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 and model weights are at the heart of data mining. Storing such vectors accounts for most of the space usage of many algorithms, and operating on them accounts for most of the compute time. Recently, vector quantization methods have shown great promise in reducing these costs when operating on fixed collections of vectors. However, the high cost of accurately encoding the vectors using these schemes makes them less practical for changing collections, such as production databases, streams of sensor data, or machine learning model weights during training.

We introduce a vector quantization technique that can encode vectors up to 10× faster than existing schemes while also accelerating operations such as distance and dot product computations by over 5×. Moreover, because it can encode over two megabytes of vectors per millisecond (2 GB/s), it makes vector quantization cheap enough to employ as a subroutine to accelerate other algorithms.

Specifically, we show experimentally that our approach can be used to accelerate neural network training, k-means clustering, nearest neighbor search, and maximum inner product search by up to 20× compared to floating point operations and 5× compared to other vector quantization methods. Furthermore, our approximate Euclidean distance and dot product computations are faster not only than those of related algorithms with much slower encodings, but even faster than Hamming distance computations, which have direct hardware support on the tested platforms.

CCS CONCEPTS

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

KEYWORDS

ACM proceedings, LaTeX, text tagging

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),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

KDD 2017, Halifax, Nova Scotia Canada

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As datasets grow larger, so too do the costs of learning from them. These costs include not only the space to store the data, but also the compute time to operate on it. When the dataset is fixed, vector quantization methods enable significant reductions in both of these costs. By replacing each vector with a learned approximation, these methods both reduce the space needed to store the vectors and enable fast approximate scalar reductions, such as dot products and Euclidean distances. With as little as 8B per vector, these techniques can preserve distances and dot products with extremely high accuracy

However, computing the approximation for a given vector can be time-consuming. The state-of-the-art method of [LSQ], for example, requires up to *4ms* to encode a single 128-dimensional vector. Other techniques are faster, but as we show experimentally, virtually all suffer from encoding times that yield significant overhead when data is being added or changed rapidly.

We describe a vector quantization method, Bolt, that greatly reduces both the time to encode vectors and the time to compute scalar reductions over them. This makes encoding worthwhile even for fast-changing or fast-arriving data. Our key idea is to use much smaller codebooks than similar techniques for the quantization, which both facilitates finding the optimal encoding and allows scans over codes to be done in a vectorized manner.

Contributions:

1. A vector quantization algorithm that encodes vectors significantly faster than existing algorithms for a given level of representational fidelity.
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.
3. Empirical demonstration that approximate distances can be used to accelerate several popular algorithms, such as k-means clustering and word2vec word embedding.
4. Theoretical analysis of both our approach and popular related approaches.

1.1 Problem Statement

Formally, the problem our algorithm addresses is the following.

Let $\mathbf{q} \in \mathbb{R}^D$ be a *query* vector and let $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}, \mathbf{x}_i \in \mathbb{R}^D$ be a collection of *database* vectors. Further let $d : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ be a distance or similarity function that can be written as:

$$d(\mathbf{q}, \mathbf{x}) = f(\sum_{j=1}^{D} d_j(q_j, x_j)) \tag{1}$$

Table 1: Performance of Vector Quantization Algorithms. I should reference + explain this somewhere.

	Data Vector Encoding Speed	Query Vector Encoding Speed	Similarity/Distance Computation Speed	Compression
Bolt (proposed)	**Very High**	**Very High**	**Very High**	Medium
PQ	High	High	High	Low
OPQ	High	High	High	Medium
RVQ	Medium	High	High	Medium
GRVQ, LSQ, CQ, AQ, OTQ	Low, Very Low	Medium	High	**High**
Raw Floats	N/A	N/A	Low	None

where $f : \mathbb{R} \to \mathbb{R}$, $d_j : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$. This includes both distances in L_p spaces and dot products as special cases. In the former case, $d_j(q_j, x_j) = (q_j - x_j)^p$ and $f(r) = r^{(1/p)}$; in the latter case, $d_j(q_j, x_j) = q_j x_j$ 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 $g : \mathbb{R}^D \to \mathcal{G}$, $h : \mathbb{R}^D \to \mathcal{H}$, and $\hat{d} : \mathcal{G} \times \mathcal{H} \to R$ such that the loss:

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

is minimized. Moreover, each of these functions should be as fast to compute as possible.

Intuitively, the function g encodes the query, h encodes the database vectors, and $\hat{d}$ computes an approximate similarity or distance based on the encodings. In general, it is possible to drive the loss arbitrarily low by increasing the time and space usage of these functions. Indeed, the loss can be fixed at 0 by setting g and h to identity functions and setting $\hat{d} = d$. Consequently, our metric of interest is **computation time for a given loss**. The primary contribution of this work is the introduction of g, h and $\hat{d}$ functions that are significantly faster than those of existing work for a given loss $\mathcal{L} > 0$.

1.2 Assumptions

Like other work [][][][], we assume that there is an initial training phase during which the functions g and h may be learned. This phase contains a training dataset for X, but not necessarily for $\mathbf{q}$. Following this training phase, there is a testing 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. $\mathbf{x}$ vectors 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 recieved [][][][].

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 [][] or Locality-Sensitive Hashing hash tables [][], 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.

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, so our review will necessarily be incomplete. We refer the reader to [learningToHash, thatOtherSurvey] for detailed surveys.

Many existing approaches in the computer vision and information retrieval literature fall into one of two categories [thatOtherSurvey]: binary embedding and vector quantization. Binary embedding techniques seek to map vectors in $\mathbb{R}^D$ to B-dimensional Hamming space, typically with $B < D$. 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 [][]. In fact, He et al. [] demonstrated that the popular technique of [ITQ] is a more constrained version of their vector quantization algorithm, and that the objective function of another state-of-the art binary embedding [isohash], can be understood as maximizing only one of two sufficient conditions for optimal encoding of gaussian data.

Vector quantization approaches yield lower errors 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) [], 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 [] / Optimized Product Quantization (OPQ) [], Additive Quantization (AQ) [], Residual Vector Quantization (RVQ) [], Generalized Residual Vector Quantization (GRVQ), Composite Quantization (CQ) [], Optimized Tree Quantization (OTQ) [], and Local Search Quantization (LSQ) []. 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. []. 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. Like our own algorithm, it seeks the best of both binary embedding speed and vector quantization accuracy. However, our technique obtains this by speeding up the vector quantization distance computations directly, obviating the need for multiple passes and the possibility of false dismissals from unrepresentative Hamming distances.

In the machine learning community, accelerating vector operations has been done primarily through 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 [JL][JLIsTight], and even stronger empirical results [SuperBitLSH, thatWeirdCompressoinOne]. However, because embedding *per se* only entails reducing the number of floating-point numbers stored, without reducing the size of each, we find that it is not at all competitive with vector quantization methods on its own. Moreover, even if it were, 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 [adaptiveFastfood][hashNets], the Fast Fourier Transform [][], the Discrete Cosine Transform [], or the Fast Hadamard Transform [crossPolytope][][], possibly with learned elementwise weights, instead of performing a matrix multiply. These methods have strong theoretical grounding [structuredSpinners] and sometimes outperform non-structured matrices [adaptiveFastfood]. 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 nerual network weights, to fewer bits [][]; 2) reusing weights [hashNets][]; 3) pruning weights in a model after training [diversityNets][learningWeightsAndConnections]; and 4) training a small model to approximate the outputs of a larger model [][]. 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 uncompressed weights and data matching the form of Eq **??**.

In addition to all of the above approaches, there is the possibility of running algorithms on GPUs [], clusters [], ASICs [], or FPGAs []. Since such alternate hardware could accelerate both our own algorithm and the most similar comparisons, we do not address this possibility further.

3 METHOD

As mentioned in the problem statement, our goal is to construct a distance (or similarity) 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 (or similarity) function d. To explain how we do this, we first begin with a review of Product Quantization, and then describe how our method differs.

3.1 Background: Product Quantization

Recall that, by assumption, the function d can be written as:

$$d(\mathbf{q}, \mathbf{x}) = f(\sum_{j=1}^{D} d_j(q_j, x_j))$$

where $f : \mathbb{R} \to \mathbb{R}$, $d_j : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$. Now, suppose one has a partition $\mathcal{P} = \{p_1, \ldots, p_M\}$ of the indices j, such that $i \neq j \implies p_i \cap p_j = \emptyset$ and $\bigcup_m p_m = \{1, \ldots, D\}$. The argument to f can then be written as:

$$\sum_{m=1}^{M} \sum_{j \in p_m} d_j(q_j, x_j) = \sum_{m=1}^{M} d_m(\mathbf{q}_m, \mathbf{x}_m)$$

where $\mathbf{q}_m$ and $\mathbf{x}_m$ are the vectors formed by gathering the indices of $vecq$ and $\mathbf{x}$ at the indices $j \in p_m$, and d_m sums the relevant d_j functions. The idea of Product Quantization (PQ) is to replace each x_m with one vector $\mathbf{c}_m$ from a *codebook* C_m of possibilities. That is:

$$\sum_{m=1}^{M} d_m(\mathbf{q}_m, \mathbf{x}_m) \approx \sum_{m=1}^{M} d_m(\mathbf{q}_m, \mathbf{c}_m)$$

This allows one to store only m indices into codebooks instead of the original vector $\mathbf{x}$. Moreover, under the assumption that $x \sim MVN(\mu, \Sigma)$ with $\Sigma_{ij} = 0$ for all i, j in different subspaces p_m and $|\Sigma_m| = const$, where Σ_m is the covariance within subspace p_m, this formulation achieves the information-theoretic lower bound on code length for a given squared error [OPQ].

Using a fixed set of codebooks also enables construction of a fast query encoding g and distance approximation $\hat{d}$.

SELF: pick up here by explaining LUT creation.

4 RESULTS

show that it makes matrix-vector muls way faster for various matrix sizes -also show approximation error -which suggests getting lots of meaningful matrices, ideally fc layer weights -prolly have to show both speed and err as a function of code length same for matrix-matrix; speed and acc -is there any way we'll be better here? maybe for skinny mats...

show that we can speed up kmeans a lot, and point out that this is orthogonal to other ppl's speedups based on pruning which ones you consider moving and/or using minibatches -prolly pick a couple datasets and show it as a function of k

a couple experiments on sift1m scan and mnist scan, prolly; maybe sift10m / deep10m also

somewhere introduce the "cold scan" problem, wherein we also have to add in time to compress everything, but then we get a batch of queries -if just 1 query, you're always worse off doing the encoding -metric of interest is how many queries you have to do @k database vectors before it's faster than a matmul -and

you should also report the times for different query counts and db counts

-and prolly also the ”warm scan” problem; you only have to encode some subset of the db (cuz it changed)

-or maybe the ”hot scan” problem cuz data is hot; in contrast to fixed data, which is cold/frozen

5 CONCLUSION