# SketchML: Accelerating Distributed Machine Learning with Data Sketches

Jiawei Jiang[†§] Fangcheng Fu[†] Tong Yang[†] Bin Cui[†]

[†]School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University [§]Tencent Inc.
[†]{blue.jwjiang, ccchengff, yang.tong, bin.cui}@pku.edu.cn [§]jeremyjiang@tencent.com

## ABSTRACT

To address the challenge of explosive big data, distributed machine learning (ML) has drawn the interests of many researchers. Since many distributed ML algorithms trained by stochastic gradient descent (SGD) involve communicating gradients through the network, it is important to compress the transferred gradient. A category of low-precision algorithms can significantly reduce the size of gradients, at the expense of some precision loss. However, existing low-precision methods are not suitable for many cases where the gradients are sparse and nonuniformly distributed. In this paper, we study *is there a compression method that can efficiently handle a sparse and nonuniform gradient consisting of key-value pairs?*

Our first contribution is a sketch based method that compresses the gradient values. Sketch is a class of algorithms using a probabilistic data structure to approximate the distribution of input data. We design a quantile-bucket quantification method that uses a quantile sketch to sort gradient values into buckets and encodes them with the bucket indexes. To further compress the bucket indexes, our second contribution is a sketch algorithm, namely MinMaxSketch. MinMaxSketch builds a set of hash tables and solves hash collisions with a MinMax strategy. The third contribution of this paper is a delta-binary encoding method that calculates the increment of the gradient keys and stores them with fewer bytes. We also theoretically discuss the correctness and the error bound of three proposed methods. To the best of our knowledge, this is the first effort combining data sketch with ML. We implement a prototype system in a real cluster of our industrial partner Tencent Inc., and show that our method is up to 10× faster than existing methods.

## CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**;

## KEYWORDS

Distributed Machine Learning; Stochastic Gradient Descent; Quantification; Quantile Sketch; Frequency Sketch


*SIGMOD'18, June 10–15, 2018, Houston, TX, USA*

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196894

**ACM Reference Format:**
Jiawei Jiang, Fangcheng Fu, Tong Yang, Bin Cui. 2018. SketchML: Accelerating Distributed Machine Learning with Data Sketches. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196894

## 1 INTRODUCTION

### 1.1 Background and Motivation

Machine learning (ML) techniques have been widely used in numerous applications, such as recommendation [21], text mining [42], image recognition [29], video detection [25], urban computing city [47], *etc.* With the unprecedented increase of data volume, a centralized system is unable to run machine learning tasks efficiently. To meet the trending of big data era, it is inevitable to deploy ML in a decentralized environment [23].

We focus on a subclass of ML models, such as Logistic Regression [20], Support Vector Machine [40], Linear Regression [38], and Neural Network [29]. Generally, they are trained with a widely used family of first-order-gradient optimization methods, namely stochastic gradient descent (SGD) [8, 48]. To distribute these gradient-based algorithms, we partition a training dataset over workers and make each worker independently propose gradients [13, 22].

Under such setting, a major problem is how to efficiently exchange gradients among workers since the communication often dominates the total cost. Although the network infrastructure is becoming faster and faster nowadays, reducing gradient movement is still beneficial in many cases we try to support.

**Case 1: Large Model.** A recent phenomenon of ML is the rapid growth of model size. It has been acknowledged that a large model gives a better representation of users or objects. A more representative model is more likely to produce a higher prediction [24]. However, a large model also brings considerable communications in a distributed cluster, which impedes the overall performance. Motivated as such, it is nontrivial to squeeze the transferred data in this large model case.

**Case 2: Cloud Environment.** Cloud platforms, such as Amazon EC2, Alibaba Cloud, and Microsoft Azure, provide resizable virtual services to make distributed computing easier [6]. And they often adopt an on-demand pricing that charges a user according to the used bandwidth. To minimize cost, it is an everlasting goal to minimize the transmission through network.

**Case 3: Geo-Distributed ML.** For many international companies, it is infeasible to move data between data centers before running ML algorithms. Data movement over wide-area-network (WAN) is much slower than local-area-network (LAN). Reducing the communication between data centers can help geo-distributed ML.

**Case 4: Internet of Things (IoT).** IoT infrastructure tries to integrate mobile phones, physical devices, vehicles and many other embedded objects in a unified network [17]. IoT controls these objects to collect and exchange information. In this huge and heterogeneous network, an efficient communication infrastructure is of great value.

In the above ML cases, it is significant to reduce the communicated gradients through network and guarantee algorithmic correctness meanwhile. Often, compression techniques are used to address this problem. Existing compression approaches can be summarized into two categories — lossless methods and lossy methods.

Lossless methods for repetitive integer data, such as Huffman Coding [28], RLE (Run-Length Encoding) [18], DEFLATE [14], and Rice [49], cannot be used for non-repetitive gradient keys and floating-point gradient values. Methods such as Compressed Sparse Row (CSR) can store matrix-type data via taking advantage of data sparsity [7, 41], but the performance improvement is not large enough due to limited compression performance.

Lossy methods are proposed to compress floating-point gradients by a threshold based truncation [39] or a quantification strategy [30, 45]. The threshold based truncation is too aggressive to make ML algorithm converged. At a high level, the quantification approach is more promising since it achieves a tradeoff between compression and convergence. But the existing quantification approaches have two assumptions in common, which are not true in real cases. 1) First, they assume that a gradient vector needed to be compressed is dense. However, in many real large-scale ML applications, gradient vectors are sparse due to the sparsity of training data. On the one hand, a lot of time is wasted if we compress all the dimensions of a sparse gradient vector. On the other hand, the gradient keys cannot be compressed if we store a sparse gradient vector in (key, value) pairs. 2) Second, they assume that the gradient values follow a uniform distribution. But, according to our observation, the gradient values in a gradient vector generally conform to a nonuniform distribution. Worse, most gradient values locate in a small range near zero. The uniform quantification approach is unable to fit the statistical distribution of gradient values.

According to the above analysis, the existing compression solutions are not powerful enough for large-scale gradient optimization algorithms. Motivated by this challenge, we study the question that *what data structure should we use to compress a sparse gradient vector?* Unsurprisingly, methods designed for dense and uniform-distributed gradients can perform poorly in a sparse and nonuniform-distributed setting. To address this problem, we propose SketchML, a general compression framework that supports sparse gradients and fits the statistical distribution of gradients. Briefly speaking, for a sparse gradient vector consisting of $\{(k_j, v_j)\}_{j=1}^{d}$ pairs, we use a novel sketch-based algorithm to compress values and a delta-binary encoding method to compress keys. They bring an improvement over state-of-the-art algorithms of 2-10×. We also theoretically analyze the error bound and the correctness of the proposed algorithms.

## 1.2 Overview of Technical Contributions

We first introduce the context for describing our proposed method and then describe each contribution individually.

**Data Model.** We focus on a subclass of ML algorithms that are trained with stochastic gradient descent (SGD), *e.g.*, Logistic Regression and Support Vector Machine. The input dataset contains training instances and their labels — $\{x_i, y_i\}_{i=1}^{N}$. The purpose is to find a predictive model $\theta \in \mathbb{R}^D$ that minimizes a loss function $f$. SGD iteratively scans each $x_i$, calculates the gradient $g_i = \nabla f(x_i, y_i, \theta)$, and updates $\theta$ in the opposite direction [9]:

$$\theta = \theta - \eta g_i$$

where $\eta$ is a hyper-parameter called the learning rate. Note that $g_i \in \mathbb{R}^D$ is generally a sparse vector due to the data sparsity prevalent in large-scale ML. To save space, we store the nonzero elements in a gradient vector, denoted by key-value pairs $\{k_j, v_j\}_{j=1}^{d}$. In a distributed setting, we choose the data-parallel strategy that partitions the dataset over $W$ workers [13]. With this scenario, we need to aggregate gradients proposed by $W$ workers, denoted by $\{g^w\}_{w=1}^{W}$.

**How to Compress Gradient Values?** The first goal is to compress the gradient values in the key-value pairs, i.e., $\{v_j\}_{j=1}^{d}$. Since the uniform quantification is ill-suited for nonuniform distributed gradients, an alternative probabilistic data structure is the sketch algorithm which is widely used to analyze a stream of data. Existing sketch algorithms include the quantile sketch [11, 16] and the frequency sketch [12]. Quantile sketches are used to estimate the distribution of items, while frequency sketches are used to estimate the occurring frequency of items.

We propose to use a quantile sketch to summarize the gradient values into several buckets, and then encode each value by the corresponding bucket index $b(v_j)$. The number of buckets is a relatively small integer. We use a binary representation to encode the bucket indexes and thereby reduce the communication cost. We further investigate the possibility of compressing the bucket indexes. At the first glance, the frequency sketch seems a good candidate by using multiple hash tables to approximately store integers. However, according to our intuitive and empirical analysis, we find that it cannot be extended to solve our problem since our context is completely different from the frequency scenario. To address this problem, we propose a novel sketch algorithm, called MinMaxSketch. MinMaxSketch encodes the bucket indexes using a multiple-hashing approximation, and design a MinMax strategy to solve the hash collision problem during the insertion phase and the query phase. Besides, we choose a dynamic learning rate schedule to compensate the vanishing of gradients, and devise a grouping method to decrease quantification error. Empirically, the sketch-based algorithm is able to significantly reduce the communication cost. To the best of our knowledge, this is the first effort that introduces a sketch algorithm to optimize the performance of machine learning tasks.

**How to Compress Gradient Keys?** The second goal is to compress the gradient keys in the key-value pairs. Different from gradient values that can bear a low-precision avenue, gradient keys are vulnerable to inaccuracy. Assuming we encode a key but fail to decode it accurately due to the precision loss during compression, we will unfortunately update a wrong dimension of $\theta$. Therefore, we need a lossless method to compress gradient keys, otherwise we cannot guarantee the correct convergence of optimization algorithms. Since the key-value pairs are sorted by keys, meaning that the keys are in ascending order, we propose to store the keys with a delta format. Specifically, we store the difference of adjacent keys. Although a

gradient key can be very large for a high dimensional model, the difference between two neighboring keys is often in a small range. We can hence store them with a binary representation and transfer them with fewer bytes. According to our empirical results, each key only consumes an average of about 1.27 bytes — 3.2× smaller for a four-byte integer or 6.3× for an eight-byte long-integer.

**Evaluation.** In order to systematically assess our proposed methods, we implement a prototype on the top of Spark. On a fifty-node real cluster of Tencent, we use two large-scale datasets to run a range of ML workloads. Our proposed framework SketchML is 2-10× faster than the state-of-the-art approaches.

**Roadmap.** The rest of this paper is organized as follows. We introduce the preliminary in Section 2. We describe the compression framework SketchML in Section 3, and its theoretical proof in Appendix A. We show the experimental results in Section 4, describe related work in Section 5, and conclude this work in Section 6.

## 2 PRELIMINARIES

In this section, we introduce some preliminary materials related to the processed data and the sketch algorithms.

### 2.1 Definition of Notations

To help the readers understand this work, we use the following notations throughout the paper.

- $W$: number of workers.
- $N$: number of training instances.
- $D$: number of model dimensions.
- $g$: a gradient vector.
- $d$: number of nonzero dimensions in a gradient vector.
- $(k_j, v_j)$: $j$-th nonzero gradient key and gradient value in a sparse gradient vector.
- $m$: size of a quantile sketch.
- $q$: number of quantile splits.
- $s$, $t$: row and column of MinMaxSketch. $s$ denotes the number of hash tables, and $t$ denotes the number of bins in a hash table.
- $r$: group number of MinMaxSketch.

### 2.2 Data Model

The ML problem that we tackle can be formalized as follows. Given a dataset $\{x_i, y_i\}_{i=1}^{N}$ and a loss function $f$, we try to find a model $\theta \in \mathbb{R}^D$ that best predicts $y_i$ for each $x_i$. For this supervised ML problem, a common training avenue is to use the first-order gradient optimization algorithm SGD. The executions involve repeated calculations of the gradient $g_i = \nabla f(x_i, y_i, \theta)$ over the loss function. Typically, $g_i \in \mathbb{R}^D$ is a sparse vector since the training instance $x_i$ is generally sparse. In a distributed environment, since each worker proposes gradient independently, we need to gather all the gradients and update the trained model. Assuming there are $W$ workers, our goal is to compress the gradients $\{g^w\}_{w=1}^{W}$ before sending them. Once SGD finishes a pass over the entire dataset, we say SGD has finished an epoch.

### 2.3 Quantile Sketch

Consider a case of one billion comparable items, whose values are unknown beforehand. An important scenario is analyzing the distribution of item values in a single pass. A brute-force sorting can provide the exact solution, but the computation complexity is $O(N \log N)$ and the space complexity is $O(N)$. The expensive computation and space cost make it infeasible for a large volume of items.

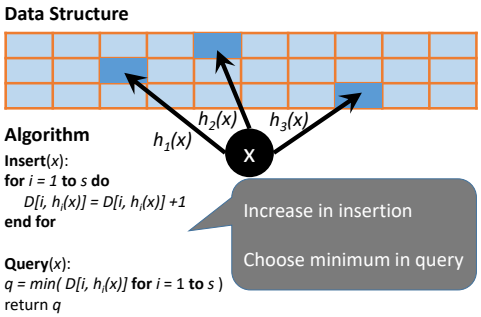



**Figure 1: A Frequency Sketch**

Quantile sketch addresses this problem by using a small data structure to approximate the exact distribution of item value in a single pass over the items. The main component of quantile sketch is the quantile summary which consists of a small number of points from the original items [16]. Two major operations, `merge` and `prune`, are defined for quantile summary. The `merge` operation combines two summaries into a merged summary, while the `prune` operation reduces the number of summaries to avoid exceeding the maximal size. Since there are $m$ quantile summaries in a quantile sketch, the computation complexity is $O(N)$ and the space complexity is $O(m)$. In contrast to the brute-force sorting, the total cost is reduced significantly. Meanwhile, the existing quantile sketches also provide solid error bounds. For example, Yahoo DataSketches [1] guarantees 99% correctness when $m = 256$. Once a quantile sketch is built for these one billion items, the quantile summaries are used to give approximate answers to any quantile query $q \in [0, 1]$. For example, a query of 0.5 refers to the median value of the items, and the quantile sketch returns an estimated value for the item ranking 0.5 billion. With the same manner, a query of 0.01 returns an estimated value for the item ranking 10 million.

One classical quantile sketch is GK algorithm [16]. Some works also design extensions of the GK algorithm [11, 16, 46]. GK algorithm maintains a summary data structure $S(n, k)$ in which there is an ordered sequence of $k$ tuples in $n$ previous items. These tuples correspond to a subset of items seen so far. For each stored item $v$ in $S$, we maintain implicit bounds on the minimum and the maximum possible rank of the item $v$ in total $n$ items.

### 2.4 Frequency Sketch

Another popular real case in a stream of data is the repeated occurrences of items. Since it is impractical to store every possible item due to the large value range of items, the frequency sketch is proposed to estimate the frequency of different values of items. Count-min sketch is a widely used frequency sketch [12], as shown in Figure 1. Essentially, count-min sketch is similar to the principle of Bloom Filter. The data structure is a two-dimensional array of $s$ rows and $t$ columns, denoted by $H$. Each row is a $t$-bin hash table, and associated with each row is a separate hash function $h_i(-)$. In the insertion phase, an item $x$ is processed as follows: for each row $i$, we use the hash function to calculate a column index $h_i(x)$, and increment the corresponding value in $H$ by one. In the query

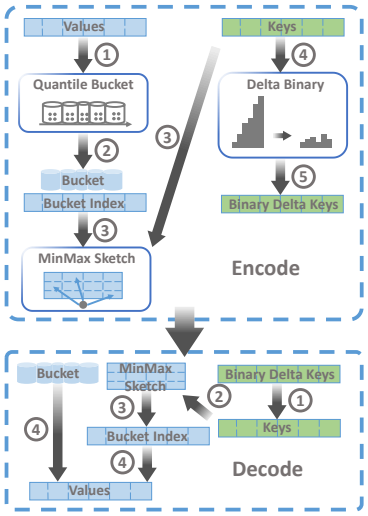



**Figure 2: Framework Overview of SketchML**

phase, the same hash procedure obtains $s$ candidates from $H$, and the minimum is chosen as the final result.

Despite of the query efficiency, the hash methods all face a collision problem that two different items might be mapped to the same hash bin by the hash function. How to address the hash collision is therefore a vital issue. Count-min sketch ignores hash collisions and increases the hash bin once it is chosen. Obviously, the queried candidates are equal or larger than the true frequency $\widetilde{q}$ due to the possibility of hash collision. Therefore, the minimum operation chooses the one closest to $\widetilde{q}$.

# 3 THE FRAMEWORK OF SKETCHML

Our proposed compression framework, SketchML, is described in this section. We first walk through an overview of the framework, and then describe each component individually.

## 3.1 Overview of The Framework

Figure 2 illustrates the overview of our proposed framework. There are three major components in the framework, i.e., quantile-bucket quantification, MinMaxSketch, and dynamic delta-binary encoding. The first two components together compress gradient values, while the third component compresses gradient keys.

**Encode Phase.** In the encode phase, the framework operates as follows:

(1) Quantile sketch is used to scan the values and generate candidate splits, with which we use bucket sort to summarize the values.
(2) The values are represented by the corresponding bucket indexes.
(3) The bucket indexes are inserted into the MinMaxSketch by applying the hash functions on the keys.
(4) The keys are transformed to their increments, denoted by delta keys in this paper.
(5) We use binary encoding to encode the delta keys with fewer bytes, instead of using four-byte integers.

**Decode Phase.** In the decode phase, the framework recovers the compressed gradients by the following procedures:

(1) The delta keys are recovered to the original keys.

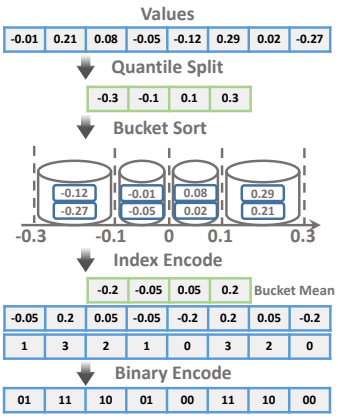



**Figure 3: Quantile-Bucket Quantification**

(2) The recovered keys are used to query the MinMaxSketch.
(3) The bucket index of each value is obtained from the sketch.
(4) The value is recovered by querying the bucket value with the bucket index.

## 3.2 Quantile-Bucket Quantification

The component of quantile-bucket quantification compresses the gradient values in a gradient consisting of key-value pairs $\{k_j, v_j\}_{j=1}^{d}$.

**Motivation.** Different from the integer gradient keys, the gradient values are floating-point numbers. Many existing works have shown that gradient optimization algorithms are capable of working properly in the presence of noises [30, 34]. Taking SGD as an example, it calculates a gradient with only one training instance, resulting in inevitable gradient noises due to noisy data. Although SGD might oscillate for a while due to noisy gradients, it can go back to the correct convergence trajectory afterwards [9].

Driven by the robustness requirement against noises, we ask *can optimization algorithms converge with quantified low-precision gradients?* Intuitively, since SGD can converge in the presence of random noises, low-precision gradients are able to work as well. Compared with unpredictable noises, the error incurred by quantification is usually controllable and bounded [5]. Therefore, SGD is likely to converge normally.

**Quantification Choices.** The current quantification methods mostly adopt the uniform strategy in which the floating-point numbers are linearly mapped to integers [45]. However, uniform quantification is ill-suited for gradients. Figure 4 is an example of the distribution of gradient values. We train a public dataset [2] with SGD and select the first generated gradient. The x-axis refers to the gradient values, while the y-axis refers to the count of gradient values falling into an interval. In this example, the value range of the gradient values is $[-0.353, 0.004]$, but most of them are near zero. It verifies that gradient values generally conform to a nonuniform distribution rather than a uniform distribution. A uniform quantification equally divides the range of gradient values, and cannot capture the nonuniform distribution of data. Since most gradient values are close to zero, methods such as ZipML quantify them to zero. Therefore, many gradient values are ignored, causing slower convergence.

To address the defect of uniform quantification, we investigate the employment of quantile sketch to capture the data distribution of

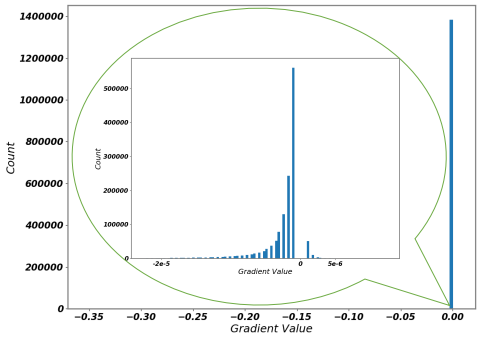



**Figure 4: Nonuniform Gradient Values**

gradient values. Briefly speaking, we equally divide all the values into several parts, instead of equally dividing the range of values. The proposed quantile-bucket quantification has three steps.

**Step 1: Quantile Split.** In this step, we build a quantile sketch with the gradient values and generate quantile splits, as shown in Figure 3.

(1) We scan all the gradient values and insert them into a quantile sketch. Here we choose Yahoo DataSketches [1], a state-of-the-art quanitle sketch.
(2) $q$ quantiles are used to get candidate splits from the quantile sketch. Detailedly, we generate $q$ averaged quantiles $\{0, \frac{1}{q}, \frac{2}{q}, ..., \frac{q-1}{q}\}$.
(3) We use the quantiles and the maximal value as split values, denoted by $\{rank(0), rank(\frac{1}{q}), rank(\frac{2}{q}), ..., rank(1)\}$. Note that the number of items whose values are between two sequential splits is $\frac{N}{q}$, meaning that we divide items by the number rather than the value. Each interval between two splits has the same number of gradient values.

**Step 2: Bucket Sort.** Given quantile splits, we proceed to quantify the gradient values with bucket sort.

(1) We call each interval between two splits a bucket. The smaller split is the lower threshold of the bucket, and the larger split is the higher threshold.
(2) Based on the bucket thresholds, each gradient value belongs to one specific bucket. For instance, the value of 0.21 in Figure 3 is classified to the fourth bucket.
(3) Each bucket is represented by the mean value, i.e., the average of two splits.
(4) Each gradient value is transformed into the corresponding bucket mean. This operation introduces quantification errors since the bucket mean does not always equal to the original value.

**Step 3: Index Encode.** Although we quantify gradient values with bucket mean values, the consumed space remains the same because we still store them as floating-point numbers. For the purpose of reducing space cost, we choose an alternative that stores the bucket index. We encode the mean value of a bucket as the bucket index. For example, after quantifying 0.21 to the mean value of the fourth bucket, we further encode it by the bucket index starting from zero, i.e., three for 0.21.

**Step 4: Binary Encode.** Generally, the number of buckets is a small integer. We compress the bucket indexes through encoding them to binary numbers. If $q = 256$, one byte is enough to encode

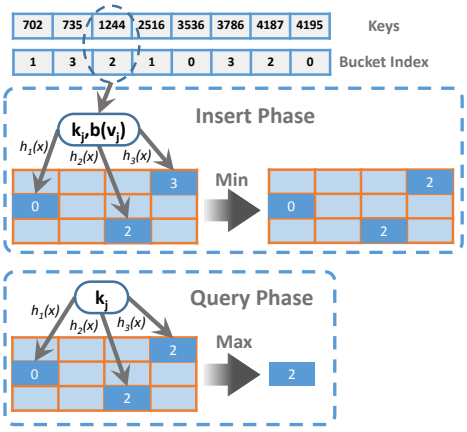



**Figure 5: MinMaxSketch**

the bucket indexes. In this way, we reduce the space taken from $8d$ bytes to $d$ bytes. Besides, we need to transfer the mean values of buckets in order to decode the gradient values. Therefore, the total space cost is $d + 8q$ bytes. Since $q << d$ in most cases, we can decrease the transferred data to a large extent.

**Proof of Variance Bound.** The proposed quantification based method ineluctably incurs quantification variances. We statistically analyze the bound of variance in Appendix A.1.

**Summary.** Through an in-depth anatomy of existing quantification methods, we find that they cannot capture the distribution property of gradients. We therefore investigate nonuniform quantification methods. By designing a technique that combines quantile sketch and bucket sort, we successfully encode gradient values to small binary numbers and achieve self-adaption to data nonuniformity. The key-value pair $(k_j, v_j)$ is transformed into $(k_j, b(v_j))$ where $b(v_j)$ denotes the binary bucket index. In practice, we find that $q = 256$ is often enough to obtain comparable prediction accuracy.

## 3.3 MinMaxSketch

The component of quantile-bucket quantification has compressed gradient values with a compression rate close to eight. We next study the possibility of going a step further. Fundamentally, the gradient keys need to be recovered precisely so that low-precision techniques cannot be used. As a result, we focus on the bucket index.

**Motivation.** Since we have converted the gradient values to bucket indexes, which are integers, we consider low-precision methods designed for integers. Among the existing works, frequency sketch is a classical probabilistic data structure that reveals powerful capability in processing a stream of data [12]. However, the underlying scenario of frequency sketch is totally different from our setting. Frequency sketch aims at a set of items, each of which might appear repeatedly. Frequency sketch tries to approximately guess the frequency of an item with a relatively small space. In contrast, there is no repeated gradient key in our targeted task and our goal is to approximate each single bucket index.

If we use the additive strategy of frequency sketch, it is nearly impossible to get a good result. Assuming that we add an inserted bucket index to the current hash bin, the hash bin might be updated arbitrarily. Intuitively, hash bins ever collided are magnified in an unpredictable manner. Therefore, most decoded gradient values are

much larger than the original value. Amplified gradients then cause unstable convergence. In fact, according to our empirical results, optimization methods often easily get diverged with larger gradients.

Due to the problem described above, we need to design a completely different data structure for our targeted scenario. Although frequency sketch does not work, its multiple hash strategy is useful in solving hash collisions. The same strategy is also adopted in other methods such as Bloom Filter. Based on this principle, we propose a new sketch, namely MinMaxSketch, in this section.

**Insert Phase.** To begin with, we scan all the items and insert them into the sketch. Figure 5 illustrates how the insertion works.

(1) Each input item is composed of original key and the encoded bucket index — $(k_j, b(v_j))$.
(2) We use $s$ hash functions to calculate the hash codes. In Figure 5, there are three hash functions, $h_1(-)$, $h_2(-)$, and $h_3(-)$.
(3) Once a hash bin is chosen in the $i$-th hash table, we compare the current value $H(i, h_i(k_j))$ and $b(v_j)$. If $H(i, h_i(k_j)) > b(v_j)$, we replace the current value by $b(v_j)$. Otherwise, we do not change the current value.

As the name of MinMaxSketch implies, the symbol of Min refers to the choice of minimum bucket index in the insert phase. The reason behind this design decision is to avoid the increase of hash bin and therefore avoid the increase of decoded gradients.

**Query Phase.** Once a MinMaxSketch is built, the next question is how to query results from the sketch. In accordance with the insert phase, the query phase operates as follows.

(1) The input is a gradient key, denoted by $k_j$. $s$ hash functions are applied to $k_j$ and each hash function chooses one hash bin from the hash table.
(2) Given $s$ candidates from different rows, we select the maximal one as the final result. In Figure 5, three candidates are $\{0, 2, 2\}$, we choose 2 as the result.

The choice of maximal candidates corresponds to the Max symbol of MinMaxSketch. Since we select the minimum candidate in the insert phase, the choice of maximal candidate in the query phase produces the one closest to the original value.

**Analysis.** As a probabilistic data structure, MinMaxSketch and many other sketch algorithms suffer from a problem, that is, the queried result is not guaranteed to be exactly the same as the original value. Therefore, it is necessary to analyze the queried performance of MinMaxSketch.

Basically, there are two kinds of errors when querying a sketch: overestimated error and underestimated error. Overestimated error brings larger queried results, while underestimated error brings smaller queried results. All existing frequency sketches either have both errors or only have overestimated error [12]. That is to say, they all have overestimated error. Unfortunately, overestimated error brings non-trivial degradation for our setting. As analyzed before, if we query an overestimated bucket index from the sketch, the decoded gradient value is generally amplified. The overestimation of gradient values often gives rise to an unpredictable and unstable convergence.

In contrast, MinMaxSketch introduces underestimated error. In the insert phase, we choose the smaller value in the presence of hash collisions. Therefore, the hash bin is not larger than all related bucket indexes. Consequently, the queried result is underestimated. The underestimation of bucket index then generally incurs underestimated gradient value.

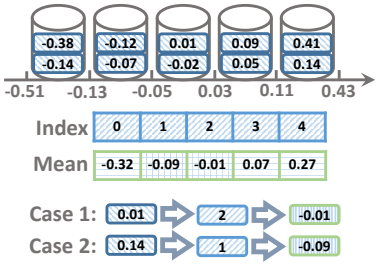



**Figure 6: Reversed Gradient**

As the readers might suspect, can optimization algorithms converge with underestimated gradients? Theoretically, optimization algorithms such as SGD move towards the optimality following the opposite direction of gradients. Obviously, reducing the scale of gradients might slow down the convergence rate somewhat, yet still on the correct convergence track. On the contrary, uncontrolled increase of the scale of gradients might risk jumping over the optimality.

To sum up, MinMaxSketch might decrease the scale of gradients, yet still guarantees the correct convergence. However, although MinMaxSketch makes sense intuitively and theoretically, it cannot work empirically with this original version. Next, we will discuss two major problems and describe our solutions.

**Problem 1: Reversed Gradient.** Above, we state that MinMaxSketch often provides decayed gradients. However, this statement is not always true. Indeed, the bucket index is decayed with MinMaxSketch. But the parsed gradient value is uncertain as we need to query the bucket mean value with the bucket index. We find that the sign of the decoded gradient value could be reversed. Figure 6 shows an example of reversed gradients. Ten gradient values are put into five buckets. Nevertheless, there are two cases where MinMaxSketch produces reversed gradients.

- **Case 1.** The third bucket includes gradient values from -0.05 to 0.03. The mean of the bucket is -0.01. On this occasion, 0.01 is encoded to -0.01. Therefore, even if MinMaxSketch decodes the correct bucket index, it reverses the sign of 0.01 anyway.
- **Case 2.** The other four buckets fortunately avoid the first case as they exclude the value of zero. But, MinMaxSketch might produce reversed gradients for them too. For example, the value of 0.14 belongs to the fifth bucket. However, if MinMaxSketch produces a smaller bucket index, e.g., one in Figure 6, the queried value becomes -0.09.

Gradient optimization algorithms such as SGD are robust to decayed gradients, yet vulnerable to reversed gradients. With reversed gradients, they are likely to diverge.

**Solution 1: Separation of Positive/Negative Gradients.** The reason of problem 1 is that we quantify positive and negative gradient values together. To address this problem, we design a mechanism that handles positive and negative gradients independently.

(1) For positive and negative gradients, we build two separate quantile sketches and quantify them with separate buckets. With this strategy, the first bucket for positive gradients is closest to zero, while the last bucket for negative gradients is closest to zero.

(2) Based on the quantified gradient values, we build one positive MinMaxSketch and one negative MinMaxSketch.
(3) In the insert phase, in order to achieve the goal of decaying gradients, we choose the bucket index closest to the "minimum bucket". Here, the "minimum bucket" refers to the bucket having the minimum mean, i.e., the first bucket for positive gradients and the last bucket for negative gradients.

**Problem 2: Vanishing Gradient.** As aforementioned, MinMaxSketch yields decayed gradients, which we call the problem of vanishing gradient. Although the correct convergence is not harmed, the convergence rate is inevitably reduced due to reduced step in each SGD iteration.

**Solution 2: Adaptive Learning Rate and Grouped MinMaxSketch.** We design two methods to compensate the problem of vanishing gradient.

- **Adaptive Learning Rate.** Adam algorithm is used to adaptively adjust the learning rate [27]. Due to the data skewness, different dimensions of the trained model converge in different speed. Adam is proposed to solve this convergence imbalance by choosing a learning rate schedule that is inversely proportional to the change of gradients so far. It gives larger learning rate for slower dimensions and smaller learning rate for faster dimensions.
- **Grouped MinMaxSketch.** According to our empirical results, the introduction of adaptive learning rate can significantly enhance the convergence rate. However, we find that it cannot achieve the optimality. With the mechanism of MinMaxSketch, the difference between the original bucket index and the decoded bucket index can be as large as $q$, the number of quantile splits. When the trained model is near the optimality, the gradients are very small themselves. Adaptive learning rate is unable to fully compensate the decline of decoded bucket index. To address this problem, we divide all the buckets into $r$ groups and create one MinMaxSketch for each of them. For example, if $q = 256$ and $r = 8$, we divide the buckets into 8 groups — [0,32), [32,64), etc. The maximal decoded error of bucket index is reduced from $q$ to $\frac{q}{r}$. And the error of decoded gradient is therefore reduced.

**Proof of Error Bound.** We also theoretically discuss the error bound and correctness of MinMaxSketch. Due to the space limitation, we present the detailed proof in Appendix A.2.

**Summary.** MinMaxSketch is designed to compress the bucket index generated by the component of quantile-bucket quantification. MinMaxSketch handles the disturbance of hash collision through a min protocol in the insert phase and a max protocol in the query phase. We further propose techniques to address the reversal and decay of decoded gradients.

## 3.4 Delta-Binary Encoding

The above two components emphasize on the compression of gradient values. Next, we introduce the component of dynamic delta-binary encoding, which compresses the gradient keys in a gradient consisting of key-value pairs $\{k_j, v_j\}_{j=1}^{d}$.

**Motivation.** In many cases related to floating-point numbers and integer numbers, we can use low-precision compression methods if they can bear certain precision loss. However, the integer gradient keys are unable to tolerate errors. Assuming a case that we compress a key but cannot recover it accurately, a wrong dimension of the trained model will be updated. This phenomenon will cause unpredictable convergence and divergence even worse. Therefore, we must design a lossless compression method for the gradient keys.

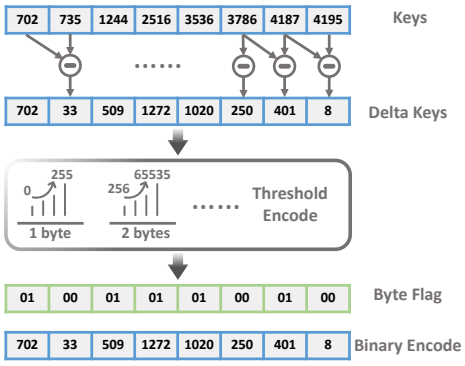



**Figure 7: Delta-Binary Encoding**

Through an analysis of the data distribution of gradient keys, we find that they have three characteristics. First, the keys are non-repetitive. Second, the keys are ordered in an ascending order. Third, although the keys can be very large in many high dimensional applications, the difference between two neighboring keys is much smaller. Motivated by this intuition, we propose to only store the increment of keys. Our method is composed of two major steps, as introduced below.

**Step 1: Delta Encoding.** As shown in Figure 7, the gradient keys are ordered in an array. We scan the array from the end to the start, and calculate the difference between two adjacent keys. Afterwards, we get the increments of keys, which we call the `delta keys`.

**Step 2: Binary Encoding.** Through delta encoding, it is obvious that the delta keys are much smaller than the original keys. If we store the delta keys in the format of integers or long-integers, then the compression is meaningless because the consumed memory space and communication cost remain the same.

To solve this problem, we assign different spaces to different delta keys and encode them in the binary format. A threshold module receives each delta key and outputs the least number of bytes needed to hold it. Specifically, one byte can handle a range of $[0, 255]$, two bytes $[256, 65535]$, three bytes $[65536, 16777215]$, four bytes $[16777216, 4294967295]$. The number of required bytes is encoded to a binary number, called the `byte flag`. For example, the flag of one byte is 00, that of two bytes is 01, and so forth. Finally, the delta keys are encoded into binary numbers based on the byte flags.

Note that, there are several existing methods that can be used to compress integers, such as RLE (Run-length Encode) and Huffman Coding. However, RLE and Huffman Coding are typically used to compress a data sequence in which a same data value might occur consecutively. They need to store every gradient key without compressing and introduce extra data structure. Therefore, they are useless for non-repetitive gradient keys.

**Summary.** In order to compress gradient keys without precision loss, we store the increment of keys and use a threshold mechanism to encode them into the binary format. The key-value gradient pair $(k_j, v_j)$ is transformed into $(\Delta k_j, v_j)$ where $\Delta k_j$ denotes the binary incremental key. As we will theoretically analyze in Appendix A.3

and evaluate in the experiment, the average byte needed by each key is below 1.5 bytes.

### 3.5 Analysis of Space Cost

Combining the above three components, we get a unified framework. In this section, we explicitly analyze the space cost of our methods.

- **Quantile-Bucket Quantification.** The mean values of buckets need to be transferred by the network. The size is $8q$ bytes. Generally, $q$ is a small integer.
- **MinMaxSketch.** We build $r$ grouped sketches. The size of each individual MinMaxSketch is $\frac{s\times t}{r} \times \lceil \log_{256} q \rceil = \frac{s\times t}{r} \times \lceil \frac{1}{8}\log_2 q \rceil$. The total size of MinMaxSketch is $s \times t \times \lceil \frac{1}{8}\log_2 q \rceil$.
- **Delta-Binary Encoding.** As we will discuss in Appendix A.3, the expected bytes taken for each delta key is $\lceil \frac{1}{8}\log_2 \frac{rD}{d} \rceil$. The byte flag needs $\frac{1}{4}$ byte per key. In practice, we find that the average size for each key (including byte flag) is 1.27 bytes approximately.

**Summary.** To sum up, the total space cost of our method is $d \times (\lceil \frac{1}{8}\log_2 \frac{rD}{d} \rceil + \frac{1}{4}) + 8q + s \times t \times \lceil \frac{1}{8}\log_2 q \rceil$. Compared with the original size $12d$, we can save a lot of communication cost by choosing appropriate hyper-parameters.

## 4 EXPERIMENTS

We validate the effectiveness and efficiency of our proposed methods by conducting extensive experiments.

### 4.1 Experiment Setting

**Implementation.** We implement a prototype system on Spark. The prototype is compiled with Java 8 and Scala 2.11.7. There are two types of nodes in Spark, the driver and the executor. The training dataset is partitioned over executors. Each executor reads the subset, and calculates gradients. The driver aggregates gradients from the executors, updates the trained model, and broadcasts the updated model to the executors. This process iterates until convergence.

**Clusters.** We use two clusters in our experiments. *Cluster-1* is a ten-node cluster in our lab. Each machine is equipped with 32GB RAM, 4 cores, and 1-Gbps Ethernet. We use this cluster to assess the effectiveness of our proposed methods. *Cluster-2* is a 300-node productive cluster in Tencent Inc. In this large-scale cluster, each machine is equipped with 64GB RAM, 24 cores, and 10-Gbps Ethernet. We compare the end-to-end performance of three competitors in Cluster-2. As shared by many users in an industrial environment, Cluster-2 is governed by Yarn and has a 8GB memory constraint per node for each task.

**Datasets.** As shown in Table 1, we use three datasets in our experiments. The first dataset KDD10 is a public dataset published by KDD CUP 2010 [2], consisting of 19 million instances and 29 million features. The second dataset KDD12 is the next generation of KDD10 [3], consisting of 149 million instances and 54 million features. The task is predicting whether a user will follow an item recommended to the user in a social networking site. Items can be persons, organizations or groups. The third dataset CTR is a proprietary datasest of Tencent Inc. CTR is used to predict the click-through-rate of advertisements.

| Dataset | Size | # Instance | # Features |
|---|---|---|---|
| KDD10 | 5GB | 19M | 29M |
| KDD12 | 22GB | 149M | 54M |
| CTR | 100GB | 300M | 58M |

**Table 1: Datasets**

**Statistical Models.** For statistical models, we choose three popular machine learning models – $\ell_2$-regularized Logistic Regression (LR), Support Vector Machine (SVM), and Linear Regression (Linear). Their loss functions can be formalized as follows:

$$LR: f(x,y,\theta) = \sum_{i=1}^{N} log(1 + e^{-y_i \theta^{\mathrm{T}} x_i}) + \frac{\lambda}{2}||\theta||_2$$

$$SVM: f(x,y,\theta) = \sum_{i=1}^{N} max(0, 1 - y_i \theta^{\mathrm{T}} x_i) + \frac{\lambda}{2}||\theta||_2$$

$$Linear: f(x,y,\theta) = \sum_{i=1}^{N} (y_i - \theta^{\mathrm{T}} x_i)^2 + \frac{\lambda}{2}||\theta||_2$$

We train three algorithms with Adam SGD, which is the most popular choice of relevant works [27]. Adam SGD stores a decaying average of past gradients and decaying average of squared gradients.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

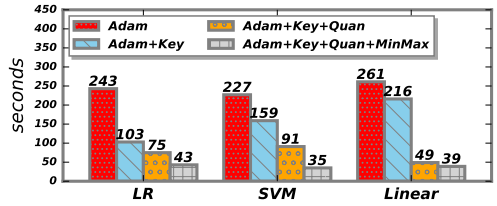
where $\beta_1$ and $\beta_2$ denote two hyper-parameters close to 1. Then, $m$ and $v$ are used to update the trained model:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} m_t$$

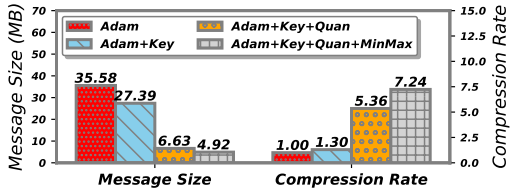
**Baselines.** We compare SketchML with two competitors: Adam SGD [27] and ZipML [45]. Adam SGD is the most widely used first-order gradient optimization recently. It combines the advantages of momentum [36, 37] and adaptive learning rate [15, 32, 43]. It hence automatically adapts to both the slope of the objective function and the importance of gradient dimensions. ZipML designs a fixed-point quantification method to compress gradient values to integers. It has shown powerful performance on a range of machine learning algorithms. Note that the Adam strategy is applied to all the baselines for the purpose of fairness.

**Metrics.** To measure the performance of SketchML and other competitors, we follow prior art and measure the average run time per epoch and the loss function with respect to the run time. We do not count the time used for data loading and result outputting for all systems [44].

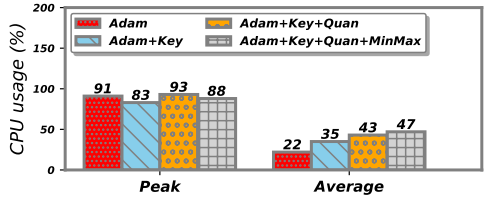
**Protocol.** The input dataset is partitioned into two subsets — 75% as the train dataset and 25% as the test dataset. We train the ML models on the train dataset and assess the quality of the trained model on the test dataset. To achieve a tradeoff between convergence robustness and convergence speed, we adopt a popular trick of SGD that uses a batch of instances instead of only one instance [8]. Better, in a distributed environment, mini-batch SGD can decrease the synchronization frequency and save a lot of communication cost. Following the choice of [19], we set the batch size as 10% of the size of the train dataset. As the authors of Adam SGD [27] suggested, we choose 0.9 for $\beta_1$, 0.999 for $\beta_2$, and $10^{-8}$ for $\epsilon$. We use grid



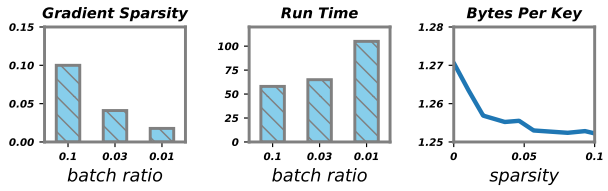

(a) Run Time Per Epoch




(b) Message Size and Compression Rate




(c) CPU Overhead




(d) Impact of Batch Size and Sparsity

**Figure 8: Effeciency of Proposed Methods. The evaluated metric is the run time per epoch. Adam refers to the basic method without our methods. Key refers to the component of delta-binary encoding. Quan refers to the component of quantile-bucket quantification. MinMax refers to the component of MinMaxSketch.**

search to tune the optimal learning rate $\eta$. Specifically, we tune the optimal learning rate with Adam SGD and use this value for all the candidates. The regularization coefficient $\lambda$ is set to be 0.01. We find that ZipML converges badly if we set its quantified size to be one byte, thus we set it to be two bytes via fine tuning. There are a few hyper-parameters in SketchML. The size of quantile sketch is 128 by default. The size of MinMaxSketch is $2\times\frac{d}{5}$.

## 4.2 Efficiency of Proposed Methods

SketchML consists of three components. In this section, we train the KDD10 dataset on ten executors of Cluster-1 to validate the efficiency of our proposed components. We assign 5GB memory for the driver and each executor. We begin with the basic method Adam, and consolidate our proposed components gradually. The results are presented in Figure 8.

**Run time.** According to the results in Figure 8(a), our proposed methods can significantly accelerate the execution of three different ML algorithms. Compared with Adam, the component of delta-binary encoding alone improves the system performance by up to 2.3×. The addition of quantile-bucket quantification further accelerates the speed by up to 4.4×. Finally, the MinMaxSketch alone achieves at most 4.3× improvements. The results demonstrate that our proposed methods are efficient in reducing the data movement through network.

**Message Size and Compression Rate.** The main advantage of compression is reducing the size of messages. Figure 8(b) presents the average message size and compression rate during the execution. Due to the space constraint, we present the result of LR, and the results of other algorithms are similar. Compared with the uncompressed gradient message, our method decreases the message size from 35.58 MB to 4.92 MB — a 7.24× compression rate.

**CPU Overhead.** To evaluate the computation overhead brought by compression, we conduct an experiment and present the result in Figure 8(c). Unsurprisingly, our method introduces 25% CPU usage in average. The peak CPU usage is not obviously influenced.

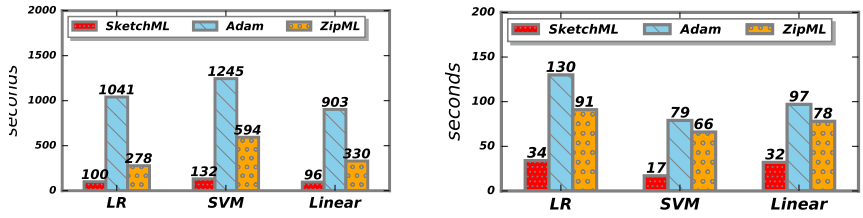
**Impact of Batch Size and Sparsity.** Since our method compresses sparse gradients, it raises a question how the data sparsity affects the performance. In our setting, the sparsity of a gradient is influenced by the batch size. Therefore, we change the sparsity by changing the ratio of the batch size. The default ratio is 10% of the dataset. As Figure 8(d) illustrates, the sparsity of gradient decreases from 10% to 1.77% when we decrease the ratio from 10% to 1%. Meanwhile, a smaller batch size incurs more frequent communication, and therefore increases the run time per epoch from 58 seconds to 105 seconds.

According to the analysis in Section 3.5, the communication cost of delta-binary encoding is directly affected by the data sparsity. Therefore, we record the performance of delta-binary encoding against the variation of data sparsity in Figure 8(d). The average size taken by each gradient key is about 1.25 bytes when the sparsity is 10%, and the size is increased to about 1.27 bytes as the sparsity approaches zero. Compared with original 4 bytes, the delta-binary encoding achieves significant compression performance. This result is consistent with the theoretical analysis in Section 3.5.

## 4.3 End-to-end Performance

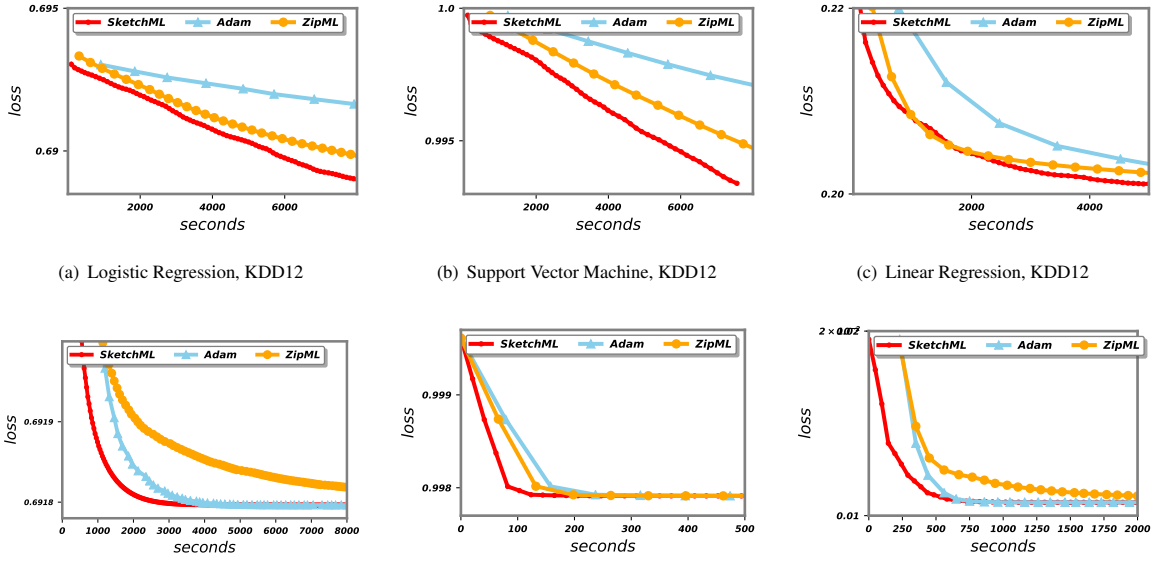
In this section, we compare the end-to-end performance of SketchML, Adam, and ZipML on Cluster-2. Due to the space constraint, we present the results of KDD12 and CTR below. We decouple the end-to-end performance as the run time per epoch and the loss in terms of run time. The average run time per epoch is presented in Figure 9. The loss regarding run time is presented in Figure 10.

*4.3.1* **Results on KDD12 Dataset.** For the KDD12 dataset, we use ten executors to run the combinations of three methods and three ML algorithms. We assign 5GB memory for the driver and each executor. Figure 9(a) shows the average run time per epoch. Figure 10 reports the convergence rate which is measured by the loss function in terms of run time.




(a) KDD12 Dataset (b) CTR Dataset

**Figure 9: End-to-end System Comparison (Run Time). The evaluated metric is the run time per epoch. Run time is in seconds. LR refers to Logistic Regression. SVM refers to Support Vector Machine. Linear refers to Linear Regression. We take three runs and report the average (standard deviation for all numbers $< 10\%$ of the mean).**




(a) Logistic Regression, KDD12 (b) Support Vector Machine, KDD12 (c) Linear Regression, KDD12

(d) Logistic Regression, CTR (e) Support Vector Machine, CTR (f) Linear Regression, CTR

**Figure 10: End-to-end System Comparison (Convergence Rate). The evaluated metric is the testing loss in terms of run time. Run time is in seconds. We take three runs and report the average (standard deviation for all numbers $< 10\%$ of the mean).**

**Logistic Regression.** As shown in Figure 9(a), SketchML runs much faster than Adam and ZipML. Adam and ZipML take 1041 seconds and 278 seconds to complete an epoch. Adam needs to communicate the original gradients without any compression. Therefore, Adam is the slowest. ZipML is 3.7× faster than Adam by compressing the gradient values. However, ZipML is unable to compress the gradient keys. In contrast, SketchML only needs 100 seconds to process an epoch, bringing 10.4× and 2.8× improvements over two competitors. The performance improvements come from the reduction of gradient data transferred through the network. And the improvement will become more significant with the increase of executors because more executors inevitably yield more communications. Although Cluster-2 is equipped with faster network switch, the network is more congested than Cluster-1 since Cluster-2 serves many applications simultaneously. Therefore, SketchML runs slower on Cluster-2 than on Cluster-1. It demonstrates that compressing the communicated messages is of great value even in a high-speed environment. For the convergence rate, we can see in Figure 10(a) that SketchML achieves the fastest convergence rate. Unsurprisingly, Adam is the slowest since it consumes the longest time to finish an epoch. ZipML converges much faster than Adam. Our method, SketchML, beats the two competitors significantly.

**Support Vector Machine.** The result of Support Vector Machine is similar to that of Logistic Regression. Adam is the slowest, followed by ZipML. We can observe in Figure 9(a) that Adam needs 1245 seconds per epoch and ZipML needs 594 seconds. SketchML only takes 132 seconds — 9.4× and 4.5× faster than Adam and ZipML. Meanwhile, as can be seen in Figure 10(b), the convergence rate of SketchML is significantly faster than Adam and ZipML. With a same time budget, SketchML is capable of converging to a much lower statistical loss than other two methods. We can find an interesting phenomenon in Figure 10(b), that is, SketchML reveals its advantages more clearly than ZipML as time goes by. As explained above, ZipML quantifies many small gradients to zero. As the training algorithm proceeds, the gradients become even smaller since the model is approaching the optimal solution. As a result, the convergence of ZipML becomes slower.

**Linear Regression.** For linear regression, Adam and ZipML take 903 and 330 seconds per epoch, respectively, while SketchML only needs 96 seconds. As Figure 10(c) shows, ZipML and SketchML are significantly faster than Adam. At the beginning of the training process, SketchML outperforms ZipML. Then, ZipML is slightly faster within a small interval. Nevertheless, as the trained model approaches the optimal, the convergence rate of ZipML slows down due to a large quantification error. Therefore, SketchML outperforms ZipML in terms of the overall performance.

#### 4.3.2 Results on CTR Dataset.

For the larger dataset CTR, we use 50 executors on Cluster-2. We assign 8GB memory for the driver and each executor due to the memory limitation. The run time statistics and convergence curves are presented in Figure 9(b) and Figure 10 (d–f).

**Logistic Regression.** On this larger dataset, Adam still runs the slowest, followed by ZipML. SketchML is 3.8× and 2.7× faster than the other two methods. Note that the speedup of SketchML on CTR is smaller than that on KDD12 since KDD12 is sparser than CTR. As each instance of CTR generates more nonzero gradient pairs, the computation cost is much higher. As a consequence, the performance improvement brought by the reduction of communication cost is not as large as that on KDD12 dataset. Although ZipML runs faster than Adam, its convergence rate is worse. This phenomenon verifies that the uniform quantification of ZipML is unable to work on all datasets due to distinct distribution of gradients. In contrast, the convergence rate of SketchML is much better. SketchML reveals an ability of generalization across different datasets.

**Support Vector Machine.** According to the results in Figure 9(b), SketchML brings 4.59× and 3.88× improvements than Adam and ZipML. Compared with Logistic Regression and Linear Regression, Support Vector Machine is easier to get converged on this dataset. Therefore, as Figure 10(e) shows, the convergence rate of ZipML is faster than Adam due to faster communication. SketchML outperforms them significantly and is able to converge a tolerance in a shorter time.

**Linear Regression.** As illustrated in Figure 9(b), SketchML takes 32 seconds to train an epoch of Linear Regression, while Adam and ZipML need 97 and 78 seconds. Figure 10(f) shows the convergence of three approaches. ZipML is slower than Adam. As explained before, it is caused by the defect of uniform quantification. Overall, SketchML is the fastest and is able to converge to the same loss as Adam.

## 4.4 Model Accuracy

Since MinMaxSketch produces underestimated gradients, there is a doubt whether our method can correctly converge. We report the convergence performance over KDD12 dataset. The experiment settings are the same as Section 4.3. An algorithm is considered as converged if the variation of loss is less than 1% within five epochs.

As illustrated in Table 2, three methods can converge to almost the same model quality. However, SketchML converges much faster than other two systems. According to our analysis, MinMaxSketch causes underestimated gradients, while it dose not change the directions of all the gradient dimensions. If a specific dimension of a gradient is always underestimated, its convergence will be extremely slow. However, the dynamic learning rate and the grouping strategy in Section 3.3 solve this problem by giving larger learning rate for a slow dimension and reducing quantification error.

| | SketchML | Adam | ZipML |
|---|---|---|---|
| LR | 0.6885 / 8.1h | 0.6885 / 23h | 0.6887 / 11h |
| SVM | 0.9784 / 4.9h | 0.9785 / 23h | 0.9788 / 10h |
| Linear | 0.2111 / 4.8h | 0.2109 / 22h | 0.2111 / 9.4h |

**Table 2: Model Accuracy. The metric is minimal loss against converged time, separated by symbol "/". Run time is in hours.**

## 4.5 Scalability

Next, we assess the scalability of three methods. We change the number of used workers (executors) and study how the cluster size affects the performance. The results are presented in Figure 11. Due to the space constraint, we only provide the results on the KDD12 dataset here. The results on the CTR dataset are similar. We increase the number of workers (executors) from five to ten, then to fifty, and evaluate the average run time taken by each epoch.

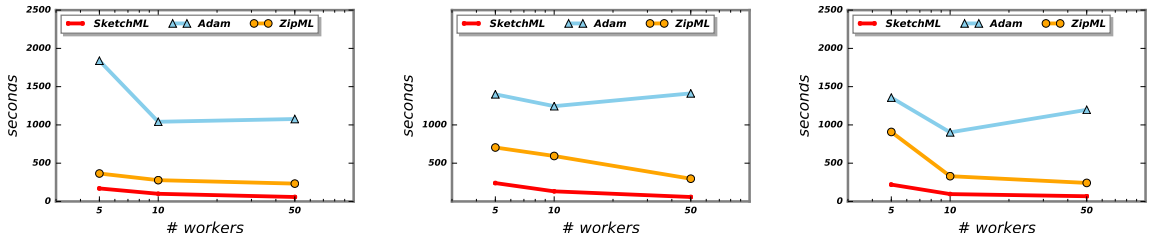
**Logistic Regression.** As shown in Figure 11(a), the performance of three methods increases when we increase the number of workers from five to ten, i.e., SketchML becomes 1.7× faster, Adam 1.8×, and ZipML 1.3×. Afterwards, we use fifty workers and find that Adam suffers a performance deterioration. The reason is that the increase of communication cost overwhelms the benefit of computation cost. SketchML and ZipML, to the contrary, achieve 1.7× and 1.6× improvements. Ideally, the performance speedup is five when we use fifty workers instead of ten. However, in a distributed environment, this ideal case is rare due to the influence of extra communication.

**Support Vector Machine.** For Support Vector Machine, the results are similar as Logistic Regression. Three methods become significantly faster when we increase the number of workers from five to ten. However, when we next use fifty workers, Adam unfortunately gets slower. SketchML and ZipML still benefit from the increase of workers. Their performance is improved up to 2.3× and 2×, respectively.

**Linear Regression.** With ten workers, all the three approaches are significantly faster in processing an epoch of Linear Regression than using five workers. And if we further use fifty workers, SketchML and ZipML even run faster. Nevertheless, Adam encounters a worse performance for the same reason we have discussed.

## 4.6 Summary

In summary, SketchML outperforms Adam and ZipML on a range of ML algorithms and datasets. SketchML consumes remarkably less time to execute an epoch. Although it needs more epoch to get converged, the overall performance still surpasses the other two competitors. Besides, SketchML reveals a good ability of scalability. We have also conducted more experiments to evaluate the performance of SketchML, including the performance of a single node system, the sensitivity against hyper-parameters, results on neural nets, and different wight types. Due to the space constraint, we put the experimental results and analysis in Appendix B.




(a) Logistic Regression, KDD12 (b) Support Vector Machine, KDD12 (c) Linear Regression, KDD12

**Figure 11: Scalability Comparison. The evaluated metric is the average run time per epoch in terms of the number of workers (executors). Run time is in seconds.**

**Limitation.** As stated in the introduction, our scenario has two properties — sparse gradients and communication-intensive workloads. Therefore, there are a few cases where our method is not such efficient. 1) For dense gradients, the value compression still works, but the key compression is redundant. 2) For computation-intensive workloads, the benefit of compression is not so significant.

## 5 RELATED WORK

Distributed machine learning (ML) has attracted more and more interests in recent years. Many machine learning algorithms are trained with the first-order-gradient optimization methods, stochastic gradient descent (SGD) in most cases [8]. To distribute SGD, a prevalent avenue is to partition the training dataset across a set of workers and let each worker calculate gradients independently. A coordinator then collects gradients from all the workers and updates the trained model. With the trend of increasing data size and model size, the phase of aggregating gradients becomes the main bottleneck of the system. To address this problem, many works have studied how to compress the gradients to save the communication cost.

Lossless compression methods have been widely used to compress integer data, such as digital images [18]. However, these methods cannot be used for floating-point gradients. A class of lossy methods was proposed to address this problem by transforming floating-point data to low-precision representations. Some approaches choose threshold based truncation [39] to encode floating-point data to one bit. But this strategy is too aggressive for SGD to get converged since a lot of gradients are abandoned. Another type of lossy method is the quantification based method such as ZipML [5, 30, 45]. Instead of using one bit, a quantification strategy transforms a floating-point number to an integer according to the value range of original data. Although this method seems a good solution as it achieves a tradeoff between efficient compression and correct convergence, it cannot fit the context of many large-scale ML cases. First, it is a general phenomenon that the transferred gradients are sparse. This is unsurprising since many trained datasets are high dimensional and sparse. To save space, we often store the nonzero elements in a gradient as key-value pairs where the key refers to a gradient dimension and the value refers to the corresponding dimension value. Existing quantification methods only compress values, therefore the compression performance is limited. Second, due to the data skew and complex slopes of the objective function, the distribution of gradient values is often nonuniform. Worse still, most gradient values locate in a small range near zero. The current quantification techniques assume that the processed data is uniformly distributed. They equally divide the value range into several intervals. Therefore, many small gradient values are quantified to zero, inducing a large quantization error.

The data sketch algorithm is an orthogonal technique that uses a small data structure to approximate the original data distribution. Currently, there are two categories of sketch algorithms, i.e., the quantile sketch and the frequency sketch. The quantile sketch takes a stream of items and produces a probabilistic data structure that depicts the value distribution of items. Different from quantification methods, a quantile sketch divides the value range into intervals such that each interval contains the same number of items. In this way, it can discover the pattern of a nonuniform distribution. As the most classical method, GK sketch and its variants are extensively used to conduct big data analytics [11, 16, 46]. The frequency sketch is designed to estimate the occurring frequency of items [12]. Specifically, the Count-Min sketch builds a few hash tables for the input items, and addresses the hash collision by an additive-and-minimum strategy. Although the existing sketch techniques are powerful in their targeted scenarios, they cannot be directly applied to compress gradient data. To the best of our knowledge, there is no work that uses the sketch algorithms to compress floating-point gradients to a low-precision representation and strengthen distributed machine learning workloads.

## 6 CONCLUSION

In this paper, in order to accelerate distributed machine learning, we proposed a sketch based method, namely SketchML, to compress the communicated key-value gradients. First, we introduced a method that uses a quantile sketch and a bucket sort to represent the gradient values with smaller binary encoded bucket indexes. Then, we designed a MinMaxSketch algorithm to approximately compress the bucket indexes. Further, we presented a delta-binary method to encode the gradient keys. We also theoretically analyzed the error bounds of proposed methods. Empirical results on a range of large-scale datasets and machine learning algorithms demonstrated that SketchML can be up to 10× faster than the state-of-the-art methods.

## ACKNOWLEDGEMENTS

This research is funded by National Natural Science Foundation of China (No. 61572039, 61702016, U1536201), 973 program (No. 2014CB340405), China Postdoctoral Science Foundation (2017M610019), and PKU-Tencent joint research Lab. Tong Yang is the corresponding author.

# REFERENCES


[1] [n. d.]. Data Sketches. https://datasketches.github.io/. ([n. d.]).
[2] [n. d.]. KDD Cup 2010. http://www.kdd.org/kdd-cup/. ([n. d.]).
[3] [n. d.]. KDD Cup 2012. https://www.kaggle.com/c/kddcup2012-track1. ([n. d.]).
[4] [n. d.]. MNIST. http://yann.lecun.com/exdb/mnist/. ([n. d.]).
[5] Dan Alistarh, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2016. QSGD: Randomized Quantization for Communication-Optimal Stochastic Gradient Descent. *arXiv preprint arXiv:1610.02132* (2016).
[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (2010), 50–58.
[7] Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Nvidia Corporation.
[8] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. 177–186.
[9] Léon Bottou. 2012. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*. 421–436.
[10] Sébastien Bubeck et al. 2015. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning* 8, 3-4 (2015), 231–357.
[11] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.
[12] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.
[13] Jeffrey Dean, Greg Corrado, Rajat Monga, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[14] L Peter Deutsch. 1996. DEFLATE compressed data format specification version 1.3. (1996).
[15] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12, Jul (2011), 2121–2159.
[16] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, Vol. 30. 58–66.
[17] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems* 29, 7 (2013), 1645–1660.
[18] Stuart C Hinds, James L Fisher, and Donald P D'Amato. 1990. A document skew detection method using run-length encoding and the Hough transform. In *Pattern Recognition, 1990. Proceedings., 10th International Conference on*, Vol. 1. IEEE, 464–468.
[19] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*. 1223–1231.
[20] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Vol. 398. John Wiley & Sons.
[21] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. 2015. Tencentrec: Real-time stream recommendation in practice. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 227–238.
[22] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 463–478.
[23] Jiawei Jiang, Ming Huang, Jie Jiang, and Bin Cui. 2017. TeslaML: Steering Machine Learning Automatically in Tencent. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*. Springer, 313–318.
[24] Jie Jiang, Jiawei Jiang, Bin Cui, and Ce Zhang. 2017. TencentBoost: A Gradient Boosting Tree System with Parameter Server. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. 281–284.
[25] Jiawei Jiang, Yunhai Tong, Hua Lu, Bin Cui, Kai Lei, and Lele Yu. 2017. GVoS: A General System for Near-Duplicate Video-Related Applications on Storm. *ACM Transactions on Information Systems (TOIS)* 36, 1 (2017), 3.
[26] Rie Johnson and Tong Zhang. 2013. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*. 315–323.
[27] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[28] Donald E Knuth. 1985. Dynamic huffman coding. *Journal of algorithms* 6, 2 (1985), 163–180.
[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[30] Mu Li, Ziqi Liu, Alexander J Smola, and Yu-Xiang Wang. 2016. DiFacto: Distributed Factorization Machines. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. ACM, 377–386.
[31] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 661–670.
[32] Brendan McMahan and Matthew Streeter. 2014. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems*. 2915–2923.
[33] Deanna Needell, Rachel Ward, and Nati Srebro. 2014. Stochastic gradient descent, weighted sampling, and the randomized kaczmarz algorithm. In *Advances in Neural Information Processing Systems*. 1017–1025.
[34] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. 2015. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807* (2015).
[35] Arkadi Nemirovski, Anatoli Juditsky, Guanghui Lan, and Alexander Shapiro. 2009. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization* 19, 4 (2009), 1574–1609.
[36] Yurii Nesterov. 1983. A method for unconstrained convex minimization problem with the rate of convergence O (1/kˆ 2). In *Doklady AN USSR*, Vol. 269. 543–547.
[37] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks* 12, 1 (1999), 145–151.
[38] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 936. John Wiley & Sons.
[39] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs.. In *INTERSPEECH*. 1058–1062.
[40] Johan AK Suykens and Joos Vandewalle. 1999. Least squares support vector machine classifiers. *Neural processing letters* 9, 3 (1999), 293–300.
[41] Reginald P Tewarson. 1973. *Sparse matrices*. Academic Press.
[42] Lele Yut, Ce Zhang, Yingxia Shao, and Bin Cui. 2017. LDA*: a robust and large-scale topic modeling system. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1406–1417.
[43] Matthew D Zeiler. 2012. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).
[44] Ce Zhang and Christopher Ré. 2014. DimmWitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1283–1294.
[45] Hantian Zhang, Kaan Kara, Jerry Li, Dan Alistarh, Ji Liu, and Ce Zhang. 2016. ZipML: An End-to-end Bitwise Framework for Dense Generalized Linear Models. *arXiv:1611.05402* (2016).
[46] Qi Zhang and Wei Wang. 2007. A fast algorithm for approximate quantiles in high speed data streams. In *Scientific and Statistical Database Management, 2007. SSBDM'07. 19th International Conference on*. IEEE, 29–29.
[47] Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma. 2009. Mining interesting locations and travel sequences from GPS trajectories. In *Proceedings of the 18th international conference on World wide web*. ACM, 791–800.
[48] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. 2010. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*. 2595–2603.
[49] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.


# A MATHEMATICAL ANALYSIS OF SKETCHML

In this section, we theoretically analyze the correctness and the error bound of the three components of SketchML.

## A.1 Quantile-Bucket Quantification

### A.1.1 Variance of Stochastic Gradients.

A series of existing works has indicated that stochastic gradient descent (SGD) suffers from a slower convergence rate than gradient descent (GD) due to the inherent variance [35]. To be precise, we refer to Theorem A.1.

THEOREM A.1. *(Theorem 6.3 of [10]) Let $f$ be convex and $\theta^*$ the optimal point. Choosing step length appropriately, the convergence rate of SGD is*

$$\mathbb{E}[f(\frac{1}{T}\sum_{t=1}^{T}\theta_{t+1}) - f(w^*)] \leqslant \Theta(\frac{1}{T} + \frac{\sigma}{T}),$$

*where $\sigma$ is the upper bound of mean variance*

$$\sigma^2 \geqslant \frac{1}{T}\sum_{t=1}^{T}\mathbb{E}||g_t - \nabla f(\theta_t)||^2.$$

A key property of a stochastic gradient is the variance. Many methods are applied to reduce the variance, such as mini-batch [31], weight sampling [33] and SVRG [26].

We refer $\tilde{g} = \{\tilde{g}_i\}_{i=1}^{d}$ to the quantificated gradient. Here we abuse the notation that in Theorem A.1 the subscript of $g_t$ indicates the $t$-th epoch to which it belongs, while in the following analysis that of $g_i$ indicates the $i$-th nonzero value of gradient. The variance of $\tilde{g}$ can be decomposed into

$$\mathbb{E}||\tilde{g} - \nabla f(\theta)||^2 \leqslant \mathbb{E}||\tilde{g} - g||^2 + \mathbb{E}||g - \nabla f(\theta)||^2.$$

The second term comes from the stochastic gradient, which can be reduced by the methods mentioned above. Our goal is to find out a quantification method to make the first term as small as possible.

#### A.1.2 Variance Bound of Quantile-Bucket Quantification.

In our framework, we use the quantile-bucket quantification method. For the sake of simplicity, we regard the maximum value in the gradient vector as the $q+1$-st quantile. The value range of gradients, denoted by $[\phi_{min}, \phi_{max}]$, is split into $q$ intervals by $q+1$ quantiles $v = \{v_j\}_{j=1}^{q+1}$. Since we separate positive and negative values and create one quantile sketch for each of them, we assume there is always a quantile split that equals to 0. Specifically, $\phi_{min} = v_1 < \cdots < v_{b_{zero}} = 0 < \cdots < v_{q+1} = \phi_{max}$. Also, we assume $[\phi_{min}, \phi_{max}] \subset [-1, 1]$, otherwise we can use $M(g) = ||g||$ as the scaling factor.

THEOREM A.2. *The variance $\mathbb{E}||\tilde{g} - g||^2$ introduced by quantile-bucket quantification is bounded by*

$$\frac{d}{4q}(\phi_{min}^2 + \phi_{max}^2),$$

*where $\phi_{min}$ and $\phi_{max}$ are the minimum and maximum values in the gradient vector: $\phi_{min} = \min\{g_i\}, \phi_{max} = \max\{g_i\}$.*

`Proof:` Using the quantiles as split values, the expected number of values that fall into the same interval should be $\frac{d}{q}$, and for each $g_i$,

$$(\tilde{g}_i - g_i)^2 = \Big(\frac{1}{2}(v_{b(i)} + v_{b(i+1)}) - g_i\Big)^2 \leqslant \frac{1}{4}(v_{b(i+1)} - v_{b(i)})^2,$$

where $b(i)$ is the index of bucket into which $g_i$ falls. Thus we have

$$\begin{aligned}
\mathbb{E}||\tilde{g} - g||^2 = \mathbb{E}[\sum_{i=1}^{d}(\tilde{g}_i - g_i)^2] &\leqslant \frac{d}{4q}\sum_{j=1}^{q}(v_{j+1} - v_j)^2 \\
&= \frac{d}{4q}\Big(\sum_{j=1}^{b_{zero}-1}(v_{j+1} - v_j)^2 + \sum_{j=b_{zero}}^{q}(v_{j+1} - v_j)^2\Big) \\
&\leqslant \frac{d}{4q}\Big(\Big(\sum_{j=1}^{b_{zero}-1}(v_{j+1} - v_j)\Big)^2 + \Big(\sum_{j=b_{zero}}^{q}(v_{j+1} - v_j)\Big)^2\Big) \\
&= \frac{d}{4q}(\phi_{min}^2 + \phi_{max}^2).
\end{aligned} \tag{1}$$

COROLLARY A.3. *When the distribution of gradients is not biased, i.e., there exists $\delta > 1$ such that $\frac{||v||^2}{v_1^2 + v_{q+1}^2} \geqslant \delta$, Equation (1) is bounded by $\frac{1}{4(\delta-1)}||g||^2$.*

`Proof:` Obviously $\phi_{min}^2 + \phi_{max}^2 = v_1^2 + v_{q+1}^2 \geqslant \frac{1}{\delta-1}\sum_{j=2}^{q} v_j^2$. Thus we have

$$\frac{d}{4q}(\phi_{min}^2 + \phi_{max}^2) \leqslant \frac{1}{4(\delta-1)}\sum_{j=2}^{q}\frac{d}{q}v_j^2 \leqslant \frac{1}{4(\delta-1)}||g||^2.$$

Considering the most widely used uniform quantification method, Alistarh et al. proved the bound of its variance is $\min(\frac{d}{q^2}, \frac{\sqrt{d}}{q})||g||^2$ [5]. Therefore quantile-bucket quantification generates a better bound when $d$ goes to infinite.

## A.2 MinMaxSketch

#### A.2.1 Error Bound of the MinMaxSketch.

Let $\alpha$ represent the average number of counters in any given array of the MinMaxSketch that are incremented per insertion. Note that for the standard CM-sketch, the value of $\alpha$ is equal to 1 because in the standard CM-sketch, exactly one counter is incremented in each array when inserting an item. For the MinMaxSketch, $\alpha$ is less than or equal to 1. For any given item $e$, let $f_{(e)}$ represent its actual frequency and let $\hat{f}_{(e)}$ represent the estimate of its frequency returned by the MinMaxSketch. Let $N$ represent the total number of insertions of all items into the MinMaxSketch. Let $h_i(.)$ represent the hash function associated with the $i^{\text{th}}$ array of the MinMaxSketch, where $1 \leqslant i \leqslant d$. Let $X_{i,(e)}[j]$ be the random variable that represents the difference between the actual frequency $f_{(e)}$ of the item $e$ and the value of the $j^{\text{th}}$ counter in the $i^{\text{th}}$ array, *i.e.*, $X_{i,(e)}[j] = A_i[j] - f_{(e)}$, where $j = h_i(e)$. Due to hash collisions, multiple items will be mapped by the hash function $h_i(.)$ to the counter $j$, which increases the value of $A_i[j]$ beyond $f_e$ and results in over-estimation error. As all hash function have uniformly distributed output, $Pr[h_i(e_1) = h_i(e_2)] = 1/w$. Therefore, the expected value of any counter $A_i[j]$, where $1 \leqslant i \leqslant d$ and $1 \leqslant j \leqslant w$, is $\alpha N/w$. Let $\epsilon$ and $\delta$ be two numbers that are related to $d$ and $w$ as follows: $d = \lceil \ln(1/\delta) \rceil$ and $w = \lceil \exp/\epsilon \rceil$. The expected value of $X_{i,(e)}[j]$ is given by the following expression.

$$E(X_{i,(e)}[j]) = E(A_i[j] - f_{(e)}) \leqslant E(A_i[j]) = \frac{\alpha N}{w} \leqslant \frac{\epsilon\alpha}{\exp}N.$$

Finally, we derive the probabilistic bound on the over-estimation error of the MinMaxSketch.

$$\begin{aligned}
Pr[\hat{f_{(e)}} \geqslant f_{(e)} + \epsilon\alpha N] &= Pr[\forall i, A_i[j] \geqslant f_{(e)} + \epsilon\alpha N] \\
&= (Pr[A_i[j] - f_{(e)} \geqslant \epsilon\alpha N])^d \\
&= (Pr[X_{i,(e)}[j] \geqslant \epsilon\alpha N])^d \\
&\leqslant (Pr[X_{i,(e)}[j] \geqslant \exp E(X_{i,(e)}[j]))^d \\
&\leqslant \exp^{-d} \leqslant \delta.
\end{aligned}$$

#### A.2.2 The Correctness Rate of the MinMaxSketch.

Next, we theoretically derive the correctness rate of the MinMaxSketch, which is *defined as the expected percentage of elements in the multiset for which the query response contains no error*. In deriving the correctness rate, we make one assumption: all hash functions are pairwisely independent. Before deriving this correctness rate, we first prove following theorem.

THEOREM A.4. *In the MinMaxSketch, the value of any given counter is equal to the frequency of the least frequent element that maps to it.*

`Proof`: We prove this theorem using mathematical induction on number of insertions, represented by $k$.

**Base Case $k$ = 0:** The theorem clearly holds for the base case, because before the insertions, the frequency of the least frequent element is 0, which is also the value of all counters.

**Induction Hypothesis $k = n$:** Suppose the statement of the theorem holds true after $n$ insertions.

**Induction Step $k = n + 1$:** Let $n + 1^{\text{st}}$ insertion be of any element $e$ that has previously been inserted $a$ times. Let $\alpha_i(k)$ represent the values of the counter $F_i[h_i(e)\%w]$ after $k$ insertions, where $0 \leqslant i \leqslant d-1$. There are two cases to consider: 1) $e$ was the least frequent element when $k = n$; 2) $e$ was not the least frequent element when $k = n$.

***Case 1:*** If $e$ was the least frequent element when $k = n$, then according to our induction hypotheses, $\alpha_i(n) = a$. After inserting $e$, it will still be the least frequent element and its frequency increases to $a+1$. As per our MinMaxSketch scheme, the counter $F_i[h_i(e)\%w]$ will be incremented once. Consequently, we get $\alpha_i(n+1) = a+1$. Thus for this case, the theorem statement holds because the value of the counter $F_i[h_i(e)\%w]$ after insertion is still equal to the frequency of the least frequent element, which is $e$.

***Case 2:*** If $e$ was not the least frequent element when $k = n$, then according to our induction hypotheses, $\alpha_i(n) > a$. After inserting $e$, it may or may not become the least frequent element. If it becomes the least frequent element, it means that $\alpha_i(n) = a + 1$ and as per our MinMaxSketch scheme, the counter $F_i[h_i(e)\%w]$ will stay unchanged. Consequently, we get $\alpha_i(n+1) = \alpha_i(n) = a+1$. Thus for this case, the theorem statement again holds because the value of the counter $F_i[h_i(e)\%w]$ after insertion is equal to the frequency of the new least frequent element, which is $e$.

After inserting $e$, if it does not become the least frequent element, then it means $\alpha_i(n) > a + 1$ and as per our the MinMaxSketch scheme, the counter $F_i[h_i(e)\%w]$ will stay unchanged. Consequently, $\alpha_i(n + 1) = \alpha_i(n) > a + 1$. Thus, the theorem again holds because the value of the counter $F_i[h_i(e)\%w]$ after insertion is still equal to the frequency of the element that was the least frequent after $n$ insertions. □

Next, we derive the correctness rate of the MinMaxSketch. Let $v$ be the number of distinct elements inserted into the MinMaxSketch and are represented by $e_1, e_2, \ldots, e_v$. Without loss of generality, let the element $e_{l+1}$ be more frequent than $e_l$, where $1 \leqslant l \leqslant v-1$. Let $X$ be the random variable representing the number of elements hashing into the counter $F_i[h_i(e_l)\%w]$ given the element $e_l$, where $0 \leqslant i \leqslant d-1$ and $1 \leqslant l \leqslant v$. Clearly, $X \sim \text{Binomial}(v-1, 1/w)$.

From Theorem 1, we conclude that if $e_l$ has the highest frequency among all elements that map to the given counter $F_i[h_i(e_l)\%w]$, then the query result for $e_l$ will contain no error. Let $A$ be the event that $e_l$ has the maximum frequency among $x$ elements that map to $F_i[h_i(e_l)\%w]$. The probability $P\{A\}$ is given by the following equation:

$$P\{A\} = \binom{l-1}{x-1} / \binom{v-1}{x-1.} \qquad (\text{where } x \leqslant l)$$

Let $P'$ represent the probability that the query result for $e_l$ from any given counter contains no error. It is given by:

$$P' = \sum_{x=1}^{l} P\{A\} \times P\{X = x\}$$
$$= \sum_{x=1}^{l} \frac{\binom{l-1}{x-1}}{\binom{v-1}{x-1}} \binom{v-1}{x-1} \left(\frac{1}{w}\right)^{x-1} \left(1-\frac{1}{w}\right)^{v-x} = \left(1-\frac{1}{w}\right)^{v-l}.$$

As there are $d$ counters, the overall probability that the query result of $e_l$ is correct is given by the following equation.

$$P_{\text{CR}}\{e_l\} = 1 - \left(1 - \left(1-\frac{1}{w}\right)^{v-l}\right)^{d}.$$

The equality above holds when all $v$ elements have different frequencies. If two or more elements have equal frequencies, the correctness rate increases slightly. Consequently, the expected correctness rate $Cr$ of the MinMaxSketch is bound by:

$$Cr \geqslant \frac{\sum_{l=1}^{v} P_{\text{CR}}\{e_l\}}{v} = \frac{\sum_{l=1}^{v} \left(1 - \left(1 - (1-\frac{1}{w})^{v-l}\right)^{d}\right)}{v}. \quad (2)$$

## A.3 Delta-Binary Encoding

Delta-Binary Encoding is a lossless compression method, but its average space cost cannot be calculated exactly. Here we focus on the expected size for one key. As aforementioned, we divide all the quantile buckets into $r$ groups. Therefore, the number of nonzero keys that fall into the same group is expected to be $\frac{d}{r}$. Assuming the arrangement of dimensions in dataset is random, the expected difference between two keys should be $\frac{rD}{d}$. As a result, the expected bytes for each key is $\lceil \log_{256} \frac{rD}{d} \rceil = \lceil \frac{1}{8} \log_2 \frac{rD}{d} \rceil$. For instance, with $r = 8$, we can compress each key into 1 byte if we choose a large batch size such that $\frac{d}{D} \geqslant \frac{1}{32}$.

Fortunately, the arrangement of dimensions in dataset is usually not random, i.e., dimensions with strong relationship happen to appear in consecutive keys, which makes the difference between two nonzero keys smaller. In practice, we find that the average size for one key (including two flag bits) is around 1.5 bytes.

Considering bitmap, another useful data structure for storing keys with compression rate up to 8. Nonetheless, in our framework bitmap is not so useful as it should be. In order to indicate the keys for different groups, we have to create one bitmap for each of them, which comes out with $\lceil \frac{rD}{8} \rceil$ bytes in total. As a result, *Delta-Binary Encoding* is a better choice.

# B MORE EXPERIMENTS

## B.1 Comparison with A Single Node System

To give a reference point of performance, we compare SketchML with SkLearn, a state-of-the-art single node system on Cluster-1. Due to the memory constraint, we choose KDD10 dataset. SkLearn is executed on a single machine, while SketchML is executed on five and ten machines, denoted by SketchML-5 and SketchML-10. The other settings are the same as Section 4.2. Figure 12 shows the run time of twenty epochs. SketchML-5 is 2.1×, 2.7×, and 2× faster than SkLearn in training three algorithms. SketchML-10 further brings 1.3×, 1.6×, and 1.5× speedup compared with SketchML-5.

Although the distribution of SketchML brings nonnegligible communication overhead, it still outperforms SkLearn as a result of computation speedup, message compression, and faster data loading. For example, SkLearn consumes more than ten minutes to load the dataset owing to slow disk I/O. Using five machines reduces the

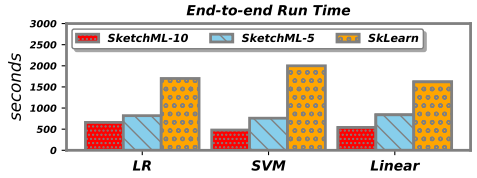



**Figure 12: Comparison with SkLearn. (KDD10, LR)**

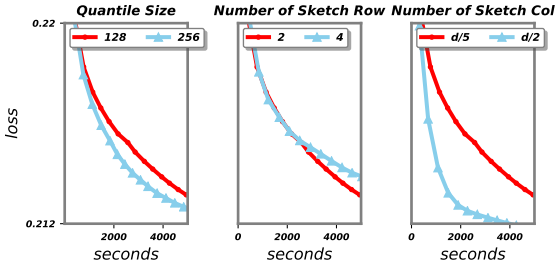



**Figure 13: Sensitivity (KDD12, Linear, Convergence).**

time of data loading to two minutes. For a small dataset, a single machine is enough in many cases. However, for a large dataset, a single machine is often impracticable owing to expensive data loading, insufficient memory capacity, and limited computation power.

## B.2 Sensitivity Against Hyper Parameters

SketchML contains three hyper-parameters — the size of quantile sketch (default 128), the row of MinMaxSketch (default 4), and the column of MinMaxSketch (default $\frac{d}{5}$). Here, we vary their values and investigate the sensitivity of our method. We run KDD12 dataset to train a linear regression model on Cluster-2 and use the same setting as Section 4.3.1.

| | default | quan_256 | row_4 | col_d/2 |
|---|---|---|---|---|
| Run time | 360 | 353 | 420 | 383 |

**Table 3: Sensitivity (KDD12, Linear, second per epoch).**

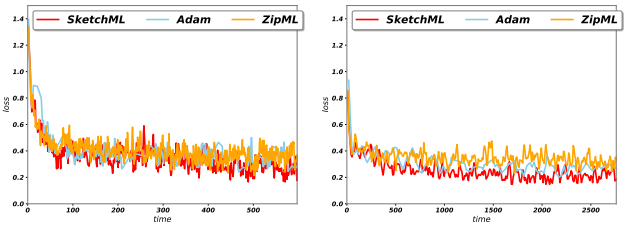
**Size of Quantile Sketch.** As shown in Figure 13, a larger size accelerates the training because the quantization error is reduced. According to Table 3, the time consumed by each epoch is not obviously affected.

**Row of MinMaxSketch.** We next study the influence of the row of MinMaxSketch, i.e., the number of hash tables. More hash tables can reduce the possibility of hash collision, at the expense of more communication cost. Therefore, the convergence is slower when we increase the number of rows to four.

**Column of MinMaxSketch.** The default column is $\frac{d}{5}$ where $d$ is the number of nonzero gradient items. Increasing the number of column from $\frac{d}{5}$ to $\frac{d}{2}$ brings less efficient, yet more accurate, compression. Overall, the convergence performance is significantly enhanced.

## B.3 Experiment on Neural Nets

The above evaluated algorithms belong to generalized linear models. However, our Sketch mechanism can be applied on Neural Network models, such as multilayer perceptron (MLP) and Convolutional Neural Networks (CNN) by transferring gradients with our compression method. In this section, we train an MLP model




(a) Short-Term Convergence (b) Long-Term Convergence

**Figure 14: Performance on Neural Nets.**

as a representative on MNIST [4], which consists of 60000 training images and 10000 testing images. The network is composed of one input layer (size: $20 \times 20$), two fully connected layers (size: 600), and one output layer (size: 10). The batch size is 0.1% of the dataset, i.e., 60 images. The learning rate is 0.005.

As shown in Figure 14(a), SketchML and ZipML converge faster than Adam at the beginning of the training phase. In a long term, SketchML achieves the fastest convergence rate and the smallest loss, followed by Adam. Though MLP is a nonlinear model, SketchML outperforms Adam because MinMaxSketch reduces the gradient variance and therefore avoids model oscillation. ZipML, however, cannot perform well in a long-term training. Since gradients become smaller as the training proceeds, ZipML quantizes many of them to zero and causes slow convergence.

We can also see that although SketchML still performs better NN models, the performance gap is not as big as that on Linear Models. As stated in the introduction, our scenario has two properties — sparse gradients and communication-intensive workloads. For dense gradients, the value compression still works, but the key compression is redundant; and for computation-intensive workloads, the benefit of compression is not that significant.

## B.4 Weight Type of Compression

In Section 4.3, we use 16 bits for ZipML and double type for Adam. Here, we evaluate more weight types and show the result in Table 4. The experimental settings are the same as Section 4.3.1.

| SketchML | ZipML-8bit | ZipML-16bit | Adam-float | Adam-double |
|---|---|---|---|---|
| 100 | 231 | 278 | 725 | 1041 |
| 0.6905 | 0.6932 | 0.6919 | 0.6911 | 0.6914 |

**Table 4: Different Weight Types (KDD12, LR). The first row denotes the run time per epoch in seconds. The second row denotes the minimal loss after two hours.**

ZipML runs 1.2× faster with 8 bits than 16 bits. However, as stated in Section 4.1, ZipML converges badly even if we fine tune the learning rate. Adam with float-type converges 1.4× faster than the double-type counterpart. The performance improvement is unsurprising since the communication cost is reduced. SketchML achieves the fastest speed — 2.3× and 7.1× faster than ZipML and Adam. Within the same time, i.e., two hours, SketchML can converge to the smallest loss compared with other four competitors, verifying the fast convergence of SketchML.