Residual Quantization with Implicit Neural Codebooks

Iris A. M. Huijben [1 2 *] **Matthijs Douze** [1] **Matthew Muckley** [1] **Ruud J. G. van Sloun** [2] **Jakob Verbeek** [1]

Abstract

Vector quantization is a fundamental operation for data compression and vector search. To obtain high accuracy, multi-codebook methods represent each vector using codewords across several codebooks. Residual quantization (RQ) is one such method, which iteratively quantizes the error of the previous step. While the error distribution is dependent on previously-selected codewords, this dependency is not accounted for in conventional RQ as it uses a fixed codebook per quantization step. In this paper, we propose QINCo, a neural RQ variant that constructs specialized codebooks per step that depend on the approximation of the vector from previous steps. Experiments show that QINCo outperforms state-of-the-art methods by a large margin on several datasets and code sizes. For example, QINCo achieves better nearest-neighbor search accuracy using 12-byte codes than the state-of-the-art UNQ using 16 bytes on the BigANN1M and Deep1M datasets.

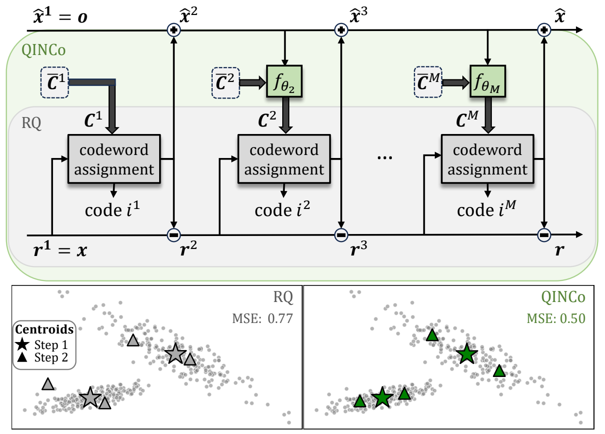


Figure 1. Top: Given a vector $\boldsymbol{x}$, RQ iteratively quantizes the residuals of previous quantization steps, using a single codebook $\boldsymbol{C}^m$ for each step $m = 1, \ldots, M$. QINCo extends RQ by using data-dependent codebooks that are implicitly parameterized via a neural network f_{θ_m} that takes as input a base-codebook $\bar{\boldsymbol{C}}^m$ and partial reconstruction $\hat{\boldsymbol{x}}^m$ of the data vector $\boldsymbol{x}$. Bottom: Toy data example with $M = 2$ quantization steps, each with $K = 2$ centroids. In RQ, codebook centroids in the 2nd level are independent of the 1st level centroids, while QINCo adapts 2nd level centroids to the residuals, reducing the mean-squared-error (MSE) by 35%.

1. Introduction

Vector embedding is a core component of many machine learning systems for tasks such as analysis, recognition, search, matching, and others. Part of the utility of vector embeddings is their adaptivity to different data modalities, such as text (Schwenk & Douze, 2017; Devlin et al., 2018; Izacard et al., 2022) and images (Radford et al., 2021; Pizzi et al., 2022; Ypsilantis et al., 2023). In similarity search and recommender systems (Paterek, 2007), representing entities as vectors is efficient as it enables simple vector comparison. Many techniques and libraries have, nowadays, been developed to search through large collections of embedding vectors (Malkov & Yashunin, 2018; Guo et al., 2020; Morozov & Babenko, 2019; Douze et al., 2024).

[*]Work done when interning at Meta. [1]FAIR at Meta [2]Eindhoven University of Technology. Correspondence to: Matthijs Douze <matthijs@meta.com>, Jakob Verbeek <jjverbeek@meta.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

Vector embeddings can be extracted in different ways, *e.g.* by taking the feature representation of a deep learning model. After extraction, embeddings are typically compressed into fixed-length codes to improve efficiency for storage, transmission, and search. However, a fundamental trade-off exists, where shorter codes introduces higher distortion (Cover & Thomas, 1991), measured as the difference between the initial vector and its decoded approximation. In our work we focus on this vector compression process, and consider the data embedding approach itself as fixed.

A widespread technique to compress embeddings is vector quantization (VQ) (Gray, 1984), which consists of representing each vector with a nearby "prototype" vector. Effective quantizers adapt to the data distribution by learning a codebook of centroids from a representative set of training vectors. The number of distinct centroids grows exponentially with the code size. The k-means VQ algorithm represents all centroids of the codebook explicitly. It tends to be near-optimal, but it does not scale to codes larger than a few bytes because of this exponential growth. Multi-codebook quantization (MCQ) represents centroids as combinations of several codebook entries to avoid the exponential growth. Seminal MCQ techniques —such as product quantization (PQ), residual quantization (RQ), and additive quantization (AQ)— are based on clustering and linear algebra techniques (Jégou et al., 2010; Chen et al., 2010; Babenko & Lempitsky, 2014; Martinez et al., 2016; 2018), while more recent approaches rely on deep neural networks (Yu et al., 2018; Liu et al., 2018; Morozov & Babenko, 2019; Wang et al., 2022; Niu et al., 2023).

Conventional RQ (Chen et al., 2010), being a special case of AQ, iteratively quantizes the residual between the original vector and its reconstruction from the previous quantization steps. Standard RQ methods use a fixed codebook for each quantization step. This is sub-optimal, as the data distribution for the residuals is dependent on previous steps. To address this, we propose a neural variant of RQ. Our method adapts the codebooks at each quantization step using a neural network, leading to large reductions in error rates for the final compressed vectors. We call our method QINCo for **Q**uantization with **I**mplicit **N**eural **Co**debooks. Figure 1 shows the conceptual difference between RQ and QINCo.

In contrast to earlier neural MCQ methods (Morozov & Babenko, 2019; Zhu et al., 2023), QINCo transforms the codebook vectors, rather than the vectors to be quantized. The similarity of QINCo to a standard RQ enables combining it with inverted file indexes (IVF) (Jégou et al., 2010) and re-ranking techniques for fast approximate decoding, making QINCo, as well, suitable for highly accurate large-scale similarity search. Our contributions are as follows:

- We introduce QINCo, a neural residual vector quantizer that — instead of using fixed codebooks — adapts quantization vectors to the distribution of residuals. It is stable to train and has few hyperparameters.
- QINCo sets state-of-the-art performance for vector compression on multiple datasets and rates, and thanks to its compatibility with fast approximate search techniques, it also beats state-of-the-art similarity search performance for high recall operating points.
- QINCo codes can be decoded from the most to the least significant byte, with prefix codes yielding accuracy on par with codes specifically trained for that code length, making QINCo an effective multi-rate codec.

Code can be found at `https://github.com/facebookresearch/QINCo`.

2. Related Work

Vector quantization. A vector quantizer maps a vector $\boldsymbol{x} \in \mathbb{R}^D$ to a vector taken from a finite set of size K (Gray, 1984). This set is called the codebook, and each codebook entry is referred to as "centroid", or "codeword". The objective is to minimize the distortion between $\boldsymbol{x}$ and its quantization. Lloyd's algorithm, a.k.a. k-means, produces a set of codewords, leading to codes of size $\lceil \log_2(K) \rceil$ bits. K-means, however, only scales well up to a few million centroids, resulting in code lengths in the order of 20 bits, which is too coarse for many applications.

Multi-codebook quantization. To scale beyond the inherent limitations of k-means, MCQ uses *several* k-means quantizers, for which various approaches exist. PQ slices vectors into sub-vectors that are quantized independently (Jégou et al., 2010). AQ, on the other hand, represents each vector as a sum of multiple codebook entries (Babenko & Lempitsky, 2014; Martinez et al., 2016; 2018), and RQ progressively quantizes residuals (Chen et al., 2010). We build upon RQ, using neural networks to improve its accuracy by adapting codebooks to residual distributions.

Neural quantization. Neural quantization has been explored to learn discrete data representations, which can be used in discrete sequence models for the generation of images (van den Oord et al., 2017; Esser et al., 2021; Lee et al., 2022; Chang et al., 2022) and audio (Copet et al., 2023). Instead, in this work we focus on discrete representation learning for the purpose of compression and retrieval. Previous works have combined existing MCQ approaches, *e.g.* PQ, with neural encoders for improving compression and/or retrieval of specific data modalities, like images (Agustsson et al., 2017; Liu et al., 2018; Yu et al., 2018; Klein & Wolf, 2019; Jang & Cho, 2021; Wang et al., 2022; El-Nouby et al., 2023), audio (Défossez et al., 2023; Kumar et al., 2023) and graph networks (He et al., 2023). Improvements in these works typically arise from adjustments in the learning objective or improving the optimization of MCQ using regularizers or relaxations, while not fundamentally changing the MCQ procedure itself. On the contrary, in this work, we focus on a fundamental new approach for MCQ, while assuming data embeddings are readily available and fixed.

Most similar to our work are UNQ (Morozov & Babenko, 2019) and DeepQ (Zhu et al., 2023), who also focus on improving MCQ for already-embedded vectors, using neural networks. Both models include a trainable data transformation that precedes the non-differentiable quantization step and, therefore, model the selected quantization vector as a sample from a categorical distribution, for which gradient estimators exist. DeepQ leverages the REINFORCE estimator (Glynn, 1990; Williams, 1992) with additional control variates to reduce its variance, and UNQ uses the straight-through-Gumbel-Softmax estimator (Jang et al., 2017; Maddison et al., 2017) with carefully initialized and trainable Boltzmann temperatures (Huijben et al., 2022). Both models use the nearest centroids, rather than a sampled centroid, for encoding after training. Opposed to UNQ and DeepQ, QINCo transforms the codebooks, rather than the data to be quantized, and thus encodes in the data space directly without leveraging a trainable transformation before quantization. This omits the need for gradient estimation. Moreover, it prevents posterior collapse after which all transformed embeddings are projected on the same centroid, something that requires additional regularization in training of UNQ and DeepQ.

Re-ranking. It is common practice to accelerate large-scale nearest neighbor search with approximation techniques that rely on a cheap distance measure to select a "shortlist" of nearest neighbors, which are subsequently re-ordered using a more accurate measure. This re-ordering can, *e.g.*, be done using a finer quantizer (Jégou et al., 2011) —or in the limit without quantizer (Guo et al., 2020)— compared to the one used for creating the shortlist. It is also possible to *re-interpret* the same codes with a more complex decoding procedure. For example, polysemous codes (Douze et al., 2016) can be compared both as binary codes with Hamming distances, similar to (He et al., 2013), and as PQ codes. UNQ (Morozov & Babenko, 2019) uses a fast AQ for search and re-ranks with a slower decoding network. It has also been shown that in some cases, given a codec, it is possible to train a neural decoder that improves the accuracy (Amara et al., 2022), and use the trained decoder to re-rank the shortlist. To enable fast search with QINCo, we also create a shortlist for re-ranking with a less accurate but faster linear decoder for which —given the QINCo encoder— a closed-form solution is available in the least-squared sense (Babenko & Lempitsky, 2014). On top of the approximate decoding, an inverted file structure (IVF) can direct the search on a small subset of database vectors. UNQ was extended in this way by Noh et al. (2023). We show that the IVF structure integrates naturally with QINCo.

Other connections. In our work a network is used to dynamically parameterize residual quantization codebooks. This is related to weight generating networks, see *e.g.* (Ma et al., 2020), and in a more remote manner to approaches that use one network to perform gradient-based updates of another network, see *e.g.* (Andrychowicz et al., 2016).

3. RQ with Implicit Neural Codebooks

We first briefly review RQ to set some notation; for more details see, *e.g.*, Chen et al. (2010). We use $\boldsymbol{x} \in \mathbb{R}^D$ to denote vectors we aim to quantize using M codebooks of K elements each. Let $\hat{\boldsymbol{x}}^m$ for $m = 1, \dots, M$ be the reconstruction of $\boldsymbol{x}$ based on the first $m-1$ quantization steps, with $\hat{\boldsymbol{x}}^1 := \boldsymbol{0}$. For each step m, RQ learns a codebook $\boldsymbol{C}^m \in \mathbb{R}^{D\times K}$ to quantize the residuals $\boldsymbol{r}^m = \boldsymbol{x} - \hat{\boldsymbol{x}}^m$. We denote the centroids in the columns of $\boldsymbol{C}^m$ as $\boldsymbol{c}_k^m$ for $k = 1, \dots, K$. To encode $\boldsymbol{x}$, at each step the selected centroid is $\boldsymbol{c}_{i^m}^m$, where $i^m = \arg\min_{k=1,\dots,K} \|\boldsymbol{r}^m - \boldsymbol{c}_k^m\|_2^2$. The M quantization indices $\boldsymbol{i} = (i^1, \dots, i^M)$ are finally stored to represent $\boldsymbol{x}$ using $M\lceil \log_2(K)\rceil$ bits. To decode $\boldsymbol{i}$, the corresponding codebook elements are summed to obtain the approximation $\hat{\boldsymbol{x}} = \sum_{m=1}^M \boldsymbol{c}_{i^m}^m$.

3.1. Implicit neural codebooks

At each step of the previously-described RQ scheme, all residuals are quantized with a single step-dependent codebook $\boldsymbol{C}^m$. This is sub-optimal, as in general the distribution of residuals differs across quantization cells. In theory, one could improve upon RQ by using a different specialized codebook for each hierarchical Voronoi cell. In practice, however, as the number of hierarchical Voronoi cells grows exponentially with the number of quantization steps M, training and storing such local codebooks is feasible only for very shallow RQs. For example, for short 4-byte codes with $M = 4$ and $K = 256$ we already obtain over four billion centroids. Since training *explicit* specialized codebooks is infeasible, we instead make these codebooks *implicit*: they are generated by a neural network. The trainable parameters are not the codebooks themselves, but included in the neural network that generates them.

For each quantization step m we train a neural network f_{θ_m} that produces specialized codebook $\boldsymbol{C}^m$ for the residuals $\boldsymbol{r}^m$ in the corresponding hierarchical Voronoi cell. We condition f_{θ_m} upon the reconstruction so far $\hat{\boldsymbol{x}}^m$, and a base codebook $\bar{\boldsymbol{C}}^m$, and use it for improving the K vectors in the mth codebook in parallel: $\boldsymbol{c}_k^m = f_{\theta_m}(\hat{\boldsymbol{x}}^m, \bar{\boldsymbol{c}}_k^m)$. Base codebooks $\bar{\boldsymbol{C}} = (\bar{\boldsymbol{C}}^1, \dots, \bar{\boldsymbol{C}}^M)$ are initialized using a pre-trained conventional RQ, and f_{θ_m} contains residual connections (He et al., 2016) that let the base codebook pass-through, while allowing trainable multi-layer perceptrons (MLPs) to modulate the codebook. This architecture prevents spending many training cycles to achieve RQ baseline performance. The base codebooks are made trainable parameters themselves as well, so that $\bar{\boldsymbol{C}}^m \subset \theta_m$. See Fig. 1 for an overview of QINCo and its relation to RQ.

More precisely, for all K centroids in the mth codebook, f_{θ_m} first projects the concatenation of $\bar{\boldsymbol{c}}_k^m$ and $\hat{\boldsymbol{x}}^m$ using an affine transformation: $\mathbb{R}^{2D} \to \mathbb{R}^D$, after which L residual blocks are used, each containing a residual connection that sums the input to the output of an MLP with two linear layers (ReLU-activated in between): $\mathbb{R}^D \to \mathbb{R}^h \to \mathbb{R}^D$. Since $\hat{\boldsymbol{x}}^1 = \boldsymbol{0}$ by construction, it does not provide useful context for conditioning, so we simply set f_{θ_1} to identity, resulting in $\boldsymbol{C}^1 = \bar{\boldsymbol{C}}^1$. Therefore, the number of trainable parameters $|\theta| = \sum_m |\theta_m|$ in QINCo equals:

$$|\theta| = M \underbrace{KD}_{\bar{\boldsymbol{C}}^m} + (M-1)\Big[\underbrace{(2D^2 + D)}_{\text{concat. block}} + \underbrace{2LDh}_{\text{residual-MLPs}} \Big]. \quad (1)$$

3.2. Encoding, decoding and training

Encoding a vector into a sequence of quantization indices proceeds as in conventional RQ encoding, with the only difference that QINCo constructs the m^{th} codebook via f_{θ_m}, instead of using a fixed codebook per step.

As for decoding, unlike conventional RQ, QINCo follows a sequential process, as codebook-generating network f_{θ_m} requires partial reconstruction $\hat{\boldsymbol{x}}^m$. Given code $\boldsymbol{i}$, for each quantization step $m = 1, \dots, M$ reconstruction follows: $\hat{\boldsymbol{x}}^{m+1} \leftarrow \hat{\boldsymbol{x}}^m + f_{\theta_m}(\hat{\boldsymbol{x}}^m, \bar{\boldsymbol{c}}^m_{i^m})$, with $\hat{\boldsymbol{x}} := \hat{\boldsymbol{x}}^{M+1}$ being the final reconstruction.

To train parameters $\theta = (\theta_1, \dots, \theta_M)$ we perform stochastic gradient decent to minimize the mean-squared-error (MSE) between each residual and the selected centroid. For each quantization step, we optimize the following elementary training objective, defined per data point as:

$$\mathcal{L}^m(\theta) = \min_{k=1,\dots,K} \|\boldsymbol{r}^m - f_{\theta_m}(\hat{\boldsymbol{x}}^m, \bar{\boldsymbol{c}}^m_k)\|_2^2. \tag{2}$$

Note that both $\boldsymbol{r}^m$ and $\hat{\boldsymbol{x}}^m$ implicitly depend on parameters $(\theta_1, \dots, \theta_{m-1})$. Therefore, gradients from later quantization steps propagate back to earlier ones as well. Combining this loss for all M steps yields the final loss:

$$\mathcal{L}_{\text{QINCo}}(\theta) = \sum_{m=1}^{M} \mathcal{L}^m(\theta). \tag{3}$$

4. Large-scale Search with QINCo

For nearest-neighbor search in billion-scale datasets it is prohibitive to exhaustively decompress all vectors with QINCo, and compute distances between the query and the decompressed vectors. The resemblance of QINCo to conventional MCQ enables the use of existing methods to speed up similarity search. To this end, we introduce a fast search pipeline, referred to as IVF-QINCo, that includes IVF (Sec. 4.1), approximate decoding (see Sec. 4.2), and reranking with the QINCo decoder. This pipeline gradually refines the search, and concentrates compute on the most promising database vectors.

4.1. Inverted file index (IVF)

A common technique in large-scale search consists of partitioning the database in K_{IVF} buckets using k-means, and maintaining for each bucket a list of assigned vectors (Jégou et al., 2010). Given a query, only data in the $P_{\text{IVF}} \ll K_{\text{IVF}}$ buckets corresponding to the P_{IVF} centroids closest to the query are accessed to speed up search. In addition, since a database vector is assigned to a bucket, this means that the nearest centroid is the bucket centroid. This prior is used to make the codec more accurate (Noh et al., 2023). IVF integrates naturally with QINCo: each database vector is assigned to one IVF bucket i^{IVF}, and that bucket's centroid is then used as the first estimate $\hat{\boldsymbol{x}}^1 = \boldsymbol{c}_{i^{\text{IVF}}}$ (instead of $\boldsymbol{0}$) of the QINCo code. Thus, in contrast to vanilla QINCo, the first codebook $\bar{\boldsymbol{C}}^1$ is not fixed but generated by (non-identity) f_{θ_1}. The subsequent QINCo coding steps remain the same.

4.2. Approximate decoding

Searching with IVF reduces the number of distance computations by a factor $K_{\text{IVF}}/P_{\text{IVF}}$. However, compared to PQ and RQ, this does not result in competitive search times when combined with QINCo. This is because PQ and RQ, in addition to being cheaper to decode, can benefit from pre-computation of inner products between the query and all codebook elements. Distance computation between the query and a compressed database vector then reduces to summing M pre-computed dot-products per database vector, which amounts to M look-ups and additions (Jégou et al., 2010). Note that, for RQ, when using ℓ_2 distances instead of dot-products for search, the norm of the vectors must also be stored (Babenko & Lempitsky, 2014).

QINCo codebooks are not fixed, so this speed-up by table look-ups can not be applied directly. However, it is possible to fit an additive decoder with fixed and explicit codebooks per quantization level, using codes from the QINCo encoder. This returns approximate distances that can be used to create a short-list of database vectors for which the more accurate QINCo decoder is applied. More precisely, let $\boldsymbol{G} = (\boldsymbol{G}^1, \dots, \boldsymbol{G}^M)$ denote a set of M explicit codebooks, and let $\boldsymbol{g}^m_k \in \mathbb{R}^D$ denote the k^{th} element in the m^{th} codebook. The MSE, defined per data point $\boldsymbol{x}$, yields:

$$\mathcal{L}_{\text{MSE}}(\boldsymbol{G}) = ||\boldsymbol{x} - \sum_{m=1}^{M} \boldsymbol{g}^m_{i^m}||_2^2, \tag{4}$$

where $\sum_{m=1}^{M} \boldsymbol{g}^m_{i^m}$ is the reconstruction of $\boldsymbol{x}$ using code $\boldsymbol{i}$ from the QINCo encoder. This optimization can be solved in closed form (Babenko & Lempitsky, 2014). We refer to this approximate decoder as "AQ decoder".

4.3. Implementation

We implement IVF-QINCo in Faiss (Douze et al., 2024), starting from a standard IVF index with AQ encoding. For each query, we use HNSW (Malkov & Yashunin, 2018) to search the P_{IVF} nearest centroids (Baranchuk et al., 2018) and do compressed-domain distance computations in the corresponding inverted lists (note that, similar to RQ, this requires one additional byte per vector to encode the norms). We retrieve the top-n_{short} nearest vectors with approximate distances from the AQ decoder. Then we run QINCo decoding on the shortlist to compute the final results. See App. A.1 for more implementation details.

5. Experiments

5.1. Experimental setup

Datasets and metrics. We leverage datasets that vary in dimensionality (D) and modality: Deep1B (D=96) (Babenko & Lempitsky, 2016) and BigANN (D=128) (Jégou et al., 2011) are widely-used benchmark datasets for VQ and similarity search that contain CNN image embeddings and SIFT descriptors, respectively. Facebook SimSearchNet++ (FB-ssnpp; D=256) (Simhadri et al., 2022) contains image embeddings intended for image copy detection that were generated using the SSCD model (Pizzi et al., 2022) for a challenge on approximate nearest neighbor search. It is considered challenging for indexing, as the vectors are spread far apart. SIFT1M (D=128) (Jégou et al., 2010) is a smaller-scale dataset of SIFT descriptors used for vector search benchmarks. For all datasets, we use available data splits that include a database, a set of queries and a training set, and we hold out a set of 10k vectors from the original training set for validation, except for the smaller SIFT1M dataset for which we use 5k of the 100k vectors as validations vectors. Lastly, we introduce a new *Contriever* dataset that consists of 21M 100-token text passages extracted from Wikipedia, embedded (D=768) using the Contriever model (Izacard et al., 2022). This model is a BERT architecture (Devlin et al., 2018) fine-tuned specifically for text retrieval. We randomly split the embeddings in 1M database vectors, 10k queries, and 20M training vectors, of which we use 10k as a hold-out validation set.

We report compression performance using MSE on 1M database vectors. To evaluate search performance we additionally report the nearest-neighbor recall percentages at ranks 1, 10 and 100 using 10k non-compressed queries and 1M or 1B compressed database vectors. For resource consumption we focus on parameter counts: since QINCo contains essentially linear layers, the decoding time is proportional to this count, making it a good proxy for run time.

Baselines. We compare QINCo to widely-adopted baselines OPQ (Ge et al., 2013), RQ (Chen et al., 2010), and LSQ (Martinez et al., 2018), for which we use implementations in the Faiss library with default settings(Douze et al., 2024). We also compare to state-of-the-art neural baselines UNQ (Morozov & Babenko, 2019), RVPQ (Niu et al., 2023), and DeepQ (Zhu et al., 2023). RVPQ slices vectors into chunks like PQ and subsequently performs RQ separately in each block rather than using a single quantizer per block. For UNQ, RVPQ and DeepQ we quote performance from the original papers. For UNQ we also reproduced results using the author's public code, and run additional experiments, see App. A.2 for more details.

Training details. We train models on 500k or 10M vectors (except for SIFT1M, that contains only 95k training vectors), and perform early stopping based on the validation loss. During training, all data is normalized by dividing the vector components by their maximum absolute value in the training set. Appendix A.3 provides additional training details.

The number of trainable parameters in QINCo scales linearly with the number of residual blocks L and the hidden dimension h of the residual-MLPs. Preliminary experiments showed that the performance gain of increasing either L or h by the same factor, was very similar, see App. B.1. Therefore, to vary the capacity of QINCo, we varied the number of residual blocks L, and fixed the hidden dimension to $h = 256$. For most experiments we use $M \in \{8, 16\}$ quantization levels and vocabulary size $K = 256$, which we denote as "8 bytes" and "16 bytes" encoding.

Table 1. Comparison of QINCo with state-of-the-art methods in terms of reconstruction error (MSE) and nearest-neighbor search recall (R@1) in percentages. We report QINCo with $L = 16$, except for Contriever1M, where $L = 12$ is used.

		BigANN1M		**Deep1M**		**Contriever1M**		**FB-ssnpp1M**	
		MSE ($\times 10^4$)	R@1	MSE	R@1	MSE	R@1	MSE ($\times 10^4$)	R@1
8 bytes	OPQ	2.95	21.9	0.26	15.9	1.87	8.0	9.52	2.5
	RQ	2.49	27.9	0.20	21.4	1.82	10.2	9.20	2.7
	LSQ	1.91	31.9	0.17	24.6	1.65	13.1	8.87	3.3
	UNQ	1.51	34.6	0.16	26.7	—	—	—	—
	QINCo	**1.12**	**45.2**	**0.12**	**36.3**	**1.40**	**20.7**	**8.67**	**3.6**
16 bytes	OPQ	1.79	40.5	0.14	34.9	1.71	18.3	7.25	5.0
	RQ	1.30	49.0	0.10	43.0	1.65	20.2	7.01	5.4
	LSQ	0.98	51.1	0.09	42.3	1.35	25.6	6.63	6.2
	UNQ	0.57	59.3	0.07	47.9	—	—	—	—
	QINCo	**0.32**	**71.9**	**0.05**	**59.8**	**1.10**	**31.1**	**6.58**	**6.4**

Table 2. Recall values at different ranks for similarity search. QINCo with $L = 4$ is reported.

	4 bytes			**8 bytes**		
	R@1	R@10	R@100	R@1	R@10	R@100
			SIFT1M			
RVPQ	10.2	34.7	74.5	30.3	73.8	97.4
DeepQ	11.0	37.7	76.8	28.0	70.2	96.4
QINCo	**14.9**	**45.5**	**82.7**	**35.8**	**80.4**	**98.6**
			Deep1M			
DeepQ	7.4	30.0	72.5	20.9	62.1	94.1
QINCo	**9.1**	**36.3**	**77.8**	**25.4**	**72.1**	**97.4**

5.2. Quantization performance

In Tab. 1 we compare QINCo against the baselines on four datasets. For Contriever we report QINCo with $L = 12$, for the other datasets we report $L = 16$. QINCo outperforms all baselines on all datasets with large margins. On BigANN for example, QINCo reduces the MSE by 26% and 44% for 8 and 16 bytes encodings respectively, and search recall (R@1) is improved by more than 10 points for both encodings. In general we find that QINCo optimally uses all

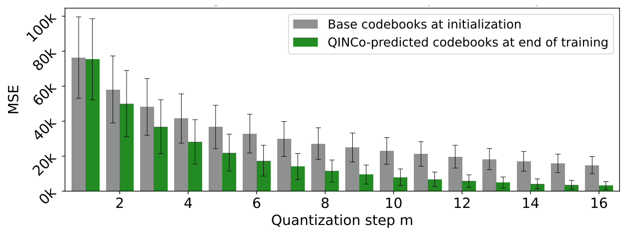


Figure 2. MSE (mean $\pm$ std. dev.) on BigANN1M across 16 quantization steps before training of QINCo ($L = 16$), and after training on 10M samples.

Table 3. Complexity of encoding and decoding per vector (in floating point operations, FLOPS) and indicative timings on 32 CPU cores (in μs) with parameters: D=128; QINCo: L=2, M=8, h=256; UNQ: h'=1024; b=256; RQ: beam size B=5. In practice, at search time for OPQ and RQ we perform distance computations in the compressed domain, which takes M FLOPS (0.16 ns).

	Encoding		**Decoding**	
	FLOPS	time	FLOPS	time
OPQ	$D^2 + KD$	1.5	$D(D+1)$	1.0
RQ	$KMDB$	8.3	MD	1.3
UNQ	$h'(D+h'+Mb+MK)$	18.8	$h'(b+h'+D+M)$	13.0
QINCo	$2MKD(D+Lh)$	823.4	$2MD(D+Lh)$	8.3

codewords without explicitly enforcing this using regularization during training, see App. B.2. Note that the methods that we compare have different numbers of parameters and training set sizes, and also vary in encoding and decoding speed. These factors are analyzed in sections 5.3 and 5.4.

To compare to reported results for DeepQ (Zhu et al., 2023) and RVPQ (Niu et al., 2023), we train a smaller QINCo ($L = 4$) on 100k vectors for Deep1B and 95k vectors for SIFT1M. Table 2 shows that QINCo substantially outperforms these methods as well on both datasets.

Figure 2 shows that QINCo gains accuracy with respect to the base RQ in all quantization steps, but the relative improvement is larger in the deeper ones. An explanation is that for deeper quantization steps, the residual distributions tend to become more heterogeneous across cells, so specialized codebooks predicted by QINCo become more useful.

5.3. Search performance

In Tab. 3 we report the complexity and corresponding encoding/decoding times of QINCo and baselines. All timings are performed on 32 threads of a 2.2 GHz E5-2698 CPU with appropriate batch sizes. In particular for encoding, QINCo is slower than the competing methods both in terms of complexity and timings. Given the encoding complexity of QINCo on CPU, we run encoding on GPU for all QINCo experiments not related to timing. The encoding time for the same QINCo model on a Tesla V100 GPU is 28.4 μs per vector.

Table 4. Search accuracy (R@1) using the approximate AQ decoder only (row 1), AQ in combination with QINCo (with $L = 2$) to re-rank a shortlist of size n_{short} obtained using the AQ decoder (rows 2, 3, 4), and QINCo to decode the full database (row 5).

	BigANN1M		**Deep1M**	
	8 bytes	16 bytes	8 bytes	16 bytes
AQ	12.7	15.6	11.9	17.6
$n_{\text{short}} = 10$	30.5	43.1	25.3	40.3
$n_{\text{short}} = 100$	38.9	62.8	30.3	53.0
$n_{\text{short}} = 1000$	40.1	67.2	31.2	54.9
QINCo	40.2	67.5	31.1	55.0

Table 5. MSE of QINCo and IVF-QINCo for 8- and 16-byte codes on BigANN1M for $L = 4$.

	8 bytes	**16 bytes**
QINCo	1.24×10^4	3.77×10^3
IVF-QINCo	0.78×10^4	2.74×10^3

Since the search speed depends on the decoding speed of the model, we experiment with approximate decoding for QINCo, as described in Sec. 4.2. For each query we fetch n_{short} results using the approximate AQ decoding and do a full QINCo decoding on these to produce the final search results. Table 4 shows that the R@1 accuracy of the approximate AQ decoding is low compared to decoding with QINCo (and compared to RQ). However, re-ranking the top-1000 results (*i.e.*, 0.1% of the database) of the AQ decoder with QINCo brings the recall within 0.3% of exhaustive QINCo decoding.

Only using approximate decoding to create a shortlist does not yield competitive search speeds yet. As such, we experiment with IVF-QINCo on billion-scale datasets, which combines AQ approximate decoding with IVF (see Sec. 4). We use IVF-QINCo with K_{IVF}=10^6 buckets. In terms of pure encoding (*i.e.* without AQ decoding), IVF-QINCo already improves the MSE of regular QINCo thanks to the large IVF quantizer, see Tab. 5.

In Fig. 3 we plot the speed-accuracy trade-offs obtained on BigANN1B (database of size 10^9) using IVF-QINCo, IVF-PQ and IVF-RQ. We report IVF-RQ results and IVF-QINCo with two settings of build-time parameters (number of blocks L for IVF-QINCo and beam size B for IVF-RQ) that adjust the trade-off between search time and accuracy. There are three search-time parameters: P_{IVF}, efSearch (a HNSW parameter) and n_{short}. For each method we evaluate the same combinations of these parameters and plot the Pareto-optimal set of configurations. We observe that there is a continuum from IVF-PQ, via IVF-RQ to IVF-QINCo: IVF-PQ is fastest but its accuracy saturates quickly, IVF-RQ is a bit slower but gains about 5 percentage points of recall; IVF-QINCo is again slower but results in 10 to 20 per-

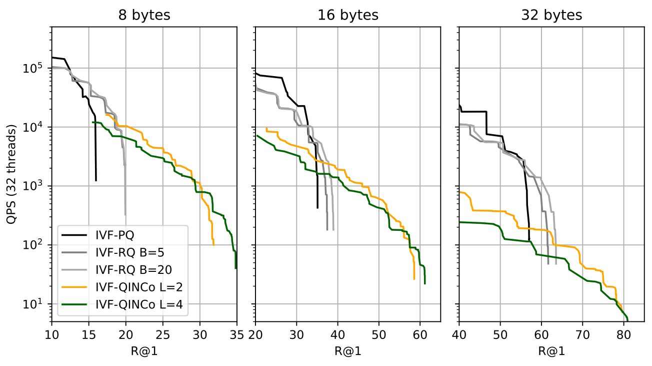


Figure 3. Speed-accuracy trade-off in terms of queries per second (QPS) and recall@1 for IVF-QINCo, on BigANN1B (10^9 vectors), compared to IVF-PQ and IVF-RQ.

centage points of recall above IVF-RQ. The impact of the build-time parameters is significant but does not bridge the gap between the methods. For the operating points where IVF-QINCo is interesting, it can still sustain hundreds to thousands of queries per second. This is the order of speeds at which hybrid memory-flash methods operate (Subramanya et al., 2019), except that QINCo uses way less memory. Appendix B.3 presents additional analyses on fast search with IVF-QINCo.

5.4. Further analyses

Scaling experiments. To investigate the interaction between training set size and model capacity, we train QINCo on both 500k and 10M vectors for codes of 8 and 16 bytes, and vary the number of residual blocks L. Figure 4 shows that in all cases the accuracy significantly improves with more training data, and that given enough training data it keeps improving with larger model capacity L. For less training data (500k vectors), increasing the capacity too much can degrade the accuracy, due to overfitting.

To test whether baselines benefit similarly from more training data, we train OPQ, RQ and LSQ on 10M training vectors. Table S2 in App. B.4 shows that these algorithms hardly benefit from more training data. UNQ was originally trained on 500k training vectors using shallow encoder and decoder designs: both only contained a two-layer MLP with $h'=1024$ hidden dimensions. By increasing either the depth (L') or width (h') of these MLPs, while training on 500k vectors, we found that UNQ suffered from overfitting and test performance decreased (also when deviating from the hyperparameter settings given by the authors). However, training UNQ on 10M vectors improved the MSE for deeper (larger L') and wider (higher h') MLPs. However, when evaluating the number of trainable parameters against MSE performance, Fig. S5 in App. B.4 shows that the Pareto front of these better-performing UNQ models remains far from QINCo's performance.

Dynamic Rates. We evaluate whether a QINCo model trained for long codes can be used to generate short codes, or equivalently, whether partial decoding can be performed by stopping the decoding after $m < M$ steps. Figure 5 shows the MSE per quantization step on BigANN1M for both the 8- and 16-byte models ($L=16$), which is almost identical for $m \leq 8$. This has several benefits: compressed domain rate adjustment (vectors can be approximated by cropping their codes); amortized training cost by only training for the largest M; and simple model management (only a single model is required). This also implies that the loss at step m hardly influences the trainable parameters in steps $< m$. Appendix B.5 shows similar graphs for Deep1M and the R@1 metric for both datasets. They show that with 12 bytes and more, QINCo outperforms 16-byte-UNQ's R@1=59.3% for BigANN1M and R@1=47.9% for Deep1M.

Integration with product quantization. For efficiency when generating large codes, RQ is often combined with PQ to balance sequential RQ stages with parallel PQ coding (Babenko & Lempitsky, 2015; Niu et al., 2023). In this setup, the vector is divided into sub-vectors, and an RQ is trained on each sub-vector. QINCo can equivalently be combined with PQ. We train QINCo and PQ-QINCo ($L=2$) on 10M vectors of FB-ssnpp for 32-byte encoding. Figure 6 shows the trade-off between number of parameters and performance for PQ-QINCo and QINCo. Interestingly, using more PQ blocks deteriorates performance until a turning point, where performance improves again. Vanilla PQ (Jégou et al., 2010) has 65.5k trainable parameters (way fewer than the PQ-QINCo variants) and obtains MSE=55.7k (much worse than PQ-QINCo). Compared to QINCo, PQ-QINCo speeds up encoding and search in high-rate regimes, at the cost of accuracy.

QINCo variant for high-dimensional data. The number of trainable parameters in QINCo scales in $\mathcal{O}(D^2)$, see Equation (1). For high-dimensional embeddings, we propose QINCo-LR, a variant of QINCo that contains an additional low-rank (LR) projection: for each QINCo step, we replace the first affine layer $\mathbb{R}^{2D} \rightarrow \mathbb{R}^D$ by two linear layers that map $\mathbb{R}^{2D} \rightarrow \mathbb{R}^h \rightarrow \mathbb{R}^D$. QINCo-LR scales in $\mathcal{O}(hD)$. We fix $h=256$ (same as the residual blocks) and observe that QINCo-LR (8 bytes; $L=4$) trained on 10M Contriever embeddings achieves a database MSE of 1.46 with 16.71M trainable parameters, as compared to an MSE of 1.45 for vanilla QINCo with 20.85M parameters. QINCo-LR is thus 20% more parameter-efficient, while barely loosing performance, making QINCo-LR interesting for even larger embeddings, as more than 1,000 dimensions is not uncommon (Devlin et al., 2018; Oquab et al., 2023).

Allocating bits. Given a fixed bits budget $M\log_2(K)$, PQ and additive quantizers are more accurate with a few

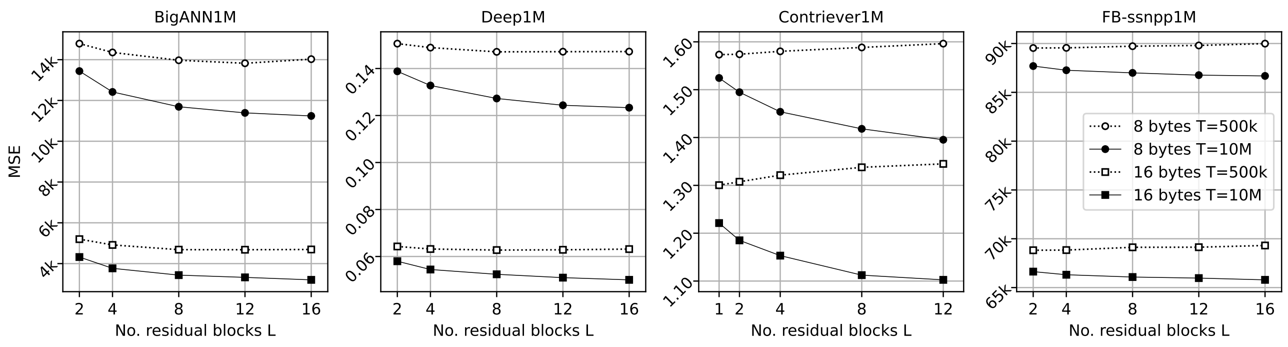


Figure 4. Performance of QINCo of residual blocks L and a training set size T of 500k (open) or 10M (solid).

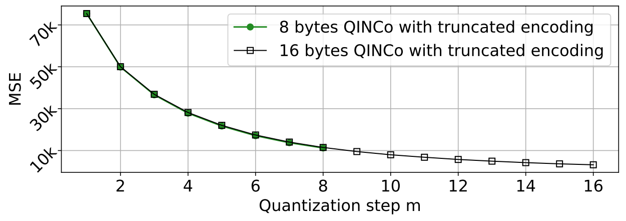


Figure 5. The MSE after the m^{th} quantization step is very similar for the 8 bytes and 16 bytes models for BigANN1M.

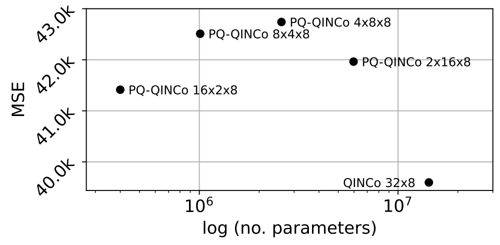


Figure 6. Comparing 32 byte encodings of FB-ssnpp for QINCo and PQ-QINCo. The setting $16 \times 2 \times 8$ means we use 16 PQ blocks, $M = 2$ residual steps and $K = 2^8 = 256$ centroids.

large codebooks (small M, large K) than with many small codebooks (large M, small K), as the latter setting has a lower capacity (fewer trainable parameters). To investigate whether QINCo behaves similarly, we trained QINCo ($T = 500\text{k}, L = 4$, and a base learning rate of 10^{-3}) on BigANN1M with $M = 10$ codebooks with the default $K = 2^8$; and $M = 8$ codebooks with $K = 2^{10}$. Table 6 shows that these two modes of operation are more similar, *i.e.* only 2.1% decrease in MSE, than for RQ and LSQ, for which MSE decreased 11.1% and 6.5%, respectively.

The reason for this different behavior of QINCo with respect to additive quantizers, is that the relation between M, K and the number of trainable parameters in QINCo depends on the number of residual blocks L. For increasing L, the two modes of operation (small M, large K vs small K, large M) get closer in terms of trainable parameters, which reduces the gap in performance.

Table 6. Performance trade-offs on BigANN1M for two QINCo settings that yield 10-byte codes.

	$\mathbf{M=10, K=2^8}$		$\mathbf{M=8, K=2^{10}}$			
	MSE ($\times 10^4$)	R@1	MSE ($\times 10^4$)	R@1	Δ MSE	Δ R@1
RQ	2.07	35.5	1.84	37.2	-11.1%	+4.8%
LSQ	1.55	37.6	1.45	39.3	-6.5%	+4.5%
QINCo	0.96	49.9	0.94	50.1	-2.1%	+0.4%

Additional ablations studies. Finally, we summarize main findings from more ablations presented in App. B.6.

(i) QINCo can be trained using only the MSE loss after the last quantization step, *i.e.* $\mathcal{L}^M(\theta)$, instead of summing the M losses from all quantization steps as in Equation (3). However, this drastically reduced performance and the optimization became unstable.

(ii) QINCo's M losses can be detached, such that each loss only updates the parameters θ_m of one QINCo step. This slightly deteriorated or did not affect MSE, while recall levels remained similar, or slightly improved in some cases. In general, each loss thus has a marginal impact on earlier quantization steps. This corroborates our finding that QINCo can be used with dynamic rates during evaluation.

(iii) The number of trainable parameters in QINCo scales linearly with the number of quantization steps M. To test whether QINCo benefits from having M different neural networks f_{θ_m}, we share (a subset of the) parameters among the M steps and observed drops in performance. Yet, performance remained superior to LSQ in all tested cases.

6. Conclusion

We introduced QINCo, a neural vector quantizer based on residual quantization. QINCo has the unique property that it adapts the codebook for each quantization step to the distribution of residual vectors in the current quantization cell. To achieve this, QINCo leverages a neural network that is conditioned upon the selected codewords in previous steps, and generates a specialized codebook for the next step.

The implicitly-available set of available codebooks grows exponentially with the number of quantization steps, which makes QINCo a very flexible multi-codebook quantizer. We experimentally validate QINCo and compare it to state-of-the-art baselines on six different datasets. We observe substantial improvements in quantization performance, as measured by the reconstruction error, and nearest-neighbor search accuracy. We show that QINCo can be combined with inverted file indexing for efficient large-scale vector search, and that this reaches new high-accuracy operating points. Finally, we find that truncating QINCo codes during encoding or decoding, results in quantization performance that is on par with QINCo models trained for smaller bit rates. This makes QINCo an effective multi-rate quantizer.

QINCo opens several directions for further research, *e.g.* to explore implicit neural codebooks for other quantization schemes such as product quantization, in designs specifically tailored to fast nearest-neighbor search, and for compression of media such as audio, images or videos. On the algorithmic level, we plan to explore the use of beam search during QINCo encoding in future work to investigate whether a possible improvement in accuracy outweighs the added complexity.

Impact Statement

This paper presents work whose goal is to advance the state of the art in data compression and similarity search. Although there are many potential societal consequences of our work, we feel none of them must be specifically highlighted here as our contributions do not enable specific new use cases but rather improve existing ones.

References

Agustsson, E., Mentzer, F., Tschannen, M., Cavigelli, L., Benini, L., and Van Gool, L. Soft-to-hard vector quantization for end-to-end learning compressible representations. In *NeurIPS*, 2017.

Amara, K., Douze, M., Sablayrolles, A., and Jégou, H. Nearest neighbor search with compact codes: A decoder perspective. In *ICMR*, 2022.

Andrychowicz, M., Denil, M., Gómez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. Learning to learn by gradient descent by gradient descent. In *NeurIPS*, 2016.

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

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

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

Baranchuk, D., Babenko, A., and Malkov, Y. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *ECCV*, 2018.

Chang, H., Zhang, H., Jiang, L., Liu, C., and Freeman, W. T. MaskGIT: Masked generative image transformer. In *CVPR*, 2022.

Chen, Y., Guan, T., and Wang, C. Approximate nearest neighbor search by residual vector quantization. *Sensors*, 10(12):11259–11273, 2010.

Copet, J., Kreuk, F., Gat, I., Remez, T., Kant, D., Synnaeve, G., Adi, Y., and Défossez, A. Simple and controllable music generation. In *NeurIPS*, 2023.

Cover, T. M. and Thomas, J. A. *Elements of Information Theory*. John Wiley & Sons, 1991.

Défossez, A., Copet, J., Synnaeve, G., and Adi, Y. High fidelity neural audio compression. *Transactions on Machine Learning Research*, 2023.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of North American Chapter of the Association for Computational Linguistics (NAACL)*, 2018.

Douze, M., Jégou, H., and Perronnin, F. Polysemous codes. In *ECCV*, 2016.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The Faiss library. *arXiv preprint*, 2401.08281, 2024.

El-Nouby, A., Muckley, M. J., Ullrich, K., Laptev, I., Verbeek, J., and Jégou, H. Image compression with product quantized masked image modeling. *Transactions on Machine Learning Research*, 2023.

Esser, P., Rombach, R., and Ommer, B. Taming transformers for high-resolution image synthesis. In *CVPR*, 2021.

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

Glynn, P. W. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*, 33 (10):75–84, 1990.

Gray, R. Vector quantization. IEEE *Transactions on Acoustics, Speech and Signal Processing*, 1(2):4–29, 1984.

Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., and Kumar, S. Accelerating large-scale inference with anisotropic vector quantization. In *ICML*, 2020.

He, K., Wen, F., and Sun, J. K-means hashing: An affinity-preserving quantization method for learning binary compact codes. In *CVPR*, 2013.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, 2016.

He, T., Gao, L., Song, J., and Li, Y.-F. Semisupervised network embedding with differentiable deep quantization. IEEE *Transactions on Neural Networks and Learning Systems*, 34(8):4791–4802, 2023.

Huijben, I. A., Kool, W., Paulus, M. B., and Van Sloun, R. J. A review of the Gumbel-max trick and its extensions for discrete stochasticity in machine learning. IEEE *Transactions on Pattern Analysis and Machine Intelligence*, 45 (2):1353–1371, 2022.

Izacard, G., Caron, M., Hosseini, L., Riedel, S., Bojanowski, P., Joulin, A., and Grave, E. Unsupervised dense information retrieval with contrastive learning. *Transactions on Machine Learning Research*, 2022.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with Gumbel-Softmax. In *ICLR*, 2017.

Jang, Y. K. and Cho, N. I. Self-supervised product quantization for deep unsupervised image retrieval. In *ICCV*, 2021.

Jégou, H., Douze, M., and Schmid, C. Product quantization for nearest neighbor search. IEEE *Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2010.

Jégou, H., Tavenard, R., Douze, M., and Amsaleg, L. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, 2011.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Klein, B. and Wolf, L. End-to-end supervised product quantization for image search and retrieval. In *CVPR*, 2019.

Kumar, R., Seetharaman, P., Luebs, A., Kumar, I., and Kumar, K. High-fidelity audio compression with improved RVQGAN. In *NeurIPS*, 2023.

Lee, D., Kim, C., Kim, S., Cho, M., and Han, W.-S. Autoregressive image generation using residual quantization. In *CVPR*, 2022.

Liu, B., Cao, Y., Long, M., Wang, J., and Wang, J. Deep triplet quantization. In *ACM International conference on Multimedia*, 2018.

Ma, N., Zhang, X., Huang, J., and Sun, J. Weightnet: Revisiting the design space of weight networks. In *ECCV*, 2020.

Maddison, C. J., Mnih, A., and Teh, Y. W. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR*, 2017.

Malkov, Y. A. and Yashunin, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE *Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2018.

Martinez, J., Clement, J., Hoos, H. H., and Little, J. J. Revisiting additive quantization. In *ECCV*, 2016.

Martinez, J., Zakhmi, S., Hoos, H. H., and Little, J. J. LSQ++: Lower running time and higher recall in multi-codebook quantization. In *ECCV*, 2018.

Morozov, S. and Babenko, A. Unsupervised neural quantization for compressed-domain similarity search. In *ICCV*, 2019.

Niu, L., Xu, Z., Zhao, L., He, D., Ji, J., Yuan, X., and Xue, M. Residual vector product quantization for approximate nearest neighbor search. *Expert Systems with Applications*, 232, 2023.

Noh, H., Hyun, S., Jeong, W., Lim, H., and Heo, J.-P. Disentangled representation learning for unsupervised neural quantization. In *CVPR*, 2023.

Oquab, M., Darcet, T., Moutakanni, T., Vo, H., Szafraniec, M., Khalidov, V., Fernandez, P., Haziza, D., Massa, F., El-Nouby, A., et al. DINOv2: Learning Robust Visual Features Without Supervision. *Transactions on Machine Learning Research*, 2023.

Paterek, A. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD cup and workshop*, 2007.

Pizzi, E., Roy, S. D., Ravindra, S. N., Goyal, P., and Douze, M. A self-supervised descriptor for image copy detection. In *CVPR*, 2022.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. Learning transferable visual models from natural language supervision. In *ICML*, 2021.

Schwenk, H. and Douze, M. Learning joint multilingual sentence representations with neural machine translation. In *Workshop on Representation Learning for NLP*, 2017.

Simhadri, H. V., Williams, G., Aumüller, M., Douze, M., Babenko, A., Baranchuk, D., Chen, Q., Hosseini, L., Krishnaswamny, R., Srinivasa, G., et al. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*, 2022.

Subramanya, S. J., Kadekodi, R., Krishaswamy, R., and Simhadri, H. V. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In *NeurIPS*, 2019.

van den Oord, A., Vinyals, O., and Kavukcuoglu, K. Neural discrete representation learning. In *NeurIPS*, 2017.

Wang, J., Zeng, Z., Chen, B., Dai, T., and Xia, S.-T. Contrastive quantization with code memory for unsupervised image retrieval. In *AAAI*, 2022.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Ypsilantis, N.-A., Chen, K., Cao, B., Lipovskỳ, M., Dogan-Schönberger, P., Makosa, G., Bluntschli, B., Seyedhosseini, M., Chum, O., and Araujo, A. Towards universal image embeddings: A large-scale dataset and challenge for generic image representations. In *ICCV*, 2023.

Yu, T., Yuan, J., Fang, C., and Jin, H. Product quantization network for fast image retrieval. In *ECCV*, 2018.

Zhu, X., Song, J., Gao, L., Gu, X., and Shen, H. T. Revisiting multi-codebook quantization. IEEE *Transactions on Image Processing*, 32:2399–2412, 2023.

A. Implementation details

A.1. IVF Faiss implementation

Faiss has a residual quantization implementation combined with an inverted file (IVF-RQ). The corresponding index factory name that we use for the 16-byte experiments is `IVF1048576_HNSW32,RQ16x8_Nqint8`, which gives the number of IVF centroids ($K_{\text{IVF}} = 2^{20}$), indexed with a HNSW graph-based index (32 links per node), the size of the RQ (16×8 bits) and how the norm is encoded for fast search (with an 8-bit integer). To build the IVF-RQ we also set the beam size directly in the index. The 1M IVF centroids are obtained by running k-means on GPU, but otherwise the IVF-RQ experiments run only on CPU, as IVF-RQ is not implementated on GPU in Faiss.

It turns out that this index structure can be used as-is for the IVF-QINCo experiments because the decoder and fast-search functionality of IVF-RQ and IVF-QINCo are the same: both are an AQ decoder. Therefore, we build an IVF-RQ index, set the codebook tables to $\boldsymbol{G}$ (Sec. 4.2) and fill in the index with pre-computed QINCo codes for the databse vectors.

At search time, the Faiss index is used to retrieve the top-n_{short} search results and the corresponding codes (that are extracted from the inverted lists). The decoding and re-ranking is performed in Pytorch. The total search time is thus the sum of (1) the initial search time (that depends on P_{IVF} and `efSearch`), (2) the QINCo decoding time (that depends on n_{short}) and (3) the distance computations and reranking (that are normally very fast).

A.2. Training UNQ

We use the author's code of UNQ (Morozov & Babenko, 2019) to replicate their experimental results and run additional experiments. We noticed that the original code picks the best model based on R@1 accuracy on the query set that was also used to report results, which is overly optimistic for real-world settings. To correct for this, we use the same validation set as in the QINCo experiments, but exploited those vectors as validation queries and picked the best model based on R@1 performance of those. As such, for our UNQ reproductions, recall numbers may be slightly lower than reported in the original paper (Morozov & Babenko, 2019).

We wanted to test the scalability of UNQ, both in terms of model capacity and number of training vectors. However, UNQ's triplet loss requires substantial compute for mining negative samples, as it does a nearest-neighbor search of all vectors in the training set, each time a new set of negatives needs to be drawn. Running this search is feasible on 500k training vectors, as used in the experiments reported in the original UNQ paper, but for 10M vectors it results in infeasible running times where a single negative mining pass takes over eight hours. However, as noted by the UNQ authors in an ablation of their paper (Morozov & Babenko, 2019, Table 5), the triplet loss term does not contribute substantially, and actually decreases performance for R@1 and R@10 for the tested setting (BigANN1M, 8 bytes). As such, we set $\alpha = 0$ in (Morozov & Babenko, 2019, Eq. 12) when running UNQ on 10M vectors, which turns off the triplet loss. This enables scaling experiments to 10M training vectors. UNQ* models in Tab. S2 and all results in Fig. S5 are trained as described above.

A final challenge we faced when training UNQ was instability. When increasing the capacity (either by increasing the width or depth of the encoder/decoder), the training gets stuck due to large gradients when the learning rate is set to 10^{-3} as proposed by the authors. For this reason, we also experimented with a learning rate of 10^{-4}, which stabilized a substantial portion of the runs. For all UNQ experiments reported in this supplemental material, we tested both learning rates (10^{-3} and 10^{-4}), and report the best performing UNQ model.

A.3. Training QINCo

QINCo and its variants were implemented in Pytorch 2.0.1 and trained using the Adam optimizer with default settings (Kingma & Ba, 2015) across eight GPUs with an effective batch size of 1,024. The same seed for randomization was used in all experiments. The base learning rate was reduced by a factor 10 every time the loss on the validation set did not improve for 10 epochs. We stopped training when the validation loss did not improve for 50 epochs. In general this happened within 200–350 epochs, depending on the model size and dataset.

During training, we compute the loss from Equation (3) in two passes: (1) an encoding of the training batch without tracking the gradients, and (2) computation of the loss with gradients when the codes are known. This speeds up the computation $2.5\times$ compared to a naive implementation.

When we trained QINCo on the small training set (*i.e.* T=500k) we noticed that for some datasets, a base learning rate

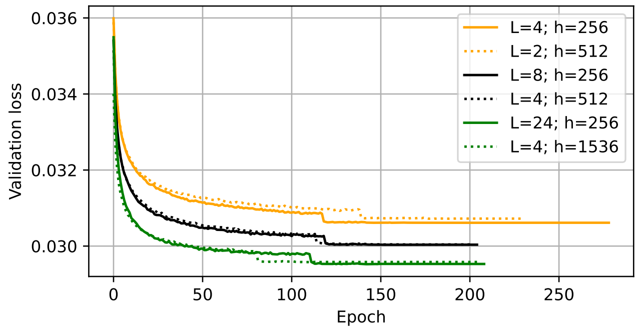


Figure S1. Validation loss on 8 bytes encoding QINCo models trained on 10M BigANN. Changing the model capacity using either L or h with the same factor similarly affects validation loss.

Table S1. Entropy $\mathcal{H}$ of codeword assignments, averaged over codebooks, of the compressed database.

		BigANN1M	**Deep1M**
8 bytes	OPQ	7.90	7.95
	RQ	7.95	7.96
	LSQ	7.95	7.95
	UNQ	8.00	7.99
	QINCo	7.99	7.99
16 bytes	OPQ	7.94	7.93
	RQ	7.97	7.98
	LSQ	7.93	7.94
	UNQ	7.99	7.99
	QINCo	7.99	7.99

of 10^{-3} resulted in slightly better performance than a base rate of 10^{-4}. However for some of the larger QINCo models trained on 10M vectors a lower base learning rate worked better. We opted for a uniform setting of 10^{-4} that can be used in all models and datasets, 10^{-3} was only used when mentioned explicitly in the text.

To initialize the base codebooks $\bar{\boldsymbol{C}}$, we used the RQ implementation from the Faiss library (Douze et al., 2024), with a beam size $B = 1$. This resulted in competitive or slightly better performance than the default $B = 5$, presumably because for QINCo we also used a greedy assignment (equivalent to a beam size of one).

B. Additional analyses

B.1. Capacity of QINCo

The number of trainable parameters scales linearly with both the number of residual blocks L and the hidden dimension h of the residual-MLPs, see Equation (1). Figure S1 plots the validation loss of different 8-bytes QINCo models trained on BigANN. Curves with the same color have the same model capacity, but differ in L and h. It can be seen that changing one or the other has a similar effect on model performance. A slight advantage is visible for increasing L rather than h. For that reason —in order to create only one parameter that influences model capacity— we propose to fix $h{=}256$ and adjust L to change the capacity of QINCo.

B.2. Codeword usage

To investigate whether QINCo suffers from codebook collapse — a common problem in neural quantization models — one can use the average Shannon entropy (averaged over codebooks) to expresses the distribution of selected codewords by the compressed database. It is defined as: $\mathcal{H} = -\frac{1}{M}\sum_{m=1}^{M}\sum_{k=1}^{K} p_k^m \log_2(p_k^m)$, and upper-bounded by $\log_2(K)$ bits. Here, p_k^m is the empirical probability that the k^{th} codeword gets assigned in the m^{th} codebook when compressing the full database.

We find that QINCo achieves near-optimal codeword usage, $\mathcal{H} \approx \log_2(K)$ bits, in all cases, see Tab. S1. Note that UNQ (Morozov & Babenko, 2019) also achieves this, but it requires regularization at training time, which introduces an additional hyperparameter that weighs this regularizing term. Also the authors of DeepQ (Zhu et al., 2023) propose to use

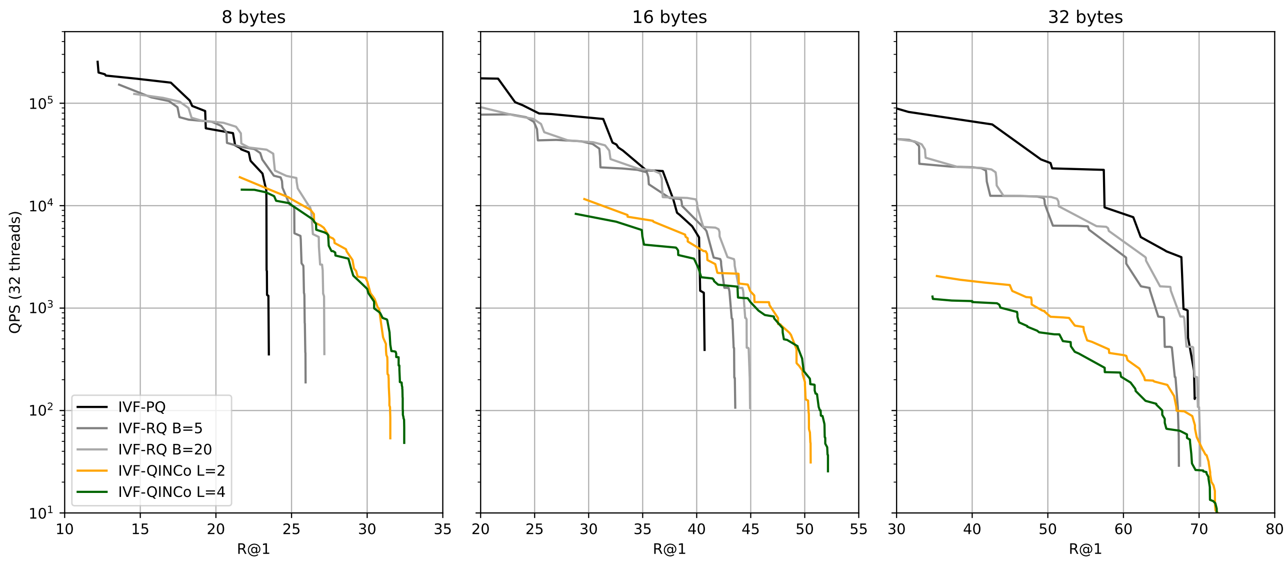


Figure S2. Speed in queries per second (QPS) vs search accuracy (R@1) trade-offs for the Deep1B dataset.

such a regularization term.

The fact that QINCo is not reliant on such additional regularization can be attributed to (i) QINCo is initialized with base codebooks using RQ that enforces a good initial spread of assignments, and (ii) since QINCo does not deploy an encoder before quantization, codebook collapse by the encoder, where all data vectors are mapped to a similar point in latent space, cannot occur.

B.3. Fast search

Results on Deep1B. Figure S2 shows the speed-recall trade-offs for the Deep1B dataset, similar to the results shown for BigANN1B in Fig. 3 of the main paper. There is a wide range of high-accuracy operating points where QINCo is competitive or outperforms IVF-PQ and IVF-RQ for 8 and 16-byte encoding. The trade-offs for the 32-byte setting are less interesting compared to RQ and PQ, because here the upper bound accuracy of QINCo w.r.t. these methods is not high enough. It is possible that PQ-QINCo would be a better option in this case.

Both for BigANN1B (Fig. 3) and Deep1B (Fig. S2), it can be seen that the capacity parameter L slightly changes the Pareto front (green *vs*. yellow curves). At high accuracy operating points, IVF-QINCo with $L=2$ starts to become slower than IVF-QINCo with $L=4$, which seems counter-intuitive. This, however, is caused by the fact that in this regime, IVF-QINCo with $L=2$ requires a longer short-list (higher n_{short}) than IVF-QINCo with $L=4$ to achieve the same accuracy, while at lower accuracies IVF-QINCo with $L=2$ is faster due to its lower decoding complexity.

Decomposing performance over parameters. Pareto-optimal curves do not show the runtime parameters that are used in each experiment. Figure S3 shows all the combination of parameters for a small experiment with 10M database elements and an IVF index of just $2^{16}=64$k centroids. In this case, the IVF centroids are searched exhaustively, without an approximate HNSW index, so there is no `efSearch` parameter involved. This makes it possible to show all parameter combinations. The Pareto-optimal points are indicated in gray squares: they are the ones that give the best accuracy for a given time budget or conversely the fastest search for a given recall requirement.

Figure S4 show the same trade-offs for the BigANN1B dataset for a subset of the parameter sets. It shows that for Pareto-optimal points, the three considered parameters need to be set to "compatible" values: it is useless to set a high P_{IVF} with a low n_{short} and vice-versa. The granularity of the parameter we tried out is relatively coarse. The settings for P_{IVF} are clearly separated and there are probably slightly better operating points for intermediate settings like $n_{\text{short}}=30$ or $n_{\text{short}}=700$.

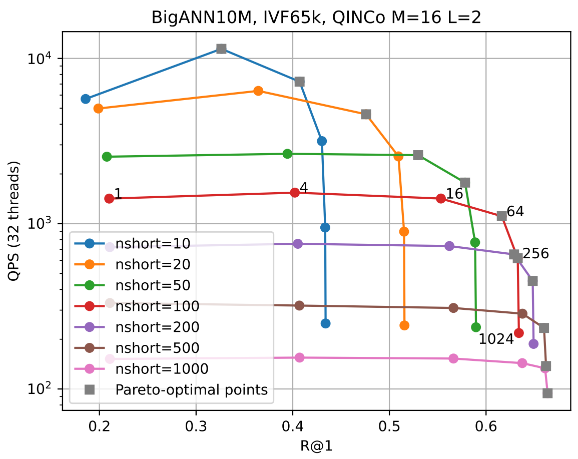


Figure S3. All combinations of P_{IVF} and n_{short} for one dataset. For some points we indicate the P_{IVF} value.

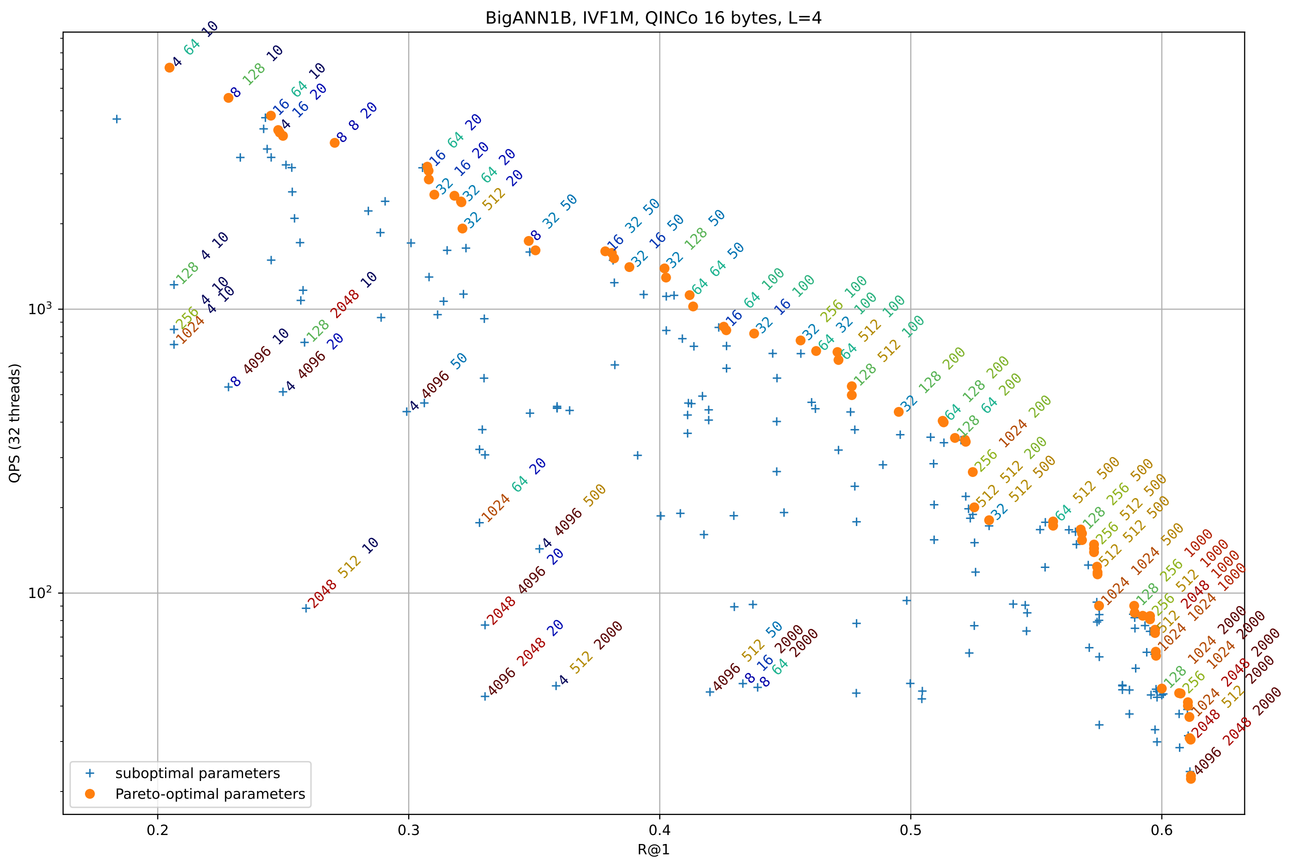


Figure S4. The set of parameters that are tried out for one of the curves of Fig. 3. Each point is obtained by setting three parameters: the P_{IVF}, HNSW's `efSearch` and n_{short}. We indicate the values of these parameters (in this order) for some of the results and color them from lowest (blue) to highest (red) with green in-between.

Table S2. Performance gain by scaling up from $T = 500$k training vectors to $T = 10$M vectors is limited for OPQ, RQ and LSQ, while QINCo improves further when more training data is available. Also UNQ improves from more training data, see App. B.4 for more details on scaleability of UNQ. Training on 500k vectors, QINCo is reported with the number of residual blocks L that resulted in best performance. For both rates, this was $L = 12$ for BigANN1M and Deep1M, $L = 1$ for Contriever1M, and $L = 2$ for Fb-ssnpp1M. When using 10M training vectors we report QINCo with $L = 16$ in general, and $L = 12$ for Contriever1M. For UNQ we report numbers from the original paper (Morozov & Babenko, 2019), where models were trained on 500k vectors, as well as the results of models we trained on 10M vectors using their codebase, denoted UNQ*. For the 8-byte setting, UNQ* achieved highest performance using a hidden dimension of $h' = 1,536$ and $L' = 6$ encoder/decoder layers. For 16 bytes, best performance was found using $h' = 1,536$ and $L' = 4$.

		BigANN1M				**Deep1M**				**Contriever1M**				**FB-ssnpp1M**			
	T	MSE ($\times 10^4$)	R@1	R@10	R@100	MSE	R@1	R@10	R@100	MSE	R@1	R@10	R@100	MSE ($\times 10^4$)	R@1	R@10	R@100
								8 bytes									
OPQ	500k	2.95	21.9	64.8	95.4	0.26	15.9	51.2	88.2	1.87	8.0	24.7	50.8	9.52	2.5	5.1	10.9
OPQ	10M	2.99	21.3	64.3	95.6	0.26	15.1	51.1	87.9	1.87	8.5	24.3	50.4	9.52	2.5	5.0	11.2
RQ	500k	2.49	27.9	75.2	98.2	0.20	21.4	63.5	95.2	1.82	10.2	26.9	52.4	9.20	2.7	6.1	13.6
RQ	10M	2.49	27.9	75.2	98.0	0.20	21.9	64.0	95.2	1.82	9.7	27.1	52.6	9.18	2.7	5.9	14.3
LSQ	500k	1.91	31.9	79.5	98.9	0.17	24.6	69.4	97.0	1.65	13.1	33.9	62.7	8.87	3.3	7.5	17.3
LSQ	10M	1.89	30.6	78.7	98.9	0.17	24.5	68.8	96.7	1.64	13.1	34.9	62.5	8.82	3.5	8.0	18.2
UNQ	500k	1.51	34.6	82.8	99.0	0.16	26.7	72.6	97.3	—	—	—	—	—	—	—	—
UNQ*	10M	1.12	39.7	88.3	99.6	0.14	29.2	77.5	98.8	—	—	—	—	—	—	—	—
QINCo	500k	1.38	40.2	88.0	99.6	0.15	29.4	77.6	98.5	1.57	15.4	38.0	65.5	8.95	3.0	7.7	17.1
QINCo	10M	**1.12**	**45.2**	**91.2**	**99.7**	**0.12**	**36.3**	**84.6**	**99.4**	**1.40**	**20.7**	**47.4**	**74.6**	**8.67**	**3.6**	**8.9**	**20.6**
								16 bytes									
OPQ	500k	1.79	40.5	89.9	99.8	0.14	34.9	82.2	98.9	1.71	18.3	40.9	65.4	7.25	5.0	11.8	25.9
OPQ	10M	1.79	41.3	89.3	99.9	0.14	34.7	81.6	98.8	1.71	18.1	40.9	65.8	7.25	5.2	12.2	27.5
RQ	500k	1.30	49.0	95.0	100.0	0.10	43.0	90.8	99.8	1.65	20.2	43.5	68.2	7.01	5.4	13.0	29.0
RQ	10M	1.30	49.1	94.9	100.0	0.10	42.7	90.5	99.9	1.65	19.7	43.8	68.6	7.00	5.1	12.9	30.2
LSQ	500k	0.98	51.1	95.4	100.0	0.09	42.3	89.7	99.8	1.35	25.6	53.8	78.6	6.63	6.2	14.8	32.3
LSQ	10M	0.97	49.8	95.3	100.0	0.09	41.4	89.3	99.8	1.33	25.8	55.0	80.1	**6.55**	6.3	16.2	35.0
UNQ	500k	0.57	59.3	98.0	100.0	0.07	47.9	93.0	99.8	—	—	—	—	—	—	—	—
UNQ*	10M	0.47	64.3	98.8	100.0	0.06	51.5	95.8	100.0	—	—	—	—	—	—	—	—
QINCo	500k	0.47	65.5	99.1	100.0	0.06	53.0	96.2	100.0	1.30	26.5	54.3	79.5	6.88	5.7	14.4	31.6
QINCo	10M	**0.32**	**71.9**	**99.6**	**100.0**	**0.05**	**59.8**	**98.0**	**100.0**	**1.10**	**31.1**	**62.0**	**85.9**	6.58	**6.4**	**16.8**	**35.5**

B.4. Scaling baselines

Table S2 shows the performance for QINCo and all baselines both trained on 500k vectors and 10M vectors. OPQ, RQ and LSQ do not benefit from more training data in general, while UNQ did improve. A more detailed analysis on UNQ's scalability follows in this section.

In Tab. S2, for 500k training vectors we use the original numbers from the paper (Morozov & Babenko, 2019), while we denote with UNQ* results we obtained by training on 10M vectors by re-running the author's codebase, while model selection was based on the hold-out validation set that we created, see App. A.2. The triplet loss was not used in this scenario as the negative mining on 10M training vectors resulted in prohibitively slow training.

On 500k training vectors, we found that any increase in model size led to overfitting and increasing MSE numbers. However, we did find that UNQ scaled to 10M training vectors quite well for both BigANN1M and Deep1M, with R@1 numbers improving from 34.6% to 39.7% and from 26.7% to 29.2% on Deep1M, respectively for 8 bytes. Similar results are observed for 16 bytes. Despite this, from Fig. S5 we see that QINCo scales even better; MSE rapidly decreases with increasing capacity with far fewer parameters, for both quantities of training data. This shows that QINCo outperforms UNQ both in the low- and high-data regime (with capacity being scaled accordingly).

Note that we experimented with changing the depth L' of the encoder and decoder of UNQ. This parameter was fixed to $L' = 2$ by the authors, and therefore we did not parameterize L' in Tab. 3. Including L' in the number of FLOPS for encoding and decoding of UNQ, results in $h'\big(D+(L'-1)h'+Mb+MK\big)$ and $h'\big(b+(L'-1)h'+D+M\big)$, respectively.

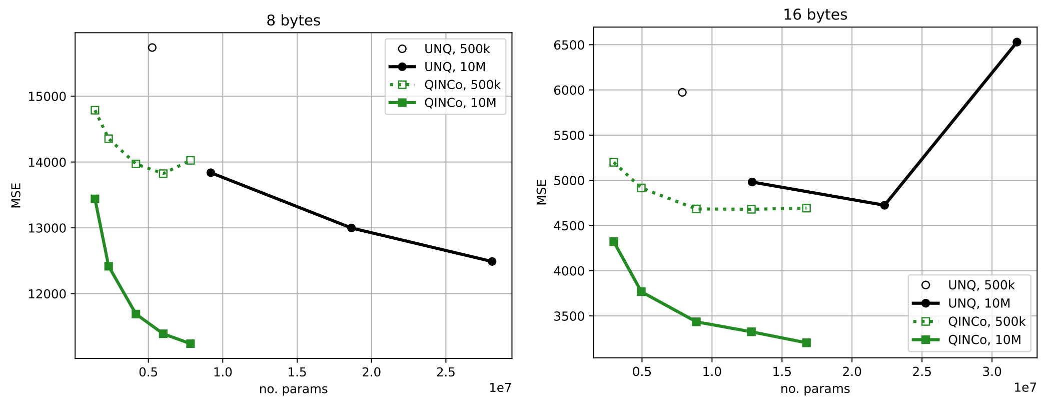


Figure S5. Scaling results comparing UNQ to QINCo. All UNQ models were trained by us using the author's code. For the UNQ training with $T = 500$k vectors, all increases in parameter counts based on expanding the encoder/decoder led to overfitting, and so we observed optimal model performance with hyperparameters from the paper. The single point visualized for "UNQ, 500k" in both graphs is close to the MSE of the models presented by Morozov & Babenko (2019), but with the model selection criteria outlined in A.2. For $T = 10$M vectors, we found the best UNQ model used a hidden dimension of $h' = 1,536$ (instead of the default 1,024), and so in our plots we scale the number of layers in the encoder and decoder using $L' \in \{2, 4, 6\}$. Note that these Pareto curves include the optimal performance point for UNQ reported in Tab. S2. For QINCo we show curves with $h = 256$ and $L \in \{2, 4, 8, 12, 16\}$. With all settings, UNQ has worse operating points for both model and data scaling than QINCo. In some cases, stability was an issue, as can be seen for the highest parameter count setting with UNQ for the 16-byte results.

Table S3. Comparison of UNQ with 16-byte encoding, and QINCo with 12- and 13-byte encoding.

		BigANN1M		**Deep1M**	
	Code length	MSE ($\times 10^4$)	R@1	MSE	R@1
UNQ	16 bytes	0.57	59.3	0.07	47.9
QINCo	12 bytes	0.57	61.8	0.08	49.7
QINCo	13 bytes	0.49	64.1	0.07	53.0

B.5. Dynamic rates

Figure S6 shows the MSE and R@1 performance for QINCo trained for 8-byte and 16-byte encoding. We observe that QINCo trained for 8- and 16-byte encoding performs very similar at the varying rates.

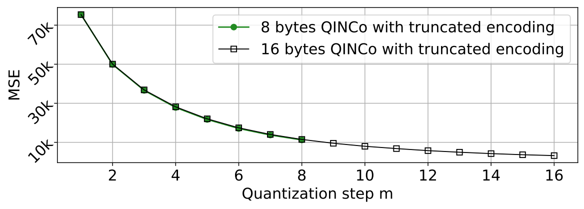
In Tab. S3 we recap the results of UNQ from Tab. 1 of the main paper using 16-byte encoding, and compare them to QINCo results using 12 and 13 byte encoding. The results of QINCo using 12 bytes equal or improve over those of UNQ using 16 bytes, except for MSE on Deep1M where QINCo matches UNQ's 16 bytes results with only 13 bytes.

B.6. Ablations

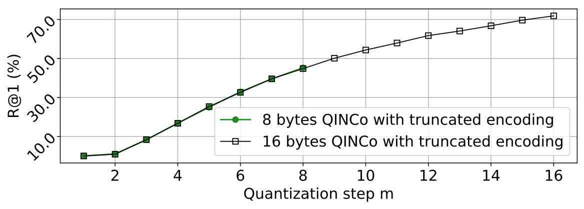
Table S4 shows results of the ablations for which the main conclusions were provided in Sec. 5.4. Below we provide more details for each of those.

One loss vs M losses. QINCo can be trained using only an MSE loss after the last quantization step, *i.e.* $\mathcal{L}^M(\theta)$, instead of using the M losses as given in Equation (3). In Tab. S4, however, we show that this drastically reduces performance. Additionally, we observed that optimization became more unstable, which could not be circumvented by using a lower (base) learning rate.

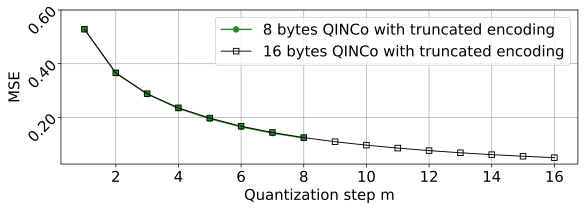
Training the M models separately. The M losses in QINCo can be detached, such that each m^{th} loss only updates the trainable parameters in the m^{th} part of QINCo. Table S4 shows that MSE in all cases deteriorated, while the recall performances remained rather similar, or slightly increased for 8 bytes Deep1B encoding. In general, we might thus conclude that there is no large effect of the m^{th} loss function on earlier quantization steps (i.e. $< m$). This corroborates the earlier-made observation that QINCo can be used with dynamic rates during evaluation.



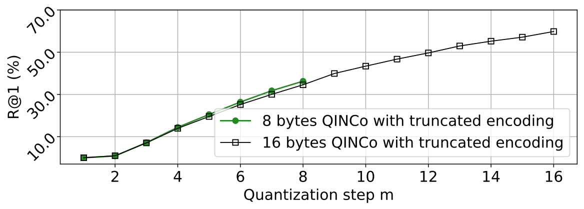
(a) MSE on BigANN1M



(b) R@1 on BigANN1M



(c) MSE on Deep1M



(d) R@1 on Deep1M

Figure S6. MSE and R@1 for BigANN1M and Deep1M for QINCo ($L = 16$) trained for 8-byte and 16-byte encodings, truncated at a varying number of bytes.

Sharing parameters over quantization steps. The number of trainable parameters in QINCo scales linearly with M, the number of bytes used for quantization, see Equation (1). To test whether QINCo actually benefits from having M specialized codebook-updating models, we share (a subset of the) parameters of each of those models over all M steps. We run three variants: (i) only the parameters of the first concatenation block are shared, (ii) only the parameters of the residual-MLPs are shared, and (iii) both the concatenation block and residual-MLP parameters are shared over M. All models were trained on $T = 500$k vectors, and with $L = 8$ residual blocks. Table S4 shows that performance indeed drops when the codebook-predicting models are shared over the M quantization steps. A direct relation is visible between the number of parameters that gets reduced by these actions, and the drop in performance. This finding suggests that the QINCo benefits from learning M specialized codebook-predicting models.

Table S4. Ablation performance for QINCo models trained on $T = 500$k vectors, with $L = 8$. Compared to the base QINCo model (I), performance heavily degrades when using only the MSE loss on the last quantization step (II). Detaching the M losses does slightly deteriorate the MSE reconstruction performance in all cases, but does not seem to affect recall that much (III). Sharing trainable parameters across the M quantization steps reduces performance (IV-VI), mainly when a large part of the parameters are shared (VI).

		BigANN1M					**Deep1M**				
		MSE ($\times 10^4$)	R@1	R@10	R@100	no. params.	MSE	R@1	R@10	R@100	no. params.
						8 bytes					
I	QINCo	**1.40**	**39.7**	87.4	**99.6**	4.2M	**0.15**	29.6	77.6	98.5	3.1M
II	QINCo only last loss $\mathcal{L}^M(\theta)$	2.81	16.2	55.4	90.8	4.2M	0.20	17.4	55.8	91.6	3.1M
III	QINCo M detached losses	1.42	39.1	**87.6**	99.5	4.2M	**0.15**	**30.0**	**78.0**	**98.8**	3.1M
IV	QINCo share concatenate blocks over M	1.46	38.8	87.5	99.5	4.0M	**0.15**	28.7	75.7	98.4	3.0M
V	QINCo share residual-MLPs over M	1.69	37.0	85.4	99.3	1.0M	0.16	27.4	74.5	98.1	0.7M
VI	QINCo share concatenate blocks & residual-MLPs	1.66	37.1	85.2	99.4	0.8M	0.16	28.4	75.4	97.9	0.6M
						16 bytes					
I	QINCo	**0.47**	65.7	**99.0**	**100.0**	8.9M	**0.06**	**53.2**	**96.6**	**100.0**	6.6M
II	QINCo only last loss $\mathcal{L}^M(\theta)$	2.85	16.1	53.2	90.1	8.9M	0.14	27.1	72.3	97.1	6.6M
III	QINCo M detached losses	0.52	65.2	98.7	**100.0**	8.9M	**0.06**	53.1	96.5	**100.0**	6.6M
IV	QINCo share concatenate blocks over M	0.49	**66.2**	99.0	**100.0**	8.4M	0.07	51.4	95.7	**100.0**	6.3M
V	QINCo share residual-MLPs over M	0.69	61.8	98.5	**100.0**	1.5M	0.08	50.0	94.7	**100.0**	1.1M
VI	QINCo share concatenate blocks & residual-MLPs	0.71	59.4	98.3	**100.0**	1.1M	0.08	49.6	95.2	**100.0**	0.8M