Building Efficient Algorithms by Learning to Compress

by

Davis W. Blalock

B.S., University of Virginia (2014)
S.M., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 28, 2020

Certified by .....
John V. Guttag
Dugald C. Jackson Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Building Efficient Algorithms by Learning to Compress

by

Davis W. Blalock

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and
Computer Science

Abstract

The amount of data in the world is doubling every two years. Such abundant data offers immense opportunities, but also imposes immense computation, storage, and energy costs. This thesis introduces efficient algorithms for reducing these costs for bottlenecks in real world data analysis and machine learning pipelines.

Concretely, we introduce algorithms for:

- Lossless compression of time series. This algorithm compresses better than any existing method, despite requiring only the resources available on a low-power edge device.
- Approximate matrix-vector multiplies. This algorithm accelerates approximate similarity scans by an order of magnitude relative to existing methods.
- Approximate matrix-matrix multiplies. This algorithm often outperforms existing approximation methods by more than 10× and non-approximate computation by more than 100×.

We provide extensive empirical analyses of all three algorithms using real-world datasets and realistic workloads. We also prove bounds on the errors introduced by the two approximation algorithms.

The theme unifying all of these contributions is learned compression. While compression is typically thought of only as a means to reduce data size, we show that specially designed compression schemes can also dramatically increase computation speed and reduce memory requirements.

Thesis Supervisor: John V. Guttag

Title: Dugald C. Jackson Professor of Electrical Engineering and Computer Science

Acknowledgments

This PhD would not have been possible without the support and efforts of many people.

At the top of this list is my advisor, John Guttag. It's difficult to identify all the positive aspects of having John as an advisor, so I will stick to only a few key points. First, John is extremely supportive and kind—he genuinely wants what's best for his graduate students and is generous and understanding far beyond the call of duty. In addition, one could not ask for a more knowledgeable mentor; John has deep expertise in many areas of computer science, and I doubt I could have tackled the problems I did without his willingness and ability to straddle many subfields at once. This knowledgeability also extends to the practice of research itself; John can often transform writing, presentations, and problem framings from lackluster to exceptional in the span of a single round of feedback.

I am also indebted to my collaborators and labmates. I would particularly like to thank Divya Shanmugam and Jose Javier Gonzalez Ortiz for managing to put up with me over the course of multiple research projects. I would also like to thank Anima, Yun, Guha, Joel, Jen, Amy, Maggie, Adrian, Tristan, Marzyeh, Tiam, Harini, Katie, Marianne, Emily, Addie, Wayne, Sra, Matt, Dina, Maryann, Roshni, and Aniruddh for their friendship, feedback, and ideas over the years.

Along similar lines, I would like to thank Sam Madden and Tamara Broderick for serving on my thesis committee and being great collaborators in research (and other) endeavors in the past few years. Sam and Tamara not only possess deep expertise in their fields, but are also enjoyable and interesting to work with.

I would be remiss not to include some mentors from before MIT. Foremost, I'd like to thank John Lach and Ben Boudaoud for taking a chance on me when I was an undergrad who didn't know anything. I'd also like to thank Jeff, Kevin, Jermaine, Jim, Jake, Drake, and the rest of the PocketSonics team for mentoring me throughout many years of internships.

Finally, I should probably mention my family, who may have had some tangential

influence on me getting where I am today. But seriously, I could never thank them enough or adequately describe their positive impact on my life in a paragraph, so I will just note that I'm profoundly grateful for all the sacrifices they've made for me and for the good fortune of getting to be part of this family.

Contents

List of Figures

Chapter 1

Introduction

As datasets grow larger, so too do the costs of using them. These costs come in the form of computation, storage, memory, network, and energy usage. This thesis introduces algorithms to reduce these costs for various common elements of real-world data collection and analysis pipelines.

The central theme of our work is using machine learning to construct and manipulate efficient, low-level representations of data. We show that by learning to compress data into specific, algorithm-friendly formats, we can obtain dramatic speed and space savings—often 10× to 100×—compared to either the original data or vector-valued embeddings. This is not merely a matter of quantizing floating point scalars, projecting data into lower-dimensional spaces, or otherwise employing well-known techniques; it is instead a matter of examining the "full stack," from application requirements to processor instruction sets, in order to jointly formalize problems, define efficient algorithms, and design compute-friendly data representations. Where applicable, we also prove theoretical guarantees about the resulting methods.

To make these claims more concrete, it is helpful to introduce the three problems discussed in this thesis:

Compression of time series data (Chapter 2). Time series constitute a large and growing fraction of the world's data, but there has been little work on how best to compress them. Such compression is especially important at the point of

collection; low-power devices such as wearable health trackers must expend a great deal of energy to transmit data, so reductions in data size yield vital reductions in power consumption. We introduce a lossless compression algorithm for time series that achieves state-of-the-art compression ratios and speed, despite using only the limited resources available on low-power hardware. A key element of this method is an algorithm for simultaneous online prediction of and training on samples that operates at over 5GB/s in a single CPU thread. This extreme speed makes it practical to replace fixed compression heuristics with a learned model, yielding significant space savings.

One application of this method is in gathering data from fitness wearables for gesture and activity recognition. This task is difficult and has attracted enormous attention both in industry [7, 119, 8, 6] and academia [88, 134, 135, 136, 89, 151, 30, 66, 21]. In order to construct accurate recognition models, it is desirable to collect as much accelerometer, gyroscope, and other data from each device as possible. However, transmitting this data from a wearable device to a phone or the cloud consumes precious battery life. This has led to interest in on-device compression [21]. We demonstrate using numerous datasets acquired from both wearable devices and a wide range of other sensors that our compression algorithm significantly outperforms existing alternatives. Moreover, it does so despite using 100× less memory than many competing methods.

Acceleration of Similarity Scans (Chapter 3). Similarity search is one of the core operations behind modern web and mobile applications. Such searches decompose into a coarse retrieval step and a linear scan step. We describe an algorithm to dramatically accelerate the linear scan step. It is conceptually similar to existing linear scan algorithms, but achieves large speedups by taking into account characteristics of modern hardware. The core of the method is a learning-based algorithm for compressing the database being searched, along with a compression format that can be scanned without decompressing the data.

Among other applications, the algorithms our method improves upon are used for

image retrieval at Google [79, 163] and Facebook [4]. We demonstrate our method's superiority on various benchmark datasets for both similarity search in general and image retrieval in particular.

Fast Approximate Matrix Products (Chapter 4). Matrix multiplication is one of the most fundamental operations in machine learning and data analysis. It is also the computational bottleneck in many workloads. We introduce an algorithm to quickly approximate matrix products that greatly outperforms existing methods for realistic matrix sizes. Our approach is a significant methodological departure from most work in this area in two ways: first, it exploits a training dataset to learn useful approximations; and second, it uses *nonlinear* transformations of the matrices, a counter-intuitive approach when approximating linear operations. Our method is well suited for accelerating neural network layers, which have linear operations as their computational bottleneck.

As stated previously, these methods are unified by a pattern of learning efficient representations and designing algorithms that exploit these representations.

In the first algorithm, we learn to represent time series as a sequence of low-bitwidth integers in a vectorizable data layout. The learning comes in the form of the online forecaster. Reducing space is the main focus, but, as we demonstrate, our compressed format can also be decompressed at extreme speed—often 2GB/s or more in a single CPU thread.

In the second algorithm, we learn to map vectors of floating point numbers to vectors of categorical values, each of which can be stored using only a few bits. This representation is both more compact than the original and much faster to operate on; for example, it allows us to approximate 256-element dot products in roughly two CPU cycles.

In the third algorithm, the form of the learned representations is the same as in the second, but the functions creating and using them are more efficient for matrix multiplication. First, we replace an exact clustering step with a learned locality-

sensitive hash function. Second, we replace an exact summation with a noisy estimator whose errors we analyze and correct for in closed form. And finally, we relax a set of constraints on parameters that other methods cannot relax without an immense performance penalty. These changes significantly increase quality for a given level of speed; among other results, we accelerate a softmax classifier on the CIFAR-10 dataset by a factor of 15× with only a .25% drop in accuracy, or over 100× with a 1.1% drop in accuracy, while also compressing the input representation by factors of 128× and 512× respectively. Our second algorithm, as well as all other existing methods, are more than an order of magnitude slower at these accuracies.

We discuss these three algorithms in more detail in Chapters 2-4, and conclude in Chapter 5 with additional observations and a discussion of the lessons that can be drawn from this work.

Chapter 2

Compressing Integer Time Series

2.1 Introduction

Thanks to the proliferation of smartphones, wearables, autonomous vehicles, and other connected devices, it is becoming common to collect large quantities of sensor-generated time series. Once this data is centralized in servers, many tools exist to analyze and create value from it [184, 50, 169, 160, 29, 178, 152, 162]. However, centralizing the data can be challenging because of power constraints on the devices collecting it. In particular, transmitting data wirelessly is extremely power-intensive—on a representative set of chips [95, 96], transmitting data over Bluetooth Low Energy (BLE) costs tens of *milli*watts, while computing at full power costs only tens of *micro*watts.

One strategy for reducing this power consumption is to extract information locally and only transmit summaries [172, 11, 35]. This can be effective in some cases, but requires both a predefined use case for which a summary is sufficient and an appropriate method of constructing this summary. Devising such a method can be a significant endeavor [172]. For common but difficult tasks such as gesture and activity recognition from wearables [7, 119, 8, 6, 88, 135, 136, 89, 151, 30, 66, 21], collecting large quantities of sensor data is desirable, and it is not clear how appropriate on-device summaries could be constructed; furthermore, even if some summarization method worked well for all present use cases, discarding the raw data would both

make further algorithmic advances difficult and reduce the competitive moat available to the device's manufacturer.

A complementary and more general approach is to compress the data before transmitting it [11, 21, 93, 33, 170]. This allows arbitrary subsequent analysis and does not require elaborate summary construction algorithms. Unfortunately, existing compression methods either 1) are only applicable for specific types of data, such as timestamps [148, 14, 118], audio [41, 154, 2, 138] or EEG [172, 124] recordings; or 2) use algorithms that are ill-suited to sensor-generated time series.

More specifically, existing methods (e.g., [122, 63, 133, 105, 43, 44, 55, 69, 116]) violate one or more of the following design requirements:

1. **Small block size**. On devices with only a few kilobytes of memory, it is not possible to buffer large amounts of data before compressing it. Moreover, even with more memory, buffering can add unacceptable latency; for example, a smartwatch transmitting nine axes of 8-bit motion data at 20Hz to a smartphone would need to wait $10000/(9 \times 1 \times 20) = 56$ seconds to fill even a 10KB buffer. This precludes using this data for gesture recognition and would add unacceptable user interface latency for step counting, activity recognition, or most other purposes.
2. **High decompression speed**. While the device collecting the data may not need to decompress it, it is desirable to have an algorithm that could also function well in a central database. This eliminates the need to transcode the data at the server and simplifies the application. In a database, time series workloads are not only read-heavy [35, 14, 29], but often necessitate materializing data (or downsampled versions thereof) for visualization, clustering, computing correlations, or other operations [35]. At the same time, writing is often append-only [148, 35]. As a result, decompression speed is paramount, while compression speed need only be fast enough to keep up with the rate of data ingestion.
3. **Lossless**. Given that time series are almost always noisy and often oversampled, it might not seem necessary to compress them losslessly. However, noise and oversampling 1) tend to vary across applications, and 2) are often best addressed in an application-specific way as a preprocessing step. Consequently, instead of assuming

that some level of downsampling or some particular smoothing will be appropriate for all data, it is better for the compression algorithm to preserve what it is given and leave preprocessing up to the application developer.

The primary contribution of this work is Sprintz, a compression algorithm for time series that offers state-of-the-art compression ratios and speed while also satisfying all of the above requirements. It requires <1KB of memory, can use blocks of data as small as eight samples, and can decompress at up to 3GB/s in a single thread. Sprintz's effectiveness stems from exploiting 1) temporal correlations in each variable's value and variance, and 2) the potential for parallelization across different variables, realized through the use of vector instructions. Sprintz operates directly only on integer time series. However, as we discuss in Section 2.5.8, straightforward preprocessing allows it to be applied to most floating point time series as well.

A key component of Sprintz's operation is a novel, vectorized forecasting algorithm for integers. This algorithm can simultaneously train online and generate predictions at close to the speed of `memcpy`, while significantly improving compression ratios compared to delta coding.

A second contribution is an empirical comparison of a range of algorithms currently used to compress time series, evaluated across a wide array of public datasets. We also make available code to easily reproduce these experiments, including the plots and statistical tests in the paper. To the best of our knowledge, this constitutes the largest public benchmark for time series compression.

The remainder of this paper is structured as follows. In Section 2.2, we introduce relevant definitions, background, and details regarding the problem we consider. In Section 4.2, we survey related work and what distinguishes Sprintz. In Sections 4.4 and 4.5, we describe Sprintz and evaluate it across a number of publicly-available datasets. We also discuss when Sprintz is advantageous relative to other approaches.

2.2 Definitions and Background

Before elaborating upon how our method works, we introduce necessary definitions and provide relevant information regarding the problem being solved.

2.2.1 Definitions

Definition 2.2.1. ***Sample.*** *A sample is a vector $\boldsymbol{x} \in \mathbb{R}^D$. D is the sample's* ***dimensionality****. Each element of the sample is an integer represented using a number of bits w, the* ***bitwidth****. The bitwidth w is shared by all elements.*

Definition 2.2.2. ***Time Series.*** *A time series $\boldsymbol{X}$ of length T is a sequence of T samples, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$. All samples $\boldsymbol{x}_t$ share the same bitwidth w and dimensionality D. If $D = 1$, $\boldsymbol{X}$ is called* ***univariate****; otherwise it is* ***multivariate****.*

Definition 2.2.3. ***Rows, Columns.*** *When represented in memory, we assume that each sample of a time series is one row and each dimension is one column. Because data arrives as samples and memory constraints may limit how many samples can be buffered, we assume that the data is stored in row-major order—i.e., such that each sample is stored contiguously.*

2.2.2 Hardware Constraints

Many connected devices are powered by batteries or harvested energy [83]. This results in strict power budgets and, in order to satisfy them, omission of certain functionality. In particular, many devices lack hardware support for floating point operations, SIMD (vector) instructions, and integer division. Moreover, they often have no more than a few kilobytes of memory, clocks of tens of MHz at most, and 8-, 16-, or 32-bit processors instead of 64-bit [95, 96, 5].

In contrast, we assume that the hardware used to decompress the data does not share these limitations. It is likely a modern server with SIMD instructions, gigabytes of RAM, and a multi-GHz clock. However, because the amount of data it must

store and query can be large, compression ratio and decompression speed are still important.

2.2.3 Data Characteristics

From a compression perspective, time series have four attributes uncommon in other data.

1. **Lack of exact repeats**. In text or structured records, there are many sequences of bytes—often corresponding to words or phrases—that will exactly repeat many times. This makes dictionary-based methods a natural fit. In time series, however, the presence of noise makes exact repeats less common [30, 150].
2. **Multiple variables**. Real-world time series often consist of multiple variables that are collected and accessed together. For example, the Inertial Measurement Unit (IMU) in modern smartphones collects three-dimensional acceleration, gyroscope, and magnetometer data, for a total of nine variables sampled at each time step. These variables are also likely to be read together, since each on its own is insufficient to characterize the phone's motion.
3. **Low bitwidth**. Any data collected by a sensor will be digitized into an integer by an Analog-to-Digital Converter (ADC). Nearly all ADCs have a precision of 32 bits or fewer [3], and typically 16 or fewer of these bits are useful. For example, even lossless audio codecs store only 16 bits per sample [41, 154]. Even data that is not collected from a sensor can often be stored using six or fewer bits without loss of performance for many tasks [150, 90, 122].
4. **Temporal correlation**. Since the real world usually evolves slowly relative to the sampling rate, successive samples of a time series tend to have similar values. However, when multiple variables are present and samples are stored contiguously, this correlation is often present only with a lag—e.g., with nine IMU variables, every ninth value is similar. Lag correlations violate the assumptions of most compressors, which treat adjacent bytes as the most likely to be related.

Much of the reason SPRINTZ outperforms existing methods is that it exploits or accounts for all of these characteristics, while existing methods do not.

2.3 Related Work

SPRINTZ draws upon ideas from time series compression, time series forecasting, integer compression, general-purpose compression, and high-performance computing. From a technical perspective, SPRINTZ is unusual or unique in its abilities to:

1. Bit pack with extremely small block sizes
2. Bit pack low-bitwidth integers effectively
3. Efficiently exploit correlation between nearby samples in *multivariate* time series
4. Naturally integrate both run-length encoding and bit packing
5. Exploit vectorized hardware through forecaster, learning algorithm, and bit packing method co-design

From an application persective, SPRINTZ is distinct in that it enables higher-ratio lossless compression with far less memory and latency than competing methods.

2.3.1 Compression of Time Series

Most work on compressing time series has focused on lossy techniques. The most common approach is to approximate the data as a sequence of low-order polynomials [106, 116, 63, 170, 105, 104]. An alternative, commonly seen in the data mining literature, is to discretize the time series using Symbolic Aggregate Approximation (SAX) [122] or its variations [159, 37]. These approaches are designed to preserve enough information about the time series to support indexing or specific data mining algorithms (e.g. [159, 149, 107]), rather than to compress the time series *per se*. As a result, they are extremely lossy; a hundred-sample time series might be compressed into one or two bytes, depending on the exact discretization parameters.

For audio time series specifically, there are a large number of lossy codecs [138, 154, 2, 171], as well as a small number of lossless [41, 20] codecs. In principle, some

of these could be applied to non-audio time series. However, modern codecs make such strong assumptions about the possible numbers of channels, sampling rates, bit depths, or other characteristics that it is infeasible to use them on non-audio time series.

Many fewer algorithms exist for lossless time series compression. For floating point time series, the only algorithm of which we are aware is that of the Gorilla database [148]. This method XORs each value with the previous value to obtain a diff, and then bit packs the diffs. In contrast to our approach, it assumes that time series are univariate and have 64-bit floating point elements.

For lossless compression of integer time series (including timestamps), existing approaches include directly applying general-purpose compressors [35, 162, 157, 84, 178], (double) delta encoding and then applying an integer compressor [29, 148], or predictive coding and byte packing [114]. These approaches can work well, but, as we will show, tend to offer both less compression and less speed than Sprintz.

2.3.2 Compression of Integers

The fastest methods of compressing integers are generally based on bit packing—i.e., using at most b bits to represent values in $\{0, 2^b - 1\}$, and storing these bits contiguously [161, 190, 118]. Since b is determined by the largest value that must be encoded, naively applying this method yields limited compression. To improve it, one can encode fixed-size blocks of data at a time, so that b can be set based on the largest values in a block instead of the whole dataset [156, 190, 118]. A further improvement is to ignore the largest few values when setting b and store their omitted bits separately [190, 118].

Sprintz bit packing differs significantly from existing methods in two ways. First, it compresses much smaller blocks of samples. This reduces its throughput as compared to, e.g., FastPFor [118], but significantly improves compression ratios (c.f. Section 4.5). This is because large values only increase b for a few samples instead of for many. Second, Sprintz is designed for 8- and 16-bit integers, rather than 32- or 64-bit integers. Existing methods are often inapplicable to lower-bitwidth data (unless

converted to higher-bitwidth data) thanks to strong assumptions about bitwidth and data layout.

A common [41, 154] alternative to bit packing is Golomb coding [73], or its special case Rice coding [153]. The idea is to assume that the values follow a geometric distribution, often with a rate constant fit to the data.

Both bit packing and Golomb coding are bit-based methods in that they do not guarantee that encoded values will be aligned on byte boundaries. When this is undesirable, one can employ byte-based methods such as 4-Wise Null Suppression [156], LEB128 [45], or Varint-G8IU [165]. These methods reduce the number of bytes used to store each sample by encoding in a few bits how many bytes are necessary to represent its value, and then encoding only that many bytes. Some, such as Simple8B [19] and SIMD-GroupSimple [187], allow fractional bytes to be stored while preserving byte alignment for groups of samples.

2.3.3 General-Purpose Compression

A reasonable alternative to using a time series compressor would be to apply a general-purpose compression algorithm, possibly after delta coding or other preprocessing. Thanks largely to the development of Asymmetric Numeral Systems (ANS) [60] for entropy coding, general purpose compressors have advanced greatly in recent years. In particular, Zstd [44], Brotli [13], LZ4 [43] and others have attained speed-compression tradeoffs significantly better than traditional methods such as GZIP [69], LZO [144], etc. However, these methods have much higher memory requirements than Sprintz and, empirically, often do not compress time series as well and/or decompress as quickly.

2.3.4 Predictive Filtering

For numeric data such as time series, there are four types of predictive coding commonly in use: predictive filtering [34], delta coding [118, 161], double delta coding [29, 148], and XOR-based encoding [148]. In predictive filtering, each prediction is a

linear combination of a fixed number of recent samples. This can be understood as an autoregressive model or the application of a Finite Impulse Response (FIR) filter. When the filter is learned from the data, this is termed "adaptive filtering." Many audio compressors use some form of adaptive filtering [154, 41, 2].

Delta coding is a special case of predictive filtering where the prediction is always the previous value. Double delta coding, also called delta-delta coding or delta-of-deltas coding, consists of applying delta coding twice in succession. XOR-based encoding is similar to delta coding, but replaces subtraction of the previous value with the XOR operation. This modification is often desirable for floating point data [148].

Our forecasting method can be understood as a special case of adaptive filtering. While adaptive filtering is a well-studied mathematical problem in the signal processing literature, we are unaware of a practical algorithm that attains speed within an order of magnitude of our own. I.e., our method's primary novelty is as a vectorized *algorithm* for fitting and predicting multivariate time series, rather than as a mathematical *model* of multivariate time series. That said, it does incorporate different modeling assumptions than other compression algorithms for time series in that it reduces the model to one parameter and omits a bias term; this imposed structure enables the model's high speed.

2.4 Method

To describe how Sprintz works, we first provide an overview of the algorithm, then discuss each of its components in detail.

2.4.1 Overview

Sprintz is a bit-packing-based predictive coder. It consists of four components:

1. **Forecasting.** Sprintz employs a forecaster to predict each sample based on previous samples. It encodes the difference between the next sample and the predicted

sample, which is typically smaller in magnitude than the next sample itself.

2. **Bit packing.** SPRINTZ then bit packs the errors as a "payload" and prepends a header with sufficient information to invert the bit packing.
3. **Run-length encoding.** If a block of errors is all zeros, SPRINTZ waits for a block in which some error is nonzero and then writes out the number of all-zero blocks instead of the empty payload.
4. **Entropy coding.** SPRINTZ Huffman codes the headers and payloads.

These components are run on blocks of eight samples (motivated in Section 2.4.3), and can be modified to yield different compression-speed tradeoffs. Concretely, one can 1) skip entropy coding for greater speed and 2) choose between delta coding and our online learning method as forecasting algorithms. The latter is slightly slower but often improves compression.

We chose these steps since they allow for high speed and exploit the characteristics of time series. Forecasting leverages the high correlation of nearby samples to reduce the entropy of the data. Run-length encoding allows for extreme compression in the common scenario that there is no change in the data—e.g., a user's smartphone may be stationary for many hours while the user is asleep. Our method of bit packing exploits temporal correlation in the variability of the data by using the same bitwidth for points that are within the same block. Huffman coding is not specific to time series but has low memory requirements and improves compression ratios.

An overview of how SPRINTZ compresses one block of samples is shown in Algorithm 1. In lines 2-5, SPRINTZ predicts each sample based on the previous sample and any state stored by the forecasting algorithm. For the first sample in a block, the previous sample is the last element of the previous block, or zeros for the initial block. In lines 6-8, SPRINTZ determines the number of bits required to store the largest error in each column and then bit packs the values in that column using that many bits. (Recall that each column is one variable of the time series). If all columns require 0 bits, SPRINTZ continues reading in blocks until some error requires >0 bits (lines 11-13). At this point, it writes out a header of all zeros and then the number of all-zero blocks. Finally, it writes out the number of bits required by each column

in the latest block as a header, and the bit packed data as a payload. Both header and payload are compressed with Huffman coding.

Algorithm 1 encodeBlock($\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_B\}$, `forecaster`)

1: Let `buff` be a temporary buffer
2: **for** $i \leftarrow 1, \ldots, B$ **do** // For each sample
3: $\hat{\boldsymbol{x}}_i \leftarrow$ `forecaster`.predict($\boldsymbol{x}_{i-1}$)
4: $\texttt{err}_i \leftarrow \boldsymbol{x}_i - \hat{\boldsymbol{x}}_i$
5: `forecaster`.train($\boldsymbol{x}_{i-1}$, $\boldsymbol{x}_i$, $\texttt{err}_i$)
6: **for** $j \leftarrow 1, \ldots, D$ **do** // For each column
7: $\texttt{nbits}_j \leftarrow \max_i\{\text{requiredNumBits}(\texttt{err}_{ij})\}$
8: $\texttt{packed}_j \leftarrow \text{bitPack}(\{\texttt{err}_{1j}, \ldots, \texttt{err}_{Bj}\}, \texttt{nbits}_j)$
9: // Run-length encode if all errors are zero
10: **if** $\texttt{nbits}_j$ == 0, $1 \leq j \leq D$ **then**
11: **repeat** // Scan until end of run
12: Read in another block and run lines 2-8
13: **until** $\exists_j[\texttt{nbits}_j \neq 0]$
14: Write D 0s as headers into `buff`
15: Write number of all-zero blocks as payload into `buff`
16: Output huffmanCode(`buff`)
17: Write $\texttt{nbits}_j$, $j = 1, \ldots, D$ as headers into `buff`
18: Write $\texttt{packed}_j$, $j = 1, \ldots, D$ as payload into `buff`
19: Output huffmanCode(`buff`)

Sprintz begins decompression (Algorithm 2) by decoding the Huffman-coded bitstream into a header and a payload. Once decoded, these two components are easy to separate since the header is always first and of fixed size. If the header is all zeros, the payload indicates the length of a run of zero errors. In this case, Sprintz runs the predictor until the corresponding number of samples have been predicted. Since the errors are zero, the forecaster's predictions are the true sample values. In the nonzero case, Sprintz unpacks the payload using the number of bits specified for each column by the header.

2.4.2 Forecasting

Sprintz can, in principle, employ an arbitrary forecasting algorithm. To achieve our desired decompression speeds, however, we restrict our focus to two methods: delta coding and Fire (Fast Integer REgression), a forecasting algorithm we introduce.

Algorithm 2 decodeBlock(`bytes`, B, D, `forecaster`)

```
nbits, payload ← huffmanDecode(bytes, B, D)
if nbits_j == 0 ∀j then                          // Run-length encoded
    numblocks ← readRunLength()
    for i ← 1, ..., (B · numblocks) do
        x_i ← forecaster.predict(x_{i−1})
        Output x_i
else                                             // Not run-length encoded
    for i ← 1, ..., B do
        x̂_i ← forecaster.predict(x_{i−1})
        err_i ←unpackErrorVector(i, nbits, payload)
        x_i ← err_i + x̂_i
        Output x_i
        forecaster.train(x_{i−1}, x_i, err_i)
```

Delta Coding

Forecasting with delta coding consists of predicting each sample $\boldsymbol{x}_i$ to be equal to the previous sample $\boldsymbol{x}_{i-1}$, where $\boldsymbol{x}_0 \triangleq \mathbf{0}$. This method is stateless given $\boldsymbol{x}_{i-1}$ and is extremely fast. It is particularly fast when combined with run-length encoding, since it yields a run of zero errors if and only if the data is constant. This means that decompression of runs requires only copying a fixed vector, with no additional forecasting or training. Moreover, when answering queries, one can sometimes avoid decompression entirely—e.g., one can compute the max of all samples in the run by computing the max of only the first value.

FIRE

Forecasting with FIRE is slightly more expensive than delta coding but often yields better compression. The basic idea of FIRE is to model each value as a linear combination of a fixed number of previous values and learn the coefficients of this combination. Specifically, we learn an autoregressive model of the form:

$$x_i = ax_{i-1} + bx_{i-2} + \varepsilon_i \tag{2.1}$$

where x_i denotes the value of some variable at time step i and ε_i is a noise term.

Different values of a and b are suitable for different data characteristics. If $a = 2$, $b = -1$, we obtain double delta coding, which extrapolates linearly from the previous two points and works well when the time series is smooth. If $a = 1$, $b = 0$, we recover delta coding, which models the data as a random walk. If $a = \frac{1}{2}$, $b = \frac{1}{2}$, we predict each value to be the average of the previous two values, which is optimal if the x_i are i.i.d. Gaussians. In other words, these cases are appropriate for successively noisier data.

The reason FIRE is effective is that it learns online what the best coefficients are for each variable. To make prediction and learning as efficient as possible, FIRE restricts the coefficients to lie within a useful subspace. Specifically, we exploit the observation that all of the above cases can be written as:

$$x_i = x_{i-1} + \alpha x_{i-1} - \alpha x_{i-2} + \varepsilon_i \tag{2.2}$$

for $\alpha \in [-\frac{1}{2}, 1]$. Letting $\delta_i \triangleq x_i - x_{i-1}$ and subtracting x_{i-1} from both sides, this is equivalent to

$$\delta_i = \alpha \delta_{i-1} + \varepsilon_i \tag{2.3}$$

This means that we can capture all of the above cases by predicting the next delta as a rescaled version of the previous delta. This requires only a single addition and multiplication, reducing the learning problem to that of finding a suitable value for a single parameter.

To train and predict using this model, we use the functions shown in Algorithm 3. First, to initialize a FIRE forecaster, one must specify three values: the number of columns D, the learning rate η, and the bitwidth w of the integers stored in the columns. Internally, the forecaster also maintains an accumulator for each column (line 4) and the difference (delta) between the two most recently seen samples (line 5). The accumulator is a scaled version of the current α value with a bitwidth of $2w$. It enables fast updates of α with greater numerical precision than would be possible if modifying α directly. The accumulators and deltas are both initialized to zeros.

Algorithm 3 FIRE_Forecaster Class

function INIT(D, η, w)
    `self`.learnShift $\leftarrow \lg(\eta)$
    `self`.bitWidth $\leftarrow w$ // 8-bit or 16-bit
    `self`.accumulators $\leftarrow$ zeros(D)
    `self`.deltas $\leftarrow$ zeros(D)
function PREDICT($\boldsymbol{x}_{i-1}$)
    `alphas` $\leftarrow$ `self`.accumulators $\gg$ `self`.learnShift
    $\hat{\boldsymbol{\delta}} \leftarrow$ (`alphas` $\odot$ `self`.deltas) $\gg$ `self`.bitWidth
    return $\boldsymbol{x}_{i-1} + \hat{\boldsymbol{\delta}}$
function TRAIN($\boldsymbol{x}_{i-1}$, $\boldsymbol{x}_i$, $\texttt{err}_i$)
    `gradients` $\leftarrow -\operatorname{sign}(\texttt{err}_i) \odot$ `self`.deltas
    `self`.accumulators $\leftarrow$ `self`.accumulators $-$ `gradients`
    `self`.deltas $\leftarrow \boldsymbol{x}_i - \boldsymbol{x}_{i-1}$

To predict, the forecaster first derives the coefficient α for each column based on the accumulator. By right shifting the accumulator $\log 2(\eta)$ bits, the forecaster obtains a learning rate of $2^{-\log 2(\eta)} = \eta$. It then estimates the next deltas as the elementwise product (denoted $\odot$) of these coefficients and the previous deltas. It predicts the next sample to be the previous sample plus these estimated deltas.

Because all values involved are integers, the multiplication is done using twice the bitwidth w of the data type—e.g., using 16 bits for 8-bit data. The product is then right shifted by an amount equal to the bit width. This has the effect of performing a fixed-point multiplication with step size equal to 2^{-w}.

The forecaster trains by performing a gradient update on the L_1 loss between the true and predicted samples. This can be done independently for each column j using the loss function:

$$\mathcal{L}(x_i, \hat{x}_i) = |x_i - \hat{x}_i| = |x_i - (x_{i-1} + \frac{\alpha}{2^w} \cdot \delta_{i-1})| \tag{2.4}$$

$$= |\delta_i - \frac{\alpha}{2^w} \cdot \delta_{i-1}| \tag{2.5}$$

where $x_i \triangleq \boldsymbol{x}_{ij}$ and $\delta_i = \boldsymbol{\delta}_{ij}$, the previous delta. For coefficient α, the gradient is

therefore:

$$\frac{\partial}{\partial \alpha}|\delta_i - \frac{\alpha}{2^w} \cdot \delta_{i-1}| = \begin{cases} -2^{-w}\delta_{i-1} & x_i > \hat{x}_i \\ 2^{-w}\delta_{i-1} & x_i \leq \hat{x}_i \end{cases} \tag{2.6}$$

$$= -\operatorname{sign}(\varepsilon) \cdot 2^{-w}\delta_{i-1} \tag{2.7}$$

$$\propto -\operatorname{sign}(\varepsilon) \cdot \delta_{i-1} \tag{2.8}$$

where we define $\varepsilon \triangleq x_i - \hat{\boldsymbol{x}}_i$ and ignore the 2^{-w} as a constant that can be absorbed into the learning rate. In all experiments reported here, we set the learning rate to $\frac{1}{2}$. This value is unlikely to be ideal for any particular dataset, but preliminary experiments showed that it consistently worked reasonably well.

In practice, FIRE differs from the above pseudocode in three ways. First, instead of computing the coefficient for each sample, we compute it once at the start of each block. Second, instead of performing a gradient update after each sample, we average the gradients of all samples in each block and then perform one update. Finally, we only compute a gradient for every other sample, since this has little or no effect on the accuracy and slightly improves speed.

2.4.3 Bit Packing

An illustration of SPRINTZ's bit packing is given in Figure 2-1. The prediction errors from delta coding or FIRE are zigzag encoded [75] and then the minimum number of bits required is computed for each column. Zigzag encoding is an invertible transform that interleaves positive and negative integers such that each positive integer is represented by twice its absolute value and each negative integer is represented by twice its absolute value minus one. This makes all values nonnegative and maps integers farther from zero to larger numbers.

Given the zigzag-encoded errors, the number of bits w' required in each column can be computed as the bitwidth minus the fewest leading zeros in any of that column's errors. E.g., in Figure 2-1a, the first column's largest encoded value is 16, represented as `00010000`, which has three leading zeros. This means that we require $w' = 8 - 3 = 5$

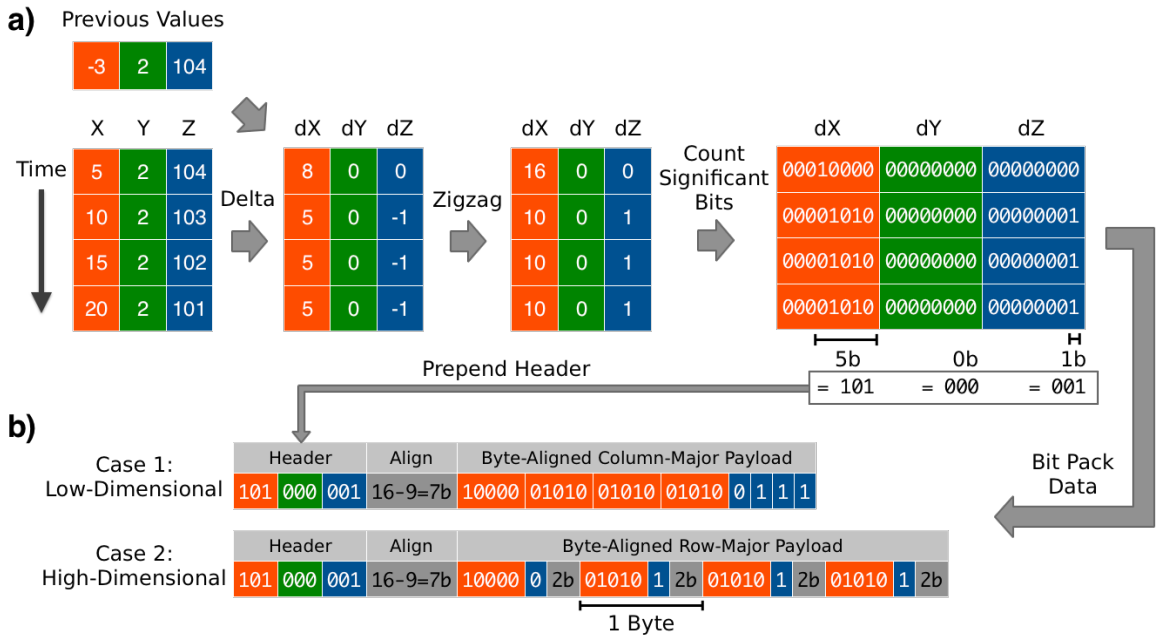

Figure 2-1: Overview of Sprintz using a delta coding predictor. ***a)*** **Delta coding of each column, followed by zigzag encoding of resulting errors. The maximum number of significant bits is computed for each column.** ***b)*** **These numbers of bits are stored in a header, and the original data is stored as a byte-aligned payload, with leading zeros removed. When there are few columns, each column's data is stored contiguously. When there are many columns, each row is stored contiguously, possibly with padding to ensure alignment on a byte boundary.**

bits to store the values in this column. One can find this value by OR-ing all the values in a column together and then using a built-in function such as GCC's `__builtin_clz` to compute the number of leading zeros in a single assembly instruction (c.f. [118]). This optimization motivates our use of zigzag encoding to make all values nonnegative.

Once the number of bits w' required for each column is known, the zigzag-encoded errors can be bit packed. First, Sprintz writes out a header consisting of D unsigned integers, one for each column, storing the bitwidths. Each integer is stored in $\log 2(w)$ bits, where w is the bitwidth of the data. Since there are $w + 1$ possible values of w' (including 0), width $w - 1$ is treated as a width of w by both the encoder and decoder. E.g., 8-bit data that could only be compressed to seven bits is both stored and decoded with a bitwidth of eight.

After writing the headers, Sprintz takes the appropriate number of low bits from each element and packs them into the payload. When there are few columns, all the bits for a given column are stored contiguously (i.e., column-major order). When

there are many columns, the bits for each sample are stored contiguously (i.e., row-major order). In the latter case, up to seven bits of padding are added at the end of each row so that all rows begin on a byte boundary. This means that the data for each column begins at a fixed bit offset within each row, facilitating vectorization of the decompressor. The threshold for choosing between the two formats is a sample width of 32 bits.

The reason for this threshold is as follows. Because the block begins in row-major order and we seek to reconstruct it the same way, the row-major bit packing case is more natural. For small numbers of columns, however, the row padding can significantly reduce the compression ratio. Indeed, for univariate 8-bit data, it makes compression ratios greater than one impossible. This gives rise to the column-major case; when using a block size of eight samples and column-major order, each column's data always falls on a byte boundary without any padding. The downside of this approach is that both encoder and decoder must transpose the block. However, for up to four 8-bit columns or two 16-bit columns, this can be done quickly using SIMD shuffling instructions.[1] This gives rise to the cutoff of 32-bit sample width for choosing between the formats.

As a minor bit packing optimization, one can store the headers for two or more blocks contiguously, so that there is one group of headers followed by one group of payloads. This allows many headers to share one set of padding bits between the headers and payload. Grouping headers does not require buffering more than one block of raw input, but it does require buffering the appropriate number of blocks of compressed output. In addition to slightly improving the compression ratio, it also enables more headers to be unpacked with a given number of vector instructions in the decompressor. Microbenchmarks show up to 10% improvement in decompression speed as the number of blocks in a group grows towards eight. However, we use groups of only two in all reported experiments to ensure that our results tend towards pessimism and are applicable under even the most extreme buffer size constraints.

[1]For recent processors with AVX-512 instructions, one could double these column counts, but we refrain from assuming that these instructions will be available.

2.4.4 Entropy Coding

We entropy code the bit packed representation of each block using Huff0, an off-the-shelf Huffman coder [42]. This encoder treats individual bytes as symbols, regardless of the bitwidth of the original data. We use Huffman coding instead of Finite-State Entropy [42] or an arithmetic coding scheme since they are slower, and we never observed a meaningful increase in compression ratio.

The benefit of adding Huffman coding to bit packing stems from bit packing's inability to optimally encode individual bytes. For a given packed bitwidth w, bit packing models its input as being uniformly distributed over an interval of size 2^w. Appropriately setting w allows it to exploit the similar variances of nearby values, but does not optimally encode individual values (unless they truly are uniformly distributed within the interval). Huffman coding is complementary in that it fails to capture relationships between nearby bytes but optimally encodes individual bytes.

We Huffman code after bit packing, instead of before, for two reasons. First, doing so is faster. This is because the bit packed block is usually shorter than the original data, so less data is fed to the Huffman coding routines. These routines are slower than the rest of Sprintz, so minimizing their input size is beneficial. Second, this approach increases compression. Bit packed bytes benefit from Huffman coding, but Huffman coded bytes do not benefit from bit packing, since they seldom contain large numbers of leading zeros. This absence of leading zeros is unsurprising since Huffman codes are not byte-aligned and use ones and zeros in nearly equal measure.

2.4.5 Vectorization

Much of Sprintz's speed comes from vectorization. For headers, the fixed bitwidths for each field and fixed number of fields allows for packing and unpacking with a mix of vectorized byte shuffles, shifts, and masks. For payloads, delta (de)coding, zigzag (de)coding, and Fire all operate on each column independently, and so naturally vectorize. Because the packed data for all rows is the same length and aligned to a byte boundary (in the high-dimensional case), the decoder can compute the bit offset

of each column's data one time and then use this information repeatedly to unpack each row. In the low-dimensional case, all packed data fits in a single vector register which can be shuffled and masked appropriately for each possible number of columns. This is possible since there are at most four columns in this case. On an `x86` machine, bit packing and unpacking can be accelerated with the `pext` and `pdep` instructions, respectively.

2.5 Experimental Results

To assess SPRINTZ's effectiveness, we compared it to a number of state-of-the-art compression algorithms on a large set of publicly available datasets. All of our code and raw results are publicly available on the SPRINTZ website.[2] All experiments use a single thread on a 2013 Macbook Pro with a 2.6GHz Intel Core i7-4960HQ processor.

All reported timings and throughputs are the best of ten runs. We use the best, rather than average, since this is 1) a best practice in microbenchmarking [117], and 2) desirable in the presence of the non-random, purely additive noise characteristic of microbenchmarks. The best values are nearly always within 10% of the averages.

2.5.1 Datasets

- **UCR** [103] — The UCR Time Series Archive is a repository of 85 univariate time series datasets from various domains, commonly used for benchmarking time series algorithms. Because each dataset consists of many (often short) time series, we concatenate all the time series from each dataset to form a single longer time series. This is to allow dictionary-based methods to share information across time series, instead of compressing each in isolation. To mitigate artificial jumps in value from the end of one time series to the start of another, we linearly interpolate five samples between each pair.
- **PAMAP** [151] — The PAMAP dataset consists of inertial motion and heart rate

[2] https://smarturl.it/sprintz

data from wearable sensors on subjects performing everyday actions. It has 31 variables, most of which are accelerometer and gyroscope readings.

- **MSRC-12** [66] — The MSRC-12 dataset consists of 80 variables of (x, y, z, depth) positions of human joints captured by a Microsoft Kinect. The subjects performed various gestures one might perform when interacting with a video game.
- **UCI Gas** [65] — This dataset consists of 18 columns of gas concentration readings and ground truth concentrations during a chemical experiment.
- **AMPDs** [128] — The Almanac of Minutely Power Datasets describes electricity, water, and natural gas consumption recorded once per minute for two years at a single home.

For datasets stored as delimited files, we first parsed the data into a contiguous, numeric array and then dumped the bytes as a binary file. Before obtaining any timing results, we first loaded each dataset into main memory. Because the datasets are provided as floating point values (despite most reflecting analog-to-digital converter output that was originally integer-valued), we quantized them into integers before operating on them. We did so by linearly rescaling them such that the largest and smallest values corresponded to the largest and smallest values representable with the number of bits tested—e.g., 0 and 255 for 8 bits—and then applying the floor function. Note that this is the worst case scenario for our method since it maximizes the number of bits required to represent the data.

For multivariate datasets, we allowed all methods but our own to operate on the data one variable at a time; i.e., instead of interleaving values for every variable, we stored all values for each variable contiguously. This corresponds to allowing them an unlimited buffer size in which to store incoming data before compressing it. We allow these ideal conditions in order to eliminate buffer size as a lurking variable and ensure that our results for methods developed by others err towards optimism.

2.5.2 Comparison Algorithms

- **SIMD-BP128** [118] — The fastest known method of compressing integers.

- **FastPFor** [118] — An algorithm similar to SIMD-BP128, but with better compression ratios.
- **Simple8b** [19] — An integer compression algorithm used by the popular time series database InfluxDB [29].
- **Snappy** [77] — A general-purpose compression algorithm developed by Google and used by InfluxDB, KairosDB [84], OpenTSDB [162], RocksDB [167], the Hadoop Distributed File System [160], and numerous other projects.
- **Zstd** [44] — Facebook's state-of-the-art general-purpose compression algorithm. It is based on LZ77 and entropy codes using a mix of Huffman coding and Finite State Entropy (FSE) [42]. It is available in RocksDB [167].
- **LZ4** [43] — A widely used general-purpose compression algorithm optimized for speed and based on LZ77. It is used by RocksDB and ChronicleDB [157].
- **Zlib** [55] — A popular implementation of the DEFLATE [54] dictionary coder, which also underlies gzip [69].

For Zlib and Zstd, we use a compression level of 9 unless stated otherwise. This level heavily prioritizes compression ratio at the expense of increased compression time. We use it to improve the results for these methods in experiments in which compression time is not penalized.

We also assess three variations of Sprintz, corresponding to different speed versus ratio tradeoffs. In ascending order of speed, they include:

1. `SprintzFIRE+Huf`. The full algorithm described in Section 4.4.
2. `SprintzFIRE`. Like `SprintzFIRE+Huf`, but without Huffman coding.
3. `SprintzDelta`. Like `SprintzFIRE`, but with delta coding instead of Fire as the forecaster.

2.5.3 Compression Ratio

In order to rigorously assess the compression performance of both Sprintz and existing algorithms, it is desirable to evaluate each on a large corpus of time series from

heterogeneous domains. Consequently, we use the UCR Time Series Archive [103]. This corpus contains dozens of datasets and is almost universally used for evaluating time series classification and clustering algorithms in the data mining community.

The distributions of compression ratios on these datasets for the above algorithms are shown in in Figure 2-2. Sprintz exhibits consistently strong performance across almost all datasets. High-speed codecs such as Snappy, LZ4, and the integer codecs (FastPFor, SIMDBP128, Simple8B) hardly compress most datasets at all.

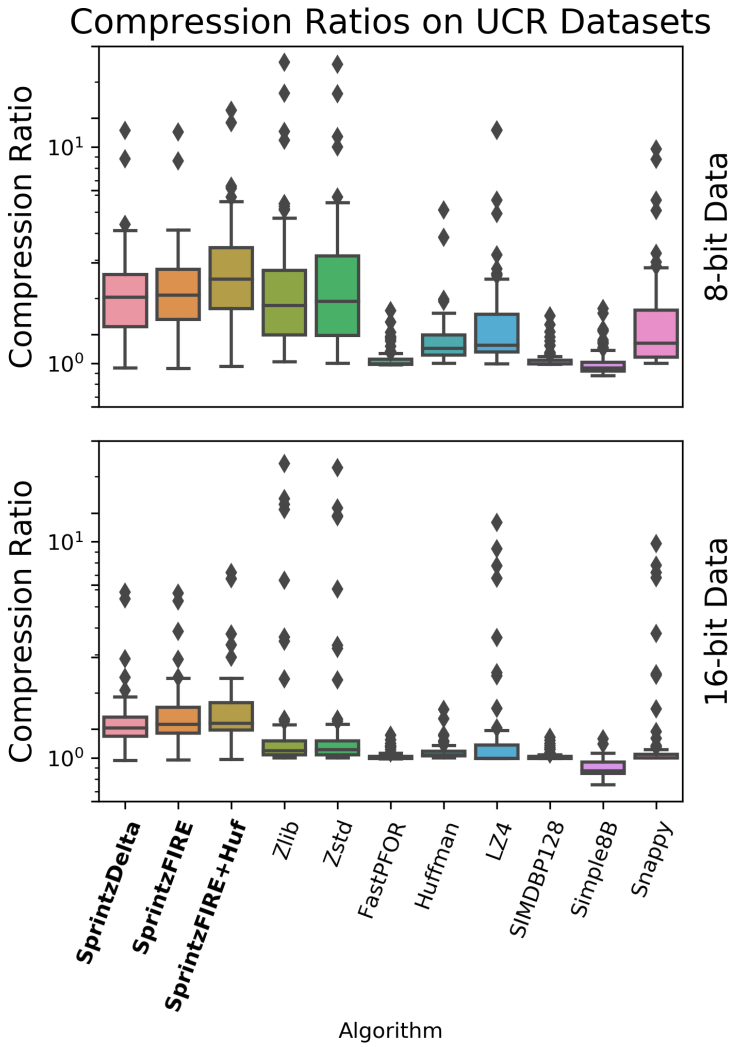

Figure 2-2: Boxplots of compression performance of different algorithms on the UCR Time Series Archive. Each boxplot captures the distribution of one algorithm across all 85 datasets.

Perhaps counter-intuitively, 8-bit data tends to yield higher compression ratios than 16-bit data. This is a product of the fact that the number of bits that are "pre-

dictable" is roughly constant. I.e., suppose that an algorithm can correctly predict the four most significant bits of a given value; this enables a 2:1 compression ratio in the 8-bit case, but only a 16:12 = 4:3 ratio in the 16-bit case.

To assess SPRINTZ's performance statistically, we use a Nemenyi test [141] as recommended by Demsar [52]. This test compares the mean rank of each algorithm across all datasets, where the highest-ratio algorithm is given rank 1, the second-highest rank 2, and so on. The intuition for why this test is desirable is that it not only accounts for multiple hypothesis testing in making pairwise comparisons, but also prevents a small number of large or highly compressible datasets from dominating the results.

The results of the Nemenyi test are shown in the Critical Difference Diagrams [52] in Figure 2-3. These diagrams show the mean rank of each algorithm on the x-axis and join methods that are not statistically significantly different with a horizontal line. SPRINTZ on high-compression settings is significantly better than any existing algorithm. On lower settings, it is still as effective as the best current methods (Zlib and Zstd).

In addition to this overall comparison, it is important to assess whether FIRE improves performance compared to delta coding. Since this is a single hypothesis with matched pairs, we assess it using a Wilcoxon signed rank test. This yields p-values of .0094 in the 8-bit case and 4.09e-12 in the 16-bit case. FIRE obtains better compression on 51 of 85 datasets using 8 bits and 74 of 85 using 16. These results suggest that FIRE is helpful on 8-bit data but even more helpful on 16-bit data.

To understand why 16-bit data benefits more, we examined datasets where FIRE gives differing benefits in the two cases. The difference most commonly occurs when the data is highly compressible with just delta coding. With 8 bits and $\sim$4$\times$ compression, the forecaster's task is effectively to guess whether the next delta is -1, 0, or 1 given a current delta drawn from this same set. The Bayes error rate is high for this problem, and FIRE's attempt to learn adds variance compared to the delta coding approach of always predicting zero. In contrast, with 16 bits, the deltas span many more values and retain continuity that FIRE can exploit.

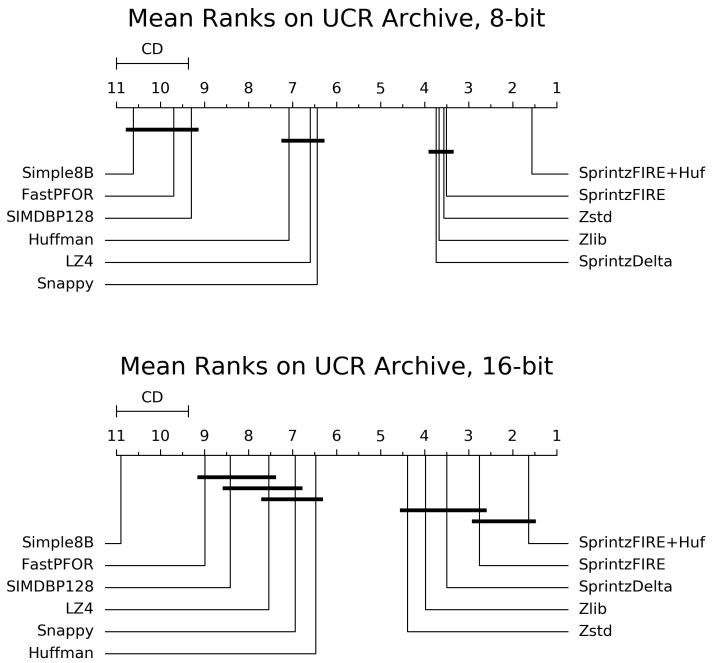

Figure 2-3: Compression performance of different algorithms on the UCR Time Series Archive. The x-axis is the mean rank of each method, where rank 1 on a given dataset has the highest ratio. Methods joined with a horizontal black line are not statistically significantly different.

2.5.4 Decompression Speed

To systematically assess the speed of Sprintz, we ran it on time series with varying numbers of columns and varying levels of compressibility. Because real datasets have a fixed and limited number of columns, we ran this experiment on synthetic data. Specifically, we generated a synthetic dataset of 100 million random values uniformly distributed across the full range of those possible for the given bitwidth. This data is incompressible and thus provides a worst-case estimate of Sprintz's speed (though in practice, we find that the speed is largely consistent across levels of compressibility).

We compressed the data with Sprintz set to treat it as if it had 1 through 80 columns. Numbers that do not evenly divide the data size result in Sprintz `memcpy`-ing the trailing bytes.

While using this synthetic data cannot tell us anything about Sprintz's compression ratio, it is suitable for throughput measurement. This is because both Sprintz's sequence of executed instructions and memory access patterns are effectively indepen-

dent of the data distribution—Sprintz's core loop has no conditional branches and Sprintz's memory accesses are always sequential. Moreover, it exhibits throughputs on real data matching or slightly exceeding the numbers in this subsection for the corresponding number of columns (c.f. Figure 2-7).

As shown in Figure 2-4, Sprintz becomes faster as the number of columns increases and as the number of columns approaches multiples of 32 for 8-bit data or 16 for 16-bit data. These values correspond to the 256-bit width of a SIMD register on the machine used for testing. There is small but consistent overhead associated with using Fire over delta coding, but both approaches are extremely fast. Without Huffman coding, Sprintz decompresses at multiple GB/s once rows exceed ∼16B. With Huffman coding, the other components of Sprintz are no longer the bottleneck and Sprintz consistently decompresses at over 500MB/s. Note that we omit comparison to other algorithms in this section since their speed varies with compressibility, not number of columns; see Section 2.5.7 for a direct comparison. Further note that the speed's dependence on number of columns is not an artifact of more columns yielding larger blocks of data. The limiting factor is serial dependence between decoding one sample and predicting the next one; this is accelerated by having wider samples that fill a vector register, but not by having longer blocks. There is a slight downward trend in the 16-bit data as the number of dimensions becomes large; this is a result of poor read locality in our prototype implementation.

2.5.5 Compression Speed

It is important that Sprintz's compression speed be fast enough to keep up with the rate of data ingestion. We measured Sprintz's compression speed using the same methodology as in the previous subsection. As shown in Figure 2-5, Sprintz compresses 8-bit data at over 200MB/s on the highest-ratio setting and 600MB/s on the fastest setting. These numbers are roughly 50% larger on 16-bit data. We refrained from vectorizing this prototype implementation because 1) 200MB/s is already fast enough to run in real time even if every thread were fed data from its own gigabit

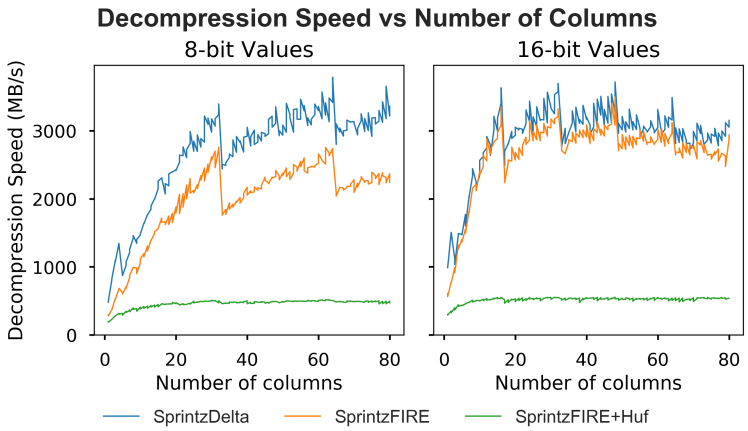

Figure 2-4: Sprintz becomes faster as the number of columns increases and as the width of each sample approaches multiples of 32B (on a machine with 32B vector registers).

network connection, and 2) low-power devices often lack vector instructions, so the measured speeds are more indicative of the rate at which these devices could compress (if scaled to the appropriate clock frequency). We again omit comparison to other compressors for the same reason as in the previous section. The dips after four columns in 8-bit data and two columns in 16-bit data correspond to the switch from column-major bit packing to row-major bit packing.

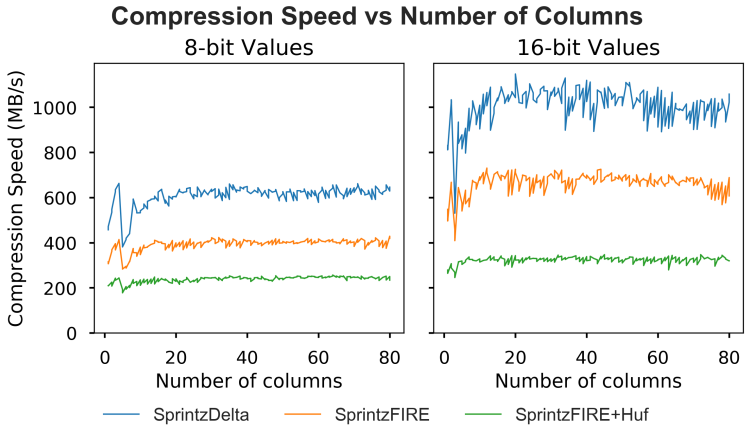

Figure 2-5: Sprintz compresses at hundreds of MB/s even in the slowest case: its highest-ratio setting with incompressible 8-bit data. On lower settings with 16-bit data, it can exceed 1GB/s.

2.5.6 FIRE Speed

To characterize the speed of FIRE , we repeated the above throughput experiments with both it and two other predictors commonly seen in the literature: delta and double delta coding. As shown in Figure 2-6, FIRE can encode at up to 5GB/s and decode at up to 6GB/s. This is nearly the same speed as the competing methods and close to the 7.5GB/s speed of `memcpy` on the tested machine. Note that "encode" and "decode" here mean converting raw samples to errors and reconstructing samples from sequences of errors, respectively. These operations do not change the data size, but are the subroutines run in the SPRINTZ compressor and decompressor. The reason that there is less discrepancy between delta and FIRE encoding in isolation versus when embedded in SPRINTZ compression (Figure 2-5) is that, in this experiment, the implementations are vectorized.

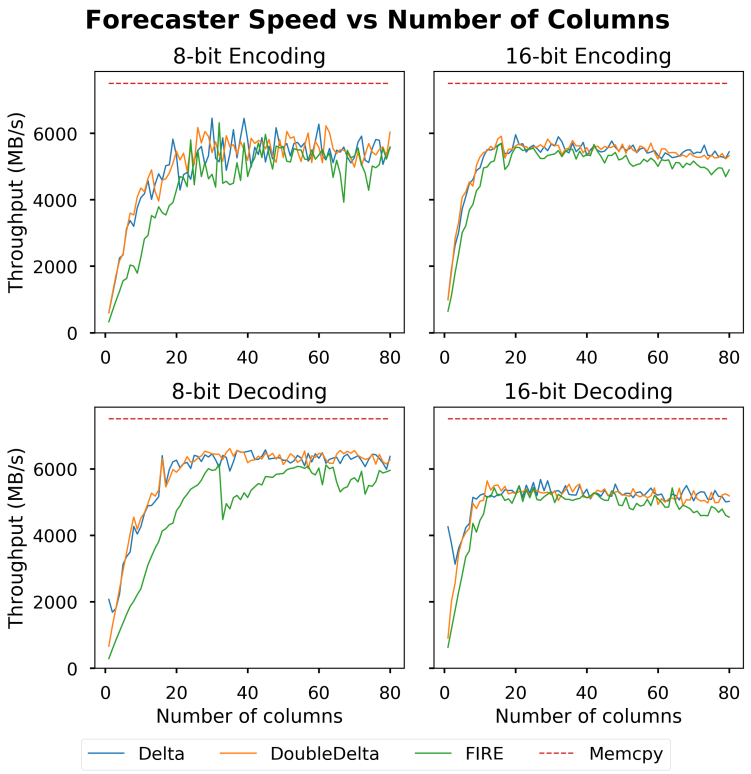

Figure 2-6: Fire is nearly as fast as delta and double delta coding. For a moderate number of columns, it runs at 5-6GB/s on a machine with 7.5GB/s `memcpy` speed.

2.5.7 When to Use Sprintz

The above experiments provide a characterization of Sprintz's speed and a statistically meaningful assessment of its compression ratio in general. However, because one often wants to obtain the best results on a particular type of data, it is helpful to know when Sprintz is likely to work well or poorly.

Regarding speed, Sprintz is most useful when there are many variables to be compressed. We have found that the speed is largely insensitive to compression ratio, so the results in Sections 2.5.4 and 2.5.5 offer a good estimate of the speed one could expect on similar hardware. The exception to this is if the data contains long runs of constants (or constant slopes if using Fire). In this case, the decompression speed approaches the speed of `memcpy` for `SprintzDelta` or the speed of Fire for `SprintzFIRE` and `SprintzFIRE+Huf`.

Regarding compression ratio, the dominant requirement is that the data must have relatively strong correlations between consecutive values. This occurs when the sampling rate is fast relative to the time scale over which the measured quantity changes—the typical case when one seeks reasonably high-quality measurements. When these correlations are absent, predictive filtering (with only a two-component filter) has little value. Indeed, it can even be counterproductive. Consider the case of data that has an isolated nonzero value every few samples—e.g., the sequence $\{0, 1, 0, 0\}$. When delta coded, this yields $\{0, 1, -1, 0\}$, which requires an extra bit for Sprintz bit packing. In general, Sprintz has to pay the cost of abrupt changes twice—once when they happen, and once when they "revert" to the previous level. This scenario may occur when the measured quantity experiences transient events; e.g., in the AMPD water dataset, a house may consume no water most of the time, but experience a temporary spike when a toilet is flushed.

Another specific case in which Sprintz is less useful is when the data distribution tends to switch between discrete states. For example, in electricity consumption data, an appliance tends to use little or no electricity when it is off and a relatively constant amount when it is on. Switches between these states are expensive for Sprintz, and

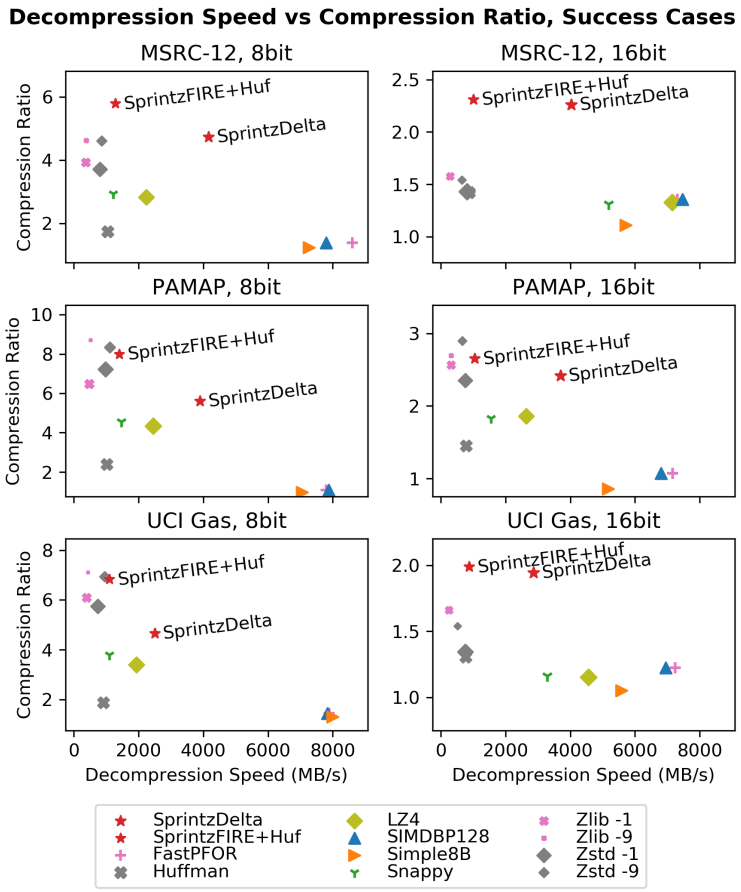

Figure 2-7: Sprintz achieves excellent compression ratios and speed on relatively slow-changing time series with many variables.

predictive filtering offers little benefit on sequences of samples that are already almost constant. Sprintz can still achieve reasonably good compression in this situation, but dictionary-based compressors will likely perform better. This is because they suffer no penalty from state changes, and runs of constants are their best-case input in terms of both ratio and speed. Their ratio benefits because they can often run-length encode the number of repeated values, and their speed benefits because they can decode runs at memory speed by `memcpy`-ing the repeated values.

As an illustration of when Sprintz is and is not preferable, we ran it and the comparison algorithms on several real-world datasets with differing characteristics. In Figure 2-7, we use the MSRC-12, PAMAP, and UCI Gas datasets. These datasets contain time series that change slowly relative to the sampling rate and have 80, 31, and 18 variables, respectively. Sprintz achieves an excellent ratio-speed tradeoff on all three datasets, and the highest compression of any method *even on its lowest-*

compression setting on the MSRC-12 dataset.

In contrast, SPRINTZ performs poorly on the AMPD Gas and AMPD Water datasets (Figure 2-8). These datasets chronicle the natural gas and water consumption of a house over a year, and often switch between discrete states and/or have isolated nonzero values. They also have only three and two variables, respectively. SPRINTZ achieves more than 10× compression, but dictionary-based methods such as Zstd and LZ4 achieve even greater compression, while also decompressing faster.

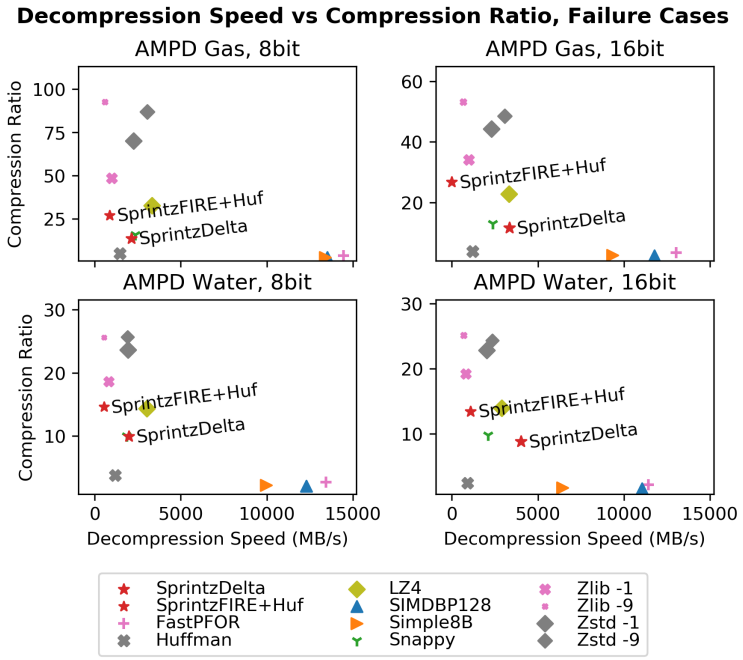

Figure 2-8: Sprintz is less effective than other methods when the time series has large, abrupt changes and few variables.

2.5.8 Generalizing to Floats

While floating point values are not the focus of this work, it is possible to apply SPRINTZ to floats by first quantizing the floating point data. The downside of doing this is that, because floating point formats have non-uniform resolution along the real line, such quantization is lossy. To assess the degree of loss, we carried out an experiment to measure the error induced when quantizing real data. Note that this experiment does not assess whether SPRINTZ is the *best* means of compressing floats—it merely suggests that using integer compressors like SPRINTZ as lossy floating point compressors is reasonable and could be a fruitful avenue for future work.

We assessed the magnitude of typical quantization errors by quantizing the UCR time series datasets. Specifically, we linearly offset and rescaled the time series in each dataset such that the minimum and maximum values in any time series correspond to $(0, 255)$ for 8-bit quantization or $(0, 65535)$ for 16-bit quantization. We then obtained the quantized data by applying the floor function to this linearly transformed data.

To measure the error this introduced, we then inverted the linear transformation and computed the mean squared error between the original and the "reconstructed" data. The resulting error values for each dataset, normalized by the dataset's variance, are shown in Figure 2-9. These normalized values can be understood as signal-to-noise ratio measurements, where the noise is the quantization error. As the figure illustrates, the quantization error is orders of magnitude smaller than the variance for nearly all datasets, and never worse than $10\times$ smaller, even for 8-bit quantization. This of course does not indicate that all time series can be safely quantized. Two

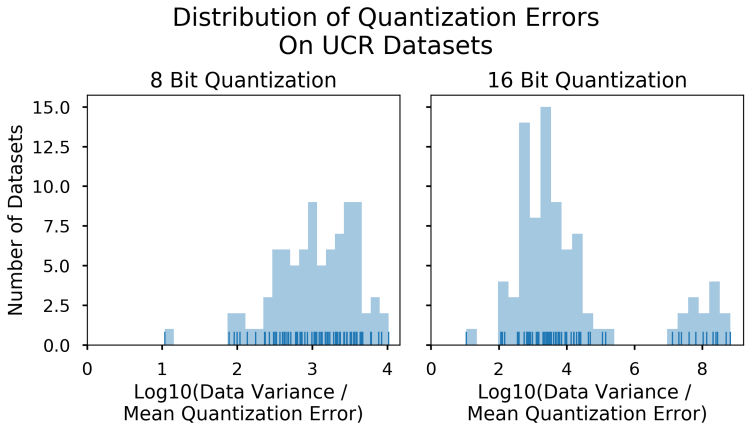

Figure 2-9: Quantizing floating point time series to integers introduces error that is orders of magnitude smaller than the variance of the data. Even with eight bits, quantization introduces less than 1% error on 82 of 85 datasets.

counterexamples of which we are aware are 1) timestamps where microsecond or nanosecond resolution matters, and 2) GPS coordinates, where small decimal places may correspond to many meters. However, the above results suggest that quantization is a suitable means of applying SPRINTZ to floating point data in many applications. This is bolstered by previous work showing that quantization even to a mere six bits [150] rarely harms classification accuracy, and quantizing to two bits is enough to support many data mining tasks [122, 107, 159, 158].

2.6 Summary

We introduced Sprintz, a compression algorithm for multivariate integer time series that achieves state-of-the-art compression ratios across a large number of publicly available datasets. It also attains speeds of up to 3GB/s in a single thread and predictable performance as a function of the number of variables being compressed. Moreover, it only needs to buffer eight samples at a time, enabling low latency for continuously arriving data. Finally, Sprintz has extremely low memory requirements, making it feasible to run even on resource-constrained devices.

As part of evaluating Sprintz, we also conducted what is, to the best of our knowledge, the largest empirical investigation of time series compression that has been reported. To both ensure reproducibility of our work and facilitate future research in this area, we make available all of our experiments as a public benchmark.

In future work, we hope to characterize the relationship between compression and power savings, both for Sprintz and for other methods. The savings are upper bounded by the compression ratio in the limit of data transmission consuming all power, but real-world systems have various overheads that cause significant deviation from this idealized model.

Chapter 3

Fast Approximate Scalar Reductions

3.1 Introduction

Similarity search is among the most common tasks in machine learning, data mining, and information retrieval. Often, such searches take the form of comparing a received *query* vector to a large set of *database* vectors in order to discover approximate matches. For example, when an image search engine seeks to display "related images" for a given image the user clicks on, an embedding of the clicked image is compared to a database of embeddings of many other images to identify suitable candidates. The similarity search process typically decomposes into two steps:

1. A coarse retrieval step, in which blocks of possibly similar database vectors are identified as meriting more detailed comparison.
2. A scan step, in which each vector within each block is compared to the query.

Because the set of vectors in the database may change, and queries may arrive at different rates, it is useful to decompose the overall computation cost of running a vector similarity search system as

$$\texttt{Cost}_{\text{time}} = \texttt{Cost}_{\text{read}} + \texttt{Cost}_{\text{write}} \tag{3.1}$$

where $\texttt{Cost}_{\texttt{read}}$ is the time cost of operations that read the data (such as performing a search), and $\texttt{Cost}_{\texttt{write}}$ is the time cost of creating, updating, or deleting data.

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 [131, 25, 70, 185, 132]. As a result, vector quantization methods are used in production at companies such as Google [79, 163] and Facebook [4].

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 Martinez et al. [131], for example, requires up to *four ms* 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. 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.

3.1.1 Problem Statement

Let $\boldsymbol{q} \in \mathbb{R}^J$ be a *query* vector and let $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}, \boldsymbol{x}_i \in \mathbb{R}^J$ 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:

$$d(\boldsymbol{q}, \boldsymbol{x}) = f\Big(\sum_{j=1}^{J} \delta(q_j, x_j)\Big) \tag{3.2}$$

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 otherwise noted.

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 $\mathcal{L}$,

$$\mathcal{L} = E_{\boldsymbol{q},\boldsymbol{x}}[(d(\boldsymbol{q}, \boldsymbol{x}) - \hat{d}(g(\boldsymbol{q}), h(\boldsymbol{x})))^2] \tag{3.3}$$

the computation time T,

$$T = T_g + T_h + T_d \tag{3.4}$$

is minimized, where T_g is the time to encode received queries $\boldsymbol{q}$ using g,[1] T_h is the time to encode the database $\mathcal{X}$ using h, and T_d is 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

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

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.

3.1.2 Assumptions

Like other vector quantization work [98, 131, 25, 70, 185], 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 $\mathcal{X}$ and $\boldsymbol{q}$. Following this offline phase, there is an online phase wherein we are given database vectors $\boldsymbol{x}$ that must be encoded and query vectors $\boldsymbol{q}$ 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 $\boldsymbol{X}$ 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 $\boldsymbol{x}$ vectors are all added at once before any queries are received [98, 131, 25, 70, 185], and therefore that encoding speed is less of a concern.

In practice, one might require the distances between $\boldsymbol{q}$ 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 [23, 26] or Locality-Sensitive Hashing hash tables [48, 15], 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.

Finally, we assume that both $\mathcal{X}$ and $\boldsymbol{q}$ are relatively dense. Bolt can be applied to sparse data but does not leverage the sparsity. Consequently, it is advisable to embed sparse vectors into a dense, lower-dimensional space before using Bolt.

3.2 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 recent surveys [176, 175] for further detail.

Many existing approaches in the computer vision and information retrieval literature fall into one of two categories: binary embedding and vector quantization [175]. 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 eight 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 [70, 175]. He et al. [70] showed that the popular binary embedding technique of Gong et al. [74] is a more constrained version of their vector quantization algorithm, and that the objective function of another state-of-the art binary embedding method [109] 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) [98], 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 [143], Optimized Product Quantization [70], Generalized Residual Vector Quantization [123], Additive Quantization [24], Composite Quantization [185], Optimized Tree Quantization [25], Stacked Quantizers [132], and Local Search

Quantization [131]. 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. We discuss PQ in more detail in the following section.

An interesting hybrid of binary embedding and vector quantization is the recent Polysemous Coding of Douze et al. [56]. 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 Fabien et al. [16], 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 not practical when the data is rapidly changing or when using an indexing structure, which would split the data into 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 real-valued 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 guarantees regarding the level of reduction attainable for a given level of distortion in pairwise distances [47, 12, 113], as well as strong empirical results [99, 173]. 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 [179], the Fast Fourier Transform [181], the Discrete Cosine Transform [137], or the Fast Hadamard Transform [15, 32], possibly with learned elementwise weights, instead of performing a matrix multiply. These methods have strong theoretical grounding [32] and sometimes outperform non-structured matrices [179]. They are orthogonal to our work in that they bypass the need for a dense matrix entirely, while our approach can accelerate operations for which a dense matrix is used.

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 [87]; 2) reusing weights [39]; 3) pruning weights in a model after training [130, 82]; and 4) training a small model to approximate the outputs of a larger model [166]. This has been a subject of intense research for neural networks in recent years, and we do not believe that our approach would 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.

3.3 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 $\hat{d}(g(\boldsymbol{q}), h(\boldsymbol{x})) \approx d(\boldsymbol{q}, \boldsymbol{x})$ for some "true" distance function d. To explain how we do this, we first begin with a review of Product Quantization [98], and then describe how our method differs.

3.3.1 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

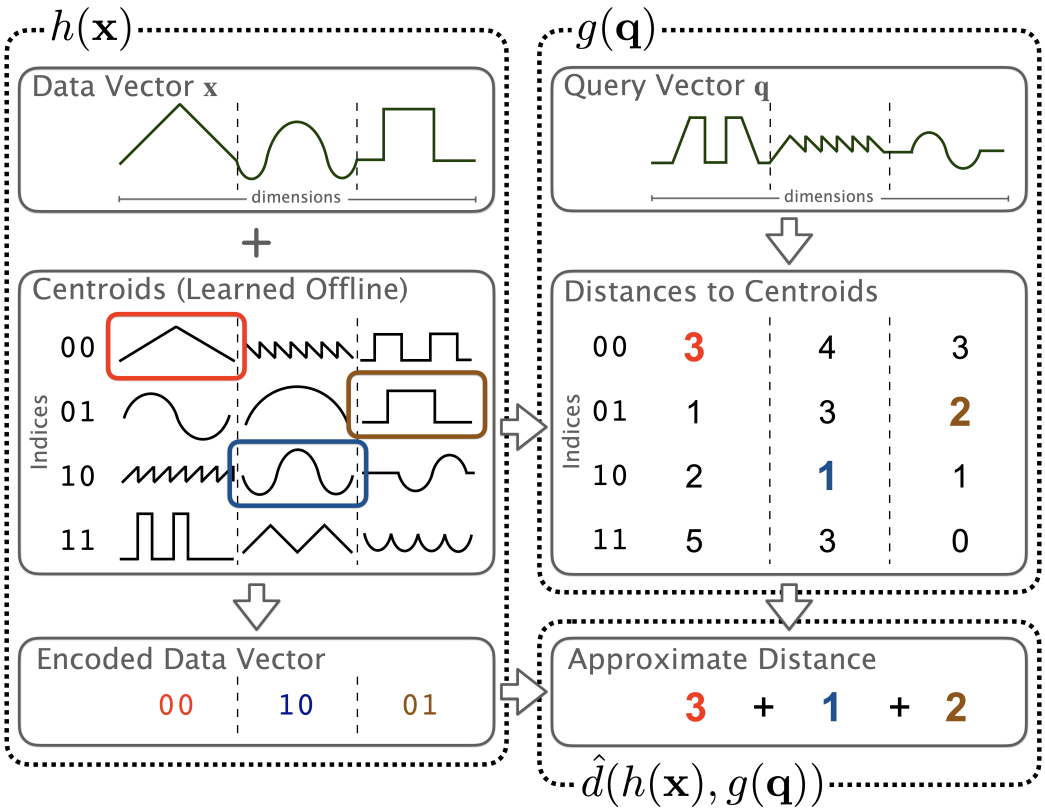

Figure 3-1: Product Quantization. The $h(\cdot)$ function returns the index of the most similar centroid to the data vector x in each subspace. The $g(\cdot)$ function computes a lookup table of distances between the query vector q and each centroid in each subspace. The aggregation function $\hat{d}(\cdot, \cdot)$ sums the table entries corresponding to each index.

as the distance between the query and the vector's 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 $\boldsymbol{q}$ and the chosen centroids for each corresponding subvector of $\boldsymbol{x}$. We elaborate upon each of these steps below, and depict them graphically in Figure 3-1.

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

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

where $f : \mathbb{R} \to \mathbb{R}$, $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Now, suppose one partitions the indices j into M disjoint subsets $\{p_1, \ldots, p_M\}$. Typically, each subset is a sequence of J/M consecutive indices. The argument to f can then be written as:

$$\sum_{m=1}^{M} \sum_{j \in p_m} \delta(q_j, x_j) = \sum_{m=1}^{M} \boldsymbol{\delta}\left(\boldsymbol{q}^{(m)}, \boldsymbol{x}^{(m)}\right) \tag{3.5}$$

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

$$\sum_{m=1}^{M} \boldsymbol{\delta}\left(\boldsymbol{q}^{(m)}, \boldsymbol{x}^{(m)}\right) \approx \sum_{m=1}^{M} \boldsymbol{\delta}\left(\boldsymbol{q}^{(m)}, \boldsymbol{c}_i^{(m)}\right) \tag{3.6}$$

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

$$h(\boldsymbol{x}) = [i_1; \ldots; i_M],\ i_m = \underset{i}{\operatorname{argmin}}\, d\left(\boldsymbol{c}_i^{(m)}, \boldsymbol{x}^{(m)}\right) \tag{3.7}$$

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

Using codebooks 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 $\boldsymbol{D} = g(\boldsymbol{q})$ as:

$$\boldsymbol{D}_{im} \triangleq \boldsymbol{\delta}\left(\boldsymbol{q}^{(m)}, \boldsymbol{c}_i^{(m)}\right) \tag{3.8}$$

Then we can rewrite the approximate distance on the right hand side of 3.6 as:

$$\sum_{m=1}^{M} \boldsymbol{D}_{im}, \;\; i = h(\boldsymbol{x})_m \tag{3.9}$$

In other words, the approximate distance can be reduced to a sum of precomputed distances between $\boldsymbol{q}^{(m)}$ and the codebook vectors $\boldsymbol{c}_i^{(m)}$ used to approximate $\boldsymbol{x}$. Each of the M columns of $\boldsymbol{D}$ represents the distances between $\boldsymbol{q}^{(m)}$ and the K centroids in codebook $\mathcal{C}_M$. Computation of the distance proceeds by iterating through the columns, looking up the distance in row $h(\boldsymbol{x})_m$, and adding it to a running total. By reintroducing f, one can now define:

$$\hat{d}(g(\boldsymbol{q}), h(\boldsymbol{x})) \triangleq f\Big(\sum_{m=1}^{M} \boldsymbol{D}_{im}, i = h(\boldsymbol{x})_m \Big) \tag{3.10}$$

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

The total computational cost of product quantization is $\Theta(KJ)$ to encode each $\boldsymbol{x}$, $\Theta(KJ)$ to encode each query $\boldsymbol{q}$, and $\Theta(M)$ to compute the approximate distance between an encoded $\boldsymbol{q}$ and encoded $\boldsymbol{x}$. 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 $h(\boldsymbol{x})$ can be encoded as one byte.

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

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

PQ achieves the information-theoretic lower bound on code length for a given quantization error [70]. This means that PQ encoding is optimal if $\boldsymbol{x}$ 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 [70, 143, 24, 25, 131].

In summary, PQ consists of three components:

1. Encoding each $\boldsymbol{x}$ in the database using $h(\boldsymbol{x})$. This transforms $\boldsymbol{x}$ to a list of M 8-bit integers, representing the indices of the closest centroids in the M codebooks. $h(\boldsymbol{x})$ must be recomputed when $\boldsymbol{x}$ is inserted or modified.
2. Encoding a query $\boldsymbol{q}$ when it is received using $g(\boldsymbol{q})$. This returns a $K \times M$ matrix $\boldsymbol{D}$ where the mth column is the distances to each centroid in codebook $\mathcal{C}_m$.
3. Scanning the database. Once a query is computed, the approximate distance to each $\boldsymbol{x}$ is computed using (3.10) by looking up and summing the appropriate entries from each column of $\boldsymbol{D}$.

3.3.2 Bolt

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

1. It uses much smaller codebooks than is typical for PQ.
2. It approximates the distance matrix $\boldsymbol{D}$.

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 $\boldsymbol{x}^{(m)}$ or $\boldsymbol{q}^{(m)}$ 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 $\boldsymbol{D}$, allows us to reduce the size of $\boldsymbol{D}$. This approximation is separate from approximating the overall distance—in other algorithms, the entries of $\boldsymbol{D}$ are the exact distances between each $\boldsymbol{q}^{(m)}$ and the

corresponding centroids $\mathcal{C}_m$. In Bolt, the entries of $\boldsymbol{D}$ are learned 8-bit quantizations of these exact distances.

Together, changes (1) and (2) allow hardware vectorization of the lookups in $\boldsymbol{D}$. Instead of looking up the entry in a given column of D for one $\boldsymbol{x}$ (a standard load from memory), we can leverage vector instructions to instead perform V lookups for V consecutive $h(\boldsymbol{x})$, $h(\boldsymbol{x}_i), \ldots, h(\boldsymbol{x}_{i+V})$, 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 $\boldsymbol{D}$. 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 zeros 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 learn a quantization function at training time. The basic approach is to learn the distribution of distances within a given column of $\boldsymbol{D}$ (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 $\boldsymbol{D}$ (henceforth, referred to as a *lookup table*), let Q be the distribution of query subvectors $\boldsymbol{q}^{(m)}$, X be the distribution of database

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

subvectors $\boldsymbol{x}^{(m)}$, and Y be the distribution of distances within that table. I.e.:

$$p(Y = y) \triangleq \int_{Q,X} p(\boldsymbol{q}^{(m)}, \boldsymbol{x}^{(m)}) I\{\boldsymbol{\delta}\left(\boldsymbol{q}^{(m)}, \boldsymbol{x}^{(m)}\right) = y\} \tag{3.11}$$

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

$$\beta_m(y) = \max(0, \min(255, \lfloor ay - b \rfloor)) \tag{3.12}$$

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

$$E_Y[(\hat{y} - y)^2] \tag{3.13}$$

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 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.

Our approach is 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 with 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 table differently. The b values can be table-specific because they sum to one overall bias, 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.

3.3.3 Theoretical Guarantees

We describe our main theoretical results regarding the quality of BOLT's approximations below. 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 $[b_{max}, \infty)$. See Section A for proofs and additional results that hold under different assumptions about the quantization error distributions.

Lemma 3.3.1. $b_{min} \leq y \leq b_{max} \implies |y - \hat{y}| < \Delta$.

Lemma 3.3.2. *For all* $\varepsilon > \Delta$, $p(|y - \hat{y}| > \varepsilon) <$

$$\frac{1}{\sigma_Y}\left(e^{-(b_{max}-E[Y])/\sigma_Y} + e^{-(E[Y]-b_{min})/\sigma_Y}\right)e^{-\varepsilon/\sigma_Y} \tag{3.14}$$

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

Lemma 3.3.3. $|\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}| < \|\boldsymbol{q}\| \cdot \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|$

Lemma 3.3.4. $|\|\boldsymbol{q} - \boldsymbol{x}\| - \|\boldsymbol{q} - \hat{\boldsymbol{x}}\|| < \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|$

Using these worst-case uniform bounds, it is possible to obtain tighter, probabilistic bounds using Hoeffding's inequality.

Definition 3.3.1 (Reconstruction)**.** *Let* $\boldsymbol{C}$ *be the set of codebooks used to encode* $\boldsymbol{x}$*. The vector obtained by replacing each* $\boldsymbol{x}^{(m)}$ *with its nearest centroid in codebook* $\mathcal{C}_m$ *is the reconstruction of* $\boldsymbol{x}$*, denoted* $\hat{\boldsymbol{x}}$*.*

Lemma 3.3.5. *Let* $\boldsymbol{r}^{(m)} \triangleq \boldsymbol{x}^{(m)} - \hat{\boldsymbol{x}}^{(m)}$*, and assume that the values of* $\|\boldsymbol{r}^{(m)}\|$ *are independent for all* m*. Then:*

$$p(|\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}| \geq \varepsilon) \leq 2 \exp\left(\frac{-\varepsilon^2}{2\sum_{m=1}^{M}(\|\boldsymbol{q}^{(m)}\| \cdot \|\boldsymbol{r}^{(m)}\|))^2}\right) \tag{3.15}$$

Lemma 3.3.6. *Let* $\boldsymbol{r}^{(m)} \triangleq \boldsymbol{x}^{(m)} - \hat{\boldsymbol{x}}^{(m)}$*, and assume that the values of* $\|\boldsymbol{q}^{(m)} - \boldsymbol{x}^{(m)}\|^2 - \|\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)}\|^2$ *are independent for all* m*. Then:*

$$p(|\|\boldsymbol{q} - \boldsymbol{x}\|^2 - \|\boldsymbol{q} - \hat{\boldsymbol{x}}\|^2| > \varepsilon) \leq 2 \exp\left(\frac{-\varepsilon^2}{2\sum_{m=1}^{M}\|\boldsymbol{r}^{(m)}\|^4}\right) \tag{3.16}$$

3.4 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] 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 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. 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 exact distance tables.

For all experiments, we assess Bolt and the comparison methods using the commonly employed encoding sizes of 8B, 16B, and 32B to characterize the relationships

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

between space, speed, and accuracy.

All reported timings and throughputs are the best of five 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.

3.4.1 Datasets

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

- **Sift1M** [98] — 1 million 128-dimensional SIFT [125] descriptors of images. Sift1M vectors tend to have high correlations among many dimensions, and are therefore highly compressible. 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** [132] — 1 million 128-dimensional Convnet descriptors of images. These vectors have some amount of correlation, but less than Sift1M. It has a query/train/test split matching that of Sift1M.

- **LabelMe22k** [142] — 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 existing work [131, 185] 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** [115] — 60,000 28x28-pixel greyscale images, flattened to 784-dimensional vectors. This dataset is sparse and has high correlations across dimensions. Again following existing work [131, 185], we split it the same way as the LabelMe dataset.

For all datasets, we use 10% of the training database as queries when learning Bolt's lookup table quantization. These vectors are selected uniformly at random without replacement and removed from the database during this learning phase.

3.4.2 Comparison Algorithms

Our comparison algorithms include 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 [123] or LSQ [131] are more appropriate than Bolt.[4]

Our primary baselines are Product Quantization (PQ) [98] and Optimized Product Quantization (OPQ) [70], since they offer the fastest encoding times. There are several algorithms that extend these basic approaches by adding indexing methods [102, 26], or more sophisticated training-time optimizations [78, 22, 56], 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, since 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. Note that, because the number of bits per codebook is fixed in all methods, varying the encoded representation size means varying the number of codebooks.

We do not compare to binary embedding methods in terms of accuracy since they are known to yield much lower accuracy for a given code length than MCQ methods [175, 70] 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, Martinez et al. [132] report encoding ∼190,000 128-dimensional vectors per second with PQ, while our implementation encodes nearly 300,000.

As a final comparison, we include a modified version of Bolt, *Bolt No Quantize*,

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

in our accuracy experiments. This version does not quantize the distance lookup tables. It is not a useful algorithm since it sacrifices Bolt's high speed, but it allows us to assess whether our codebook quantization reduces accuracy.

3.4.3 Encoding Speed

Before a vector quantization method can compute approximate distances, it must first encode the data. We measured how many vectors each algorithm can encode per second as a function of the vectors' length. As shown in Figure 3-2.*left*, Bolt can encode data vectors over 10× faster than PQ, the fastest comparison. Encoding 5 million 128-dimensional vectors of $4B$ floats per second (top left plot) translates to an encoding speed of 2.5GB/s. For perspective, this encoding rate is sufficient to encode the entire Sift1M dataset of 1 million vectors in 200ms, and the Sift1B dataset of 1 billion vectors in 200s. This rate is also an order of magnitude higher than that of fast but general-purpose compression algorithms such as Snappy [77], which reports an encoding speed of 250MB/s.

Similarly, Bolt can compute the distance matrix constituting a query's encoding at over 6 million queries per second (top right plot), while PQ obtains less than 350,000 queries per second. Both of these numbers are sufficiently high that encoding the query is unlikely to be a bottleneck in computing distances to it.

3.4.4 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' speeds in computing Euclidean distances from a set of queries to each vector in a compressed dataset. We do not present results for other distances and similarities since they only affect the computation of queries' distance matrices and therefore have speeds nearly identical to those shown here. In all experiments, the number of compressed data vectors N is fixed at 100,000 and their dimensionality is fixed at 256.

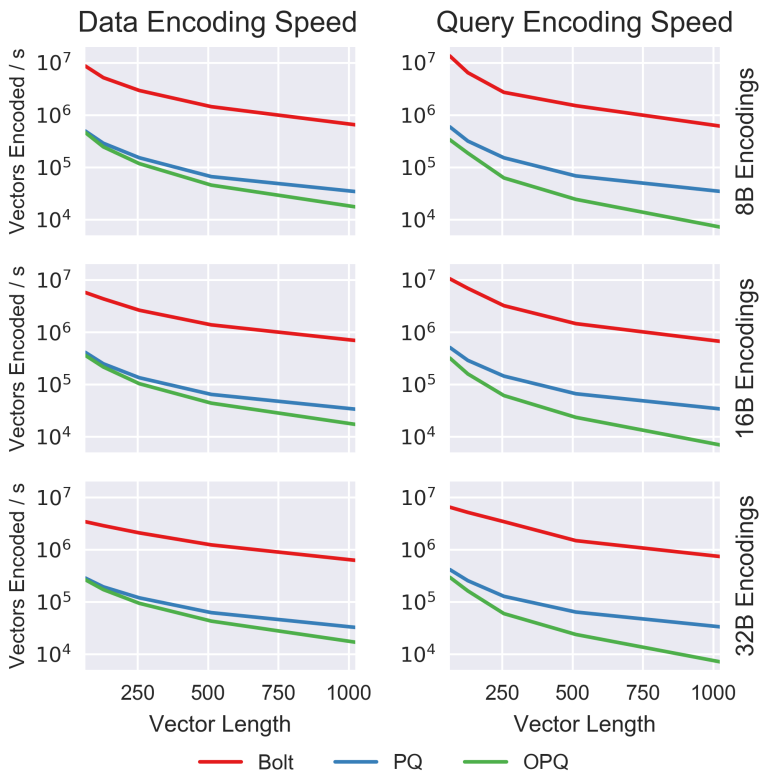

Figure 3-2: Bolt encodes both data and query vectors significantly faster than similar algorithms.

We compare Bolt not only to other MCQ methods, but also to 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 4.2, 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 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 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.

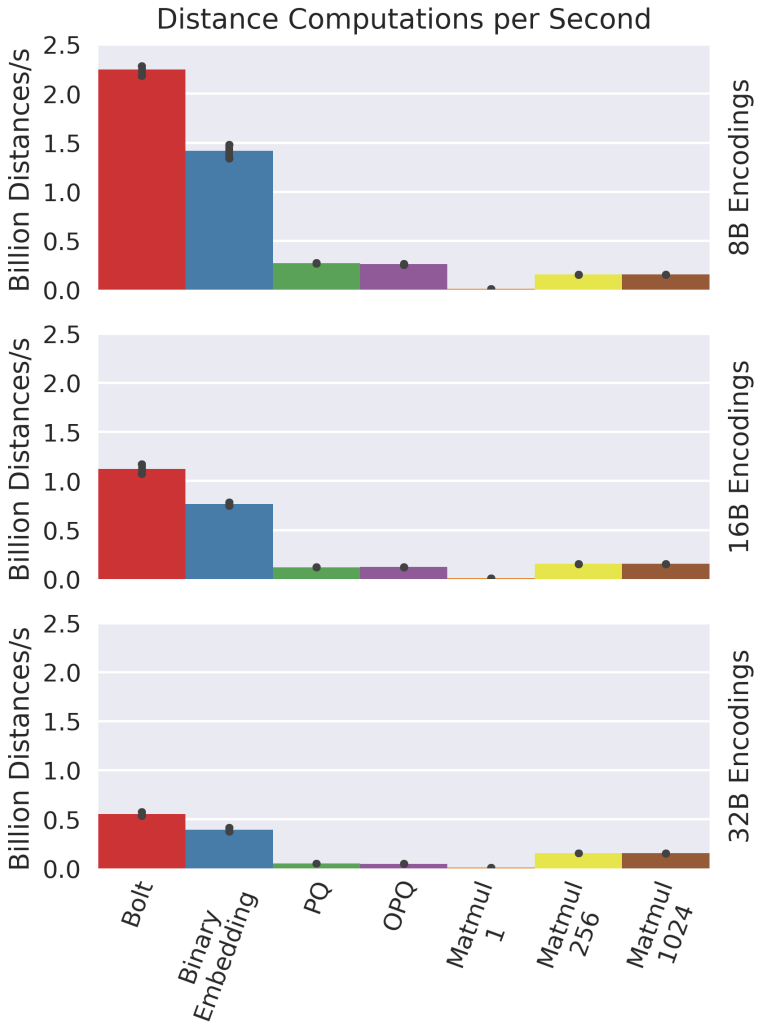

Figure 3-3: Bolt can compute the distances or similarities between a query and the vectors of a compressed database up to 10× faster than other MCQ algorithms. It is also faster than binary embedding methods, which use the hardware `popcount` instruction, and matrix-vector multiplies using batches of 1, 256, or 1024 vectors.

Bolt computes Euclidean distances up to ten times faster than any other MCQ algorithm and significantly faster than binary embedding methods can compute Hamming distances (Figure 3-3). 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 roughly 250× faster, Bolt 16B is 140× faster, and Bolt 32B is 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×.

Because matrix multiplies are so ubiquitous in data mining, machine learning, and many other fields, we compare Bolt to matrix multiplication using dedicated matrix-multiply routines. In Figure 3-4, we profile the time that Bolt and a state-of-the-art BLAS implementation [76] take to do matrix multiplies of various sizes. Bolt computes matrix multiplies by treating each row of the first matrix as a query, treating the second matrix as the 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 for moderately large matrices.

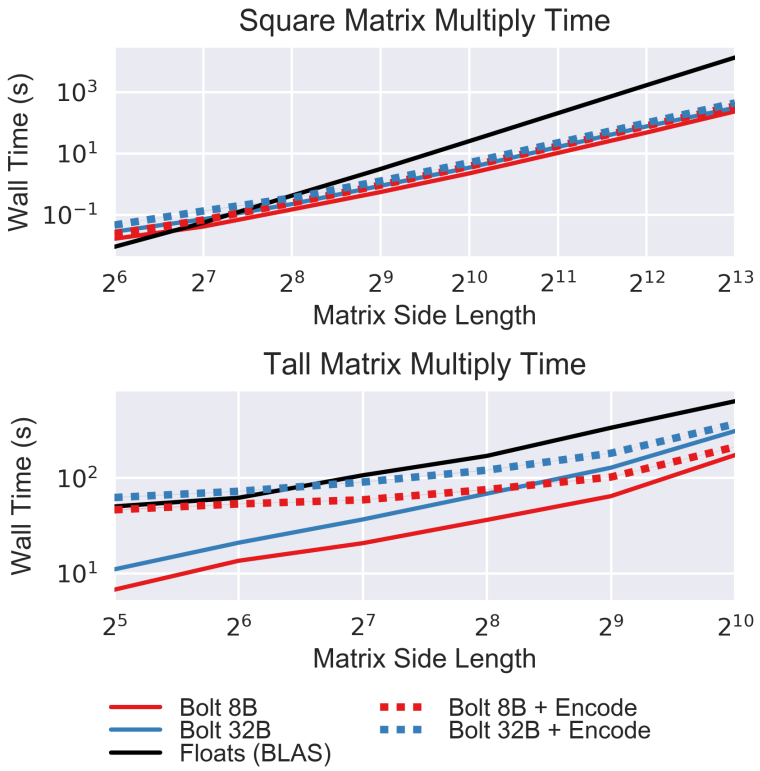

Figure 3-4: Using a naive nested loop implementation, Bolt can compute approximate matrix products faster than optimized matrix multiply routines. Except for small matrices, Bolt is faster even when it must encode the matrices from scratch as a first step.

In Figure 3-4.*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 (e.g. 4096×4096) since encoding so many dimensions accurately would almost certainly require more than 32B.

In Figure 3-4.*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 pay the cost of encoding 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.

3.4.5 Nearest Neighbor Accuracy

The most common assessment of MCQ algorithms' accuracy is their Recall@R. This is defined as the fraction of the queries $\boldsymbol{q}$ for which the true nearest neighbor in Euclidean space is among the top R points with smallest approximate distances to $\boldsymbol{q}$. This is a proxy for how many points would likely have to be re-ranked in a retrieval context when using an approximate distance measure to generate a set of candidates. As shown in Figure 3-5, 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 lookup table quantization introduces little or no error.

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. In 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.

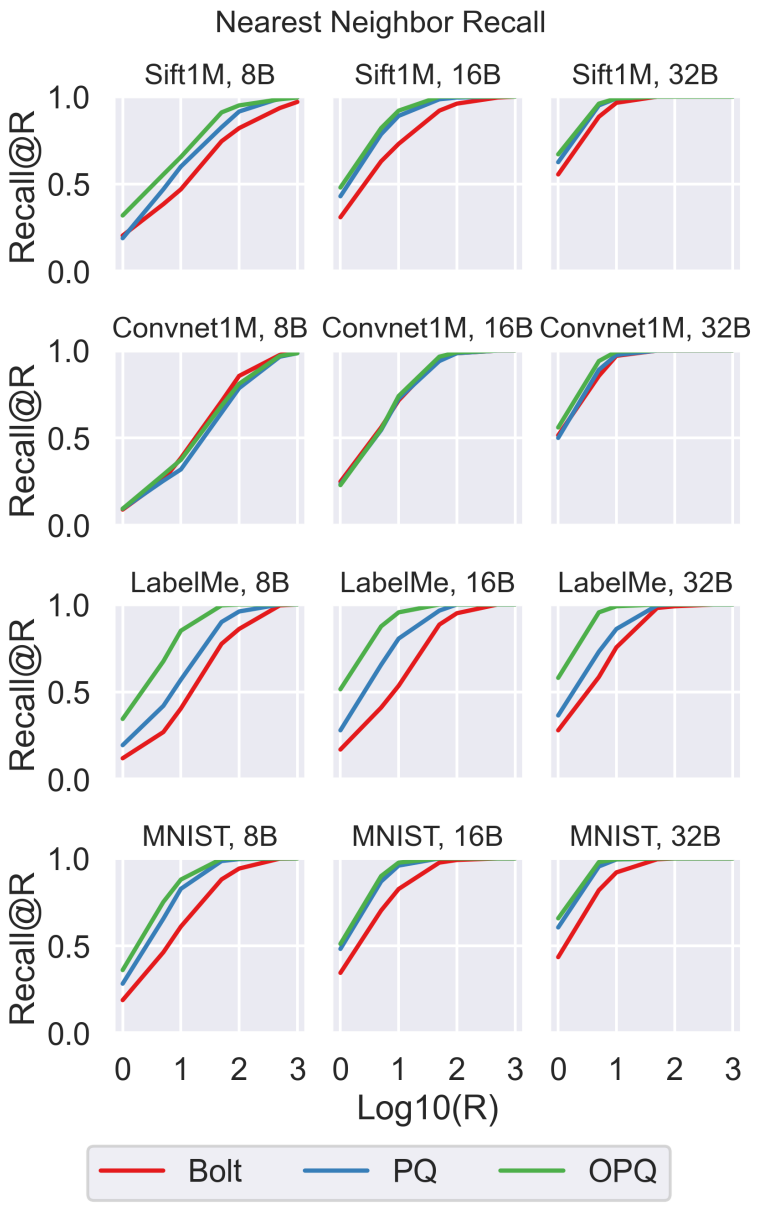

Figure 3-5: Compared to other MCQ algorithms, Bolt is slightly less accurate in retrieving the nearest neighbor for a given encoding length.

Bolt, with its larger number of smaller codebooks, must quantize more sets of dimensions independently, which does not allow it to exploit this mutual information. Much the same phenomena explain the results on MNIST.

For the LabelMe dataset, the correlations between dimensions 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.

3.4.6 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. Distances to dissimilar points are not needed in a similarity search, but if Bolt preserves *all* distances well, it may be fruitful to apply it to other tasks in the future (e.g., approximate matrix multiplication). To assess the extent to which Bolt preserves distances in general, 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. We use correlation, rather than a different metric, since it is invariant to affine transforms; this is important because Bolt produces quantized estimates that are offset and scaled relative to the true distances. As Figure 3-6 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.

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 smaller speedups with compression ratios of 16×, 64×, and 98×.

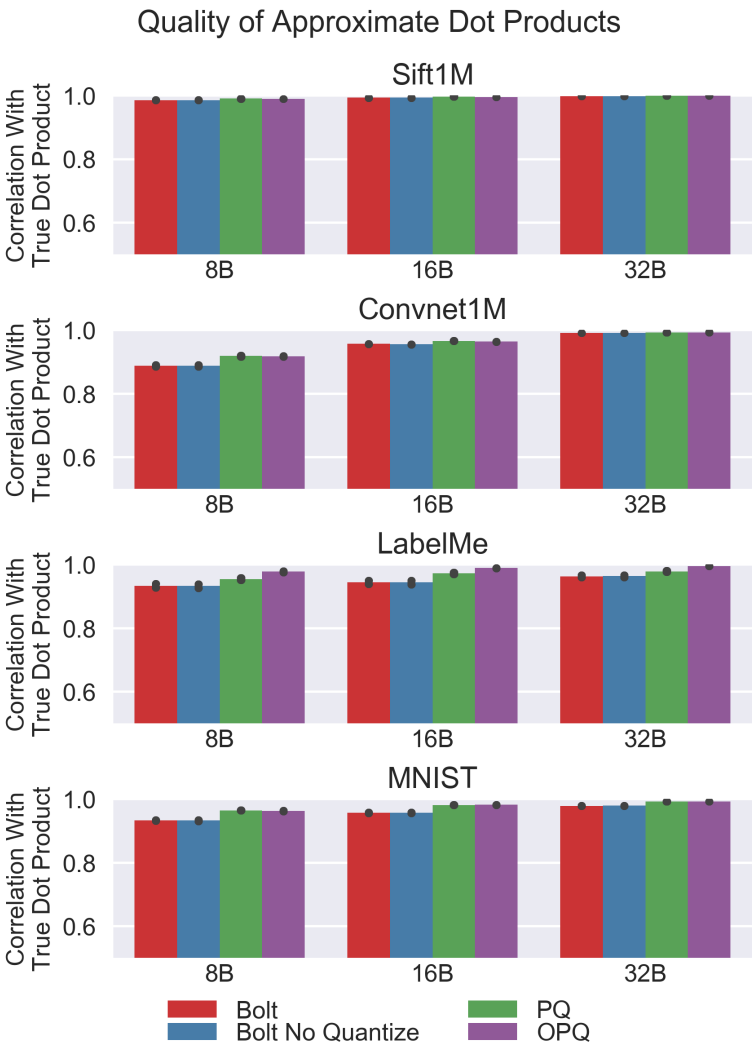

Figure 3-6: Bolt dot products are highly correlated with true dot products, though slightly less so than those from other MCQ algorithms.

3.5 Summary

We describe 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 up to 10× faster than existing algorithms, making it advantageous both in read-heavy and write-heavy scenarios. Its approximate computations can be over 100× 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 or maximum inner product searches.

Chapter 4

Fast Approximate Matrix Multiplication

4.1 Introduction

Matrix multiplication is among the most fundamental subroutines used in machine learning and scientific computing. As a result, there has been a great deal of work on implementing high-speed matrix multiplication libraries [147, 76, 9], designing custom hardware to accelerate multiplication of certain classes of matrices [81, 40, 146, 100], scaling matrix multiplication across many machines [183, 61, 182, 97], and designing efficient Approximate Matrix Multiplication (AMM) algorithms under various assumptions and problem settings [57, 129, 180, 139, 31].

We focus on the AMM task under the assumptions that the matrices are tall, too dense to merit use of sparsity-exploiting kernels, and resident in a single machine's memory. In this setting, the primary challenge is minimization of the amount of CPU time required to approximate linear operations with a given level of fidelity.

This setting arises naturally in machine learning and data mining when one has a data matrix $\boldsymbol{A}$ whose rows are samples and a linear operator $\boldsymbol{B}$ one wishes to apply to these samples. $\boldsymbol{B}$ could be a linear classifier, linear regressor, or an embedding matrix, among other possibilities.

As a concrete example, consider the task of approximating a softmax classifier

trained to predict image labels given embeddings derived from a neural network. Here, the rows of $\boldsymbol{A}$ are the embeddings for each image, and the columns of $\boldsymbol{B}$ are the weight vectors for each class. Classification is performed by computing the product $\boldsymbol{AB}$ and taking the argmax within each row of the result. Because taking the argmax is inexpensive, the bottleneck is computing the matrix product.

Our method represents a significant methodological departure from most traditional approaches to this problem. Traditional AMM methods construct matrices $\boldsymbol{V}_A, \boldsymbol{V}_B \in \mathbb{R}^{D \times d}, d \ll D$ such that

$$\boldsymbol{AB} \approx (\boldsymbol{AV}_A)(\boldsymbol{V}_B^\top \boldsymbol{B}). \tag{4.1}$$

Often, $\boldsymbol{V}_A$ and $\boldsymbol{V}_B$ are sparse, embody some sort of sampling scheme, or have other structure such that these projection operations are faster than a dense matrix multiply. In short, these methods use linear functions to preprocess $\boldsymbol{A}$ and $\boldsymbol{B}$ and reduce the problem to exact matrix multiplication in a lower-dimensional space.

Our proposed method, MADDNESS[1], instead employs a *nonlinear* preprocessing function and reduces the problem to table lookups. Moreover, in the case that $\boldsymbol{B}$ is known ahead of time—which happens when applying a trained linear model to new data, among other situations—MADDNESS does not require any multiply-add operations.

Our method is most closely related to vector quantization methods used for similarity search (e.g., [17, 18, 98, 70] and Chapter 3). However, instead of using an expensive quantization function that requires many multiply-adds, we introduce a fast family of quantization functions that require no multiply-adds.

Our contributions can be summarized as follows:

- An efficient family of vector quantization functions that can encode over 100GB of data per second in a single CPU thread.
- A high-speed summation algorithm for low-bitwidth integers that avoids upcasting, saturation, and overflow.

[1] Multiply-ADDitioN-lESS

- An algorithm based on these functions for approximate matrix multiplication. Experiments across hundreds of diverse matrices demonstrate that this algorithm significantly outperforms existing methods.

4.1.1 Problem Formulation

Let $\boldsymbol{A} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{B} \in \mathbb{R}^{D \times M}$ be two matrices, with $N \gg D \geq M$. Given a computation time budget τ, our task is to construct three functions $g(\cdot)$, $h(\cdot)$, and $f(\cdot)$, along with constants α and β, such that

$$\|\alpha f(g(\boldsymbol{A}), h(\boldsymbol{B})) + \boldsymbol{\beta} - \boldsymbol{AB}\|_F < \varepsilon(\tau)\|\boldsymbol{AB}\|_F \tag{4.2}$$

for the smallest error $\varepsilon(\tau)$ possible. The constants α and β are separated from $f(\cdot, \cdot)$ so that $f(\cdot, \cdot)$ can produce low-bitwidth outputs (e.g., in the range $[0, 255]$) even when the entries of $\boldsymbol{AB}$ do not fall in this range.

We assume the existence of a training set $\tilde{\boldsymbol{A}}$, whose rows are drawn from the same distribution as the rows of $\boldsymbol{A}$. This is a natural assumption in the case that rows of $\boldsymbol{A}$ represent examples in training data, or structured subsets thereof (such as patches of images). This assumption is common when using vector quantization for information retrieval [31, 22, 78], but is a significant departure from most theoretical work.

4.2 Related Work

Because our work draws on ideas from randomized algorithms, approximate matrix multiplication, vector quantization, and other fields, the body of work related to our own is vast. Here, we provide only a high-level overview, and refer the interested reader to surveys of this area [176, 175, 53] for more information. We also defer discussion of related vector quantization methods to the following sections, since it is easier to appreciate how they differ from our own method once our method has been introduced.

4.2.1 Linear Approximation

Most AMM methods work by projecting $\boldsymbol{A}$ and $\boldsymbol{B}$ into lower-dimensional spaces and then performing an exact matrix multiply. One simple option for choosing the projection matrices is to use matrix sketching algorithms. The most prominent deterministic matrix sketching methods are the Frequent Directions algorithm [121, 72] and its many variations [168, 68, 180, 92, 126, 67]. There are also many randomized sketching methods [155, 112, 145, 46, 140] and sampling methods [58, 59].

A weakness of matrix sketching methods in the context of matrix multiplication is that they consider each matrix in isolation. To exploit information about both matrices simultaneously, Drineas et al. [57] sample columns of $\boldsymbol{A}$ and rows of $\boldsymbol{B}$ according to a sampling distribution dependent upon both matrices. Later work by Manne et al. [129] reduce approximation of the matrices to an optimization problem, which is solved by steepest descent. More recently, several authors have introduced variations of the Frequent Directions algorithm that take into account both matrices [139, 180, 67].

All of the above methods differ from our own not merely in specifics, but also in problem formulation. These methods all assume that there is no training set $\tilde{\boldsymbol{A}}$ and nearly all focus on large matrices, where provably reduced asymptotic complexity for a given level of error is the goal.

4.2.2 Hashing to Avoid Linear Operations

In the neural network acceleration literature, there have been several efforts to accelerate dense linear layers using some form of hashing [164, 38, 27, 51, 39]. These methods differ from our own in the hash functions chosen, in not exploiting a training set, and in the overall goal of the algorithm. While we seek to approximate the entire output matrix, these methods seek to either sample outputs [164, 38], approximate only the largest outputs [27, 51], or implement a fixed, sparse linear operator [39].

4.3 Background - Product Quantization

To lay the groundwork for our own method, we begin by reviewing Product Quantization (PQ) [98]. PQ is a classic vector quantization algorithm for approximating inner products and Euclidean distances and serves as the basis for nearly all vector quantization methods similar to our own. Readers who already understand PQ or have read Chapter 3 can safely skip this section. The only significant notation introduced is the use of C to denote the number of codebooks, K to denote the number of prototypes per codebook, and $\mathcal{J}^{(c)}$ to denote the set of indices associated with a given codebook c.

The basic intuition behind PQ is that $\boldsymbol{a}^\top \boldsymbol{b} \approx \hat{\boldsymbol{a}}^\top \boldsymbol{b}$, where $\|\hat{\boldsymbol{a}} - \boldsymbol{a}\|$ is small but $\hat{\boldsymbol{a}}$ has special structure allowing the product to be computed quickly. This structure consists of $\hat{\boldsymbol{a}}$ being formed by concatenating learned prototypes in disjoint subspaces; one obtains a speedup by precomputing the dot products between $\boldsymbol{b}$ and the prototypes once, and then reusing these values across many $\boldsymbol{a}$ vectors. The $\boldsymbol{a}$ vectors here are the transposed rows of $\boldsymbol{A}$ and the $\boldsymbol{b}$ vectors are the columns of $\boldsymbol{B}$.

In somewhat more detail, PQ consists of the following:

1. **Prototype Learning** - In an initial, offline training phase, cluster the rows of $\boldsymbol{A}$ (or a training set $\tilde{\boldsymbol{A}}$) using k-means to create prototypes. A separate k-means is run in each of C disjoint subspaces to produce C sets of K prototypes.
2. **Encoding Function**, $g(\boldsymbol{a})$ - Determine the most similar prototype to $\boldsymbol{a}$ in each subspace. Store these assignments as integer indices using $C \log_2(K)$ bits.
3. **Table Construction**, $h(\boldsymbol{B})$ - Precompute the dot products between $\boldsymbol{b}$ and each prototype in each subspace. Store these partial dot products in C lookup tables of size K.
4. **Aggregation**, $f(\cdot, \cdot)$ - Use the indices and tables to lookup the estimated partial $\boldsymbol{a}^\top \boldsymbol{b}$ in each subspace, then sum the results across all C subspaces.

We elaborate upon each of these steps below and refer the reader to Figure 3-1 for a graphical depiction of PQ approximating the product of two vectors.

Prototype Learning: Let $\tilde{A} \in \mathbb{R}^{N \times D}$ be a training set, K be a number of prototypes per subspace, C be a number of subspaces, and $\{\mathcal{J}^{(c)}\}_{c=1}^{C}$ be the mutually exclusive and collectively exhaustive sets of indices associated with each subspace. The training-time task of PQ is to learn C sets of prototypes $\boldsymbol{P}^{(c)} \in \mathbb{R}^{K \times |\mathcal{J}^{(c)}|}$ and assignments $\boldsymbol{z}^{(c)} \in \mathbb{R}^{N}$ such that

$$\sum_{i=1}^{N} \sum_{c=1}^{C} \sum_{j \in \mathcal{J}^{(c)}} \left(\tilde{\boldsymbol{A}}_{ij} - \boldsymbol{P}^{(c)}_{\boldsymbol{z}^{(c)},j} \right)^2 \tag{4.3}$$

is minimized. It does this by running k-means separately in each subspace $\mathcal{J}^{(c)}$ and using the resulting centroids and assignments to populate $\boldsymbol{P}^{(c)}$ and $\boldsymbol{z}^{(c)}$.

Encoding Function, $\boldsymbol{g(A)}$: Given the learned prototypes, PQ replaces each row $\boldsymbol{a}$ of $\boldsymbol{A}$ with the concatenation of its C k-means centroid assignments in each of the C subspaces. Formally:

$$g^{(c)}(\boldsymbol{a}) \triangleq \operatorname*{argmin}_{k} \sum_{j \in \mathcal{J}^{(c)}} \left(\boldsymbol{a}_j - \boldsymbol{P}^{(c)}_{k,j} \right)^2. \tag{4.4}$$

We will refer to the resulting sequence of indices as the *encoding* of $\boldsymbol{a}$ and the set of K centroids as a *codebook*. For convenience, we will also refer to the vector $\boldsymbol{a}^{(c)} \triangleq \langle a_j \rangle, j \in \mathcal{J}^{(c)}$ as the *subvector* of $\boldsymbol{a}$ in subspace c.

Table Construction, $\boldsymbol{h(B)}$: Using these same prototypes, PQ constructs a lookup table $h^{(c)}(\boldsymbol{b}) \in \mathbb{R}^{K}$ in each of the C subspaces for each column $\boldsymbol{b}$ of $\boldsymbol{B}$, where

$$h^{(c)}(\boldsymbol{b})_k \triangleq \sum_{j \in \mathcal{J}^{(c)}} \boldsymbol{b}_j \boldsymbol{P}^{(c)}_{k,j}. \tag{4.5}$$

Existing work has shown that setting $K = 16$ and quantizing the lookup tables to 8 bits can offer enormous speedups compared to larger K and/or floating point tables [31, 17, 18]. This is because 16 1-byte entries can be stored in a SIMD register, allowing 16 or more table lookups to be performed in parallel in a single instruction. Since the table entries naturally occupy more than 8 bits even for 8-bit data, some means of quantizing these entries is necessary. This can easily be done by subtracting

off the minimum entry in each table and linearly rescaling such that the maximum entry in any table is at most 255. Ignoring rounding error, this affine transform is invertible, and is reflected by the constants α and β in Equation 4.2. See Section C.1 for additional details.

Aggregation, $f(\cdot,\cdot)$: Given the encoding of $\boldsymbol{a}$ and the lookup tables for $\boldsymbol{b}$, the product can be approximated as

$$\boldsymbol{a}^\top \boldsymbol{b} = \sum_{c=1}^{C} \boldsymbol{a}^{(c)\top} \boldsymbol{b}^{(c)} \approx \sum_{c=1}^{C} h^{(c)}(\boldsymbol{b})_k, \;\; k = g^{(c)}(\boldsymbol{a}). \tag{4.6}$$

4.4 Our Method

Product Quantization and its variants yield a large speedup with $N, M \gg D$. However, we require an algorithm that only needs $N \gg M, D$, a more relaxed scenario common when using linear models and transforms. In this setting, the preprocessing time $g(\boldsymbol{A})$ can be significant, since ND may be similar to, or even larger than, NM.

To address this case, we introduce a new $g(\boldsymbol{A})$ function that yields large speedups even on much smaller matrices. The main idea behind our function is to determine the "most similar" prototype through locality-sensitive hashing [94]; i.e., rather than compute the Euclidean distance between a subvector $\boldsymbol{a}^{(c)}$ and each prototype, we hash $\boldsymbol{a}^{(c)}$ to one of K buckets where similar subvectors tend to hash to the same bucket. The prototypes are set to the means of the subvectors hashing to each bucket.

4.4.1 Hash Function Family, $g(\cdot)$

Because we seek to exploit a training set while also doing far less work than even a single linear transform, we found that existing hash functions did not meet our requirements. Consequently, we designed our own family of trainable hash functions. This function family may be of independent interest, but we leave exploration of its efficacy in other contexts to future work.

The family of hash functions we choose is balanced binary regression trees, with each leaf of the tree acting as one hash bucket. The leaf for a vector $\boldsymbol{x}$ is chosen

by traversing the tree from the root and moving to the left child if the value x_j at some index j is below a node-specific threshold v, and to the right child otherwise. To enable the use of SIMD instructions, the tree is limited to 16 leaves and all nodes at a given level of the tree are required to split on the same index j. The number 16 holds across many processor architectures, and we refer the reader to Section C.2 for further vectorization details.

Formally, consider a set of four indices $j^1, \ldots, j^4$ and four arrays of split thresholds $\boldsymbol{v}^1, \ldots, \boldsymbol{v}^4$, with v_t having length 2^{t-1}. A vector $\boldsymbol{x}$ is mapped to an index using Algorithm 4.

This function is simple, only depends on a constant number of indices in the input vector, and can easily be vectorized provided that the matrix whose rows are being encoded is stored in column-major order.

Algorithm 4 MADDNESSHASH

Input: vector $\boldsymbol{x}$, split indices $j^1, \ldots, j^4$, split thresholds $\boldsymbol{v}^1, \ldots, \boldsymbol{v}^4$
$i \leftarrow 1$ // node index within level of tree
for $t \leftarrow 1$ **to** 4 **do**
    $v \leftarrow \boldsymbol{v}_i^t$ // split threshold for node i at level t
    $b \leftarrow x_{j^t} \geq v \ ? \ 1 : 0$ // above split threshold?
    $i \leftarrow 2i - 1 + b$ // assign to left or right child
return i

4.4.2 Learning the Hash Function Parameters

The split indices $j^1, \ldots, j^4$ and split thresholds $\boldsymbol{v}^1, \ldots, \boldsymbol{v}^4$ are optimized on the training matrix $\tilde{\boldsymbol{A}}$ using a greedy tree construction algorithm. To describe this algorithm, we introduce the notion of a *bucket* $\mathcal{B}_i^t$, which is the set of vectors mapped to node i in level t of the tree. The root of the tree is level 0 and $\mathcal{B}_1^0$ contains all the vectors. It will also be helpful to define the sum of squared errors (SSE) loss associated with

a bucket, or a specific (index, bucket) pair:

$$\mathcal{L}(j,\,\mathcal{B}) \triangleq \sum_{\boldsymbol{x}\in\mathcal{B}} \left(x_j - \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x}'\in\mathcal{B}} x'_j \right)^2 \tag{4.7}$$

$$\mathcal{L}(\mathcal{B}) \triangleq \sum_{j} \mathcal{L}(j,\,\mathcal{B}). \tag{4.8}$$

Using this notation, it suffices to characterize the learning algorithm by describing the construction of level t of the tree given the buckets $\mathcal{B}_1^{t-1}, \ldots, \mathcal{B}_{2^{t-1}}^{t-1}$ from the previous level. This procedure is given in Algorithm 5.

In line 2, we select a fixed number of indices to evaluate. Several heuristics are possible, including evaluating all indices. We found that simply selecting the top n indices that contributed the most loss summed across all buckets was difficult to beat. In preliminary experiments, we found that using $n > 4$ indices offered little or no additional benefit, and even choosing $n = 1$ was nearly as good; consequently, all reported results use $n = 4$.

In lines 4-15, we find the minimal loss obtainable by splitting all buckets along that index, but with bucket-specific cutoffs. This loss is minimal not in the sense that it leads to a globally optimal tree, but in that it minimizes the sum of the losses in the buckets produced in this iteration. To do this, we invoke the subroutine `optimal_split_threshold`, which takes in a bucket $\mathcal{B}$ and an index j and tests all possible thresholds to find one minimizing $\mathcal{L}(j,\,\mathcal{B})$. This can be done in time $O(|\mathcal{J}^{(c)}||\mathcal{B}|\log(|\mathcal{B}|))$. The pseudocode for this subroutine is given in Algorithms 6 and 7 in Section C.3.

Once a split index j and an array of split thresholds $\boldsymbol{v}$ are chosen, all that remains is to split the buckets to form the next level of the tree (lines 16-21). This entails forming two child buckets from each current bucket by grouping vectors whose jth entries are above or below the bucket's split threshold.

Algorithm 5 Adding The Next Level to the Hashing Tree

Input: buckets $\mathcal{B}_1^{t-1}, \ldots, \mathcal{B}_{2^{t-1}}^{t-1}$, training matrix $\tilde{\boldsymbol{A}}$
// Greedily choose next split index and thresholds
$\hat{\mathcal{J}} \leftarrow$ `heuristic_select_idxs`$(\mathcal{B}_1^{t-1}, \ldots, \mathcal{B}_{2^{t-1}}^{t-1})$
l^{min}, j^{min}, $\boldsymbol{v}^{min} \leftarrow \infty$, NaN, NaN
for $j \in \hat{\mathcal{J}}$ **do**
    $l \leftarrow 0$ // initialize loss for this index to 0
    $\boldsymbol{v} \leftarrow [\,]$ // empty list of split thresholds
    for $i \leftarrow 1$ **to** 2^{t-1} **do**
        v_i, $l_i \leftarrow$ `optimal_split_threshold`$(j, \mathcal{B}_i^{t-1})$
        `append`$(\boldsymbol{v}, v_i)$ // append threshold for bucket i
        $l \leftarrow l + l_i$ // accumulate loss from bucket i
    if $l < l^{min}$ **then**
        $l^{min} \leftarrow l$, $j^{min} \leftarrow j$, $\boldsymbol{v}^{min} \leftarrow \boldsymbol{v}$ // new best split
// Create new buckets using chosen split
$\boldsymbol{\mathcal{B}} \leftarrow [\,]$
for $i \leftarrow 1$ **to** 2^{t-1} **do**
    $\mathcal{B}_{below}$, $\mathcal{B}_{above} \leftarrow$ `apply_split`$(v_i^{min}, \mathcal{B}_i^{t-1})$
    `append`$(\boldsymbol{\mathcal{B}}, \mathcal{B}_{below})$
    `append`$(\boldsymbol{\mathcal{B}}, \mathcal{B}_{above})$
return $\boldsymbol{\mathcal{B}}$, l^{min}, j^{min}, v^{min}

4.4.3 Optimizing the Prototypes

At this point, we have a complete algorithm. We could simply drop our hash-based encoding function into PQ and approximate matrix products. However, we contribute two additional enhancements: a means of optimizing the prototypes with no runtime overhead, and a means of quickly summing low-bitwidth integers.

First, we introduce a means of optimizing the prototypes given only the matrix $\tilde{\boldsymbol{A}}$. Several works propose prototype or table optimizations based on knowledge of $\boldsymbol{B}$ [22, 174], and others optimize them at the expense of slowing down the function $g(\cdot)$ [185, 186]. We introduce a means of optimizing the centroids that does not do either of these. The idea is to choose prototypes such that $\tilde{\boldsymbol{A}}$ can be reconstructed from its prototypes with as little squared error as possible—this improves results since less error means that less information about $\tilde{\boldsymbol{A}}$ is being lost.

Let $\boldsymbol{P} \in \mathbb{R}^{KC \times D}$ be a matrix whose diagonal blocks of size $K \times |\mathcal{J}^{(c)}|$ consist

of the K learned prototypes in each subspace c. The training matrix $\tilde{\boldsymbol{A}}$ can be approximately reconstructed as

$$\tilde{\boldsymbol{A}} \approx \boldsymbol{G}\boldsymbol{P} \tag{4.9}$$

where $\boldsymbol{G}$ serves to select the appropriate prototype in each subspace. Rows of $\boldsymbol{G}$ are formed by concatenating the one-hot encoded representations of each assignment for the corresponding row of $\tilde{\boldsymbol{A}}$. For example, if a row were assigned prototypes $\langle$`3 1 2`$\rangle$ with $K = 4$, $C = 3$, its row in $\boldsymbol{G}$ would be $\langle$`0010 1000 0100`$\rangle \in \mathbb{R}^{12}$. Our idea is to optimize $\boldsymbol{P}$ conditioned on $\boldsymbol{G}$ and $\tilde{\boldsymbol{A}}$. This is an ordinary least squares problem, and we solve it with ridge regression:

$$\boldsymbol{P} \triangleq (\boldsymbol{G}^\top \boldsymbol{G} + \lambda \boldsymbol{I})^{-1} \boldsymbol{G}^\top \tilde{\boldsymbol{A}}. \tag{4.10}$$

One could obtain better performance by cross-validating to find λ, but for simplicity, we hardcode $\lambda = 1$.

In short, the initial prototypes are used to produce an assignment matrix $\boldsymbol{G}$, and it is from this assignment matrix that the final prototypes are derived. This procedure allows the prototypes to be nonzero outside of their original subspaces. Because of our hashing procedure, we avoid the overhead faced by other methods with non-orthogonal prototypes (c.f. [25, 24, 185, 123, 131, 132]).

4.4.4 Fast 8-Bit Aggregation, $f(\cdot, \cdot)$

Let $\boldsymbol{T} \in \mathbb{R}^{M \times C \times K}$ be the tensor of lookup tables for all M columns of $\boldsymbol{B}$. Given the encodings $\boldsymbol{G}$, the function $f(\cdot, \cdot)$ is defined as

$$f(g(\boldsymbol{A}), h(\boldsymbol{B}))_{n,m} \triangleq \sum_{c=1}^{C} \boldsymbol{T}_{m,c,k}, \;\; k = g^{(c)}(\boldsymbol{a}_n). \tag{4.11}$$

Because the entries of $\boldsymbol{T}$ are stored as 8-bit values, exact summation requires immediately upcasting each looked-up entry to 16 bits before performing any addition

instructions [31]. This not only imposes overhead directly, but also means that one must perform 16-bit additions, which have half the throughput of 8-bit additions.

We propose an alternative that sacrifices a small amount of accuracy for a significant increase in speed. Instead of using *addition* instructions, we use *averaging* instructions, such as `vpavgb` on x86 or `vrhadd` on ARM. While non-saturating additions compute $(a + b)$ % 256, these instructions compute $(a + b + 1)/2$. This means that they lose information about the low bit instead of the high bit of the sum. We estimate the overall mean by averaging pairs of values, then pairs of pairs, and so on. One could reduce all C values this way, but we find that one obtains a better speed-accuracy tradeoff by computing the average of blocks of U values and then upcasting to obtain exact sums of these averages.[2] Multiplying this sum of averages by U and adding in a bias correction term gives an overall estimate of the sum. One could tune U for a particular problem and hardware, but we simply set $U = 16$ in all our experiments.

The challenging part of this approach is computing the bias in the estimated sum in order to correct for it. We prove in Section B.2 that this bias is equal to $C \log_2(U)/4$ under the realistic assumption that the low bits are equally likely to be 0 or 1.

Because of our assumption that we are operating on matrices, rather than a matrix and a vector, we can also improve on the aggregation of existing methods [31, 17, 18] by fusing the aggregation of two or more output columns to hide read latency. Conceptually, this amounts to tiling the loop over output columns and alternating reads between the two corresponding tables within the innermost loop.

4.4.5 Complexity

Our encoding function $g(\boldsymbol{A}), \boldsymbol{A} \in \mathbb{R}^{N \times D}$ has complexity $\Theta(NC)$, since it does a constant amount of work per row per codebook. Our table creation function $h(\boldsymbol{B}), \boldsymbol{B} \in \mathbb{R}^{D \times M}$ has complexity $\Theta(MKCD)$, since it must compute the inner product between

[2]This tradeoff exists because upcasting more rarely yields diminishing speedups, but not diminishing accuracy loss. The diminishing speedups stem from the upcasting (and subsequent 16-bit summation) being less of a bottleneck as they become less frequent.

each column of $\boldsymbol{B}$ and KC prototypes of length D. This is a factor of C worse than PQ since we do not require the prototypes for different codebooks to have disjoint nonzero indices. However, this reduction in the speed of $h(\cdot)$ is not a concern because $N \gg M, D$; moreover, the matrix $\boldsymbol{B}$ is often known ahead of time in realistic settings, allowing $h(\boldsymbol{B})$ to be computed offline. Finally, the complexity of our aggregation function $f(\cdot)$ is $\Theta(NCM)$, since it performs C table lookups for each of M output columns and N output rows. This means our overall algorithm has complexity $\Theta(MC(KD + N))$, which reduces to $\Theta(NCM)$ since we fix $K = 16$ and our problem statement requires $N \gg D$.

4.4.6 Theoretical Guarantees

We begin by noting that all of the guarantees for Bolt also apply to Maddness, modulo a small amount of additional error from averaging integers rather than summing exactly. This follows from Bolt's guarantees being dependent only upon the quantization errors, rather than the method used to obtain them. Beyond these existing guarantees, our central theoretical result is a generalization guarantee for the overall performance of Maddness:

Theorem 4.4.1 (Generalization Error of Maddness)**.** *Let $\mathcal{D}$ be a probability distribution over $\mathbb{R}^D$ and suppose that* Maddness *is trained on a matrix $\tilde{\boldsymbol{A}} \in R^{N \times D}$ whose rows are drawn independently from $\mathcal{D}$ and with maximum singular value bounded by σ_A. Let C be the number of codebooks used by* Maddness *and $\lambda > 0$ the regularization parameter used in the ridge regression step. Then for any $\boldsymbol{b} \in \mathbb{R}^D$, any $\boldsymbol{a} \sim \mathcal{D}$, and any $0 < \delta < 1$, we have with probability at least $1 - \delta$ that*

$$\mathbb{E}_{\mathcal{D}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] \leq \mathbb{E}_{\tilde{\boldsymbol{A}}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] + \frac{C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \left(\frac{1}{256} + \frac{8 + \sqrt{C(4\lceil \log_2(D) \rceil + 256) \log 2 - \log \delta}}{\sqrt{2n}} \right) \tag{4.12}$$

where $\mathcal{L}(\boldsymbol{a}, \boldsymbol{b}) \triangleq |\boldsymbol{a}^\top \boldsymbol{b} - \alpha f(g(\boldsymbol{a}), h(\boldsymbol{b})) - \boldsymbol{\beta}|$, α is the scale used for quantizing the lookup tables, and $\boldsymbol{\beta}$ is the constants used in quantizing the lookup tables plus the

debiasing constant of Section 4.4.4.

See Section B for a proof and additional analysis.

4.5 Experiments

To assess MADDNESS's effectiveness, we implemented both it and existing algorithms in C++ and Python. All of our code and raw numerical results are publicly available at `https://smarturl.it/Maddness`. All experiments use a single thread on a Macbook Pro with a 2.6GHz Intel Core i7-4960HQ processor. Unless stated otherwise, all timing results use five trials, with each trial reporting the fastest among 20 executions. We use the best, rather than average, since this is standard practice in performance benchmarking and is robust to the purely additive noise introduced by competing CPU tasks. Standard deviations are shown for all curves as shaded areas, but are often too small to see. Since training can be performed offline and all methods except SparsePCA [127] train in at most a few minutes, we omit profiling of training times. We also do not profile the time to preprocess $\boldsymbol{B}$, since 1) this time is inconsequential in most cases, and 2) $\boldsymbol{B}$ is fixed and could be processed offline in all the problems we consider.

We do not need to tune any hyperparameters for MADDNESS, but we do take steps to ensure that other methods are not hindered by insufficient hyperparameter tuning. Concretely, we sweep a wide range of hyperparameter values and allow them to cherrypick their best hyperparameters on each test matrix. Additional details about data cleaning, hyperparameter tuning, and other minutiae can be found in Section C.4.

Because nearly all existing work on approximate matrix multiplication either focuses on special cases that do not satisfy our problem definition [18, 98, 70] or synthetic matrices, there is not a clear set of benchmark matrix multiply tasks to use. We therefore use a collection of tasks that we believe are both reproducible and representative of many real-world matrices. To the best of our knowledge, our experiments use over an order of magnitude more matrices than any previous study.

4.5.1 Methods Tested

Recall that most baselines take the form of selecting a matrix $\boldsymbol{V} \in \mathbb{R}^{D \times d}, d < D$ such that $\boldsymbol{AB} \approx (\boldsymbol{AV})(\boldsymbol{V}^\top \boldsymbol{B})$. Here d is a free parameter that adjusts the quality versus speed tradeoff. We therefore characterize most of these methods by how they set $\boldsymbol{V}$.

- **PCA**. Set $\boldsymbol{V}$ equal to the top principal components of $\tilde{\boldsymbol{A}}$.
- **SparsePCA** [127]. Set $\boldsymbol{V} = \text{argmin}_{\boldsymbol{V}} \min_{\boldsymbol{U}} \|\tilde{\boldsymbol{A}} - \boldsymbol{U}\boldsymbol{V}^\top\|_F^2 + \lambda \|\boldsymbol{V}\|_1$, where $\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}$. This is not the only dictionary learning formulation referred to as SparsePCA [189, 36], but it is a good representative and is the only one with support in a major Python library.
- **FastJL** [12]. $\boldsymbol{V}$ is set to Rademacher random variables composed with a Fast Hadamard Transform (FHT). For simplicity, we do not include the FHT in the timing.
- **HashJL** [46]. $\boldsymbol{V}$ is zero except for a ± 1 in each row, with both sign and position chosen uniformly at random.
- **ScalarQuantize**. The matrices are not projected, but instead linearly quantized to eight bits such that the largest and smallest entries map to 0 and 255.[3]. We use FBGEMM [108] to perform the quantized matrix multiplies. We neglect the time required to convert from other formats to eight bits, reflecting the optimistic scenario in which the matrices are already of the appropriate types.
- **Bolt** (Chapter 3). Bolt is an extension of PQ that uses quantized lookup tables and a reduced number of codebooks to obtain 10× speedups over traditional PQ. It is the most similar method to Maddness, differing only in the encoding function, the use of averaging instead of upcasting, and the optimization of centroids.
- **Exact Multiplication**. We simply compute the matrix product $\boldsymbol{AB}$ using a modern BLAS implementation.

We also test two variations of our method:

- **Maddness**. The algorithm described in Section 4.4.

[3]FBGEMM requires one unsigned eight bit input and one signed eight bit input, so one matrix's values are actually mapped from -128 to 127

- **Maddness-PQ**. A handicapped version of MADDNESS without the prototype optimization step. The gap between MADDNESS and MADDNESS-PQ is the gain from optimizing the prototypes.

We also compared to many additional methods (see Section C.4), but omit their results since they were not competitive with those listed here.

4.5.2 How Fast is Maddness?

We begin by profiling the raw speed of MADDNESS. In Figure 4-1, we time the $g(\boldsymbol{A})$ functions for various vector quantization methods. The $\boldsymbol{A}$ matrices have 2^{14} rows and varying numbers of columns D. Following Chapter 3, we also vary the number of codebooks C, profiling 8-, 16-, and 32-byte encodings. We measure in bytes rather than codebooks since PQ and OPQ use eight bits per codebook while BOLT and MADDNESS use four.

MADDNESS is up to two orders of magnitude faster than existing methods, and its throughput increases with row length. This latter property is because its encoding cost per row is $O(C)$ rather than $O(D)$.

We also profile the speed of our aggregation function $f(\cdot, \cdot)$ using the same baselines as Chapter 3. As Figure 4-2 shows, our average-based, matrix-aware aggregation is significantly faster than the upcasting-based (and one-vector-at-a-time) method of BOLT.

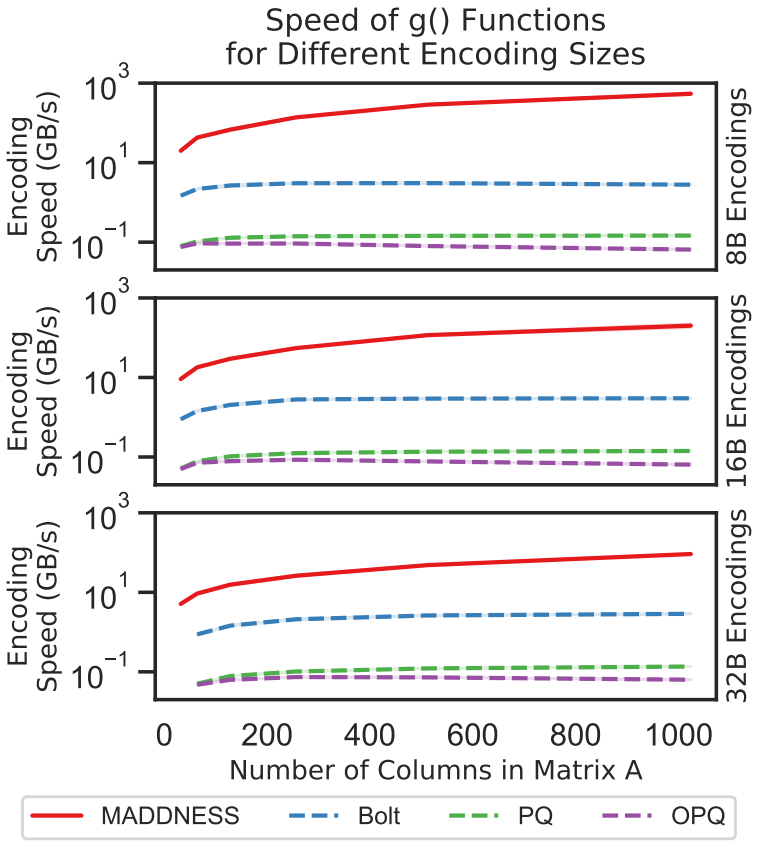

Figure 4-1: Maddness encodes the A matrix orders of magnitude more quickly than existing vector quantization methods.

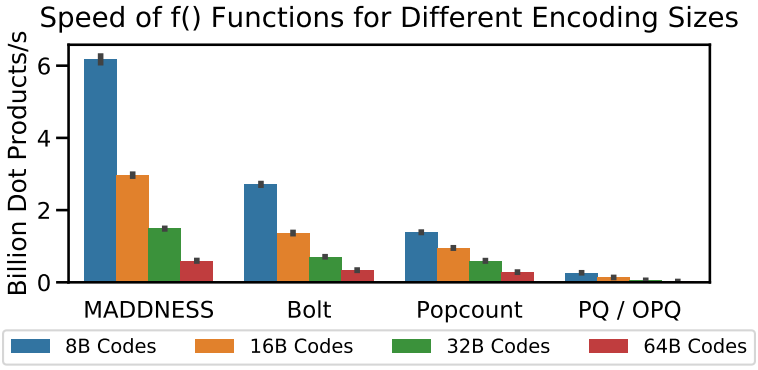

Figure 4-2: Given the preprocessed matrices, Maddness computes the approximate output twice as fast as the fastest existing method.

4.5.3 Softmax Classifier

We approximated linear classifiers on the widely used CIFAR-10 and CIFAR-100 datasets [110]. The classifiers use as input features the 512-dimensional activations of open-source, VGG-like neural networks trained on each dataset [71]. The matrices $\boldsymbol{A}$

are the 10000×512-dimensional floating point activations for the full test sets, and the matrices $\boldsymbol{B}$ are each network's final dense layer. The 50000×512-dimensional activations from the training set serve as the training matrices $\tilde{\boldsymbol{A}}$. As shown in Figure 4-3, MADDNESS significantly outperforms all existing methods. Moreover, MADDNESS achieves virtually the same accuracy as exact multiplication more than an order of magnitude faster.

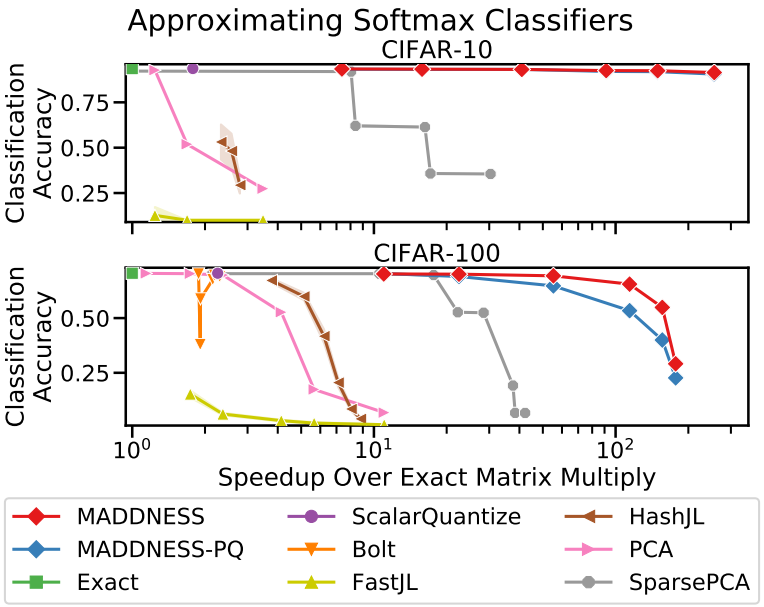

Figure 4-3: Maddness achieves a far better speed-accuracy tradeoff than any existing method when approximating two softmax classifiers.

4.5.4 Kernel-Based Classification

To assess the efficacy of our method on a larger and more diverse set of datasets than simply CIFAR-10 and CIFAR-100, we trained kernel classifiers on the datasets from the UCR Time Series Archive [49]. To enable meaningful speed comparison across datasets, we resampled the time series in all datasets to the median length and obtained the matrix $\boldsymbol{B}$ for each dataset by running Stochastic Neighbor Compression [111] on the training set with an RBF kernel of bandwidth one. We set the number of returned neighbors to 128 (results with 64 and 256 were similar). We approximated the Euclidean distances used by the kernel via the identity $\|\boldsymbol{x}-\boldsymbol{y}\|_2^2 = \|\boldsymbol{x}\|_2^2 - 2\boldsymbol{x}^\top\boldsymbol{y} + \|\boldsymbol{y}\|_2^2$, which consists only of dot products.

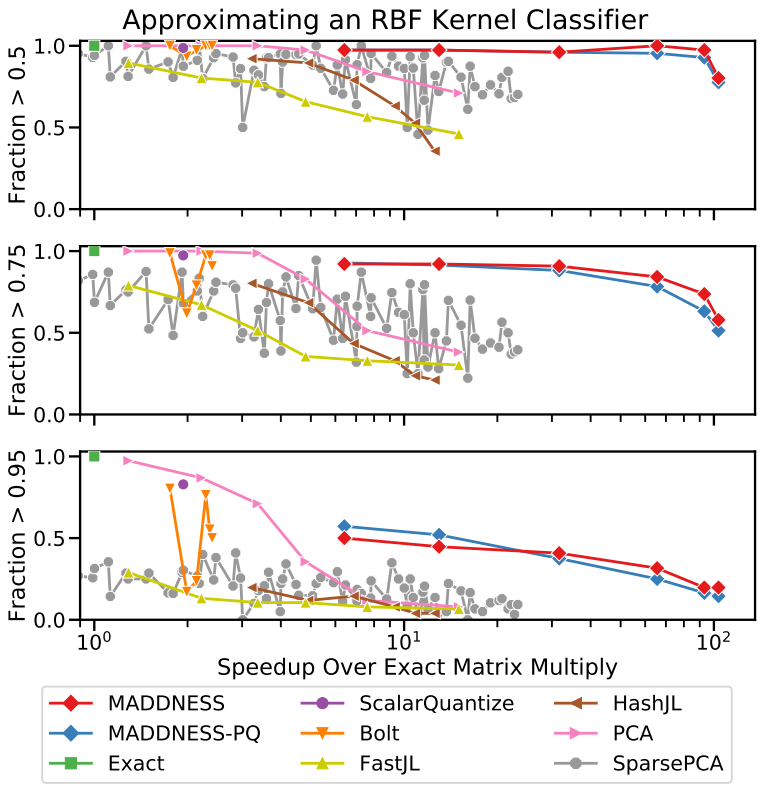

Figure 4-4: Fraction of UCR datasets for which each method preserves a given fraction of the original accuracy versus the method's degree of speedup. Maddness enables much greater speedups for a given level of accuracy degradation.

This is not the state-of-the-art means of classifying time series, but it does yield fixed-sized matrices and is representative of several modern techniques for constructing highly efficient classifiers [111, 177, 188, 80]. This setup also complements the CIFAR results by being an extremely difficult task, since Stochastic Neighbor Compression has already optimized the classifiers to avoid redundancy. As shown in Figure 4-4, MADDNESS is significantly faster at a given level of accuracy. A counter-intuitive result, however, is that optimization of the prototypes occasionally reduces accuracy (see the red line dipping below the blue one in the lowest subplot). Since the optimization strictly increases the expressive power, we believe that this is a product of overfitting and could be corrected were we to tune the ridge regression's parameter (instead of always using $\lambda = 1$). This subplot also reveals that the most stringent degradation requirements can sometimes only be met using PCA at a low speedup (and not MADDNESS), since this is the only curve approaching 1.

4.5.5 Image Filtering

To test the extreme limits of MADDNESS, we benchmarked the various techniques' ability to apply small filters to images. This task is extreme in that D and M are tiny, affording almost no opportunity to amortize preprocessing costs. As representative example filters, we chose 3×3 Sobel kernels and 5×5 Gaussian kernels. These are common high-pass and low-pass filters, respectively. We took the first 10 images from the first 50 classes of the Caltech101 dataset [64] as a single training set, and the first 10 images from the remaining 51 classes as 510 test sets. We constructed the $\boldsymbol{A}$ matrices by extracting each patch of each image as one row. The $\boldsymbol{B}$ matrices have two columns, corresponding to one pair of Sobel or Gaussian filters (since using these filters in pairs is common).

We report the normalized mean-squared error (NMSE), defined as

$$\|\hat{\boldsymbol{C}}_{i,j} - \boldsymbol{A}\boldsymbol{B}\|_F^2 / \|\boldsymbol{A}\boldsymbol{B}\|_F^2 \tag{4.13}$$

where $\hat{\boldsymbol{C}}$ is the method's estimate of $\boldsymbol{A}\boldsymbol{B}$. An NMSE of 0 is perfect and an NMSE of 1 corresponds to always predicting 0. In Figure 4-5, we see that it is only

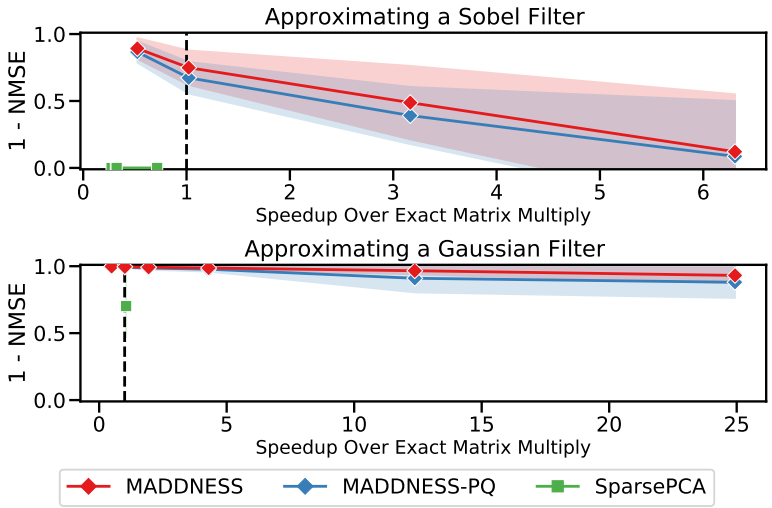

Figure 4-5: Despite there being only two columns in the matrix $\boldsymbol{B}$, Maddness still achieves a significant speedup with reasonable accuracy. Methods that are Pareto dominated by exact matrix multiplication on both tasks are not shown; this includes all methods but Maddness and SparsePCA.

MADDNESS that offers any advantage over exact matrix products. This is likely because two columns afford almost no time to preprocess $\boldsymbol{A}$; indeed, rival vector quantization methods cannot logically do less work than brute force in this setting, and dense linear methods can only save work by embedding rows of $\boldsymbol{A}$ in one-dimensional space.

MADDNESS performs much worse on the high-pass filters *(top)* than the low-pass filters *(bottom)*. This is likely because the former produce outputs with variance that is orders of magnitude lower than that of the original image; this means that the NMSE denominator, $\|\boldsymbol{AB}\|_F^2$, is tiny.

4.6 Summary

We introduce an approximate matrix multiplication algorithm that achieves a significantly better speed-quality tradeoff than existing methods, as measured on various machine learning and other tasks using hundreds of real-world matrices from diverse domains. Our method's performance stems from 1) its replacement of the bottleneck in existing methods with a learned locality-sensitive hash function, 2) its use and direct optimization of non-orthogonal codebooks, which other methods cannot employ without a large slowdown, and 3) its use of an inexact estimator for computing sums. In future work, we plan to specialize our method for convolutions, implement it on GPUs, and integrate it into neural networks.

Chapter 5

Summary and Conclusion

In this thesis, we presented three algorithms for speeding up common tasks in real-world data analysis. We obtained significantly better empirical performance than existing methods in all cases, as well as theoretical guarantees where applicable.

In Chapter 2, we introduced Sprintz, an algorithm for compressing integer time series. Such time series are commonly produced by low-power sensors, and constitute a large and growing fraction of the world's data. The devices collecting these time series are often extremely power-constrained, making the high energy cost of data transmission a problem. On-device compression algorithms address this problem by allowing processors to transmit fewer bytes. Our algorithm combines a custom bit packing format with a learned prediction function to obtain state-of-the-art compression ratios and throughputs, while also using 100× less memory than competing algorithms.

In Chapter 3, we described Bolt, an algorithm for approximating scalar reductions on pairs of vectors, with a focus on Euclidean distances and dot products. We built on the classic Product Quantization [98] algorithm, designing a related algorithm that takes into account the characteristics of modern hardware. We demonstrated that our improved algorithm allows one to conduct approximate nearest neighbor and maximum inner product scans far more quickly than previously possible. In certain cases, it even allows one to approximately multiply matrices more quickly than state-of-the-art exact matrix multiplication routines, despite doing so with the

naive strategy of serially computing dot products. We showed that, under various assumptions, our method provably produces small approximation errors.

In Chapter 4, we built upon the ideas in Chapter 3 to produce MADDNESS, an approximate matrix multiplication algorithm that significantly outperforms existing methods. Several ideas contribute to this algorithm's performance. First, it replaces an exact search operation present in BOLT with a trained locality-sensitive hash function. Second, the hash function it uses is designed to execute quickly on SIMD or SIMT architectures. And finally, it relaxes constraints on its parameters that other methods cannot relax without an immense slowdown. In addition to showing that it works well in practice, we prove a bound on this method's generalization gap as a function of the training set size, the data's colinearity, and the method's hyperparameter values.

In future work, we intend to extend our approximate matrix multiplication algorithm to approximate convolutional layers in neural networks. We also plan to introduce faster training for this algorithm so that it can be trained alongside the associated neural network without becoming a bottleneck.

In addition to our contributions regarding particular problems, we believe this work also contains broader lessons regarding the design of practical algorithms for data analysis. First, our results suggest that, even when using commodity hardware and tackling well-studied problems, it is still sometimes possible to obtain 10× improvements or more. In order to do this, however, one typically needs to consider not merely an isolated subroutine, but the "full stack" from hardware capabilities to problem requirements. For example, an effort to implement a faster exact matrix multiplication function likely would have failed; however, our effort to exploit the tolerance of downstream tasks to *inexact* matrix products yielded enormous speedups. An important special case of considering the full stack is paying attention to the amount of accuracy required for a given problem. Because real-world data is often noisy, 32-bit floating point computation may be more precise than necessary.

A second lesson is that it can be useful to embed machine learning algorithms within other algorithms, even when the other algorithms are already lightweight.

While the practice of replacing heuristics with end-to-end optimization is now well-known within machine learning, it is not common to do so in the middle of a tight loop as we do throughout this work; in Sprintz, we perform gradient descent while compressing each block of samples; in Bolt, we perform a cluster assignment in the middle of a matrix-vector multiply; and in Maddness, we perform inference in a collection of decision tree models for every row in a matrix as part of a matrix multiply.

Finally, the efficacy of Bolt and Maddness suggests that categorical random variables hold a great deal of promise for replacing low-precision integer and floating point representations. We predict a future in which multiplexers replace many of the multiplication units in hardware accelerators.

In summary, this thesis has introduced state-of-the-art algorithms for several foundational tasks in modern data analaysis and machine learning pipelines. These algorithms are effective empirically, and, in some cases, also feature theoretical guarantees regarding their accuracy. We hope that these algorithms, and others inspired by them, will have a lasting impact on the costs of analyzing and learning from data.

Appendix A

Additional Theoretical Analysis of Bolt

In this section, we prove the results described in Section 3.3.3 and introduce a collection of less significant results.

A.1 Quantization Error

We begin by proving the bound on the errors introduced by quantizing the lookup tables.

A.1.1 Definitions

Let Q be the distribution of query subvectors $\boldsymbol{q}_m$ for lookup table m, X be the distribution of database subvectors $\boldsymbol{x}_m$ for this table, and Y be the scalar-valued distribution of distances within that table. That is,

$$p(Y = y) \triangleq \int_{Q,X} p(\boldsymbol{q}_m, \boldsymbol{x}_m) I\{d_m(\boldsymbol{q}_m, \boldsymbol{x}_m) = y\} \tag{A.1}$$

where $I\{\cdot\}$ is the indicator function. Recall that we seek to learn a quantization function $\beta_m : \mathbb{R} \to \{0, \ldots, 255\}$ of the form:

$$\beta_m(y) = \max(0, \min(255, \lfloor ay - b \rfloor)) \tag{A.2}$$

that minimizes the loss:

$$E_Y[(\hat{y} - y)^2] \tag{A.3}$$

where $\hat{y} \triangleq (\beta_m(y) + b)/a$ is the *reconstruction* of y.

Recall that we set $b = F^{-1}(\alpha)$ and $a = 255/(F^{-1}(1-\alpha) - b)$ for some suitable α. F^{-1} is the inverse CDF of Y, estimated empirically on a training set. The value of α is optimized using a simple grid search.

To analyze the performance of β_m from a theoretical perspective, let us define the following:

- Let $|\hat{y} - y|$ be the *quantization error*.
- Let B be the number of quantization bins. In practice, $B = 256$.
- Let $b_{min} \triangleq F^{-1}(\alpha)$ be the smallest value that can be quantized without clipping.
- Let $b_{max} \triangleq F^{-1}(1-\alpha)$ be the largest value that can be quantized without clipping.
- Let $\Delta \triangleq \frac{b_{max} - b_{min}}{B}$ be the width of each quantization bin.

Using these quantities, the quantization error for a given y value can be decomposed into:

$$|y - \hat{y}| = \begin{cases} b_{min} - y & \text{if } y \leq b_{min} \\ \Delta c(y) & \text{if } b_{min} < y \leq b_{max} \\ y - b_{max} & \text{if } y > b_{max} \end{cases} \tag{A.4}$$

where $c(y) = (y - \hat{y})/\Delta$ returns a value in $[0, 1)$ indicating where $\hat{y}$ lies within its

quantization bin. These three cases represent y being clipped at b_{min}, being rounded down to the nearest bin boundary, or being clipped at b_{max}, respectively.

It will also be helpful to define the following properties.

Definition A.1.1. *A random variable X is (l, h)-exponential if and only if:*

$$l < E[X] < h \tag{A.5}$$

$$p(X < \gamma) < \frac{1}{\sigma_X} e^{-(E[X]-\gamma)/\sigma_X},\ \gamma \leq l \tag{A.6}$$

$$p(X > \gamma) < \frac{1}{\sigma_X} e^{-(\gamma-E[X])/\sigma_X},\ \gamma \geq h \tag{A.7}$$

where σ_X is the standard deviation of X.

In words, X is (l, h)-exponential if its tails are bounded by exponential distributions. For appropriate l and h, Laplace, Exponential, Gaussian, and all subgaussian distributions are (l, h)-exponential.

A.1.2 Guarantees

Lemma A.1.1. *Let $p(Y < b_{min}) = 0$ and $p(Y > b_{max}) = 0$. Then $|\hat{y} - y| < \Delta$.*

Proof. The error $|\hat{y} - y| > \varepsilon$ can be decomposed according to (A.4). By assumption, the first and last terms in this decomposition, wherein Y clips, have probability 0. This leaves only:

$$|y - \hat{y}| = \Delta c(y) \tag{A.8}$$

where $0 \leq c(y) < 1$. For any value of $c(y)$, $|y - \hat{y}| < \Delta$. Intuitively, this means that if the distribution isn't clipped, the worst quantization error is the width of a quantization bin.

□

Theorem A.1.1 (Two-tailed generalization bound)**.** *If Y is (b_{min}, b_{max})-exponential,*

then

$$p(|y-\hat{y}|>\varepsilon) < \frac{1}{\sigma_Y}\left(e^{-(b_{max}-E[Y])/\sigma_Y} + e^{-(E[Y]-b_{min})/\sigma_Y}\right)e^{-\varepsilon/\sigma_Y} \tag{A.9}$$

for all $\varepsilon > \Delta$.

Proof. Using the decomposition in (A.4), we have:

$$\begin{aligned} p(|y-\hat{y}|>\varepsilon) = p(c(y)\Delta>\varepsilon)p(b_{min}<y\leq b_{max}) \\ + p(b_{min}-y>\varepsilon) \\ + p(y-b_{max}>\varepsilon). \end{aligned} \tag{A.10}$$

The first term corresponds to y being truncated within a bin, and the latter two correspond to y clipping. Since $0 \leq c(y) < 1$ and $p(b_{min} < y \leq b_{max}) \leq 1$, the first term can be bounded as:

$$p(c(y)\Delta>\varepsilon)p(b_{min}<y\leq b_{max}) < I\{\varepsilon<\Delta\} \tag{A.11}$$

The latter two terms can be bounded using the assumption that Y is (b_{min}, b_{max})-exponential:

$$p(b_{min}-y>\varepsilon) = p(y<b_{min}-\varepsilon) < \frac{1}{\sigma_Y}e^{-(E[Y]-b_{min}-\varepsilon)/\sigma_Y} \tag{A.12}$$

$$p(y-b_{max}>\varepsilon) = p(y>b_{max}+\varepsilon) < \frac{1}{\sigma_Y}e^{-(b_{max}+\varepsilon-E[Y])/\sigma_Y} \tag{A.13}$$

Combining (A.11)-(A.13), we have

$$p(|y-\hat{y}|>\varepsilon) < I\{\varepsilon<\Delta\} + \frac{1}{\sigma_Y}\left(e^{-(b_{max}+\varepsilon-E[Y])/\sigma_Y} + e^{-(E[Y]-b_{min}-\varepsilon)/\sigma_Y}\right) \tag{A.14}$$

When $\varepsilon \geq \Delta$, the first term is zero and we obtain (A.9). □

For ease of understanding, it is helpful to consider the case wherein b_{min} and b_{max} are symmetric about the mean. When this holds, the bound of Theorem A.1.1

simplifies to the more concise expression of Lemma A.1.2. This shows that the error probability decays exponentially with the number of standard deviations b_{min} and b_{max} are from the mean, as well as the size of ε (normalized by the standard deviation).

Lemma A.1.2 (Symmetric generalization bound)**.** *If z is a scalar such that Y is $(E[y] - z\sigma_Y, E[y] + z\sigma_Y)$-exponential, then*

$$p(|y - \hat{y}| > \varepsilon) < \frac{1}{\sigma_Y} 2e^{-(z+\varepsilon/\sigma_Y)} \tag{A.15}$$

where $\varepsilon > \Delta = 2z\sigma_Y/B$.

Proof. This follows immediately from Theorem A.1.1 using $b_{min} = E[y] - z\sigma_Y$, $b_{max} = E[y] + z\sigma_Y$. □

The bound of A.1.1 is effective when Y is roughly symmetric, but less so when Y is heavily skewed. Such skewness is sometimes present when estimating L_p distances. In the presence of severe skewness, $E[Y]$ is close to either b_{min} or b_{max}, and so one of the two exponentials in Equation A.14 approaches 1. Theorem A.1.3 describes a tighter bound for the case of right skew and a hard lower limit of 0, since this is often the distribution observed for L_p distances. The corresponding bound for left skew and a hard upper limit is similar, so we omit it. Note that this theorem is useful only if $b_{min} \approx \Delta$, but this is commonly the case when the L_p distances are highly skewed.

Lemma A.1.3 (One-tailed generalization bound)**.** *If Y be (b_{min}, b_{max})-exponential, with $p(Y < 0) = 0$, then*

$$p(|y - \hat{y}| > \varepsilon) < \frac{1}{\sigma_Y} e^{-(b_{max}+\varepsilon-E[Y])/\sigma_Y} \tag{A.16}$$

for all $\varepsilon > \max(\Delta, b_{min})$.

Proof. Using (A.10) with the fact that $\varepsilon > b_{min}$, we have:

$$\begin{aligned} p(|y - \hat{y}| > \varepsilon) = p(c(y)\Delta > \varepsilon) p(b_{min} < y \leq b_{max}) \\ + p(y - b_{max} > \varepsilon) \end{aligned} \tag{A.17}$$

Again applying the bounds from (A.11) and (A.13), we obtain (A.16). □

A.2 Dot Product Error

In this section, we bound the error in Bolt's approximate dot products. We also introduce a useful closed-form approximation that helps to explain the high performance of product quantization-based algorithms in general. We again begin with definitions before moving to the guarantees.

A.2.1 Definitions and Preliminaries

Definition A.2.1 (Codebook)**.** *A (K, J)-codebook $\mathcal{C}$ is an ordered collection of K vectors $\{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_K\} \in \mathbb{R}^J$. Each vector is referred to as a "centroid" or "codeword." The notation $\boldsymbol{c}_i$ denotes the ith centroid in the codebook.*

Definition A.2.2 (Codelist)**.** *A (K, M, J)-codelist $\boldsymbol{C}$ is an ordered collection of M $(K, J/M)$-codebooks. Because zero-padding is trivial and does not affect any relevant measure of accuracy, we assume that J is a multiple of M. The notation $\boldsymbol{c}_i^{(m)}$ denotes the ith centroid in the mth codebook. A codelist can be thought of (and stored) as a rank-3 tensor whose columns are codebooks, treated as row-major 2D arrays.*

Definition A.2.3 (Subvectors of a vector)**.** *Let $\boldsymbol{x} \in \mathbb{R}^J$ be a vector, let $M > 0$ be an integer, and let $L = J/M$. Then $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(M)}$ are the subvectors of $\boldsymbol{x}$, where $\boldsymbol{x}^{(m)} \in \mathbb{R}^L \triangleq x_{(k-1)L+1}, \ldots, x_L$. As with codelists, J is assumed to be a multiple of M.*

Definition A.2.4 (Encoding of a vector)**.** *Let $\boldsymbol{x} \in \mathbb{R}^J$ be a vector with subvectors $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(M)}$ and let $\boldsymbol{C}$ be a (K, M, J)-codelist. Then the encoding of $\boldsymbol{x}$ is the sequence of integers $a_1, \ldots, a_M$, $0 < a_m \leq M$ where*

$$a_m \triangleq \operatorname*{argmin}_i ||\boldsymbol{x}^{(m)} - \boldsymbol{c}_i^{(m)}||^2 \tag{A.18}$$

Definition A.2.5 (Reconstruction)**.** *Let $a_1, \ldots, a_M$, $0 < a_m \leq M$ be the encoding of some vector $\boldsymbol{x}$, and let $\boldsymbol{C}$ be a (K, M, J)-codelist. Then the concatenation of the*

vectors $\boldsymbol{c}_{a_1}^{(1)}, \boldsymbol{c}_{a_2}^{(2)}, \ldots, \boldsymbol{c}_{a_M}^{(M)}$ *is the reconstruction of* $\boldsymbol{x}$*, denoted* $\hat{\boldsymbol{x}}$*.*

Definition A.2.6 (Residuals)**.** *Let* $\hat{\boldsymbol{x}}$ *be the reconstruction of* $\boldsymbol{x}$*. Then* $\boldsymbol{r} \triangleq \boldsymbol{x} - \hat{\boldsymbol{x}}$ *is the residual vector for* $\boldsymbol{x}$*.*

Apart from these definitions, it is also necessary to establish several geometric properties of random encoded vectors in high-dimensional spaces.

Lemma A.2.1 (Dot product bias [22])**.** *Let* $\hat{\boldsymbol{x}}$ *be the reconstruction of* $\boldsymbol{x}$ *using codelist* $\boldsymbol{C}$*, and suppose that the centroids of all codebooks within* $\boldsymbol{C}$ *were learned using k-means. Then* $E[\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}] = 0$*.*

Lemma A.2.2 (Euclidean distance bias [98, 22])**.** *Let* $a_1, \ldots, a_M$*,* $0 < a_m \leq M$ *be the encoding of some vector* $\boldsymbol{x}$ *using codelist* $\boldsymbol{C}$ *and* $\hat{\boldsymbol{x}}$ *be the reconstruction of* $\boldsymbol{x}$*. Further suppose that the centroids of all codebooks within* $\boldsymbol{C}$ *were learned using k-means. Then:*

$$E[\|\boldsymbol{q} - \boldsymbol{x}\|^2 - \|\boldsymbol{q} - \hat{\boldsymbol{x}}\|^2] = \sum_{m=1}^{M} \mathrm{MSE}(a_m, m) \tag{A.19}$$

where $\mathrm{MSE}(a_m, m)$ *is the expected squared Euclidean distance between centroid* $\boldsymbol{c}_{a_m}^{(m)}$ *and the subvectors assigned to it by k-means. I.e.,*

$$\mathrm{MSE}(a_m, m) \triangleq E_X[\|\boldsymbol{c}_{a_m}^{(m)} - \boldsymbol{x}^{(m)}\|^2],\ a_m = \underset{i}{\mathrm{argmin}} \|\boldsymbol{c}_i^{(m)} - \boldsymbol{x}^{(m)}\|^2 \tag{A.20}$$

Lemma A.2.3 (Area of a hyperspherical cap (Li. 2011 [120]))**.** *Suppose that a hypersphere in* $\mathbb{R}^J$ *with radius* r *is cut into two caps by a hyperplane, with the angle* $\theta, 0 \leq \theta \leq \frac{\pi}{2}$ *defining the radius of the smaller cap. Then the area of the smaller cap is given by*

$$A_J(r) = \frac{1}{2} A_J^s(r) I_{\sin^2(\theta)} \left(\frac{J-1}{2}, \frac{1}{2} \right) \tag{A.21}$$

where $A_J^s(r)$ *is the area of the hyperpsphere and* $I_x(\alpha, \beta)$ *denotes the regularized incomplete beta function (i.e., the CDF of a* $\mathrm{Beta}(\alpha, \beta)$ *distribution).*

Lemma A.2.4 (Angle between random vectors)**.** *Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^J$ be vectors such that $\frac{\boldsymbol{x}^\top \boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}\boldsymbol{x}$ is sampled uniformly from the surface of the unit hypersphere S^{J-1}, and let $\theta \triangleq \arccos(\frac{\boldsymbol{x}^\top \boldsymbol{y}}{\|\boldsymbol{x}\| \cdot \|\boldsymbol{y}\|})$ be the angle between $\boldsymbol{x}$ and $\boldsymbol{y}$. Then for $0 \leq a \leq \frac{\pi}{2}$,*

$$p(|\theta| \geq a) = I_{\sin^2(a)}\left(\frac{J-1}{2}, \frac{1}{2}\right) \tag{A.22}$$

Proof. Since the angle between $\boldsymbol{x}$ and $\boldsymbol{y}$ is independent of their norms, assume without loss of generality that $\boldsymbol{x}$ and $\boldsymbol{y}$ have been scaled such that $\|\boldsymbol{x}\| = \|\boldsymbol{y}\| = 1$. For a given $\boldsymbol{x}$, the set of $\boldsymbol{y}$ vectors such that $\cos(\theta) \geq a, \theta \leq \frac{\pi}{2}$ is exactly the set of vectors comprising a hyperspherical cap of S^{J-1} with radius defined by a. Because the projection onto $\boldsymbol{x}$ of $\boldsymbol{y}$ has probability mass uniformly distributed across S^{J-1}, the probability that $\boldsymbol{y}$ lies within this cap is equal to the area of the cap divided by the area of the hypersphere. Using Lemma A.2.3, this ratio is given by:

$$\frac{1}{2} I_{\sin^2(a)}\left(\frac{J-1}{2}, \frac{1}{2}\right) \tag{A.23}$$

By symmetry, this is also $p(\theta < -a)$. Summing the probabilities of these two events yields (A.22).

□

Lemma A.2.5 (Gaussian approximation to angle between random vectors)**.** *Let $\boldsymbol{x}$, $\boldsymbol{q}$, and θ be defined as in Lemma A.2.4. Then*

$$p(cos(\theta) > a) \approx \frac{1}{2} + \frac{1}{2}\operatorname{erf}\left(-cos(\theta)\sqrt{\frac{J}{2}}\right) \tag{A.24}$$

Proof. Using the identity $I_z(\alpha, \alpha) = \frac{1}{2} I_{4z(1-z)}(\alpha, \frac{1}{2})$ [1, Eq. 8.17.6], we can rewrite (A.23) as:

$$I_\phi(\frac{J-1}{2}, \frac{J-1}{2}) \tag{A.25}$$

where $\phi = \frac{1}{2}(1 - \cos(\theta))$. Recall that a Beta$(\alpha, \beta)$ distribution can be approximated

by a normal distribution with:

$$\mu = \frac{\alpha}{\alpha + \beta} \tag{A.26}$$

$$\sigma^2 = \frac{\alpha\beta}{(\alpha + \beta)^2(1 + \alpha + \beta)} \tag{A.27}$$

Using $\alpha = \beta = \frac{J-1}{2}$, this yields

$$\mu = \frac{1}{2} \tag{A.28}$$

$$\sigma^2 = \frac{\left(\frac{J-1}{2}\right)^2}{4\left(\frac{J-1}{2}\right)^2\left(1 + 2\frac{J-1}{2}\right)} = \frac{1}{4J} \tag{A.29}$$

Further recall that the CDF of a normal distribution with a given mean μ and variance σ^2 is given by

$$\Phi(a) = \frac{1}{2} + \frac{1}{2}\,\mathrm{erf}\left(\frac{a - \mu}{\sigma\sqrt{2}}\right) \tag{A.30}$$

Substituting (A.28) and (A.29) into (A.30), we obtain

$$I_\phi(\frac{J-1}{2}, \frac{J-1}{2}) \approx \mathrm{erf}((\phi - \frac{1}{2})\sqrt{2J}) \tag{A.31}$$

Finally, substituting $\frac{1}{2}(1 - \cos(\theta))$ for ϕ yields (A.24). □

Lemma A.2.6 (Gaussian PDF approximation)**.** *Let $\boldsymbol{x}$, $\boldsymbol{q}$, and θ be defined as in Lemma A.2.4 and let the dimensionality J be sufficiently large that the Beta distribution can be accurately approximated with a normal distribution. Then,*

$$\cos(\theta) \sim \mathcal{N}(0, J^{-1}) \tag{A.32}$$

Proof. Writing (A.24) in the form of (A.30) gives

$$\mu = 0 \tag{A.33}$$

$$\sigma^2 = \frac{1}{J} \tag{A.34}$$

Because (A.24) is the CDF of a Gaussian random variable with this μ and σ^2, the PDF is given by $\mathcal{N}(0, J^{-1})$. □

A.2.2 Guarantees

We now prove several bounds on the errors caused by product quantization using an arbitrary number of subvectors. We begin with no distributional assumptions, and then prove increasingly tight bounds as more assumptions are added.

Lemma A.2.7 (Worst-case dot product error)**.** *Let $\hat{\boldsymbol{x}}$ be the reconstruction of $\boldsymbol{x}$ and let $\boldsymbol{q} \in \mathbb{R}^J$ be a vector. Then $|\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}| < \|\boldsymbol{q}\| \cdot \|\boldsymbol{r}\|$, where $\boldsymbol{r}$ is the residual vector $\boldsymbol{x} - \hat{\boldsymbol{x}}$.*

Proof. This follows immediately from application of the Cauchy-Schwarz inequality.

$$|\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}| = |\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top (\boldsymbol{x} - \boldsymbol{r})| = |\boldsymbol{q}^\top \boldsymbol{r}| \leq \|\boldsymbol{q}\| \cdot \|\boldsymbol{r}\| \tag{A.35}$$

□

If we are willing to make the extremely pessimistic assumption that the cosine of the angle between $\boldsymbol{q}$ and $\boldsymbol{r}$ is uniformly distributed, a tighter bound (and indeed, an exact expression for the error probability) is possible (Theorem A.2.1). This assumption is pessimistic because angles close to 0, which yield smaller errors, are much more probable in high dimensions.

Theorem A.2.1 (Pessimistic dot product error)**.** *Let θ denote the angle between $\boldsymbol{r}$ and some vector $\boldsymbol{q}$, and suppose that $\cos(\theta) \sim Unif(-1, 1)$. Then*

$$p(|\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}| > \varepsilon) = \max(0, 1 - \frac{\varepsilon}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{r}\|}) \tag{A.36}$$

Proof. Simple algebra shows that

$$\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}} = \boldsymbol{q}^\top (\hat{\boldsymbol{x}} + \boldsymbol{r}) - \boldsymbol{q}^\top \hat{\boldsymbol{x}} = \boldsymbol{q}^\top \boldsymbol{r} = \|\boldsymbol{q}\| \cdot \|\boldsymbol{r}\| \cos(\theta) \tag{A.37}$$

Since $\cos(\theta) \sim Unif(-1, 1)$, we have that $|\cos(\theta)| \sim Unif(0, 1)$, and therefore

$$p(|\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}| > \varepsilon) = p(|\|\boldsymbol{q}\| \cdot \|\boldsymbol{r}\| \cos(\theta)| > \varepsilon) \tag{A.38}$$

$$= p\left(|\cos(\theta)| > \frac{\varepsilon}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{r}\|}\right) \tag{A.39}$$

$$= \max\left(0, 1 - \frac{\varepsilon}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{r}\|}\right) \tag{A.40}$$

□

The assumption that the cosine similarity of vectors is uniform can be replaced with the more optimistic assumption that the errors in quantizing each subvector are independent, yielding Theorem A.2.2.

Theorem A.2.2 (Dot product error with independent subspaces)**.** *Let $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(M)}$ be the subvectors of $\boldsymbol{x}$, let $\hat{\boldsymbol{x}}^{(1)}, \ldots, \hat{\boldsymbol{x}}^{(M)}$ be the subvectors of $\hat{\boldsymbol{x}}$, and let $\boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(M)}$ be the subvectors of an arbitrary vector $\boldsymbol{q} \in R^J$. Further let $\boldsymbol{r}^{(m)} \triangleq \boldsymbol{x}^{(m)} - \hat{\boldsymbol{x}}^{(m)}$, and assume that the values of $\|\boldsymbol{r}^{(m)}\|$ are independent for all k. Then*

$$p(|\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}| \geq \varepsilon) \leq 2 \exp\left(\frac{-\varepsilon^2}{2 \sum_{m=1}^{M} (\|\boldsymbol{q}^{(m)}\| \cdot \|\boldsymbol{r}^{(m)}\|))^2}\right) \tag{A.41}$$

Proof. The quantity $\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}$ can be expressed as the sum

$$\sum_{m=1}^{M} \boldsymbol{q}^{(m)\top} (\boldsymbol{x}^{(m)} - \hat{\boldsymbol{x}}^{(m)}) = \sum_{m=1}^{M} \boldsymbol{q}^{(m)\top} \boldsymbol{r}^{(m)} \tag{A.42}$$

Each element of this sum can be viewed as an independent random variable v_k. By Lemma A.2.7, $-\|\boldsymbol{q}^{(m)}\| \cdot \|\boldsymbol{r}^{(m)}\| < v_k < \|\boldsymbol{q}^{(m)}\| \cdot \|\boldsymbol{r}^{(m)}\|$. The inequality (A.41) then follows from Hoeffding's inequality. □

This bound assumes the worst-case distribution of errors for each subvector. If we instead assume that the errors are random as defined in Lemma A.2.4, it is possible to obtain not only a bound, but a closed-form expression for the probability of a given error.

Theorem A.2.3 (Dot product error approximation)**.** *Let* $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(M)}$, $\hat{\boldsymbol{x}}^{(1)}, \ldots, \hat{\boldsymbol{x}}^{(M)}$, $\boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(M)}$, $\boldsymbol{r}^{(1)}, \ldots, \boldsymbol{r}^{(M)}$ *be defined as in Theorem A.2.2. Suppose that each* $(\boldsymbol{q}^{(m)}, \boldsymbol{r}^{(m)})$ *satisfy the conditions of Lemma A.2.4 and the values of* $\boldsymbol{q}^{(m)\top}\boldsymbol{r}^{(m)}$ *are independent across all* k*. Then*

$$p(\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}}) \approx \mathcal{N}(0, \sigma^2) \tag{A.43}$$

where $\sigma^2 \triangleq \frac{1}{L}\sum_{m=1}^{M} \|\boldsymbol{q}^{(m)}\|^2 \cdot \|\boldsymbol{r}^{(m)}\|^2$.

Proof. Applying Lemma A.2.6 to a given $(\boldsymbol{q}^{(m)}, \boldsymbol{r}^{(m)})$, we have the approximation:

$$\cos(\theta_m) \sim \mathcal{N}(0, L^{-1}) \tag{A.44}$$

where $\theta_m \triangleq \frac{\boldsymbol{q}^{(m)\top}\boldsymbol{r}^{(m)}}{\|\boldsymbol{q}^{(m)}\| \cdot \|\boldsymbol{r}^{(m)}\|}$. Recalling from (A.37) that $\boldsymbol{q}^\top \boldsymbol{x} - \boldsymbol{q}^\top \hat{\boldsymbol{x}} = \|\boldsymbol{q}\| \cdot \|\boldsymbol{r}\| \cos(\theta)$, this implies that

$$\boldsymbol{q}^{(m)\top}\boldsymbol{x}^{(m)} - \boldsymbol{q}^{(m)\top}\hat{\boldsymbol{x}}^{(m)} \sim \mathcal{N}(0, \sigma_m^2) \tag{A.45}$$

where $\sigma_m^2 = \frac{\|\boldsymbol{q}^{(m)}\|^2 \cdot \|\boldsymbol{r}^{(m)}\|^2}{L}$. Because the errors from each subspace are independent, one can sum their variances to obtain (A.43). □

This approximation is optimistic if the codebooks are trained from k-means using the Euclidean distance, since the residuals' directions are unlikely to be uniformly distributed on the unit hypersphere. However, if the centroids are trained under the Mahalanobis distance [78, 22], then this approximation may be pessimistic. This is because the latter approach tends to concentrate $\cos(\theta)$ around 0 (by construction), which yields even smaller variances in each subspace.

A.2.3 Euclidean Distance Error

The guarantees in this section closely parallel those of the previous section, so we state them without comment.

Theorem A.2.4 (Worst-case L_2 error)**.** *Let $\hat{\boldsymbol{x}}$ be the reconstruction of $\boldsymbol{x}$ and let $\boldsymbol{q} \in \mathbb{R}^J$ be a vector. Then $|\|\boldsymbol{q}-\boldsymbol{x}\| - \|\boldsymbol{q}-\hat{\boldsymbol{x}}\|| < \|\boldsymbol{r}\|$.*

Proof. This follows immediately from application of the triangle inequality.

$$\|\boldsymbol{q}-\boldsymbol{x}\| - \|\boldsymbol{r}\| \leq \|\boldsymbol{q}-\hat{\boldsymbol{x}}\| = \|\boldsymbol{q}-\boldsymbol{x}+\boldsymbol{r}\| \leq \|\boldsymbol{q}-\boldsymbol{x}\| + \|\boldsymbol{r}\| \tag{A.46}$$

and therefore

$$|\|\boldsymbol{q}-\boldsymbol{x}\| - \|\boldsymbol{q}-\hat{\boldsymbol{x}}\|| \leq \|\boldsymbol{r}\| \tag{A.47}$$

□

Theorem A.2.5 (Pessimistic L_2 error)**.** *Let θ denote the angle between $\boldsymbol{r}$ and some vector $\boldsymbol{q}$, and suppose that $\cos(\theta) \sim Unif(-1,1)$. Then*

$$p(|\|\boldsymbol{q}-\boldsymbol{x}\|^2 - \|\boldsymbol{q}-\hat{\boldsymbol{x}}\|^2| > \varepsilon) = \max\left(0, \frac{\|\boldsymbol{r}\|^2-\varepsilon}{2\|\boldsymbol{r}\|\|\boldsymbol{q}-\boldsymbol{x}\|} - 1\right) \tag{A.48}$$

Proof. Using the Law of Cosines, we have

$$\begin{aligned}\|\boldsymbol{q}-\boldsymbol{x}\|^2 &= \|\boldsymbol{q}-\hat{\boldsymbol{x}}\|^2 + \|\boldsymbol{x}-\hat{\boldsymbol{x}}\|^2 - 2\|\boldsymbol{q}-\hat{\boldsymbol{x}}\|\|\boldsymbol{x}-\hat{\boldsymbol{x}}\|\cos(\theta) \\ &= \|\boldsymbol{q}-\hat{\boldsymbol{x}}\|^2 + \|\boldsymbol{r}\|^2 - 2\|\boldsymbol{q}-\hat{\boldsymbol{x}}\|\|\boldsymbol{r}\|\cos(\theta)\end{aligned} \tag{A.49}$$

and therefore

$$\|\boldsymbol{q}-\boldsymbol{x}\|^2 - \|\boldsymbol{q}-\hat{\boldsymbol{x}}\|^2 = \|\boldsymbol{r}\|^2 - 2\|\boldsymbol{r}\|\|\boldsymbol{q}-\hat{\boldsymbol{x}}\|\cos(\theta) \tag{A.50}$$

This implies that

$$\begin{aligned}p(\|\boldsymbol{q}-\boldsymbol{x}\|^2 - \|\boldsymbol{q}-\hat{\boldsymbol{x}}\|^2 > \varepsilon) &= p(\|\boldsymbol{r}\|^2 - 2\|\boldsymbol{r}\|\|\boldsymbol{q}-\hat{\boldsymbol{x}}\|\cos(\theta) > \varepsilon) \\ &= p\left(\frac{\|\boldsymbol{r}\|^2-\varepsilon}{2\|\boldsymbol{r}\|\|\boldsymbol{q}-\hat{\boldsymbol{x}}\|} > \cos(\theta)\right) \\ &= \frac{1}{2}\max\left(0, \frac{\|\boldsymbol{r}\|^2-\varepsilon}{2\|\boldsymbol{r}\|\|\boldsymbol{q}-\hat{\boldsymbol{x}}\|} - 1\right)\end{aligned} \tag{A.51}$$

Equation (A.48) follows by symmetry.

□

Theorem A.2.6 (L_2 error with independent subspaces)**.** *Let $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(M)}$ be the subvectors of $\boldsymbol{x}$, let $\hat{\boldsymbol{x}}^{(1)}, \ldots, \hat{\boldsymbol{x}}^{(M)}$ be the subvectors of $\hat{\boldsymbol{x}}$, and let $\boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(M)}$ be the subvectors of an arbitrary vector $\boldsymbol{q} \in R^J$. Further let $\boldsymbol{r}^{(m)} \triangleq \boldsymbol{x}^{(m)} - \hat{\boldsymbol{x}}^{(m)}$, and assume that the values of $\|\boldsymbol{q}^{(m)} - \boldsymbol{x}^{(m)}\|^2 - \|\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)}\|^2$ are independent for all k.*

$$p(|\|\boldsymbol{q} - \boldsymbol{x}\|^2 - \|\boldsymbol{q} - \hat{\boldsymbol{x}}\|^2| > \varepsilon) \leq 2 \exp\left(\frac{-\varepsilon^2}{2\sum_{m=1}^{M} \|\boldsymbol{r}^{(m)}\|^4}\right) \tag{A.52}$$

Proof. The quantity $\|\boldsymbol{q} - \boldsymbol{x}\|^2 - \|\boldsymbol{q} - \hat{\boldsymbol{x}}\|^2$ can be expressed as the sum

$$\sum_{m=1}^{M} \|\boldsymbol{q}^{(m)} - \boldsymbol{x}^{(m)}\|^2 - \|\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)}\|^2 \tag{A.53}$$

By assumption, each element of this sum can be viewed as an independent random variable v_k. By Lemma A.2.4, $-\|\boldsymbol{r}^{(m)}\|^2 \leq v_k \leq \|\boldsymbol{r}^{(m)}\|^2$. Assuming that one adds in the bias correction described in Lemma A.2.2, one can apply Hoeffding's inequality to obtain (A.41). □

Theorem A.2.7 (L_2 error approximation)**.** *Let $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(M)}$, $\hat{\boldsymbol{x}}^{(1)}, \ldots, \hat{\boldsymbol{x}}^{(M)}$, $\boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(M)}$, $\boldsymbol{r}^{(1)}, \ldots, \boldsymbol{r}^{(M)}$ be defined as in Theorem A.2.6. Suppose that each pair $(\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)}, \boldsymbol{r}^{(m)})$ satisfy the conditions of Lemma A.2.4 and the values of $(\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)})^\top \boldsymbol{r}^{(m)}$ are independent across all k.*

$$p(\|\boldsymbol{q} - \boldsymbol{x}\|^2 - \|\boldsymbol{q} - \hat{\boldsymbol{x}}\|^2) \approx \mathcal{N}(\|r\|^2, \sigma^2) \tag{A.54}$$

where $\sigma^2 \triangleq 4\|\boldsymbol{r}\|^2 \|\boldsymbol{q} - \boldsymbol{x}\|^2 L^{-1}$.

Proof. Applying Lemma A.2.6 to a given $(\boldsymbol{q}^{(m)}, \boldsymbol{r}^{(m)})$, we have the approximation:

$$\cos(\theta_m) \sim \mathcal{N}(0, L^{-1}) \tag{A.55}$$

where $\theta_m \triangleq \frac{\boldsymbol{q}^{(m)^\top}\boldsymbol{r}^{(m)}}{\|\boldsymbol{q}^{(m)}\|\cdot\|\boldsymbol{r}^{(m)}\|}$. Further recall from (A.50) that

$$\|\boldsymbol{q}^{(m)} - \boldsymbol{x}^{(m)}\|^2 - \|\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)}\|^2 = \|\boldsymbol{r}^{(m)}\|^2 - 2\|\boldsymbol{r}^{(m)}\|\|\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)}\|\cos(\theta_m) \quad \text{(A.56)}$$

Combining (A.56) and (A.55) yields

$$\|\boldsymbol{q}^{(m)} - \boldsymbol{x}^{(m)}\|^2 - \|\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)}\|^2 \sim \mathcal{N}(\|\boldsymbol{r}^{(m)}\|^2, \sigma_m^2) \quad \text{(A.57)}$$

where $\sigma_m^2 \triangleq 4\|\boldsymbol{r}^{(m)}\|^2\|\boldsymbol{q}^{(m)} - \hat{\boldsymbol{x}}^{(m)}\|^2 L^{-1}$. Because the errors from each subspace are independent, one can sum their variances to obtain (A.54).

□

Appendix B

Additional Theoretical Analysis of Maddness

B.1 Proof of Generalization Guarantee

In this section, we prove Theorem B.1, restated below for convenience.

Theorem (Generalization Error of MADDNESS)**.** *Let $\mathcal{D}$ be a probability distribution over $\mathbb{R}^D$ and suppose that* MADDNESS *is trained on a matrix $\tilde{\boldsymbol{A}} \in R^{N \times D}$ whose rows are drawn independently from $\mathcal{D}$ and with maximum singular value bounded by σ_A. Let C be the number of codebooks used by* MADDNESS *and $\lambda > 0$ the regularization parameter used in the ridge regression step. Then for any $\boldsymbol{b} \in \mathbb{R}^D$, any $\boldsymbol{a} \sim \mathcal{D}$, and any $0 < \delta < 1$, we have with probability at least $1 - \delta$ that*

$$\mathbb{E}_{\mathcal{D}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] \leq \mathbb{E}_{\tilde{\boldsymbol{A}}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] + \frac{C \sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \left(\frac{1}{256} + \frac{8 + \sqrt{C(4 \lceil \log_2(D) \rceil + 256) \log 2 - \log \delta}}{\sqrt{2n}} \right) \tag{B.1}$$

where $\mathcal{L}(\boldsymbol{a}, \boldsymbol{b}) \triangleq |\boldsymbol{a}^\top \boldsymbol{b} - \alpha f(g(\boldsymbol{a}), h(\boldsymbol{b})) - \boldsymbol{\beta}|$, α is the scale used for quantizing the lookup tables, and $\boldsymbol{\beta}$ is the constants used in quantizing the lookup tables plus the debiasing constant of Section 4.4.4.

The proof relies on the observation that MADDNESS's training procedure can be

decomposed into two sequential subroutines: `Maddness-Build-Tree`, which learns the function $g(\boldsymbol{a})$ by constructing a binary decision tree, and `Maddness-Regress`, which learns the function $h(\boldsymbol{b})$ by optimizing a prototype matrix $\boldsymbol{P}$ such that $g(\tilde{\boldsymbol{A}})\boldsymbol{P} \approx \tilde{\boldsymbol{A}}$. This observation allows us to prove B.1 by first providing a guarantee for `Maddness-Regress` for a fixed `Maddness-Build-Tree` hypothesis, and then union bounding over the hypothesis space for `Maddness-Build-Tree`. Bounding the size of the hypothesis space is straightforward (Lemma B.1.1), so the bulk of this section focuses on providing a guarantee for `Maddness-Regress`. We must also prove a bound on the loss contributed by quantizing the lookup tables array $\boldsymbol{P}^\top \boldsymbol{b}$.

Lemma B.1.1 (Number of Hypotheses for `Maddness-Build-Tree`)**.** *Let C be the number of codebooks used by* MADDNESS *and let D be the number of columns in the matrix $\tilde{\boldsymbol{A}}$ on which* MADDNESS *is trained. Then there are at most $2^{C(4\lceil \log_2(D) \rceil + 256)}$ unique trees that `Maddness-Build-Tree` can generate.*

Proof. `Maddness-Build-Tree` learns four sets of parameters for each of the C trees it produces: split indices, split offsets, split scales, and split values.

There are four split indices per tree because there is one for each of the tree's four levels. Each index requires $\lceil \log_2(D) \rceil$ bits to store, so the split indices require a total of $4 \lceil \log_2(D) \rceil$ bits per tree. For each split index, there is one split offset and scale, used to map floating point data in an arbitrary range to the interval $[0, 255]$ to match up with the 8-bit split values.

The offsets require at most 25 bits each for 32-bit floating point data, since the low 7 bits can be dropped without affecting the post-scaling quantized output. The scales are constrained to be powers of two, and so require at most 9 bits for non-subnormal 32-bit floating point inputs (which have one sign bit and eight exponent bits). The offsets and scales together therefore contribute $4(25 + 9) = 136$ bits per tree.

There are 15 split values because there is one for the root of each tree, then two for the second level, four for the third, and eight for the fourth. Each split value is stored using eight bits, so each tree requires $15 \cdot 8 = 120$ bits for split values. The total number of bits used for all trees is therefore $C(4 \lceil \log_2(D) \rceil + 256)$. Note that the

constant 256 being a power of two is just an accident of floating point formats. The claimed hypothesis count follows from the number of expressible hypotheses being at most 2 to the power of the largest number of bits used to store any hypothesis. □

We now turn our attention to bounding the errors of the regression component of training. Our strategy for doing so is to bound the largest singular value of the learned matrix of prototypes $\boldsymbol{P}$. Given such a bound, the norms of both $g(\boldsymbol{a})^\top \boldsymbol{P}$ and $\boldsymbol{P}^\top \boldsymbol{b}$ can be bounded.

Lemma B.1.2 (Regularized Pseudoinverse Operator Norm Bound)**.** *Let $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ be an arbitrary matrix with finite elements. Then every singular value σ_i of the matrix $\boldsymbol{Z} \triangleq (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top$, $\lambda > 0$ is at most $\frac{1}{2\sqrt{\lambda}}$.*

Proof. Let $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ be the singular value decomposition of $\boldsymbol{X}$. Then we have

$$\begin{aligned}
\boldsymbol{Z} &= (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top && \text{(B.2)} \\
&= (\boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top + \lambda \boldsymbol{I})^{-1} \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^\top && \text{(B.3)} \\
&= (\boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^\top + \lambda \boldsymbol{I})^{-1} \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^\top && \text{(B.4)} \\
&= (\boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^\top + \boldsymbol{V}\lambda\boldsymbol{I}\boldsymbol{V}^\top)^{-1} \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^\top && \text{(B.5)} \\
&= (\boldsymbol{V}\boldsymbol{\Sigma}_\lambda\boldsymbol{V}^\top)^{-1} \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^\top && \text{(B.6)} \\
&= \boldsymbol{V}\boldsymbol{\Sigma}_\lambda^{-1}\boldsymbol{V}^\top \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^\top && \text{(B.7)} \\
&= \boldsymbol{V}\boldsymbol{\Sigma}_\lambda^{-1}\boldsymbol{\Sigma}\boldsymbol{U}^\top && \text{(B.8)} \\
&= \boldsymbol{V}\boldsymbol{\Sigma}'\boldsymbol{U}^\top && \text{(B.9)}
\end{aligned}$$

where $\boldsymbol{\Sigma}_\lambda \triangleq \boldsymbol{\Sigma}^2 + \lambda \boldsymbol{I}$ and $\boldsymbol{\Sigma}' \triangleq (\boldsymbol{\Sigma}^2 + \lambda \boldsymbol{I})^{-1}\Sigma$. Step B.5 follows from the equality $\boldsymbol{V}\lambda\boldsymbol{I}\boldsymbol{V}^\top = \lambda \boldsymbol{V}\boldsymbol{V}^\top = \lambda \boldsymbol{I}$. Because the matrices $\boldsymbol{V}$ and $\boldsymbol{U}^\top$ are orthonormal and $\boldsymbol{\Sigma}'$ is diagonal, the singular values of $\boldsymbol{Z}$ are equal to the diagonal entries of $\boldsymbol{\Sigma}'$. Each entry σ_i' is equal to

$$\sigma_i' = \frac{\sigma_i}{\sigma_i^2 + \lambda}. \tag{B.10}$$

This expression attains its maximal value of $\frac{1}{2\sqrt{\lambda}}$ when $\sigma_i^2 = \lambda$. □

Lemma B.1.3 (Ridge Regression Singular Value Bound)**.** *Let $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{Y} \in \mathbb{R}^{D \times M}$ be arbitrary matrices and let $\boldsymbol{W} \triangleq (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{Y}$, $\lambda > 0$ be the ridge regression weight matrix. Then $\|\boldsymbol{W}\|_\infty \leq \frac{\|\boldsymbol{Y}\|_\infty}{2\sqrt{\lambda}}$, where $\|\cdot\|_\infty$ denotes the largest singular value.*

Proof. Observe that $\boldsymbol{W} = \boldsymbol{Z}\boldsymbol{Y}$, where $\boldsymbol{Z} \triangleq (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top$. Then by applying Lemma B.1.2 and recalling that Schatten norms are submultiplicative, we have

$$\|\boldsymbol{W}\|_\infty \leq \|\boldsymbol{Z}\|_\infty \|\boldsymbol{Y}\|_\infty \leq \frac{\|\boldsymbol{Y}\|_\infty}{2\sqrt{\lambda}}. \tag{B.11}$$

□

Lemma B.1.4 (Bound on MADDNESS Embedding Norm)**.** *Let $\boldsymbol{g} = g(\boldsymbol{a})$ be the encoding of an arbitrary vector $\boldsymbol{a}$ using C codebooks and let $\boldsymbol{P}$ be the prototype matrix learned by* MADDNESS *using training matrix $\tilde{\boldsymbol{A}}$ with ridge regression parameter $\lambda > 0$. Then*

$$\|\boldsymbol{g}^\top \boldsymbol{P}\|_2 \leq \frac{C}{2\sqrt{\lambda}} \|\tilde{\boldsymbol{A}}\|_\infty \tag{B.12}$$

where $\|\tilde{\boldsymbol{A}}\|_\infty$ denotes the largest singular value of $\tilde{\boldsymbol{A}}$.

Proof. We have

$$\begin{aligned} \|\boldsymbol{g}^\top \boldsymbol{P}\|_2 &\leq \|\boldsymbol{g}\|_2 \|\boldsymbol{P}\|_\infty && \text{(B.13)} \\ &= C\|\boldsymbol{P}\|_\infty && \text{(B.14)} \\ &\leq \frac{C}{2\sqrt{\lambda}} \|\tilde{\boldsymbol{A}}\|_\infty. && \text{(B.15)} \end{aligned}$$

The first step follows from Cauchy-Schwarz. The second follows from $\boldsymbol{g}$ being zero except for exactly C ones. The last is an application of Lemma B.1.3. □

Lemma B.1.5 (Maximum Table Quantization Loss)**.** *Let $\hat{\boldsymbol{a}} = g(\boldsymbol{a})^\top \boldsymbol{P}$, where $g(\cdot)$ and P are trained using C codebooks and ridge regression penalty $\lambda > 0$ on a matrix $\tilde{\boldsymbol{A}}$ with maximum singular value at most σ_A, and $\boldsymbol{a} \in \mathbb{R}^D$ is an arbitrary vector.*

Then for any vector $\boldsymbol{b} \in \mathbb{R}^D$, $|\hat{\boldsymbol{a}}^\top \boldsymbol{b} - \hat{\boldsymbol{y}}| < \frac{C\sigma_A \|\boldsymbol{b}\|_2}{512\sqrt{\lambda}}$, *where* $\hat{\boldsymbol{y}} \triangleq \alpha g(\boldsymbol{a})^\top g(\boldsymbol{b}) + \boldsymbol{\beta}$ *is* MADDNESS*'s approximation to* $\boldsymbol{a}^\top \boldsymbol{b}$, *with* α *and* $\boldsymbol{\beta}$ *the scale and offsets used to quantize the lookup tables.*

Proof. If MADDNESS had infinite-precision lookup tables, $\hat{\boldsymbol{y}}$ would exactly equal $\hat{\boldsymbol{a}}^\top \boldsymbol{b}$. We therefore need only bound the error introduced by the quantization. By Lemma B.1.4, $\|\hat{\boldsymbol{a}}\|_2 \leq \frac{C\sigma_A}{2\sqrt{\lambda}}$. This implies that

$$\|\hat{\boldsymbol{a}}^\top \boldsymbol{b}\| \leq \frac{C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \tag{B.16}$$

and therefore

$$\frac{-C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \leq \hat{\boldsymbol{a}}^\top \boldsymbol{b} \leq \frac{C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}}. \tag{B.17}$$

For each of the C codebooks, this means that the value to be quantized lies in the interval $[\frac{-\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}}, \frac{\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}}]$ of width $\frac{\sigma_A \|\boldsymbol{b}\|_2}{\sqrt{\lambda}}$. Because MADDNESS quantizes the lookup tables such that largest and smallest entries for any row of $\boldsymbol{P}$ are linearly mapped to 255.5 and -0.5,[1] respectively, the worst-case quantization error is when the quantized value lies exactly between two quantization levels.We therefore need to compute the largest possible gap between a value and its quantization. Using 256 quantization levels, the largest possible gap is $1/(256/.5) = 1/512$ of the interval width. Multiplying by the above interval width yields a maximum quantization error for a given codebook of $\frac{\sigma_A \|\boldsymbol{b}\|_2}{512\sqrt{\lambda}}$. Because the errors in each subspace may not agree in sign, their sum is an upper bound on the overall quantization error. □

At this point, we have all of the pieces necessary to prove a generalization guarantee for `Maddness-Regress` save one: a theorem linking the norms of the various vectors and matrices involved to a probabilistic guarantee. Kakade et al. [101] provide such a gaurantee, based on Rademacher complexity [28].

[1] We use 255.5 and -0.5 rather than 255 and 0 because the latter only guarantees that a point is within 1/510 of the interval width, not 1/512. This is not an important choice and either option would be fine.

Theorem B.1.1 ([101], Corollary 5)**.** *Let $\mathcal{F} = \{\boldsymbol{w}^\top \boldsymbol{x} : \|\boldsymbol{w}\|_2 \leq W\}$ be the class of linear functions with bounded L_2 norms, let $\mathcal{S}$ be a set of n samples drawn i.i.d. from some distribution $\mathcal{D}$ over the L_2 ball of radius X, and let $\mathcal{L}(f), f \in \mathcal{F}$ be a loss function with Lipschitz constant L. Then for any $0 < \delta < 1$, it holds with probability at least $1 - \delta$ over the sample $\mathcal{S}$ that*

$$\mathbb{E}_{\mathcal{D}}[\mathcal{L}(f)] \leq \mathbb{E}_S[\mathcal{L}(f)] + \frac{LXW}{\sqrt{2n}} \left(8 + \sqrt{-log(\delta)}\right). \tag{B.18}$$

We can now obtain our desired guarantee for the regression step.

Lemma B.1.6 (Generalization Error of `Maddness-Regress`)**.** *Let $\mathcal{D}$ be a probability distribution over $\mathbb{R}^D$ and suppose that* Maddness *is trained on a matrix $\tilde{\boldsymbol{A}} \in R^{N \times D}$ whose rows are drawn independently from $\mathcal{D}$ and with maximum singular value bounded by σ_A. Let C be the number of codebooks used by* Maddness *and $\lambda > 0$ the regularization parameter used in the ridge regression step. Further let $g(\boldsymbol{a})$ be a fixed (data-independent) function and $\mathcal{L}(\boldsymbol{a}, \boldsymbol{b}) \triangleq |\boldsymbol{a}^\top \boldsymbol{b} - f(g(\boldsymbol{a}), h(\boldsymbol{b}))|$. Then for all vectors $\boldsymbol{b}$, any vector $\boldsymbol{a} \sim \mathcal{D}$, and any $0 < \delta < 1$, we have with probability at least $1 - \delta$ that*

$$\mathbb{E}_{\mathcal{D}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] \leq \mathbb{E}_{\tilde{\boldsymbol{A}}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] + \frac{C\sigma_A \|\boldsymbol{b}\|_2}{512\sqrt{\lambda}} + \frac{C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{2n\lambda}} \left(8 + \sqrt{-log(\delta)}\right). \tag{B.19}$$

Proof. The output of `Maddness-Regress` can be decomposed into

$$\hat{\boldsymbol{y}} \triangleq f(g(\boldsymbol{a}), h(\boldsymbol{b})) = \boldsymbol{g}^\top \boldsymbol{P} \boldsymbol{b} + \varepsilon + \zeta \tag{B.20}$$

where $\boldsymbol{g} = g(\boldsymbol{a})$, $\boldsymbol{P}$ is the matrix of prototypes, ε is data-independent noise from the averaging process[2], and ζ is noise from quantizing the lookup table entries. By Lemma B.1.5, $\zeta \leq \frac{C\sigma_A \|\boldsymbol{b}\|_2}{512\sqrt{\lambda}}$ (accounting for the second term in Equation B.19). We therefore need only obtain a guarantee for $|\boldsymbol{g}^\top \boldsymbol{P} \boldsymbol{b} - \boldsymbol{a}^\top \boldsymbol{b}|$. Defining $\boldsymbol{w} \triangleq \boldsymbol{P}\boldsymbol{b}$, we see

[2]We continue to make the assumption that the least significant bits of the lookup table entries are independent Bernoulli(0.5) random variables, which is nearly true in practice. Even if this assumption does not hold, this noise does not contribute to the generalization gap unless it differs between train and test sets.

that `Maddness-Regress` is a linear model, and therefore subject to Theorem B.1.1. Given an upper bound on the Lipschitz constant of the loss, a bound on the L_2 norm of $\boldsymbol{g}$, and a bound on the L_2 norm of $\boldsymbol{w}$, we can apply this theorem. The Lipschitz constant for the absolute loss is 1. The L_2 norm of $\boldsymbol{g}$ is exactly C. The L_2 norm of $\boldsymbol{w}$ can be bounded as

$$\|\boldsymbol{w}\|_2 = \|\boldsymbol{P}\boldsymbol{b}\|_2 \leq \|\boldsymbol{P}\|_\infty \|\boldsymbol{b}\|_2 \leq \frac{\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \tag{B.21}$$

using Lemma B.1.3. □

Using this lemma, the proof of Theorem B.1 is immediate; we begin with Lemma B.1.6 and simply union bound over all $2^{C(4\lceil \log_2(D)\rceil + 120)}$ hypotheses from Lemma B.1.1.

B.2 Aggregation Using Pairwise Averages

In this section, we analyze the errors introduced by using averaging instructions rather than upcasting and computing sums exactly.

Definition B.2.1 (Averaging Integer Sum Estimator)**.** *Let $\boldsymbol{x} \in \{0,1\}^C, C \,\%\, U = 0, U = 2^p, p \geq 0$. The Averaging Integer Sum Estimator (AISE) $\hat{s}(\boldsymbol{x})$ is defined as:*

$$\hat{s}(\boldsymbol{x}) \triangleq \sum_{k=1}^{C/U} \hat{s}_U(\boldsymbol{x}_{i_k:j_k}) \tag{B.22}$$

$$\hat{s}_U(\boldsymbol{x}) \triangleq \begin{cases} x_1 & \boldsymbol{x} \in \mathbb{R}^1 \\ \lfloor \frac{1}{2}(\hat{s}_U(\boldsymbol{x}_{left}) + \hat{s}_U(\boldsymbol{x}_{right}) + 1) \rfloor & otherwise \end{cases} \tag{B.23}$$

where $i_k = (k-1)\cdot U + 1, j_k = i_U + U$ and $\boldsymbol{x}_{left}$ and $\boldsymbol{x}_{right}$ denote vectors formed by taking the initial and final $D/2$ indices of a given $\boldsymbol{x} \in \mathbb{R}^D$.

Definition B.2.2 (Pairwise Integer Sum and Sum Estimator)**.** *For integers a and b,*

define

$$s(a,b) \triangleq a+b \tag{B.24}$$

$$\hat{s}(a,b) \triangleq 2\mu(a,b) \tag{B.25}$$

where $\mu(a,b) \triangleq \lfloor \frac{1}{2}(a+b+1) \rfloor$.

Lemma B.2.1 (Bias when averaging one pair)**.** *Consider two scalars a and b, with* $a, b \overset{iid}{\sim}$ *Bernoulli(.5). Define* $\varepsilon(a,b) \triangleq \hat{s}(a,b) - s(a,b)$*. Then*

$$E[\varepsilon(a,b)] = \frac{1}{2}.$$

Proof. The proof follows immediately from considering the four equiprobable realizations of the pair a, b. In the cases $(0,0)$ and $(1,1)$, $2\mu(a,b) = s(a,b)$. In the cases $(0,1)$ and $(1,0)$, $2\mu(a,b) = 2$, while $s(a,b) = 1$. □

Lemma B.2.2 (Variance of error when averaging one pair)**.** *Consider two scalars a and b, with* $a, b \overset{iid}{\sim}$ *Bernoulli(.5). Then*

$$E[\varepsilon(a,b)^2] - E[\varepsilon(x,y)]^2 = \frac{1}{4}.$$

Proof. Using Lemma B.2.1, the above can be rewritten as:

$$E[\varepsilon(a,b)^2] = \frac{1}{2}.$$

The proof then follows by again considering the four equiprobable cases as in Lemma B.2.1. In the cases $(0,0)$ and $(1,1)$, $\varepsilon(a,b)^2 = 0$. In the cases $(0,1)$ and $(1,0)$, $(2\hat{s}(a,b) - s(a,b))^2 = (2-1)^2 = 1$. □

Lemma B.2.3 (Bias of AISE within a subspace)**.** *Suppose that the scalar elements* x_i *of* $\boldsymbol{x}$ *are drawn from independent Bernoulli(.5) distributions. Then*

$$E[s_U(\boldsymbol{x}) - \hat{s}_U(\boldsymbol{x})] = U \log_2(U)/4. \tag{B.26}$$

Proof. Observe that the computation graph can be cast as a balanced binary tree with U leaves and each parent equal to the integer average of its children. Consider the bias introduced at each level t of the tree, where $t = 0$ corresponds to the leaves and $t = \log_2(U)$ corresponds to the root. The expected error $E[\xi(t, n)]$ introduced at a node n in level $t > 0$ is given by

$$E[\xi(t, n)] = \frac{1}{2} \cdot 2^{t-1} \tag{B.27}$$

where the $\frac{1}{2}$ follows from Lemma B.2.1 and the scale 2^{t-1} is the number of leaf nodes to which the bias is applied. E.g., adding one to the estimated average of four leaf nodes would increase the estimated sum by four. Since there are $U \cdot 2^{-t}$ nodes per level, this means that the total expected error introduced at level t is $\frac{1}{2} \cdot 2^{t-1} \cdot 2^{-t} = \frac{1}{4}$. Summing from $t = 1$ to $t = \log_2(U)$ completes the proof of the expected error. Note that $t = 0$ is omitted since the leaf nodes are not the result of averaging operations and so introduce no error.

□

Theorem B.2.1 (Bias of AISE over all subspaces)**.** *Suppose that the scalar elements x_i of $\boldsymbol{x}$ are drawn from independent Bernoulli(.5) distributions. Then*

$$E[s(\boldsymbol{x}) - \hat{s}(\boldsymbol{x})] = C \log_2(U)/4. \tag{B.28}$$

Proof. This follows immediately from Lemma B.2.3, the fact that the overall sum is estimated within each of C/U subspaces of size U, and the assumption that the errors in each subspace are independent. □

Note that the assumption that the elements are independent is not especially strong in reality. This is because this section focuses on the effects on the least significant bits (which are the ones affected by each averaging operation), and the least significant bit does tend to be nearly uniformly random in a great deal of real-world data. An exception might be data with an extremely large number of zeros, which would have zeros as low bits much more often than ones. However, in the

particular case of MADDNESS, this is less of a concern; the elements being averaged are inner products between vectors, and therefore tend to be nonzero even when many or most of the individual scalars within the vectors are zero.

Appendix C

Additional Method and Experiment Details for Maddness

C.1 Quantizing Lookup Tables

Since the lookup table entries naturally occupy more than 8 bits even for 8-bit data, some means of quantizing these entries is necessary to enable vectorization.[1] Unfortunately, existing quantization methods are not applicable to our problem setting. The scheme of Bolt requires knowledge of $\boldsymbol{B}$ at training time, while the scheme of Quick ADC [17] and Quicker ADC [18] is only applicable for nearest neighbor search. We instead use the following approach, where $\boldsymbol{T} \in \mathbb{R}^{M \times C \times K}$ is the tensor of lookup tables for all M columns of $\boldsymbol{B}$, $\boldsymbol{T}^q$ is the quantized version of $\boldsymbol{T}$, $\boldsymbol{\delta} \in \mathbb{R}^C$ is a vector of table-specific offsets, and α^{-1} is an overall scale factor:

$$\boldsymbol{\delta}_c \triangleq \min_{m,k} \ \boldsymbol{T}_{m,c,k} \tag{C.1}$$

$$\alpha^{-1} \triangleq 2^l,\ l = \max_c \left\lfloor \log_2 \left(\frac{255}{\max_{m,k}(\boldsymbol{T}_{m,c,k} - \delta_c)} \right) \right\rfloor . \tag{C.2}$$

$$\boldsymbol{T}^q_{m,c,k} \triangleq \alpha^{-1}(\boldsymbol{T}_{m,c,k} - \delta_c) \tag{C.3}$$

[1] The table entries are products of 8-bit values, and therefore require 16 bits.

The α used here is the same as that in Equation 4.2, and the matrix β in Equation 4.2 has entries equal to $\sum_c \boldsymbol{\delta}_c$ (plus the debiasing constant from our averaging-based aggregation).

C.2 Quantization and MaddnessHash

The use of at most 16 leaves is so that the resulting codes use 4 bits. This allows the use of these same shuffle instructions to accelerate the table lookups as in Bolt.

The only subtlety in vectorizing our hash function is that one must execute line 4 using shuffle instructions such as `vpshufb` on x86 and `vtbl` on ARM. In order to do this, the split values and scalars x_{j^t} must be 8-bit integers. We quantize them by learning for each split index j a pair of scalars (γ_j, δ_j), where

$$\delta_j \triangleq \min_i \boldsymbol{v}_i^j \tag{C.4}$$

$$\gamma_j \triangleq 2^l, l = \left\lfloor \log_2 \left(\frac{255}{\max_i \boldsymbol{v}_i^j - \delta_j} \right) \right\rfloor \tag{C.5}$$

This restriction of γ_j to powers of two allows one to quantize x_{j^t} values with only shifts instead of multiplies. The $\boldsymbol{v}$ values can be quantized at the end of the training phase, while the x_{j^t} values must be quantized within Algorithm 4 before line 5.

C.3 Subroutines for Training MaddnessHash

The `optimal_split_threshold` algorithm (Algorithm 6) finds the best threshold at which to split a bucket within a given dimension. To do this, it uses the `cumulative_sse` function (Algorithm 7) to help evaluate the loss associated with the resulting child buckets.

These algorithms exploit the fact that the sum of squared errors can be computed using only the sum of values and the sum of squared values, both of which can be updated in $O(1)$ time when a vector is moved from one side of the split to the other.

Algorithm 6 Optimal Split Threshold Within a Bucket

Input: bucket $\mathcal{B}$, index j
$\boldsymbol{X} \leftarrow$ `as_2d_array`$(\mathcal{B})$
$\boldsymbol{X}^{sort} =$ `sort_rows_based_on_col`$(\boldsymbol{X}, j)$
`sses_head` $\leftarrow$ `cumulative_sse`$(\boldsymbol{X}^{sort}, \texttt{false})$
`sses_tail` $\leftarrow$ `cumulative_sse`$(\boldsymbol{X}^{sort}, \texttt{true})$
`losses` $\leftarrow$ `sses_head`
$\texttt{losses}_{1:N-1} \leftarrow \texttt{losses}_{1:N-1} + \texttt{sses\_tail}_{2:N}$
$n^* \leftarrow \text{argmin}_n \texttt{losses}_n$
return $(\boldsymbol{X}^{sort}_{n^*,\,j} + \boldsymbol{X}^{sort}_{n^*+1,\,j})/2$, $\texttt{losses}_{n^*}$

C.4 Additional Experimental Details

In this section, we discuss details of our experiments that are necessary for reproducibility but that would have distracted from the body of Section 4.5.

SparsePCA Details

We took steps to ensure that SparsePCA's results were not hampered by insufficient hyperparameter tuning. First, for each matrix product, we tried a range of lambda values which we found to encompass the full gamut of nearly 0% to nearly 100% sparsity: $\lambda \in 2^i, i \in \{-5, -4, -3, -2, -1, 0, 1, 2, 3\}$. Second, because different sparsity patterns may yield different execution times, we report the best time from any of ten random matrices of the same size and at most the same sparsity, rather than the time

Algorithm 7 Cumulative SSE

Input: 2D array $\boldsymbol{X}$, boolean `reverse`
$N, D \leftarrow \texttt{shape}(\boldsymbol{X})$
if `reverse` **then**
    $\forall_i$ $\texttt{swap}(\boldsymbol{X}_{i,d}, \boldsymbol{X}_{N-i+1,d})$
$\texttt{out} \leftarrow \texttt{empty}(N)$
$\texttt{cumX} \leftarrow \texttt{empty}(D)$
$\texttt{cumX2} \leftarrow \texttt{empty}(D)$
// Initialize first row of output and cumulative values
$\texttt{out}_1 \leftarrow 0$
for $d \leftarrow 1$ **to** D **do**
    $\texttt{cumX}_d \leftarrow X_{1,d}$
    $\texttt{cumX2}_d \leftarrow (X_{1,d})^2$
// Compute remaining output rows
for $n \leftarrow 2$ **to** N **do**
    $\texttt{out}_n \leftarrow 0$
    for $d \leftarrow 1$ **to** D **do**
        $\texttt{cumX}_d \leftarrow \texttt{cumX}_d + X_{1,d}$
        $\texttt{cumX2}_d \leftarrow \texttt{cumX2}_d + (X_{1,d})^2$
        $\texttt{out}_n \leftarrow \texttt{out}_n + \texttt{cumX2}_d - (\texttt{cumX}_d \times \texttt{cumX}_d / n)$
return `out`

from the single matrix SparsePCA produced for a given (d, λ) pair. Finally and most importantly, we plot only the Pareto frontier of (speed, quality) pairs produced for a given matrix multiply. I.e., we let SparsePCA cherrypick its best results on each individual matrix multiply.

C.4.1 Exact Matrix Multiplication

We also implemented our own matrix product function specialized for tall, skinny matrices. In all cases, we report the timings based on the faster of this function and Eigen's matrix multiply function for a given matrix product.

C.4.2 Additional Baselines

We also tested Frequent Directions / Fast Frequent Directions [121, 72, 53], many variations of the sampling method of [57], projections using Gaussian random matrices [94, 48], projection using orthogonalized Gaussian random matrices [99], projection

using matrices of scaled i.i.d. Rademacher random variables [10], projection using orthonormalized matrices of Rademacher random variables, the co-occuring directions sketch [139], OSNAP [140], Product Quantization [98], and Optimized Product Quanization [70].

The poor performance of many of these methods is unsurprising in our setting. Given that we have access to a training set on which to learn the true principal components, the Eckart-Young-Mirsky theorem [62] indicates that PCA should outperform any other individual matrix sketching method employing dense projection matrices, at least in the limit of infinite training data. Also, since PQ and OPQ use 256 dense centroids (except in the Bolt/ QuickerADC variations), it is also impossible for them to perform well when $\min(D, M)$ is not significantly larger than 256.

C.4.3 UCR Time Series Archive

We omitted datasets with fewer than 128 training examples, since it is not possible for Stochastic Neighbor Compression to draw 128 samples without replacement in this case.

In addition to being a large, public corpus of over a hundred datasets from a huge variety of different domains, the UCR Time Series Archive also has the advantage that it can be used to produce matrix multiplication tasks of a fixed size. This is necessary for meaningful comparison of speed versus accuracy tradeoffs across datasets. We constructed training and test matrices $\tilde{\boldsymbol{A}}$ and $\boldsymbol{A}$ by resampling each time series in each dataset's train and test set to a length of 320 (the closest multiple of 32 to the median length of 310). We obtained the matrix $\boldsymbol{B}$ for each dataset by running Stochastic Neighbor Compression [111] on the training set with an RBF kernel of bandwidth one. We set the number of returned neighbors to 128 (results with 64 and 256 were similar), yielding a $\boldsymbol{B}$ matrix of size 320×128. Since different datasets have different test set sizes, all results are for a standardized test set size of 1000 rows. We wanted the length to be a multiple of 32 since existing methods operate best with sizes that are either powers of two or, failing that, multiples of large powers of two.

We approximate Euclidean distances using the identity $\|\boldsymbol{x}-\boldsymbol{y}\|_2^2 = \|\boldsymbol{x}\|_2^2 - 2\boldsymbol{x}^\top\boldsymbol{y} +$

$\|\boldsymbol{y}\|_2^2$. We approximate only the inner products $\boldsymbol{x}^\top \boldsymbol{y}$, since $\|\boldsymbol{y}\|_2^2$ can be precomputed for fixed exemplars $\boldsymbol{y}$ and $\|\boldsymbol{x}\|_2^2$ does not affect the class prediction.

C.4.4 Caltech101

We only extracted valid windows—i.e., never past the edge of an image. We extracted the windows in CHW order, meaning that scalars from the same color channel were placed at contiguous indices. The "first" images are based on filename in lexicographic order.

We used pairs of filters because using a single filter would mean timing a matrix-vector product instead of a matrix-matrix product.

To allow meaningful speed comparisons across images, we resized and center cropped each image to 224×224 as commonly done in image classification pipelines [85, 86, 91]. We then extracted sliding windows of the appropriate size and used each flattened window as one row of $\tilde{\boldsymbol{A}}$ or $\boldsymbol{A}$. We similarly flattened the filters, with each set of coefficients forming one column of $\boldsymbol{B}$. In both cases, $\boldsymbol{B}$ has two columns—this is because using a single filter would mean timing a matrix-vector product instead of a matrix-matrix product. Two columns also made sense since Sobel filters are often used in horizontal and vertical pairings, and Gaussian filters are often used together to perform difference-of-Gaussians transforms.

Even though the RGB values at each position are naturally unsigned 8-bit integers, we allowed rival methods to operate on them as 32-bit floating point numbers, without including the conversion when timing them. Because it only requires checking whether values are above a threshold, MADDNESS can operate on 8-bit data directly.

C.4.5 Additional Results

In Section 4.5, we showed the classification accuracy as a function of wall time for the CIFAR-10 and CIFAR-100 softmax classifiers. In Figure C-1, we instead show normalized mean squared error versus time. In Figure C-2, we show accuracy versus number of operations performed, where one operation is either one multiply-add or

one table lookup, depending on the method. The first figure illustrates that NMSE is closely related to classification accuracy, but with imperfect NMSE still yielding excellent accuracy in many cases. The second figure shows that our method's superior results are not merely caused by the use of faster CPU instructions, but also by the use of fewer basic operations at the algorithm level.

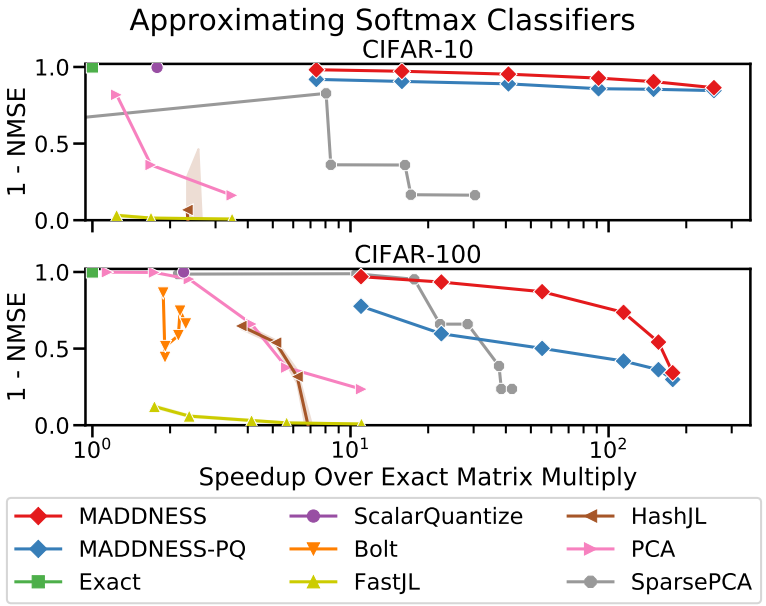

Figure C-1: Maddness achieves a far better speed versus squared error tradeoff than any existing method when approximating two softmax classifiers. These results parallel the speed versus classification accuracy results, except that the addition of our ridge regression is much more beneficial on CIFAR-100.

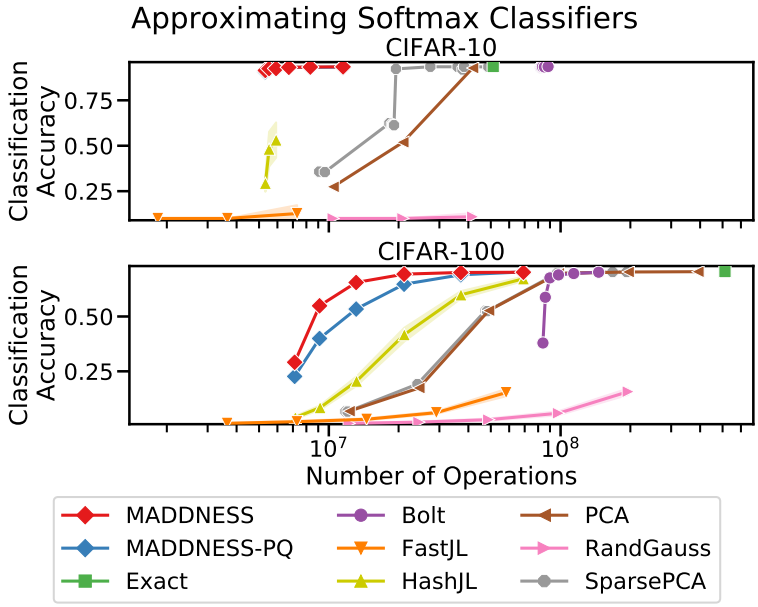

Figure C-2: Maddness also achieves a far better speed versus accuracy tradeoff when speed is measured as number of operations instead of wall time. Fewer operations with a high accuracy (up and to the left) is better.

Bibliography

[1] *NIST Digital Library of Mathematical Functions*. http://dlmf.nist.gov/, Release 1.0.14 of 2016-12-21. F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller and B. V. Saunders, eds.

[2] Information technology – generic coding of moving pictures and associated audio information – part 7: Advanced audio coding (aac), 2006. `https://www.iso.org/standard/43345.html`.

[3] Digi-key electronics, 2017. `https://www.digikey.com/products/en/integrated-circuits-ics/data-acquisition-analog-to-digital-converters-adc/700`.

[4] Faiss: A library for efficient similarity search. `https://engineering.fb.com/data-infrastructure/faiss-a-library-for-efficient-similarity-search/`, March 2017.

[5] Intel quark microcontrollers, 2017. `https://www.intel.com/content/www/us/en/embedded/products/quark/overview.html`.

[6] Apple watch. the number one smartwatch in the world. `https://www.apple.com/watch`, 2020.

[7] Fitbit official site. `https://www.fitbit.com/`, 2020.

[8] Misfit wearables. `https://www.misfit.com/`, 2020.

[9] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[10] Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 274–281, 2001.

[11] Nadav Aharony, Wei Pan, Cory Ip, Inas Khayal, and Alex Pentland. Social fmri: Investigating and shaping social mechanisms in the real world. *Pervasive and Mobile Computing*, 7(6):643–659, 2011.

[12] Nir Ailon and Bernard Chazelle. The Fast Johnson-Lindenstrauss Transform and Approximate Nearest Neighbors. *SIAM Journal on Computing (SICOMP)*, 39(1):302–322, 2009.

[13] Jyrki Alakuijala and Zoltan Szabadka. Brotli compressed data format. Technical report, 2016.

[14] Michael P Andersen and David E Culler. Btrdb: Optimizing storage system design for timeseries processing. In *FAST*, pages 39–52, 2016.

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

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

[17] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Accelerated nearest neighbor search with quick adc. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, pages 159–166, 2017.

[18] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Quicker adc: Unlocking the hidden potential of product quantization with simd. *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[19] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

[20] Apple. Apple lossless audio codec, 2011. `https://github.com/macosforge/alac`.

[21] Saad Arrabi and John Lach. Adaptive lossless compression in wireless body sensor networks. In *Proceedings of the Fourth International Conference on Body Area Networks*, page 19. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[22] Artem Babenko, Relja Arandjelović, and Victor Lempitsky. Pairwise quantization. *arXiv preprint arXiv:1606.01550*, 2016.

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

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

[25] Artem Babenko and Victor Lempitsky. Tree Quantization for Large-Scale Similarity Search and Classification. *CVPR*, pages 1–9, 2015.

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

[27] Amir H Bakhtiary, Agata Lapedriza, and David Masip. Speeding up neural networks for large scale classification using wta hashing. *arXiv preprint arXiv:1504.07488*, 2015.

[28] Peter L Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 3(Nov):463–482, 2002.

[29] S Beckett. Influxdb, 2017. `https://influxdata.com`.

[30] Davis W Blalock and John V Guttag. Extract: Strong examples from weakly-labeled sensor data. In *Proceedings of IEEE ICDM*, pages 799–804. IEEE, 2016.

[31] Davis W Blalock and John V Guttag. Bolt: Accelerated data mining with fast vector compression. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 727–735. ACM, 2017.

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

[33] Tulika Bose, Soma Bandyopadhyay, Sudhir Kumar, Abhijan Bhattacharyya, and Arpan Pal. Signal characteristics on sensor data compression in iot-an investigation. In *Sensing, Communication, and Networking (SECON), 2016 13th Annual IEEE International Conference on*, pages 1–6. IEEE, 2016.

[34] Thomas Boutell. Png (portable network graphics) specification version 1.0. 1997.

[35] Maxim Buevich, Anne Wright, Randy Sargent, and Anthony Rowe. Respawn: A distributed multi-resolution time-series datastore. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 288–297. IEEE, 2013.

[36] J Camacho, AK Smilde, E Saccenti, and JA Westerhuis. All sparse pca models are wrong, but some are useful. part i: Computation of scores, residuals and explained variance. *Chemometrics and Intelligent Laboratory Systems*, 196:103907, 2020.

[37] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn Keogh. isax 2.0: Indexing and mining one billion time series. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 58–67. IEEE, 2010.

[38] Beidi Chen, Tharun Medini, and Anshumali Shrivastava. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *arXiv preprint arXiv:1903.03129*, 2019.

[39] Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *ICML*, pages 2285–2294, 2015.

[40] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[41] Josh Coalson. Flac-free lossless audio codec, 2008. `http://flac.sourceforge.net`.

[42] Yann Collet. Finite state entropy. `https://github.com/Cyan4973/FiniteStateEntropy`.

[43] Yann Collet. Lz4–extremely fast compression, 2017. `https://github.com/Cyan4973/lz4`.

[44] Yann Collet. Zstandard - fast real-time compression algorithm, 2017. `https://facebook.github.io/zstd/`.

[45] DWARF Debugging Information Format Committee et al. Dwarf debugging information format, version 4. *Free Standards Group*, 2010.

[46] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. A sparse johnson: Lindenstrauss transform. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 341–350, 2010.

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

[48] M. Datar, N. Immorlica, Piotr Indyk, and V.S. Mirrokni. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pages 253–262, 2004.

[49] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. The ucr time series classification archive, October 2018. `https://www.cs.ucr.edu/~eamonn/time_series_data_2018/`.

[50] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[51] Thomas Dean, Mark A Ruzon, Mark Segal, Jonathon Shlens, Sudheendra Vijayanarasimhan, and Jay Yagnik. Fast, accurate detection of 100,000 object classes on a single machine. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1814–1821, 2013.

[52] Janez Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30, 2006.

[53] Amey Desai, Mina Ghashami, and Jeff M Phillips. Improved practical matrix sketching with guarantees. *IEEE Transactions on Knowledge and Data Engineering*, 28(7):1678–1690, 2016.

[54] L Peter Deutsch. Deflate compressed data format specification version 1.3. 1996.

[55] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

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

[57] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast Monte Carlo Algorithms for Matrices I: Approximating Matrix Multiplication. *SIAM Journal on Computing*, 36(1):132–157, January 2006.

[58] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast Monte Carlo Algorithms for Matrices II: Computing a Low-Rank Approximation to a Matrix. *SIAM Journal on Computing*, 36(1):158–183, January 2006.

[59] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast Monte Carlo Algorithms for Matrices III: Computing a Compressed Approximate Matrix Decomposition. *SIAM Journal on Computing*, 36(1):184–206, January 2006.

[60] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

[61] Sanghamitra Dutta, Viveck Cadambe, and Pulkit Grover. Short-dot: Computing large linear transforms distributedly using coded short dot products. In *Advances In Neural Information Processing Systems*, pages 2100–2108, 2016.

[62] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[63] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. A time-series compression technique and its application to the smart grid. *The VLDB Journal*, 24(2):193–218, 2015.

[64] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In *2004 conference on computer vision and pattern recognition workshop*, pages 178–178. IEEE, 2004.

[65] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629, 2015.

[66] Simon Fothergill, Helena M Mentis, Pushmeet Kohli, and Sebastian Nowozin. Instructing people for training gestural interactive systems. In Joseph A Konstan, Ed H Chi, and Kristina Höök, editors, *CHI*, pages 1737–1746. ACM, 2012.

[67] Deena P. Francis and Kumudha Raimond. An improvement of the parameterized frequent directions algorithm. *Data Mining and Knowledge Discovery*, 32(2):453–482, March 2018.

[68] Deena P. Francis and Kumudha Raimond. A practical streaming approximate matrix multiplication algorithm. *Journal of King Saud University - Computer and Information Sciences*, September 2018.

[69] Jean-Loup Gailly and Mark Adler. The gzip home page, 2003. `https://www.gzip.org/`.

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

[71] Yonatan Geifman. cifar-vgg, 3 2018. `https://github.com/geifmany/cifar-vgg`.

[72] Mina Ghashami, Edo Liberty, Jeff M. Phillips, and David P. Woodruff. Frequent Directions: Simple and Deterministic Matrix Sketching. *SIAM Journal on Computing*, 45(5):1762–1792, January 2016.

[73] Solomon Golomb. Run-length encodings. *IEEE transactions on information theory*, 12(3):399–401, 1966.

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

[75] Google. Protocol buffers encoding, 2001. `https://developers.google.com/protocol-buffers/docs/encoding#types`.

[76] Gael Guennebaud, Benoit Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[77] SH Gunderson. Snappy: A fast compressor/decompressor, 2015. `https://code.google.com/p/snappy`.

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

[79] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization.

[80] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. Protonn: Compressed and accurate knn for resource-scarce devices. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1331–1340. JMLR. org, 2017.

[81] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.

[82] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[83] Mark A Hanson, Harry C Powell Jr, Adam T Barth, Kyle Ringgenberg, Benton H Calhoun, James H Aylor, and John Lach. Body area sensor networks: Challenges and opportunities. *Computer*, 42(1), 2009.

[84] Brian Hawkins. Kairos db: Fast time series database on cassandra. `https://github.com/kairosdb/kairosdb`.

[85] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[86] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[87] Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. *arXiv preprint arXiv:1611.01600*, 2016.

[88] B Hu, Y Chen, and E Keogh. Time series classification under more realistic assumptions. In *SDM 2013*, pages 578–586, Philadelphia, PA, 2013. Society for Industrial and Applied Mathematics.

[89] Bing Hu, Yanping Chen, Jesin Zakaria, Liudmila Ulanova, and Eamonn Keogh. Classification of multi-dimensional streaming time series by weighting each classifier's track record. In *2013 IEEE 13th International Conference on Data Mining*, pages 281–290. IEEE, 2013.

[90] Bing Hu, Thanawin Rakthanmanon, Yuan Hao, Scott Evans, Stefano Lonardi, and Eamonn Keogh. Discovering the intrinsic cardinality and dimensionality of time series using mdl. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 1086–1091. IEEE, 2011.

[91] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[92] Zengfeng Huang. Near Optimal Frequent Directions for Sketching Dense and Sparse Matrices. *Journal of Machine Learning Research*, 20(1):23, February 2019.

[93] Nguyen Quoc Viet Hung, Hoyoung Jeung, and Karl Aberer. An evaluation of model-based approaches to sensor data compression. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2434–2447, 2013.

[94] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[95] Texas Instruments. 2.4-ghz bluetoothÂő low energy system-on-chip, 2013. `http://www.ti.com/lit/ds/symlink/cc2540.pdf`.

[96] Texas Instruments. Cc2640 simplelink bluetooth wireless mcu, 2016. `http://www.ti.com/lit/ds/swrs176b/swrs176b.pdf`.

[97] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

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

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

[100] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

[101] Sham M Kakade, Karthik Sridharan, and Ambuj Tewari. On the complexity of linear prediction: Risk bounds, margin bounds, and regularization. In *Advances in neural information processing systems*, pages 793–800, 2009.

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

[103] E J Keogh, Q Zhu, B Hu, Yuan Hao, Xiaopeng Xi, Li Wei, and C A Ratanamahatana. The UCR Time Series Classification/Clustering Homepage, 2011.

[104] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems*, 3(3):263–286, 2001.

[105] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Sigmod Record*, 30(2):151–162, 2001.

[106] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An online algorithm for segmenting time series. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 289–296. IEEE, 2001.

[107] Eamonn Keogh, Jessica Lin, and Ada Fu. Hot sax: Efficiently finding the most unusual time series subsequence. In *Data mining, fifth IEEE international conference on*, pages 8–pp. Ieee, 2005.

[108] Daya Khudia, Protonu Basu, and Summer Deng. Open-sourcing fbgemm for state-of-the-art server-side inference, 2018.

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

[110] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[111] Matt Kusner, Stephen Tyree, Kilian Weinberger, and Kunal Agrawal. Stochastic neighbor compression. In *International Conference on Machine Learning*, pages 622–630, 2014.

[112] Anastasios Kyrillidis, Michail Vlachos, and Anastasios Zouzias. Approximate Matrix Multiplication with Application to Linear Embeddings. *arXiv:1403.7683 [cs, math, stat]*, March 2014. arXiv: 1403.7683.

[113] Kasper Green Larsen and Jelani Nelson. The johnson-lindenstrauss lemma is optimal for linear dimensionality reduction. *arXiv preprint arXiv:1411.2404*, 2014.

[114] Eugene Lazin. Akumuli time-series database. `https://akumuli.org`.

[115] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[116] Daniel Lemire. A better alternative to piecewise linear time series segmentation. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 545–550. SIAM, 2007.

[117] Daniel Lemire. Microbenchmarking calls for idealized conditions, 2018. `https://lemire.me/blog/2018/01/16/microbenchmarking-calls-for-idealized-conditions/`.

[118] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[119] Peter Li and Michael Kasparian. Atlas wearables. `https://atlaswearables.com/`, 2020.

[120] Shengqiao Li. Concise formulas for the area and volume of a hyperspherical cap. *Asian Journal of Mathematics and Statistics*, 4(1):66–70, 2011.

[121] Edo Liberty. Simple and Deterministic Matrix Sketching. *arXiv:1206.0594 [cs]*, June 2012. arXiv: 1206.0594.

[122] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11. ACM, 2003.

[123] Shicong Liu, Junru Shao, and Hongtao Lu. Generalized Residual Vector Quantization for Large Scale Data. *Proceedings - IEEE International Conference on Multimedia and Expo*, 2016-Augus, 2016.

[124] Yipeng Liu, Maarten De Vos, and Sabine Van Huffel. Compressed sensing of multichannel eeg signals: the simultaneous cosparsity and low-rank optimization. *IEEE Transactions on Biomedical Engineering*, 62(8):2055–2061, 2015.

[125] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[126] Luo Luo, Cheng Chen, Zhihua Zhang, Wu-Jun Li, and Tong Zhang. Robust Frequent Directions with Application in Online Learning. *Journal of Machine Learning Research*, 20(1):41, February 2019.

[127] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online dictionary learning for sparse coding. In *Proceedings of the 26th annual international conference on machine learning*, pages 689–696, 2009.

[128] Stephen Makonin, Bradley Ellert, Ivan V. Bajic, and Fred Popowich. Electricity, water, and natural gas consumption of a residential house in Canada from 2012 to 2014. *Scientific Data*, 3(160037):1–12, 2016.

[129] Shiva Manne and Manjish Pal. Fast Approximate Matrix Multiplication by Solving Linear Systems. *arXiv:1408.4230 [cs]*, August 2014. arXiv: 1408.4230.

[130] Zelda Mariet and Suvrit Sra. Diversity networks. *arXiv preprint arXiv:1511.05077*, 2015.

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

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

[133] Shaou-Gang Miaou and Heng-Lin Yen. Multichannel ecg compression using multichannel adaptive vector quantization. *IEEE transactions on biomedical engineering*, 48(10):1203–1207, 2001.

[134] D Minnen, C Isbell, I Essa, and T Starner. Detecting Subdimensional Motifs: An Efficient Algorithm for Generalized Multivariate Pattern Discovery. In *ICDM 2007*, pages 601–606. IEEE Computer Society, 2007.

[135] D Minnen, T Starner, I Essa, and C Isbell. Discovering Characteristic Actions from On-Body Sensor Data. pages 11–18, 2006.

[136] D Minnen, T Starner, J A Ward, P Lukowicz, and G Tröster. *Recognizing and Discovering Human Actions from On-Body Sensor Data*. IEEE, 2005.

[137] Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas. ACDC: A Structured Efficient Linear Layer. *ICLR*, (2):1–11, 2016.

[138] Jack Moffitt. Ogg vorbis. *Linux journal*, 2001(81es):9, 2001.

[139] Youssef Mroueh, Etienne Marcheret, and Vaibhava Goel. Co-Occuring Directions Sketching for Approximate Matrix Multiply. *arXiv:1610.07686 [cs]*, October 2016. arXiv: 1610.07686.

[140] Jelani Nelson and Huy L Nguyên. Osnap: Faster numerical linear algebra algorithms via sparser subspace embeddings. In *2013 ieee 54th annual symposium on foundations of computer science*, pages 117–126. IEEE, 2013.

[141] Peter Nemenyi. Distribution-free multiple comparisons. In *Biometrics*, volume 18, page 263. INTERNATIONAL BIOMETRIC SOC 1441 I ST, NW, SUITE 700, WASHINGTON, DC 20005-2210, 1962.

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

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

[144] MFXJ Oberhumer. Lzo-a real-time data compression library. *http://www.oberhumer.com/opensource/lzo/*, 2008.

[145] Rasmus Pagh. Compressed matrix multiplication. *ACM Transactions on Computation Theory*, 5(3):1–17, August 2013.

[146] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[147] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[148] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[149] Thanawin Rakthanmanon and Eamonn Keogh. Fast shapelets: A scalable algorithm for discovering time series shapelets. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 668–676. SIAM, 2013.

[150] Thanawin Rakthanmanon, Eamonn J Keogh, Stefano Lonardi, and Scott Evans. Time series epenthesis: Clustering time series streams requires ignoring some data. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 547–556. IEEE, 2011.

[151] Attila Reiss and Didier Stricker. Towards global aerobic activity monitoring. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*, page 12. ACM, 2011.

[152] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. Littletable: a time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 125–138. ACM, 2017.

[153] Robert F Rice. Some practical universal noiseless coding techniques, part 3, module psl14, k+. 1991.

[154] Tony Robinson. Shorten: Simple lossless and near-lossless waveform compression, 1994.

[155] Tamas Sarlos. Improved Approximation Algorithms for Large Matrices via Random Projections. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 143–152, Berkeley, CA, October 2006. IEEE.

[156] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, pages 34–40. ACM, 2010.

[157] Marc Seidemann and Bernhard Seeger. Chronicledb: A high-performance event store. In *EDBT*, pages 144–155, 2017.

[158] Pavel Senin and Sergey Malinchik. Sax-vsm: Interpretable time series classification using sax and vector space model. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1175–1180. IEEE, 2013.

[159] Jin Shieh and Eamonn Keogh. isax: disk-aware mining and indexing of massive time series datasets. *Data Mining and Knowledge Discovery*, 19(1):24–57, 2009.

[160] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[161] Hendrik Siedelmann, Alexander Wender, and Martin Fuchs. High speed lossless image compression. In *German Conference on Pattern Recognition*, pages 343–355. Springer, 2015.

[162] B Sigoure. Opentsdb: The distributed, scalable time series database. *Proc. OSCON*, 11, 2010.

[163] David Simcha, Erik Lindgren, Felix Chern, Nathan Cordeiro, Ruiqi Guo, Sanjiv Kumar, and Zonglin Li. Announcing scann: Efficient vector similarity search. `https://ai.googleblog.com/2020/07/announcing-scann-efficient-vector.html`, July 2020.

[164] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454, 2017.

[165] Alexander A Stepanov, Anil R Gangolli, Daniel E Rose, Ryan J Ernst, and Paramjit S Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 317–326. ACM, 2011.

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

[167] Facebook Database Engineering Team. Rocksdb: A persistent key-value store for fast storage environments. `http://rocksdb.org`.

[168] Dan Teng and Delin Chu. A Fast Frequent Directions Algorithm for Low Rank Approximation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(6):1279–1293, June 2019.

[169] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[170] Arijit Ukil, Soma Bandyopadhyay, and Arpan Pal. Iot data compression: Sensor-agnostic approach. In *Data Compression Conference (DCC), 2015*, pages 303–312. IEEE, 2015.

[171] Jean-Marc Valin, Koen Vos, and Timothy Terriberry. Definition of the opus audio codec. Technical report, 2012.

[172] Naveen Verma, Ali Shoeb, Jose Bohorquez, Joel Dawson, John Guttag, and Anantha P Chandrakasan. A micro-power eeg acquisition soc with integrated feature extraction processor for a chronic seizure detection system. *IEEE Journal of Solid-State Circuits*, 45(4):804–816, 2010.

[173] Michail Vlachos, Nikolaos M. Freris, and Anastasios Kyrillidis. Compressive mining: Fast and optimal data mining in the compressed domain. *VLDB Journal*, 24(1):1–24, 2015.

[174] Jianfeng Wang, Heng Tao Shen, Shuicheng Yan, Nenghai Yu, Shipeng Li, and Jingdong Wang. Optimized distances for binary code ranking. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 517–526, 2014.

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

[176] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. Learning to hash for indexing big dataâĂŤa survey. *Proceedings of the IEEE*, 104(1):34–57, 2016.

[177] Wenlin Wang, Changyou Chen, Wenlin Chen, Piyush Rai, and Lawrence Carin. Deep metric learning with data summarization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 777–794. Springer, 2016.

[178] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2014.

[179] Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep Fried Convnets. *ICCV*, 2014.

[180] Qiaomin Ye, Luo Luo, and Zhihua Zhang. Frequent Direction Algorithms for Approximate Matrix Multiplication with Applications in CCA. In *IJCAI*, page 7, 2016.

[181] Felix X Yu, Ananda Theertha Suresh, Krzysztof M Choromanski, Daniel N Holtmann-Rice, and Sanjiv Kumar. Orthogonal random features. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 1975–1983. Curran Associates, Inc., 2016.

[182] Qian Yu, Mohammad Ali, and A Salman Avestimehr. Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding. *IEEE Transactions on Information Theory*, 2020.

[183] Qian Yu, Mohammad Maddah-Ali, and Salman Avestimehr. Polynomial codes: an optimal design for high-dimensional coded matrix multiplication. In *Advances in Neural Information Processing Systems*, pages 4403–4413, 2017.

[184] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

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

[186] Ting Zhang, Guo-Jun Qi, Jinhui Tang, and Jingdong Wang. Sparse composite quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4548–4556, 2015.

[187] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. A general simd-based approach to accelerating compression algorithms. *ACM Transactions on Information Systems (TOIS)*, 33(3):15, 2015.

[188] Kai Zhong, Ruiqi Guo, Sanjiv Kumar, Bowei Yan, David Simcha, and Inderjit Dhillon. Fast classification with binary prototypes. In *Artificial Intelligence and Statistics*, pages 1255–1263, 2017.

[189] Hui Zou and Lingzhou Xue. A selective overview of sparse principal component analysis. *Proceedings of the IEEE*, 106(8):1311–1320, 2018.

[190] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006.