The VLDB Journal
https://doi.org/10.1007/s00778-019-00596-3

REGULAR PAPER

SKCompress: compressing sparse and nonuniform gradient in distributed machine learning

Jiawei Jiang[1,2] · Fangcheng Fu[1,3] · Tong Yang[4,5] · Yingxia Shao[6] · Bin Cui[7]

Received: 16 December 2018 / Revised: 29 November 2019 / Accepted: 12 December 2019

Abstract

Distributed machine learning (ML) has been extensively studied to meet the explosive growth of training data. A wide range of machine learning models are trained by a family of first-order optimization algorithms, i.e., stochastic gradient descent (SGD). The core operation of SGD is the calculation of gradients. When executing SGD in a distributed environment, the workers need to exchange local gradients through the network. In order to reduce the communication cost, a category of quantification-based compression algorithms are used to transform the gradients to binary format, at the expense of a low precision loss. Although the existing approaches work fine for dense gradients, we find that these methods are ill-suited for many cases where the gradients are sparse and nonuniformly distributed. In this paper, we study *is there a compression framework that can efficiently handle sparse and nonuniform gradients?* We propose a general compression framework, called SKCompress, to compress both gradient values and gradient keys in sparse gradients. Our first contribution is a sketch-based method that compresses the gradient values. Sketch is a class of algorithm that approximates the distribution of a data stream with a probabilistic data structure. We first use a quantile sketch to generate splits, sort gradient values into buckets, and encode them with the bucket indexes. Our second contribution is a new sketch algorithm, namely MinMaxSketch, which compresses the bucket indexes. MinMaxSketch builds a set of hash tables and solves hash collisions with a MinMax strategy. Since the bucket indexes are nonuniform, we further adopt Huffman coding to compress MinMaxSketch. To compress the keys of sparse gradients, the third contribution of this paper is a delta-binary encoding method that calculates the increment of the gradient keys and encode them with binary format. An adaptive prefix is proposed to assign different sizes to different gradient keys, so that we can save more space. We also theoretically discuss the correctness and the error bound of our proposed methods. To the best of our knowledge, this is the first effort utilizing data sketch to compress gradients in ML. We implement a prototype system in a real cluster of our industrial partner Tencent Inc. and show that our method is up to 12× faster than the existing methods.

Keywords Distributed machine learning · Stochastic gradient descent · Quantification · Quantile sketch · Frequency sketch · Huffman coding

Jiawei Jiang and Fangcheng Fu contributed equally to this work.

✉ Tong Yang
yangtongemail@gmail.com

Jiawei Jiang
blue.jwjiang@pku.edu.cn ; jiawei.jiang@inf.ethz.ch

Fangcheng Fu
ccchengff@pku.edu.cn ; fangchengfu@tencent.com

Yingxia Shao
shaoyx@bupt.edu.cn

Bin Cui
bin.cui@pku.edu.cn

[1] School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University, Beijing, China

[2] Department of Computer Science, ETH Zürich, Zürich, Switzerland

[3] Department of Data Platform, TEG, Tencent Inc., Beijing, China

[4] Department of Computer Science and Technology, Peking University, Beijing, China

[5] Peng Cheng Laboratory, Shenzhen, China

[6] School of Computer Science & Beijing Key Lab of Intelligent Telecommunications Software and Multimedia, Beijing University of Posts and Telecommunications, Beijing, China

[7] Department of Computer Science and Technology & Key Laboratory of High Confidence Software Technologies (MOE), Peking University, Beijing, China

Published online: 01 January 2020

1 Introduction

1.1 Background and motivation

Machine learning (ML) techniques have been widely used in many applications, such as recommendation [19], text mining [59], image recognition [30], video detection [23], smart urban computing [65], and more [4,31,53,64]. With the proliferation of training datasets, a centralized system is unable to run ML tasks efficiently. Therefore, it is inevitable to deploy ML in a decentralized environment [21]. We focus on a subclass of ML models, such as logistic regression [18], support vector machine [49], linear regression [46], and neural network [30]. Generally, they are trained with a widely used family of first-order gradient optimization methods, namely stochastic gradient descent (SGD) [5,66]. To distribute these gradient-based algorithms, we partition a training dataset over workers. The workers independently propose gradients [11,20].

Under such setting, a major problem is the heavy communication because the workers need to exchange gradients with each other. The communication cost often dominates the total cost. Although the network infrastructure is becoming faster and faster nowadays, reducing gradient movement is still beneficial in many fields we try to support, including but not limited to the following cases.

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 and produces a better prediction [22]. However, a large model also brings considerable communications in a distributed cluster, which impedes the overall performance. Motivated as such, it is non-trivial 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 [2]. 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 the network.

Case 3: Geo-distributed ML For many international companies, it is infeasible to move data between data centers before running ML algorithms. Generally, data movement over wide-area-network (WAN) is much slower than local-area-network (LAN). Reducing the communication cost 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 [15]. 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. The existing compression approaches can be summarized into two categories—lossless methods and lossy methods.

Lossless methods for repetitive integer data, such as Huffman coding, run-length encoding (RLE), DEFLATE, and Rice [12,16,29,67], 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 [3,51], but the performance improvement is not large enough due to limited compression performance.

Lossy methods are proposed to compress floating-point gradients by a sparsification-based strategy [35,47] or a quantification-based strategy [1,32,54,62]. The sparsification approaches filter large gradients according to a threshold. Some of them accumulate small gradients locally until reaching the threshold. But the accumulated gradients, which are stale, might harm the convergence. Instead, some other methods abandon small gradients, at the risk of losing useful information, especially for skewed datasets. At a high level, the quantification approach is more promising since it achieves a trade-off between compression performance and convergence performance. 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. If we store all the dimensions of a sparse gradient vector and compress all of them, a lot of time is wasted on zero gradient values. If we store a sparse gradient vector in (key, value) pairs, the gradient keys cannot be compressed. (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 SKCompress, a general compression framework that supports sparse gradients and fits the statistical distribution of gradients. Briefly

speaking, for a sparse gradient vector consisting of key-value pairs, denoted by $\{(k_j, v_j)\}_{j=1}^{d}$, we use a novel sketch-based algorithm to compress gradient values $\{v_j\}_{j=1}^{d}$ and a delta-binary encoding method to compress gradient keys $\{k_j\}_{j=1}^{d}$. They bring an improvement over state-of-the-art algorithms of $2-12\times$. 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 θ in the opposite direction [6]: $\theta = \theta - \eta g_i$ where η is a hyper-parameter called the learning rate. Note that $g_i \in \mathbb{R}^D$ is generally a sparse vector when the training dataset is sparse. 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 [11]. 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, we try to use other types of data structure that can approximate the distribution of data. We consider an alternative, called the sketch algorithm, which is widely used to analyze a stream of data. The existing sketch algorithms include the quantile sketch [8,14] and the frequency sketch [10]. 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 read the gradient values and generate several quantile splits. With the splits, we summarize the gradient values into several buckets and then encode each value by the corresponding bucket index $b(v_j)$. As each bucket index is an integer, we still need four bytes for each of them. 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. The frequency sketch might unpredictably increase the gradient values in the query phase, causing unstable convergence. To address this problem, we propose a novel sketch algorithm, called MinMaxSketch. MinMaxSketch encodes the bucket indexes using a multiple-hashing approximation. It employs 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. Another potential problem of MinMaxSketch is the nonuniform distribution of bucket indexes, which is incurred by the Min protocol in the insertion phase. We further encode the bucket indexes with Huffman coding.

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 θ. 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 transform the keys to *delta keys*. Specifically, each delta key stores 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 then transform each delta key to a binary representation, with the minimal byte that is enough to hold it. According to our empirical results, each delta key only consumes an average of about 1.27 bytes—$3.2\times$ smaller for a four-byte integer or $6.3\times$ for an eight-byte long-integer. In this binary transformation, a prefix is created to indicate the number of bits consumed by each gradient key. The above method uses a fixed-length prefix. However, this is not ideal since the distribution of delta keys is nonuniform that small delta keys appear more frequently. In order to better fit this property, we propose an adaptive approach that leverages a statistical cost model to choose the best prefix scheme from the fixed-length candidate and the Huffman coding candidate.

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 Inc., we use two large-scale datasets to run a range of ML workloads. Our proposed framework SKCompress is $2-12\times$ faster than the state-of-the-art approaches.

Roadmap The rest of this paper is organized as follows. We introduce the preliminary in Sect. 2. We give the overview of SKCompress in Sect. 3, describe the compression of gradient values in Sect. 4, and describe the compression of gradient keys in Sect. 5. Section 6 analyzes the space cost. We show the experimental results in Sect. 7, describe related work in Sect. 8, and conclude this work in Sect. 9. We also present the theoretical proof of SKCompress in "Appendix A."

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): jth 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 cost makes it infeasible for a large volume of items.

Quantile sketch uses 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 [14]. 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 [56] 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 [14]. Some works also design extensions of the GK algorithm [8,14,63]. 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 [10,57], as shown in Fig. 1. Essentially, count-min sketch is similar to the principle of Bloom filter [58]. 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 fol-

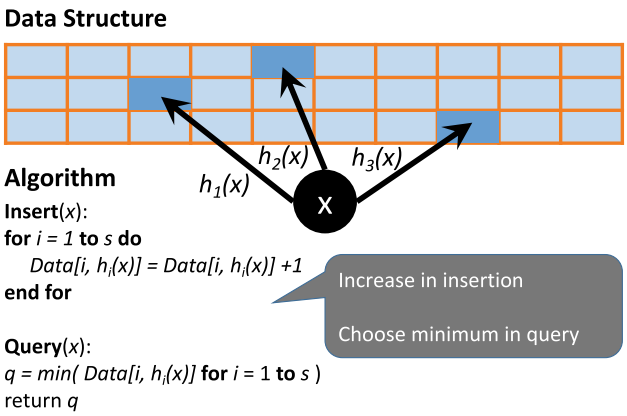


Fig. 1 An example of count-min sketch

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

Despite 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 to or larger than the true frequency $\widetilde{q}$ due to the possibility of hash collision. Therefore, the minimum operation of frequency sketch chooses the one closest to $\widetilde{q}$.

3 The overview of SKCompress

We first walk through an overview of the framework in this section and then describe each component individually in the following sections.

Figure 2 illustrates the overview of our proposed framework SKCompress. There are five major components in the framework, i.e., quantile-bucket quantification, MinMaxSketch, Huffman coding, dynamic delta-binary encoding, and adaptive prefix. The first three components together compress the gradient values, while the fourth and the fifth components together compress the gradient keys.

Encode phase The framework performs encoding as follows:

1. Quantile sketch is used to generate candidate splits, with which we use bucket sort to summarize the gradient values.
2. The gradient values are represented by the bucket indexes.
3. The bucket indexes are inserted into the MinMaxSketch by applying the hash functions on the keys.

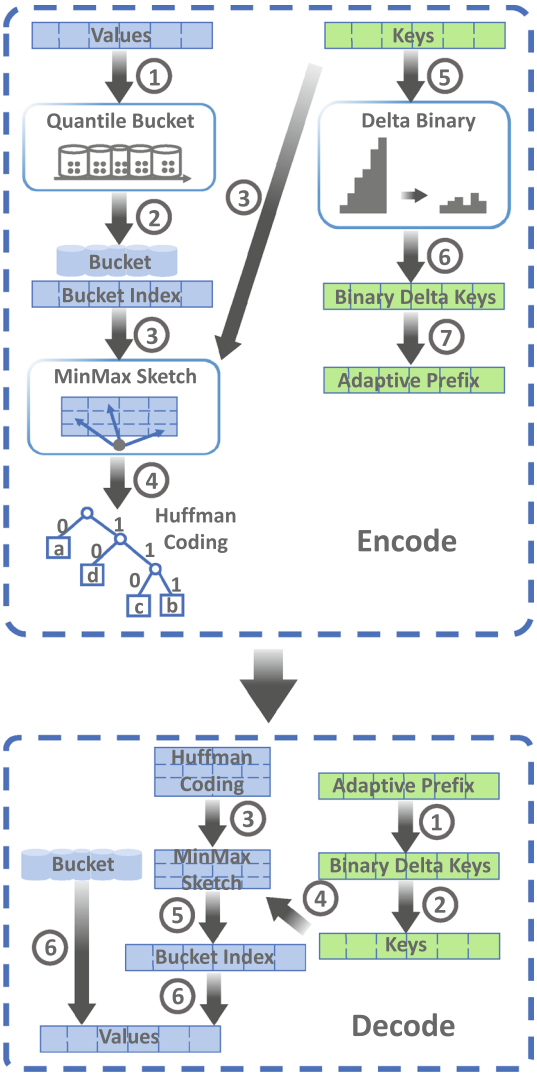


Fig. 2 The framework overview of SKCompress

4. We use Huffman coding to encode the items in the MinMaxSketch.
5. The keys are transformed into their increments, denoted by delta keys in this paper.
6. We use binary encoding to encode the delta keys with flexible bytes, instead of using fixed four-bytes. Each encoded delta key has a fixed-length prefix indicating the consumed bytes.
7. We further convert each fixed-length prefix to an adaptive prefix.

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

1. The adaptive prefixes are recognized so that each delta key can be identified.
2. The delta keys are recovered to the original gradient keys.

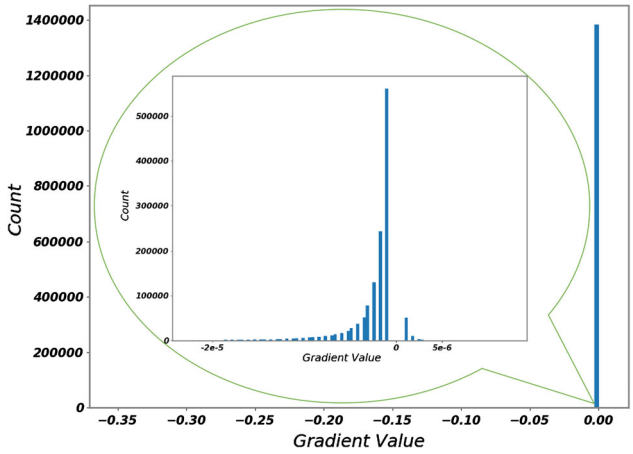


Fig. 3 An example of nonuniform gradient values

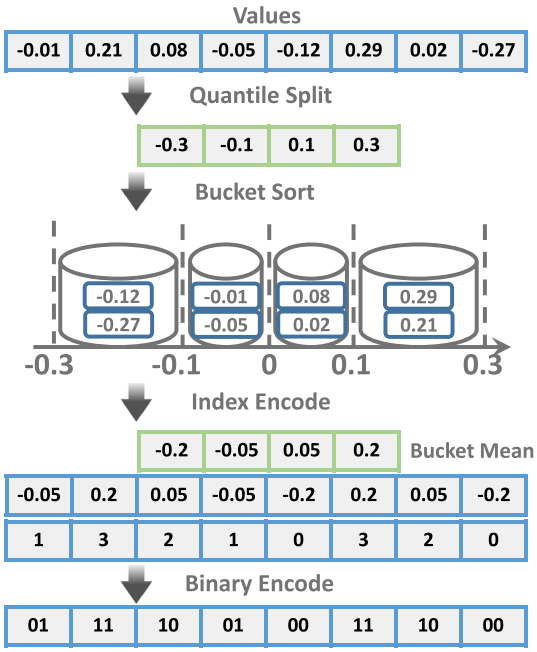


Fig. 4 The module of quantile-bucket quantification

3. The items in MinMaxSketch are decoded according to Huffman coding.
4. The recovered gradient keys are used to query the MinMaxSketch.
5. The bucket index of each gradient value is obtained from the MinMaxSketch.
6. The gradient value is recovered by choosing the bucket value with the bucket index.

4 Compression of gradient values

In this section, we introduce the mechanism of compressing gradient values, including three components—quantile-bucket quantification, MinMaxSketch, and Huffman coding for MinMaxSketch.

4.1 Quantile-bucket quantification

The component of quantile-bucket quantification compresses the gradient values $\{v_j\}_{j=1}^{d}$.

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 [32,38]. For example, SGD 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 afterward [6].

Driven by the requirement of robustness against noises, we ask *can optimization algorithms converge with quantified low-precision gradients?* Intuitively, since SGD can converge with random noises, low-precision gradients are able to work as well. Compared with unpredictable noises, the quantification error is usually controllable and bounded [1]. 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 [62]. However, uniform quantification is ill-suited for gradients. Figure 3 is an example of the distribution of gradient values. We train a public dataset [25] 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. 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 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 consists of three steps.

Step 1: Quantile split We build a quantile sketch with the gradient values and generate quantile splits, as shown in Fig. 4.

1. We scan all the gradient values and insert them into a quantile sketch. Here, we choose Yahoo DataSketches [56], a state-of-the-art quantile sketch.
2. q quantiles are used to get candidate splits from the quantile sketch. Detailed, we generate q averaged quantiles $\{0, \frac{1}{q}, \frac{2}{q}, \ldots, \frac{q-1}{q}\}$.
3. We use the generated quantiles and the maximal value as candidate split values, denoted by $\{rank(0), rank(\frac{1}{q}), rank(\frac{2}{q}), \ldots, rank(1)\}$. 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 Fig. 4 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.

Step 3: Index encode Although we quantify gradient values with bucket mean values, the consumed space remains the same. 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 by encoding them to binary numbers. If $q = 256$, one byte is enough to encode 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 $q \ll d$ 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 the 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 encoded to $(k_j, b(v_j))$ where $b(v_j)$ denotes the binary bucket index. In practice, we find that $q = 256$ is often enough to obtain comparable prediction accuracy.

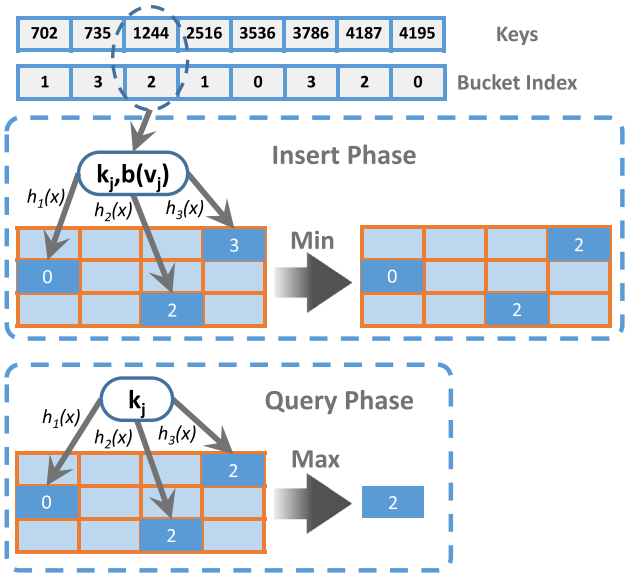


Fig. 5 MinMaxSketch. The insert phase uses a Min strategy, and the query phase uses a Max strategy

4.2 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. 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 [10]. 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 a 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. 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 Fig. 5, there are three hash functions, $h_1(-)$, $h_2(-)$, and $h_3(-)$.
3. Once a hash bin is chosen in the ith 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 in hash bin and therefore avoid the increase in 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 Fig. 5, three candidates are {0, 2, 2}, and 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 type of 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 [10]. 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.

As the readers might suspect, can optimization algorithms converge with underestimated gradients? Theoretically, optimization algorithms such as SGD move toward 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, an uncontrolled increase in 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.

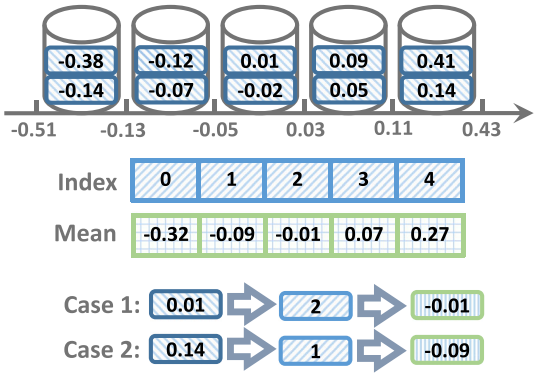


Fig. 6 An example of reversed gradient. There are two cases.

- **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 Fig. 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 gradients 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** Due to the data skewness, different dimensions of the trained model converge at a different speed. Adaptive learning rate methods such as Adam and AMSGrad [27,44] are proposed to solve this convergence imbalance by scheduling the learning rates to be inversely proportional to the historical gradients. Motivated by this, we introduce to solve the problem of vanishing gradient via an adaptive learning rate method. In "Appendix C," we show that both Adam and AMSGrad can help SKCompress converge faster. Since Adam and AMSGrad show similar convergence in our experiments, we choose Adam as the optimizer of our method, which has also been adopted by many research studies [28,34].
- **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. The adaptive learning rate is unable to fully compensate the decline of decoded bucket index. To address this problem, we design a group strategy for MinMaxSketch throughout the training. 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 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.

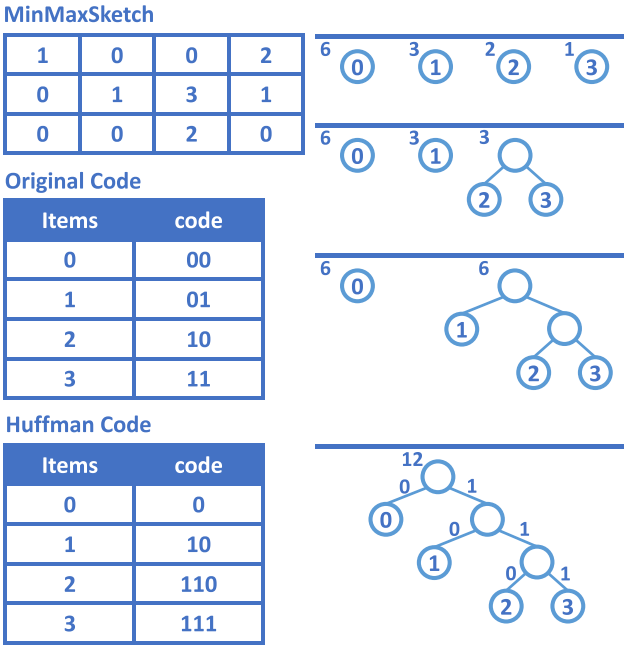


Fig. 7 An example of Huffman coding for MinMaxSketch

4.3 Huffman coding for MinMaxSketch

Although MinMaxSketch has significantly compressed gradient values, we proceed to use the technique of Huffman coding to compress MinMaxSketch.

Motivation Each item in MinMaxSketch takes a fixed-length space. In particular, we often use $q = 256$ meaning that each item needs one byte. If the values of items are uniformly distributed, giving them a fixed length is convincing. *However, are items in MinMaxSketch uniformly distributed?*

Since quantile sketch assures that each bucket contains the same number of gradient values, the bucket indexes are uniformly distributed before building MinMaxSketch. However, owing to the `Min` protocol in the insertion phase of MinMaxSketch, the distribution of items changes and smaller values occur more. With this nonuniform setting, the original method is not ideal. We propose to use Huffman coding to encode items in MinMaxSketch.

Step 1: Frequency of occurrence During the insertion operation of MinMaxSketch, we summarize the occurrence frequencies of all the items. Figure 7 showcases an example. The frequencies of four items are 6, 3, 2, and 1, respectively. The value range of item is [0,3] so that we use two bits to encode them. For example, the original code of 1 is 0b01, and that of 3 is 0b11. Note that since MinMaxSketch consists of several rows, we can parallelize this process with multi-threading.

Step 2: Build Huffman tree We build the Huffman tree with the frequencies of items. The construction algorithm uses a priority queue where the item with the lowest frequency is given the highest priority. The procedure is presented below:

1. Create a leaf node for each item and add it to the priority queue.
2. While there is more than one node in the queue:
   2-1. Remove the two nodes of highest priority (lowest frequency) from the queue.
   2-2. Create a new internal node with these two nodes as children. The frequency of the new node is equal to the sum of the two nodes' frequencies.
   2-3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Step 3: Generate Huffman code Once the Huffman tree is set, we assign different bits to different branches, i.e, 0 to the left branch and 1 to the right branch. Then, we can directly obtain the Huffman code of each item according to their locations in the tree. Specifically, the example in Fig. 7 encodes item 0 as 0b0, item 1 as 0b10, item 2 as 0b110, and item 3 as 0b111.

Summary As shown in Fig. 7, the original space cost of MinMaxSketch is 24 bits, while the space cost after Huffman coding is 21 bits, bringing a 12.5% improvement. With more small items, the performance improvement will be more obvious.

5 Compression of gradient keys

In this section, we introduce the mechanism of compressing gradient keys, including two components—delta-binary encoding and adaptive prefix.

5.1 Delta-binary encoding

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

Motivation For floating-point numbers and integer numbers, we can use low-precision compression methods if they can bear a 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.

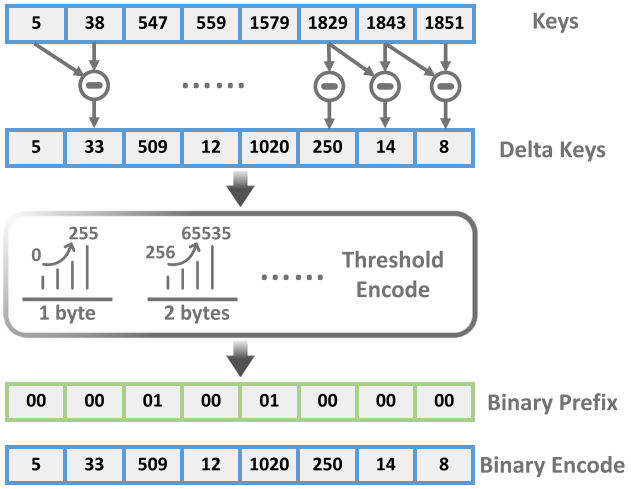


Fig. 8 An example of delta-binary encoding

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 shown in Fig. 8 and introduced below.

Step 1: Delta encoding The gradient keys are stored in an array. We scan the array from the end to the start and calculate the difference between two adjacent keys. Afterward, 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 `prefix`. For example, the prefix of one byte is 0b00, that of two bytes is 0b01, and so forth. Finally, the delta keys are encoded into binary numbers with byte prefixes.

Note that, there are several existing methods that can be used to compress integers, such as run-length encode (RLE) and Huffman coding. However, RLE and Huffman coding are typically used to compress a data sequence in which a data value might occur consecutively. They need to store every gradient key without compressing and introduce an 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 Δk_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.

5.2 Adaptive prefix for delta keys

Above, we describe delta-binary encoding, which uses binary representation to encode each delta key according to its range. Basically, this method divides the range of delta key evenly and generates several intervals. With the thresholds of these intervals, we decode each delta key with a different number of intervals. In this way, we can use fewer bytes for smaller delta keys and therefore save space. To help the decoding of delta key, we use a `prefix` to indicate the number of intervals taken by each delta key. The prefix uses a fixed strategy that each prefix consumes the same number of bits. However, since the distribution of delta keys is unknown beforehand, can this fixed prefix adapts to the distribution?

Motivation To understand this problem, we conduct experiments to assess the distribution of delta keys with a range of datasets. We find that the distribution of delta keys is not always uniform. In many cases, small delta keys occur more frequently, and large gradient keys occur less. This is unsurprising since relevant features are often closer in the feature vector. They are likely to appear together and yield small delta keys. In order to deal with this nonuniform distribution, we revisit the fixed-length prefix and possible alternatives.

Briefly speaking, our goal is to divide an integer into m intervals, where $m \in \{2, 4, 8, 16\}$ typically and assign an appropriate number of intervals to each delta key. The method of delta-binary encoding divides the value range of an integer evenly, so that each interval contains $b = \frac{32}{m}$ bits. When $m = 8$, for instance, one interval can handle a range of $[0, 15]$, two intervals $[16, 255]$, etc. Specifically, the method proposed in Sect. 5.1 chooses $m = 4$ so that the size of each interval is actually a byte.

After encoding the delta keys with different intervals, the next step is building a prefix array that indicates the number of intervals. The method proposed in Sect. 5.1 uses $m = 4$, and each prefix is two bits. The fixed-length prefix cannot adapt to the nonuniform distribution of delta keys. Intuitively, if the prefix module can adapt to the distribution of delta keys and use smaller space for more frequent items, the space cost can be reduced.

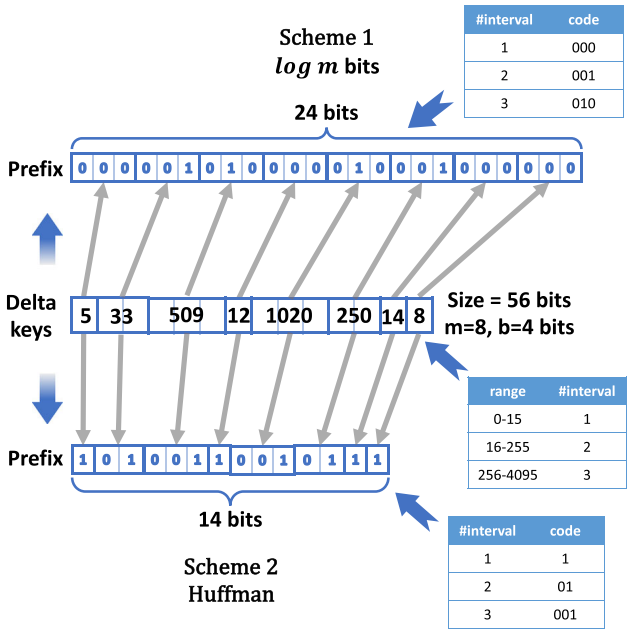


Fig. 9 Two possible schemes for delta key encoding

Two schemes To indicate the number of required intervals for each delta key, we consider two schemes: (i) using $\log m$ bits for each interval; or (ii) using Huffman coding according to the frequency of intervals. Note that the first scheme is actually the fixed-length prefix proposed in Sect. 5.1. A prefix with $\log m$ bits can represent a range of $[0, m-1]$ and therefore is enough to encode m intervals. It is obvious that these two schemes fit different scenarios. If the features of the training data occur randomly, the distribution of delta keys is close to a uniform distribution. In this case, the first scheme works well. Nevertheless, in many cases, the distribution of features is not uniform. In this case, the second scheme is better since more frequent intervals are represented with fewer bits.

We showcase two schemes in Fig. 9. The range of an integer, which is 32 bits, is divided into $m = 8$ intervals so that each interval contains $b = 4$ bits. One interval can encode [0.15], two intervals [16,255], and three intervals [256, 4096]. The delta keys are transformed according to this rule, and the size is 14 intervals, i.e., 56 bits. Once the binary delta keys are set, we next calculate the prefix array. If we choose the first scheme to calculate the prefix, each prefix consumes $\log 8 = 3$ bits, hence the size of the prefix array is 24 bits. If we choose the second scheme, the Huffman code is generated for each interval. Using Huffman coding, the space cost is reduced from 24 bits to 14 bits, yielding an improvement of 42%.

The choice of optimal scheme In our method, the choice of m and the choice of prefix scheme can be decided on-the-fly by scanning the delta keys for only one pass. The procedure is presented as follows.

1. For each delta key Δk_j, we find the index k such that $2^k \le \Delta k_j < 2^{k+1}$. Then, $k+1$ is the minimal number of bits required to represent Δk_j. The occurrences of k are recorded in an array $occur_{k=0}^{31}$.
2. We enumerate all the possible values of $m \in \{2, 4, 8, 16\}$ over $occur$ and calculate the number of intervals needed to encode each delta key.
3. With the number of intervals, we can accurately calculate the number of bits taken by each scheme. From all the candidates, we can easily select the optimal choice, including the encoding scheme and the value of m.

Summary With the adaptive prefix and the deterministic selection mechanism, the optimal strategy is obtained according to the space cost. And the computation complexity is linear to the number of delta keys.

6 Analysis of space cost

Combining the above components, we get a unified framework SKCompress. 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 is 1.27 bytes approximately.
- **Huffman coding and adaptive prefix** We use Huffman coding to compress MinMaxSketch. Similarly, the adaptive prefix proposed in Sect. 5.2 also contains Huffman coding. Since the effect of Huffman coding is affected by the distribution of items, the compression ratio is not deterministic. It is hence impossible to give a certain number beforehand. However, the introduction of Huffman coding and adaptive prefix at least brings no extra space. As we will show in Sect. 7.2, these two approaches make the system 1.2×faster.

Summary To sum up, the total space cost of our method is at most $d \times (\lceil \frac{1}{8} \log_2 \frac{rD}{d} \rceil + \frac{1}{4}) + 8q + s \times t \times \lceil \frac{1}{8} \log_2 q \rceil$.

Compared with the original size $12d$, we can save a lot of communication cost by choosing appropriate hyper-parameters.

7 Experiments

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

7.1 Experiment setting

Implementation We implement a prototype 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 and broadcasts the aggregated gradients to the executors. Specifically, the driver first (1) decodes each encoded gradient from each executor upon receiving, then (2) sums over the decoded gradients, and finally (3) encodes the summed gradient and broadcasts the encoded gradient to all executors. In practice, we use the `aggregate` API in Spark to implement such customized all-reduce operation. This process iterates until convergence. Note that there are other communication patterns, such as parameter-server, all-gather, and reduce-scatter. But the study of communication approaches is orthogonal to our research goal.[1]

Clusters We use two clusters in our experiments. *Cluster-1* is a ten-node cluster in our lab. Each machine is equipped with 32 GB 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 64 GB 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 constraint of 8 GB memory 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 [25], consisting of 19 million instances and 29 million features. The second dataset KDD12 is the next generation of KDD10 [26], 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 dataset of Tencent Inc. CTR is used to predict the click-through-rate of advertisements.

Table 1 The information of evaluated datasets, including the size, the number of instances, and the number of features

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

Machine learning models For statistical models, we choose three popular machine learning models—ℓ_2-regularized logistic regression (LR), support vector machine (SVM), and linear regression (linear). Their loss functions are formalized in Table 2.

We train three ML models 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:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

where β_1 and β_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 SKCompress with two competitors: Adam SGD [27] and ZipML [62]. Adam SGD is the most widely used first-order gradient optimization recently. It combines the advantages of momentum [40,43] and adaptive learning rate [13,36,60]. 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.

We compare SKCompress with five competitors: Adam SGD [27], ZipML [62], DGC [35], QSGD [1], and TernGrad [54].

1. Adam SGD is the most widely used first-order gradient optimization method recently. It combines the advantages of momentum [40,43] and adaptive learning rate [13,36, 44,60].
   Note that the adaptive strategy of Adam is applied to all the baselines for the purpose of fairness, and the Adam we implement communicates with sparse gradients.

[1] Gradient compression can bring larger speedup for all-reduce systems than parameter-server systems, owing to the single bottleneck problem of the driver node. Parameter-sever systems accelerate communication by using more machines and larger bandwidth to aggregate the gradients. Following the setting of previous works on gradient compression, our work tries to compress gradients in all-reduce systems without using more machines.

Table 2 The evaluated ML models, including logistic regression, support vector machine, and linear regression

Machine learning model	Loss function	Gradient
Logistic regression	$f(x, y, \theta) = \sum_{i=1}^{N} \log(1 + \exp(-y_i \theta^{\mathrm{T}} x_i)) + \frac{\lambda}{2}\|\|\theta\|\|_2$	$\frac{\partial f}{\partial \theta} = \sum_{i=1}^{N} -\frac{y_i}{1+\exp(y_i \theta^{\mathrm{T}} x_i)} x_i + \lambda\theta$
Support vector machine	$f(x, y, \theta) = \sum_{i=1}^{N} \max(0, 1 - y_i \theta^{\mathrm{T}} x_i) + \frac{\lambda}{2}\|\|\theta\|\|_2$	$\frac{\partial f}{\partial \theta} = \sum_{i=1}^{N} -y_i x_i \mathbb{I}\{y_i \theta^{\mathrm{T}} x_i < 1\} + \lambda\theta$
Linear regression	$f(x, y, \theta) = \sum_{i=1}^{N} \frac{1}{2}(y_i - \theta^{\mathrm{T}} x_i)^2 + \frac{\lambda}{2}\|\|\theta\|\|_2$	$\frac{\partial f}{\partial \theta} = \sum_{i=1}^{N} -(y_i - \theta^{\mathrm{T}} x_i) x_i + \lambda\theta$

Labels are stored as -1 and 1. $\mathbb{I}\{\cdot\}$ is the indicator function

2. ZipML designs a fixed-point quantification method to compress gradient values to integers. It has shown powerful performance on a range of linear models.
3. DGC drops up to 99.9% gradients to reduce the communication and proposes a series of optimizations to amortize the accuracy loss.
4. QSGD uses the $l2$-norm of the gradient to quantize gradient values to several bits. Besides, a stochastic rounding strategy is proposed to introduce randomness.
5. TernGrad quantizes gradient values to ternary levels—$\{-1,0,1\}$. Similar as QSGD, TernGrad also adopts a stochastic rounding method.

Metrics To measure the performance of SKCompress 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 [61].

Protocol The input dataset is partitioned into two subsets—75% as the training 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 trade-off between convergence robustness and convergence speed, we adopt a popular trick of SGD that uses a batch of instances instead of only one instance [5]. Better, in a distributed environment, mini-batch SGD can decrease the synchronization frequency and save a lot of communication cost. Following the choice of [17], we set the batch size as 10% of the size of the training dataset. As the authors of Adam SGD [27] suggested, we choose 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ. We use a grid search to tune the optimal learning rate η. Specifically, we tune the optimal learning rate with Adam SGD and use this value for all the candidates. The regularization coefficient λ 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 SKCompress. The size of quantile sketch is 128 by default. The size of MinMaxSketch is $2 \times \frac{d}{5}$. We set the sparsity level of DGC as 90%, and choose the compression size of QSGD as one byte. We also compare different choices of compression size for QSGD via a tuning experiment. The detailed results are in "Appendix B."

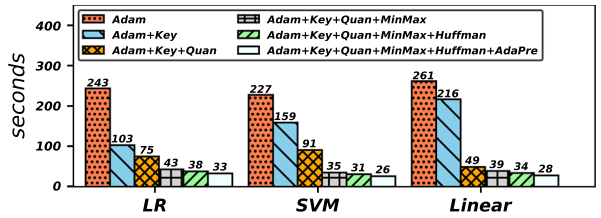
7.2 Efficiency of proposed methods

SKCompress consists of five 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 5 GB 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 Fig. 10.

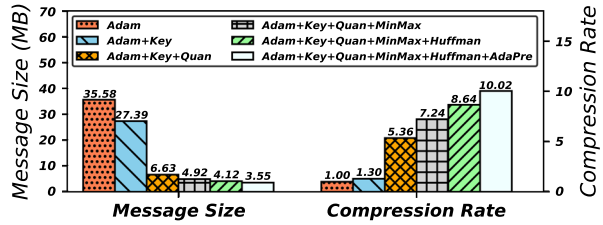
Run time According to the results in Fig. 10a, our proposed methods can significantly accelerate the execution of three different ML algorithms. Compared with Adam, the component of delta-binary encoding alone improves the system performance by up to 2.3×. The addition of quantile-bucket quantification further accelerates the speed by up to 4.4×. The MinMaxSketch alone achieves at most 4.3× improvements. Finally, the Huffman coding and adaptive prefix together make the system 1.2× faster. The results demonstrate that our proposed methods are efficient in reducing the data movement through the network.

Breakdown of run time To better illustrate the effectiveness of compression, we decouple the run time of Adam into computation and communication, as presented in Table 3. For all the three models, the majority of run time, more than 90 percentages, is spent on communication. This is unsurprising because the computation complexity of linear models is not large, while the communication involves several time-consuming stages, including serialization, network buffering, network transmission, and deserialization. By significantly squeezing the communication, our proposed compression method can enhance the overall speed.

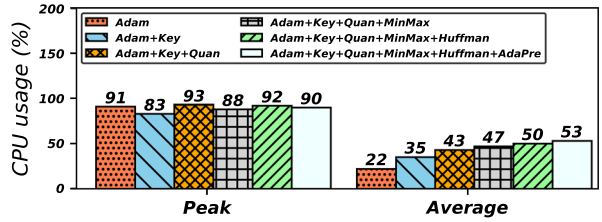
Message size and compression rate The main advantage of compression is reducing the size of messages. Figure 10b presents the average message size and compression rate during the execution. Due to the space constraint, we present the result of LR, and the results of other algorithms are similar. Compared with the uncompressed gradient message, our method decreases the message size from 35.58 to 3.55 MB—a 10× compression rate. As the reader might suspect, sparsification-based methods can obtain a larger compres-



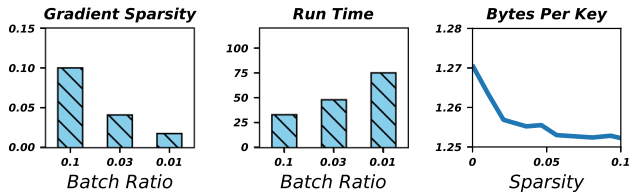
(a) Run time per epoch



(b) Message size and compression rate



(c) CPU overhead



(d) Impact of batch size and sparsity

Fig. 10 Efficiency 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. Huffman refers to the component of Huffman coding. AdaPre refers to the component of adaptive prefix

sion ratio. For instance, DGC reports a 270× improvement on deep neural networks. However, the ultimate goal of gradient compression is the overall convergence rate rather than the compression ratio. If a compression method cannot increase the convergence rate of ML models, the compression effort is in vain. As we will illustrate and discuss in the experiments, sparsification-based methods such as DGC are not suited for linear models although they work well for deep neural networks.

CPU overhead To evaluate the computation overhead brought by compression, we conduct an experiment and present the result in Fig. 10c. Unsurprisingly, our method introduces 31% CPU usage in average. The peak CPU usage is not obviously influenced.

Table 3 Time breakdown of Adam (KDD10, per epoch in seconds)

Model	Run time	Computation	Communication
LR	243	6.3 (2.6%)	236.7 (97.4%)
SVM	227	6.6 (2.9%)	220.4 (97.1%)
Linear	261	4.2 (1.9%)	256.8 (98.1%)

Impact of batch size and sparsity Since our method compresses sparse gradients, it raises a question that 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 Fig. 10d 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 33 to 75 s.

According to the analysis in Sect. 6, 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 Fig. 10d. 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 the original 4 bytes, the delta-binary encoding achieves significant compression performance. This result is consistent with the theoretical analysis in Sect. 6.

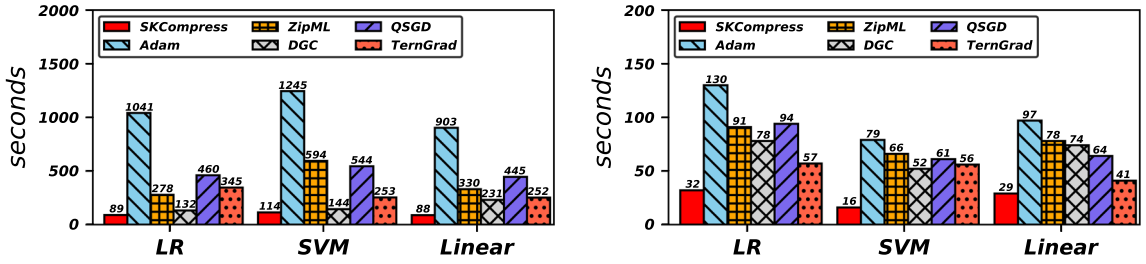
7.3 End-to-end performance

In this section, we compare the end-to-end performance of SKCompress, Adam, ZipML, DGC, QSGD, and TernGrad on Cluster-2. 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 of KDD12 and CTR is presented in Fig. 11. The loss regarding run time is presented in Fig. 12.

7.3.1 Results on KDD12 dataset

For the KDD12 dataset, we use ten executors to run the combinations of three compression methods and three ML algorithms. We assign 5 GB memory for the driver and each executor. Figure 11a shows the average run time per epoch. Figure 12 reports the convergence rate which is measured by the loss function in terms of run time.

Logistic regression As shown in Fig. 11a, SKCompress runs much faster than Adam and three quantization approaches—ZipML, QSGD, and TernGrad. Adam needs to communicate the original gradients without any compression. Therefore,



(a) KDD12 Dataset

(b) CTR Dataset

Fig. 11 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)

Adam is the slowest. ZipML, QSGD, and TernGrad are 3.7×, 2.3×, and 3× faster than Adam by compressing the gradient values. However, ZipML, QSGD, and TernGrad are unable to compress the gradient keys. SKCompress only needs 89 s to process an epoch, bringing 11.7×, 3.1×, 5.2×, 3.8× improvements over four 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 in executors because more executors inevitably yield more communications. DGC performs similarly as SKCompress, because DGC drops 90% of the gradients at the expense of sorting gradient values. Although Cluster-2 is equipped with faster network, the network is more congested than Cluster-1 since Cluster-2 serves many applications simultaneously. Therefore, SKCompress 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 Fig. 12a that ZipML converges much faster than Adam. QSGD achieves similar performance as ZipML while TernGrad cannot converge. The reason is that TernGrad quantizes a gradient to only three levels and recovers the gradient using the maximal absolute value in the gradient. This operation could largely increase most original gradient values, causing large quantization error and unpredictable convergence. Maybe TernGrad is suitable for deep learning models where gradient noise can help jump from the local optimal, it is not suitable for convex linear models. In contrast, QSGD is much better by using more quantization levels. DGC converges the fastest at the beginning but converges very slow afterward. As we will analyze in Sect. 7.3.2, the datasets trained by linear models are often skewed. DGC might drop the gradients of some features throughout the training process, even though these features can provide useful information. Overall, SKCompress beats the competitors significantly and achieves the best trade-off between efficiency and convergence rate.

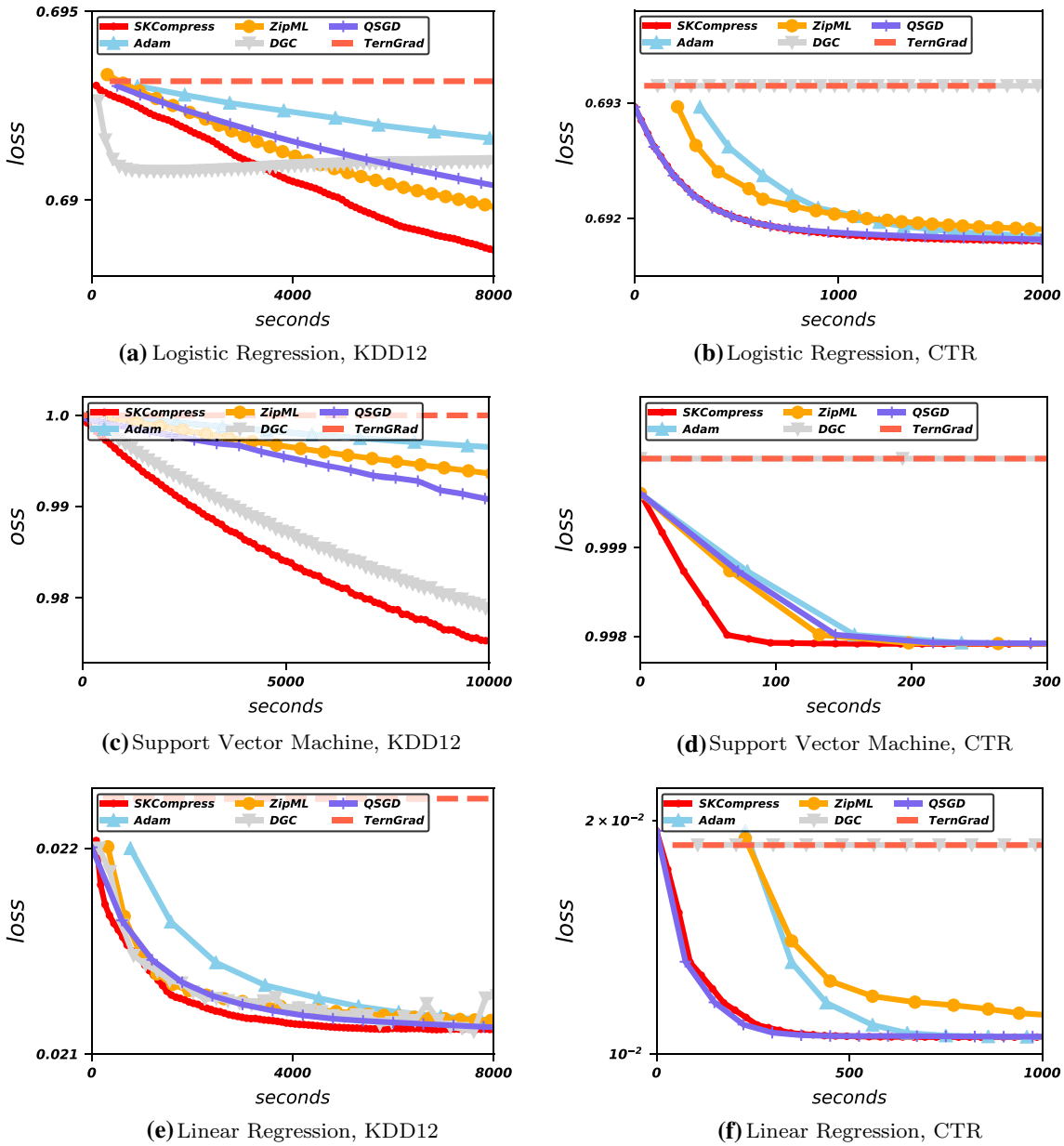
Support vector machine The result of support vector machine is similar to that of logistic regression. Adam is the slowest, followed by three quantization methods. SKCompress and DGC only take 114 and 144 s—10.9× and 8.6× faster than Adam. Meanwhile, as can be seen in Fig. 12c, the convergence rate of SKCompress is significantly faster than the baselines. We can find an interesting phenomenon in Fig. 12c, that is, SKCompress 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 since more gradients are quantified to zero.

Linear regression For linear regression, Adam takes 903 seconds per epoch, while SKCompress only needs 88 seconds. As Fig. 12e shows, ZipML, DGC, QSGD, and SKCompress are significantly faster than Adam. The convergence of DGC is not stable, especially when it approaches the optimum, since the sparsification strategy makes sparse gradients even sparser and drops many useful values. TernGrad cannot converge owing to its aggressive quantization approach.

7.3.2 Results on CTR dataset

For the larger dataset CTR, we use 50 executors on Cluster-2. We assign 8 GB memory for the driver and each executor due to the memory limitation. The run time statistics and convergence curves are presented in Figs. 11b and 12.

Logistic regression On this larger dataset, Adam still runs the slowest. The speed of ZipML, DGC, and QSGD is similar. SKCompress is 4.1× and 2.8× faster than Adam and ZipML. Note that the speedup of SKCompress on CTR is smaller



(a) Logistic Regression, KDD12 **(b)** Logistic Regression, CTR

(c) Support Vector Machine, KDD12 **(d)** Support Vector Machine, CTR

(e) Linear Regression, KDD12 **(f)** Linear Regression, CTR

Fig. 12 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)

than that on KDD12 since KDD12 is sparser than CTR. As each instance of CTR generates more nonzero gradient pairs, the gradient density is higher, resulting in higher computation cost. As a consequence, the performance improvement brought by the reduction of communication cost is not as large as that on KDD12 dataset. The performance of DGC is much worse on this dataset than on KDD12 owing to the same reason. Since the gradient of CTR has more nonzero gradient pairs, the extra computation cost brought by sorting operation amortizes the benefit of communication reduction. 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. TernGrad cannot converge on this dataset owing to the same reason described in Sect. 7.3.1. In contrast, the convergence rate of SKCompress and QSGD is much better. DGC cannot converge on this dataset as well. According to our observation, this is caused by the data skew of CTR dataset in which some features occur frequently while some others occur rarely. The less frequent features often contain more discriminating information, but DGC probably drops these features with a batch training setting. Therefore, the convergence rate of DGC is very slow. This phenomenon shows that DGC cannot work well on all

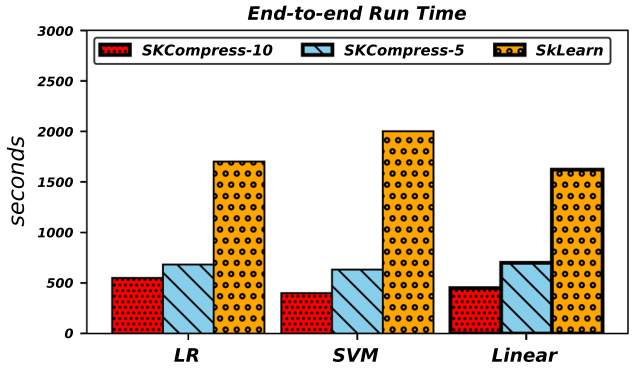


Fig. 13 Comparison with a single node system. (KDD10, LR)

datasets, while SKCompress reveals the ability of generalization across different datasets.

Support vector machine According to the results in Fig. 11b, SKCompress brings 4.9× and 4.1× improvements than Adam and ZipML. Other three baselines run in similar speed. Compared with logistic regression and linear regression, support vector machine is easier to get converged on this dataset. Therefore, as Fig. 12d shows, the convergence rates of ZipML and QSGD are faster than Adam due to faster communication. SKCompress outperforms them significantly and is able to converge to a tolerance in a shorter time. Similar to the above results, DGC and TernGrad cannot converge.

Linear regression As illustrated in Fig. 11b, SKCompress takes 29 s to train an epoch, while Adam and ZipML need 97 and 78 s. Figure 12f shows the convergence rates. ZipML is slower than Adam, caused by the defect of uniform quantification. Overall, SKCompress and QSGD perform the best and are able to converge to the same loss as Adam.

Convergence of QSGD The convergence of QSGD can be similar to SKCompress on LR and linear regression (e.g., Fig. 12b, f), while significantly worse on SVM (e.g. Fig. 12d). This is caused by the property of the algorithm. As shown in Table 2, the gradient of each wrongly classified instance in SVM is $y_i x_i$, even if the actual difference between the prediction and the label is small. As a result, the gradient of SVM has a large value range. With the uniform quantization strategy adopted in QSGD, such a large value range inevitably leads to high quantization errors and therefore slows down the convergence. In contrast, for LR and linear regression, the gradient shrinks to zero when the prediction is close to label. The value range of gradients is smaller than SVM, especially when the model approaches the optimum. Therefore, QSGD performs better on LR and linear regression. The results verify that QSGD's uniform quantization cannot generalize to diverse scenarios.

7.3.3 Comparison with a single node system

To give a reference point of performance, we compare SKCompress 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 SKCompress is executed on five and ten machines, denoted by SKCompress-5 and SKCompress-10. The other settings are the same as Sect. 7.2. Figure 13 shows the run time of twenty epochs. SKCompress-5 is 2.5×, 3.1×, and 2.3× faster than SkLearn in training three algorithms. SKCompress-10 further brings 1.2×, 1.6×, and 1.6× speedup compared with SKCompress-5.

Although the distribution of SKCompress 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 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.

7.4 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 Sect. 7.3. An algorithm is considered as converged if the variation of loss is less than 1% within five epochs.

As illustrated in Table 4, Adam, ZipML, QSGD, and SKCompress can converge to almost the same model quality. However, SKCompress converges much faster than the other methods. According to our analysis, MinMaxSketch causes underestimated gradients, while it does 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 Sect. 4.2 solve this problem by giving larger learning rate for a slow dimension and reducing quantification error. TernGrad cannot converge to the optimal within one day since it does not fit sparse linear models.

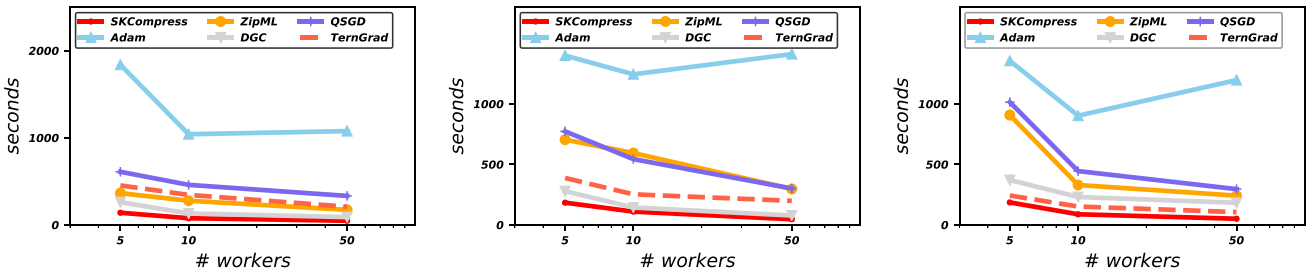
7.5 Scalability evaluation

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 Fig. 14. Due to the space constraint, we only provide

Table 4 Model accuracy (KDD12)

	SketchML	Adam	ZipML	DGC	QSGD	TernGrad
LR	0.6885/6.7 h	0.6885/23 h	0.6887/11 h	0.6910/4 h	0.6885/8.1 h	0.69314/24 h
SVM	0.9784/4.1 h	0.9785/23 h	0.9788/10 h	0.9786/4 h	0.9784/9 h	1.0001/24 h
Linear	0.2111/3.8 h	0.2109/22 h	0.2111/9.4 h	0.2112/6.5 h	0.2110/5.8 h	0.2224/24 h

The metric is minimal loss against converged time in hours, separated by symbol "/"



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

Fig. 14 Scalability evaluation. The evaluated metric is the average run time per epoch in terms of the number of workers (executors). Run time is in seconds

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 Fig. 14a, the performance of all the methods increases when we increase the number of workers from five to ten, i.e., SKCompress becomes 1.8× faster, Adam 1.8×, ZipML 1.3×, DGC 1.9×, QSGD 1.3×, and TernGrad 1.3×.

Afterward, we use fifty workers and find that Adam suffers a performance deterioration. The reason is that the increase in communication cost overwhelms the benefit of computation cost. SKCompress, to the contrary, achieves 1.6× improvements. Ideally, the performance speedup is five when we use fifty workers instead of ten. To understand this concern, we show a breakdown of per-batch run time in Table 5. We decouple the total run time into four parts—gradient computation, SKCompress encoding, SKCompress decoding, and communication.

- Gradient computation. The system needs 0.9 s to compute a gradient with fifty workers, 4× faster than that with ten workers.
- SKCompress encoding. With more workers, each worker processes fewer instances. Therefore, the gradient before aggregation is sparser, and SKCompress takes less time to encode the gradient. The improvement of 2.2× is reasonable since the gradient sparsity increases about 2× according to our observation.[2]
- SKCompress decoding. Different from the encoding phase, the decoding phase includes merging multiple sketches and queries gradient values from the merged sketch. Therefore, using fifty workers increases the time cost by 1.26×.
- Communication. When using more workers, each worker processes fewer instances in a batch, resulting in a higher gradient sparsity. In other words, the higher the number of workers, the lower d. Since the size of MinMaxSketch is $2 \times \frac{d}{5}$, SKCompress outputs a smaller encoded gradient. The time cost for the communication of encoded gradients decreases from 3.7 to 1.8 s when we increase the number of workers from 10 to 50.

The above results explain why SKCompress cannot achieve linear speedup with fifty workers. The time cost of gradient computation decreases almost linearly. However, the speedups of some operations, such as encoding and communication, are sublinear. Worse, the decoding of SKCompress takes more time with more workers. Unsurprisingly, the overall speedup is sublinear. We consider this as a potential limitation of SKCompress and treat it as future work to improve the scalability.

Support vector machine For support vector machine, the results are similar as logistic regression. All the 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. Other methods still benefit from the increase in workers. The performance of SKCompress is improved up to 2.3×.

[2] The sparsity of the aggregated gradient remains unchanged since the total batch size of all workers is the same.

Table 5 Time breakdown for scalability evaluation (KDD12, LR, per batch in seconds)

# workers	Run time	Gradient computation	SKCompress encoding	SKCompress decoding	Communication
10	8.9	3.6	0.34	1.5	3.5
50	4.8	0.9	0.15	1.9	1.8

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

7.6 Sensitivity evaluation

SKCompress 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 Sect. 7.3.1.

Size of quantile sketch As shown in Fig. 15, a larger size accelerates the training because the quantization error is reduced. According to Table 6, 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.

7.7 Evaluation on neural net model

The above-evaluated algorithms belong to generalized linear models. However, our Sketch mechanism can also 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 conduct a simulation experiment to train a ResNet18 model on Cifar10 dataset [9], which consists of 50,000 training images and 10,000 testing images. Since it is much more computation-intensive to train a deep learning model, the experiment is performed by PyTorch on a server with 8 Titan RTX GPUs, rather than Spark on CPU-cluster as in our previous experiments. Nevertheless, it is non-trivial to adapt SKCompress to GPUs. First, the gradients in ResNet are generally dense, making the compression of gradient keys meaningless. Second, the encoding of MinMaxSketch requires the gradient values to be inserted sequentially, which is inefficient for GPUs. As a result, we (i) remove the compression of gradient keys in SKCompress, (ii) implement quantile-bucket quantification on GPU via `Trust` library, and (iii) implement MinMaxSketch and Huffman coding on CPU. By doing so, there is an extra IO overhead since we have to copy the gradients from GPU to CPU for MinMaxSketch and Huffman coding.

We choose Adam as our optimizer, and set the batch size to be 512, learning rate to be 0.003, and weight decay (L2 regularization) to be 1e-4. For hyper-parameters in SKCompress, we set the number of quantile buckets to be 16 and the size of MinMaxSketch to be $2 \times \frac{d}{5}$, where d is the size of each gradient tensor. Figure 16 shows the convergence with and without SKCompress. We discuss the convergence performance and compression performance respectively:

- SKCompress has a good short-term convergence; however, it suffers from the error caused by gradient compression later. Consequently, there is a 1.7% drop of accuracy in SKCompress (8.8% vs. 7.1%).
- SKCompress achieves around 21.7× compression; therefore, the network transmission can be greatly reduced. However, the time cost of compression and decompression is about 5× larger than that of communication in Adam when the GPUs are located in the same server and connected with high-speed NVLink lanes.

Limitation To summarize, SKCompress causes a small accuracy drop but is able to convey a high compression rate. If the communication of training neural networks is the bottleneck rather than the computation, SKCompress can still bring performance improvement. Nevertheless, SKCompress is not suitable for deep learning tasks for two reasons: (i) the gradients in deep learning are usually dense while SKCompress targets at sparse gradients; (ii) the encoding of SKCompress is hard to be parallelized on GPUs, and hence

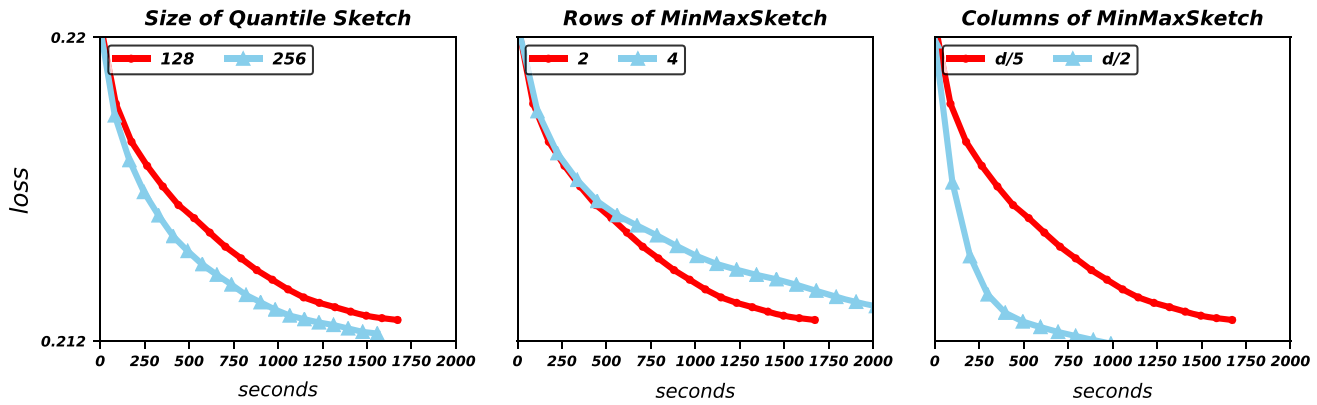
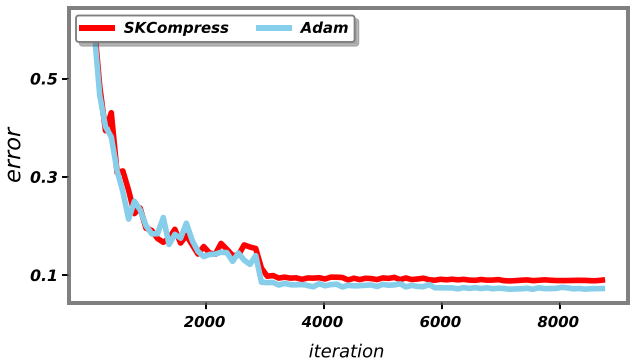


Fig. 15 Sensitivity evaluation (convergence rate, KDD12, linear). We change the size of quantile sketch, the number of MinMaxSketch rows, and the number of MinMaxSketch columns

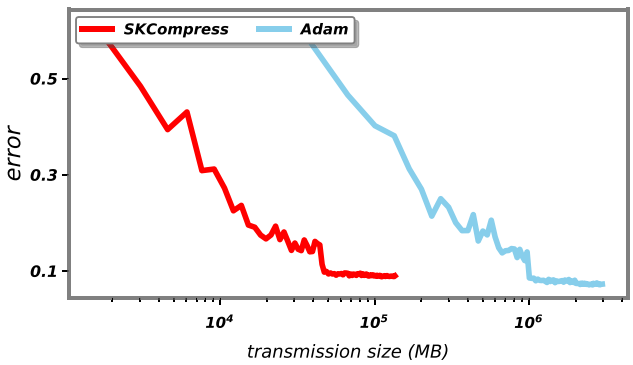
Table 6 Sensitivity evaluation (seconds per epoch, KDD12, linear)

	Default	Quantile size = 256	MinMaxSketch row = 4	MinMaxSketch column= $\frac{d}{2}$
Run time	88	82	112	99

We change the size of quantile sketch, the number of MinMaxSketch rows, and the number of MinMaxSketch columns



(a) testing error in terms of iteration



(b) testing error in terms of transmission size

Fig. 16 Evaluation on ResNet18 model and Cifar10 dataset

it might slow down the compression phase and deteriorate the overall performance in practice.

Table 7 Assessment of different precisions (KDD12, LR)

Method	Run time per epoch	Loss in 2 h
SKCompress	89	0.6909
ZipML-8 bit	231	0.6932
ZipML-16 bit	278	0.6919
Adam-float	725	0.6911
Adam-double	1041	0.6914

7.8 Assessment of different precisions

In Sect. 7.3, we use 16 bits for ZipML and double type for Adam. Here, we evaluate more precision choices and show the result in Table 7. The experimental settings are the same as Sect. 7.3.1.

ZipML runs 1.2× faster with 8 bits than 16 bits. However, as stated in Sect. 7.1, 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. SKCompress achieves the fastest speed—2.6× and 8.1× faster than ZipML and Adam. Within the same time, i.e., 2 h, SKCompress can converge to the smallest loss compared with other four competitors, verifying the fast convergence of SKCompress.

7.9 Summary

In summary, SKCompress outperforms the baselines on a range of ML algorithms and datasets. SKCompress consumes

remarkably less time to execute a training epoch. Although it needs more epochs to get converged, the overall performance still surpasses the other competitors. Besides, SKCompress reveals a good ability of scalability. We also evaluate the sensitivity of SKCompress against related hyper-parameters, the performance of SKCompress on a neural network model, and the effect of different precisions.

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 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.

8 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 [5]. 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 previous 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 [16]. 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 a threshold to filter large gradients. This category is called the sparsification compression method, including 1bit-SGD [47], DGC [35], and Sparsified-SGD [48]. 1bit-SGD sends the gradient values larger than the threshold, and DGC sends the top-0.1% gradient values according to the absolute value. However, all of them have drawbacks. 1-bit SGD locally accumulates the small gradients until the accumulation reaches the threshold, but it induces a staleness problem that some stale gradients might lead to unstable model convergence. DGC completely abandons the small gradients, hence the ML model might converge slower for skewed datasets. Sparsified-SGD combines 1-bit SGD and DGC that sends top-k gradients and accumulates small ones in memory. Another type of lossy methods uses the quantification technique, such as ZipML, QSGD, TernGrad, DCD-PSGD, and ECQ-SGD [1,50,54,55,62]. They use a quantification strategy to transform a floating-point number to a small integer according to the value range of original data. Then, they transfer the encoded integers with fewer bits. ZipML linearly converts the original gradient values to integer numbers. QSGD and TernGrad further introduce a stochastic rounding strategy to assure unbiased gradients. DCD-PSGD introduces communication compression into the *decentralized* scenario. In decentralized training, each worker only accepts the model updates from its neighbors rather than all other workers. As stated in their paper, the proposed method DCD-PSGD is suitable for slow network conditions, which is a special case for distributed machine learning. Therefore, it is significantly different from the *centralized* scenario to which our work belongs. In fact, DCD-PSGD uses the QSGD-style method to quantize the gradients, and we have compared QSGD in the centralized scenario. ECQ-SGD combines error compensation with QSGD and proves its convergence. However, since ECQ-SGD accumulates the quantization errors every time, it eventually incurs dense compensated gradients. Therefore, it is not suitable for the sparse scenario studied in our work. Although these methods can achieve a trade-off between efficient compression and correct convergence, they 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. The 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 are 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. There is also a line of research that compresses the gradients by matrix decomposition. For instance, ATOMO [52] decomposes the gradients with singular value decomposition (SVD) and drops small singular values to achieve sparsification. However, since the gradients of linear models are sparse vectors (one-dimensional tensors), applying SVD on these gradients is infeasible. Thus, ATOMO is unfit for our targeted scenario.

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 [8,14,63]. The frequency sketch is designed to estimate the occurring frequency of items [10]. 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.

There are also some works that study distributed machine learning over large models. Rendle et al. [45] propose a distributed scalable coordinate descent algorithm. *Coordinate descent* iteratively (1) selects one or several coordinates of the model, (2) scans the dataset in a *column-wise* manner and computes necessary statistics, (3) aggregates statistics from all workers, and finally (4) updates the chosen coordinates. Rendle et al. accelerate the training by carefully partitioning the dataset into blocks and utilizing techniques such as load balancing and caching. Parnell et al. [41,42] develop TPA-SCD, an *asynchronous* variant of stochastic coordinate descent on GPUs. They use the power of GPU to accelerate computation and handle large-scale data by optimizing a subset of dimensions alternatively. Nevertheless, these works are orthogonal to ours. First, none of them propose to reduce the communication cost by compression when they collect distributed statistics for model updates. Second, SKCompress focuses on the family of stochastic gradient descent (SGD), while the mentioned works are based on coordinate descent (CD) methods. The comparison of SGD and CD is orthogonal to our goal, and we therefore only consider SGD baselines in this work.

9 Conclusion

In this paper, in order to accelerate distributed machine learning, we proposed a sketch-based method, namely SKCompress, 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 the proposed methods. Empirical results on a range of large-scale datasets and machine learning algorithms demonstrated that SKCompress can be up to 10× faster than the state-of-the-art methods.

Acknowledgements This work is supported by NSFC (No. 61832001, 61702016, 61702015, 61572039, U1936104), the National Key Research and Development Program of China (No. 2018YFB1004403), Beijing Academy of Artificial Intelligence (BAAI), PKU-Tencent joint research Lab, and the project PCL Future "Regional Network Facilities for Large-scale Experiments and Applications under Grant PCL2018KP001".

A Mathematical analysis of SKCompress

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

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 [39]. To be precise, we refer to Theorem 1.

Theorem 1 (Theorem 6.3 of [7]) *Let f be convex and θ^* the optimal point. Choosing step length appropriately, the convergence rate of SGD is*

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

where σ 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 [33], weight sampling [37], and SVRG [24].

We refer $\tilde{g}=\{\tilde{g}_i\}_{i=1}^{d}$ to the quantificated gradient. Here, we abuse the notation that in Theorem 1 the subscript of g_t indicates the tth epoch to which it belongs, while in the following analysis that of g_i indicates the ith 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 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 ϕ_{min} and ϕ_{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,

$$\begin{aligned}(\tilde{g}_i - g_i)^2 &= \left(\frac{1}{2}(v_{b(i)} + v_{b(i+1)}) - g_i\right)^2 \\ &\leqslant \frac{1}{4}(v_{b(i+1)} - v_{b(i)})^2,\end{aligned}$$

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

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

□

Corollary 1 *When the distribution of gradients is not biased, i.e., there exists $\delta > 1$ such that $\frac{||v||^2}{v_1^2 + v_{q+1}^2} \geqslant \delta$, Eq. (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 $\min(\frac{d}{q^2}, \frac{\sqrt{d}}{q})||g||^2$ [1]. 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 α 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 α 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, α 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 ith 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 jth counter in the ith 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 functions 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 ϵ and δ 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 3 *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 $(n + 1)$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:

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

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

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

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

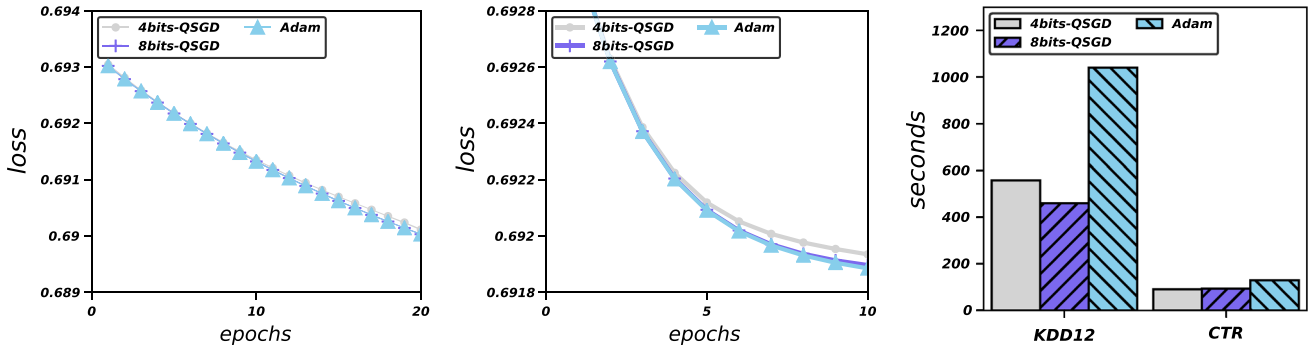
$$P_{\text{CR}}\{e_l\} = 1 - \left(1 - \left(1 - \frac{1}{w}\right)^{v-l}\right)^{d}.$$

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:

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

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.



(a) Convergence (LR, KDD12 dataset) **(b)** Convergence (LR, CTR dataset) **(c)** Run time per epoch

Fig. 17 Assessment of compression size for QSGD

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 Tuning QSGD

To choose an appropriate compression size for QSGD, we conduct an experiment to compare different choices. Taking LR as a representative, we compare 4bits-QSGD, 8bits-QSGD, and Adam on KDD12 and CTR datasets. (2bits-QSGD is equal to TernGrad so we do not consider it) As shown in Fig. 17a, b, 8bits-QSGD achieves almost the same convergence rate (loss in terms of epoch) as Adam. Intuitively, the number of quantization buckets of QSGD is 256 when using 8 bits, which is able to provide enough precision for desirable convergence. When using 4 bits for QSGD, however, the convergence rate is slower than 8bits-QSGD and Adam. This is reasonable because the number of quantization buckets is only 16 for 4bits-QSGD, which inevitably incurs higher quantization error and harms the convergence.

We then assess the run time of QSGD with different numbers of bits. Both QSGD variants run faster than Adam, as shown in Fig. 17c, due to the reduction in communication cost. However, the run time does not significantly decrease along with the number of bits as expected. This is not a surprising phenomenon for two reasons: (i) since QSGD only compresses gradient values and stores gradient keys in 4-byte integers, the communication overhead only decreases from $5d$ bytes to $4.5d$ bytes, where d is the number of nonzero values in gradient; (ii) if the compression size is less than one byte, there is an extra overhead of bit manipulation during encoding and decoding, while we can use the primitive `byte` type to store the compressed value in 8bits-QSGD.

Owing to the experimental results, we determine to choose the compression size of QSGD as 8 bits in our end-to-end comparison in Sect. 7.

C Effectiveness of adaptive learning rate

As introduced in Sect. 4.2, we introduce to solve the problem of vanishing gradient via an adaptive learning rate method. To choose an appropriate technique for adaptive learning rate, we compare Adam and AMSGrad by training LR on KDD10 dataset, which is described in Table 1. As shown in Fig. 18, the convergence rates of Adam and AMSGrad are almost the same without compression. AMSGrad achieves lower testing loss eventually but the performance gap is small, which is consistent with the results on LR in [44]. After applying SKCompress, their convergence rates are still matching. Although the convergence rate with SKCompress is slower than the counterpart in the first few epochs due to the property of under-estimation of MinMaxSketch, it eventually converges to a comparable testing loss with the help of adaptive learning rate.

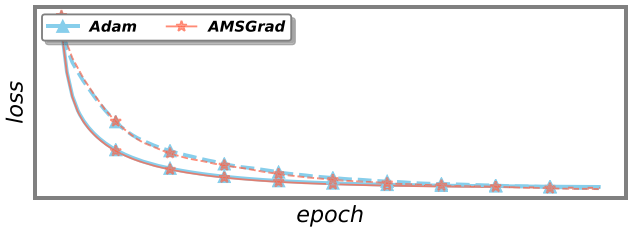


Fig. 18 The comparison of Adam and AMSGrad. The evaluated metric is the testing loss in terms of epochs. We plot convergence without SKCompress in solid lines and plot convergence with SKCompress in dashed lines

Since Adam and AMSGrad have similar convergence performance, we choose Adam as our optimizer since it achieves the state-of-the-art performance and is one of the most widely-used adaptive methods.

References

1. Alistarh, D., Li, J., Tomioka, R., Vojnovic, M.: Qsgd: randomized quantization for communication-optimal stochastic gradient descent. arXiv preprint. arXiv:1610.02132 (2016)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
3. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on cuda. Tech. rep., Nvidia Corporation (2008)
4. Bonner, S., Kureshi, I., Brennan, J., Theodoropoulos, G., McGough, A.S., Obara, B.: Exploring the semantic content of unsupervised graph embeddings: an empirical study. Data Sci. Eng. **4**(3), 269–289 (2019)
5. Bottou, L.: Large-scale machine learning with stochastic gradient descent. In: Proceedings of COMPSTAT'2010, pp. 177–186 (2010)
6. Bottou, L.: Stochastic gradient descent tricks. In: Neural Networks: Tricks of the Trade, pp. 421–436 (2012)
7. Bubeck, S., et al.: Convex optimization: algorithms and complexity. Found. Trends® Mach. Learn. **8**(3–4), 231–357 (2015)
8. Chen, T., Guestrin, C.: Xgboost: a scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794 (2016)
9. Cifar: Cifar dataset. https://www.cs.toronto.edu/~kriz/cifar.html
10. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005)
11. Dean, J., Corrado, G., Monga, R., et al.: Large scale distributed deep networks. In: Advances in Neural Information Processing Systems, pp. 1223–1231 (2012)
12. Deutsch, P.: Deflate compressed data format specification version 1.3. Tech. rep. (1996)
13. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12**(Jul), 2121–2159 (2011)
14. Greenwald, M., Khanna, S.: Space-efficient online computation of quantile summaries. ACM SIGMOD Record **30**, 58–66 (2001)
15. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of things (iot): a vision, architectural elements, and future directions. Future Gener. Comput. Syst. **29**(7), 1645–1660 (2013)
16. Hinds, S.C., Fisher, J.L., D'Amato, D.P.: A document skew detection method using run-length encoding and the hough transform. In: Pattern Recognition, 1990. Proceedings., 10th International Conference on, Vol. 1, pp. 464–468. IEEE (1990)
17. Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J.K., Gibbons, P.B., Gibson, G.A., Ganger, G., Xing, E.P.: More effective distributed ml via a stale synchronous parallel parameter server. In: Advances in Neural Information Processing Systems, pp. 1223–1231 (2013)
18. Hosmer Jr., D.W., Lemeshow, S., Sturdivant, R.X.: Applied Logistic Regression, vol. 398. Wiley, New York (2013)
19. Huang, Y., Cui, B., Zhang, W., Jiang, J., Xu, Y.: Tencentrec: real-time stream recommendation in practice. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 227–238. ACM (2015)
20. Jiang, J., Cui, B., Zhang, C., Yu, L.: Heterogeneity-aware distributed parameter servers. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 463–478. ACM (2017)
21. Jiang, J., Huang, M., Jiang, J., Cui, B.: 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, pp. 313–318. Springer (2017)
22. Jiang, J., Jiang, J., Cui, B., Zhang, C.: Tencentboost: a gradient boosting tree system with parameter server. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 281–284 (2017)
23. Jiang, J., Tong, Y., Lu, H., Cui, B., Lei, K., Yu, L.: Gvos: a general system for near-duplicate video-related applications on storm. ACM Trans. Inf. Syst. (TOIS) **36**(1), 3 (2017)
24. Johnson, R., Zhang, T.: Accelerating stochastic gradient descent using predictive variance reduction. In: Advances in Neural Information Processing Systems, pp. 315–323 (2013)
25. KDD: Kdd cup 2010 (2010). http://www.kdd.org/kdd-cup/
26. KDD: Kdd cup 2012 (2012). https://www.kaggle.com/c/kddcup2012-track1
27. Kingma, D., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint. arXiv:1412.6980 (2014)
28. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint. arXiv:1609.02907 (2016)
29. Knuth, D.E.: Dynamic Huffman coding. J. Algorithms **6**(2), 163–180 (1985)
30. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances In Neural Information Processing Systems, pp. 1097–1105 (2012)
31. Li, B., Drozd, A., Guo, Y., Liu, T., Matsuoka, S., Du, X.: Scaling word2vec on big corpus. Data Sci. Eng. 1–19 (2019)
32. Li, M., Liu, Z., Smola, A.J., Wang, Y.X.: Difacto: distributed factorization machines. In: Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, pp. 377–386. ACM (2016)
33. Li, M., Zhang, T., Chen, Y., Smola, A.J.: Efficient mini-batch training for stochastic optimization. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 661–670. ACM (2014)
34. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. arXiv preprint. arXiv:1509.02971 (2015)
35. Lin, Y., Han, S., Mao, H., Wang, Y., Dally, W.J.: Deep gradient compression: reducing the communication bandwidth for distributed training. arXiv preprint. arXiv:1712.01887 (2017)
36. McMahan, B., Streeter, M.: Delay-tolerant algorithms for asynchronous distributed online learning. In: Advances in Neural Information Processing Systems, pp. 2915–2923 (2014)
37. Needell, D., Ward, R., Srebro, N.: Stochastic gradient descent, weighted sampling, and the randomized kaczmarz algorithm. In: Advances in Neural Information Processing Systems, pp. 1017–1025 (2014)

38. Neelakantan, A., Vilnis, L., Le, Q.V., Sutskever, I., Kaiser, L., Kurach, K., Martens, J.: Adding gradient noise improves learning for very deep networks. arXiv preprint. arXiv:1511.06807 (2015)
39. Nemirovski, A., Juditsky, A., Lan, G., Shapiro, A.: Robust stochastic approximation approach to stochastic programming. SIAM J. Optim. **19**(4), 1574–1609 (2009)
40. Nesterov, Y.: A method for unconstrained convex minimization problem with the rate of convergence o (1/k^ 2). Doklady AN USSR **269**, 543–547 (1983)
41. Parnell, T., Dünner, C., Atasu, K., Sifalakis, M., Pozidis, H.: Large-scale stochastic learning using gpus. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 419–428 (2017)
42. Parnell, T., Dünner, C., Atasu, K., Sifalakis, M., Pozidis, H.: Tera-scale coordinate descent on gpus. Future Gener. Comput. Syst. (2018)
43. Qian, N.: On the momentum term in gradient descent learning algorithms. Neural Netw. **12**(1), 145–151 (1999)
44. Reddi, S.J., Kale, S., Kumar, S.: On the convergence of Adam and beyond. arXiv preprint. arXiv:1904.09237 (2019)
45. Rendle, S., Fetterly, D., Shekita, E.J., Su, B.y.: Robust large-scale machine learning in the cloud. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1125–1134 (2016)
46. Seber, G.A., Lee, A.J.: Linear Regression Analysis, vol. 936. Wiley, Hoboken (2012)
47. Seide, F., Fu, H., Droppo, J., Li, G., Yu, D.: 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In: INTERSPEECH, pp. 1058–1062 (2014)
48. Stich, S.U., Cordonnier, J.B., Jaggi, M.: Sparsified sgd with memory. In: Advances in Neural Information Processing Systems, pp. 4447–4458 (2018)
49. Suykens, J.A., Vandewalle, J.: Least squares support vector machine classifiers. Neural Process. Lett. **9**(3), 293–300 (1999)
50. Tang, H., Gan, S., Zhang, C., Zhang, T., Liu, J.: Communication compression for decentralized training. In: Advances in Neural Information Processing Systems, pp. 7652–7662 (2018)
51. Tewarson, R.P.: Sparse Matrices. Academic Press, New York (1973)
52. Wang, H., Sievert, S., Liu, S., Charles, Z., Papailiopoulos, D., Wright, S.: Atomo: communication-efficient learning via atomic sparsification. In: Advances in Neural Information Processing Systems, pp. 9850–9861 (2018)
53. Wang, Y., Lin, P., Hong, Y.: Distributed regression estimation with incomplete data in multi-agent networks. Sci. China Inf. Sci. **61**(9), 092202 (2018)
54. Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., Li, H.: Terngrad: ternary gradients to reduce communication in distributed deep learning. In: Advances in Neural Information Processing Systems, pp. 1509–1519 (2017)
55. Wu, J., Huang, W., Huang, J., Zhang, T.: Error compensated quantized sgd and its applications to large-scale distributed optimization. arXiv preprint. arXiv:1806.08054 (2018)
56. Yahoo: Data sketches (2004). https://datasketches.github.io/
57. Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., Uhlig, S.: Elastic sketch: adaptive and fast network-wide measurements. In: ACM SIGCOMM, pp. 561–575 (2018)
58. Yang, T., Liu, A.X., Shahzad, M., Zhong, Y., Fu, Q., Li, Z., Xie, G., Li, X.: A shifting bloom filter framework for set queries. Proc. VLDB Endow. **9**(5), 408–419 (2016)
59. Yu, L., Zhang, C., Shao, Y., Cui, B.: Lda*: a robust and large-scale topic modeling system. Proc. VLDB Endow. **10**(11), 1406–1417 (2017)
60. Zeiler, M.D.: Adadelta: an adaptive learning rate method. arXiv preprint. arXiv:1212.5701 (2012)
61. Zhang, C., Ré, C.: Dimmwitted: a study of main-memory statistical analytics. Proc. VLDB Endow. **7**(12), 1283–1294 (2014)
62. Zhang, H., Kara, K., Li, J., Alistarh, D., Liu, J., Zhang, C.: Zipml: an end-to-end bitwise framework for dense generalized linear models. arXiv:1611.05402 (2016)
63. Zhang, Q., Wang, W.: A fast algorithm for approximate quantiles in high speed data streams. In: Scientific and Statistical Database Management, 2007. 19th International Conference on SSBDM'07, pp. 29–29. IEEE (2007)
64. Zheng, T., Chen, G., Wang, X., Chen, C., Wang, X., Luo, S.: Real-time intelligent big data processing: technology, platform, and applications. Sci. China Inf. Sci. **62**(8), 82101 (2019)
65. Zheng, Y., Zhang, L., Xie, X., Ma, W.Y.: Mining interesting locations and travel sequences from gps trajectories. In: Proceedings of the 18th International Conference on World Wide Web, pp. 791–800. ACM (2009)
66. Zinkevich, M., Weimer, M., Li, L., Smola, A.J.: Parallelized stochastic gradient descent. In: Advances in Neural Information Processing Systems, pp. 2595–2603 (2010)
67. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)