Tree Quantization for Large-Scale Similarity Search and Classification

Artem Babenko
Yandex, Moscow
National Research University
Higher School of Economics
arbabenko@yandex-team.ru

Victor Lempitsky
Skolkovo Institute of Science and Technology
(Skoltech)
lempitsky@skoltech.ru

Abstract

We propose a new vector encoding scheme (tree quantization) that obtains lossy compact codes for high-dimensional vectors via tree-based dynamic programming. Similarly to several previous schemes such as product quantization, these codes correspond to codeword numbers within multiple codebooks. We propose an integer programming-based optimization that jointly recovers the coding tree structure and the codebooks by minimizing the compression error on a training dataset. In the experiments with diverse visual descriptors (SIFT, neural codes, Fisher vectors), tree quantization is shown to combine fast encoding and state-of-the-art accuracy in terms of the compression error, the retrieval performance, and the image classification error.

1. Introduction

As very large datasets of high-dimensional vectors proliferate, machine learning, computer vision, and information retrieval systems that work with such datasets increasingly rely on lossy vector compression or hashing schemes. A crucial requirement for these schemes is the ability to evaluate distances and scalar products between compressed and uncompressed vectors efficiently and without explicit decompression. At the moment, systems that rely on the *product quantization* compression [10] are often preferred to hashing approaches due to a more favourable memory-accuracy tradeoff as evidenced by comparisons in e.g. [8, 15].

Given a dataset of vectors in $\mathcal{R}^D$, product quantization starts by splitting the vector dimensions into M groups. Each dimension group is then quantized separately and independently from others using codebooks of small size (most often 256 codewords), whereas codewords in the codebooks have the dimension D/M. In the PQ compression scheme, an input vector is approximated as a concatenation of M codewords (one codeword from each codebook). Product quantization implicitly relies on the limited amount of correlation between the dimension groups, since each codebook is learned independently from others. The encoding process within PQ is very simple and fast, and the computation of scalar product and distances between a large number of PQ-compressed vectors and an uncompressed vector can be implemented very efficiently using look-up tables.

Recently, we have proposed an alternative compression scheme called *additive quantization* (AQ) that pushes the coding accuracy of PQ-based methods even further [3]. Similarly to PQ, AQ maintains a set of M codebooks. However, the codewords within the codebooks are full-length, i.e. D-dimensional. During the compression stage AQ represents a vector as a sum of M codewords (one codeword from each codebook). The vector code is thus the same as within PQ, i.e. M codeword numbers. The additive nature of the compression means that the evaluation of scalar products between AQ-compressed and uncompressed vectors can use the same look-up table trick and is thus very fast. Evaluation of Euclidean distances takes slightly more time or an extra byte of memory but is still efficient. In general, AQ achieves a significant boost in coding accuracy (for the same code length) over PQ, which can be explained by the lack of low-correlation assumption between dimension groups. Furthermore, AQ codebooks possess an increased number of parameters that can be adjusted at the codebook learning stage in order to fit the data distribution.

The main limitation of the AQ-compression is the inefficiency of the encoding step. As we show in [3], finding the optimal combination of the codebook vectors is equivalent to the MAP-inference in the fully-connected Markov random field with unstructured and highly non-submodular pairwise potentials. As reported in [3] none of the standard MRF optimization methods [12] work well and there-

fore [3] uses a heuristics-driven beam search, which is able to find approximate codings resulting in lower coding error than PQ-compression. Still, this approximate inference takes orders of magnitude more time than PQ encoding, and can be prohibitively slow for many practical applications, especially when online encoding of new vectors is needed.

Here, we propose a new coding scheme called Tree Quantization (TQ) that belongs to the same family as PQ and AQ. Similarly to PQ and AQ, TQ maintains a set of M codebooks and, similarly to AQ, it encodes a vector as a sum of M codewords from different codebooks. The TQ-code for a vector is thus, once again, a set of M codeword numbers. The difference from AQ lies in the special structure that TQ imposes onto its codebooks. The encoding is based on a tree graph (the *coding tree*), where vertices correspond to codebooks, while each of the D dimensions is assigned to an edge. Each codebook then encodes only the dimensions that are assigned to edges that are incident to the vertex corresponding to this codebook (Figure 1). All other dimensions are then fixed to zero for all codewords in a given codebook.

By construction, the encoding process within the tree quantization is implemented via the MAP-inference in a tree-shaped model, and is therefore exact and efficient [16]. Perhaps the most interesting part of the TQ scheme is the codebook learning stage. Standard quantization, product quantization, and additive quantization all use k-means like processes to learn their codebooks, which alternate the encoding steps ("E-steps") with the codebook-reestimation steps, during which the codebook assignments of the training vectors are kept fixed ("M-steps"). Crucially, we demonstrate that TQ can follow the same scheme, and that given the codebook assignments of the training vectors, it is possible to estimate (i) the tree structure, (ii) the dimensions to edges assignments, and (iii) the codewords, all jointly and in a globally optimal way. Such global estimation during the M-step requires solving an integer linear program (ILP), which in our experiments was always solvable to optimality using a modern ILP solver [1] for D and M that were typically used in previous works.

We evaluate the tree quantization scheme in terms of coding errors as well as within the contexts of the nearest neighbor search (matching uncompressed queries to a compressed dataset) and classification (where either the training or the test sets are compressed). We compare TQ with several methods, namely PQ and AQ, the "optimized" versions of PQ and TQ, which additionally estimate a global rotation of the data that optimizes the coding accuracy [15, 8] and with the recent Composite Quantization (CQ) method [19] which approximates a vector as a sum of several codewords with fixed pairwise scalar products. Overall, the global optimality of the TQ encoding (given the coding tree) as well as the global optimality of the extended M-step within the coding tree learning, allowed TQ to achieve coding error, recall, and classification accuracy that were similar to the AQ encoding and much better than the PQ encoding. While achieving similar coding accuracy, TQ outperformed AQ by a large margin in terms of the encoding time.

2. Tree quantization

In this section, we first discuss the representation employed by the tree quantization. We then briefly discuss how this representation facilitates fast scalar product and Euclidean distance computations between uncompressed and compressed vectors in a way that is similar to product and additive quantizations.

2.1. Coding tree

We assume that we are dealing with vectors in the $D-$dimensional space $\mathcal{R}^D$, and that vectors are to be encoded with M codebooks $C^1, C^2, \ldots, C^M$. Each codebook has K vectors (*codewords*) and in our experiments, as well as in most previous works on PQ and AQ, K is fixed to 256. We denote $c^m(i)$ to be the ith codeword in the mth codebook, and $c^m(i)[d]$ to be the dth entry (dimension) of this vector. In TQ, each vectors is encoded using M numbers between 1 and 256, i.e. M bytes, corresponding to codeword numbers in each of the codebooks. TQ (as well as AQ) approximates a vector x with the code $[i_1, i_2, \ldots, i_M]$ as a sum of the corresponding codewords:

$$x \approx \sum_{m=1}^{M} c^m(i_m), \;\; i_m \in 1..K \tag{1}$$

Unlike AQ that does not impose any structure on the codewords, TQ uses the *coding tree* (Figure 1) to impose such structure. The coding tree $\mathcal{T}$ is a tree graph with M vertices, where each vertex corresponds to a codebook. We further use the notation $(m, n) \in \mathcal{T}$ for $m \in 1..M$ and $n \in 1..M$ to denote the fact that the mth and the nth vertices are connected by a tree edge. Given the tree $\mathcal{T}$, we assign each of the D dimensions in $\mathcal{R}^D$ to one of the edges in the tree. We denote with $\mathcal{D}_{m,n}$ the set of dimensions that are assigned to an edge $(m, n) \in \mathcal{T}$. Since each dimension is assigned to one edge, these sets are disjoint. Here and below, when using (m, n) as an index, we do not distinguish between (m, n) and (n, m) as the coding tree is undirected.

We further define $\mathcal{D}_m$ to be the union of all dimension sets for edges that are incident to the vertex m, i.e.:

$$\mathcal{D}_m = \{ \bigsqcup \mathcal{D}_{m,n} \,|\, n \in 1..M, \; (m, n) \in \mathcal{T} \} \,. \tag{2}$$

TQ requires all codewords in the mth codebook to have dimensions that are not in $\mathcal{D}_m$ to be zero, i.e:

$$\forall i, d \notin \mathcal{D}_m, \; c^m(i)[d] = 0 \,. \tag{3}$$

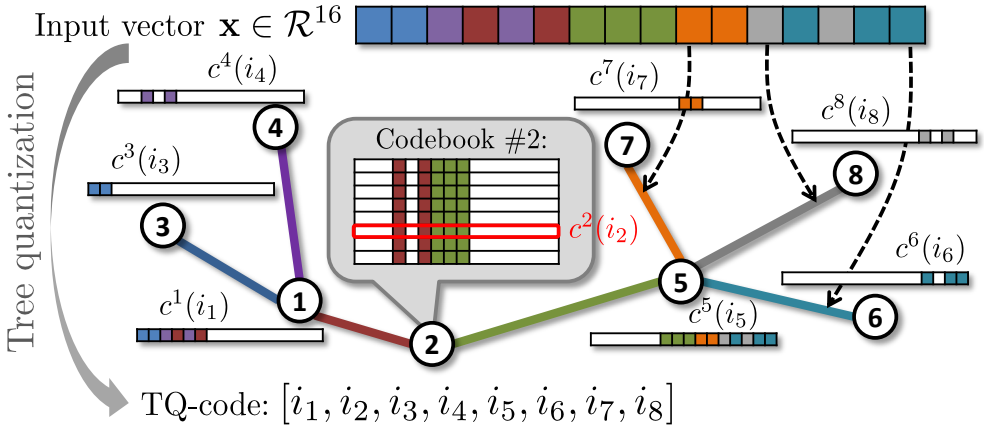

Figure 1. **Tree quantization** encoding for D-dimensional vectors (here D=16). Each dimension is assigned to an edge of the **coding tree** (the assignment is color-coded). Each of the M=8 vertices of the coding tree contains a codebook (shown for vertex #2). Each codebook encodes dimensions from the incident edges (also color-coded). An input vector $\mathbf{x}$ is then represented as a sum of M codewords $c^t(i_t)$ from vertex codebooks (shown as rectangles with active dimensions color-coded; the position of the codeword $c^2(i_2)$ within the second codebook is highlighted in red).

As a result, each dimension is encoded by **two** codebooks corresponding to the end vertices of the edge the dimension is assigned to. This is different from PQ, where each dimension is coded by only one codebook and from AQ, where it is coded by all M codebooks. An important (for TQ) consequence of the codebook structure imposed by TQ is the orthogonality of any pair of codewords coming from two codebooks that are not adjacent in the tree, i.e.:

$$\forall (m,n) \notin \mathcal{T}, \forall i,j: \quad \langle c^m(i), c^n(j) \rangle = 0\,. \tag{4}$$

2.2. Efficient operations

The representation (1) permits efficient operations between compressed and uncompressed vectors. The efficiency is attained through the use of look-up tables. The algorithms here are essentially the same as those employed by AQ, although the Euclidean distances can be evaluated more efficiently.

Generally, we consider an uncompressed *query* vector q and a dataset of L compressed vectors $x_1, \ldots, x_L$ with $L \gg K$. Each x_j is represented with the code $[i_1^j, i_2^j, \ldots i_M^j]$, i.e.:

$$x_j = \sum_{m \in 1..M} c^m(i_m^j)\,. \tag{5}$$

Then, the scalar product between q and x_j can be evaluated as:

$$\langle q, x_j \rangle = \sum_{m=1}^{M} \langle q, c^m(i_m^j) \rangle = \sum_{m=1}^{M} \Theta^m(i_m^j)\,, \tag{6}$$

where $\Theta^m(\cdot) = \langle q, c^m(\cdot) \rangle$ can be precomputed and stored, given the query q. The complexity of computing these look-up tables is independent of L and therefore for large enough datasets becomes negligible. Apart from this overhead, the complexity of computing a scalar product is M lookups and $M-1$ additions per each compressed vector.

When one considers the (squared) Euclidean distance between q and the compressed vectors, extra operations are needed. Since:

$$\|q - x\|^2 = \|q\|^2 - 2\langle q, x \rangle + \|x\|^2\,, \tag{7}$$

one also needs to evaluate $\|x_j\|^2$ for each compressed vector (the term $\|q\|^2$ can be precomputed once for all compressed vectors). Evaluating the norm $\|x_j\|^2$ based on the representation (1) can be done as follows:

$$\|x_j\|^2 = \|\sum_{m=1}^{M} c^m(i_m^j)\|^2 = \sum_{m=1}^{M} \|c^m(i_m^j)\|^2 + 2 \sum_{(m,n) \in \mathcal{T}} \langle c^m(i_m^j), c^n(i_n^j) \rangle \tag{8}$$

Here, we use the orthogonality (4) to eliminate all cross-terms for $(m,n) \notin \mathcal{T}$. To facilitate fast computation, the terms $\|c^m(\cdot)\|^2$ can be added to the values in the look-up tables $\Theta^m(\cdot)$ discussed above, while the cross-terms $\Theta_2^{m,n}(\cdot,\cdot) = 2\langle c^m(\cdot), c^n(\cdot) \rangle$ can be stored in separate query-independent look-up tables. The overhead of computing the Euclidean distance over the scalar product (and over the distance computation with the PQ compression) is then M look-ups and additions (i.e. about two times slower). Note that for AQ the same overhead is quadratic in M, i.e. much larger, since cross-terms for all (m,n) have to be looked up and summed.

3. Encoding and Learning

A tree quantizer is characterized by the tree $\mathcal{T}$ and the codebooks $C^m(k)$, that are consistent with the tree. In this

section, we discuss (1) how an optimal TQ code can be inferred for a given vector given the tree quantizer, and (2) how a tree quantizer can be learned in an unsupervised way from a training dataset of vectors.

3.1. Encoding

To find the optimal TQ-code $[i_1, i_2, \dots i_m]$ for a vector x given the codebooks $C^1 \dots C^M$, the reconstruction error E in the representation (1) is minimized:

$$E(i_1, i_2, \dots i_M) = \|x - \sum_{m=1}^{M} c^m(i_m)\|^2 \longrightarrow \min_{i_m} \quad (9)$$

Using (7), this function decomposes as:

$$E(i_1, i_2, \dots i_M) = \sum_{m=1}^{M} \left(-2\,\langle x, c^m(i_m)\rangle + \|c^m(i_m)\|^2\right) + \sum_{(m,n)\in\mathcal{T}} 2\,\langle c^m(i_m), c^n(i_n)\rangle, \quad (10)$$

where once again the orthogonality (4) is used to eliminate cross-terms for $(m, n) \notin \mathcal{T}$, and the constant term $\|x\|^2$ is also omitted.

The minimization (10) is then equivalent to the MAP-inference in the tree-shaped pairwise Markov random field defined by the coding tree, where the terms $U_m(i_m) = -2\langle x, c^m(i_m)\rangle + \|c^m(i_m)\|^2$ correspond to the unary terms. The terms $V_{m,n}(i_m, i_n) = 2\langle c^m(i_m), c^n(i_n)\rangle$ can be precomputed independently of the query and constitute the pairwise terms.

The inference can thus be performed using dynamic programming (max product algorithm) in the tree graph [16], which is exact and has the complexity $O(MK^2)$, while the precomputation of the unary terms has the complexity $O(KD)$. While the first term will typically be much larger and harder to vectorize, it is still much faster than the inference in the fully-connected model that has to be performed in the case of AQ [3] (e.g. the heuristic beam search algorithm proposed in [3] has the complexity $O(M^2K^2(M + \log MK))$). The exactness and the efficiency of the encoding process is the key advantage of TQ over AQ.

3.2. Codebook learning

We now focus on the task of learning a tree quantizer that is well adapted to a certain data distribution. We assume that a training dataset $X = \{x_1, x_2, \dots, x_L\}$ of L vectors is given, for which we minimize global reconstruction error over the codes and the tree quantizer parameters.

Let us introduce the assignment variables $A = \{a_{(m,n)}[d]\}$ that are binary indicator variables, i.e. $a_{(m,n)}[d] = 1$ iff the dimension d is assigned to the edge (m, n). Note that for each dimension d exactly one $a_{(m,n)}[d] = 1$. Recall that if a training example x_j has the code $[i_1, i_2, \dots, i_M]$ and if the dimension d is assigned to the edge (m, n) then $x_j[d]$ is approximated as $x_j[d] \approx c^m(i_j^m)[d] + c^n(i_j^n)[d]$. The global reconstruction error $G(X)$ over all examples can therefore be written as:

$$G(\{i_j^m\}, \{C^m\}, A; X) = \sum_{j=1}^{L}\sum_{d=1}^{D}\sum_{(m,n)\in\mathcal{F}} a_{(m,n)} \left(c^m(i_j^m)[d] + c^n(i_j^n)[d] - x_j[d]\right)^2 . \quad (11)$$

Here, $\mathcal{F}$ denotes the set of edges of a full graph on M vertices: $\mathcal{F} = \{(m, n) \mid 1 \le m < n \le M\}$.

To build the optimal quantizer, we need to minimize the functional G in (3.2) over all arguments, subject to the constraint that the assignment variables have to be consistent with some tree ($\exists\mathcal{T} : a_{(m,n)}[d] = 1 \Rightarrow (m, n) \in \mathcal{T}$). Similarly to all other quantization-based algorithms, we perform this minimization by k-means-like alternations. Thus, we alternate the minimization over the codes $\{i_j^m\}$ given the quantizer parameters $\{C^m\}$ and A ("E-step") and vice-versa ("M-step"). The minimization over the codes given the tree quantizer is equivalent to finding the optimal encoding for every training example, which has already been discussed in Section 3.1. Below, we focus on the M-step, i.e. optimizing G over $\{C^m\}$ and A given the codes $\{i_j^m\}$.

A key observation from (3.2) is that when the codes are given, the objective decomposes into the sum of the reconstruction errors for independent dimensions. Denote with $r_{(m,n)}[d]$ the reconstruction error for the dimension d that would accumulate over all training examples if this dimension is assigned to the edge (m, n). Assuming that the codeword entries $c^m(k)[d], c^n(k)[d]$ are set optimally for all $k = 1..K$ we have:

$$r_{(m,n)}[d] = \min_{\substack{c^m(\cdot)[d]\\ c^n(\cdot)[d]}} \sum_{j=1}^{L} \left(c^m(i_j^m)[d] + c^n(i_j^n)[d] - x_j[d]\right)^2 . \quad (12)$$

Finding the value $r_{(m,n)}[d]$ thus requires solving the least-square problem (12), which has a tall (L rows, $2K$ columns) and very sparse matrix (in each row there are two non-zero entries both equal to 1). For a given (m, n), the least-squares matrix is the same for all dimensions d as only the right-hand side differs across the LS problems. This can be additionally exploited when computing $r_{(m,n)}[d]$ for all $d \in 1..D$.

The codebook parameters can then be minimized out of the quantizer update, i.e. the M-step is reduced to the minimization over the assignment variables only:

$$\min_{\{C^m\},A} G(\{i_j^m\}, \{C^m\}, A; X) = \min_{A} \sum_{d=1}^{D} \sum_{(m,n)\in\mathcal{F}} a_{(m,n)}[d] \cdot r_{(m,n)}[d]\,. \quad (13)$$

Algorithm 3.1: TRAINTREEQUANTIZATIONCODEBOOKS()

input $X = \{x_1, \ldots, x_L\},\ M,\ K$
$\{a_{(m,n)}[d]\},\ \{C^m\},\ \{i_j^m\} = \text{Initialize}(X, M, K)$ // randomly or via PQ
repeat until convergence:
$\{C^m\} = \text{SolveLeastSquares}\left(\{a_{(m,n)}[d]\}, \{i_j^m\}\right)$ // see (12)
$\{a_{(m,n)}[d]\} = \text{SolveILP}\left(\{C^m\}, \{i_j^m\}\right)$ // see (14)-(19)
$\{i_j^m\} = \text{MaxProduct}\left(\{C^m\}, \{a_{(m,n)}[d]\}\right)$ // see (10)
output $\{a_{(m,n)}[d]\},\ \{C^m\}$

Figure 2. The pseudocode of the codebook learning process.

The minimization (3.2), once again, has to be subject to the assignments being consistent with some tree. Let us introduce the binary indicator variables $e_{(m,n)}$ that define whether (m, n) is included into the tree. The constrained minimization of (3.2) can then be formulated using the following binary integer linear program (ILP):

$$\underset{a_{(m,n)}[d],\, e_{(m,n)}}{\text{minimize}} \quad \sum_{d=1}^{D} \sum_{(m,n)\in\mathcal{F}} r_{(m,n)}[d] \cdot a_{(m,n)}[d] \tag{14}$$

$$\text{subject to} \quad a_{(m,n)}[d] \in \{0,1\},\ e_{(m,n)} \in \{0,1\},\ (m,n) \in \mathcal{F},\ d = 1..D \tag{15}$$

$$\sum_{(m,n)\in\mathcal{F}} a_{(m,n)}[d] = 1, \quad d = 1..D \tag{16}$$

$$a_{(m,n)}[d] \le e_{(m,n)},\ (m,n) \in \mathcal{F},\ d = 1..D \tag{17}$$

$$\sum_{\substack{m\in V \\ n\in V, m<n}} e_{(m,n)} \le |V| - 1,\ V \subset \{1, \ldots, M\} \tag{18}$$

$$\sum_{(m,n)\in\mathcal{F}} e_{(m,n)} = M - 1\,. \tag{19}$$

Here, (14) is a linear objective with $r_{(m,n)}[d]$ serving as coefficients, (16) ensures that each dimension is assigned to a single edge, (17) is a consistency constraint ensuring that dimensions can only be assigned to edges in the tree, (18) are the *loop elimination* constraints that are defined for all possible subsets of vertices and are in practice handled using delayed constraint generation. Finally, (19) ensures that the tree has $M - 1$ edges and is therefore a spanning tree rather than a forest.

The resulting ILP is generally a hard one. However in our experiments we found that for the considered datasets and for the practical range of dimensionalities D (upto several hundred) and code lengths M (4–32 bytes), the state-of-the-art general purpose solver [1] was able to solve the ILPs within several minutes (typically much faster). Consequently, it was possible to find globally optimal M-steps within the codebook learning. Once the optimal tree structure $\mathcal{T}$ and the optimal assignment variables A are recovered from the ILP, the minimized-out variables $c^m(\cdot)[d]$ can be recovered via the least-squares optimization (12). The pseudocode of the whole training pipeline is presented in Figure 2.

Global rotation. [8, 15] have recently suggested an optimized version of product quantization (OPQ). OPQ augments PQ codebook learning with the estimation of the global rotation of the data. We can use the same Orthogonal Procrustes analysis method as in OPQ [8, 15] to find the global rotation that further minimizes the reconstruction error of TQ. We refer to this variant of the method as *optimized tree quantization* (OTQ).

Initialization. The learning process for OTQ thus alternates (i) the M-step (re-estimating the tree, the dimension assignments, and the codebook entries), (ii) the E-step (re-estimating the codes for training examples), and (iii) the global rotation re-estimation. While each of the steps attains a global minimum (and thus never increases the reconstruction error), the overall alternation scheme converges to a local minimum that is dependent on initialization. Importantly, one can prove the following statement relating the encoding accuracy of (O)TQ and (O)PQ for the same code length:

Corollary: *Let $X = \{x_1, \ldots, x_L\}$ be a training dataset, let $C^1, \ldots, C^M,\ |C^m| = K$ be a set of (O)PQ codebooks trained on this dataset, and let $\{i_j^m\}_{j=1\ldots L}^{m=1\ldots M}$ be the (O)PQ codes of the training vectors. Then it is possible to train (O)TQ codebooks for the same X, M, K that provide smaller or equal reconstruction error on X. In other words, one can guarantee that (O)TQ encodes X with better (or same) accuracy than (O)PQ.*

Proof: see supplementary material.

While the corollary refers to the encoding of a training set, we never observed any considerable overfitting in our experiments for either of the methods. Consequently, as will be shown in the experiments, (O)TQ consistently outperforms (O)PQ in terms of the encoding accuracy on hold-out datasets.

4. Experiments

In this section we evaluate the optimized tree quantization (OTQ) approach for the tasks of nearest neighbor search and large-scale classification. We compare OTQ with other codebook-based methods PQ [10], OPQ [8, 15], and AQ [3]. As AQ encoding becomes prohibitively slow for long codes, we used AQ only for extremely short codes

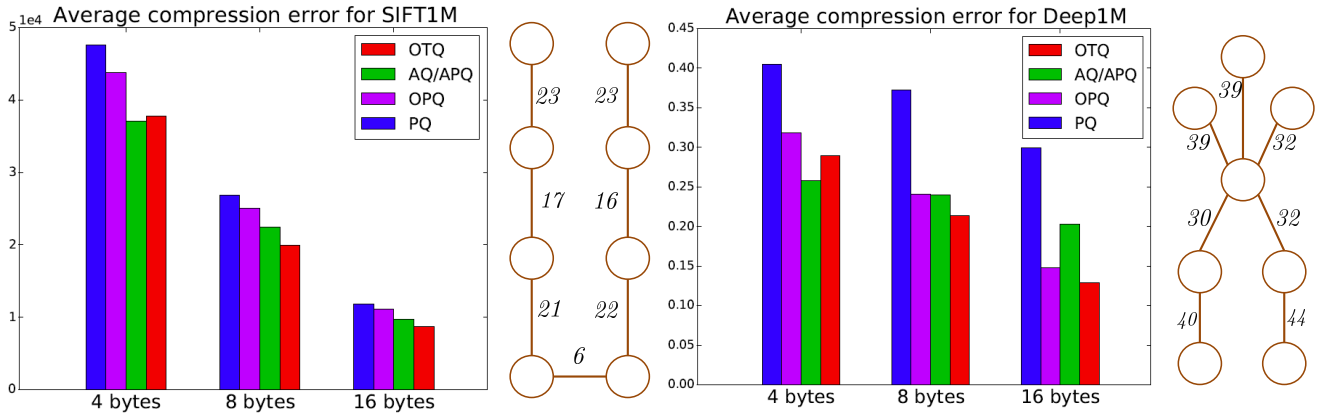

Figure 3. Average compression errors on SIFT1M (left) and Deep1M (right) datasets for different methods and visualizations of coding tree topologies of OTQ for the case M=8. AQ was used for 4-byte codes and APQ was used for 8-byte and 16-byte codes. OTQ provides the lowest error except for the shortest codes (M=4), where AQ performs best. On tree visualizations each edge is marked by a number of dimensions assigned to it.

(M=4). For longer codes (M=8, 16) we include comparison with "APQ" (i.e. a hybrid of AQ and PQ), which splits vectors into two or four parts and applies AQ to each part (see [3] for more information).

We first compare methods in the context of **approximate nearest-neighbor search** on the following two datasets:

(I) SIFT1M: This dataset introduced in [10] contains one million of 128-dimensional SIFT descriptors [14] in the main set and 100,000 descriptors in a hold out training set. It also contains 10,000 queries with known true Euclidean nearest neighbors (within the main dataset).

(II) Deep1M: this dataset contains deep neural codes of natural images obtained from the activations of a convolutional neural network [13]. The codes were L_2-normalized and PCA-compressed to $D = 256$. Recent works [4, 17] show that such neural codes can serve as powerful holistic descriptors for similar image search. The main set contains one million vectors and the training set contains 100,000 vectors. The query set contains 1,000 vectors, for which we precomputed the ground truth neighbors in the main set.

We train all methods on the training sets and compress the main sets. We then look at (i) the compression (reconstruction) error and (ii) the usefulness of the obtained codes for retrieving true nearest neighbors. For the latter, we follow the popular protocol of [10]. We thus used the *recall@T* measure [10], defined as a probability (computed over a number of queries) that the set of T closest compressed vectors contains the true nearest neighbor of an uncompressed query. Three compression levels (M=4, 8, 16 bytes) were evaluated.

As can be observed in Figure 3, for both datasets AQ performs best with 4−byte codes but APQ becomes inferior with longer codes. Meanwhile OTQ outperforms all other methods with $M = 8, 16$ and provides almost the same quality as AQ with $M = 4$. Curiously, Figure 3 also shows that the trees learned by OTQ for the two datasets have quite different topologies. Interestingly, TQ outperforms AQ/APQ for long codes (M=8, 16) even with tree structure constraints imposed on its codebooks. The reason is that the TQ encoding is optimal for given codebooks. The AQ/APQ encoding (via Beam Search) is approximate, which results in higher compression error for $M > 4$ despite having more parameters.

	OTQ	AQ	APQ
	$M = 4$		
Absolute time (ms)	2.9	52.6	52.6
OTQ speed-up	—	**18x**	**18x**
	$M = 8$		
Absolute time (ms)	6.0	235.8	103.0
OTQ speed-up	—	**39x**	**17x**
	$M = 16$		
Absolute time (ms)	12.3	1127.3	204.7
OTQ speed-up	—	**92x**	**17x**

Table 1. The average timings (in ms) of a 128-dimensional SIFT descriptor encoding with OTQ, AQ and APQ along with speed-ups factors provided by OTQ. A speedup over AQ/APQ is especially significant for large M.

The main advantage of OTQ over AQ/APQ is fast encoding especially for larger M. Table 1 demonstrates average times of a 128-dimensional SIFT descriptor encoding with both methods implemented with Python and Numpy. OTQ encoding is 17 times faster than APQ and up to 92 times faster than AQ encoding [3].

The relative performance of all methods in terms of the compression accuracy translates into the nearest-neighbor search accuracy (Figure 4). For reference, in one case we also provide the results for a state-of-the-art binary hashing method (ITQ [9] reproduced from [15]).

(O)TQ also provides faster retrieval than AQ/APQ as demonstrated on Figure 5. The reason of this advantage

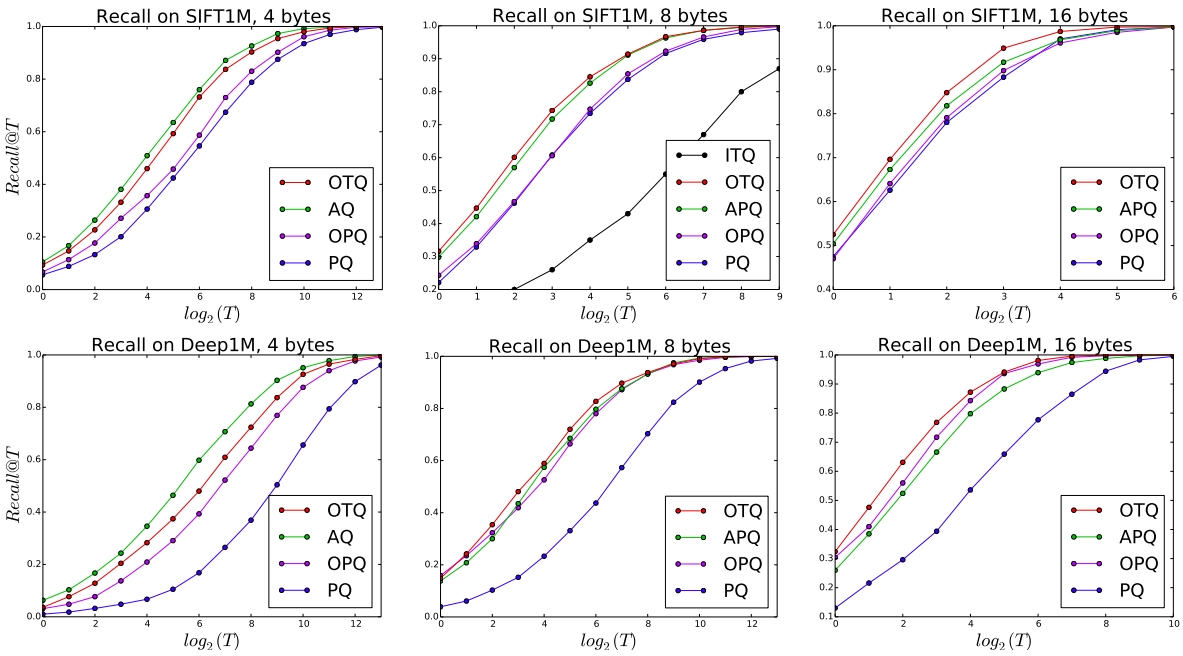

Figure 4. Euclidean nearest neighbor search accuracy based on different compression methods for SIFT1M and Deep1M datasets with the different code lengths. The x-axis shows the number T of items retrieved (in log-scale), while the y-axis shows Recall@T (the probability of the true neighbor being retrieved). For both datasets the recall@T with the the OTQ encoding is uniformly higher than for with the PQ and the OPQ encodings. AQ outperforms OTQ with $4-$byte codes, but for longer codes OTQ performs better than APQ.

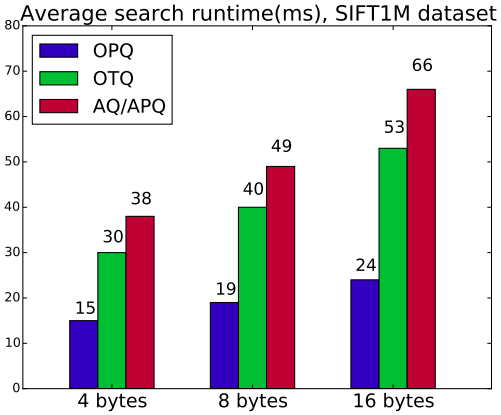

Figure 5. Average timing of exhaustive nearest neighbor search in SIFT1M dataset using OPQ, OTQ and AQ/APQ. AQ was used for 4-byte codes and APQ was used for 8-byte and 16-byte codes. OTQ provides faster search than AQ/APQ because the complexity of distance evaluation with OTQ is linear in M (see Section 2.2)

is the fact that number of terms in (8) is linear in M while within AQ/APQ this number is quadratic in M. In most of our experiments retrieval with OTQ is just 2 times slower than efficient OPQ retrieval providing significantly better recall. Hence the usage of OTQ is justified when the memory budget is limited and the compression quality is more crucial when the runtime. We also note that OTQ is slower

$M = 8$			
	R@1	R@10	R@50
CQ	0.29	0.71	0.97
OTQ	0.32	0.75	0.97
$M = 16$			
	R@1	R@2	R@5
CQ	0.54	0.71	0.88
OTQ	0.53	0.70	0.89

Table 2. Comparison of Composite Quantization and Optimized Tree Quantization for the nearest neighbor search on the SIFT1M dataset.

than (O)PQ in the case of Euclidean distance but for other similarity measures (e.g. dot-product similarity) OTQ can provide the same speed as PQ.

Another challenging competitor for TQ is the recent Composite Quantization (CQ) method [19]. Table 2 demonstrates the comparison of OTQ and CQ on the SIFT1M dataset (using figures provided by the authors of [19]). Overall, both methods achieve comparable performance.

Interestingly, CQ and OTQ achieve the advantage over PQ by analogous but complementary ways. Both CQ and OTQ relax codebooks orthogonality in PQ. In particular, CQ enforces "slight non-orthogonality" of all codebooks, while OTQ allows "arbitrary non-orthogonality" between several codebooks (which are connected in the graph). One can incorporate CQ into OTQ framework and combine both

	Reconstruction error			Classification mAP		
	320x	640x	1280x	320x	640x	1280x
Optimized Product Quantization	0.711	0.846	0.957	0.464	0.389	0.306
Additive Quantization	0.548	0.637	0.737	0.490	0.459	0.431
Optimized Tree Quantization	**0.521**	**0.607**	**0.637**	**0.497**	**0.467**	**0.441**

Table 3. Reconstruction error and average precision of image classification with Fisher vectors for learning on uncompressed data and testing on compressed data. Codebooks for OPQ and OTQ were learned on the train+val set and were used to encode the test set. The classifiers were learned on the training and validation sets and were tested on the test set. Better coding approximation of the OTQ results in higher classification accuracy. The mAP for uncompressed descriptors is 0.577.

methods, by allowing slight non-orthogonality (as in CQ) for the codebooks not connected in the OTQ graph. OTQ inference will then still apply. Codebooks can be learned with our method and refined using local optimization (as in [19]). We did not compare runtime of CQ and OTQ as CQ implementation is not avaliable but CQ should perform faster for L2-distance queries and encoding and have similar speed for dot product queries.

Classification. We performed extra experiments on PASCAL VOC 2007 [7]. We compare OPQ and OTQ for a scenario when a classifier trained on uncompressed descriptors is applied to a very large dataset of compressed descriptors in order to find images with the highest classification score [6, 5]. In this scenario, it is only necessary to evaluate the scalar products between the query (the classifier) and the compressed vectors.

We used Fisher Vector descriptors[18] with 256 components over SIFT descriptors PCA-compressed to 80 components. We then evaluated the degradation from the compression by OPQ and OTQ for different compression rates. As in [3], we split original Fisher vectors into R subvectors and compress each subvector by $8-$byte OPQ and OTQ. Different compression rates can be obtained by varying R. We use the standard mean average precision measure for PASCAL classification experiments. Both OPQ and OTQ codebooks were learned on the train+val set and used to compress descriptors from the test set.

Table 3 shows the compression error and the classification accuracy for three levels of compression (320x, 640x, 1280x). OTQ provides significantly lower reconstruction error and consequently smaller degradation in classification accuracy. The advantage is particularly large (0.135 mAP) for the highest compression level.

5. Discussion

Experiments show that the accuracy of Tree Quantization, and in particular its "optimized" variant (OTQ) exceeds that of the optimized product quantization. This advantage is due to a larger number of parameters ($2KD$ vs KD) that can be used to fit the data distribution and the ability to model dependencies between all dimensions. The advantage is more pronounced for descriptors with easily identifiable parts (such as spatial bins within SIFT, or separate GMM components within Fisher vectors).

While the AQ scheme has even more parameters (MKD) that can fit the distribution and potentially achieve lower reconstruction error, it is severely hindered by the slowness and inexactness of the encoding. As was shown in the original AQ paper the problem of AQ encoding is equivalent to the problem of an inference in a fully-connected MRF in probabilistic modeling. The usage of TQ is then an analogy of Chow-Liu tree approximation for this MRF. Similarly to Chow-Liu tree, TQ can capture second-order correlations while remaining tractable for inference.

Overall, tree quantization provides a combination of high compression accuracy and fast encoding that is attractive for practical retrieval and classification systems. Tree quantization can also be combined with any indexing structure for non-exhaustive search. The state-of-the-art methods for large-scale nearest neighbor search [11, 2] currently use PQ/OPQ to encode database points. In fact, during retrieval they reconstruct points from a short-list of candidates using their PQ-codes. Then system calculates distances from candidates to queries explicitly thus not exploiting PQ fast distance evaluation procedure. Within such a system PQ can be easily replaced by OTQ or any other similar encoding scheme.

References

[1] Gurobi optimizer. `http://www.gurobi.com/`, 2013.

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

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

[4] A. Babenko, A. Slesarev, A. Chigorin, and V. S. Lempitsky. Neural codes for image retrieval. *CoRR*, abs/1404.1777, 2014.

[5] A. Bergamo and L. Torresani. Meta-class features for large-scale object categorization on a budget. In *CVPR*, pages 3085–3092, 2012.

[6] A. Bergamo, L. Torresani, and A. W. Fitzgibbon. Picodes: Learning a compact code for novel-category recognition. In *NIPS*, pages 2088–2096, 2011.

[7] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

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

[9] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, 2011.

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

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

[12] J. H. Kappes, B. Andres, F. A. Hamprecht, C. Schnörr, S. Nowozin, D. Batra, S. Kim, B. X. Kausler, J. Lellmann, N. Komodakis, and C. Rother. A comparative study of modern inference techniques for discrete energy minimization problems. In *CVPR*, pages 1328–1335, 2013.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.

[14] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2), 2004.

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

[16] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[17] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. *CoRR*, abs/1403.6382, 2014.

[18] J. Sánchez, F. Perronnin, T. Mensink, and J. J. Verbeek. Image classification with the fisher vector: Theory and practice. *International Journal of Computer Vision*, 105(3):222–245, 2013.

[19] T. Zhang, C. Du, and J. Wang. Composite quantization for approximate nearest neighbor search. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 838–846, 2014.