Quantized Training of Gradient Boosted Decision Trees

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recent years have witnessed significant success in Gradient Boosted Decision Trees (GBDT) for a wide range of machine learning applications. Generally, a consensus is agreement among GBDT's training algorithms that gradients and statistics are computed based on high-precision floating point. In this paper, we investigate an essentially important question but has been largely ignored by the previous literature - *how many bits are in need for representing gradients in training GBDT?* To solve this mystery, we propose to quantize all the high-precision gradients in a very simple yet effective way in the GBDT's training algorithm. Surprisingly, both our theoretical analysis and empirical studies show that the necessary precisions of gradients without hurting any performance can be quite low, e.g., 2 or 3 bits. With low-precision gradients, most arithmetic operations in GBDT training can be replaced by integer operations of 8, 16, or 32 bits. Promisingly, these findings may pave the way for much more efficient training of GBDT from several aspects: (1) speeding up the computation of gradients and histograms; (2) compressing the communication cost of high-precision statistical information during distributed training; (3) the inspiration of utilization and development of hardware architectures which well support low-precision computation. Benchmarked on CPU, GPU, and distributed clusters, we observe up to $2\times$ speedup of our simple quantization strategy comparing with SOTA GBDT systems on extensive datasets, demonstrating the effectiveness and potential of the low-precision training of GBDT.

1 Introduction

Gradient Boosted Decision Trees (GBDT) is a powerful machine learning algorithm. Despite the success of deep learning in recent years, GBDT is one of the best off-the-shelf choices of machine learning algorithms in many real-world tasks, including online advertising[30], search ranking[23, 14], finance prediction[26], etc. Along with carefully designed tools, including XGBoost[9], LightGBM[17] and CatBoost[21], GBDT has shown outstanding performance in various data science competitions[33] and industrial applications[29].

Despite the success of GBDT, we found there's still some space for GBDT algorithm to fully exploit modern computation resources. First, the training of GBDT requires arithmetic operations of high-precision FP (Floating Point) numbers, which hinders the usage of low-precision computation resources. Low-precision training has became a standard technique to significantly accelerate the training of neural networks[19]. The design of new processors for machine learning tends to encourage high throughput for low-precision computation. This results in a gap between GBDT training algorithm and the hardware architecture. Second, distributed training of GBDT is required for large datasets. The communication cost for distributed training, however, is huge due to high-precision statistic information passing among machines. The cost is especially high with a large number of features in the dataset, which hurts the scalability of GBDT in distributed systems.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

In this paper, we propose a low-precision training algorithm for GBDT, based on gradient quantization. To our best knowledge, this is the first low-precision training algorithm for GBDT. The main computation cost of training GBDT is the arithmetic operations over gradients. Before training of each decision tree, we quantize the gradients into very low-precision (e.g., 2-bit or 3-bit) integers. Thus we replace most of the FP arithmetic operations with integer arithmetic operations, which reduces computation time. In addition, low-bitwidth gradients result in a smaller memory footprint and better cache utilization. Several techniques are proposed to preserve the accuracy of the model, including stochastic rounding in gradient quantization, and leaf value refitting with original gradient values. We show both empirically and theoretically that quantized training of GBDT with low-bitwidth gradients is almost lossless in terms of model accuracy. Thus we empower GBDT to utilize the low-precision computation resources.

Distributed training of GBDT relies on the synchronization of gradient statistics. The gradient statistics are summarized into a histogram for each feature. With quantized gradients, only integer histogram values are needed in our algorithm. The communication among hosts or GPUs becomes sending and reducing of these integer values. And the size of these histograms is half of the original histograms with FP values. So quantized training of GBDT has an advantage in communication cost by nature, and improves the scalability of GBDT on distributed systems.

We implement our system on both CPUs and GPUs. The results show that our methods accelerate the training of GBDT in various settings, including single process on CPUs, single process on a GPU, and distributed training over CPUs. This validates that our algorithm is general for different types of computing resources. Besides acceleration on existing hardware architectures, our algorithm also lay the foundation for GBDT to exploit new hardware architectures with more flexible support for low-precision operations in the future. Based on LightGBM, we implement a quantized GBDT training system [1]. With quantized training, we achieve up to $2\times$ speedup compared with SOTA GBDT tools on both CPUs and GPUs. Experiments also show that quantized GBDT training scales much better in distributed systems.

Our algorithm also reveals huge redundancy in the information contained by gradient values in GBDT (2-bit or 3-bit gradient is enough for training), which is not known before. We believe that this interesting fact will inspire new understandings and improvements for GBDT in the future.

2 Related Work

Trials to use low-precision training in deep neural networks (NN) are abundant. Using lower bitwidth numbers in NN training can reduce memory access and accelerate training [13, 15, 32, 19]. In addition to efficiency, the effectiveness of quantized NN training has also been discussed[10, 8]. Besides quantization during the forward and backward calculations, quantization of gradients has also been applied in distributed training of neural networks[27] to reduce communication cost. Unlike NN, the possibility of low-precision training for GBDT is almost not discussed by existing literature.

Distributed training of GBDT is necessary for large-scale datasets. Training data are partitioned either by rows (data samples) or by columns (features) across different machines. We call the former strategy data-distributed training and the later feature-distributed training. The communication cost for feature-distributed training grows linearly with the number of training samples[5, 18]. On the other hand, the communication cost for data-distributed training is proportional with the number of features[5, 18]. The two strategies can be combined with each machine getting only part of features and samples[11]. Communication complexity of these strategies are analysis in detailed[28], with empirical studies[11]. Since GBDT often summarize the gradient statistics into a histogram for each feature, the cost for data-distributed training is reduced with the more concise histogram information[9]. Thus, data-distributed training is more favorable with large-scale datasets. With a large number of features, feature-distributed or the combination of two strategies can be faster[11].

Based on these distributed training strategies, some efforts have been made to reduce the communication cost. Meng et al[18] proposed Parallel-Voting Tree (PV Tree) to further reduce the cost of data-distributed training. However, PV Tree guarantees good performance only when different parts of the partitioned dataset have similar statistical distributions, which can require expensive random shuffling over the whole large dataset. Moreover, quantized training can be applied along with PV Tree. This is because PV Tree reduces communication cost by reducing the number of

[1] We will merge the code to LightGBM official repo.

features to synchronize the statistics in the histograms. Since quantized training can reduce the size of histograms, it can bring a general benefit for both PV Tree and ordinary data-distributed training. DimBoost[16] keeps the number of communicated histograms unchanged, but reduces the size of each histogram by compressing the histograms instead. Histograms represented by floating-point numbers in DimBoost are compressed into low-precision values (8-bit integers) before sending, and decoding into high-precision values. Note that in DimBoost, only the communication message for distributed training is in low-precision, and the compression is lossy. In addition, reduction of the histograms still requires floating-point additions. In comparison, we quantized the whole training process of GBDT with low-precision integers, and only need to maintain histograms with integer values.

In this paper, we compare our algorithm with SOTA efficient implementations of GBDT, including XGBoost[9], LightGBM[17] and CatBoost[21]. They support training on both CPUs and GPUs[31], and distributed training. However, none of these support low-precision training. We show that with quantized gradients, GBDT can be trained much faster compared with these SOTA tools in both single process and distributed settings, on both CPUs and GPUs. We implement quantized training based on LightGBM. But Our method is general and can be adopted by all these tools.

3 Preliminaries of GBDT Algorithms

Gradient Boosted Decision Trees (GBDT) is an ensemble learning algorithm that trains decision trees as base components iteratively. In each iteration, for each sample, the gradient and hessian of the loss function w.r.t. current prediction value are calculated. Then a decision tree is trained to fit the negative gradients. Formally, in iteration $k+1$, let $\hat{y}_i^k$ be the current prediction value of instance i. And g_i and h_i are the first-order and second-order derivatives of loss function l:

$$g_i = \frac{\partial l(\hat{y}_i^k, y_i)}{\partial \hat{y}_i^k}, h_i = \frac{\partial^2 l(\hat{y}_i^k, y_i)}{\left(\partial \hat{y}_i^k\right)^2} \tag{1}$$

We call g_i's as gradients and h_i's as hessians. For a leaf s, denote I_s as the set of data indices in the leaf. Let $G_s = \sum_{i \in I_s} g_i$ and $H_s = \sum_{i \in I_s} h_i$ be the summations of g_i's and h_i's over instances on leaf s. With the tree structure being fixed in iteration $k+1$, the training loss can be approximated by second-order Taylor series:

$$\mathcal{L}_{k+1} \approx \mathcal{C} + \frac{1}{N} \sum_s \left(\frac{1}{2} H_s w_s^2 + G_s w_s \right) \tag{2}$$

where $\mathcal{C}$ is a constant and w_s is the prediction value of leaf s. By minimizing the approximated loss, we obtain the optimal value for leaf s and the minimal loss contributed by data in I_s:

$$w_s^* = -\frac{G_s}{H_s}, \quad \mathcal{L}_s^* = -\frac{1}{2} \cdot \frac{G_s^2}{H_s}. \tag{3}$$

Finding the optimal tree structure is difficult. Thus the tree is trained by greedily splitting a leaf into two child leaves, starting from a single root leaf. When splitting leaf s into two children s_1 and s_2 in tree T_{k+1}, we can calculate the reduction in the approximated loss

$$\Delta \mathcal{L}_{s \to s_1, s_2} = \mathcal{L}_s^* - \mathcal{L}_{s_1}^* - \mathcal{L}_{s_2}^* = \frac{G_{s_1}^2}{2H_{s_1}} + \frac{G_{s_2}^2}{2H_{s_2}} - \frac{G_s^2}{2H_s} \tag{4}$$

To find the best split condition for leaf l, all candidate splits of all features should be enumerated and the one with the largest loss reduction is chosen.

To accelerate the best split finding process, an algorithm based on histograms is used by most state-of-the-art GBDT toolkits. The basic idea of histogram-based GBDT is to divide the values of a feature into bins, each bin corresponds to a range of the feature values. And construct histograms with the bins. Each bin will record the summation of gradients and hessians of data in that bin. Only the boundaries of the bins (ranges) will be considered as candidate split thresholds. Algorithm 1 describes the process of histogram construction. Bin data matrix $\mathbf{data}$ records for each feature and instance the index of the bin in the histogram, i.e. which range the feature value falls in. The g_i's and h_i's are accumulated in the corresponding bins in histograms of the features. Since the split criterion (4) only relates to the summation of gradient statistics, we can easily obtain the optimal split threshold by iterating over bins in the histogram. Traditionally 32-bit floating point numbers are used for g_i's and h_i's. And the accumulations in the histograms usually require 32-bit or 64-bit FP numbers. In next section, we show how to quantizes g_i's and h_i's into low-bitwidth integers so that most of the arithmetic operations can be replaced with integer operations, which substantially saves computation cost.

Algorithm 1 Histogram Construction for Leaf s

Input: Gradients $\{g_1, ..., g_N\}$, Hessians $\{h_1, ..., h_N\}$
Input: Bin data $\mathbf{data}[N][J]$, Data indices in leaf s I_s
Output: Histogram $\mathbf{hist}_s$
for $i \in I_s, j \in \{1...J\}$ **do**
  $bin \leftarrow \mathbf{data}[i][j]$
  $\mathbf{hist}_s[j][bin].g \leftarrow \mathbf{hist}_s[j][bin].g + g_i$
  $\mathbf{hist}_s[j][bin].h \leftarrow \mathbf{hist}_s[j][bin].h + h_i$

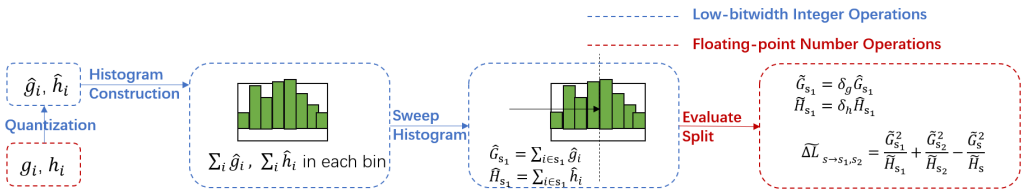

Figure 1: Workflow of the quantized GBDT training.

4 Quantized Training of GBDT

We describe the quantized training algorithm for GBDT. First we show the overall framework. Then we propose the two critical techniques to preserve the accuracy of quantized traninig, including stochastic gradient discretization and leaf value refitting.

4.1 Framework for Quantized Training

We first quantize g_i and h_i into low-bitwidth integers $\widetilde{g}_i$ and $\widetilde{h}_i$. To do so, we divide the ranges of all g_i's and h_i's in the dataset into intervals of equal length. To use B-bit integer gradients, we use $2^B - 2$ intervals. Each end of the intervals corresponds to a integer value, resulting in $2^B - 1$ integer values in total. Since the first-order derivatives g_i's can take both positive and negative values, half of the intervals will be allocated for the negatives values, and the other half for the positive values. For the second order derivatives h_i's, almost all common tasks have positive values. Thus the interval lengths are

$$\delta_g = \frac{\max_{i \in [N]} |g_i|}{2^{B-1} - 1}, \quad \delta_h = \frac{\max_{i \in [N]} h_i}{2^B - 2} \tag{5}$$

for g_i's and h_i's, respectively. Then we can calculate the low-bitwidth gradients

$$\widetilde{g}_i = Round\left(\frac{g_i}{\delta_g}\right), \quad \widetilde{h}_i = Round\left(\frac{h_i}{\delta_h}\right) \tag{6}$$

The function $Round$ rounds a floating point number into integer numbers. We left the detailed rounding strategy in Section 4.2. We replace g_i's and h_i's in Algorithm 1 with $\widetilde{g}_i$'s and $\widetilde{h}_i$'s. Then addition and subtraction arithmetic operations of the original gradients can be directly replaced by integer addition and subtraction. Concretely, the statistics g and h in the histogram bins will become integers. Thus accumulating gradients into histogram bins in Algorithm 1 only requires integer additions. In equation (4), G_{s_1}, H_{s_1}, G_{s_2} and H_{s_2} will replaced by their integer counterparts $\widetilde{G}_{s_1}$, $\widetilde{H}_{s_1}$, $\widetilde{G}_{s_2}$, and $\widetilde{H}_{s_s}$. We found that $B = 2$ or $B = 3$ is enough to preserve accuracy. As we will discuss in Section 6.1, in most cases, 8-bit or 16-bit integers will be enough to accumulate these gradients in histograms. Thus most of the operations are done with low-bitwidth integers. We only need floating point operations when calculating the split gain. Specifically, the split gain is estimated as

$$\Delta\widetilde{\mathcal{L}}_{k+1}^{*} = \frac{\left(\widetilde{G}_{s_1}\delta_g\right)^2}{2\widetilde{H}_{s_1}\delta_h} + \frac{\left(\widetilde{G}_{s_2}\delta_g\right)^2}{2\widetilde{H}_{s_2}\delta_h} - \frac{\left(\widetilde{G}_{s}\delta_g\right)^2}{2\widetilde{H}_{s}\delta_h}, \tag{7}$$

where the scale of the gradient statistics are recovered by multiplying δ_g and δ_h. Figure 1 summarizes the workflow for quantized GBDT training.

4.2 Rounding Strategies and Leaf-Value Refitting

We find that the $Round$ strategy in Equation (6) has a significant influence on the accuracy of quantized training algorithm. Rounding to the nearest integer number seems to be a reasonable choice.

However, we found the accuracy drops severely with this strategy, especially with small gradient bit-width. Instead, we adopt a stochastic rounding strategy. $\widetilde{g}_i$ randomly takes values $\lfloor g_i/\delta_g \rfloor$ or $\lceil g_i/\delta_g \rceil$ such that $\mathbb{E}[\widetilde{g}_i] = g_i/\delta_g$. The formal definitions of the two rounding strategies are

$$\mathrm{RN}\left(x\right) = \begin{cases} \lfloor x \rfloor, & x < \lfloor x \rfloor + \dfrac{1}{2} \\ \lceil x \rceil, & x \geq \lfloor x \rfloor + \dfrac{1}{2} \end{cases}, \quad \mathrm{SR}\left(x\right) = \begin{cases} \lfloor x \rfloor, & \text{w.p.} \quad \lceil x \rceil - x \\ \lceil x \rceil, & \text{w.p.} \quad x - \lfloor x \rfloor \end{cases} \tag{8}$$

where we use RN for round-to-nearest and SR for stochastic rounding. The key insight is that the split gain is calculated with the summation of gradients. And stochastic rounding would provide an unbiased estimation for the summations, i.e. $\mathbb{E}[\widetilde{G}\delta_g] = G$, and $\mathbb{E}[\widetilde{H}\delta_h] = H$ in Equation (7). The importance of stochastic rounding is also recognized in other scenarios for quantization, including NN quantization training[13] and histogram compression in DimBoost[16]. With stochastic rounding, we prove in Section 5 that, with proper regularization, the error in split gain estimation caused by quantized gradients is bounded by a small value with high probability.

With quantized gradients, the optimal leaf value in Equation (3) becomes $\widetilde{w}_s^* = -\frac{\widetilde{G}_s\delta_g}{\lambda + \widetilde{H}_s\delta_h}$. In most cases, $\widetilde{w}_s^*$ is enough for good results. But we found that for some loss functions, especially ranking objectives, refitting the leaf values with the original gradient values after the tree has stopped growing is useful to improve the accuracy of quantized training. It is cheap to recalculate w_s, since only a single pass over g_i's and h's is needed to sum them up into different leaves.

5 Theoretical Analysis

We prove that the difference in the gain of a split due to gradient quantization is bounded by a small number, with enough training data. The discussion is divided into two parts. We consider loss functions with constant second-order derivatives (i.e., the h_i's are constant). The square error loss function $l(\hat{y}, y) = \frac{1}{2}\left(\hat{y} - y\right)^2$ for regression tasks falls in this category. We leave the case for loss functions with non-constant second-order derivatives in Appendix D. We focus on the discussion of the conclusion in this section and leave details of the proof in Appendix B. Our theoretical analysis is based on a more specific form of weak-learnability assumption in boosting. Specifically, we consider only two-leaf decision trees (a.k.a. stumps) as weak learners.

Definition 5.1 (Weak Learnability of Stumps) Given a binary classification dataset $\mathcal{D} = \{(\mathbf{x}_i, c(\mathbf{x}_i))\}_{i=1}^N$ where $c(\mathbf{x}_i) \in \{-1, 1\}$ and weights $\{w_i\}_{i=1}^N$, where $w_{i_i} \geq 0$ and $\sum_i w_i > 0$, there exists $\gamma > 0$, and a two-leaf decision tree with leaf values in $\{-1, 1\}$ s.t. the weighted classification error rate on $\mathcal{D}$ is less than $\frac{1}{2} - \gamma$. Then the dataset $\mathcal{D}$ is γ-empirical weak-learnable by stumps w.r.t. c and $\{w_i\}_{i=1}^N$.

The weak-learnability of stump states that given a binary-class concept c and weights for dataset $\mathcal{D}$, there exists a stump which can produce slightly better classification accuracy than a random guess. Our analysis is based on the following assumption over the data in a single leaf s.

Assumption 5.2 For data subset $\mathcal{D}_s \subset \mathcal{D}$ in leaf s, there exists a stump and a $\gamma_s > 0$ s.t. $\mathcal{D}_s$ is γ_s-empirical weak-learnable by stumps, w.r.t. concept $c(x_i) = \mathrm{sign}(g_i)$ and weights $w_i = |g_i|$, where $i \in I_s$.

An equivalence of the assumption is: $\mathcal{D}$ is γ_s-empirical weak-learnable by stumps, w.r.t. concept $c(x_i) = \mathrm{sign}(g_i)$ and weights $w_i = |(g_i)|$ for $i \in I_s$ and $w_i = 0$ for $i \in [N]/I_s$. Similar assumption has been adopted in previous theoretical analysis for gradient boosting[12]. The difference is that here we restrict the weak learner to be a stump. In Appendix B, we show that for any leaf s with positive split gain, there exists such a $\gamma_s > 0$. As we will see in the experiments, for most leaves during the training of GBDT, γ_s won't be too small.

Let $\mathcal{G}_{s\to s_1,s_2} = \left|\mathcal{L}_{s_1}^*\right| + \left|\mathcal{L}_{s_2}^*\right|$, where $\mathcal{L}_{s_1}^*$ and $\mathcal{L}_{s_2}^*$ are defined in equation (3). And let $\widetilde{\mathcal{G}}_{s\to s_1,s_2} = \left|\widetilde{\mathcal{L}}_{s_1}^*\right| + \left|\widetilde{\mathcal{L}}_{s_2}^*\right|$ be the estimated version of $\mathcal{G}_{s\to s_1,s_2}$ after gradient quantization. And let $G_s^* = G_{s\to s_1^*,s_2^*}$ be the value for the optimal split $s \to s_1^*, s_2^*$ in leaf s. Note that on leaf s, the optimal split is chosen only according to $G_{s\to s_1,s_2}$, since $\mathcal{L}_s^*$ is a constant. Then based on Assumption 5.2, we have the following theorem.

Theorem 5.3 For loss functions with constant hessian value, if Assumption 5.2 holds for the subset $\mathcal{D}_s$ in leaf s for some $\gamma_s > 0$, then with stochastic rounding and leaf-value refitting, for any $\epsilon > 0$, and $\delta > 0$, at least one of the following conclusion holds:

1. With any split on leaf s and its descendents, the resultant average prediction values for data in $\mathcal{D}_s$ is no greater than ϵ.

2. For any split $s \to s_1, s_2$ on leaf s,

$$\frac{\left|\widetilde{\mathcal{G}}_{s\to s_1,s_2} - \mathcal{G}_{s\to s_1,s_2}\right|}{\mathcal{G}_s^*} \leq \frac{\max_{i\in[N]} |g_i| \sqrt{2\ln\frac{4}{\delta}}}{\gamma_s^2 \epsilon \cdot 2^B} \left(\sqrt{\frac{1}{n_{s_1}}} + \sqrt{\frac{1}{n_{s_2}}}\right) + \frac{\left(\max_{i\in[N]} |g_i|\right)^2 \ln\frac{4}{\delta}}{\gamma_s^2 \epsilon^2 n_s \cdot 4^{B-1}}. \quad (9)$$

In equation (9), the bound of gradients $\max_{i\in[N]} |g_i|$ is determined by the labels and the loss function. For regression tasks, we can normalize the labels to a fixed range, e.g., $[0, 1]$, thus the gradients are bounded. For classification tasks, the gradients are bounded (in the range $[-1, 1]$) by nature with cross-entropy loss. With enough data in leaf s, if the splits is not too imbalanced and partition enough data into both children s_1 and s_2, then we can expect the error caused by quantization is small. Theorem 5.3 ensures that with quantized training, either a split on a leaf has limited influence on the prediction values, or quantization does not change the gains of the splits too much.

In equation (9), we also notice that with more bits (larger B) to represent gradients in quantization, the error caused by quantization will be smaller. This is consistent with our intuition that with more quantized values, we can approximate the original gradients better. But in practice, we found that a smaller number of bits (e.g., 2 to 4 bits) is enough for a good accuracy.

We have a similar conclusion for the cases where hessians h_i's are not constant, due to limited space we left the details in Appendix B.

6 System Implementation

In this section, we show how to implement quantized GBDT for good efficiency. Current CPU and GPU architectures have limited support for the representation and calculation of low-precision numbers. For example, most CPUs by today only support 8-bit integers as the smallest data type. Under these limitations, the benefits of quantized training cannot be fully exploited. For example, though the gradients can be compressed into integers with only 2 to 4 bits, we have to use at least 8-bit arithmetics for accumulations in histograms. Even with these limitations, we achieve a considerable speedup on existing CPU and GPU architectures, given the techniques introduced in this section.

6.1 Hierarchical Histogram Buffers

Note that in Algorithm 1 most operations are accumulating gradients. Thus, low-bitwidth gradients will not bring much speedup if high-bitwidth integers are used to store the accumulations. On the contrary, integer overflow may occur with low-bitwidth integers storing the accumulations. To maximally exploit low-precision computation resources while avoiding integer overflow, we partition the training dataset by rows (instances), and assign one partition to a thread on CPU or multiple CUDA blocks on GPU. Thus, each thread or CUDA block constructs a local histogram over the instances in the assigned partition. Since the number of data per partition is much smaller, in most cases 8-bit or 16-bit integers for the accumulations in the histogram is enough. In the end, local histograms are reduced to the total histogram which may use more bits per bin, e.g. with 16-bit or 32-bit integers. Figure 2 displays this divide-and-merge strategy and how the bitwidth differs in local and total histograms.

6.2 Packed Gradient and Hessian

We pack the summations of gradients and hessians in a histogram bin into a single integer. For example, if we use two 16-bit integers to store the accumulation values, then the packed accumulation value would be a single 32-bit integer. Since gradients can be both negative or positive, the accumulation of gradients must reside in the upper part of the packed integer to exploit the signed bit. The accumulation of hessians resides in the lower part of the packed integer. When accumulating the gradient and hessian of an instance into a bin, the low-bitwidth gradient and hessian are packed in the same way before the addition.

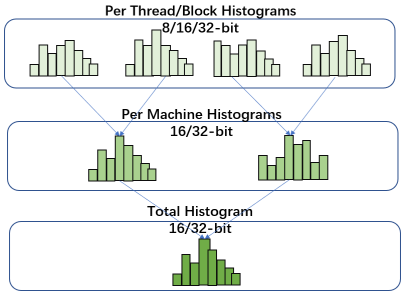

Figure 2: Hierarchical histogram buffers.

Name	#Train	#Test	#Attribute	Task	Metric
Higgs[6]	10,500,000	500,000	28	Binary	AUC↑
Epsilon[3]	400,000	100,000	2,000	Binary	AUC ↑
Kitsune[20]	14,545,195	3,999,993	115	Binary	AUC↑
Criteo[2]	45,840,617	6,042,135	67	Binary	AUC↑
Bosch[1]	1,000,000	183,747	968	Binary	AUC↑
Year [7]	463,715	51,630	90	Regression	RMSE↓
Yahoo LTR [4]	473,134	165,660	700	Ranking	NDCG@10↑
LETOR [22]	2,270,296	753,611	137	Ranking	NDCG@10↑

Table 1: Datasets used in experiments.

Table 2: Comparison of accuracy, w.r.t. different quantized bits.

Bitwidth	Binary-Class					Regression	Ranking	
	Higgs	Epsilon	Kitsune	Criteo	Bosch	Year	Yahoo LTR	LETOR
32-bit	0.845683	0.950220	0.951753	0.803841	0.702752	8.939880	0.794285	0.524303
2-bit	0.845648	0.949349	0.953491	0.803392	0.699541	8.952642	0.787795	0.518746
3-bit	0.845697	0.949997	0.951899	0.803399	0.702638	8.942274	0.791364	0.521963
4-bit	0.845638	0.950204	0.949618	0.803392	0.702614	8.945416	0.792965	0.523218
8-bit	0.845710	0.950228	0.951153	0.803704	0.703202	8.948614	0.793957	0.524708

Packed gradient and hessian halves the number of memory access in histogram construction. It also halves the number of integer additions. On GPU, the packing contributes significantly to the efficiency. This is because histogram construction on GPUs relies on atomic operations to ensure correctness when massive threads add to the same histogram. Thus packing also reduces the overhead of atomic operations.

7 Experiments

We evaluate our quantized GBDT training system on public datasets, and compare both accuracy and efficiency with other GBDT tools. First, we show that quantized GBDT training preserves the accuracy with low-bitwidth gradients. Then, we assess the efficiency on CPUs and GPUs of a single machine, and scalability with distributed training. Ablation study is provided to analyze the techniques for accuracy preserving and system implementation. Table 1 lists the datasets used for the experiments. For Criteo, we encode the categorical features by target and count encoding. We include open source GBDT tools XGBoost, LightGBM and CatBoost as baselines. We use the leaf-wise tree growing strategy[24] for all baselines. A full description of datasets and hyperparameter settings is provided in Appendix C.

7.1 Accuracy of Quantized Training

We evaluate the accuracy of quantized training. For each dataset and each algorithm, result on the best iteration is reported. We use gradients with 2-bit, 3-bit, 4-bit and 16-bit integers, and compare the results with full-precision gradients. We use AUC as evaluation metric for binary classification tasks. Table 2 lists the results. Here we focus on the effect of quantization on accuracy only, and due to limited space, we list only accuracy on CPUs here. Appendix C provides a similar table for GPU versions of these algorithms. With low-bitwidth gradients, the accuracy is not affected much. For datasets with smaller size, e.g., Epsilon, more bits are necessary to preserve accuracy. This echos the conclusion from our theoretical analysis in Section 5. But for some large datasets, the accuracy is less affected by the number of bits. In fact, low-bitwidth gradients even provide better results compared with full-precision gradients for large datasets. With stochastic rounding strategy, the low-bitwidth gradients contain random noise. Recent studies argue that noise in gradients can benefit the genearlization of NN[25]. We conjecture that the same conclusion may also hold for GBDT.

7.2 Speedup on Standalone Machine

We evaluate the acceleration brought by quantized training on a single machine by comparing it with CPU and CUDA implementations of popular open source GBDT tools. Our quantized training on CPU is based on LightGBM. We compare our CPU implementation with XGBoost, CatBoost and

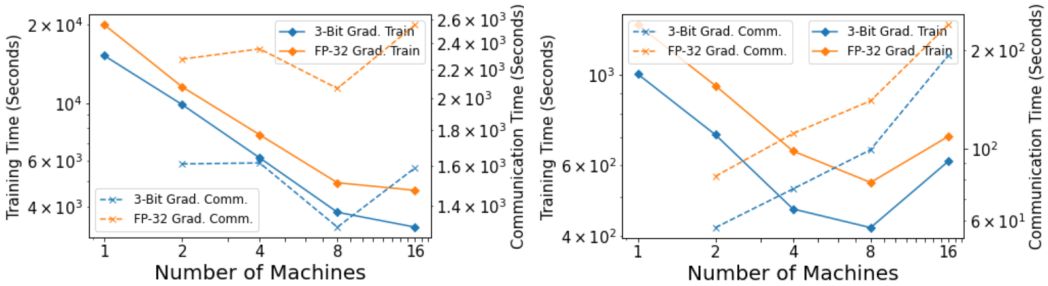

Figure 3: Scaling on distributed systems, w.r.t. #machines. Left: Epsilon-8M; Right: Criteo.

LightGBM with full-precision gradients. For the comparison on GPU, we implement a new CUDA version of LightGBM (denoted as LightGBM+, details are in Appendix E) and further implement quantized training based on it. We use 3-bit gradients for quantized training. We summarize the benchmark results in the Table 3. Overall, quantized training achieves comparable accuracy with a shorter training time. For the histogram construction time, quantized training speedups histogram construction by up to 3 times On GPU, with an average overall speedup up to 2.3 times. We also see the acceleration on CPU, but with relatively small gains compared with GPU. In summary, quantized training pushes forward the SOTA efficiency of GBDT algorithms.

Table 3: Detailed time costs for different algorithms in different datasets (seconds).

	Algorithm	Bosch	Criteo	Epsilon	Higgs	Kitsune	Year	Yahoo LTR	LETOR
GPU total time	XGBoost	77	404	128	30	208	**16**	42	50
	CatBoost	21	199	100	59	81	32	58	N/A
	LightGBM+	22	103	87	29	80	24	30	42
	LightGBM+ 3-bit	**12**	**69**	**38**	**26**	**40**	18	**24**	**34**
CPU total time	XGBoost	326	1243	2697	201	606	62	213	155
	CatBoost	2829	11880	1659	1607	2023	130	761	1283
	LightGBM	111	861	824	158	533	42	136	160
	LightGBM 3-bit	**81**	**675**	**787**	**122**	**296**	**23**	**92**	**127**
GPU Histogram time	LightGBM+	17	70	47	11	52	10	11	18
	LightGBM+ 3-bit	8	24	14	5	17	4	8	10
CPU Histogram time	LightGBM	99	631	720	101	368	31	112	104
	LightGBM 3-bit	70	421	689	62	180	9	63	128

7.3 Speedup of Distributed Training

We evaluate quantization in distributed GBDT training on an enlarged version of Epsilon dataset, which duplicates 20 copies of the training set of the original Epsilon dataset, and Criteo. Figure 3 shows how training time and communication time varies with different number of machines. Quantization and histogram compression can consistently reduce the communication cost.

7.4 Ablation Study

We analyze the influence of proposed techniques which ensure the effectiveness and efficiency of quantized training. We also provide a discussion on the feasibility of the weak-learnability assumption used in Section 5.

7.4.1 Rounding Strategies and Leaf-Value Refitting

Stochastic rounding plays an important role in the accuracy of quantized training. Table 4 compares the accuracy on test sets of Higgs and Epsilon with round-to-nearest (denoted as RN) and stoathastic rounding (denoted as SR), with different number of gradient bits. For both strategies, the AUC with and without leaf-value refitting is recorded. With smaller number of bits, stochastic rounding significantly outperforms round-to-nearest strategy. Note that leaf value refitting is not able to compensate the drawback of round-to-nearest. Table 4 also lists the influence of leaf value refitting to accuracy. For binary classification, we found that without leaf value refitting, we can still achieve a comparable accuracy. For regression and ranking tasks, leaf value refitting is important.

Table 4: Rounding strategies and leaf refitting.

Dataset	Strategy	3-bit	4-bit	8-bit	full
Higgs AUC↑	$RN_{\text{no refit}}$	0.841935	0.848074	0.849689	0.845683
	$SR_{\text{no refit}}$	0.845835	0.845874	0.845544	
	RN_{refit}	0.810151	0.826712	0.845057	
	SR_{refit}	0.845880	0.845683	0.845383	
Year RMSE↓	$RN_{\text{no refit}}$	10.499000	9.080890	8.947480	8.939880
	$SR_{\text{no refit}}$	8.999710	8.952980	8.963610	
	RN_{refit}	9.055260	8.938720	8.938720	
	SR_{refit}	8.933580	8.941520	8.956000	
LETOR NDCG@10↑	$RN_{\text{no refit}}$	0.291848	0.295465	0.349985	0.524303
	$SR_{\text{no refit}}$	0.506135	0.515961	0.524366	
	RN_{refit}	0.386724	0.351395	0.411673	
	SR_{refit}	0.521757	0.523507	0.524438	

Table 5: Speedup by packing gradient and hessian.

Algorithm	Criteo		Higgs	
	Overall	Histogram	Overall	Histogram
CPU/Packing	675	421	122	62
CPU/No Packing	946	693	156	94
GPU/Packing	69	24	26	5
GPU/No Packing	291	255	47	29

7.4.2 Packed Gradient and Hessian

Table 5 summarizes the training time and histogram construction time with and without packing gradient and hessian, testing with 3-bit gradients. Packing is important for the speed of histogram construction on both CPU and GPU.

7.4.3 Histogram Bitwidth

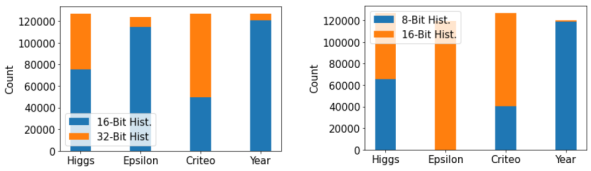

Figure 4: Histogram bitwidth frequency. L: CPU; R: GPU.

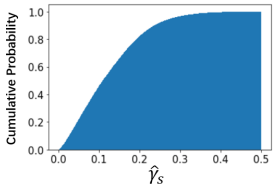

Figure 5: Cumulative distribution of $\hat{\gamma}_s$.

Figure 4 shows the frequency of histograms of different bitwidths used during training with 3-bit gradients. On CPU, our per-thread histograms can use either 16-bit or 32-bit integers. On GPU, our per-block histograms can use either 8-bit or 16-bit integers. The bitwidth in histograms is determined by the number of data in the leaves, the threads/blocks to use, and the gradient bitwidth. The minimum bitwidth of histogram integers that won't result in an integer overflow is chosen. The figure shows that quantized training exploits low-precision computations on both CPUs and GPUs. On GPUs, we found that the usage of 8-bit histograms makes little difference to the training time. For example, the Epsilon achieves significant speedup (see Table 3) almost all with 16-bit histograms in blocks.

7.4.4 Feasibility of Weak-Learnability Assumption

To verify the feasibility of Assumption 5.2, we should evaluate the largest γ_s with which a stump with weighted classification error rate $\frac{1}{2} - \gamma_s$ exists. This requires enumerating all the splits again and calculating the weighted error rate for each split, which is costing. Luckily, we have an easy approach to obtaining a lower bound of the largest γ_s, which can be naturally recorded during training of GBDT, we name the lower bound as $\hat{\gamma}_s$. Figure 5 shows the cumulative distribution of $\hat{\gamma}_s$ of all leaves during quantized training of Year dataset with 3-bit gradients. For most leaves (over 75%), a $\gamma_s > 0.05$ exists.

8 Discussion and Future Work

In this paper, we tried to answer an important but previously neglected question: Can GBDT exploit low-precision training? We propose a quantized training framework with low-bitwidth integer arithmetics which enables low-precision training of GBDT. We identify key techniques to preserve accuracy with quantized training, including stochastic rounding and leaf-value refitting. Theoretical analysis shows that quantization has limited influence over the selection of optimal splits in leaves, with enough training data. We implement our algorithm with both CPU and GPU versions. Experiments show that our quantized training GBDT method can achieve comparable accuracy with significant speedup over SOTA GBDT tools, with CPU, GPU and distributed clusters. We believe our method will bring new inspirations for GBDT community in future improvements of the algorithm as low-precision computation becomes a trend in machine learning.

References

[1] Bosch Dataset. https://www.kaggle.com/c/bosch-production-line-performance, 2022. [Online; accessed 19-May-2022].

[2] Criteo Dataset. http://labs.criteo.com/downloads/2014-kaggle-display-advertising-challenge-dataset/, 2022. [Online; accessed 19-May-2022].

[3] Epsilon Dataset. http://largescale.ml.tu-berlin.de/instructions/, 2022. [Online; accessed 19-May-2022].

[4] Yahoo Learning To Ranking Dataset. https://webscope.sandbox.yahoo.com/, 2022. [Online; accessed 19-May-2022].

[5] Firas Abuzaid, Joseph K Bradley, Feynman T Liang, Andrew Feng, Lee Yang, Matei Zaharia, and Ameet S Talwalkar. Yggdrasil: An optimized system for training deep decision trees at scale. In *Advances in Neural Information Processing Systems*, pages 3817–3825, 2016.

[6] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.

[7] Thierry Bertin-Mahieux, Daniel PW Ellis, Brian Whitman, and Paul Lamere. The million song dataset. 2011.

[8] Jianfei Chen, Yu Gai, Zhewei Yao, Michael W Mahoney, and Joseph E Gonzalez. A statistical framework for low-bitwidth training of deep neural networks. *arXiv preprint arXiv:2010.14298*, 2020.

[9] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[10] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[11] Fangeheng Fu, Jiawei Jiang, Yingxia Shao, and Bin Cui. An experimental evaluation of large scale gbdt systems. *Proceedings of the VLDB Endowment*, 12(11):1357–1370, 2019.

[12] Alexander Grubb and J Andrew Bagnell. Generalized boosting algorithms for convex optimization. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 1209–1216, 2011.

[13] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[14] Ziniu Hu, Yang Wang, Qu Peng, and Hang Li. Unbiased lambdamart: An unbiased pairwise learning-to-rank algorithm. In *The World Wide Web Conference*, pages 2830–2836, 2019.

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

[16] Jiawei Jiang, Bin Cui, Ce Zhang, and Fangcheng Fu. Dimboost: Boosting gradient boosting decision tree to higher dimensions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1363–1376, 2018.

[17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30:3146–3154, 2017.

[18] Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, and Tie-Yan Liu. A communication-efficient parallel algorithm for decision tree. *Advances in Neural Information Processing Systems*, 29, 2016.

[19] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[20] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[21] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 6639–6649, 2018.

[22] Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, 13(4):346–374, 2010.

[23] Zhen Qin, Le Yan, Honglei Zhuang, Yi Tay, Rama Kumar Pasumarthi, Xuanhui Wang, Michael Bendersky, and Marc Najork. Are neural rankers still outperformed by gradient boosted decision trees? In *International Conference on Learning Representations*, 2020.

[24] Haijian Shi. *Best-first decision tree learning*. PhD thesis, The University of Waikato, 2007.

[25] Samuel Smith, Erich Elsen, and Soham De. On the generalization benefit of noise in stochastic gradient descent. In *International Conference on Machine Learning*, pages 9058–9067. PMLR, 2020.

[26] Meng Wan, Chongxin Deng, Siyu Yu, and JD Vectorlab. Diting: A real-time distributed feature serving system for machine learning. Technical report, EasyChair, 2019.

[27] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: ternary gradients to reduce communication in distributed deep learning. In *31st International Conference on Neural Information Processing Systems (NIPS 2017)*, 2017.

[28] Zeyi Wen, Qinbin Li, Bingsheng He, and Bin Cui. Challenges and opportunities of building fast gbdt systems. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21) Survey Track*.

[29] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: a learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proceedings of the VLDB Endowment*, 13(12):1976–1989, 2020.

[30] Xiao Yang, Daren Sun, Ruiwei Zhu, Tao Deng, Zhi Guo, Zongyao Ding, Shouke Qin, and Yanfeng Zhu. Aiads: Automated and intelligent advertising system for sponsored search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1881–1890, 2019.

[31] Huan Zhang, Si Si, and Cho-Jui Hsieh. Gpu-acceleration for large-scale tree boosting. In *SysML Conference*, 2018.

[32] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.

[33] Wenjun Zhou, Taposh Dutta Roy, and Iryna Skrypnyk. The kdd cup 2019 report. *ACM SIGKDD Explorations Newsletter*, 22(1):8–17, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]

(c) Did you discuss any potential negative societal impacts of your work? [No] We believe our method has no negative societal impacts.

(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...

(a) Did you state the full set of assumptions of all theoretical results? [Yes] All assumptions are strictly stated.

(b) Did you include complete proofs of all theoretical results? [Yes] We provide complete and strict proofs.

3. If you ran experiments...

(a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] The data we used are all from public databases.

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators? [Yes] We discuss all the used datasets in the experiment section

(b) Did you mention the license of the assets? [Yes]

(c) Did you include any new assets either in the supplemental material or as a URL? [No]

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No] We use publicly available benchmark datasets.

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]