Compositional Compression of Deep Image Features Using Stacked Quantizers

by

Julieta Martinez-Covarrubias

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

December 2014

© Julieta Martinez-Covarrubias, 2014
Abstract

In computer vision, it is common for image representations to be stored as high-dimensional real-valued vectors. In many computer vision applications, such as retrieval, classification, registration and reconstruction, the computational bottleneck arises in a process known as feature matching, where, given a query vector, a similarity score has to be computed to many vectors in a (potentially very large) database. For example, it is not uncommon for object retrieval and classification to be performed by matching global representations in collections with thousands \[27\] or millions \[10, 32\] of images. A popular approach to reduce the computational and memory requirements of this process is vector quantization.

In this work, we first analyze several vector compression methods typically used in the computer vision literature in terms of their computational trade-offs. In particular, we observe that Product Quantization (PQ) and Additive Quantization (AQ) lie on the extremes of a compositional vector compression design choice, where the former assumes complete codebook independence and the latter assumes full codebook dependence. We explore an intermediate approach that exploits a hierarchical structure in the codebooks. This results in a method that is largely competitive with AQ in structured vectors, and outperforms AQ in unstructured vectors while being several orders of magnitude faster.

We also perform an extensive evaluation of our method on standard benchmarks of Scale Invariant Feature Transform (SIFT), and GIST descriptors, as well as on new datasets of features obtained from state-of-the-art convolutional neural networks. In benchmarks of low-dimensional deep features, our approach obtains the best known-to-date results, often requiring less than half the memory of PQ to achieve the same performance.
Preface

A slightly more compact version of the work in Chapters 1-4 has been submitted for publication as Martinez, J., Hoos, H. H., & Little, J. J.: Stacked Quantizers for Compositional Vector Compression. Chapters 5 and 6 consist of original, unpublished work.
# Table of Contents

Abstract ......................................................... ii

Preface ......................................................... iii

Table of Contents ............................................. iv

List of Tables .................................................. vi

List of Figures .................................................. vii

Acknowledgments .............................................. ix

1 Introduction .................................................. 1

2 Related work ................................................ 4
  2.1 Deep features and their compression .................... 4
  2.2 Vector quantization ..................................... 6
  2.3 Compositional quantization ............................. 6
  2.4 Product quantization .................................. 7
  2.5 Optimized product quantization ....................... 9
  2.6 Additive quantization ................................ 9

3 Stacked quantizers ........................................... 12
  3.1 Encoding ................................................. 12
  3.2 Initialization .......................................... 14
  3.3 Codebook refinement ................................... 15
List of Tables

Table 4.1 Summary of the datasets on which we test our approach. The two datasets at the top of the table were introduced in [18] and are standard benchmarks in the literature. The four datasets on the bottom are introduced in this work.

Table 6.1 Summary of the design choices that make AQ and SQ different.
List of Figures

Figure 3.1  A graphical model representation of codebook learning that highlights the assumptions of PQ/OPQ, AQ, and our approach. Here we show 4 subcodebooks: a) PQ and OPQ assume sub-codebook independence; b) AQ attempts to solve the fully connected problem; c) we assume a hierarchical structure.

Figure 3.2  Encoding as performed with Stacked Quantizers, shown for 4 subcodebooks. Left: the vector is passed through a series of quantizers, with residuals further encoded down the line. Right: A geometric interpretation of our approach. After recursively encoding residuals, the representation is additive in the encodings, and the quantization error is the remaining residual.

Figure 4.1  Quantization error on SIFT1M and GIST1M. SQ shows a modest advantage on structured features.

Figure 4.2  Quantization error on deep features with different dimensionalities. The non-independent approaches AQ and SQ clearly outperform PQ and OPQ. SQ achieves the best performance when using 8 and 16 codebooks (64 and 128 bits per feature) in all cases.

Figure 4.3  Recall@N curves on the SIFT1M and GIST1M datasets.

Figure 4.4  Recall@N curves on the ConvNet1M-128 and ConvNet1M-1024 datasets.
Figure 4.5 Recall@N curves on the ConvNet1M-2048 and ConvNet1M-4096 datasets.

Figure 4.6 Top-5 classification error on the ILSVRC-2012 dataset as a function of compression. The dotted black line corresponds to performance without compression. The left pane shows performance using 128-dimensional deep features, and the right pane shows performance for 1024-dimensional deep features.

Figure 4.7 Left: Training time vs. Quantization error of the benchmarked methods on the ConvNet1M-128 training dataset (100K features). For clarity, we plot each 50 iterations for PQ and OPQ and each 25 iterations for SQ after initialization. PQ and OPQ complete 100 iterations after 286 and 336 seconds respectively (4.8 and 5.6 minutes), SQ takes ~1500 seconds for initialization and ~1000 seconds for 100 iterations of codebook refinement (42 minutes in total). APQ takes ~2.7 hours for 10 iterations. Right: Encoding of the database set of 1M features. PQ and OPQ take ~5 seconds, SQ ~20 seconds, and APQ ~9.2 hours.

Figure 5.1 Precision@N curves on the ILSRVC 2012 dataset.

Figure 5.2 Retrieval errors introduced by quantization noise in natural images. Top: the roundness of the snail is incorrectly matched to an oyster in retrieval with PQ and OPQ, and to a turtle in PQ. Middle: A soccer ball is matched against other round objects such as an acorn and a tennis ball. Bottom: a lemon is matched against apples and a lime. Best viewed digitally.

Figure 5.3 Typical retrieval errors in person-made objects. Top: Binoculars are incorrectly matched to photographic cameras. Middle: A bib is incorrectly matched to t-shirts. Bottom: A pickup truck is incorrectly matched to cars. Best viewed digitally.
Figure 5.4  Top: A mantis brings up grasshoppers, which share some structure in the legs, but not the head. Bottom: A picture of a burrito is incorrectly matched with croissants and sandwiches. Best viewed digitally.

Figure 6.1  Quantization error results for compositional quantizers with local search on the SIFT1M and ConvNet1M-128 datasets.
I would like to thank my supervisors, Holger Hoos and James Little, for their encouragement and support. Their advice, both academic and professional, has been the main influence on how I do and think of research.

I also want to thank my advisor, Nick Harvey, who convinced me to pursue research in computer vision instead of computational theory. The more time goes by, the more I realize the immense value of his advice. Finally, I want to thank my mentor, Joanna McGrenere. Her professional advice and encouragement at the very beginning of my research career could not have been more timely.

I am also grateful to Peter Carr, with whom I was fortunate to spend the summer of 2014 at Disney Research Pittsburgh. This gave me the opportunity to work on vision problems with a focus on industrial applications, and allowed me to have fruitful discussions with the very talented interns at Disney. In particular, I want to thank Mohammad Haris Baig for the many discussions on the optimization of compositional quantizers.

I also want to thank my fellow members of the computer vision group at the Laboratory for Computational Intelligence, Ankur Gupta and Frederick Tung. Their passion for applied computer vision to large-scale problems spawned my own interest in large-scale visual retrieval.

Finally, I want to thank my friends, both geographically distant (Anahi, Erica, Fernanda and Silvia) and close (Alice, Hugh, Jaimie and Victor). Words cannot describe how much I value their friendship and appreciate their support.
Chapter 1

Introduction

Computer vision applications often involve computing the similarity of many high-dimensional, real-valued image representations, in a process known as feature matching. When large databases of images are used, this results in significant computational bottlenecks. For example, approaches to structure from motion, such as Photo Tourism \[30\], estimate the relative viewpoint of each image in large collections of photographs by computing the pairwise similarity of several million SIFT \[21\] descriptors, and it is now common for retrieval and classification datasets to feature thousands \[11,19,26\] or millions \[10,32\] of images.

In the context of these computational bottlenecks, vector quantization has established itself as a default approach to scale computer vision applications. Quantization is usually performed on large datasets of local descriptors \(e.g.,\) SIFT \[21\], or global representations \(e.g.,\) VLAD \[17\], Fisher vectors \[25\] or deep features \[8\]. Outside the computer vision community, the problem is also studied in information theory, multimedia retrieval and unsupervised learning.

Vector quantization is usually posed as the search for a set of codewords \(i.e.,\) a codebook and assignments \(i.e.,\) codes that minimize quantization error. The problem can be solved in a straightforward manner with the k-means algorithm which, unfortunately, scales poorly for large codebooks. While larger codebooks achieve lower quantization error, the downside is that encoding and search times scale linearly with the codebook size.

Several algorithms, such as kd-trees and hierarchical k-means, alleviate the
search and encoding problems by indexing the codebook with complex data structures [22], achieving sublinear search time as a trade-off for recall. These approaches, however, have a large memory footprint, because all the uncompressed vectors must be kept in memory, and do not scale well on very large databases.

Another line of research considers approaches with an emphasis on low memory usage, compressing vectors into small binary codes. While for a long time hashing approaches were the dominant trend [14], they were shown to be largely outperformed by Product Quantization (PQ) [18]. PQ is a compositional vector compression algorithm that decomposes the data into orthogonal subspaces, and quantizes each subspace independently. As a result, vectors can be encoded independently in each subspace, and distances between uncompressed queries and the database can be efficiently computed through a series of table lookups.

The conspicuous success of PQ can be attributed to its small memory footprint, low quantization error and fast search (1 million vectors can be searched exhaustively in around 100 milliseconds), which make it very attractive for scaling computer vision applications. In particular, PQ demonstrated impressive performance and great scalability when applied to nearest neighbour search of SIFT features, which are a good fit to the codebook orthogonality of PQ. This stems from the fact that SIFT features are, in fact, independently aggregated every 8 dimensions [21]. However, PQ proved less successful when applied to the semi-structured GIST descriptors [18], and is likely not a good choice for features obtained from deep convolutional networks. This work is, to the best of our knowledge, the first to thoroughly investigate the compression of deep features using vector quantization.

Recently, Babenko and Lempitsky [4] introduced Additive Quantization (AQ), a generalization of PQ that retains its compositional nature, but is able to handle subcodebooks of the same dimensionality as the input vectors. With a few caveats, AQ can also be used for fast approximate nearest neighbour search, and can be applied to fast dot-product evaluation in a straightforward manner. Moreover, AQ was shown to consistently achieve lower quantization error than PQ, motivating more research of similar methods. However, since in AQ the codebooks are no longer pairwise orthogonal (i.e., no longer independent), encoding cannot be done independently in each subspace. In fact, the encoding problem was demonstrated to be equivalent to inference on a fully-connected pairwise Markov Random Field,
which is well-known to be NP-hard [9]. In [4], beam search was proposed as a solution to this problem, but this results in very slow encoding, which greatly limits the scalability of the proposed solution.

In this work, we first analyze PQ and AQ as compositional quantizers, under a framework that makes the simplifying assumptions of PQ with respect to AQ rather evident. We then investigate the computational complexity implications resulting from the differences between AQ and PQ, and finally derive an intermediate approach that retains the expressive power of AQ, while being only slightly slower than PQ. We achieve a modest gain in performance (e.g., 3-4% better retrieval recall) when evaluating our approach on datasets of SIFT and GIST descriptors. However, we demonstrate large improvements, state-of-the-art compression rates and excellent scalability when benchmarking on datasets of features obtained from deep convolutional neural networks, outperforming all our competitors in the 64-128 bit range. For example, when compressing 128-dimensional deep features, our approach achieves up to 15% better recall, and 7% better classification rates in the ILSVRC 2012 dataset. We find these results particularly encouraging, since features obtained from deep networks are likely to replace hand-crafted features in the foreseeable future.

In summary, our approach can be seen as an improvement upon PQ and AQ, and in particular compares favourably to AQ in 3 ways: (i) it consistently achieves similar or lower quantization error (and therefore, lower error than PQ), (ii) it is several orders of magnitude faster and (iii), it is also simpler to implement.

Chapters 1-4 of this thesis present an extended version of work that has been submitted for publication to the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR) 2015 as Martinez, J., Hoos, H. H., & Little, J. J.: Stacked Quantizers for Compositional Vector Compression. This includes an analysis of the computational trade-offs of PQ and AQ, and the derivation of Stacked Quantizers, as well as experimental results on quantization error, nearest neighbour search and large-scale classification with compressed features. Chapters 5 and 6 present additional, original and unpublished work. In Chapter 5 we qualitatively explore the noise introduced by quantization of deep features, visually inspecting the nearest-neighbours of several image queries. Finally, in Chapter 6 we present a preliminary study of local search for encoding.
Chapter 2

Related work

We build upon recent work on compositional vector quantization, with a focus on the compression of features obtained from deep convolutional neural networks. First, we review recent work that has, to a limited extent, investigated the compression of deep features applied to retrieval and classification. We then review, in detail, compositional vector quantization methods typically used in computer vision applications.

2.1 Deep features and their compression

In 2012, Krizhevsky et al. [20] demonstrated impressive performance on image classification, outperforming all their competitors by nearly 10 points on the Image Large Scale Visual Recognition Challenge (ILSVRC). Their method, based on deep convolutional neural networks trained on raw RGB values, greatly departed from the main classification pipelines used in the computer vision community at the time, which consisted mainly of local feature extraction and aggregation in high-dimensional vectors that attempted to capture the statistics of object categories.

The ground-breaking result of Krizhevsky et al. [20] has spawned large amounts of work on the application of deep features to other visual tasks such as classification [20, 31], detection [13, 15] and retrieval [5, 8], demonstrating state-of-the-art performance on every task. In spite of the growing popularity of deep features, we find that no previous work has rigorously addressed their compression. Thus,
the question what is the best way to compress deep learning features? remains largely open. The question is also rather elusive, as more research into deep convolutional neural networks keeps producing more complex and powerful descriptors, and it seems likely that this trend will continue for at least a few years. In this work, we obtained deep features from networks of 7 layers, 5 convolutional and 2 fully connected, which follow the architecture of Krishevsky et al. [20] and yielded state-of-the-art classification results as of July 2014. We note, however, that recent work [29, 31] has shown that up to 19 convolutional layers yield features with significantly better performance (half the error rate of the network from [20]). We expect our work to generalize well to these new features, but their thorough evaluation is left for future work.

In fairness, several papers have considered compressed deep features as part of classification or retrieval pipelines, observing rather graceful degradations in performance. Agrawal et al. [1] binarized deep features for classification by simply thresholding at 0, resulting in 4096-bit long descriptors. Babenko et al. [5] experimented with PCA and discriminative dimensionality reduction down to 16 dimensions for image retrieval. Finally, Chatfield et al. [8] experimented with intra-net compression for final descriptors of 128, 1024 and 2048 dimensions. They later compressed the 128-dimensional deep features using PQ to 128 and 256 bits, and experimented with binarization in the 1024-2048 bit range for retrieval. These papers, however, did not focus on investigating state-of-the-art compression methods on deep features; rather, they simply used compression in their pipelines. To the best of our knowledge, we are the first to thoroughly study the compression of deep features through vector quantization.

We now introduce some notation mostly following [23]. We review the vector quantization problem, the scalability approaches proposed by PQ and AQ, and discuss their advantages and disadvantages.
2.2 Vector quantization

Given a set of vectors \( \mathcal{X} = \{ x_1, x_2, \ldots, x_n \} \), the objective of vector quantization is to minimize the quantization error, \( i.e., \) to determine

\[
\min_{C, b} \frac{1}{n} \sum_{x \in \mathcal{X}} \| x - Cb \|_2^2, \tag{2.1}
\]

where \( C \in \mathbb{R}^{d \times k} \) contains \( k \) cluster centers, and \( b \in \{ 0, 1 \}^k \) is subject to the constraints \( \| b \|_0 = 1 \) and \( \| b \|_1 = 1 \). That is, \( b \) may only index into one entry of \( C \). \( C \) is usually referred to as a codebook, and \( b \) is called a code.

If we let \( X = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^{d \times n} \) contain all the \( x \in \mathcal{X} \), and similarly let \( B = [b_1, b_2, \ldots, b_n] \in \{ 0, 1 \}^{k \times n} \) contain all the codes, the problem can be expressed more succinctly as determining

\[
\min_{C, B} \frac{1}{n} \| X - CB \|_2^2. \tag{2.2}
\]

Without further constraints, one may solve expression 2.2 using the k-means algorithm, which alternatively solves for \( B \) (typically exhaustively computing the distance to the \( k \) clusters in \( C \) for each point in \( X \)) and \( C \) (finding the mean of each cluster) until convergence. The performance of k-means is better as the size of the codebook, \( k \), grows larger but, unfortunately, the algorithm is infeasible for large codebook sizes (for example, \( k = 2^{64} \) clusters would far exceed the memory capacity of current machines). The challenge is thus to handle large codebooks that achieve low quantization error while having low memory overhead.

2.3 Compositional quantization

One way of scaling the codebook size looks at compositional models, where smaller subcodebooks can be combined in different ways to potentially represent an exponential number of clusters. Compositional quantization can be formulated similarly to k-means, but restricted to a series of constraints that introduce interesting computational trade-offs. The objective function of compositional quantization can be
expressed as

\[
\min_{C, b} \frac{1}{n} \sum_{x \in X} \|x - \sum_{i} C_i b_i\|_2^2, \tag{2.3}
\]

that is, the vector \(x\) can be approximated not only by a single codeword indexed by its code \(b\), but by the *addition* of its encodings in a series of codebooks. We refer to the \(C_i\) as *subcodebooks*, and similarly call the \(b_i\) *subcodes*. We let each subcodebook contain \(h\) cluster centres: \(C_i \in \mathbb{R}^{d \times h}\), and each subcode \(b_i\) remains limited to having only one non-zero entry: \(\|b_i\|_0 = 1, \|b_i\|_1 = 1\). Since each \(b_i\) may take a value in the range \([1, 2, \ldots, h]\), and there are \(m\) subcodes, the resulting number of possible cluster combinations is equal to \(h^m\), i.e., superlinear in \(m\). Now we can more succinctly write expression \(2.3\) as

\[
\min_{C, B} \|X - CB\|_2^2 = \min_{C_i, B_i} \|X - [C_1, C_2, \ldots, C_m] \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_m \end{bmatrix}\|_2^2, \tag{2.4}
\]

where \(B_i = [b_{i1}, b_{i2}, \ldots, b_{in}] \in \{0, 1\}^{h \times n}\). As we will show next, AQ, PQ and Optimized Product Quantization (OPQ) [12, 23] belong to this family of models.

### 2.4 Product quantization

PQ, introduced by Jéguo et al. [18] in 2011, can be formulated right away with Eq. \(2.4\) under the constraint that all the subcodebooks be pairwise orthogonal [23]:

\[
\forall i, j : i \neq j \to C_i^T C_j = 0_{h \times h}, \tag{2.5}
\]

that is, \(C\) is block-diagonal [23]:
\[
C = [C_1, C_2, \ldots, C_m] = \\
\begin{bmatrix}
D_1 & 0 & \cdots & 0 \\
0 & D_2 & 0 & \vdots \\
\vdots & \ddots & \ddots & \vdots \\
0 & 0 & \cdots & D_m
\end{bmatrix},
\]  
(2.6)

where the entries \(D_i \in \mathbb{R}^{(d/m) \times h}\) are the only non-zero components of \(C\). This constraint assumes that the data in \(X\) was generated from a series of mutually independent subspaces (those spanned by the subcodebooks \(C_i\)), which is rarely the case in practice. There are, however, some advantages to this formulation.

The subcodebook independence of \(PQ\) offers 3 main advantages,

1. **Fast learning.** Under the orthogonality constraint we can efficiently learn the subcodebooks \(C_i\) by independently running k-means on \(d/m\) dimensions. The complexity of k-means is \(O(nkdi)\) for \(n\) datapoints, \(k\) cluster centres, \(d\) dimensions and \(i\) iterations. \(PQ\) solves \(m d/m\)-dimensional k-means problems with \(h\) cluster centres each, resulting in a complexity of \(O(mnh(d/m)i) = O(nhdi)\); i.e., training \(PQ\) is as complex as solving a k-means problem with \(h\) cluster centres.

2. **Fast encoding.** Once training is done, the encoding of the database can also be performed efficiently in \(O(nhd)\) (in line with k-means), which is essential for very large databases.

3. **Fast distance computation.** Distance computation between a query \(q\) and an encoded vector \(\sum_{i=1}^{m} C_i b_i\) is efficient because the subcodebooks are orthogonal, and therefore the total distance is equal to the sum of the distances in each subspace [18]

\[
\|q - \sum_{i=1}^{m} C_i b_i\|_2^2 = \sum_{i=1}^{m}\|q_i - D_i b_i\|_2^2,
\]  
(2.7)

where \(q = [q_1, q_2, \ldots, q_m]\), and \(q_i \in \mathbb{R}^{d/m}\). These distances can be precomputed for each query and quickly evaluated with \(m\) table lookups. This is called
Asymmetric Distance Computation in [18] and is the mechanism that makes PQ attractive for fast approximate nearest neighbour search.

2.5 Optimized product quantization

One of the main disadvantages of PQ is that $X$ is forced to fit in a model that assumes that the data was generated from statistically independent subspaces. Lower quantization error can be achieved if more degrees of freedom are added to the model. In particular, since rotation is a distance-preserving operation, it seems natural to experiment with codebook rotations that minimize quantization error. In OPQ, the objective function becomes $\min_{R,C,B} \frac{1}{n} \|X - RCB\|_2^2$, (2.8)

where $C$ and $B$ are expanded as in Eq. 2.4 and $R$ belongs to the Special Orthogonal Group $SO(d)$. In this sense, PQ is a special case of OPQ where $R$ is the $d$-dimensional identity matrix: $R = I_d$. Later work by Jégou et al. [17] noted the importance of this additional parameter; however, the optimization of $R$ was deemed “not tractable”, and using a random orthogonal matrix was suggested. In 2013, independently, Ge et al. [12] and Norouzi & Fleet [23] proposed an iterative method similar to Iterative Quantization [14] that optimizes $R$ in expression 2.8.

We notice, however, that the orthogonality constraint is maintained from PQ to OPQ.

Lower quantization error can be achieved if the independence assumption is not enforced, at the cost of more complex encoding and distance computation. These trade-offs were first introduced in [4] and called Additive Quantization (AQ). Next, we briefly review AQ.

2.6 Additive quantization

In AQ, the subspaces spanned by the subcodebooks $C_i$ are not mutually orthogonal (i.e., not mutually independent). Formally, and although not explicitly stated in [4], AQ solves the formulation of Eq. 2.3 without any further constraints. This
makes of AQ a strictly more general model than PQ/OPQ. However, this added expressiveness comes at a cost.

The subcodebook dependence of AQ comes with 3 main disadvantages with respect to PQ/OPQ,

1. **Distance computation.** The distance between a query \( q \) and a encoded vector \( \sum_{i=1}^{m} C_i b_i \) cannot be computed with \( m \) table lookups. However, it can be found using the identity

\[
\|q - \sum_{i=1}^{m} C_i b_i\|_2^2 = \|q\|_2^2 - \sum_{i=1}^{m} 2\langle q, C_i b_i \rangle + \| \sum_{i=1}^{m} C_i b_i \|_2^2 \tag{2.9}
\]

where the first term is a constant and does not affect the query ranking; the second term can be precomputed and stored for fast evaluation with \( m \) table lookups, and the third term can either be precomputed and quantized for each vector in the database (at an additional memory cost), or can be computed on the fly as

\[
\| \sum_{i=1}^{m} C_i b_i \|_2^2 = \sum_{i} \|C_i b_i\|_2^2 + 2 \sum_{i \neq j} \langle C_i b_i, C_j b_j \rangle \tag{2.10}
\]

where the terms can also be precomputed and retrieved in \( m \) table lookups. Thus, AQ has either a time (\( 2m \) vs. \( m \) lookups) or memory overhead (for storing the quantized result of Eq. 2.10) during distance computation with respect to PQ. Although this may sound as a major problem for AQ, it was shown in [4] that sometimes the distortion error gain can be high enough that allocating memory from the code budget to store the result of Eq. 2.10 results in better recall and faster distance computation compared to PQ/OPQ. This motivates us to look for better solutions to the AQ formulation.

2. **Encoding.** For a given set of subcodebooks \( C_i \) and a vector \( x \), encoding amounts to choosing the optimal set of codes \( b_i \) that minimize quantization error

\[
\|x - \sum_{i=1}^{m} C_i b_i\|_2^2. \tag{2.11}
\]

Unfortunately, without the orthogonality constraint the choice
of $b_i$ cannot be made independently in each subcodebook. This means that, in order to guarantee optimality, the search for the best encoding must be done over a combinatorial space of codewords. Moreover, it was shown in [4] that this problem is equivalent to inference on a fully connected pairwise Markov Random Field, which is well-known to be NP-hard [9].

Since brute force search is not possible, one must settle for a heuristic search method. Beam search was proposed as a solution in [4], resulting in rather slow encoding. Beam search is done in $m$ iterations. At iteration $i$ the distance is computed from each of the $b$ candidate solutions to the set of $k \cdot (m - i)$ plausible candidates (in the $m - i$ codebooks that have not contributed to the candidate solution). At the end of the iteration we have $b^2$ candidate solutions, from which the top $b$ are kept as seeds for the next iteration [4]. The complexity of this process is $O(m^2 bhd) = O(m^2 bhd)$, where $b$ is the search depth. As we will show, this makes the original solution of AQ impractical for very large databases.

3. **Training.** Training consists of learning the subcodebooks $C_i$ and subcodebook assignments $b_i$ that minimize expression (2.3). A typical approach is to use coordinate descent by fixing the subcodebooks $C_i$ while updating the codes $b_i$ (encoding), and later fixing $b_i$ while updating $C_i$ (codebook update). As a side effect of slow encoding, we find that training is also very slow in AQ. While this might seem as a minor weakness of AQ (since training is usually done off-line, without tight time constraints), having faster training also means that for a fixed time budget we can handle larger amounts of training data. In the quantization setting, this means that we can use a larger sample to better capture the underlying distribution of the database.

In [4], codebook update is done by solving the over-constrained least-squares problem that arises from Eq. (2.4) when holding $B$ fixed and solving for $C$. Fortunately, this decomposes into $d$ independent subproblems of $n$ equations over $mh$ variables [4]. This corresponds to an optimal codebook update in the least squares sense. We find that compared to encoding this step is rather fast, and thus focus on speeding up encoding.
Chapter 3

Stacked quantizers

We now introduce our proposed approach to compositional quantization.

Within the subcodebook dependence-independence framework introduced in Chapter 2, we can see that PQ and OPQ assume subcodebook independence, while AQ embraces the dependence and tries to solve a more complex problem. As we will show next, there is a fertile middle ground between these approaches. We propose a hierarchical assumption, which has the advantage of being fast to solve while maintaining the expressive power of AQ (see Figure 3.1).

Design goals. Due to the superior performance of AQ, we want to maintain its key property: subcodebook dependence. However, we look for a representation that can compete with PQ in terms of fast training and good scalability, for which fast encoding is essential. We propose to use a hierarchy of quantizers (see Figure 3.2 left), where the vector is sequentially compressed in a coarse-to-fine manner.

3.1 Encoding

Fast encoding is at the heart of our approach. We assume that the subcodebooks $C_i$ have a hierarchical structure, where $C_1$ gives the coarsest quantization and $C_M$ the finest. Encoding is done greedily. In the first step, we choose the code $b_1$ that most minimizes the quantization error $\|x - C_1 b_1\|_2^2$. Since all the subcodebooks
Figure 3.1: A graphical model representation of codebook learning that highlights the assumptions of PQ/OPQ, AQ, and our approach. Here we show 4 subcodebooks: a) PQ and OPQ assume subcodebook independence; b) AQ attempts to solve the fully connected problem; c) we assume a hierarchical structure.

are small, the search for \( b_1 \) can be done exhaustively (as in k-means). Note that in Figure 3.1 this corresponds to solving for the assignment to the top-level codebook which, according to our model, is assumed to have no dependencies.

Next, we compute the first residual \( r_1 = x - C_1 b_1 \). We now quantize \( r_1 \) using the codewords in \( C_2 \), choosing the one that minimizes the quantization error \( \| r_1 - C_2 b_2 \|_2^2 \). Again, in Figure 3.1 this corresponds to solving for the assignment to the second codebook, which only depends on the top one and is now fixed. This process is repeated until we run out of codebooks to quantize residuals, with the last residual \( r_m \) being equal to the total quantization error (see Figure 3.2, right). Now it is clear that we satisfy our first desired property, as the representation is additive in the encodings: \( x \approx \sum_{i=1}^{m} C_i b_i \), and the codewords all are \( d \)-dimensional (i.e., not independent of each other).

The complexity of this step is \( O(mhd) \) for \( m \) subcodebooks, each having \( h \) subcodewords, and a vector of dimensionality \( d \). This corresponds to a slight increase in computation with respect to PQ (\( O(hd) \)), but is much faster than AQ.
Figure 3.2: Encoding as performed with Stacked Quantizers, shown for 4 subcodebooks. Left: the vector is passed through a series of quantizers, with residuals further encoded down the line. Right: A geometric interpretation of our approach. After recursively encoding residuals, the representation is additive in the encodings, and the quantization error is the remaining residual.

(O(m^3 b h d)). Given that encoding is only slightly more expensive than PQ, we can say that we have also achieved our second desired property.

3.2 Initialization

The goal of initialization is to create a coarse-to-fine set of codebooks. This can be achieved by simply performing k-means on X, obtaining residuals by subtracting the assigned codewords, and then performing k-means on the residuals until we run out of codebooks.

Formally, in the first step we obtain C_1 from the cluster centres computed by k-means on X, and we obtain residuals by subtracting R_1 = X - C_1 B_1. In the second step we obtain C_2 from k-means on R_1, and the residuals are refined to R_2 = R_1 - C_2 B_2. This process continues until we run out of codebooks (notice how this both is analogous to, and naturally gives rise to, the fast encoding proposed before). By the
end of this initialization, we have an initial set of codebooks $C = [C_1, C_2, \ldots, C_m]$ that have a hierarchical structure, and with which encoding can be performed in a greedy manner.

The computational cost of this step is that of running k-means on $n$ vectors $m$ times, i.e., $O(mnhdi)$ for subcodebooks of size $h$, dimensionality $d$ and $i$ k-means iterations.

### 3.3 Codebook refinement

The initial set of codebooks can be further optimized with coordinate descent. This step is based on the observation that, during initialization, we assume that in order to learn codebook $C_i$ we only need to know codebooks $C_1, C_2, \ldots, C_{i-1}$. However, after initialization all the codebooks are fixed. This allows us to fine-tune each codebook given the value of the rest.

Although it is tempting to use the least-squares-optimal codebook update proposed in [4], we have found that this tends to destroy the hierarchical subcodebook structure resulting from initialization. Without a hierarchical structure encoding cannot be done fast, which is one of the key properties that we wish to maintain. We therefore propose an ad hoc codebook refinement technique that preserves the hierarchical structure in the codebooks.

Let us define $\hat{X}$ as the approximation of $X$ from its encoding

$$\hat{X} = CB.$$  \hspace{1cm} (3.1)

Now, let us define $\hat{X}^{-i}$ as an approximation to the original dataset $X$ obtained using the learned codebooks $[C_1, C_2, \ldots, C_m]$ and codes $B = [B_1^\top, B_2^\top, \ldots, B_n^\top]^\top$, except for $C_i$, i.e.,

$$\hat{X}^{-i} = \hat{X} - C_iB_i.$$  \hspace{1cm} (3.2)

We can now see that the optimal value of $C_i$ given the rest of the codebooksis obtained by running k-means on $X - X^{i-1}$, i.e., the residual after removing the contribution of the rest of the codebooks. Since we already know the cluster membership to $C_i$ (i.e., we know $B_i$) either from initialization or the previous iteration,
we need to update only the cluster centres instead of restarting k-means (similar to how OPQ updates the codebooks given an updated rotation [12, 23]).

Enforcing codebook hierarchy is of the essence. Therefore, we run our codebook update in a top-down manner. We first update $C_1$ and update all codes. Next, we update $C_2$ and update codes again. We repeat the process until we have updated $C_m$, followed by a final update of the codes. Updating the codes after each codebook update ensures that the codebook hierarchy is maintained. A round of updates from codebooks 1 to $m$ amounts to one iteration of our codebook refinement.

The algorithm involves encoding using $m$ codebooks in the first pass, $m - 1$ in the second pass, $m - 2$ in the third pass and so on until only one set of codes is updated. This means that the time complexity of the codebook refinement procedure is quadratic in the number of codebooks. This is a significant increase with respect to PQ/OPQ, which are linear in $m$ during their training, and is compared to the scaling of AQ. However, notice that the training usually has to be done only once with a small data sample, and database encoding remains efficient. We will show that this makes a huge difference for large datasets in Chapter 4.
Chapter 4

Experiments

In this chapter, we evaluate Stacked Quantizers empirically, and compare our method against previous work on compositional quantization. Our main interest is to reduce quantization error, because it has been demonstrated to lead to better retrieval recall, mean average precision and classification performance [4, 12, 18, 23]. We also demonstrate two applications of our method: (i) approximate nearest neighbour search and (ii) classification performance with compressed features. In all our experiments we use codebooks of size $h = 256$. This is particularly convenient because the codes can be stored in one byte each, which has been the standard practice in previous work [4, 12, 18, 23]. This also means that when we refer to experiments using 2, 4, 8 and 16 codebooks, we are using codes of 16, 32, 64 and 128 bits per vector, respectively.

4.1 Experimental setup

Datasets. Initially, we test our method on 2 datasets, SIFT1M and GIST1M, introduced in [18]. SIFT1M consists of 128-dimensional SIFT [21] descriptors, and GIST1M consists of 960-dimensional GIST [24] descriptors. In SIFT1M 100 000 vectors are given for training, 10 000 for query and 1 000 000 for the database. In GIST1M, 500 000 vectors are given for training, 5 000 for query and 1 000 000 for the database.
Table 4.1: Summary of the datasets on which we test our approach. The two datasets at the top of the table were introduced in [18] and are standard benchmarks in the literature. The four datasets on the bottom are introduced in this work.

<table>
<thead>
<tr>
<th>Name</th>
<th>Structured</th>
<th>Dimensionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIFT1M</td>
<td>Yes</td>
<td>128</td>
</tr>
<tr>
<td>GIST1M</td>
<td>Yes</td>
<td>960</td>
</tr>
<tr>
<td>ConvNet1M-128</td>
<td>No</td>
<td>128</td>
</tr>
<tr>
<td>ConvNet1M-1024</td>
<td>No</td>
<td>1024</td>
</tr>
<tr>
<td>ConvNet1M-2048</td>
<td>No</td>
<td>2048</td>
</tr>
<tr>
<td>ConvNet1M-4096</td>
<td>No</td>
<td>4096</td>
</tr>
</tbody>
</table>

Since hand-crafted features are consistently being replaced by features obtained from deep convolutional neural networks, we also consider a series of datasets of deep features. We call these datasets ConvNet1M-128, ConvNet1M-1024, ConvNet1M-2048 and ConvNet1M-4096. We obtained these datasets by computing deep learning features on the ILSVRC-2012 training dataset [10], using the CNN-M-128, CNN-M-1024, CNN-M-2048 and CNN-M networks provided by Chatfield et al. [7], and then subsampling equally at random from all classes – sampling was done independently for each dataset. This networks follows the architecture proposed by Zeiler and Fergus [33], with the exception that, for CNN-M-\{128, 1024 and 2048\}, the last fully-connected layer was reduced from 4096 to 128, 1024 and 2048 units respectively. It has been shown that this intra-net compression has a minimal effect on classification performance [7] and exhibits state-of-the-art accuracy on image retrieval [8]. In these 4 datasets, we set aside 100 000 vectors for training, 10 000 for query and 1 000 000 for database. See Table 4.1 for a summary of the datasets used in our experiments.

**Baselines.** We compare against 3 baselines. The first one is AQ as described by Babenko and Lempitsky [4], which consists of beam search for encoding and a least-squares codebook update in an iterative manner. As in [4], we set the beam search depth $b$ to 16 during training and to 64 for the database encoding. Al-
though [4] does not mention the number of iterations used during training, we found that 10 iterations reproduce the results reported by the authors and, as we will show, this is already several orders of magnitude slower than our approach. For code lengths of 64 and 128 bits (8 and 16 codebooks respectively) we use the hybrid APQ algorithm suggested in [4], where the dataset is first preprocessed with OPQ, and then groups of 4 subcodebooks are refined independently with AQ. APQ was proposed for practical reasons, as otherwise AQ would require several days to complete given more than 4 subcodebooks: the need for this approximation starts to show the poor scalability of AQ. Since no code for AQ is available, we wrote our own implementation and incorporated the optimizations suggested in [4].

The second baseline is Optimized Product Quantization [12, 23], which was briefly introduced in Chapter 2. We use the publicly available implementation by Norouzi & Fleet[1] and set the number of optimization iterations to 100. The third baseline is Product Quantization [18]. We slightly modified the OPQ code to create this baseline. We also use 100 iterations in PQ.

[1] https://github.com/norouzi/ckmeans
Figure 4.1: Quantization error on SIFT1M and GIST1M. SQ shows a modest advantage on structured features.

4.2 Quantization error

Our quantization results for SIFT1M and GIST1M are shown in Figure 4.1. In SIFT1M, which features short structured vectors, we observe performance competitive with AQ/APQ. This is already good news, given the better scalability of our method. In GIST1M, OPQ achieves a large gain compared to PQ, and this gap is only slightly improved by AQ and SQ. We speculate that, due to the highly-structured nature of GIST and its high dimensionality, OPQ is able to find a rotation that makes the distribution orthogonal in the subspaces spanned by the codebooks of OPQ. Therefore, there is little performance to gain with non-orthogonal codebooks. We do not investigate this hypothesis further here, but direct the reader to the recent work of Heo et al. [16], who have shown promising results on GIST descriptors, reducing both the bias and the variance of the quantization error.

We now benchmark our method on the 128-, 1024-, 2048- and 4096-dimensional deep features from our ConvNet1M datasets. The quantization results on these datasets are shown in Figure 4.2. As expected, due to the unstructured nature of deep features, we observe that AQ and SQ maintain a large advantage over PQ and OPQ, particularly for the small 128-dimensional features. Moreover, our method remains the clear winner for 8 and 16 codebooks, and largely competitive with
Figure 4.2: Quantization error on deep features with different dimensionalities. The non-independent approaches AQ and SQ clearly outperform PQ and OPQ. SQ achieves the best performance when using 8 and 16 codebooks (64 and 128 bits per feature) in all cases.
AQ for 4 codebooks. These results suggest that codebook independence hurts the compression of deep features particularly badly and motivates more research of compositional quantization methods that follow the formulation of expression 2.3.

4.3 Approximate nearest neighbour search

We demonstrate the performance of our method on fast search of $K$ nearest neighbours with recall@$N$ curves [18]. These curves represent the probability of the true $K$ nearest neighbours being in a retrieved list of $N$ neighbours for varying $N$. We set $K = 1$ and observe little variability for other values.

Our retrieval results for SIFT1M, GIST1M and ConvNet1M-128 are shown on Figure 4.3 in these relatively low-dimensional datasets, we also benchmark our method against AQ/APQ. However, when benchmarking on higher-dimensional deep features, we do not run AQ/APQ due to its high computational cost – the method would take several days to train and then encode the base datasets. We focus on the results for deep features in the next Figures, showing our performance for ConvNet1M-128 and ConvNet-1024 on Figure 4.4 and for ConvNet1M-2048 and ConvNet-4096 on Figure 4.5.

We confirm previous observations [4, 12, 23] that correlate quantization error with nearest neighbour search performance. Our method shows performance within 3-4% of AQ on SIFT1M, consistently outperforming PQ and OPQ. On GIST1M, OPQ achieves a very competitive performance, which is barely improved by AQ and SQ.

As expected, our methods shows its best performance when evaluated on datasets of deep convolutional features: SQ largely outperforms PQ and OPQ on all the ConvNet datasets when using 32 and 64 bits. When using 32 bits, the difference in retrieval is as large as 15%, and up to 10% for 64 bits. Curiously, in spite of the large difference in quantization error observed in the training dataset for 128-bit codes, OPQ obtains a comparable retrieval performance to SQ on deep 1024-, 2048- and 4096-dimensional features. We speculate that, in these very high-dimensional spaces, features are so sparsely-located that the order in which they are retrieved is less affected by their quantization error. However, we would expect
our method to maintain its performance advantage for larger datasets. Moreover, the best retrieval performance is always observed on the 128-dimensional deep features, where our methods consistently outperforms all its competitors by $\sim 5\%$.

### 4.4 Large-scale object classification

We study the trade-off in classification performance vs. compression rate on the ILSVRC-2012 dataset using deep learning features. We trained a linear SVM on the 1.2 million uncompressed examples provided, and preprocessed the features with L2 normalization, which was found to improve performance in [7]. The 50,000 images in the validation set were preprocessed similarly and compressed before evaluation. This scenario is particularly useful when one wants to search for objects in large unlabelled datasets [2, 6], and in retrieval scenarios where clas-
Recall@N curves on the ConvNet1M-128 and ConvNet1M-1024 datasets.

Figure 4.4: Recall@N curves on the ConvNet1M-128 and ConvNet1M-1024 datasets.
Figure 4.5: Recall@N curves on the ConvNet1M-2048 and ConvNet1M-4096 datasets.
Figure 4.6: Top-5 classification error on the ILSVRC-2012 dataset as a function of compression. The dotted black line corresponds to performance without compression. The left pane shows performance using 128-dimensional deep features, and the right pane shows performance for 1024-dimensional deep features.

Classifiers are applied to large collections of images in search for high scores [8, 28]. Notice that in this scenario, the only operation needed between the support vectors and the database descriptors is a dot product; as opposed to distance computation, this can be done with $m$ lookups in AQ and SQ, the same as for PQ and OPQ. We report the classification error taking into account the top 5 predictions.

Classification results are shown on Figure 4.6. We observe a similar trend to that seen in our quantization results, with PQ and OPQ consistently outperformed by AQ and SQ. Using 128-dimensional features our method performs similarly to AQ using 4 codebooks, but shows better performance for larger code sizes. Using 1024-dimensional features AQ and SQ are practically equivalent but, curiously, it seems like the 128-dimensional features are more amenable to compression: for all compression rates the 128-dimensional features outperform the 1024-dimensional features ([0.2646, 0.2293, 0.2101] vs. [0.2917, 0.2562, 0.2246] in top-5 error), even though when uncompressed the 1024-dimensional features perform slightly better (0.1999 vs. 0.1893). This suggests that, if quantization is planned as part of a large-scale classification pipeline, low-dimensional features should be preferred
Figure 4.7: Left: Training time vs. Quantization error of the benchmarked methods on the ConvNet1M-128 training dataset (100K features). For clarity, we plot each 50 iterations for PQ and OPQ and each 25 iterations for SQ after initialization. PQ and OPQ complete 100 iterations after 286 and 336 seconds respectively (4.8 and 5.6 minutes), SQ takes ~1500 seconds for initialization and ~1000 seconds for 100 iterations of codebook refinement (42 minutes in total). APQ takes ~2.7 hours for 10 iterations. Right: Encoding of the database set of 1M features. PQ and OPQ take ~5 seconds, SQ ~20 seconds, and APQ ~9.2 hours.

over high-dimensional ones. It is also noticeable that for extreme compression rates (e.g., 32 bits) PQ and OPQ have error rates in the 35-45% range, while AQ and SQ degrade more gracefully and maintain a 25-30% error rate.

4.5 Running times

Figure 4.7 shows the running time for training and database encoding for PQ/OPQ, APQ and SQ on the ConvNet1M-128 dataset using 8 codebooks (64 bits). All measurements were taken on a machine with a 3.20 GHz processor using a single core. We see that SQ obtains most of its performance advantage out of initialization, but codebook refinement is still responsible for a 20% decrease to the final quantization error (0.12 to 0.10). We also see that APQ largely improves upon its OPQ initialization, but these iterations are extremely expensive compared to PQ/OPQ, and 3 iterations take almost as much computation as the entire SQ optimization.
Moreover, encoding the database with the learned codebooks is extremely expensive with APQ (9.2 hours), while for PQ/OPQ and SQ it stays in the 5-20 second range. Projecting these numbers to the encoding of a dataset with 1 billion features such as SIFT1B \cite{18} suggests that PQ/OPQ would need about 1.5 hours to complete, and SQ would need around 6 hours; however, APQ would need around 1.05 years (!). Although all these methods are highly parallelizable, these numbers highlight the importance of fast encoding for good scalability.
Chapter 5

Image retrieval

In Section 4.3 we empirically benchmarked our method on the task of fast approximate nearest neighbour search. This is a task that has several applications in computer vision, such as structure from motion [30] and non-parametric classification [32]. In this chapter, we investigate nearest-neighbour image retrieval in terms of semantic classification. In this scenario, we are not interested in measuring whether the retrieved neighbours are close in the Euclidean space. Rather, given an image, we want to quantify how many of the retrieved neighbours belong to the same class. This can be evaluated with Precision@N curves, which indicate the empirical probability that the top $N$ retrieved images belong to the same class as the query. Figure 5.1 shows our results on this task on the ILSVRC 2012 dataset. We used the same train and query partitions as in Chapter 4, and show results for 64-bit codes from the ConvNet1M-128 dataset. We observe that our method achieves a consistent advantage of $\sim 2\%$ over OPQ, $\sim 4\%$ over PQ in precision for all values of $N$.

5.1 Qualitative retrieval

In this section we show qualitative results of the previous task. Our goal is to observe typical error cases, to get a visual sense of the noise introduced due to quantization error in image retrieval.

We identify two types of errors: shape errors and part errors. Shape errors
Figure 5.1: Precision@N curves on the ILSRVC 2012 dataset.

occur when the shape of an object is highly discriminative, but fine details are lost due to compression artifacts. Part errors occur on objects that are made of many parts (e.g., a car that is made of lights, windows, tires, etc), and this causes confusion with other object categories that share some of the parts (e.g., the tire of a car might make it match a truck, or a motorcycle).

Qualitative results are shown in Figures 5.2, 5.3 and 5.4. In the three figures, we show the query at the left, and the closest 5-nearest neighbours under different compression techniques. Incorrect matches are bordered with yellow. At the top, we show the neighbours returned when using SQ, then using OPQ, and finally using PQ. In Figure 5.2 we show examples of shape errors: as compression gets noisier, we see soccer balls being matched to tennis balls or acorns, and snails matched to turtles. This suggests compression might make features focus more on coarse features, but also more prone to lose fine details. In Figure 5.3 we show examples of part errors: a set of binoculars is matched with photographic cameras, with which it shares a lens; a bib is matched to t-shirts, which also feature round necks, and a pickup truck is matched against racing and suburban cars. Finally, in Figure 5.4 we show examples that observe both shape and part errors; please refer to the caption for further details.
Figure 5.2: Retrieval errors introduced by quantization noise in natural images. Top: the roundness of the snail is incorrectly matched to an oyster in retrieval with PQ and OPQ, and to a turtle in PQ. Middle: A soccer ball is matched against other round objects such as an acorn and a tennis ball. Bottom: a lemon is matched against apples and a lime. Best viewed digitally.
Figure 5.3: Typical retrieval errors in person-made objects. Top: Binoculars are incorrectly matched to photographic cameras. Middle: A bib is incorrectly matched to t-shirts. Bottom: A pickup truck is incorrectly matched to cars. Best viewed digitally.
Figure 5.4: Top: A mantis brings up grasshoppers, which share some structure in the legs, but not the head. Bottom: A picture of a burrito is incorrectly matched with croissants and sandwiches. Best viewed digitally.
Chapter 6

Encoding with local search

In Chapter 2, we introduced a framework that focused on the different formulations that PQ, OPQ and AQ attempt to optimize, and later in Chapter 3 we derived a method that maintains the formulation of AQ, but is focused on keeping fast encoding, being only slightly more complex than PQ in this step. As we pointed out, our approach represents a middle ground between PQ and AQ.

In a similar spirit, we can also examine AQ and SQ under a common framework, with the intent of exploring middle grounds between them. To do so, it is necessary to think about the design choices that characterize AQ and SQ.

6.1 Design choices in compositional quantization

There are three choices one must make when designing a compositional quantizer:

1. Initialization. The optimization process starts with a set of codebooks that will be refined subsequently. In AQ, these codebooks are obtained by, first, generating random codes $B_i$ for the entries in the database, and then solving for the codebooks $C_i$ via least-squared error minimization. In SQ, we opted for initializing the codebooks in a top-down manner, with the goal of creating a hierarchical structure that allows for fast encoding.

2. Codebook update. AQ and SQ also differ in the way that, for a given set of codes $B_i$, the codebooks $C_i$ are updated. In AQ this is obtained via least-squared error minimization, resulting in optimal update. In SQ, since our main goal is to
Table 6.1: Summary of the design choices that make AQ and SQ different.

<table>
<thead>
<tr>
<th></th>
<th>Init</th>
<th>Codebook update</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQ</td>
<td>Random</td>
<td>Least-Squares</td>
<td>Beam search</td>
</tr>
<tr>
<td>SQ</td>
<td>Greedy</td>
<td>Local</td>
<td>Greedy</td>
</tr>
</tbody>
</table>

preserve a codebook hierarchy (and with it fast encoding), we proposed a more localized approach, updating each codebook individually while keeping the rest fixed. For more details about this procedure, please refer to Section 3.3.

3. **Encoding.** Encoding is performed using a modified version of beam search in AQ, while in SQ it is done in a top-down manner, resulting in drastically different running times for the encoding of very large databases (see Section 4.5 for details). In this chapter, we are mainly interested in exploring approaches between sequential, greedy encoding (as performed in SQ), and more sophisticated search (such as the beam search of AQ).

For a summary of the differences between AQ and SQ, see Table 6.1. Looking at these design choices, it is natural to wonder, how important is each design choice? For example, is greedy initialization better than initializing at random? Is local codebook update inherently better than a least-squares update? What will be the performance of different combinations of these approaches, such as a greedy initialization, followed by least-squares codebook updates and greedy encoding? Determining the importance – and the interplay – of these design choices is our first goal. Our second goal is to investigate an intermediate approach between greedy encoding and beam search, to further determine the importance of search during encoding.

### 6.2 Iterative best improvement search

We investigate local search as an intermediate approach between greedy encoding and beam search. Our method of choice is Iterative best improvement search (IBI). In IBI, an initial code configuration is given (which we obtain with greedy encoding), all the neighbouring configurations are explored, and the best neigh-
bour is selected as the new encoding. Since each code is composed of \( m \) subcodes, each of which index into a subcodebook, we define a neighbour configuration as a change in a single subcode of the original configuration. For example, assuming subcodebooks of size \( h = 256 \), the code

\[
B = [B_1, B_2, B_3, B_4] = [008, 056, 021, 253]
\]

has neighbours

\[
[001, 056, 021, 253], [002, 056, 021, 253], [003, 056, 021, 253], \ldots, [256, 056, 021, 253],
\]

\[
[008, 001, 021, 253], [008, 002, 021, 253], [008, 003, 021, 253], \ldots, [008, 256, 021, 253],
\]

\[
:\ \\
[008, 056, 021, 001], [008, 056, 021, 002], [008, 056, 021, 003], \ldots, [008, 056, 021, 256],
\]

resulting in a total of \( h \cdot m \) neighbours. This approach explores more configurations than the greedy encoding of SQ, while remaining linear in \( m \), and is cheaper than the quadratic cost (in \( m \)) incurred by beam search in AQ. Since the performance of IBI is an upper-bound in performance to more greedy approaches such as iterative first improvement search, we focus solely on IBI.

### 6.3 Experimental setup

We can now build different compositional quantizers making different design choices at each step. We explore the importance of initialization with random and greedy initializations, and of encoding with Greedy or IBI search. The local codebook update of SQ was designed with the sole purpose of preserving the codebook hierarchy of SQ, so we exclude it from these experiments.

*Competing quantizers.* We compare four quantizers: RLG, RLI, GLG, and GLI. The first letter indicates the initialization – “R” for random, and “G” for greedy. Codebook update is done via least-squared error minimization, and therefore the
Datasets We test our quantizers on the SIFT1M and ConvNet1M-128 datasets, which show the most promising results for SQ in the evaluations of Chapter 4.

Results and discussion. We report the quantization error achieved by these methods on Figure 6.1. In SIFT1M, we observe that GLI – that is, greedy initialization, least-squares codebook update and IBI encoding – outperforms other local-search methods when using 32 and 64 bits (i.e., 8 and 16 codebooks). However, all the local search methods achieve practically the same performance when using 128 bits, and in all cases they are outperformed by AQ and SQ. On ConvNet1M-128 we observe different behaviours: all the local search methods outperform AQ when using 64 and 128 bits, and achieve performance almost similar to SQ for all codebook sizes. Moreover, while the approaches with greedy initialization perform best in SIFT1M, in ConvNet1M-128 we see quite the opposite. These mixed results make it hard to draw conclusions on the importance of initialization, and we suggest...
more research to further clear this question up.

It is notable that in ConvNet1M-128 AQ shows the worst performance across all methods. This suggests that OPQ is a very bad initialization strategy, and encourages more research into local-search methods for encoding deep features. Since we observe a different behaviour in SIFT1M, we believe it is too early to comment on the general effectiveness of local search for encoding, and rather suggest more experimentation with more datasets and more iterations.
Chapter 7

Conclusions

We have introduced Stacked Quantizers as an effective and efficient approach to compositional vector compression. After analyzing PQ and AQ in terms of their codebook assumptions, we derived a method that combines the best of both worlds, being only slightly more complex than PQ, while maintaining the representational power of AQ. We have demonstrated state-of-the-art performance on datasets of SIFT, GIST and, perhaps most importantly, deep convolutional features. This final result is particularly strong, as we achieve results comparable to PQ and OPQ requiring only half the memory of these approaches. We believe that this result is particularly important, because deep features are likely to replace hand-crafted features in the foreseeable future.

We have also characterized the qualitative results of image retrieval under different compression schemes and quantified image retrieval precision using PQ, OPQ and our method. We also presented a preliminary investigation into the use of local search for encoding, in an attempt to explore the performance of a middle-ground approach between AQ and SQ.

Future work should look at the integration of SQ with non-exhaustive indexing techniques such as the inverted file [18] or the inverted multi-index [3]. We would also like to try other optimization approaches that have proven fruitful in network-like approaches, such as stochastic gradient descent and conjugate gradient.
Bibliography


V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. arXiv

data set for nonparametric object and scene recognition. Transactions on
Pattern Analysis and Machine Intelligence (TPAMI), 30(11):1958–1970,
2008. → pages ii, 1, 29

[33] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional
neural networks. In European Conference on Computer Vision (ECCV),
pages 818–833, 2014. → pages 18