Bolt: Accelerated Data Mining with Fast Vector Compression

Davis W. Blalock
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
dblalock@mit.edu

John V. Guttag
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
guttag@mit.edu

ABSTRACT

Vectors of data are at the heart of machine learning and data mining. Recently, vector quantization methods have shown great promise in reducing both the time and space costs of operating on vectors. We introduce a vector quantization algorithm that can compress vectors up to 12× faster than existing techniques while also accelerating approximate vector operations such as distance and dot product computations by over 10×. Because it can encode over two megabytes of vectors per millisecond (2 GB/s), it makes vector quantization cheap enough to employ in many more circumstances. As an example, using our technique to compute approximate dot products in a nested loop can multiply matrices faster than a state-of-the-art BLAS implementation, even when our algorithm must first compress the matrices.

In addition to showing the above speedups, we show experimentally that our approach can be used to accelerate nearest neighbor search and maximum inner product search by up to 140× compared to floating point operations and 10× compared to other vector quantization methods. Our approximate Euclidean distance and dot product computations are not only faster than those of related algorithms with slower encodings, but also faster than Hamming distance computations, which have direct hardware support on the tested platforms. We also assess the errors of our algorithm's approximate distances and dot products, and find that it is competitive with existing, slower vector quantization algorithms.

CCS CONCEPTS

•**Mathematics of computing** →**Probability and statistics;** *Probabilistic algorithms; Dimensionality reduction;* Mathematical software;

KEYWORDS

Vector Quantization, Scalability, Nearest Neighbor Search

ACM Reference format:
Davis W. Blalock and John V. Guttag. 2017. Bolt: Accelerated Data Mining with Fast Vector Compression. In *Proceedings of ACM SIGKDD, Halifax, Nova Scotia Canada, August 2017 (KDD 2017),* 9 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
KDD 2017, Halifax, Nova Scotia Canada
© 2017 Copyright held by the owner/author(s). 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As datasets grow larger, so too do the costs of mining them. These costs include not only the space to store the dataset, but also the compute time to operate on it. This time cost can be decomposed into:

$$\texttt{Cost}_{\text{time}} = \texttt{Cost}_{\text{read}} + \texttt{Cost}_{\text{write}} \tag{1}$$

where $\texttt{Cost}_{\texttt{read}}$ is the time cost of operations that read the data, and $\texttt{Cost}_{\texttt{write}}$ is the time cost of creating, updating, or deleting data.

For datasets of vectors, in which many of the read operations are scalar reductions such as Euclidean distance and dot product computations, vector quantization methods enable significant savings in both space usage and $\texttt{Cost}_{\texttt{read}}$. By replacing each vector with a learned approximation, these methods both save space and enable fast approximate distance and similarity computations. With as little as 8B per vector, these techniques can often preserve distances and dot products with extremely high accuracy [7, 14, 29, 30, 40].

However, computing the approximation for a given vector can be time-consuming, adding greatly to $\texttt{Cost}_{\texttt{write}}$. The state-of-the-art method of [29], for example, requires up to *4ms* to encode a single 128-dimensional vector. This makes it practical only if there are few writes per second. Other techniques are faster, but as we show experimentally, there is significant room for improvement.

We describe a vector quantization algorithm, *Bolt*, that greatly reduces both the time to encode vectors ($\texttt{Cost}_{\texttt{write}}$) and the time to compute scalar reductions over them ($\texttt{Cost}_{\texttt{read}}$). This not only reduces the overhead of quantization, but also increases its benefits, making it worthwhile for many more datasets. Our key ideas are to 1) learn an approximation for the lookup tables used to compute scalar reductions; and 2) use much smaller quantization codebooks than similar techniques. Together, these changes facilitate finding optimal vector encodings and allow scans over codes to be done in a computationally vectorized manner.

Our contributions consist of:

1. A vector quantization algorithm that encodes vectors significantly faster than existing algorithms for a given level of compression.
2. A fast means of computing approximate similarities and distances using quantized vectors. Possible similarities and distances include dot products, cosine similarities, and distances in L_p spaces, such as the Euclidean distance.

1.1 Problem Statement

Formally, the problem our algorithm addresses is the following.

Let $\mathbf{q} \in \mathbb{R}^J$ be a *query* vector and let $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}, \mathbf{x}_i \in \mathbb{R}^J$ be a collection of *database* vectors. Further let $d : \mathbb{R}^J \times \mathbb{R}^J \to \mathbb{R}$ be

a distance or similarity function that can be written as:

$$d(\mathbf{q}, \mathbf{x}) = f\Big(\sum_{j=1}^{J} \delta(q_j, x_j)\Big) \tag{2}$$

where $f : \mathbb{R} \to \mathbb{R}$, $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. This includes both distances in L_p spaces and dot products as special cases. In the former case, $\delta(q_j, x_j) = |q_j - x_j|^p$ and $f(r) = r^{(1/p)}$; in the latter case, $\delta(q_j, x_j) = q_j x_j$ and $f(r) = r$. For brevity, we will henceforth refer to d as a distance function and its output as a distance, though our remarks apply to all functions of the above form unless noted otherwise.

Our task is to construct three functions $g : \mathbb{R}^J \to \mathcal{G}, h : \mathbb{R}^J \to \mathcal{H}$, and $\hat{d} : \mathcal{G} \times \mathcal{H} \to \mathbb{R}$ such that for a given approximation loss $\mathcal{L}$,

$$\mathcal{L} = E_{\mathbf{q},\mathbf{x}}[(d(\mathbf{q}, \mathbf{x}) - \hat{d}(g(\mathbf{q}), h(\mathbf{x})))^2] \tag{3}$$

the computation time T,

$$T = T_g + T_h + T_d \tag{4}$$

is minimized, where T_g is the time to encode received queries $\mathbf{q}$ using g,[1] T_h the time to encode $\mathcal{X}$ using h, and T_d the time to compute the approximate distances between the encoded queries and encoded database vectors. The relative contributions of each of these terms depends on how frequently $\mathcal{X}$ changes, so many of our experiments characterize each separately.

There is a tradeoff between the value of the loss $\mathcal{L}$ and the time T, so multiple operating points are possible. In the extreme cases, $\mathcal{L}$ can be fixed at 0 by setting g and h to identity functions and setting $\hat{d} = d$. Similarly, T can be set to 0 by ignoring the data and estimating d as a constant. The primary contribution of this work is therefore the introduction of g, h and $\hat{d}$ functions that are significantly faster to compute than those of existing work for a wide range of operating points.

1.2 Assumptions

Like other work [7, 14, 21, 29, 40], we assume that there is an initial offline phase during which the functions g and h may be learned. This phase contains a training dataset for $\mathcal{X}$ but not necessarily $\mathbf{q}$. Following this offline phase, there is an online phase wherein we are given database vectors $\mathbf{x}$ that must be encoded and query vectors $\mathbf{q}$ for which we must compute the distances to all of the database vectors received so far. Once a query is received, these distances must be computed with as little latency as possible. The vectors of $\mathbf{X}$ may be given all at once, or one at a time; they may also be modified or deleted, necessitating re-encoding or removal. This is in contrast to most existing work, which assumes that $\mathbf{x}$ vectors are all added at once before any queries are received [7, 14, 21, 29, 40], and therefore that encoding speed is less of a concern.

In practice, one might require the distances between $\mathbf{q}$ and only some of the database vectors $\mathcal{X}$ (in particular, the k closest vectors). This can be achieved using an indexing structure, such as an Inverted Multi-Index [5, 8] or Locality-Sensitive Hashing hash tables [2, 12], that allow inspection of only a fraction of $\mathcal{X}$. Such indexing is complementary to our work in that our approach could be used to accelerate the computation of distances to the subset of $\mathcal{X}$ that is inspected. Consequently, we assume that the task is to compute the distances to all vectors, noting that, in a production setting, "all vectors" for a given query might be a subset of a full database.

[1] We cast the creation of query-specific lookup tables as encoding $\mathbf{q}$ rather than creating a new $\hat{d}$ (the typical interpretation in recent literature).

2 RELATED WORK

Accelerating vector operations through compression has been the subject of a great deal of research in the computer vision, information retrieval, and machine learning communities, among others. Our review will necessarily be incomplete, so we refer the reader to [36, 37] for detailed surveys.

Many existing approaches in the computer vision and information retrieval literature fall into one of two categories [37]: binary embedding and vector quantization. Binary embedding techniques seek to map vectors in $\mathbb{R}^J$ to B-dimensional Hamming space, typically with $B < J$. The appeal of binary embedding is that a B-element vector in Hamming space can be stored in B bits, affording excellent compression. Moreover, the `popcount` instruction present on virtually all desktop, smart phone, and server processors can be used to compute Hamming distances between 8 byte vectors in as little as three cycles. This fast distance computation comes at the price of reduced representational accuracy for a given code length [14, 37] . He et al. [14] demonstrated that the popular binary embedding technique of [15] is a more constrained version of their vector quantization algorithm, and that the objective function of another state-of-the art binary embedding [24], can be understood as maximizing only one of two sufficient conditions for optimal encoding of Gaussian data.

Vector quantization approaches yield lower errors than binary embedding for a given code length, but entail slower encoding and distance computations. The simplest and most popular vector quantization method is K-means, which can be seen as encoding a vector as the centroid to which it is closest. A generalization of K-means, Product Quantization (PQ) [21], splits the vector into M disjoint subvectors and runs K-means on each. The resulting code is the concatenation of the codes for each subspace. Numerous generalizations of PQ have been published, including Cartesian K-means [33], Optimized Product Quantization [14], Generalized Residual Vector Quantization [27], Additive Quantization [6], Composite Quantization [40], Optimized Tree Quantization [7], Stacked Quantizers [30], and Local Search Quantization [29]. The idea behind most of these generalizations is to either rotate the vectors or relax the constraint that the subvectors be disjoint. Collectively, these techniques that rely on using the concatenation of multiple codes to describe a vector are known as Multi-Codebook Quantization (MCQ) methods.

An interesting hybrid between binary embedding and vector quantization is the recent Polysemous Coding of Douze et al. [13]. This encoding uses product quantization codebooks optimized to also function as binary codes, allowing the use of Hamming distances as a fast approximation that can be refined for promising nearest neighbor candidates.

The most similar vector quantization-related algorithm to our own is that of [3], which also vectorizes PQ distance computations. However, their method requires hundreds of thousands or millions of encodings to be sorted lexicographically and stored contiguously ahead of time, as well as scanned through serially. This is untenable when the data is rapidly changing or when using

an indexing structure, which would split the data into far smaller partitions. Their approach also requires a second refinement pass of non-vectorized PQ distance computations, making their reported speedups significantly lower than our own.

In the machine learning community, accelerating vector operations has been done primarily through (non-binary) embedding, structured matrices, and model compression. Embedding yields acceleration by reducing the dimensionality of data while preserving the relevant structure of a dataset overall. There are strong theoretical gaurantees regarding the level of reduction attainable for a given level of distortion in pairwise distances [1, 11, 25], as well as strong empirical results [22, 35]. However, because embedding *per se* only entails reducing the number of floating-point numbers stored, without reducing the size of each, it is not usually competitive with vector quantization methods. It is possible to embed data before applying vector quantization, so the two techniques are complementary.

An alternative to embedding that reduces the cost of storing and multiplying by matrices is the use of structured matrices. This consists of repeatedly applying a linear transform, such as permutation [38], the Fast Fourier Transform [39], the Discrete Cosine Transform [31], or the Fast Hadamard Transform [2, 9], possibly with learned elementwise weights, instead of performing a matrix multiply. These methods have strong theoretical grounding [9] and sometimes outperform non-structured matrices [38]. They are orthogonal to our work in that they bypass the need for a matrix entirely, while our approach can accelerate operations in which a matrix is needed.

Another vector-quantization-like technique common in machine learning is model compression. This typically consists of some combination of 1) restricting the representation of variables, such as neural network weights, to fewer bits [20]; 2) reusing weights [10]; 3) pruning weights in a model after training [19, 28]; and 4) training a small model to approximate the outputs of a larger model [34]. This has been a subject of intense research for neural networks in recent years, so we do not believe that our approach could yield smaller neural networks than the current state of the art. Instead, our focus is on accelerating operations on weights and data that would otherwise not have been compressed.

3 METHOD

As mentioned in the problem statement, our goal is to construct a distance function $\hat{d}$ and two encoding functions g and h such that $\hat{d}(g(\mathbf{q}), h(\mathbf{x})) \approx d(\mathbf{q}, \mathbf{x})$ for some "true" distance function d. To explain how we do this, we first begin with a review of Product Quantization [21], and then describe how our method differs.

3.1 Background: Product Quantization

Perhaps the simplest form of vector quantization is the k-means algorithm, which quantizes a vector to its closest centroid among a fixed *codebook* of possibilities. As an encoding function, it transforms a vector into a $\lceil log_2(K) \rceil$-bit *code* indicating which centroid is closest, where K is the codebook size (i.e., number of centroids). Using this encoding, the distance between a query and a database vector can be approximated as the distance between the query and its associated centroid.

Product Quantization (PQ) is a generalization of k-means wherein the vector is split into disjoint *subvectors* and the full vector is encoded as the concatenation of the codes for the subvectors. Then, the full distance is approximated as the sum of the distances between the subvectors of $\mathbf{q}$ and the chosen centroids for each corresponding subvector of $\mathbf{x}$.

Formally, PQ approximates the function d as follows. First, recall that, by assumption, d can be written as:

$$d(\mathbf{q}, \mathbf{x}) = f\Big(\sum_{j=1}^{J} \delta(q_j, x_j) \Big)$$

where $f : \mathbb{R} \to \mathbb{R}$, $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Now, suppose one has a partition $\mathcal{P} = \{p_1, \ldots, p_M\}$ of the indices j, so that the subsets p_j are both mutually exclusive and collectively exhaustive. The argument to f can then be written as:

$$\sum_{m=1}^{M} \sum_{j \in p_m} \delta(q_j, x_j) = \sum_{m=1}^{M} \boldsymbol{\delta}\left(\mathbf{q}^{(m)}, \mathbf{x}^{(m)}\right) \tag{5}$$

where $\mathbf{q}^{(m)}$ and $\mathbf{x}^{(m)}$ are the *subvectors* formed by gathering the elements of $\mathbf{q}$ and $\mathbf{x}$ at the indices $j \in p_m$, and $\boldsymbol{\delta}$ sums the δ functions applied to each dimension. Product quantization replaces each $\mathbf{x}^{(m)}$ with one vector $\mathbf{c}_i^{(m)}$ from a *codebook* set C_m of possibilities. That is:

$$\sum_{m=1}^{M} \boldsymbol{\delta}\left(\mathbf{q}^{(m)}, \mathbf{x}^{(m)}\right) \approx \sum_{m=1}^{M} \boldsymbol{\delta}\left(\mathbf{q}^{(m)}, \mathbf{c}_i^{(m)}\right) \tag{6}$$

This allows one to store only the identity of the codebook vector chosen (i.e., i), instead of the elements of the original vector $\mathbf{x}^{(m)}$. More formally, let $\mathbf{C} = \{C_1, \ldots, C_M\}$ be a set of M codebooks where each codebook C_m is itself a set of K vectors $\{\mathbf{c}_1^{(m)}, \ldots, \mathbf{c}_K^{(m)}\}$; we will refer to these vectors as *centroids*. Given this set of codebooks, the PQ encoding function $h(\mathbf{x})$ is:

$$h(\mathbf{x}) = [i_1; \ldots; i_M], \; i_m = \arg\min_i d\left(\mathbf{c}_i^{(m)}, \mathbf{x}^{(m)}\right) \tag{7}$$

That is, $h(\mathbf{x})$ is a vector such that $h(\mathbf{x})_m$ is the index of the centroid within codebook m to which $\mathbf{x}^{(m)}$ is closest.

Using these codebooks also enables construction of a fast query encoding g and distance approximation $\hat{d}$. Specifically, let the query encoding space $\mathcal{G}$ be $R^{K \times M}$ and define $\mathbf{D} = g(\mathbf{q})$ as:

$$\mathbf{D}_{im} \triangleq \boldsymbol{\delta}\left(\mathbf{q}^{(m)}, \mathbf{c}_i^{(m)}\right) \tag{8}$$

Then we can rewrite the approximate distance on the right hand side of 6 as:

$$\sum_{m=1}^{M} \mathbf{D}_{im}, \quad i = h(\mathbf{x})_m \tag{9}$$

In other words, the distance can be reduced to a sum of precomputed distances between $\mathbf{q}^{(m)}$ and the codebook vectors $\mathbf{c}_i^{(m)}$ used to approximate $\mathbf{x}$. Each of the M columns of $\mathbf{D}$ represents the distances between $\mathbf{q}^{(m)}$ and the K centroids in codebook C_M. Computation of the distance proceeds by iterating through the columns, looking

up the distance in row $h(\mathbf{x})_m$, and adding it to a running total. By reintroducing f, one can now define:

$$\hat{d}(g(\mathbf{q}), h(\mathbf{x})) \triangleq f\Big(\sum_{m=1}^{M} \mathbf{D}_{im}, i = h(\mathbf{x})_m\Big) \tag{10}$$

If $M \ll D$ and $K \ll |\mathcal{X}|$, then computation of $\hat{d}$ is much faster than computation of d given the $g(\mathbf{q})$ matrix $\mathbf{D}$ and data encodings $\mathcal{H} = \{h(\mathbf{x}), \mathbf{x} \in \mathcal{X}\}$.

The total computational cost of product quantization is $\Theta(KJ)$ to encode each $\mathbf{x}$, $\Theta(KJ)$ to encode each query $\mathbf{q}$, and $\Theta(M)$ to compute the approximate distance between an encoded $\mathbf{q}$ and encoded $\mathbf{x}$. Because queries must be encoded before distance computations can be performed, this means that the cost of computing the distances to the N database vectors $\mathcal{X}$ when a query is received is $\Theta(KJ)+\Theta(NM)$. Lastly, since codebooks are learned using k-means clustering, the time to learn the codebook vectors is $O(KNJT)$, where T is the number of k-means iterations. In all works of which we are aware, K is set to 256 so that each element of $h(\mathbf{x})$ can be encoded as one byte. Further, the subspaces p_m are the contiguous blocks of J/M dimensions, possibly after a random permutation.

In certain cases, product quantization is nearly an optimal encoding scheme. Specifically, under the assumptions that:

1. $\mathbf{x} \sim MVN(\mu, \Sigma)$, and therefore $\mathbf{x}^{(m)} \sim MVN(\mu_m, \Sigma_m)$,
2. $\forall_m |\Sigma_m| = |\Sigma|^{1/m}$,

PQ achieves the information-theoretic lower bound on code length for a given quantization error [14]. In words, this means that PQ encoding is optimal if $\mathbf{x}$ is drawn from a multivariate Gaussian and the subspaces p_m are independent and have covariance matrices with equal determinants.

In practice, however, most datasets are not Gaussian and their subspaces are neither independent nor described by similar covariances. Consequently, many works have generalized PQ to capture relationships across subspaces or decrease the dependencies between them [6, 7, 14, 29, 33].

In summary, PQ consists of three components:

1. Encoding every $\mathbf{x}$ in the database using $h(\mathbf{x})$. This transforms $\mathbf{x}$ to a list of M 8-bit integers.
2. Encoding a query $\mathbf{q}$ when it is received using $g(\mathbf{q})$. This transforms returns a $K \times M$ matrix $\mathbf{D}$ whose columns are the distances to each centroid in codebook C_m.
3. Scanning the database. Once a query is computed, the approximated distance to each $\mathbf{x}$ is computed using (10) by looking up and summing the appropriate entries in each column of $\mathbf{D}$.

3.2 Bolt

Bolt is similar to product quantization but differs in two key ways:

1. It uses much smaller codebooks.
2. It approximates the distance matrix $\mathbf{D}$.

Change (1) directly increases the speeds of the encoding functions g and h. This is because it reduces the number of k-means centroids for which the distances to a given subvector $\mathbf{x}^{(m)}$ or $\mathbf{q}^{(m)}$ must be computed. More specifically, by using $K = 16$ centroids (motivated below) instead of 256, we reduce the computation by a factor of $256/16 = 16$. This is the source of Bolt's fast encoding. Using fewer centroids also reduces the k-means training time, although this is not our focus.

Change (2), approximating the query distance matrix $\mathbf{D}$, allows us to reduce the size of $\mathbf{D}$. This approximation is separate from approximating the overall distance—in other algorithms, the entries of $\mathbf{D}$ are the exact distances between each $\mathbf{q}^{(m)}$ and the corresponding centroids C_m. In Bolt, the entries of $\mathbf{D}$ are learned 8-bit quantizations of these exact distances.

Together, changes (1) and (2) allow hardware vectorization of the lookups in $\mathbf{D}$. Concretely, instead of looking up the entry in a given column of D for one $\mathbf{x}$ (a standard load from memory), we can leverage vector instructions to instead perform V lookups for V consecutive $h(\mathbf{x})$, $h(\mathbf{x}_i), \ldots, h(\mathbf{x}_{i+V})$, where $V = 16$, 32, or 64 depending on the platform. Under the mild assumption that encodings can be stored in blocks of at least V elements, this affords roughly a V-fold speedup in the computation of distances. The ability to perform such vectorized lookups is present on nearly all modern desktops, laptops, servers, tablets, and CUDA-enabled GPUs.[2] Consequently, while the performance gain comes from fairly low-level hardware functionality, Bolt is not tied to any particular architecture, processor, or platform.

Mathematically, the challenge in the above approach is quantizing $\mathbf{D}$. The distances in this matrix vary tremendously as a function of dataset, query vector, and even codebook. Naively truncating the floating-point values to integers in the range [0, 255], for example, would yield almost entirely 0s for datasets with entries $\ll 1$ and almost entirely 255s for datasets with entries $\gg 255$. This can of course be counteracted to some extent by globally shifting and scaling the dataset, but such global changes do not account for query-specific and codebook-specific variation.

Consequently, we propose to learn a quantization function at training time. The basic approach is to learn the distribution of distances within a given column of $\mathbf{D}$ (the distances to centroids within one codebook) across many queries sampled from the training set and find upper and lower cutoffs such that the expected squared error between the quantized and original distances is minimized.

Formally, for a given column m of $\mathbf{D}$ (henceforth, one *lookup table*), let Q be the distribution of query subvectors $\mathbf{q}^{(m)}$, X be the distribution of database subvectors $\mathbf{x}^{(m)}$, and Y be the scalar-valued distribution of distances within that table. I.e.:

$$p(Y = y) \triangleq \int_{Q,X} p(\mathbf{q}^{(m)}, \mathbf{x}^{(m)}) I\{\boldsymbol{\delta}\left(\mathbf{q}^{(m)}, \mathbf{x}^{(m)}\right) = y\} \tag{11}$$

We seek to learn a table-specific quantization function $\beta_m : \mathbb{R} \to \{0, \ldots, 255\}$ that minimizes the quantization error. For computational efficiency, we constrain $\beta_m(y)$ to be of the form:

$$\beta_m(y) = \max(0, \min(255, \lfloor ay - b \rfloor)) \tag{12}$$

for some constants a and b. Formally, we seek values for a and b that minimize:

$$E_Y[(\hat{y} - y)^2] \tag{13}$$

where $\hat{y} \triangleq (\beta_m(y) + b)/a$ is termed the *reconstruction* of y. Y can be an arbitrary distribution (though we assume it has finite mean

[2] The relevant instructions are `vpshufb` on x86, `vtbl` on ARM, `vperm` on PowerPC, and `__shfl` on CUDA.

and variance) and the value of $\beta_m(y)$ is constrained to a finite set of integers, so there is not an obvious solution to this problem.

We propose to set $b = F^{-1}(\alpha)$, $a = 255/(F^{-1}(1-\alpha) - b)$ for some suitable α, where F^{-1} is the inverse CDF of Y, estimated empirically. That is, we set a and b such that the α and $1-\alpha$ quantiles of Y are mapped to 0 and 255. Because both $F^{-1}(\alpha)$ and the loss function are cheap to compute, we can find a good α at training time via a simple grid search. In our experiments, we search over the values $\{0, .001, .002, .005, .01, .02, .05, .1\}$. In practice, the chosen α tends to be among the smaller values, consistent with the observation that loss from extreme values of y is more costly than reduced granularity in representing typical values of y.

To quantize multiple lookup tables, we learn a b value for each table and set a based on the CDF of the aggregated distances Y across all tables. We cannot learn table-specific a values because this would amount to weighting distances from each differently. The b values can be table-specific because they sum to one overall offset in the distance, which is known at the end of training time and can be corrected for.

In summary, Bolt is an extension of product quantization with 1) fast encoding speed stemming from small codebooks; and 2) fast distance computations stemming from adaptively quantized lookup tables and efficient use of hardware.

3.3 Theoretical Guarantees

Due to space constraints, we state the following without proof. Supporting materials, including additional bounds, can be found on Bolt's website. Throughout the following, let $b_{min} \triangleq F^{-1}(\alpha)$, $b_{max} \triangleq F^{-1}(1-\alpha)$, $\Delta \triangleq \frac{b_{max} - b_{min}}{256}$, and $\sigma_Y \triangleq \sqrt{\text{Var}[Y]}$. Furthermore, let the tails of Y be drawn from any Laplace, Exponential, Gaussian, or subgaussian distribution, where the tails are defined to include the intervals $(-\infty, b_{min}]$ and $[b_{max}, \infty)$.

Lemma 3.1. $b_{min} \leq y \leq b_{max} \implies |y - \hat{y}| < \Delta$.

Lemma 3.2. *For all* $\varepsilon > \Delta$, $p(|y - \hat{y}| > \varepsilon) <$

$$\frac{1}{\sigma_Y}\left(e^{-(b_{max} - E[Y])/\sigma_Y} + e^{-(E[Y] - b_{min})/\sigma_Y}\right) e^{-\varepsilon/\sigma_Y} \tag{14}$$

We now bound the overall errors in dot products and Euclidean distances. First, regardless of the distributions of $\mathbf{q}$ and $\mathbf{x}$, the following hold:

Lemma 3.3. $|\mathbf{q}^\top \mathbf{x} - \mathbf{q}^\top \hat{\mathbf{x}}| < \|\mathbf{q}\| \cdot \|\mathbf{x} - \hat{\mathbf{x}}\|$

Lemma 3.4. $|\|\mathbf{q} - \mathbf{x}\| - \|\mathbf{q} - \hat{\mathbf{x}}\|| < \|\mathbf{x} - \hat{\mathbf{x}}\|$

Using these lemmas, it is possible to obtain tighter, probabilistic bounds using Hoeffding's inequality.

Definition 3.5 (Reconstruction). Let $\mathbf{C}$ be the set of codebooks used to encode $\mathbf{x}$. Then the vector obtained by replacing each $\mathbf{x}^{(m)}$ with its nearest centroid in codebook C_m is the reconstruction of $\mathbf{x}$, denoted $\hat{\mathbf{x}}$.

Lemma 3.6. *Let* $\mathbf{r}^{(m)} \triangleq \mathbf{x}^{(m)} - \hat{\mathbf{x}}^{(m)}$, *and assume that the values of* $\|\mathbf{r}^{(m)}\|$ *are independent for all* m. *Then:*

$$p(|\mathbf{q}^\top \mathbf{x} - \mathbf{q}^\top \hat{\mathbf{x}}| \geq \varepsilon) \leq 2\exp\left(\frac{-\varepsilon^2}{2\sum_{m=1}^{M}(\|\mathbf{q}^{(m)}\| \cdot \|\mathbf{r}^{(m)}\|))^2}\right) \tag{15}$$

Lemma 3.7. *Let* $\mathbf{r}^{(m)} \triangleq \mathbf{x}^{(m)} - \hat{\mathbf{x}}^{(m)}$, *and assume that the values of* $\|\mathbf{q}^{(m)} - \mathbf{x}^{(m)}\|^2 - \|\mathbf{q}^{(m)} - \hat{\mathbf{x}}^{(m)}\|^2$ *are independent for all* m. *Then:*

$$p(|\|\mathbf{q} - \mathbf{x}\|^2 - \|\mathbf{q} - \hat{\mathbf{x}}\|^2| > \varepsilon) \leq 2\exp\left(\frac{-\varepsilon^2}{2\sum_{m=1}^{M}\|\mathbf{r}^{(m)}\|^4}\right) \tag{16}$$

4 EXPERIMENTAL RESULTS

To assess Bolt's effectiveness, we implemented both it and comparison algorithms in C++ and Python. All of our code and raw results are publicly available on the Bolt website[3]. This website also contains additional experiments and thorough documentation of both our code and experimental setups. All experiments use a single thread on a 2013 Macbook Pro with a 2.6GHz Intel Core i7-4960HQ processor.

The goals of our experiments are to show that 1) Bolt is extremely fast at both encoding vectors and computing scalar reductions, both compared to similar algorithms and in absolute terms; and 2) Bolt achieves this speed at little cost in accuracy compared to similar algorithms. To do the former, we record its throughput in encoding and computing reductions. To do the latter, we measure its accuracy in retrieving nearest neighbors, as well as the correlations between the reduced values it returns and the true values. Because they are by far the most benchmarked scalar reductions in related work and are widely used in practice, we test Bolt only on the Euclidean distance and dot product. Due to space constraints, we do not compare Bolt's distance table quantization method to possible alternatives, instead simply demonstrating that it yields no discernible loss of accuracy compared to non-quantized distance tables. For all experiments, we assess Bolt and the comparison MCQ methods using the commonly-employed encoding sizes of 8B, 16B, and 32B to characterize the relationships between space, speed and accuracy.

All reported timings and throughputs are the best of 5 runs, averaged over 10 trials (i.e., the code is executed 50 times). We use the best in each trial, rather than average, since this is standard practice in performance benchmarking. Because there are no conditional branches in either Bolt or the comparison algorithms (when implemented efficiently), all running times depend only on the sizes of the database and queries, not their distributions; consequently, we report timing results on random data.

4.1 Datasets

For assessing accuracy, we use several datasets widely used to benchmark Multi-Codebook Quantization (MCQ) algorithms:

- **Sift1M** [21] — 1 million 128-dimensional SIFT descriptors of images. Sift1M vectors tend to have high correlations among many dimensions, and so be highly compressible by algorithms that allow global rotations or do not quantize subspaces independently. This dataset has a predefined query/train database/test database split, consisting of 10,000 query vectors, 100,000 training vectors, and 1 million database vectors.
- **Convnet1M** [30] — 1 million 128-dimensional Convnet descriptors of images. These vectors have some amount of correlation, but less so than Sift1M. It has a query/train/test split matching that of Sift1M.

[3] https://github.com/dblalock/bolt

- **LabelMe22k** [32] — 22,000 512-dimensional GIST descriptors of images. Like Sift, it has a great deal of correlation between many dimensions. It only has a train/test split, so we follow [29, 40] and use the 2,000-vector test set as the queries and the 20,000 vector training set as both the training and test database.
- **MNIST** [26] — 60,000 28x28-pixel greyscale images, flattened to 784-dimensional vectors. This dataset is sparse and has high correlations between various dimensions. Again following [29] and [40], we split it the same way as the LabelMe dataset.

4.2 Comparison Algorithms

Our comparison algorithms include Multi-Codebook Quantization (MCQ) methods that have high encoding speeds ($\ll$ 1ms / vector on a CPU). If encoding speed is not a design consideration or is dominated by a need for maximal compression, methods such as GRVQ [27] or LSQ [29] are more appropriate than Bolt[4].

Specifically, our primary baselines are Product Quantization (PQ) [21] and Optimized Product Quantization (OPQ) [14], since they offer the fastest encoding times. There are several algorithms that extend these basic approaches by adding indexing methods [8, 23], or more sophisticated training-time optimizations [4, 13, 18], but since these extensions are compatible with our own work, we do not compare to them. We compare only to versions of PQ and OPQ that use 8 bits per codebook, as this is the setting used in all related work of which we are aware; we do not compare to using 4 bits, as in Bolt, since this both reduces their accuracy and increases their computation time.

We do not to compare to binary embedding methods in terms of accuracy as they are known to yield much lower accuracy for a given code length than MCQ methods [14, 37] and, as we show, are also slower in computing distances than Bolt.

We have done our best to optimize the implementations of the comparison algorithms, and find that we obtain running times superior to those described in previous works. For example, [30] reports encoding roughly 200,000 128-dimensional vectors per second with PQ, while our implementation encodes more than double this.

As a final comparison, we include a modified version of Bolt, *Bolt No Quantize*, in our accuracy experiments. This version does not quantize the distance lookup tables. It is not a useful algorithm as it sacrifices Bolt's high speed, but it allows us to assess whether our quantization scheme reduces accuracy.

4.3 Encoding Speed

Before a vector quantization method can compute approximate distances, it must first encode the data. We measured how many vectors each algorithm could encode per second as a function of the vectors' length. As shown in Figure 1.*left*, Bolt can encode data vectors up to 10× faster than PQ, the fastest comparison. Encoding 10^7 128-dimensional vectors of $4B$ floats per second (top left plot) translates to an encoding speed of 5.1GB/s. For perspective, Bolt's encoding rate is sufficient to encode the entire Sift1M dataset of 1 million vectors in 100ms, and the Sift1B dataset of 1 billion vectors in 100s. This rate is also much higher than that of high-speed (but general-purpose) compression algorithms such as Snappy [17], which reports an encoding speed of 250MB/s.

[4] Although Bolt *might* still be desirable for its high query speed even if encoding speed is not a consideration.

Similarly, Bolt can compute the distance matrix constituting a query's encoding at up to 10 million queries/s (top right plot), while PQ obtains less than 1 million queries/s. Both of these numbers are sufficiently high that encoding the query is unlikely to ever be a bottleneck in computing distances to it.

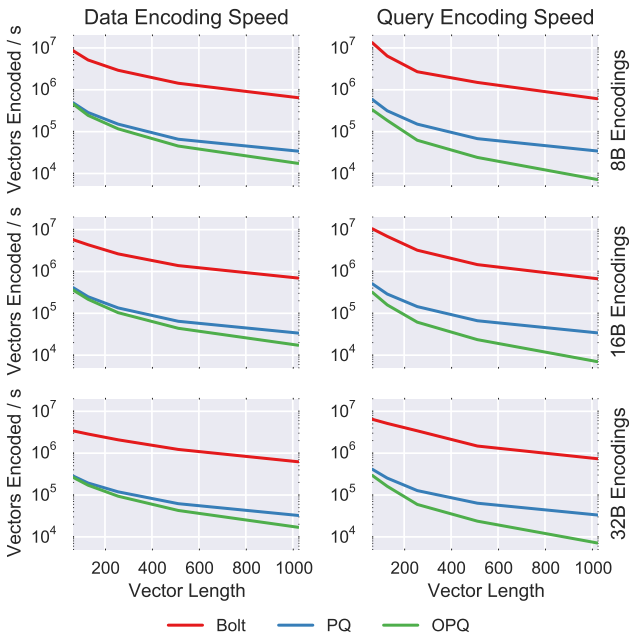

Figure 1: Bolt encodes both data and query vectors significantly faster than similar algorithms.

4.4 Query Speed

Much of the appeal of MCQ methods is that they allow fast computation of approximate distances and similarities directly on compressed data. We assessed various algorithms' speed in computing Euclidean distances from a batch of queries to each vector in a compressed dataset. We omit profiling of other distances and similarities since they only alter the computation of queries' distance matrices and therefore have nearly identical speeds. In all experiments, the number of compressed data vectors N is fixed at 100,000 and their dimensionality is fixed at 256.

In contrast to other experiments, we compare Bolt not only to other MCQ methods, but also other methods of computing distances that might serve as reasonable alternatives to using MCQ at all. These methods include:

- *Binary Embedding.* As mentioned in Section 2, the current fastest method of obtaining approximate distances over compressed vectors is to embed them into Hamming space and use the `popcount` instruction to quickly compute the Hamming distances between them.
- *Matrix Multiplies.* Given the norms of query and database vectors, Euclidean distances can be computed using matrix-vector multiplies. When queries arrive extremely quickly relative to the latency with which they must be answered, multiple queries can be batched into a matrix. Performing one matrix multiply is many times faster than performing individual matrix-vector multiplies. We compare to batch sizes of 1, 256, and 1024.

Bolt computes Euclidean distances more than ten times faster than any other MCQ algorithm and significantly faster than binary embedding methods can compute Hamming distances (Figure 2). Its speedup over matrix multiplies depends on the batch size and number of bytes used in MCQ encoding. When it is not possible to batch multiple queries (*Matmul 1*), Bolt 8B is over 250× faster, Bolt 16B is over 140× faster, and Bolt 32B is over 60× faster (see website for exact timings). When hundreds of queries can be batched (*Matmul 256*, *Matmul 1024*), these numbers are reduced to roughly 13×, 7×, and 3×.

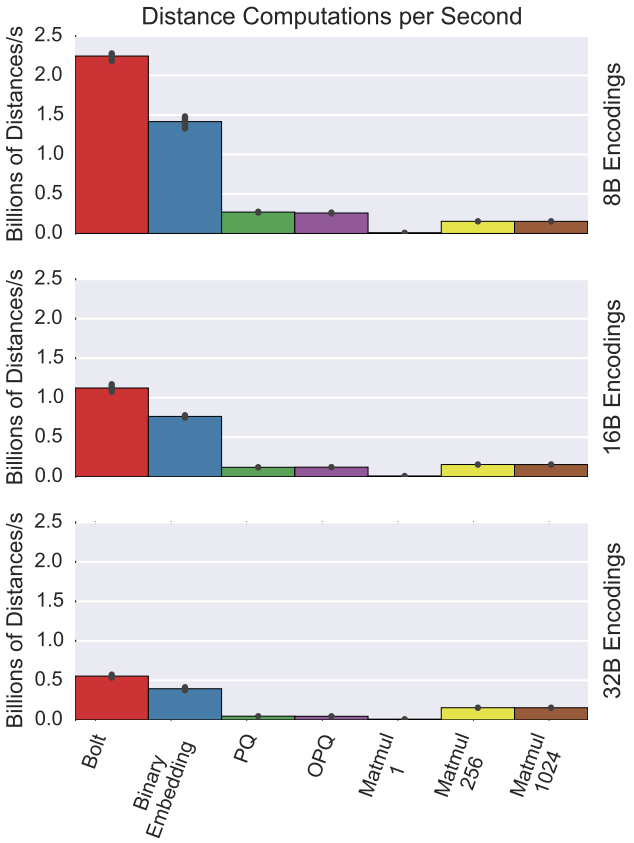

Figure 2: Bolt can compute the distances/similarities between a query and the vectors of a compressed database over 10× faster than other MCQ algorithms. It is also faster than binary embedding methods, which use the hardware `popcount` instruction, and matrix-vector multiplies using batches of 1, 256, or 1024 vectors.

Because matrix multiplies are so ubiquitous in data mining, machine learning, and many other fields, we compare Bolt to matrix multiplication in more detail. In Figure 3, we profile the time that Bolt and a state-of-the-art BLAS implementation [16] take to do matrix multiplies of various sizes. Bolt computes matrix multiplies by treating each row of the first matrix as a query, the second matrix as a database, and iteratively computing the inner products between each query and all database vectors. This nested-loop implementation is not optimal, but Bolt is still able to outperform BLAS.

In Figure 3.*top*, we multiply two square matrices of varying sizes, which is the optimal scenario for most matrix multiply algorithms. For small matrices, the cost of encoding one matrix as the database is too high for Bolt to be faster. For larger matrices, this cost is amortized over many more queries, and Bolt becomes faster. When the database matrix is already encoded, Bolt is faster for almost all matrix sizes, even using 32B encodings. Note, though, that this comparison ceases to be fair for especially large matrices, as encoding, e.g., 4096 dimensions accurately would almost certainly require more than 32B.

In Figure 3.*bottom*, we multiply a $100,000 \times 256$ matrix by a $256 \times n$ matrix. Bolt uses the rows of the former matrix as the database and the columns of the latter as the queries. Again, Bolt is slower for small matrices when it must first encode the database, but always faster for larger ones or when it does not need to encode the database. Because only the number of queries is changing and not the dimensionality of each vector, longer encodings would not be necessary for the larger matrices.

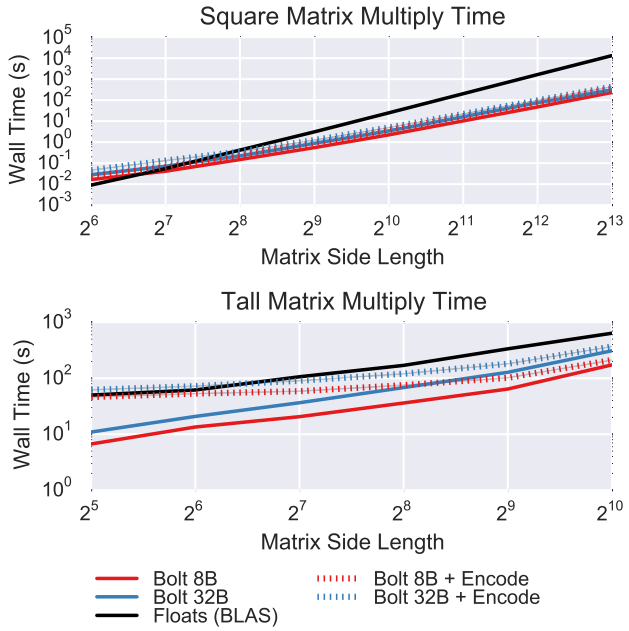

Figure 3: Using a naive nested loop implementation, Bolt can compute (approximate) matrix products faster than optimized matrix multiply routines. Except for small matrices, Bolt is faster even when it must encode the matrices from scratch as a first step.

4.5 Nearest Neighbor Accuracy

By far the most common assessment of MCQ algorithms' accuracy is their Recall@R. This is defined as the fraction of the queries $\mathbf{q}$ for which the true nearest neighbor in Euclidean space is among the top R points with smallest approximate distances to $\mathbf{q}$. This is a proxy for how many points would likely have to be reranked in a retrieval context when using an approximate distance measure to generate a set of candidates. As shown in Figure 4, Bolt yields slightly lower accuracy for a given encoding length than other (much slower) MCQ methods. The nearly identical curves for

Bolt and Bolt No Quantize suggest that our proposed quantization algorithm introduces little or no error.

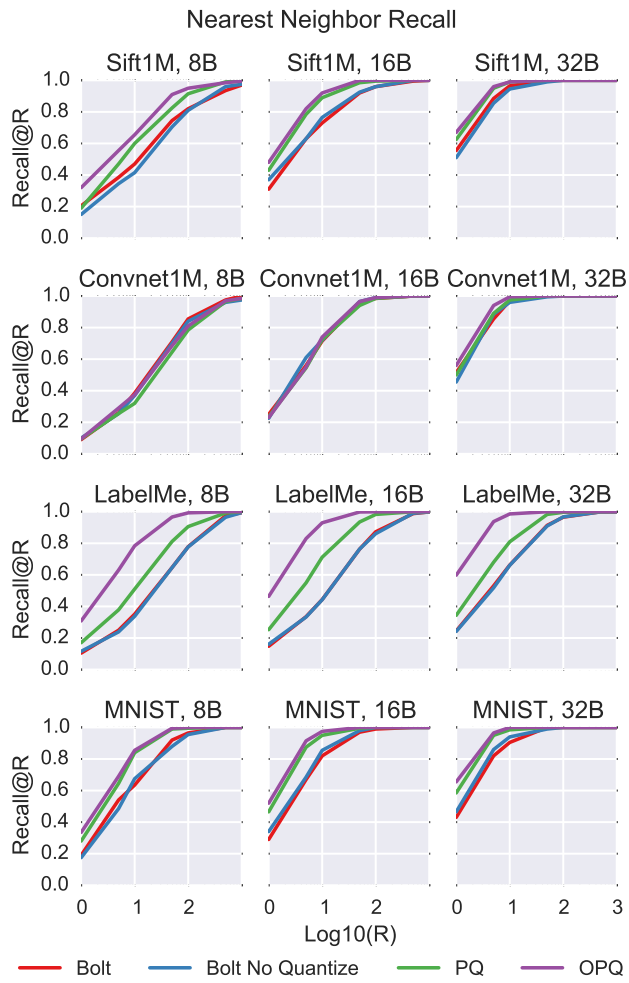

Figure 4: Compared to other MCQ algorithms, Bolt is slightly less accurate in retrieving the nearest neighbor for a given encoding length.

The differences across datasets can be explained by their varying dimensionalities and the extent to which correlated dimensions tend to be in the same subspaces. On the Sift1M dataset, adjacent dimensions are highly correlated, but they are also correlated with other dimensions slightly farther away. This first characteristic allows all algorithms to perform well, but the second allows PQ and OPQ to perform even better thanks to their smaller numbers of larger codebooks. Having fewer codebooks means that the subspaces associated with each are larger (i.e., more dimensions are quantized together), allowing mutual information between them to be exploited. Having larger codebooks also allows a more precise fit. Bolt, with its larger number of smaller codebooks, must quantize more sets of dimensions independently, which does not allow it to exploit this mutual inforation. Much the same phenomena explain the results on MNIST.

For the LabelMe dataset, the correlations tend to be even more diffuse, with small correlations spanning dimensions belonging to many subspaces. This is less problematic for OPQ, which learns a rotation such that correlated dimensions tend to be placed in the same subspaces. PQ and Bolt, which lack the ability to rotate the data, have no such option, and so are unable to encode the data as effectively.

Finally, for the Convnet1M dataset, most of the correlated dimensions tend to be immediately adjacent to one another, allowing all methods to perform roughly equally.

4.6 Accuracy in Preserving Distances and Dot Products

The Recall@R experiment characterizes how well each algorithm preserves distances to highly similar points, but not whether distances in general tend to be preserved. To assess this, we computed the correlations between the true dot products and approximate dot products for Bolt and the comparison algorithms. Results for Euclidean distances are similar, so we omit them. As Figure 5 illustrates, Bolt is again slightly less accurate than other MCQ methods. In absolute terms, however, it consistently exhibits correlations with the true dot products above .9, and often near 1.0. This suggests that its approximations could reliably be used instead of exact computations when slight errors are permissible.

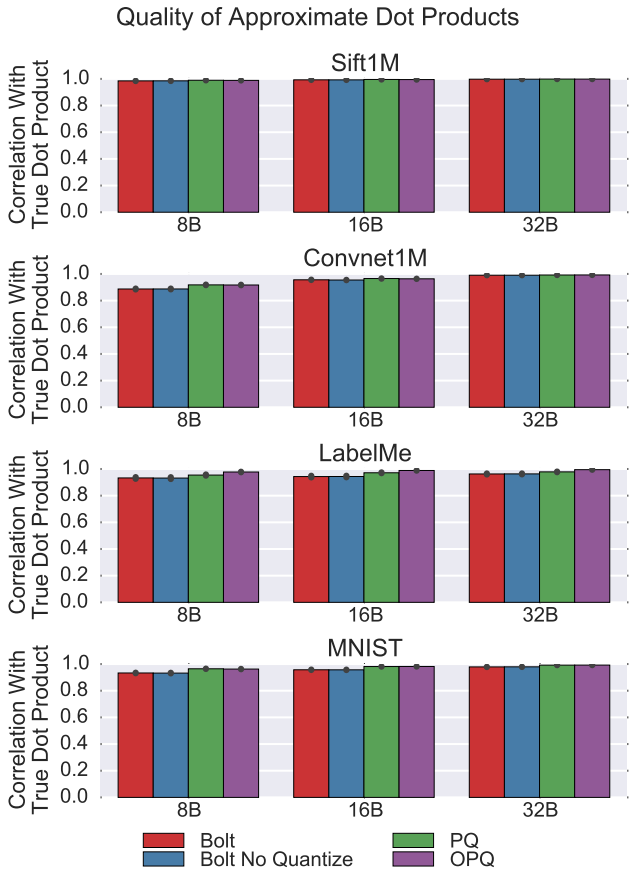

Figure 5: Bolt dot products are highly correlated with true dot products, though slightly less so than those from other MCQ algorithms.

For example, if one could tolerate a correlation of .9, one could use Bolt 8B instead of dense vectors of 4B floats and achieve dramatic speedups as well as compression ratios of 64× for SIFT1M and Convnet1M, 256× for LabelMe, and 392× for MNIST. If one required correlations of .95 or more, one could use Bolt 32B and achieve slightly smaller speedups with compression ratios of 16×, 64×, and 98×.

5 CONCLUSION

We described Bolt, a vector quantization algorithm that rapidly compresses large collections of vectors and enables fast computation of approximate Euclidean distances and dot products directly on the compressed representations. Bolt both compresses data and computes distances and dot products more than 10× faster than existing algorithms, making it advantageous both in read-heavy and write-heavy scenarios. Its approximate computations can be over 140× faster than the exact computations on the original floating-point numbers, while maintaining correlations with the true values of over .95. Moreover, at this level of correlation, Bolt can achieve 10-200× compression or more. These attributes make Bolt ideal as a subroutine in algorithms that are amenable to approximate computations, such as nearest neighbor search or maximum inner product search.

It is our hope that Bolt will be used in many production systems to greatly reduce storage and computation costs for large, real-valued datasets.

REFERENCES

[1] Nir Ailon and Bernard Chazelle. 2009. The Fast Johnson-Lindenstrauss Transform and Approximate Nearest Neighbors. *SIAM Journal on Computing (SICOMP)* 39, 1 (2009), 302–322. DOI: http://dx.doi.org/10.1137/060673096

[2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In *Advances in Neural Information Processing Systems*. 1225–1233.

[3] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: high-performance nearest neighbor search with product quantization fast scan. *Proceedings of the VLDB Endowment* 9, 4 (2015), 288–299.

[4] Artem Babenko, Relja Arandjelović, and Victor Lempitsky. 2016. Pairwise Quantization. *arXiv preprint arXiv:1606.01550* (2016).

[5] Artem Babenko and Victor Lempitsky. 2012. The inverted multi-index. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 3069–3076.

[6] Artem Babenko and Victor Lempitsky. 2014. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 931–938.

[7] Artem Babenko and Victor Lempitsky. 2015. Tree Quantization for Large-Scale Similarity Search and Classification. *CVPR* (2015), 1–9. papers3://publication/uuid/F4762974-BB97-4208-B035-508945A90EFC

[8] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[9] Mariusz Bojarski, Anna Choromanska, Krzysztof Choromanski, Francois Fagan, Cedric Gouy-Pailler, Anne Morvan, Nouri Sakr, Tamas Sarlos, and Jamal Atif. 2016. Structured adaptive and random spinners for fast machine learning computations. *arXiv preprint arXiv:1610.06209* (2016).

[10] Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. 2015. Compressing Neural Networks with the Hashing Trick.. In *ICML*. 2285–2294.

[11] Sanjoy Dasgupta and Anupam Gupta. 2003. An elementary proof of a theorem of Johnson and Lindenstrauss. *Random Structures & Algorithms* 22, 1 (2003), 60–65.

[12] M. Datar, N. Immorlica, Piotr Indyk, and V.S. Mirrokni. 2004. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (2004), 253–262. DOI: http://dx.doi.org/10.1145/997817.997857 arXiv:arXiv:1011.1669v3

[13] Matthijs Douze, Hervé Jégou, and Florent Perronnin. 2016. Polysemous codes. In *European Conference on Computer Vision*. Springer, 785–801.

[14] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2014), 744–755.

[15] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2013. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 12 (2013), 2916–2929.

[16] Gael Guennebaud, Benoit Jacob, and others. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).

[17] SH Gunderson. 2015. Snappy: A fast compressor/decompressor. *code. google. com/p/snappy* (2015).

[18] Ruiqi Guo, Sanjiv Kumar, Krzysztof Choromanski, and David Simcha. 2016. Quantization based fast inner product search. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. 482–490.

[19] Song Han, Jeff Pool, John Tran, and William J. Dally. 1995. Learning both Weights and Connections for Efficient Neural Networks. *The Lancet* 346, 8988 (1995), 1500–1501. DOI: http://dx.doi.org/10.1016/S0140-6736(95)92525-2 arXiv:arXiv:1506.02626v1

[20] Lu Hou, Quanming Yao, and James T Kwok. 2016. Loss-aware Binarization of Deep Networks. *arXiv preprint arXiv:1611.01600* (2016).

[21] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2011), 117–128.

[22] Jianqiu Ji, Jianmin Li, Shuicheng Yan, Bo Zhang, and Qi Tian. 2012. Super-bit locality-sensitive hashing. In *Advances in Neural Information Processing Systems*. 108–116.

[23] Yannis Kalantidis and Yannis Avrithis. 2014. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2321–2328.

[24] Weihao Kong and Wu-Jun Li. 2012. Isotropic hashing. In *Advances in Neural Information Processing Systems*. 1646–1654.

[25] Kasper Green Larsen and Jelani Nelson. 2014. The Johnson-Lindenstrauss lemma is optimal for linear dimensionality reduction. *arXiv preprint arXiv:1411.2404* (2014).

[26] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 1998. The MNIST database of handwritten digits. (1998).

[27] Shicong Liu, Junru Shao, and Hongtao Lu. 2016. Generalized Residual Vector Quantization for Large Scale Data. *Proceedings - IEEE International Conference on Multimedia and Expo* 2016-Augus (2016). DOI: http://dx.doi.org/10.1109/ICME.2016.7552944 arXiv:1609.05345

[28] Zelda Mariet and Suvrit Sra. 2015. Diversity networks. *arXiv preprint arXiv:1511.05077* (2015).

[29] Julieta Martinez, Joris Clement, Holger H Hoos, and James J Little. 2016. Revisiting additive quantization. In *European Conference on Computer Vision*. Springer, 137–153.

[30] Julieta Martinez, Holger H Hoos, and James J Little. 2014. Stacked quantizers for compositional vector compression. *arXiv preprint arXiv:1411.2173* (2014).

[31] Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas. 2016. ACDC: A Structured Efficient Linear Layer. *ICLR* 2 (2016), 1–11. arXiv:1511.05946 http://arxiv.org/abs/1511.05946

[32] Mohammad Norouzi and David J. Fleet. 2011. Minimal loss hashing for compact binary codes. In *Proceedings of the 28th international conference on machine learning (ICML-11)*. 353–360.

[33] Mohammad Norouzi and David J Fleet. 2013. Cartesian k-means. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3017–3024.

[34] Zhiyuan Tang, Dong Wang, and Zhiyong Zhang. 2016. Recurrent neural network training with dark knowledge transfer. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 5900–5904.

[35] Michail Vlachos, Nikolaos M. Freris, and Anastasios Kyrillidis. 2015. Compressive mining: Fast and optimal data mining in the compressed domain. *VLDB Journal* 24, 1 (2015), 1–24. DOI: http://dx.doi.org/10.1007/s00778-014-0360-3 arXiv:1405.5873

[36] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2016. Learning to hash for indexing big datafi!?a survey. *Proc. IEEE* 104, 1 (2016), 34–57.

[37] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927* (2014).

[38] Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. 2014. Deep Fried Convnets. *ICCV* (2014). DOI: http://dx.doi.org/10.1007/s13398-014-0173-7.2 arXiv:1412.7149

[39] Felix X Yu, Ananda Theertha Suresh, Krzysztof M Choromanski, Daniel N Holtmann-Rice, and Sanjiv Kumar. 2016. Orthogonal Random Features. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 1975–1983. http://papers.nips.cc/paper/6246-orthogonal-random-features.pdf

[40] Ting Zhang, Chao Du, and Jingdong Wang. 2014. Composite Quantization for Approximate Nearest Neighbor Search. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* 32 (2014), 838–846.