LSQ++: Lower Running Time and Higher Recall in Multi-codebook Quantization

Julieta Martinez[1,2](✉), Shobhit Zakhmi[1], Holger H. Hoos[1,3], and James J. Little[1]

[1] University of British Columbia (UBC), Vancouver, Canada
{julm,zakhmi,hoos,little}@cs.ubc.ca
[2] Uber ATG, Toronto, Canada
[3] Leiden Institute of Advanced Computer Science (LIACS), Leiden, Netherlands

Abstract. Multi-codebook quantization (MCQ) is the task of expressing a set of vectors as accurately as possible in terms of discrete entries in multiple bases. Work in MCQ is heavily focused on lowering quantization error, thereby improving distance estimation and recall on benchmarks of visual descriptors at a fixed memory budget. However, recent studies and methods in this area are hard to compare against each other, because they use different datasets, different protocols, and, perhaps most importantly, different computational budgets. In this work, we first benchmark a series of MCQ baselines on an equal footing and provide an analysis of their recall-vs-running-time performance. We observe that local search quantization (LSQ) is in practice much faster than its competitors, but is not the most accurate method in all cases. We then introduce two novel improvements that render LSQ (i) more accurate and (ii) faster. These improvements are easy to implement, and define a new state of the art in MCQ.

1 Introduction

The focus of this work is multi-codebook quantization (MCQ), an approach to vector compression analogous to k-means clustering, where cluster centres arise from the combinatorial combination of entries in multiple codebooks. Modern systems for very large-scale approximate nearest neighbour (ANN) search typically rely on a data structure that shortlists candidates, followed by search using the compressed representation obtained from a variant of MCQ [5,16,17,32].

Systems for efficient large-scale search in high-dimensional spaces have important applications to prominent problems in machine learning and computer vision. For example, Mussman *et al.* [24] use Gumbel variables to randomly perturb nearest neighbour queries and accelerate learning and inference in log-linear models. Douze *et al.* [11] use a large-scale similarity graph constructed via MCQ to improve learning in deep "low-shot" models. Guo *et al.*[14] use an MCQ-based system to achieve state-of-the-art performance in maximum inner product search (MIPS), and accelerate large-scale recommender systems. Finally, Blalock

© Springer Nature Switzerland AG 2018
V. Ferrari et al. (Eds.): ECCV 2018, LNCS 11220, pp. 508–523, 2018.
https://doi.org/10.1007/978-3-030-01270-0_30

and Guttag [7] use MCQ to reduce memory usage and accelerate large-scale data mining applications.

MCQ is an optimization problem over two latent variables that approximate a given dataset: the codebooks and the codes (*i.e.*, the assignments of the data to those codebooks). The error of this approximation provides a bound for Euclidean distance and dot-product approximations in ANN and MIPS. Therefore, finding optimization methods that achieve low-error solutions is crucial for improving the performance of MCQ applications.

In similarity search, our goal is often to tackle very large datasets, so it is important that the optimization techniques scale gracefully – consider, for example, the case where one wants to index one billion vectors using the classical inverted file (IVF) [17]. An IVF partitions the dataset into K disjoint cells and learns a quantizer for each subset. Typically, $K \in \{2^{12}, 2^{13}\}$, so one has to run the training method 4 096–8 192 times. If a method takes one hour to run, then one has to wait roughly 6–12 months for training to complete. On the other hand, if a method has a running time of one minute, then the total wait time is reduced to roughly 3–6 days.[1] Unfortunately, running time is often not reported in recent work on MCQ. Here, we focus on characterizing recent methods for MCQ in terms of their running time vs accuracy trade-off, and introduce novel improvements in both speed and accuracy to LSQ, a state-of-the-art MCQ method.

Problem formulation. MCQ is the task of finding a set of codes B and (multiple) codebooks C that minimize quantization error on a given dataset X. Our objective is to determine

$$\min_{C,B} \|X - CB\|_F^2, \tag{1}$$

where $X \in \mathbb{R}^{d \times n}$ contains n d-dimensional vectors, and $C = [C_1, C_2, \ldots, C_m] \in \mathbb{R}^{d \times mh}$ is composed of m subcodebooks $C_i \in \mathbb{R}^{d \times h}$, with d dimensions and h entries each. Finally, $B = [\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n] \in \{0, 1\}^{mh \times n}$ contains n binary codes, each with m entries $\mathbf{b}_{i,j} \in \{0, 1\}^h$ that select one entry from a different codebook $\mathbf{b}_i = [\mathbf{b}_{i,1}, \mathbf{b}_{i,2}, \ldots, \mathbf{b}_{i,m}]^\top$; in other words, $\|\mathbf{b}_{i,j}\|_0 = 1$ and $\|\mathbf{b}_{i,j}\|_1 = 1$.

MCQ is useful for large-scale ANN search because, in this representation, the Euclidean distance between a query vector $\mathbf{q}$ and a compressed database vector $\mathbf{x}_i \approx \hat{\mathbf{x}}_i = \sum_{j=1}^m C_j \mathbf{b}_{i,j}$, can be computed using the expansion

$$\|\mathbf{q} - \hat{\mathbf{x}}_i\|_2^2 = \|\mathbf{q}\|_2^2 - 2 \cdot \sum_{j=1}^{m} \langle \mathbf{q}, C_j \mathbf{b}_{i,j} \rangle + \|\hat{\mathbf{x}}_i\|_2^2. \tag{2}$$

When searching for nearest neighbours, the first term can be ignored, as it is constant for all database vectors; the second term can be computed with m

[1] While in an IVF each cell can be learned in parallel, an argument similar to ours can be made for overall compute. Compute time means energy, and means money.

lookups in precomputed dot-product tables, and it is typical to use one extra codebook to quantize the third (scalar) term. Note that, when MCQ is used to approximate dot-products (*e.g.*, in MIPS) or convolutions, it is not necessary to store the norm of the encoded vector, and the full memory budget can be used to improve the quality of the approximation.

We typically set $h = 256$ [3,12,17,21,27,34], which means that each index into a codebook can be stored using 8 bits. Thus, if we use $m = \{7, 15\}$ codebooks, and set aside an extra table for storing the norm of the approximation with $h = 256$ entries as well, the memory used per vector is only 64 (resp. 128) bits.

2 Related Work

Early work in MCQ adopted orthogonal codebooks [12,17,26], which considerably simplifies the problem and leads to very scalable solutions, at the expense of accuracy. More recent work has focused on using non-orthogonal codebooks, which increase accuracy but also result in increased computational costs, deterring their wider adoption. For example, the recently released FAISS library[2] [18] implements only orthogonal MCQ techniques. In this work, we aim to better characterize and understand recent work in MCQ, with the goal of accelerating and improving the performance of non-orthogonal MCQ techniques.

Non-orthogonal MCQ. Chen *et al.* [9] introduced non-orthogonal codebooks for MCQ and proposed *residual vector quantization* (RVQ), a greedy optimization method that runs k-means on each codebook in a sequential manner. Later, Ai *et al.* [1] and Martinez *et al.* [22] independently proposed *enhanced* RVQ and resp. *stacked quantizers* (SQ), a refinement of RVQ that obtains lower quantization error, but maintains the same encoding complexity.

Babenko and Lempitsky [3] proposed *additive quantization* (AQ), which uses an expectation-maximization (EM)-like approach for optimization. The authors used beam search for updating the codes, and a conjugate gradient method for the codebook update step. Although unaware of RVQ, this paper has proven influential due to its insights and proper characterization of the hard combinatorial problems that arise in non-orthogonal MCQ. Later, Martinez *et al.* [21] introduced *local search quantization* (LSQ), an encoding method based on iterated local search [19] with iterated conditional modes (ICM), which improves upon the accuracy *vs* computation tradeoffs of the beam search method of AQ, leading to overall higher recall. Initialization consists of OPQ followed by a simpler version of *optimized tree quantization* (OTQ) [4].

Zhang *et al.* [34] proposed *composite quantization* (CQ), which minimizes quantization error but also penalizes the deviation of cross-codebook terms from

[2] https://github.com/facebookresearch/faiss

Table 1. Comparison between CQ and LSQ on SIFT1M using 64 bits.

	Trained on	Init + train	Base encoding	Total	R@1
CQ [34] (C++)	*base set*	4.5 h	–	4.5 h	0.290
CQ [34] (C++)	*learn set*	42 m	10 s	42.2 m	0.162
LSQ [21] (Julia, C++)	*learn set*	9.1 m	4.35 m	13.5 m	0.294

an (also latent) constant. The method is also EM-like, and the authors use ICM for the encoding step, and the L-BFGS [25] solver for the codebook update step. Initialization consists of PQ followed by unconstrained MCQ (Expression 1).

Finally, Ozan *et al.* [27] introduced *competitive quantization* (CompQ), a method that updates the codebooks with stochastic gradient descent (SGD), and updates the codes using beam search within a search space whose size is controlled by a hyperparameter that trades-off accuracy and computation.

3 Comparative Perfomance Evaluation

While recent work has used different experimental setups, fortunately all studies have reported results on the SIFT1M dataset at 64 bits. Thus, first we focus on comparing the three methods that report the best results on this dataset: CQ [34] (R@1 of 0.290), LSQ [21] (R@1 of 0.298) and CompQ [27] (R@1 of 0.352). We measure all our timings on a desktop with an 8-core i7-7700K CPU @4.20 GHz, 32 GB of RAM and an NVIDIA Titan Xp GPU.

LSQ vs composite quantization (CQ). For LSQ, we use as a starting point the publicly available implementation due to Martinez and Clement,[3] written in Julia [6]. For CQ, we use the recently released implementation due to Zhang[4]. This release is written in multithreaded C++, and uses the heavily optimized libraries `MKL` (for matrix operations) and `libLBFGS` (for codebook update).

We let CQ use $m = 8$ codebooks and LSQ use $m = 7$ codebooks, plus an extra codebook for the database norms. This means that both methods have the same query time and use the same amount of memory. We run both methods for 30 iterations, and use all the default hyperparameters as provided in their respective code releases. To make the comparison more fair, we have ported OTQ and LSQ encodings to C++ with OpenMP multithreading. These methods are called from Julia, and we leave the rest of the code untouched.

The results reported by Zhang *et al.* [34] on SIFT1M were trained on the *base* set. SIFT1M is provided with a *learn* set, and the more common protocol is to learn the model parameters exclusively on the *learn* set [1,3,9,12,17,21,22,26,27,35]. Thus, we also run the method limiting its parameter learning to the *learn* set.

[3] https://github.com/una-dinosauria/local-search-quantization
[4] https://github.com/hellozting/CompositeQuantization

We report the results of our experiments on Table 1. LSQ achieves slightly higher recall than CQ when CQ is trained on the *base* set, but LSQ has an overall 20× faster running time. The running time of CQ decreases drastically when we train it on the *learn* set, but the learned parameters do not generalize well to the *base* set (R@1 of 0.162). On the LabelMe22K and MNIST datasets (which traditionally do not have a *learn* partition), we have observed that LSQ consistently achieves higher recall than CQ with roughly 10× faster running times. From these results, we conclude that LSQ is faster, more accurate, and more sample-efficient (*i.e.*, it requires less training data) than CQ.

LSQ vs competitive quantization (CompQ). Since there is no publicly available implementation of CompQ, we have tried to reproduce the reported results ourselves, with moderate success. We have not, for example, been able to reproduce the transform coding initialization reported in the paper, but have instead used RVQ, which was reported to achieve slightly worse results. We obtained a R@1 of 0.346 using a beam search width of 32, and training for 250 epochs (the parameters of the best reported result). The small difference in recall may be attributed to our different initialization.

However, the largest barrier to experimentation on our side is that our CompQ implementation, written in Julia, takes roughly 40 min per epoch to run. This means that our experiment on SIFT1M with 64 bits took almost one week to finish. We contacted the CompQ authors, and they mentioned using a multithreaded C++ implementation with pinned memory, an ad-hoc sort implementation, and special handling of threads. Their implementation takes 551 s per epoch, or about 38 h (∼1.5 days) in total for 250 epochs on a desktop with a 10-core Xeon E5 2650 v3 @2.3 GHz CPU. We compare CompQ to LSQ using our multithreaded C++ implementation (same as in Table 1). We also use $m = 8$ codebooks in total, which controls for query time and memory use with respect to CompQ. We train for 25 iterations in total, and again use all the default parameters of LSQ.

Table 2. Comparison between CompQ and LSQ on SIFT1M using 64 bits.

	Iters	Init	Training	Base encoding	Total	R@1
CompQ [27] (C++)	250	–	–	–	38 h	0.352
LSQ [21] (Julia, C++)	25	2.6 m	6.34 m	5.8 m (32 iters)	15.2 m	0.340
LSQ [21] (Julia, CUDA)	25	1.1 m	2.8 m	29 s (32 iters)	4.4 m	0.340

LSQ and CompQ live on opposite sides of the parallelism spectrum: while CompQ uses stochastic gradient descent (SGD) with a batch size of 1, and is thus primarily sequential, LSQ is EM-like, so it is very amenable to parallelization. This means that CompQ is unlikely to benefit from a GPU implementation, as these require fairly large batch sizes to deliver higher throughput than CPUs (in fact, using large batch sizes to accelerate the training of large deep neural networks with SGD is an active area of research [13,29]). To further explore the consequences of this algorithmic trade-off, we used the publicly available CUDA implementation of LSQ encoding [23] to accelerate training and base processing. We also implemented OTQ encoding in CUDA.

We report the results of our experiments on Table 2. Our C++ implementation of LSQ is about 150× faster than CompQ and, when using a GPU, LSQ achieves roughly a 500× speedup over CompQ. However, the recall of LSQ lags by 0.012 behind CompQ. Further research into CompQ may focus on finding ways to increase its batch size, so that it can leverage modern GPUs.

Improving LSQ: Desiderata. In the light of these results, we suggest the following criteria to improve LSQ:

(a) First, we would like to make LSQ more accurate, so that it can narrow the gap with (and ideally, surpass) CompQ in terms of recall.
(b) Second, we would like to maintain the parallelism of LSQ, because it is a distinctive feature that makes it fast in practice.
(c) Finally, because LSQ is faster than its competitors, we want to find ways to trade-off running time for accuracy. To make this trade-off more attractive in practice, we would also like to decrease LSQ's overall running time.

Next, we propose improvements to LSQ that satisfy all these criteria.

4 Lower Running Time with a Fast Codebook Update

While benchmarking LSQ using a GPU, we noticed that the codebook update step is the most computationally expensive part of LSQ. This is somewhat counterintuitive, because encoding has historically been identified as the bottleneck in MCQ [3,21]. However, recent hardware and algorithmic improvements have upended this idea. In particular, out of the 2.8 min of training time for LSQ with $m = 8$ codebooks and 25 iterations (last row of Table 2), 2.34 min are spent updating the codebook C. Thus, decreasing the running time of the codebook update step would significantly decrease the overall running time of LSQ. Formally, the codebook update step amounts to determining

$$\min_C \|X - CB\|_F^2; \tag{3}$$

the current state-of-the-art method for this step was originally proposed by Babenko and Lempitsky [3], who noticed that finding C corresponds to a least-squares problem where C can be found independently in each dimension. Since B can be seen as a very sparse matrix, the authors proposed using iterative conjugate gradient (CG) methods in this step. This has the additional advantage that B can be reused for the d problems that finding C decomposes into. We have identified two problems with this approach:

1. **Explicit sparse matrix construction is inefficient**. CG APIs typically require that B be represented as an explicit sparse matrix. Although efficient data structures for sparse matrices exist (*e.g.*, the *compressed sparse row* of numpy), in practice, B is stored as an $m \times n$ `uint8` matrix. We would like to use this representation and avoid using an additional data structure.
2. **Failure to exploit the binary nature of** B. The matrix B is composed exclusively of ones and zeros (*i.e.*, it is binary). Data structures used for sparse matrices are commonly designed for the general case when the non-zero entries are arbitrary real numbers, leaving room for additional optimization.

Direct codebook update. We now introduce a method for fast codebook update, which takes advantage of these two observations. First, we note that it is possible to use a direct method instead of iterative CG, by rewriting Expression 3 as a regularized least-squares problem:

$$\min_{C} \|X - CB\|_F^2 + \lambda \|C\|_F^2. \tag{4}$$

In this case, the optimal solution can be obtained by taking the derivative with respect to C and setting it to zero

$$C = XB^\top (BB^\top + \lambda I)^{-1}. \tag{5}$$

While we are not interested in a regularized solution, we can still benefit from this formulation by setting λ to a very small value ($\lambda = 10^{-4}$ in our experiments), which simply renders the solution numerically stable. A crucial advantage of this formulation is that the matrix $BB^\top + \lambda I \in \mathbb{R}^{mh \times mh}$ is square, symmetric, positive-definite and fairly compact; *notably, its size is independent of* n. Furthermore, thanks to regularization, $BB^\top + \lambda I$ is guaranteed to be full-rank. Thus, matrix inversion can be performed directly with the help of a Cholesky decomposition in $O(m^3 h^3)$ time. Because matrix inversion is efficient, the bottleneck of our method lies in computing $BB^\top \in \mathbb{N}^{mh \times mh}$, as well as $XB^\top \in \mathbb{R}^{d \times mh}$. We exploit the structure in B to accelerate both operations.

Computing $BB^\top$. By indexing B across each codebook, $B = [B_1, \cdots, B_m]^\top$, $BB^\top$ can be written as a block-symmetric matrix composed of m^2 blocks of size $h \times h$ each:

$$BB^\top = \begin{bmatrix} B_1B_1^\top & B_1B_2^\top & \dots & B_1B_m^\top \\ B_2B_1^\top & B_2B_2^\top & \dots & B_2B_m^\top \\ \vdots & \vdots & \ddots & \vdots \\ B_mB_1^\top & B_mB_2^\top & \dots & B_mB_m^\top \end{bmatrix}. \tag{6}$$

Here, the diagonal blocks $B_N B_N^\top$ are diagonal matrices themselves, and since B is binary, their entries are a histogram of the codes in B_N. Moreover, the off-diagonal blocks are the transpose of their symmetric counterparts: $B_N B_M^\top = (B_M B_N^\top)^\top$, and can be computed as bivariate histograms of the codes in B_M and B_N. Using these two observations, this method takes $O(m^2 n)$ time, while computing $BB^\top$ naïvely would take $O(m^2 h^2 n)$.

Computing $\boldsymbol{XB}^\top$. We again take advantage of the structure of B to accelerate this step. $XB^\top$ can be written as a matrix of m blocks of size $d \times h$ each,

$$XB^\top = [XB_1^\top, XB_2^\top, \ldots, XB_m^\top]. \tag{7}$$

Each block $XB_i^\top$ can be computed by treating the $B_i^\top$ columns as binary vectors that select the columns of X to sum together. This method takes $O(mnd)$ time, while computing $XB^\top$ naively would take $O(mhnd)$.

Codebook update in CQ. Zhang *et al.* [34] propose a formulation similar to Eq. 5 for codebook update, which they use to warm-start the CQ optimization process, but do not introduce regularization. Since $BB^\top$ is not guaranteed to have full rank, the authors use SVD for computing its inverse, disregarding solution components associated with small singular values. They also did not exploit the sparsity in B to compute the other terms of the solution. In our experiments, their method takes more than a minute to run, while our solution runs in well under a second.

5 Higher Recall with Stochastic Relaxations

Our goal in this Section is to make LSQ more accurate, while maintaining the high level of parallelism and speed that it already enjoys in practice. To this end, we note that LSQ is fast because its optimization process is EM-like, which allows it to take advantage of highly parallel architectures. However, a well-known problem with such EM-like approaches is their tendency to converge to local minima. We also note that MCQ is analogous to k-means clustering (with combinatorial codebooks). Many years of research into k-means have resulted in a number of improvements to the original Lloyd's algorithm (*e.g.*, k-means++ initialization [2], or cluster closures for faster encoding [31]), so we look into the literature for methods that may be adapted to improve MCQ.

5.1 Stochastic relaxations

A stochastic relaxation (SR), as formalized by Zeger *et al.* [33] in the early 1990s, is a method that defines an approximation to simulated annealing, with the idea of improving the quality of an approximation at reasonable computational costs. The idea was originally proposed to improve k-means clustering, and here we revisit and adapt it for MCQ.

Broadly defined, simulated annealing (SA) is a classical stochastic local search (SLS) technique that iteratively works in 3 major steps: (1) define an optimization state s, (2) create a new state s' by randomly perturbing the current state: $s' = \pi(s)$, and (3) decide whether to reject or accept the new state as the basis for the next perturbation (for a broad review of the subject, see [15]). The acceptance probability in Step 3 is controlled by a parameter traditionally called *temperature*, which is typically slowly decreased over many iterations of

Steps 2 and 3. (Various temperature schedules have been proposed and used in the many applications of simulated annealing). A stochastic relaxation modifies some of the typical SA steps in order to make them more computationally efficient. We now define these three steps for our method.

Defining a SA state: A functional view of MCQ. As a first step, we formally define an optimization state in MCQ. Expression 1 is defined over two latent variables, C and B. We assume that the optimization state is fully determined given a single variable, either C or B, which fully specifies the other via a pre-defined function. Thus, we define

- an *encoder* function $\mathcal{C}(X, B) \rightarrow C$, and
- a *decoder* function $\mathcal{D}(X, C) \rightarrow B$.

In our case, $\mathcal{C}$ amounts to the codebook-update step, for which we adopt the method described in Sect. 4. Similarly, $\mathcal{D}$ amounts to updating the codes B; in this case, we simply adopt the encoding method of LSQ. We have defined the optimization state of MCQ in two ways, which will give rise to two SR methods. The first method, called SR-C, uses the *encoder* function $\mathcal{C}$, and the second method, called SR-D, uses the *decoder* function $\mathcal{D}$.

Perturbing the SA state. The next step is to define a way to perturb the SA state at time-step i. We define two perturbation methods, one for SR-C and one for SR-D. Since we have defined the state as fully-determined given either variable via a proxy function, we can perturb the state by simply perturbing the corresponding function used in SR-C or SR-D. We define the functions

- $\mathcal{C}^* := \mathcal{C}(\pi_{\mathcal{C}}(X, i), B) \rightarrow C$ for SR-C, and
- $\mathcal{D}^* := \mathcal{D}(X, \pi_{\mathcal{D}}(C, i)) \rightarrow B$ for SR-D.

$\pi_{\mathcal{C}}(X, i) \rightarrow X + T(i) \cdot \epsilon$ amounts to adding noise ϵ to X, according to a predefined temperature schedule $T(i)$. We choose to sample the noise from a zero-mean Gaussian with a diagonal covariance proportional to X; in other words, $\epsilon \sim N(\mathbf{0}, \Sigma)$, where $\Sigma = diag(cov(X))$.

A major difference between k-means and MCQ is that, in MCQ, we use multiple codebooks. This difference is particularly important in SR-D, where the noise affects C, which represents m different codebooks. Since the centroids are obtained by summing one entry from each codebook, perturbing C amounts to perturbing the centroids m times. We thus define the perturbation function for SR-D slightly differently: $\pi_{\mathcal{D}}(C, i) \rightarrow C + (T(i)/m) \cdot \epsilon$. In other words, we multiply the noise by factor of $1/m$ in SR-D.

Temperature schedule. In simulated annealing, it is common to gradually reduce the temperature, which controls the probability of accepting a new state (the so-called Metropolis-Hastings criterion). In SR, following Zeger *et al.* [22],

Algorithm 1 EM-like approach to MCQ [3,21]

```
1: function LSQ($X, I$)
2:     $B \leftarrow initialization(X)$
3:     $i \leftarrow 1$                                            ▷ Iteration counter
4:     while $i \leq I$ do
5:         $C \leftarrow \text{argmin}_C \|X - CB\|_F^2$                ▷ Codebook update
6:         $B \leftarrow \text{argmin}_B \|X - CB\|_F^2$                  ▷ Encoding step
7:         $i \leftarrow i + 1$
8:     end while
9:     return $C, B$
10: end function
```

we instead use the temperature to control the amount of noise added in each time-step. We use the schedule

$$T(i) \rightarrow (1 - (i/I))^p , \tag{8}$$

where I is the total number of iterations, i represents the current iteration, and $p \in (0, 1]$ is a tunable hyper-parameter. We have found that a value of $p = 0.5$ produces good results, and we use this parameter in all our experiments.

Acceptance criterion. The final building block of SA is an acceptance criterion, which decides whether the new (perturbed) state will be accepted or rejected. Following Zeger *et al.*, we always accept the new state. As we will show, this simple criterion gives excellent results in practice.

Recap. To summarize, we have introduced two algorithms that define crude approximations to simulated annealing: SR-C and SR-D. These approximations are extremely simple to implement. To highlight this simplicity, we summarize the EM-like approach to MCQ in Algorithm 1; notice that

– **SR-C** follows Algorithm 1 exactly, except that line 5 is replaced by $C \leftarrow \text{argmin}_C \|\pi_{\mathcal{C}}(X, i) - CB\|_F^2$, and
– **SR-D** follows Algorithm 1 exactly, except that line 6 is replaced by $B \leftarrow \text{argmin}_B \|X - \pi_{\mathcal{D}}(C, i)B\|_F^2$.

In other words, SR-C and SR-D amount to adding noise in different parts of the EM-like MCQ optimization pipeline, but the workhorse functions that perform the codebook-update, as well as the encoding encoding step, remain unchanged. This has multiple advantages. On one hand, this means that we can fully maintain the parallelism of LSQ. On the other hand, if in the future better codebook-update or encoding functions are found, they can be seamlessly integrated into our pipelines. Finally, we note that our methods involve only minimal computational overhead, as they only require the computation of the covariance of either X (which can be computed once and re-used many times in SR-C), or C, which is a compact variable independent of n. In practice, this overhead is negligible: <0.1 s for SR-C, and <0.01 s for SR-D. We refer to the combination of SR and fast codebook update as LSQ++.

6 Experimental Evaluation

We quantify the impact of our codebook update method by measuring the time it saves per LSQ iteration (*i.e.*, between lines 4 and 8 in Algorithm 1), and with a head-to-head large-scale evaluation against conjugate gradient (CG) methods. We also measure the impact of SR-C and SR-D by reporting recall@N.

Datasets. We evaluate our contributions on five datasets. The first two datasets are LabelMe22K [28] and MNIST. These datasets were originally created for classification, and have only two partitions (training/test). We learn both B and C on the training set, and use the test set as queries. LabelMe22K has $d = 512$ dimensions, 20 019 training vectors, and 2 000 queries. MNIST has $d = 784$ dimensions, 60 000 training vectors, and 10 000 queries.

The other three datasets are SIFT1M [17], Deep1M and VGG (called "Convnet1M" in [21]). SIFT1M is a classical retrieval dataset of SIFT [20] features. We have put together the Deep1M dataset, by sampling from the 10 million *example* set provided with the recently introduced Deep1B dataset [5]. These vectors come from the last convolutional layer of a GoogLeNet v3 [30] network, and have been PCA-projected to 96 dimensions. The VGG dataset consists of vectors from the CNN-M-128 network of Chatfield *et al.* [8] evaluated on Imagenet [10] images. These datasets have three partitions: *train*, *query* and *base*. We follow the standard protocol, which uses the train set to learn the codebooks C, and then uses those codebooks to encode the base set (*i.e.*, obtain B); we then use the query set to find approximate nearest neighbours in the compressed base set [1,3,9,12,17,21,22,26,27]. SIFT1M and VGG have $d = 128$ dimensions, and Deep1M has $d = 96$ dimensions. The three datasets have 100 000 training vectors, 1 M base vectors, and 10 000 queries.

Table 3. Total time per LSQ/LSQ++ iteration, depending on how we update C (CG or Cholesky), and how we update B (using a C++ or a CUDA implementation).

	64 bits		128 bits	
SIFT1M	CG	Chol	CG	Chol
Julia, C++	14.2 s	5.6 s	38.5 s	22.5 s
Julia, CUDA	6.8 s	1.2 s	20.3 s	4.3 s

	64 bits		128 bits	
Deep1M	CG	Chol	CG	Chol
Julia, C++	9.5 s	7.2 s	30.7 s	24.2 s
Julia, CUDA	3.3 s	1.0 s	10.6 s	4.1 s

6.1 Fast Codebook Update

We show the time savings obtained due to our codebook update method on Table 3. Our method saves anywhere from 2.3 (Deep1M, 64 bits) to 16 s (SIFT1M, 128 bits) of training time per iteration. This has a bigger impact when encoding is GPU-accelerated, as it results in 2.2–5.6× speedups in practice.

Large-scale experiments. On Fig. 1, we show a "stress-test" comparison between our method for fast codebook update and CG, using dataset sizes of $n = \{10^4, 10^5, 10^6, 10^7\}$. We take the first n training vectors from the SIFT1B [17] and Deep1B [5] datasets, and generate a random B. This is a specially easy case for CG, and it takes only 2–3 iterations to converge. Even in this case, our method is orders of magnitude faster than previous work, and stays under 10 s in all cases, while CG takes up to 700 s for $n = 10^7$. Our method is only slower on small training sets due to the complexity of matrix inversion, which is independent of n.

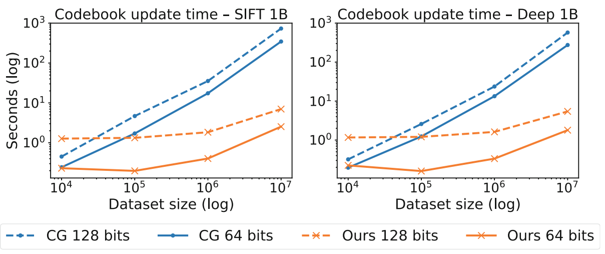

Fig. 1. Time for codebook update as a function of dataset size with up to 10^7 vectors.

6.2 Stochastic Relaxations

To evaluate our second contribution, we report recall@1, which represents the empirical probability, computed over the query set, that the actual nearest neighbour of the query is returned as the first retrieved entry. We run every method ten times on each dataset and report the average result to account for the randomness in recall.

We compare our contributions against the classical orthogonal MCQ methods PQ [17] and OPQ [12,26], as well as the more recent RVQ [9], ERVQ [1,22], CQ [34], and LSQ [21]. All methods use the same memory budget (64 or 128 bits per vector), the same codebook size of $h = 256$, and require the same number of table lookups to approximate a distance, so their query times are comparable as well. We run all the methods for 25 iterations.

Recall@1. Figures 2 and 3 show the recall@1 obtained by different methods as a function of time. We observe that SR-D obtains higher recall than LSQ in all datasets, and for both 64 and 128 bits, except for SIFT1M at 128 bits. Our fast codebook update method makes optimization faster than LSQ in all cases.

SR-C shows a more interesting behaviour. When using 64 bits, the method either gives a small boost to LSQ, or has a small detrimental effect (LabelMe22K). However, when using 128 bits, the method underperforms LSQ in all datasets, except for Deep1M and VGG. We find this result rather interesting,

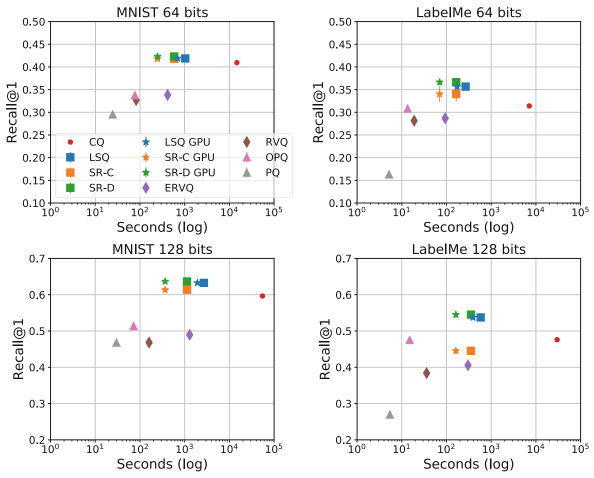

Fig. 2. Recall@1 as a function of time in the MNIST and LabelMe datasets.

as it suggests that SR-C is better suited for deep features, which currently dominate a number of machine learning and computer vision applications. However, its performance on more classical benchmarks is somewhat disappointing.

We also note that, once we account for query time by dedicating one codebook to store the database norms, RVQ [9] and ERVQ/SQ [1,22] tend to perform worse than PQ and OPQ – the only exception being the Deep1M and VGG datasets again. Previous work controlled only for memory use (with increased query time), so this detail was not obvious from previous benchmarks.

Finally, we also observe that CQ fails to generalize when trained on the *learn* set, as is the standard protocol for SIFT1M. The method, however, performs well on LabelMe and MNIST, which do not have a separate learning set. This is in line with our preliminary analysis, and suggests that CQ needs more training data (which implies more training time) to generalize well.

Software. For our experiments, we wrote Rayuela.jl, a library that implements PQ, OPQ, OTQ, RVQ, ERVQ, CompQ, LSQ and LSQ++ in Julia, with C++ and CUDA bindings for OTQ and LSQ/LSQ++ encoding – we do not include CQ, because we want to release our library under an MIT licence, and the CQ code, released under GPLv2, does not allow for stricter sublicensing. We believe that Rayuela.jl is the most comprehensive library of MCQ methods to date. Rayuela.jl is available at https://github.com/una-dinosauria/Rayuela.jl.

Comparison to CompQ. In Table 4, we update the benchmark against CompQ. Out of the box, LSQ++ (with SR-D) manages to reduce the gap to CompQ by half, from 0.012 to 0.006, and is also faster due to the faster codebook update.

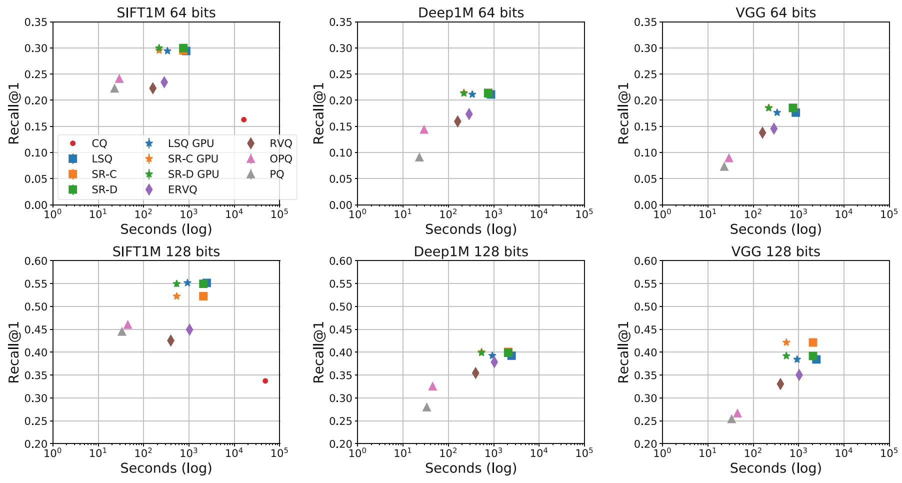

Fig. 3. Recall@1 as a function of time in the SIFT1M, Deep1M and VGG datasets.

We iteratively double the computational budget of LSQ++ (trading off computation for accuracy), and bring the difference in recall to 0.001 with 100 training iterations and 128 Base encoding ILS iterations. Doubling the budget of this final step puts our method above CompQ by 0.001 in R@1. Even under these circumstances, LSQ++ is still 200× faster than CompQ.

Table 4. Comparison between CompQ, LSQ and LSQ++ on SIFT1M using 64 bits.

	Iters	Init	Training	Base encoding	Total	R@1
CompQ [27] (C++)	250	–	–	–	38 h	0.352
LSQ [21] (Julia, CUDA)	25	1.1 m	2.8 m	29 s (32 iters)	4.4 m	0.340
LSQ++ (Julia, CUDA)	25	1.1 m	33 s	29 s (32 iters)	2.1 m	0.346
LSQ++ (Julia, CUDA)	50	2.2 m	1.1 m	58 s (64 iters)	4.3 m	0.348
LSQ++ (Julia, CUDA)	100	4.4 m	2.2 m	1.9 m (128 iters)	8.5 m	0.351
LSQ++ (Julia, CUDA)	100	4.4 m	2.2 m	3.9 m (256 iters)	10.5 m	<u>0.353</u>

7 Conclusions

We have benchmarked recent non-orthogonal MCQ algorithms and have found that (1) LSQ [21] is considerably faster than its competitors, (2) LSQ lags in accuracy behind CompQ, and (3), when using a GPU, the computational bottleneck of LSQ is, somewhat counterintuitively, the codebook update step.

Based on these observations, we have introduced two stochastic relaxation methods for MCQ that provide inexpensive approximations to simulated annealing, a technique widely used for hard combinatorial problems. One of these methods (SR-D) consistently improves recall in LSQ at negligible computational cost. We have also introduced a method for fast codebook updates that results in faster training. Both of our contributions can be used as out-of-the-box improvements on top of LSQ and are simple to implement. Furthermore, these two contributions increase the gap in running time between LSQ and its competitors, and account for the difference in accuracy between LSQ and CompQ [27].

Acknowledgements. We thank NVIDIA for the donation of GPUs used in this project. Shobhit was supported by a Mitacs Globalink research internship while at UBC. We also thank Ioan Andrei Bârsan for proofreading our work, and anonymous reviewers for multiple comments that improved this project. This research was supported in part by NSERC.

References

1. Ai, L., Yu, J., Guan, T., He, Y.: Efficient approximate nearest neighbor search by optimized residual vector quantization. In: International Workshop on Content-Based Multimedia Indexing (CBMI) (2014)
2. Arthur, D., Vassilvitskii, S.: k-means++: the advantages of careful seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM symposium on Discrete algorithms, pp. 1027–1035. Society for Industrial and Applied Mathematics (2007)
3. Babenko, A., Lempitsky, V.: Additive quantization for extreme vector compression. In: CVPR (2014)
4. Babenko, A., Lempitsky, V.: Tree quantization for large-scale similarity search and classification. In: CVPR (2015)
5. Babenko, A., Lempitsky, V.: Efficient indexing of billion-scale datasets of deep descriptors. In: CVPR (2016)
6. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: a fresh approach to numerical computing. arXiv preprint arXiv:1411.1607 (2014)
7. Blalock, D.W., Guttag, J.V.: Bolt: accelerated data mining with fast vector compression. In: KDD (2017)
8. Chatfield, K., Simonyan, K., Vedaldi, A., Zisserman, A.: Return of the devil in the details: delving deep into convolutional nets. In: BMVC (2014)
9. Chen, Y., Guan, T., Wang, C.: Approximate nearest neighbor search by residual vector quantization. Sensors **10**(12), 11259–11273 (2010)
10. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: ImageNet: a large-scale hierarchical image database. In: CVPR (2009)
11. Douze, M., Szlam, A., Hariharan, B., Jégou, H.: Low-shot learning with large-scale diffusion. In: NIPS (2017)
12. Ge, T., He, K., Ke, Q., Sun, J.: Optimized product quantization. In: CVPR (2013)
13. Goyal, P., et al.: Accurate, large minibatch SGD: training imagenet in 1 hour. arXiv preprint arXiv:1706.02677 (2017)
14. Guo, R., Kumar, S., Choromanski, K., Simcha, D.: Quantization based fast inner product search. In: AISTATS (2016)
15. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Elsevier, Amsterdam (2004)

16. Hu, H., et al.: Web-scale responsive visual search at bing. arXiv preprint arXiv:1802.04914 (2018)
17. Jégou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. TPAMI **33**(1), 117–128 (2011)
18. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with GPUs. arXiv preprint arXiv:1702.08734 (2017)
19. Lourenço, H.R., Martin, O.C., Stützle, T.: Iterated local search. In: Handbook of Metaheuristics, pp. 320–353. Springer, Berlin (2003)
20. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. IJCV **60**(2), 91–110 (2004)
21. Martinez, J., Clement, J., Hoos, H.H., Little, J.J.: Revisiting additive quantization. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9906, pp. 137–153. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46475-6_9
22. Martinez, J., Hoos, H.H., Little, J.J.: Stacked quantizers for compositional vector compression. arXiv preprint arXiv:1411.2173 (2014)
23. Martinez, J., Hoos, H.H., Little, J.J.: Solving multi-codebook quantization in the GPU. In: ECCV Workshop on Web-Scale Vision and Social Media (VSM) (2016)
24. Mussmann, S., Levy, D., Ermon, S.: Fast amortized inference and learning in log-linear models with randomly perturbed nearest neighbor search. In: UAI (2017)
25. Nocedal, J.: Updating quasi-newton matrices with limited storage. Math. Comput. **35**(151), 773–782 (1980)
26. Norouzi, M., Fleet, D.J.: Cartesian k-means. In: CVPR (2013)
27. Ozan, E.C., Kiranyaz, S., Gabbouj, M.: Competitive quantization for approximate nearest neighbor search. IEEE Trans. Knowl. Data Eng. **28**(11), 2884–2894 (2016)
28. Russell, B.C., Torralba, A., Murphy, K.P., Freeman, W.T.: Labelme: a database and web-based tool for image annotation. IJCV **77**(1–3), 157–173 (2008)
29. Smith, S.L., Kindermans, P.J., Le, Q.V.: Don't decay the learning rate, increase the batch size. In: ICLR (2018)
30. Szegedy, C., et al.: Going deeper with convolutions. In: CVPR (2015)
31. Wang, J., Wang, J., Ke, Q., Zeng, G., Li, S.: Fast approximate K-means via cluster closures. In: Baughman, A.K., Gao, J., Pan, J.-Y., Petrushin, V.A. (eds.) Multimedia Data Mining and Analytics, pp. 373–395. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14998-1_17
32. Xia, Y., He, K., Wen, F., Sun, J.: Joint inverted indexing. In: ICCV (2013)
33. Zeger, K., Vaisey, J., Gersho, A.: Globally optimal vector quantizer design by stochastic relaxation. IEEE Trans. Signal Process. **40**(2), 310–322 (1992)
34. Zhang, T., Du, C., Wang, J.: Composite quantization for approximate nearest neighbor search. In: ICML (2014)
35. Zhang, T., Qi, G.J., Tang, J., Wang, J.: Sparse composite quantization. In: CVPR (2015)