f-th dimension of the reconstruction $\mathbf{y}_i$ can be found as:

$$\mathbf{y}_i[f] = \mathbf{c}^0_{m,t^m_{r(i)}}[l] + \sum_{j \in r(i) \to i} \mathbf{c}^{\Delta}_{m,t^m_j}[l]\,, \qquad (8)$$

where square brackets denote the dimension indexing, $r(i)$ denotes the root index of the arborescence that the i-th descriptor belongs to, and $r(i) \to i$ denotes the set of indices in the path from the root to the i-th descriptor (excluding the root).

Plugging the unrolled expression (8) into the objective (3) results in a least-squares problem over the entries of the codebooks. The problem decomoses over different dimensions, with each of the resulting D least-squares problems having $2K$ variables ($\mathbf{c}^0_{m,k}[l]$ and $\mathbf{c}^{\Delta}_{m,k}[l]$ for $k \in 1..K$). Solving these problems then gives the optimal codebook entries (given the other variables fixed).

Updating topology and codes. Finally, we discuss the updates to the topology (i.e. the variables p_i and the codes t_i). We perform these updates sequentially, at each time changing the variables p_i and t_i for a single i. In other words, we iterate over descriptors one-by-one, and allow each of them to improve its reconstruction error by simultaneously choosing a different parent and encoding the displacement to this parent or becoming a root and encoding its absolute position.

The change of p_i and/or t_i changes the reconstruction $\mathbf{y}_i$, which also results in the change of reconstructions for all descendants in the arborescence. Since we want to perform updates efficiently and with the guarantee that the squared reconstruction error does not increase, we skip all descriptors with children during the updates.

To further speed-up the updates, for i-th descriptor $\mathbf{x}_i$ we only consider $k = 20$ descriptors with the closest reconstruction $\mathbf{y}_j$ as potential parents. When performing star quantization, we only consider descriptors that are currently roots. For each potential parent j, we consider the vector $\mathbf{x}_i - \mathbf{y}_j$, assess the error of its optimal product quantization using codebooks C^{Δ}, assess the error of product quantization of $\mathbf{x}_i$ using the codebook C^0, and pick the encoding variant with the smallest error.

4.2. Initializating the encoding

The iterative updates discussed above are guaranteed to not increase the reconstruction error, and given time will converge to a certain configuration. This configuration, however, is not guaranteed and most certainly will not be a global minimum to the reconstruction error. Therefore, the accuracy of the resulting encoding depends on the initialization.

We use the following initialization approach. We initialize all parent variables p_i to zero making them roots, and initialize all other variables by effectively performing OPQ. At this point, our reconstruction corresponds to OPQ. Since the reconstruction error is guaranteed to not increase in the further optimization steps and in subsequent optimization updates, the squared error of arborescence coding (or star coding) is guaranteed to be same or lower compared to OPQ.

We then initialize the codebooks C^{Δ} by running product quantization on the random subset of displacements from $\mathbf{y}_j$ to $\mathbf{x}_i$, such that $\mathbf{y}_j$ is one of the $k = 20$ nearest neighbors of $\mathbf{x}_i$ (among all reconstructions $\mathbf{y}$).

Finally, we update the parameters p_i and t_i as discussed in Section 4.1 with one additional heuristics. During the first update only, we visit the descriptors in the order of increasing OPQ reconstruction error. During this traversal, we prohibit descriptor to choose parents among yet unvisited descriptors, which have higher OPQ reconstruction error. As a result, every descriptor is childless by the moment it is visited, which gives it an opportunity to choose a parent (among more "affluent" descriptors with lower reconstruction error) and to decrease its own reconstruction error (recall, that in our optimization algorithm discussed in Section 4.1 only childless descriptors are allowed to switch parents).

5. Experiments

In this section, we present experimental evaluation of arborescence coding and star coding. In the experiments, we encode the datasets using the new coding schemes, as well as several baselines. In the majority of the experiments and unless noted otherwise, we simplify the experimental setup and optimize parameters directly on the "test" dataset rather than "learning" them on a hold-out dataset of similar nature. While we do not evaluate generalization capabilities, we still aim to compare methods with similar number of encoding parameters, making our comparisons valid. In the final experiment, we demonstrate that the relative performance of coding schemes remains approximately the same, when coding parameters are learned on the hold-out dataset.

Datasets. We consider the following four datasets:

- *SIFT1M* [14] is a dataset of million SIFT vectors [17], which are highly structured gradient-based descriptors, extracted from natural image keypoints with the hold-out set of 10.000 queries.
- *DEEP1M* and *DEEP10M* datasets contain deep descriptors, which are computed from the activations of deep neural networks. In general, deep descriptors are emerging as the new state-of-the-art in retrieval. Here, we use the first million and the first ten million vectors from the dataset of 96-dimensional deep descriptors introduced in [6].

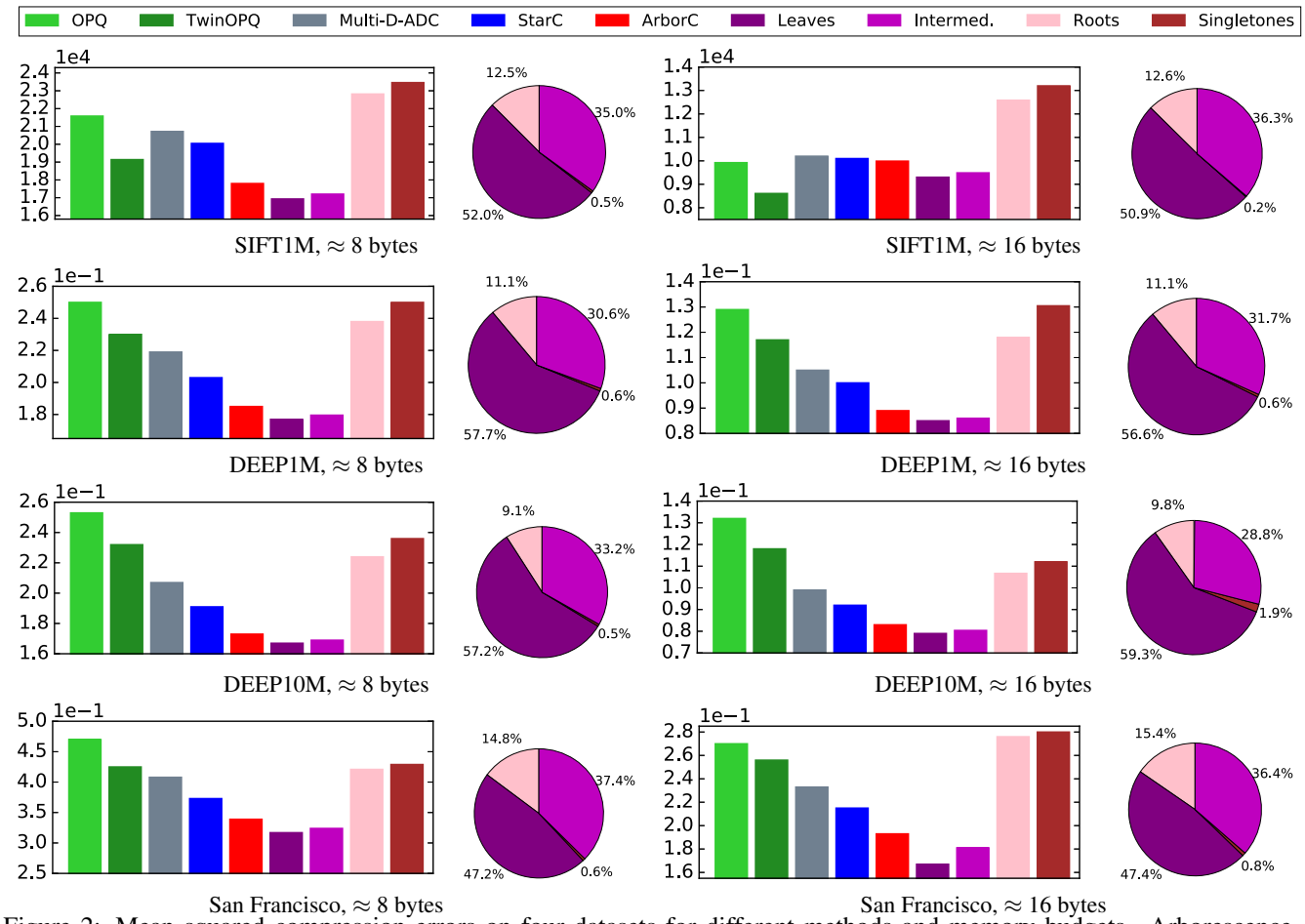

Figure 2: Mean squared compression errors on four datasets for different methods and memory budgets. Arborescence coding (red) provides considerably smaller errors comparing to the baselines except for SIFT1M (16 bytes), where TwinOPQ performs best. We also show the average compression errors for different classes of nodes within arborescence coding (leaves with parents, intermediate nodes that have both a parent and children nodes, roots with children, singletons). Compression errors for the leaves and the intermediate nodes are much smaller than for the singletones and the roots. The most of the points in arborescence coding are the leaves or the intermediate nodes (see the distributions of classes in the pie charts), which leads to arborescence coding having smaller compression error overall.

- *SFLD* (San Francisco Landmark dataset) [9] contains a database of 1.7 million images of buildings in San Francisco collected with a vehicle-mounted camera. We compute 128-dimensional deep SPoC descriptor [4] for all images.

Coding methods. We evaluate arborescence and star codings introduced in this paper. We invariably use the size of codebooks $K = 256$ as is done in most other quantization works, since it lead to small look-up tables and convenient one-byte code entries $t_{i,k}$. We consider two different codebook numbers $M = 8, 16$ (much bigger M is less interesting, because the performance of all methods start to saturate, and extremely small M such as $M = 4$ leads to an impractically poor compression). The size of the codes t_i is thus either 8 or 16 bytes, plus a few bits (less than one byte) needed to encode the number of children in the case of general arborescence coding (but not star coding). On top of that, an additional byte is needed if fast nearest neighbor search using (5) is to be performed.

We also consider three baselines. Our first baseline is "vanilla" optimized product quantization (*OPQ*) [12, 18] with the same number of codebooks M and the same codebook size K leading to same code length as star coding (although fast NN search does not need length encoding in this case). Our second baseline (*TwinOPQ*) is a variant of OPQ that uses two sets of codebooks (sharing the same rotation matrix), so that each descriptor is encoded by one of the two sets. At each iteration, after the codebooks and the rotation matrix are re-estimated, a descriptor can switch to the other codebook set, if such switch results in a lower compression error. TwinOPQ has the same number of learnable parameters as our systems (apart from the arborescence structure).

Our third and strongest baseline is the *Multi-D-ADC* [2], which has clear similarities with star coding (as well as

	≈ 8 bytes per vector					≈ 16 bytes per vector				
Method	R@1	R@4	R@16	R@64	R@256	R@1	R@4	R@16	R@64	R@256
SIFT1M										
OPQ	0.241	0.469	0.724	0.904	0.983	0.463	0.776	0.949	0.995	0.999
Twin OPQ	0.292	0.545	0.787	0.944	0.991	**0.506**	**0.812**	0.965	0.997	0.999
Multi-D-ADC	0.282	0.530	0.772	0.935	0.988	0.464	0.763	0.947	0.996	0.999
StarC	0.307	0.556	0.802	0.947	0.992	0.484	0.794	0.964	0.997	1.0
ArborC	**0.316**	**0.587**	**0.823**	**0.957**	**0.997**	0.487	0.798	**0.966**	**0.998**	**1.0**
DEEP1M										
OPQ	0.161	0.351	0.610	0.839	0.959	0.356	0.658	0.895	0.984	0.999
Twin OPQ	0.182	0.398	0.660	0.872	0.971	0.372	0.689	0.916	0.989	0.999
Multi-D-ADC	0.203	0.434	0.693	0.894	0.978	0.406	0.729	0.935	0.994	0.999
StarC	0.223	0.460	0.725	0.915	0.987	0.408	0.737	0.941	0.994	0.999
ArborC	**0.243**	**0.485**	**0.750**	**0.924**	**0.989**	**0.421**	**0.757**	**0.947**	**0.995**	**0.999**
DEEP10M										
OPQ	0.134	0.270	0.466	0.694	0.875	0.309	0.570	0.816	0.948	0.992
Twin OPQ	0.151	0.305	0.516	0.736	0.903	0.331	0.600	0.841	0.960	0.993
Multi-D-ADC	0.188	0.362	0.587	0.805	0.939	0.370	0.657	0.891	0.980	0.998
StarC	0.206	0.394	0.629	0.837	0.957	0.388	0.681	0.899	0.984	0.998
ArborC	**0.212**	**0.414**	**0.656**	**0.863**	**0.964**	**0.404**	**0.705**	**0.911**	**0.985**	**0.998**

Table 1: Euclidean nearest neighbor search accuracy based on different compression methods for three datasets with the different code lengths. The standard Recall@k measure (the probability of the true nearest neighbor being retrieved) is used to compare the compression methods. Arborescence coding performance is uniformly higher than for the baselines on datasets of deep features, while on SIFT1M (16 bytes) TwinOPQ is better for small k.

an earlier IVFADC system [14]). Multi-D-ADC first uses "coarse-level" OPQ with $M' = 2$ large codebooks ($K' = 512$ for SIFT1M and DEEP1M, $K' = 1024$ for DEEP10M, and $K' = 512$ for SFLD). K' was chosen to allow a slightly more memory for the size of the coarse-level table in Multi-D-ADC than the amount of memory spent on arborescence topologies in arborescence coding. Each descriptor is then assigned to the closest "centroid" out of K'^2 variants corresponding to different combinations of coarse-level codewords, and then the displacement w.r.t. the centroid is encoded using product quantization (in the rotated system) with the same M and K as in our method. If fast exhaustive ANN search through the dataset is desired then Multi-D-ADC also requires storing the quantized length with each descriptor (an alternative is to either re-compute look-up tables for each visited non-empty cell or to store the tables of scalar products between coarse-level codebooks and fine-level codebooks). Overall, the memory footprint of Multi-D-ADC in our comparisons is very close to the footprint of arborescence coding and slightly larger than the footprint of star coding.

5.1. Results

We compare different compression schemes using the following two metrics. The *mean squared reconstruction error* (MSRE) directly measures the reconstruction accuracy attained by different methods. For each dataset, we also consider a hold-out set of query vectors, and consider how well is the nearest neighbor search in the compressed dataset able to recover the true nearest neighbor. As is common for this task, we report *recall@k* measure (for $k = 1, 4, 16, 64, 256$), which is the probability that the true nearest neighbor is among k closest neighbors in the compressed dataset. Two compression levels ($\approx 8, 16$ bytes per vector) were evaluated.

Compression error. The average compression errors on four datasets obtained by the different coding methods are presented in Figure 2. Star coding and arborescence coding provide significant improvements in the encoding accuracy. The improvement over baselines are uniform everywhere except the setting $M = 16$ for SIFT-1M. In particular, the compression error is reduced by upto 20% on deep datasets. On the dataset of SIFT descriptors the TwinOPQ baseline provides smaller error, that, probably, reflects the fact that the SIFT data is a favourable case for (O)PQ methods due to its histogram-based construction process. We also demonstrate the average compression errors for different classes of points in arborescence coding. Each point in arborescence coding belongs to the one of four classes depending on their role in the arborescence structure. The *singleton* points do not have a parent node and children nodes. The *roots* have children nodes and do not have parent nodes. The *leaves* do not have children nodes (but have parents) and the *intermediate* nodes have both a parent and children nodes. The distributions of descriptors over classes are shown in the pie charts in Figure 2. Note, that the leaves and the intermediate nodes are compressed with much smaller errors than the nodes without parents. Interestingly, the compression errors for singletons and roots can be higher than average baseline errors, but as their numbers is relatively small (about 10-15%) the encoding accuracy of the whole dataset is higher.

Approximate nearest neighbor search. Here we evaluate different coding schemes for nearest neighbor search in compressed databases. The recall@k values obtained with

	≈ 8 bytes per vector						≈ 16 bytes per vector					
Method	R@1	R@2	R@5	R@10	R@20	R@50	R@1	R@2	R@5	R@10	R@20	R@50
San Francisco Landmark												
Uncompressed	*0.305*	*0.359*	*0.425*	*0.457*	*0.509*	*0.585*	*0.305*	*0.359*	*0.425*	*0.457*	*0.509*	*0.585*
Twin OPQ	0.164	0.215	0.289	0.349	0.407	0.499	0.242	0.299	0.377	0.436	0.489	0.567
Multi-D-ADC	0.167	0.221	0.297	0.353	0.407	0.506	0.245	0.300	0.379	0.437	0.497	0.571
StarC	0.168	0.225	0.304	0.355	0.410	0.503	0.253	0.306	0.385	0.441	0.502	0.575
ArborC	**0.183**	**0.240**	**0.318**	**0.359**	**0.413**	**0.509**	**0.260**	**0.312**	**0.393**	**0.447**	**0.509**	**0.581**

Table 2: The average recall w.r.t. ground truth matches obtained with retrieval from the San Francisco database compressed with different methods and code lengths. Images in the database are presented by 128-dimensional SPoC descriptors[4]. The performance of arborescence coding is uniformly higher than for baselines for both memory budgets and all lengths of candidate lists.

different compression schemes of the search databases are presented in Table 1. As can be observed, the higher encoding accuracy results in higher search performance. Arborescence coding provides considerable improvement over baselines for deep descriptors and perform best in general except for SIFT-1M (16 bytes), where the TwinOPQ baseline is better for small k.

Landmark recognition. We also apply the proposed methods to the problem of visual localization. We compressed the set of SPoC descriptors of SFLD database images with different coding schemes and produce the list of candidate matches for each of the uncompressed query images. Then for different methods we compare the mean recall w.r.t. the ground truth matches that are hand-labeled for each query. In this experiment the database contains both PCI and PFI images and GPS data is not used (for more details see the protocol in [10]). The recall values for different number of candidates are presented in Table 2. The advantage of arborescence coding is uniform for both compression levels and different lengths of candidate lists.

Timings. The most computationally expensive part of the AnnArbor encoding is the calculation of the topology and the codes (corresponding to the variables p_i and t_i) given codebooks and rotation matrix. In our experiments the encoding of one million points with unoptimized Python code requires 14 minutes on Xeon E5 CPU. The update of codebooks and rotation matrix during learning is typically much faster, e.g. on SIFT1M/DEEP1M one update requires four minutes. These timings are obtained with the single-thread implementation of the initialization procedure (section 4.2) while the other parts use 30 CPU threads.

Hold-out set encoding. Finally, to confirm the ability of the AnnArbor scheme to generalize to new datasets, when the parameters are learned on hold-out data, we performed the following experiment. We took the last one million points from the DEEP10M dataset and encoded them with the parameters obtained by training on DEEP1M (8 bytes) that does not overlap with the test set. The results in the Table 3 demonstrate that the MSRE increase on the hold-out set for Star and Arborescence Coding is on par with the baselines and the relative-performance of the methods on the hold-out set is the same as on the train set.

Method	OPQ	TwinOPQ	Multi-D-ADC	StarC	ArborC
In-sample MSRE	0.250	0.231	0.219	0.203	0.185
Out-of-sample MSRE	0.257	0.238	0.223	0.208	0.192

Table 3: The encoding mean-squared reconstruction error obtained with the different coding schemes trained on the test and hold-out sets. The first row corresponds to the setup where the coding parameters are trained on the DEEP1M and the same dataset is encoded. The second row presents MSRE obtained by the coding schemes trained on DEEP1M, while the test set consists of the last one million points from the DEEP10M which does not overlap with DEEP1M. Eight bytes encoding is used in both setups. The relative performance of coding schemes is the same between the two lines.

6. Discussion

We have presented a new descriptor coding scheme (arborescence coding) and its variant (star coding). The new schemes can be implemented on top of almost any of the existing quantization methods (and, in fact, almost any vector compression methods), and are able to reduce the coding error of the underlying method considerably by arranging descriptors into arborescence graphs and coding the relative displacements rather than absolute positions. The encoded datasets still permit efficient search using look up tables, while the memory overhead that is required to store the topology of the arborescences is very small.

To the best of our understanding, the source of the considerable reduction of error within arborescence coding is the ability of descriptors to choose their parents among large pools of potential parents. Even though the distribution of displacements between neighbors may not be much easier to model than the distribution of absolute positions (as even nearest neighbors in the high-dimensional space are usually far away), each descriptor can get encoded with low error if only one neighbor (not necessarily the nearest one) corresponds to the displacement with low quantization error.

Future work involves implementations of arborescence coding and star coding on top of other quantization schemes, as well as combination of arborescence coding with indexing approaches.

Acknowledgement. VL is supported by the MES RF grant 14.756.31.0001.

References

[1] R. Arandjelovic and A. Zisserman. Extremely low bit-rate nearest neighbor search using a set compression tree. *TPAMI*, 36(12):2396–2406, 2014.

[2] A. Babenko and V. Lempitsky. The inverted multi-index. In *Proc. CVPR*, 2012.

[3] A. Babenko and V. S. Lempitsky. Additive quantization for extreme vector compression. In *Proc. CVPR*, 2014.

[4] A. Babenko and V. S. Lempitsky. Aggregating deep convolutional features for image retrieval. In *Proc. ICCV*, 2015.

[5] A. Babenko and V. S. Lempitsky. Tree quantization for large-scale similarity search and classification. In *Proc. CVPR*, 2015.

[6] A. Babenko and V. S. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proc. CVPR*, 2016.

[7] S. Battiato, G. Gallo, G. Impoco, and F. Stanco. An efficient re-indexing algorithm for color-mapped images. *IEEE Transactions on Image processing*, 13(11):1419–1423, 2004.

[8] V. Chandrasekhar, Y. Reznik, G. Takacs, D. M. Chen, S. S. Tsai, R. Grzeszczuk, and B. Girod. Compressing feature sets with digital search trees. In *Proc.ICCV Workshops*, 2011.

[9] D. M. Chen, G. Baatz, K. Köser, S. S. Tsai, R. Vedantham, T. Pylvänäinen, K. Roimela, X. Chen, J. Bach, M. Pollefeys, B. Girod, and R. Grzeszczuk. City-scale landmark identification on mobile devices. In *Proc. CVPR*, 2011.

[10] Y. Chen, T. Guan, and C. Wang. Approximate nearest neighbor search by residual vector quantization. In *Sensors*, 2010.

[11] V. Cuperman and A. Gersho. Vector predictive coding of speech at 16 kbits/s. *IEEE Transactions on Communications*, 33(7):685–696, 1985.

[12] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Proc. CVPR*, 2013.

[13] V. Gripon, M. Rabbat, V. Skachek, and W. J. Gross. Compressing multisets using tries. In *Information Theory Workshop (ITW)*, 2012.

[14] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33(1), 2011.

[15] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *vldb*, 2004.

[16] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proc. CVPR*, 2014.

[17] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[18] M. Norouzi and D. J. Fleet. Cartesian k-means. In *Proc. CVPR*, 2013.

[19] Y. A. Reznik. Coding of sets of words. In *Proc. DCC*, 2011.

[20] R. Salakhutdinov and G. Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009.

[21] L. R. Varshney and V. K. Goyal. Ordered and disordered source coding. In *Proc. UCSD Workshop Inform. Theory Its Applications*, 2006.

[22] T. Zhang, C. Du, and J. Wang. Composite quantization for approximate nearest neighbor search. In *Proc. ICML*, 2014.