

Nearest neighbor search with compact codes: A decoder perspective

Kenza Amara, Matthijs Douze, Alexandre Sablayrolles, Hervé Jégou

Abstract

Modern approaches for fast retrieval of similar vectors on billion-scaled datasets rely on compressed-domain approaches such as binary sketches or product quantization. These methods minimize a certain loss, typically the mean squared error or other objective functions tailored to the retrieval problem. In this paper, we re-interpret popular methods such as binary hashing or product quantizers as auto-encoders, and point out that they implicitly make suboptimal assumptions on the form of the decoder. We design backward-compatible decoders that improve the reconstruction of the vectors from the same codes, which translates to a better performance in nearest neighbor search. Our method significantly improves over binary hashing methods or product quantization on popular benchmarks.

1 Introduction

The emergence of large-scale databases raise new challenges, one of the most prominent ones being on how to explore efficiently this data. Finding similar vectors in large sets is increasingly important with the emergence of vector *embeddings* that represent data of various modalities [7, 22]. Exact nearest-neighbor search in high-dimensional spaces is intractable [29], which is why researchers and practitioners have resorted to approximate nearest-neighbors (ANN), trading some search accuracy against orders of magnitude gains in response time, and memory consumption. Amongst the techniques widely adopted in industry [16, 12, 26], quantization-based approaches [21], like product [14] or additive quantizers [2, 17, 19, 20], estimate distances based on approximated vector representations.

In this paper, we regard search methods based on compact codes as auto-encoders, and address the problem of improving the decoder for a fixed encoder: we assume that the stage that assigns vectors to codes is fixed, and we examine how to improve decoding if we tolerate some runtime impact. This setting is especially useful in situations where (1) we need backward-compatibility on existing codes, and/or (2) for re-ranking to refine an initial short-list [15].

The motivation behind our method is to exploit the inherent suboptimality of existing decoders, which typically assume that there is no residual mutual information between bits or subindices. Lifting this assumption, we design a decoder that offers a better estimation of the reproduction value (or centroid) associated with binary sketches or structured compact codes employed in multi-codebook quantization. We demon-

strate the potential of uncoupling the encoder and decoder for several effective encoders such as binary codes [10, 24] or product quantization [14]. Our solution relies on a simple neural decoding network. On the BigANN [15] and Deep1M [3] benchmarks, it provides substantial gains w.r.t. the trade-off between reconstruction and memory budget. Noticeably, we use a very efficient encoder for index construction and initial search, like a binary or fast quantizer [1], and use our neural decoder to re-rank a short-list with high-quality neighbors.

2 Preliminaries

In this section, we first present the quantization methods involved approximate nearest neighbor search as auto-encoders. We then discuss popular quantization methods for which our paper proposes to improve the decoder while keeping the encoder fixed.

2.1 Quantization techniques for ANN search

Most vector encoding methods for approximate nearest neighbor search can be interpreted as quantization techniques [11]. A quantizer can be regarded as an auto-encoder of the form

$$x \xrightarrow{f} f(x) = k \in \mathcal{K} \xrightarrow{g} q_k = g(f(x)) \in \mathcal{Q} \subset \mathbb{R}^d, \quad (1)$$

where the input vector $x \in \mathbb{R}^d$ is first mapped by an encoder f into a code $k \in \mathcal{K}$. The encoder f implicitly defines a partitioning of \mathbb{R}^d into $K = |\mathcal{K}|$ disjoint cells $\mathcal{C}_1, \dots, \mathcal{C}_K$, where $\mathcal{C}_k = f^{-1}(k)$. The decoder reconstructs an approximation q_k from the code k , which belongs to the set $\mathcal{Q} = \{q_k\}_{k \in \mathcal{K}}$ of reproduction values. The encoder-decoder $q = g \circ f$ is usually referred to as a quantizer [11].

Lloyd’s optimality conditions. Given cells and their corresponding reproduction values q_k , Lloyd [18] derived two necessary conditions for a quantizer to be optimal in terms of the average squared loss. First x must be assigned its closest reproduction value, which translates to the usual assignment rule to the nearest centroid:

$$f(x) = \operatorname{argmin}_{k' \in \mathcal{K}} \|x - q_{k'}\|^2. \quad (2)$$

This condition defines the optimal quantizer for a given set of reproduction values, whether we can enumerate it or not. Denoting by p the p.d.f. of the input data, the second condition

is that each reproduction value q_k should be the expectation of the vectors assigned to the same cell as

$$q_k = \int_{x \in C_k} p(x) x dx, \quad (3)$$

2.2 Structured vector quantization

The most general form of vector quantization is when the set of reproduction values $\mathcal{Q} = \{q_1, \dots, q_K\}$ is unconstrained, such as the one typically produced by k-means. In the context of coding for distance estimation, a very large number of centroids (typically, 2^{128}) is required to obtain a sufficient precision. It is not feasible to run k-means at that scale.

Product Quantization (PQ). In order to learn fine-grained codebooks, Jégou et al. [14, 25] propose a product quantizer, where the set of centroids \mathcal{Q} is implicitly defined as a Cartesian product of m codebooks $\mathcal{Q} = \mathcal{Q}_1 \times \dots \times \mathcal{Q}_m$. Each codebook \mathcal{Q}_i consists of K' centroids defined in $\mathbb{R}^{\frac{d}{m}}$. The assignment is separable over the m subspaces and produces indexes of the form $k = (k_1, \dots, k_m)$. The advantage is that the total number of centroids is $(K')^m$ with an assignment step to centroid with an efficient complexity in $\mathcal{O}(dK') = \mathcal{O}(dK^{\frac{1}{m}})$, where d denotes the vector dimensionality.

Notation PQm×b. We denote by PQm×b a product quantizer defined by m subquantizers with b -bits subindices. It corresponds to a compact code of size $m \times b$.

Additive quantizers (AQ) generalize this, they define the reproduction values as

$$\mathcal{Q} = \{c_1 + \dots + c_m | c_1 \in \mathcal{Q}_1, \dots, c_m \in \mathcal{Q}_m\}, \quad (4)$$

where $\forall i \mathcal{Q}_i \subset \mathbb{R}^{\frac{d}{m}}$. Similar to product quantization, the indices are tuples. When not ambiguous, we use notation $\mathcal{Q}_i[k]$ for the element indexed by k in \mathcal{Q}_i . Functions implemented as look-up tables (LUTs) can be written as:

$$x \xrightarrow{f} \begin{pmatrix} k_1 = f_1(x) \\ \vdots \\ k_m = f_m(x) \end{pmatrix} \xrightarrow{g} g(f(x)) = \sum_{i=1}^m \mathcal{Q}_i[k_i]. \quad (5)$$

There are different forms of additive quantizers, with different encoder algorithms: the form of their decoders is identical and rely on LUTs as in Eqn. 4. For instance for a residual quantizer [17] the assignment is done sequentially, which is fast but does not guarantee to assign a vector to its closest neighbors. Subsequent additive quantizers, like the ones by Babenko et al. [2], and Local Search Quantization (LSQ) [19, 20] by Martinez et al. improve the trade-off between encoding complexity and reconstruction error.

Optimal centroids for a fixed encoder. Given a set of training vectors $(x_i)_{i=1..n} \in \mathbb{R}^d$ and their codes k_1, \dots, k_m , it is possible to construct an additive decoder (Eqn. 4) that minimizes the ℓ_2 loss. Denoting by $X \in \mathbb{R}^{n \times d}$ the matrix of training vectors,

$C \in \mathbb{R}^{mK' \times d}$ the codebook entries, and converting subindices k_1, \dots, k_m into one-hot vectors stacked in $I \in \{0, 1\}^{n \times mK'}$, the optimal solution [2, 20] is given by

$$\operatorname{argmin}_C \|X - CI\|_2^2 + \lambda \|C\|_2^2, \quad (6)$$

where the first term minimizes the reconstruction error on the training set. As noted by Martinez *et al.* [20], this estimation has numerical stability issues, which is addressed with the regularizer weighted by $\lambda > 0$. This minimization is performed component-wise [2] in closed form and is therefore efficient to obtain.

Distance estimator. At search time, the ANN algorithm estimates the distance $d(x, y)$ or similarity between a query x and each database vector y based on an imperfect representation of y or both x and y . When both the query and database vectors are quantized, it is a Symmetric Distance Comparison (SDC), which approximates any square distance $d(x, y)^2$ by the estimator

$$d_{\text{SDC}}(x, y) = d(q(x), q(y))^2. \quad (7)$$

The asymmetric distance computation (ADC) [14] estimates distances as

$$d_{\text{ADC}}(x, y) = d(x, q(y))^2. \quad (8)$$

In this case the query vector x is not quantized.

Note that the quantization is a lossy operation: the quality of neighbors strongly depends on the estimator and of the quantizer. ADC reduces the quantization noise compared to SDC, which subsequently improves the search quality [14].

Compressed-domain calculation. For both PQ and AQ, the comparison is performed in the compressed domain, one does not need to decompress the database vectors explicitly as discussed by Jégou et al. [14] and [2].

2.3 Hashing based ANN

Binary codes are quantization techniques that derive from Locality-Sensitive hashing (LSH) [4, 13, 9]. In this work we focus on binarization, which ensures that a small Hamming distance between bit vectors implies proximity in the original space for a given metric, for instance cosine [4]. Binarization maps a vector x to a sequence of bits (k_1, \dots, k_m) using m elementary projections u_i : $k_i = \operatorname{sign}(u_i^\top x)$. It is a form of quantization where the reconstruction is possible up to some scaling constant. If the $\{u_i\}_i$ is an orthonormal set, then the reconstruction on the unit-norm ℓ_2 -hyper-sphere as

$$q_k = \frac{1}{\sqrt{d}} \sum_{i=1}^m k_i u_i \propto [u_1, \dots, u_m] \begin{bmatrix} k_1 \\ \vdots \\ k_m \end{bmatrix}. \quad (9)$$

leads to the same ranking as the Hamming distance between the binary k_i . Note, we use an explicit reconstruction to compute ADC for binary vectors.

We consider two training methods for ANN search with binary codes. The first is Iterative Quantization [10] (ITQ). This simple embedding (learned rotation and sign selection) serves as a baseline in numerous publications. The second is

the catalyzer of Sablayrolles *et al.* [24], which produces high-quality binary embeddings with a neural network. We refer the reader to existing reviews for other approaches [28, 27].

2.4 Re-ranking methods

Some Locality-Sensitive Hashing algorithms such as E2LSH [6] rely on a two-stage approach, where (1) a first system selects the most promising neighbor candidate; (2) which are filtered out by a re-ranking system exact distance computation. The VA-file [29] is the ancestor of approximation-based filtering; a first approximation of the vector leads to select a short-list of neighbor candidates. This approximation being too crude, a re-ranking stage computes the exact distance between the query and the exact representation of the vectors in the short-list. This involves a significant amount of extra storage for large databases. Some approaches alleviate this constraint by refining the first-stage approximation with a secondary compact code [15].

2.5 Architectural considerations

Indexing algorithms heavily depend on the hardware on which they are run. Compared to other quantization approaches based on compact codes, binary hashing is less precise but benefits from specific low-level instructions of modern CPUs, like `XOR` and `popcount` that make the distance computation very fast. Quantization methods significantly benefit from algorithms running on the GPU [16]. With smaller PQ codebooks, order(s) of magnitude faster distance comparisons can be obtained by computing ADC distance in registers [1]. This requires to adopt smaller quantization codebooks. For instance, $K'=16$ instead of the more standard setting $K'=256$ with product or additive quantization.

3 Method

Our proposal improves the decoder given an existing encoder, such that our decoder can be used in a re-ranking stage to improve the ranking. There are several advantages to keep a fast encoder in approximate search techniques based on compact codes, noticeably a faster indexing and large-scale search.

In this section we first introduce the binary and quantization-based encoders that we focus on. We evidence sub-optimality in existing approaches on a simple case with a tractable optimal decoder. Then we introduce our approach based on a neural network decoder (denoted NN) illustrated in Figure 1, which we adopt with any type of encoder.

3.1 Towards stronger decoders & a discussion

In Table 1 we give the set of encoders that we consider: we consider popular and state-of-the-art binarization and quantization methods. We indicate the usual decoder and provide their standard decoder in the column “decoder” along with our replacement proposal in the column “proposed decoder”.

For Product Quantization (denoted PQ) and binary codes, we consider 64 bits codes for a more direct comparison with the

Table 1: Encoder-decoder considered in this paper, and proposed decoders that we propose instead for ranking or re-ranking. Our proposal is to design a stronger decoder for each binarization or quantization technique: either we compute the optimal look-up tables (w.r.t. reconstruction), as initially proposed for additive quantization (AQ), or we train a neural network decoder (NN).

encoder	decoder	proposed decoders
ITQ [10]	naive	AQ, NN
Catalyzer [24]	naive	AQ, NN
PQ 16×4 [14, 1]	PQ	AQ, NN
PQ 8×8 [14]	PQ	AQ, NN
LSQ++ [20]	AQ	-

literature. We denote by PQ 8×8 the usual product quantizer defined by $m=8$ subquantizers with 8-bits subindices (i.e., $K'=256$) and by PQ 16×4 a product quantizer such that $m=16$ and $K'=16$.

3.2 AQ: a better decoder for PQ/OPQ

Our first proposal is to adopt the Additive Quantization (AQ) decoder of Eqn. 6 for PQ, and optimized PQ (OPQ). OPQ is a variant of PQ where, similar to ITQ, the method applies a learned rotation before the subspace partitioning [8, 23]. The OPQ decoder is identical to PQ except that it rotates the vector back to compensate for the initial rotation. PQ and OPQ are special cases of AQ. Adopting an AQ decoder instead of the usual PQ decoder implies that we consider specific reconstruction LUTs Q'_i that have d dimensions instead of d/m : the reconstruction is a summation with Eqn. 4 instead of a concatenation. Therefore and in contrast to existing quantization-based methods, we disentangle the look-up tables associated with the encoder from the ones associated with the decoder: we have two sets of look-up tables.

This alleviates the decoding constraint of PQ, where each subindex only contributes to the reconstruction in its own subspace. Since the subspace are not totally independent, even after application of a pre-rotation like OPQ, the AQ decoder improves the reconstruction.

3.3 Binary codes: LUTs reconstruction

We also propose to adopt the AQ decoder of Eqn. 6 to reconstruct binary codes. While the decoding procedure is conceptually identical to the case of PQ and OPQ, in this binary context this choice departs significantly from the current practice in the literature. where there is usually no reconstruction procedure associated with the binarization, or only a simplistic one.

In our case, for a m -dimensional bit vector, we learn m LUTs of size $d\times 2$. Each LUT is indexed by a bit value k_i as $Q'_i[k_i]$. To our knowledge it is the first time that the AQ (strong) decoder is proposed for binarization techniques. It is advantageously combined with ADC to avoid any approximation on the query. As we will see, it provides a significant improvement without extra memory and at a negligible compute-cost when used for re-ranking. The only requirement compared to

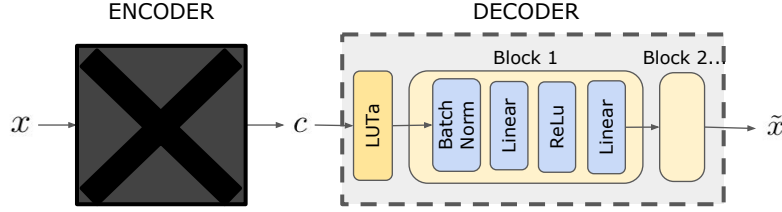


Figure 1: Neural net decoder architecture. The encoder is fixed and we train the decoder to minimize the loss $\|\tilde{x} - x\|^2$ and/or a triplet loss. The first layer of the decoder has the structure of an additive look-up table (LUTa).

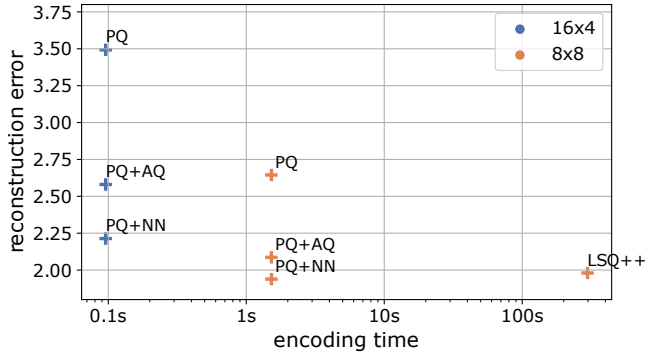


Figure 2: Trade-off between MSE and encoding time for multiple quantizers. LSQ++ (8x8) outperforms regular PQ w.r.t. MSE, but the encoding speed is prohibitively slow. In contrast, PQ16x4 allows for a quick encoding but has a poor reconstruction. We improve the compromises by changing the decoder for a given encoder (PQ+AQ & PQ+NN).

usual binary codes is that the comparison is not context-free: we need to store the lookup tables \mathcal{Q}_i to enable the comparison between a query and a vector, in contrast to the context-free Hamming distance comparison.

This stronger decoder for binary code is backward-compatible in the sense that it can be applied for an existing index of binary codes, with the following requirement: one needs a training set of vectors and corresponding binary codes, which are required to learn the LUTs with Eqn 6.

3.4 Discussion

AQ is the best possible decoder with linear reconstruction as in Eqn. 4. In the literature, different AQ methods differ by how the encoding is performed, which impacts the trade-off between speed and encoding time. However, those offering the best trade-offs like LSQ++ are computationally intensive. In Figure 2 we plot the compromise between encoding time and mean squared error (MSE).

LSQ++ vs PQ8x8. The LSQ++ encoder ($m = 8, K' = 256$) is 2 orders of magnitude slower than its PQ8x8 counterpart. It is also significantly better than PQ. However, with our PQ+AQ, that combines an AQ decoder with a PQ encoder, the gap is reduced significantly. This advocates the choice of a faster encoder.

PQ8x8 vs PQ16x4. The relatively poor reconstruction accuracy associated with a PQ16x4 decoder, when using the corresponding naive decoder, is significantly improved with AQ decoding: it even outperforms PQ8x8 while being one order of magnitude faster, due to the much lower number of centroids per subquantizers (16 versus 256). A key advantage of PQ16x4 is a strong architectural advantage at search time: The look-up table \mathcal{Q}_i can be stored in the process registries [1], leading to an even larger gap in efficiency. Our proposal to leverage such efficient implementation makes this parameter an appealing choice.

3.5 Neural Network decoder

The AQ decoder significantly improves binary codes or product quantization encoders. However the reconstruction linearly depends on the separate reconstructions of the components \mathcal{Q}_i . This is suboptimal: for instance, binary and PQ reconstruct each sub-vector independently of the others, implicitly assuming independence of the codes $\mathbb{P}(k) = \prod_{i=1}^m \mathbb{P}(k_i)$. This independence would be true if the encoding was optimal (there would be no redundant information between the m sub-vectors), but is not true in practice (sub-vectors are not independent). We address this problem by defining a neural network decoder g that, given a compound index (k_1, \dots, k_m) , produces a reconstruction from the index, as shown in Figure 1. The first layer is a structured LUT similar to \mathcal{Q} for which we adopt the same notation as PQ: LUT $m \times b$ indicates that the tensor implementing this layer contains $m \times 2^b \times d$ weights. In our experiments, we use LUT16x4 and LUT8x8 with PQ16x4 and PQ8x8, respectively. For 64-bit binary codes we use LUT64x1.

After the first layer of the decoder (LUT parameters), we stack one or more blocks. Each block consists of a batch normalization and two fully connected layers separated by a ReLU activation function. In the following, we restrict this network to one block, as we observed empirically that more blocks did not provide significant improvements. Note that if the decoder consisted of one LUT followed by an addition, it would be equivalent to the AQ decoder.

3.6 Triplet loss

We optionally consider the triplet loss as an additional term to preserve more explicitly the initial ranking in the reconstruction space, defined as

$$\mathcal{L}_{\text{triplet}} = \max(0, \|x - q(x^+)\|_2^2 - \|x - q(x^-)\|_2^2 + \delta). \quad (10)$$

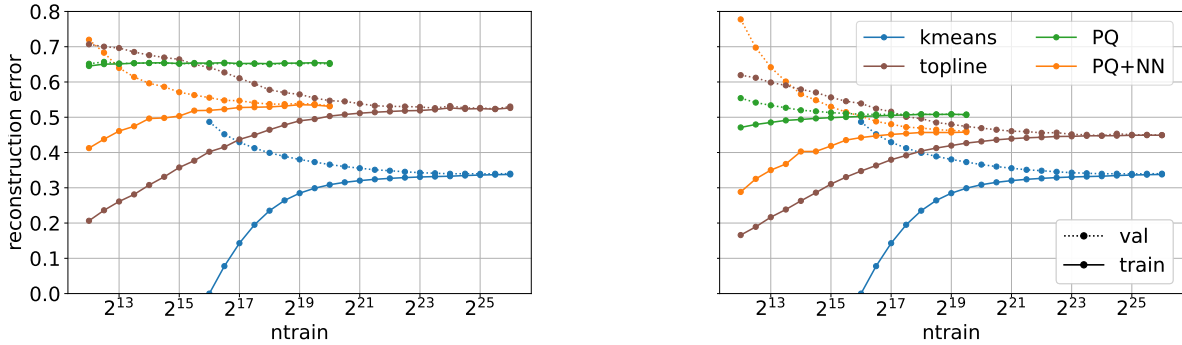


Figure 3: Small-scale experiment on the Deep1M dataset: reconstruction error when varying the size of the training set with 16-bit codes. We compare the **kmeans** baseline with different reconstruction strategies for decoding PQ codes produced by the same PQ encoder: the regular **PQ** decoding, the **topline** for a PQ encoding, and our **PQ+NN** decoder based on a neural network. We report the MSE on the training set (solid) and on the validation set (dashed). For PQ and PQ+NN, we vary the number m and bits per subspace to keep 16-bit codes: (left) PQ4×4: $m = 4$, nbits = 4 and (right) PQ2×8.

In this equation, we consider a query x , a positive match x^+ in a given neighborhood (defined by rank) and a negative match x^- selected to be a hard negative. The margin δ ensures separation between positives and negatives and prevents the weights from collapsing to zero. The overall loss combines the triplet loss and the reconstruction loss, as:

$$\mathcal{L} = \mathcal{L}_{\text{recons}} + \lambda \cdot \mathcal{L}_{\text{triplet}} \quad (11)$$

where $\mathcal{L}_{\text{recons}} = \|x - q(x)\|^2$ is the reconstruction loss and the parameter $\lambda \geq 0$ controls the trade-off between reconstruction and ranking quality. We vary the parameter λ to identify the optimal values where we reach the best recall scores. We retain the range of λ values for which we get the best 100 recall@1. For our two test datasets, a value of $\lambda = 1$ gives near-optimal results.

4 Analysis: A preliminary experiment

While the objective of this paper is to improve the performance of indexing techniques based on compact codes, we first evaluate our proposal to change the decoder on a vanilla quantization task.

The encoder f is the stage that defines the space partitioning. For a given encoder, the optimal decoder g is known and given by Eqn. 3: we refer to it as the “topline”. It can be implemented as a lookup table containing the K d -dimensional centroids. In practical settings ($K > 2^{32}$) the topline computation is not feasible. To circumvent this limitation, we consider a scale where it is feasible to estimate the optimal quantizer: we set $K = 2^{16}$ for the total number of centroids, *i.e.* we consider 16-bit codes.

4.1 Setup of the experiment

At that scale it is possible to run the full k-means quantizer. Therefore we can compare the following encoders:

- the k-means encoder that groups data points in $k = 2^{16}$ clusters. This is the topline encoder for the training set because it minimizes the MSE itself;
- PQ2×8 splits vectors into 2 sub-vectors, each encoded in 8 bits. This is a constrained setting of the k-means encoder, because it is less general;
- PQ4×4 splits vectors into 4 sub-vectors, each encoded in 4 bits. This setting is even more constrained.

For PQ encoders, we compare the decoders:

- the “natural” decoder uses the PQ tables to reconstruct the vectors, *i.e.*, those used by the PQ encoder;
- the topline decoder uses a $K \times d$ size lookup table with the optimal reconstruction from Equation (3). Note that this setting is feasible only in a very small scale like here;
- the neural net (NN) decoder reconstructs the vectors with a small neural net, see Section 3.5.

The k-means encoder can be seen as the product quantizer PQ1×16. For PQ1×16 the “natural” and the topline decoders coincide. In addition, the NN decoder of a PQ1×16 also boils down to a look-up table because all possible inputs of the NN are mapped into a table.

In Figure 3 we measure the MSE as a function of the number of training vectors. Note that the linear additive decoder (PQ as encoder and AQ as decoder) was omitted because AQ is a particular case of the neural network decoder: the AQ decoder is equivalent to our NN decoder with just the LUT layer.

4.2 Training and validation error

In all the settings, when increasing the number of training vectors, we observe the typical behavior of learning algorithms: for few training vectors, the MSE on training is much lower than that on validation vectors (overfitting); for more training vectors, the two errors become identical. This is because of the generalization capacity to unseen data of any algorithm trained on more data.

This transition from overfitting to convergence occurs for all encoder/decoder pairs, but the speed of convergence depends on the capacity of the encoder and decoder: for “natural”

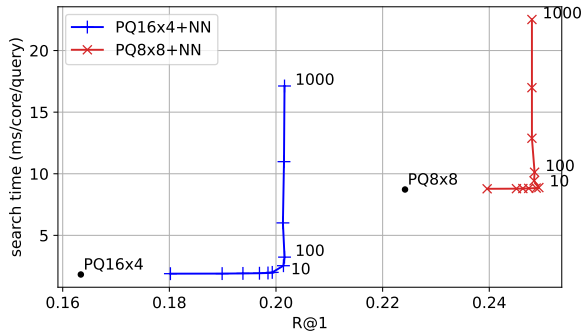


Figure 4: Accuracy vs. search time on the BigANN1M dataset when re-ranking: The NN decoder re-orders the top PQ results. The isolated points correspond to the baseline, i.e., without reranking the short-list. The curves are obtained by sweeping over the number of top elements to re-rank (2 to 1000).

decoders it is faster for PQ2x8 than for k-means because the latter has more parameters to train. It is even faster for PQ4x4. On the decoder front, the topline decoder has the same number of parameters as the regular k-means, so it is not surprising that both converge as slowly. The NN decoder is in-between the topline and natural decoders.

4.3 Discussion

The linear additive decoder achieves at best the same performance as the optimized decoder (PQ as encoder and NN as decoder).

We first compare the “topline” curves with the k-means curves. This quantifies the suboptimality of the encoder because the k-means is an optimal encoder and decoder while the topline has a PQ encoder with an optimal decoder. The difference with the topline is much higher for PQ4x4, which is a particular case (and more constrained) of PQ2x8. Then we compare the “topline” with the “natural” PQ decoder. This shows the contribution of the decoder only. We observe that the gain due to the encoder is a bit smaller than that due to the decoder.

By adding an optimized decoder after the encoding step, we attempt to approach the optimal decoder with a NN that scales beyond this toyish setup. We observe that the NN decoder has an asymptotic accuracy close to that of the topline decoder.

Interestingly our PQ+NN decoder, while asymptotically ($n_{\text{train}} \rightarrow \infty$) inferior to the topline, achieves better performance on the validation set than the topline in the data-starving regime. Our interpretation is that it has to learn fewer parameters and is therefore better able to generalize with less data.

5 Experiments

5.1 Experimental setting

We use publicly available benchmarks to evaluate the performance of nearest neighbor search techniques, namely Bi-

gANN1M [15] ($d = 128$) and DEEP1M [3] ($d = 96$). Both are image features extracted from real images, arranged in a database of 1M vectors, a query set of 10,000 queries, and a separate set of training vectors. We measure the Recall@ R , i.e. the rate of queries for which the nearest neighbor is ranked in the first R ranks, for a code of size 64 bits in all the experiments. The measurements are averaged over 5 runs of training with different random seeds. Our NN decoder minimizes the reconstruction loss with Adam optimizer. We train on 300 epochs with a batch size $bs = 256$ and a learning rate $lr = 5 \cdot 10^{-4}$. We use a scheduler that reduces the learning rate by a factor $lr_{\text{decay}} = 0.5$ when the validation loss stops improving. We do not regularize with weight decay.

5.2 Results with PQ codes

Table 2 compares the deep decoder with baselines in terms of recall for PQ/OPQ encodings. The AQ decoder already improves the accuracy with respect to the PQ/OPQ baseline. We obtain the largest improvement with the neural network (NN) decoder, especially for PQ16x4 codes. This parameter choice seems of high practical interest, since it combines a very fast encoder with a competitive indexing performance.

Re-ranking. We use this approach in a re-ranking setting: since we have a fast decoder (row with Decoder “PQ”) and a slower but more accurate one (Decoder “NN”), we consider a two-stage retrieval procedure, where we first filter out at least 99.9% of the vectors with the fast one.

Figure 4 shows the results of this approach. Most of the accuracy gain is obtained by re-ranking just the top-10 first-level results. Therefore the re-ranking time is negligible w.r.t. the initial search time. The largest gain (3.4 points) is obtained with PQ16x4 codes, that are also the fastest for the first-level decoder.

5.3 Results on binary codes

Table 3 reports results with binary encoders. We consider two encoders: ITQ [10] and the catalyzer [24]. We show how the deep decoder stands amongst popular baselines in term of reconstruction error and recall for binary encodings. Recall that for the AQ solver and the optimized decoder, the lookup table structure is $M \times b = 64 \times 1$. With binary codes, an asymmetric comparison is the element that provides the most significant boost in accuracy, which is shown by the comparison between SDC and ADC.

Our approach ITQ+NN provides an additional gain compared with the ITQ encoder. For the stronger encoder (catalyzer), our approach catalyzer+NN provides a significant improvement on the Deep1M dataset, in particular when adding a triplet loss to make our training more consistent with the one of the catalyzer. However we point that on BigANN1M, our simpler choice of using AQ as a decoder is the best. This may be due to the optimization because formally, the AQ decoder is a particular case of the NN decoder.

Table 2: Retrieval results on BigANN1M and Deep1M with PQ/OPQ 64-bit quantization and ADC comparisons. The LSQ++ results are a topline, very slow at encoding time. All results are computed with ADC. The OPQ (en/de)coder is identical to PQ (en/de)coder up to a learned rotation. The methods introduced in this paper are shaded.

		16×4			8×8		
Encoder	Decoder	R@1	R@10	R@100	R@1	R@10	R@100
BigANN1M							
PQ	PQ	0.168	0.530	0.887	0.223	0.651	0.948
PQ	AQ	0.182	0.564	0.908	0.234	0.667	0.955
PQ	NN	0.202	0.606	0.928	0.239	0.681	0.958
OPQ	OPQ	0.194	0.605	0.937	0.231	0.667	0.960
OPQ	NN	0.202	0.621	0.945	0.225	0.665	0.959
LSQ++	AQ				0.309	0.785	0.987
Deep1M							
PQ	PQ	0.087	0.324	0.703	0.091	0.339	0.730
PQ	AQ	0.083	0.313	0.670	0.094	0.355	0.749
PQ	NN	0.100	0.370	0.756	0.105	0.380	0.776
OPQ	OPQ	0.151	0.493	0.872	0.167	0.538	0.898
OPQ	NN	0.154	0.516	0.889	0.168	0.550	0.908
LSQ++	AQ				0.246	0.688	0.965

5.4 Other limiting factors

For most applications there is a single limiting factor. In this paper we mainly fix the code size and evaluate the encoding accuracy vs. speed tradeoff. However, there are other resource constraints that can become limiting. Concerning the memory requirement of storing the codes of the look-up tables itself, the neural network approach is less parsimonious than fixed (binary) quantizers that don’t need to store centroids in lookup tables. The parameters of a neural network decoder exceed that of a linear additive quantizer because they store LUTs, and also the trained network parameters. Note that the memory usage for the codec is rarely a limiting factor because it is constant w.r.t. the amount of data to process.

The optimized decoder added to a fixed PQ encoder is always the optimal solution in term of accuracy given a fixed encoding time. Note that the NN decoder training time is not a problem in this context: it is several orders of magnitude faster than training image classification networks and can easily be done on CPU.

5.5 Sensitivity to decoder parameters

We analyse the sensitivity of our neural network decoder to variations of the hyper-parameters. We run two analyses: one on the network architecture and the other on the parameters in the decoder training process. In our analysis, all results have been run on the BigANN1M dataset, with a training set of 500000 points and a validation set of 100000 points. The inputs of the network are codes returned by a PQ16×4 encoder.

Architecture. The parameters we consider for the network architecture are the type of blocks (linear or residual), the num-

ber of blocks, the number of neurons in the hidden layers, and the dropout rate. In our case, a residual architecture doesn’t significantly improve the performance of the decoder, probably because the depth of the network is low (only 1 to 3 blocks). We observe on Figure 5a that a linear network with only 2 blocks already outperforms AQ decoders and adding more blocks shows down the inference without adding much more accuracy.

Whatever the number of neurons in the hidden layers, the learning process of the decoder is stable but we reach smaller loss values with more hidden neurons. We experimented with dropout but it did not improve the validation accuracy significantly.

Optimization. The parameters we consider for the decoder optimization are the optimizer, the learning rate, the learning decay factor, the weight decay factor, and the batch size. We compare four optimizers that are commonly used in deep learning: SGD, Adam, Adadelta and RMSprop. We vary the learning rate from $5 \cdot 10^{-3}$ to $5 \cdot 10^{-5}$ to assess their stability. We observe that all networks have a stable learning process and achieve their best accuracy scores for different range of learning rates. We choose Adam optimizer because it is more locally stable and was shown to be faster and more stable than SGD when fine-tuned [5].

Having chosen Adam as the optimizer, we vary more precisely the learning rates. Figure 5b shows that both training and validation losses smoothly decrease and that the decoder is stable with regards to variations of the learning rate. We recommend to use learning rates greater than $5 \cdot 10^{-4}$. They reach a better accuracy than the fixed encoder/decoder PQ in less than 5 epochs. We tested the effect of learning rate decay: varying the learning rate decay factor from 0.2 to 1 has

Table 3: Performance of different binary quantizers (64 bits), on Deep1M and BigANN1M, with ITQ encoding or the neural network encoder of [24]. SDC refers to the case where we compare codes with Hamming distance, ADC when the database vector is reconstructed by Eqn. 9. AQ and NN decoders also use an asymmetrical comparison. The row NN/triplet corresponds to the case where we combine the ℓ_2 loss with a triplet loss similar to the one used to learn the catalyzer, as discussed in Section 3.6. The methods introduced in this paper are shaded, see Table 1.

		BigANN1M			Deep1M		
Encoder	Decoder	R@1	R@10	R@100	R@1	R@10	R@100
ITQ	SDC	0.055	0.220	0.538	0.056	0.213	0.516
ITQ	ADC	0.103	0.383	0.783	0.100	0.368	0.759
ITQ	AQ	0.098	0.372	0.768	0.097	0.362	0.753
ITQ	NN	0.118	0.427	0.819	0.112	0.401	0.790
catalyzer	SDC	0.083	0.298	0.622	0.071	0.254	0.558
catalyzer	ADC	0.158	0.520	0.879	0.137	0.457	0.830
catalyzer	AQ	0.160	0.524	0.881	0.139	0.459	0.833
catalyzer	NN	0.153	0.509	0.865	0.142	0.463	0.834
catalyzer	NN/triplet	0.157	0.519	0.876	0.145	0.471	0.841

no significant effect on the learning process. We draw similar conclusions when varying the weight decay factor from 0 to 0.2. The batch size has no significant influence on the decoder training. After epoch 40, the optimization reaches the same loss values whatever the batch size (128, 256, 512 and 1024).

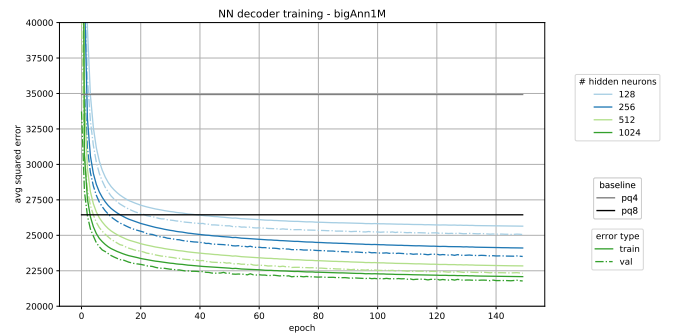
Overall, since the optimization does not appear to be sensitive to hyperparameters, we select the most lightweight architecture and the most natural hyperparameters for all our experiments (see Section 5.1).

6 Conclusion

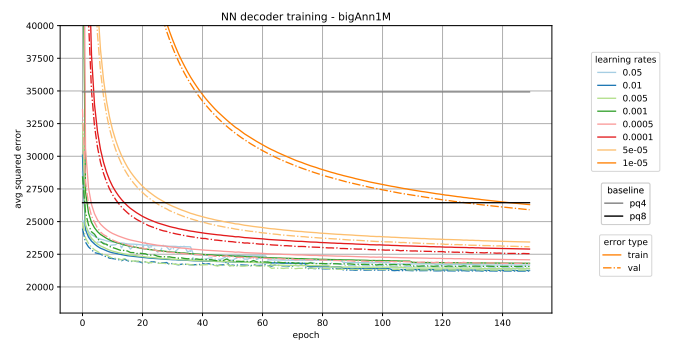
In this paper we have focused on the decoder associated with popular approximate nearest neighbor search based on compact codes. Our main proposal is to design stronger decoders for existing encoders for approximate search. We have evidenced that decoders associated with existing methods are sub-optimal in terms of reconstruction given the indices. We have proposed an enhanced decoder based on a neural network that we use with several types of encodings, such as binary hashing method or product quantization. This optimized decoder improves the accuracy when performing similarity search, and we do not compromise the efficiency since the main use-case of our method is to provide a re-ranking stage.

References

- [1] F. André, A.-M. Kermarrec, and N. Le Scouarnec. Quicker adc: Unlocking the hidden potential of product quantization with simd. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2019.
- [2] A. Babenko and V. Lempitsky. Additive quantization for extreme vector compression. In *Conference on Computer Vision and Pattern Recognition*, 2014.



(a) Number of neurons in hidden layers



(b) Learning rates

Figure 5: Training and validation losses during training of the optimized decoder for different hyperparameters: (a) number of neurons in hidden layers and (b) learning rates. The decoder is a linear neural network with 2 blocks, trained with Adam optimizer on the BigANN1M dataset, with a training set of 500k points and a validation set of 100k points.

- [3] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Conference on Computer Vision and Pattern Recognition*, 2016.
- [4] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. ACM symp. Theory of computing*, 2002.
- [5] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl. On empirical comparisons of optimizers for deep learning. *arXiv preprint arXiv:1910.05446*, 2019.
- [6] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.
- [7] M. Douze, G. Tolias, E. Pizzi, Z. Papakipos, L. Chausot, F. Radenovic, T. Jenicek, M. Maximov, L. Leal-Taixé, I. Elezi, et al. The 2021 image similarity dataset and challenge. *arXiv preprint arXiv:2106.09672*, 2021.
- [8] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Conference on Computer Vision and Pattern Recognition*, 2013.
- [9] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *International Conference on Very Large DataBases*, 1999.
- [10] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2012.
- [11] R. M. Gray and D. L. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6), 1998.
- [12] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 2020.
- [13] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. ACM symposium on Theory of computing*, 1998.
- [14] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2010.
- [15] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *International Conference on Acoustics, Speech, and Signal Processing*, 2011.
- [16] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Trans. on Big Data*, 2019.
- [17] S. Liu, H. Lu, and J. Shao. Improved residual vector quantization for high-dimensional approximate nearest neighbor search. *arXiv preprint arXiv:1509.05195*, 2015.
- [18] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 1982.
- [19] J. Martinez, J. Clement, H. H. Hoos, and J. J. Little. Revisiting additive quantization. In *European Conference on Computer Vision*, 2016.
- [20] J. Martinez, S. Zakhmi, H. H. Hoos, and J. J. Little. LSQ++: lower running time and higher recall in multi-codebook quantization. In *European Conference on Computer Vision*, 2018.
- [21] Y. Matsui, Y. Uchida, H. Jégou, and S. Satoh. A survey of product quantization. *ITE Transactions on Media Technology and Applications*, 2018.
- [22] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [23] M. Norouzi and D. J. Fleet. Cartesian k-means. In *Conference on Computer Vision and Pattern Recognition*, 2013.
- [24] A. Sablayrolles, M. Douze, C. Schmid, and H. Jégou. Spreading vectors for similarity search. *International Conference on Learning Representations*, 2019.
- [25] H. Sandhwalia and H. Jégou. Searching with expectations. In *International Conference on Acoustics, Speech, and Signal Processing*, 2010.
- [26] S. J. Subramanya, R. Kadekodi, R. Krishaswamy, and H. V. Simhadri. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In *Neurips*, 2019.
- [27] J. Wang, W. Liu, S. Kumar, and S.-F. Chang. Learning to hash for indexing big data - a survey. *Proc. of the IEEE*, 2015.
- [28] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.
- [29] R. Weber and S. Blott. An approximation based data structure for similarity search. Technical report, Citeseer, 1997.