ThunderGBM: Fast GBDTs and Random Forests on GPUs

Zeyi Wen[†], **Jiashuai Shi**[‡] {wenzeyi, shijiashuai}@gmail.com
Hanfeng Liu[†] t0917912@u.nus.edu.sg
Qinbin Li[†], **Bingsheng He**[†] {qinbin,hebs}@comp.nus.edu.sg
Jian Chen[‡] ellachen@scut.edu.cn
[†]*School of Computing, National University of Singapore, 117418, Singapore*
[‡]*School of Software Engineering, South China University of Technology, Guangzhou, 510006, China*

Editor:

Abstract

Gradient Boosting Decision Trees (GBDTs) and Random Forests (RFs) have been used in many real-world applications. They are often a standard recipe for building state-of-the-art solutions to machine learning and data mining problems. However, training and prediction are very expensive computationally for large and high dimensional problems. This article presents an efficient and open source software toolkit called *ThunderGBM* which exploits the high-performance Graphics Processing Units (GPUs) for GBDTs and RFs. ThunderGBM supports classification, regression and ranking. It uses identical command line options and configuration files as XGBoost—one of the most popular GBDT and RF libraries. ThunderGBM can be used through command line and Python, and can run on single or multiple GPUs of a machine. Our experimental results show that ThunderGBM outperforms the existing libraries while producing similar models, and can handle high dimensional problems where existing GPU-based libraries fail. Documentation, examples, and more details about ThunderGBM are available at `https://github.com/xtra-computing/thundergbm`.

Keywords: Gradient Boosting Decision Trees, Random Forests, GPUs, efficiency

1. Introduction

Gradient Boosting Decision Trees (GBDTs) and Random Forests (RFs) are widely used in advertising systems, spam filtering, sales prediction, medical data analysis, and image labeling (Chen and Guestrin, 2016; Goodman et al., 2016; Nowozin et al., 2013). For ease of presentation, we use GBDTs as a representative in the remaining of this article, rather than repeatedly mentioning both GBDTs and RFs. In contrast with deep learning, GBDTs are simple and relatively easy to explain. The wide use of GBDTs is largely due to the user-friendly open source toolkits such as XGBoost (Chen and Guestrin, 2016), LightGBM (Ke et al., 2017) and CatBoost (Prokhorenkova et al., 2018). Additionally, GBDTs have won many awards in recent Kaggle data science competitions. However, training GBDTs is often very time-consuming, especially for large and high dimensional problems.

GPUs have been used to accelerate many real-world applications (Dittamo and Cisternino, 2008), due to their abundant computing cores and high memory bandwidth. In this article, we propose a GPU-based software tool called *ThunderGBM* to improve the efficiency of GBDTs and RFs. ThunderGBM supports binary and multi-class classification,

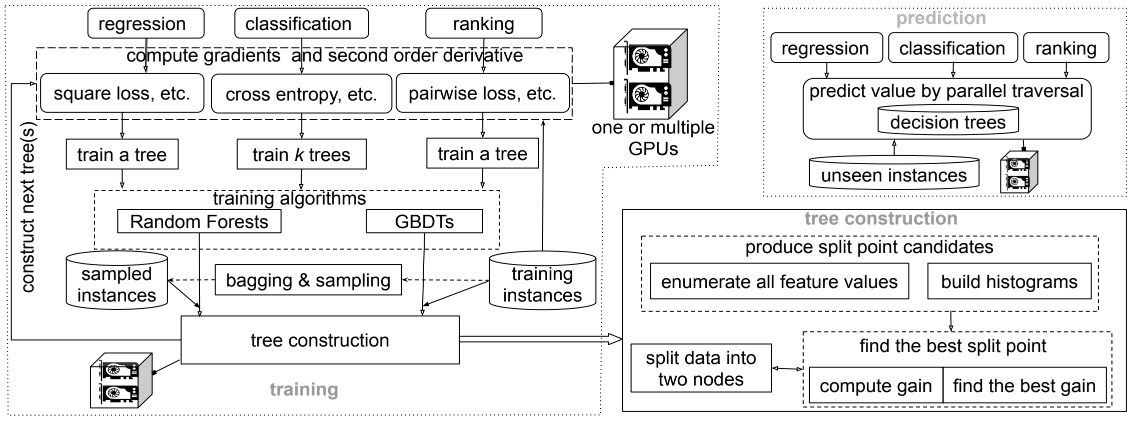

Figure 1: Overview of training and prediction in ThunderGBM.

regression and ranking. It uses the same command line options and configuration files as XGBoost—arguably the most popular GBDT library. Moreover, ThunderGBM supports the Python interface, and can run on single or multiple GPUs of a machine. Experimental results show that when the existing libraries run on CPUs, ThunderGBM is 6.4 to 10x times, 2.7 to 7.4 times, and 10.3 times faster than XGBoost, LightGBM and CatBoost, respectively, while producing similar models on the data sets tested. Compared with them on GPUs, ThunderGBM is 1 to 10x times, 1.9 to 10x times and 1.5 times faster than XGBoost, LightGBM and CatBoost, respectively. Importantly, ThunderGBM can handle high dimensional problems where those existing libraries fail. The key reason for the improvement is that existing libraries are CPU-oriented and use the GPU to accelerate only a part of GBDTs and/or require additional cost and data structures to support both CPU and GPU implementations, whereas ThunderGBM is GPU-oriented and maximizes GPU usage.

2. Overview and Design of ThunderGBM

Figure 1 shows the overview and software abstraction of ThunderGBM. The training algorithms for different tasks (i.e., classification, regression and ranking) are built on top of a generic tree construction module. This software abstraction allows us to concentrate on optimizing the performance of tree constructions. Different tasks only require different ways of computing the derivatives of the loss functions. Notably, the multi-class classification task requires training k trees where k is the number of classes (Bengio et al., 2010; Chen and Guestrin, 2016), while regression and ranking only require training one tree per iteration. The prediction module is relatively simple, and is essentially computing predicted values by concurrent tree traversal and aggregating the predicted values of the trees on GPUs. Here, we focus on the training on a single GPU. More details about using multiple GPUs and the prediction are in the supplementary file Appendix A. We develop a series of optimizations for the training. For each module that leverages GPU accelerations, we propose efficient parallel algorithmic design as well as effective GPU-aware optimizations. The techniques are used to support two major components in ThunderGBM: (i) computing the gradients and second order derivatives, and (ii) tree construction. Different from other libraries, ThunderGBM is GPU-oriented and optimizes GPU usage in-depth.

2.1 Computing the Gradients and Second Order Derivatives on GPUs

Denoting y_i and $\hat{y}_i$ the true and predicted target value of the i-th training instance, the gradients and second order derivatives are computed using the predicted values and the true values by $g_i = \partial l(y_i, \hat{y}_i)/\partial \hat{y}_i$ and $h_i = \partial^2 l(y_i, \hat{y}_i)/\partial \hat{y}_i^2$. The gradient and second order derivative of the loss function are denoted by g_i and h_i, respectively; $l(y_i, \hat{y}_i)$ denotes the loss function. ThunderGBM supports common loss functions such as mean squared error, cross-entropy and pairwise loss (De Boer et al., 2005; Cao et al., 2007; Lin et al., 2014). More details on loss functions and derivatives are in the supplementary file. Computing g_i and h_i requires the predicted value $\hat{y}_i$ of the i-th training instance, ThunderGBM computes $\hat{y}_i$ based on the intermediate training results. This is because the training instances are recursively divided into new nodes and are located in the leaf nodes at the end of training each tree. The idea of obtaining the predicted values efficiently is also used in LightGBM. To exploit the massive parallelism of GPUs, we create a sufficient number of threads to efficiently use the GPU resources. Each GPU thread keeps pulling an instance and computes its g and h.

2.2 Tree Construction on GPUs

Tree construction is a key component and time consuming in the GBDT training. We adopt and extend novel optimizations in our previous work (Wen et al., 2018, 2019) to improve the performance of ThunderGBM. Tree construction contains two key steps: (i) producing the split point candidates, and (ii) finding the best split for each node.

Step (i): ThunderGBM supports two ways of producing the split point candidates: one based on enumeration and the other based on histograms. The former approach requires the feature values of the training instances to be sorted in each tree node, such that it can enumerate all the distinct feature values quickly to serve as the split point candidates. However, the number of split point candidates may be huge for large data sets. The latter approach considers only a fixed number of split point candidates for each feature, and each feature is associated with a histogram containing the statistics of the training instances. Each bin of the histogram contains the values of the accumulated gradients and second order derivatives for all the training instances located in the bin. When using histogram-based training, the data is binned into integer-valued bins, which avoids having to sort the samples at each node, thus leading to significant speed improvement. In ThunderGBM, each histogram is built in two phases. Firstly, a partial histogram is built on the thread block level using shared memory, because a thread block only has accesses to a proportion of gradients and the second order derivatives. Secondly, all the partial histograms of a feature are accumulated to construct the final histogram. ThunderGBM automatically chooses the split point candidate producing strategy based on the data set density, i.e., histograms-based approach for dense data sets and enumeration-based approach for the others. The density is measured by $\frac{\text{total \# of feature values}}{\text{\# of instances} \times \text{\# of dimensions}}$. If the ratio is larger than a threshold, we choose the histogram-based approach; we choose the enumeration-based approach otherwise.

Step (ii): Finding the best split is to look for the split point candidate with the largest gain. The gain (Chen and Guestrin, 2016) of each split point candidate is computed by

data set			on two cpus (sec)			on the gpu (sec)				speedup (on cpus)			speedup (on gpu)		
name	card.	dim.	xgb	lgbm	cat	xgb	lgbm	cat	**ours**	xgb	lgbm	cat	xgb	lgbm	cat
higgs (reg)	11M	28	44.6	22.0	67.9	9.9	12.3	10.1	6.6	6.8	3.3	10.3	1.5	1.9	1.5
log1p (reg)	16K	4M	oom	189	oom	oom	261	oom	25.6	n.a.	7.4	n.a.	n.a.	10.2	n.a.
cifar10 (clf)	50K	3K	521	lerr	lerr	124	lerr	lerr	81.5	6.4	n.a.	n.a.	1.5	n.a.	n.a.
news20 (clf)	16K	62K	287	15.4	oom	109	16.5	oom	5.8	49	2.7	n.a.	18.8	2.8	n.a.
yahoo (rnk)	473K	700	18.8	11	n.a.	2.4	29.4	n.a.	2.4	7.8	4.6	n.a.	1.0	12.3	n.a.

Table 1: Comparison with XGBoost, LightGBM and CatBoost.

$gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right]$, where G_L and G_R (resp. H_L and H_R) denote the sum of g_i (resp. h_i) of all the instances in the resulting left and right nodes, respectively; λ is a regularization constant. In ThunderGBM, one GPU thread is dedicated to computing the gain of each split point candidate. The split point candidate with the largest gain is selected as the best split point for the node, which can be computed efficiently by a parallel reduction on GPUs. Once the best split point is obtained, the training instances in a node are divided into two child nodes. For producing the split point candidates by enumeration, ThunderGBM adopts the novel order preserving data partitioning techniques on GPUs proposed in our previous work (Wen et al., 2018), i.e., the feature values of the child nodes can be sorted more efficiently. For producing the split point candidates using histograms, each GPU thread determines which child node an instance should go to based on the best split. ThunderGBM repeats the two steps until a termination condition is met.

3. Experimental Studies

We conducted experiments on a Linux workstation with two Xeon E5-2640 v4 10 core CPUs, 256GB memory and a Tesla P100 GPU of 12GB memory. The tree depth is set to 6 and the number of trees is 40. More results on experiments and descriptions about the data sets can be found in the supplementary file Appendix C. Five data sets are used here, and regression, classification and ranking are marked with "reg", "clf" and "rnk", respectively. We used the versions of XGBoost, LightGBM and CatBoost on 21 Jul 2019.

The results are shown in Table 1, where "oom" stands for "out of memory", "lerr" stands for "large training error" and "n.a." stands for "not applicable". When the existing libraries are running on CPUs, ThunderGBM is 6.4 to 10x times, 2.7 to 7.4 times, and 10.3 times faster than XGBoost, LightGBM and CatBoost, respectively. When running on GPUs, ThunderGBM is 1 to 10x times, 1.9 to 10 times, and 1.5 times faster than XGBoost, LightGBM and CatBoost, respectively. Moreover, ThunderGBM can handle high dimensional problems (e.g., *log1p*) where the existing libraries fail or run slowly. ThunderGBM also has smaller or comparable errors to the existing libraries (cf. Appendix C).

4. Conclusion

In this article, we present *ThunderGBM* which supports classification, regression and ranking. ThunderGBM uses the same input command line options and configuration files as XGBoost, and supports the Python interface (e.g., scikit-learn). Our experimental results show that ThunderGBM outperforms the existing libraries while producing similar models, and can handle high dimensional problems where the existing libraries sometimes fail.

Appendix A. Multiple GPU Support and Parallel Prediction Algorithm

ThunderGBM supports multiple GPUs and parallel prediction. Next, we elaborate the details for these two functionalities.

A.1 Training on Multiple GPUs

One of the key limitations of GPUs is that the global memory size is relatively small (e.g., 12GB) compared with the size of main memory. A machine nowadays can have multiple GPUs, and commonly can host two to four GPUs. ThunderGBM can leverage multiple GPUs to train models on larger data sets that can fit into multiple GPUs. It supports a simple and effective approach for GBDT training on multiple GPUs. In particular, we partition the training data by features to handle large data sets (i.e., column based partitioning). There are two advantages of the feature based partitioning. First, both enumeration based and histogram based techniques for split point candidate production are natively supported. Producing the split point candidates of a feature requires accessing all the values of the feature. Storing all the feature values of a feature in one GPU helps perform finding the split points more communication efficiently. Second, the GPUs do not need to exchange the partial histograms in order to find the approximate split points, since all the feature values of a feature are stored locally and the GPU can build the whole histogram. Hence, the GPUs only need to exchange the local best split point candidates in order to obtain the global best split point candidates for the tree nodes. This reduces the communication cost from $\mathcal{O}(n \times d \times b)$ to $\mathcal{O}(n \times d)$, where n is the number of tree nodes needed to split, d is the number of features in the training data set and b is the number of bins of the histograms. The intuition is that a histogram is replaced by a local best split point when communicating to other GPUs.

A.2 The Parallel Prediction Algorithm

The prediction module in ThunderGBM is relatively simple, because the prediction mainly involves tree traversal. ThunderGBM supports different types of tasks including classification, regression and ranking. These tasks are designed in a unified prediction algorithm: traversing the decision trees to obtain the predicted value of an input instance. In ThunderGBM, a GPU thread is dedicated to the prediction of an instance by traversing all the decision trees. Hence, the number of GPU threads equals to the number of instances that we want to predict. For fast feature value lookup, we store the instances in a dense form when the shared memory is sufficient to store the instances for prediction. Thus, given the dense storage of the instances, we can decide which child to go to in a constant time for an instance. If storing the instances for prediction in a dense form requires more memory than the shared memory, the prediction is performed on the training instances in their original (i.e., sparse) form.

Appendix B. Other design and implementation details

B.1 Compression techniques

In ThunderGBM, we use run-length encoding (RLE) to compress the feature values, such that the same feature values are treated as one value when computing the gain of a split point. We have noticed that the SpareBin technique, which is also a compression technique, is used in libraries such as LightGBM (Ke et al., 2017). However, SparseBin is different from our RLE compression technique. SparseBin avoids storing empty bins for a "sparse" histogram. SparseBin is efficient for histograms with many empty bins (i.e., bins not having any instance). In comparison, RLE avoids storing repeated feature values, and is more effective for our data layouts produced by the feature-based data partitioning.

B.2 Sparsity-aware

XGBoost uses the sparsity-aware technique to handle missing values more efficiently (Chen and Guestrin, 2016). Essentially, the missing values can go to the left child node or right child node depending on which node results in better loss reduction. In ThunderGBM, we also exploit this sparsity-awareness to handling the missing values. Moreover, we have developed the sparsity-aware technique for histogram building. Our approach can adapt to different degrees of sparsity. The key idea is that when the problem is sparse and high dimensional, we use the sorting based technique to construct the histograms. After sorting, our RLE compression is able to significantly reduce the data footprint. When the problem is not sparse and high dimensional, we use the locking-based histogram building technique to build the histograms of each non-root node. The histograms of the root nodes are constructed by the parallel reduction, as all the feature values are sorted by default and the histograms of the root nodes can be constructed in a lock free manner.

B.3 High dimensional data

For handling high dimensional problems, LightGBM uses "feature bundling" to reduce the dimensionality of the problem. The key idea is that correlated features are combined together. The technique to tackle high dimensional problems in ThunderGBM is different from that of LightGBM. ThunderGBM does not try to reduce the dimensionality of the problem and thus we train the models without losing the model quality. Instead, ThunderGBM exploits efficient histogram construction for high dimensional data based on stable sort. The key steps are as follow. In step one, we sort the feature values of each dimension. Then, in step two, the gradient and the second order derivative of the instances with the same feature are accumulated together to form a bin. After the accumulation, the histograms are constructed.

B.4 Key differences of ThunderGBM over the existing libraries

The key reason for the speedup achieved by ThunderGBM is that existing libraries are CPU-oriented and use the GPU to accelerate only a part of GBDTs and/or require additional cost and data structures to support both CPU and GPU implementations. In comparison, ThunderGBM is GPU-oriented and we perform in-depth optimization to maximize the usage of

data set	cardinality	dimension
covetype	581012	54
e2006	16087	150361
higgs	1.1x10^7	28
ins	13184290	35
log1p	16087	4272228
news20	19954	1355191
real-sim	72201	20958
susy	5x10^6	18

Table 2: Information of data sets used in the experiments.

the GPU. Specifically, we exploit (i) high dimensional data histogram building technique, (ii) feature-based data partitioning and (iii) library optimization purely for GPUs. The intuitions of the techniques are as follows. First, our proposed high dimensional data histogram building technique is for addressing the large memory consumption issue. Existing libraries reserve GPU memory for d histograms, one histogram for each feature. As a result, the libraries (e.g., XGBoost) run out of memory for high dimensional data (e.g., "log1p" and "news20") on our workstation. In comparison, our technique does not require reserving memory for the d histograms. ThunderGBM constructs the histograms based on sorting, and hence can build the histograms more memory efficiently. Essentially, each unique feature value becomes a bin for high dimensional problems. Second, the feature-based data partitioning allows a block to only build a histogram for the corresponding feature, without the need to consider the histograms of other features. Hence, the communication cost is lower than the instance-based data partitioning. The additional advantage of the feature-based data partitioning is that the RLE compression can be used to avoid storing repeated feature values. Thus, our approach is more memory efficient. Third, our library is specifically dedicated to GPUs. As a result, we can perform optimization for GPUs to a fuller extent compared with the other libraries. For example, XGBoost requires the construction of "DMatrix" which is helpful for their CPU based implementation, but can become an efficiency bottleneck for their GPU implementation. In comparison, ThunderGBM stores the training and prediction data in the simple yet efficient manner using arrays, such that the data can be efficiently transferred to and accessed on GPUs.

Appendix C. Additional experimental results

Experimental setup. We used six more publicly available data sets as shown in Table 2, and *higgs* and *log1p* have been used in our main text. The data sets were downloaded from the LibSVM website. The data sets cover a wide range of the cardinality and dimensionality. The experiments were conducted on the same workstation running Linux with two Xeon E5-2640v4 10 core CPUs, 256GB main memory and one Tesla Pascal P100 GPU of 12GB memory. Each program was compiled with the -O3 option. ThunderGBM was implemented in CUDA-C. The default tree depth is 6 and the number of trees is 40. We used the default values provided by the libraries for the other hyper-parameters (e.g., learning rate). The total time measured in all the experiments includes the time of data transfer via

data set			on two cpus			on the gpu			
name	card.	dim.	xgb	lgbm	cat	xgb	lgbm	cat	**ours**
higgs	11M	28	44.6±1.2	22.0±2.4	67.9±1.5	9.9±0.1	12.3±0.3	10.1±0.3	6.6±0.3
log1p	16K	4M	oom	189±2.1	oom	oom	261±3.6	oom	25.6±0.3
cifar10	50K	3K	521±1.4	lerr	lerr	124±1.4	lerr	lerr	81.5±0.2
news20	16K	62K	287±1.5	15.4±0.1	oom	109±1.8	16.5±0.2	oom	5.8±0.1
yahoo	473K	700	18.8±0.3	11.0±1.5	n.a.	2.4±0.2	29.4±0.1	n.a.	2.4±0.1

Table 3: Comparison with XGBoost, LightGBM and CatBoost.

PCI-e bus. Although ThunderGBM supports other loss functions, the loss function in our experiments for all the libraries (including ThunderGBM) is the mean squared error: $l(y_i, \hat{y}_i) = \sum_i (y_i - \hat{y}_i)^2$.

Outline. We first present the overall performance of ThunderGBM over the three libraries on the GPU, and then we study the impact of varying tree depth and the number of trees in ThunderGBM. Following that, we show the results on handling high dimensional data, training error comparison, and efficiency on training Random Forests. Finally, we compare the efficiency of prediction and model generality among different libraries.

C.1 Overall performance study

This set of experiments aims to study the improvement of execution time of ThunderGMB over the existing libraries XGBoost, LightGBM and CatBoost. The time reported in Table 1 is the average execution time of five runs. Here, we also provide the standard deviation of the libraries on each data set in the five runs. The results are shown in Table 3. As we can see from the table, the standard deviation of ThunderGBM is quite small.

C.1.1 Execution time comparison on the GPU

We measured the total time (including data transfer from main memory to GPUs via PCI-e bus) of training all the trees for ThunderGBM, XGBoost, LightGBM and CatBoost. During training, the split point candidates are found using the histogram based method, as LightGBM, CatBoost and the GPU version of XGBoost only support producing the split point candidates using histograms.

The results of the four GPU implementations of GBDTs are shown in Figure 2. The first observation is that ThunderGBM can handle all the data sets tested efficiently, and outperforms all the existing libraries. In comparison, XGBoost and CatBoost cannot handle high dimensional data sets such as *e2006*, *news20* and *log1p* (marked with “n/a”). This is because the GPU versions of XGBoost and CatBoost do not make use of data sparsity when building histograms, which leads to running out of GPU memory. Moreover, XGBoost took 27 seconds to handle *real-sim* which CatBoost cannot handle. ThunderGBM can handle *real-sim* and is significantly faster than XGBoost and LightGBM. The GPU version of LightGBM is more reliable than XGBoost and CatBoost and can handle all the data sets, although it is much slower than ThunderGBM on handling the data sets.

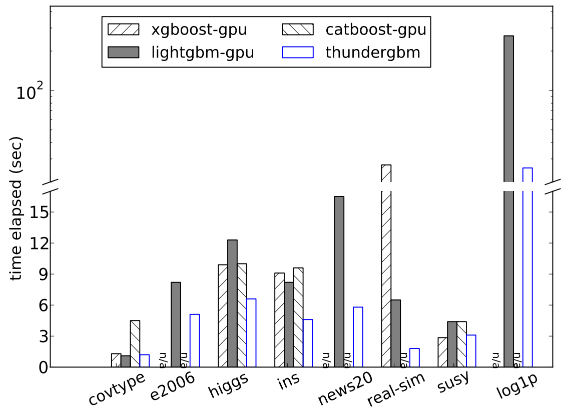

Figure 2: Comparison with XGBoost, LightGBM and CatBoost on the GPU.

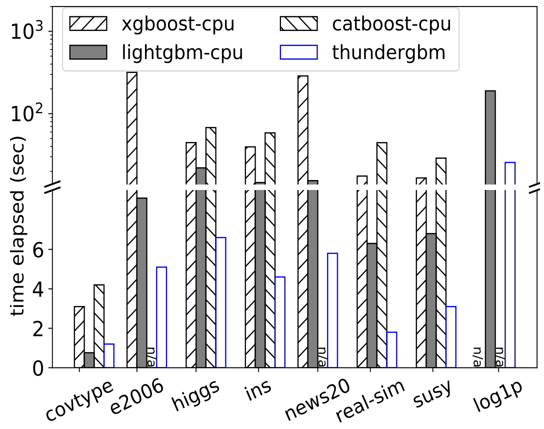

Figure 3: Comparison with XGBoost, LightGBM and CatBoost on CPUs.

C.1.2 Execution time comparison on CPUs

We study the speedup of ThunderGBM on the GPU over XGBoost, LightGBM and CatBoost on CPUs. Note that the number of CPU threads (i.e., 40 threads) in XGBoost is automatically selected by the XGBoost library. We have also tried XGBoost with 10, 20, 40 and 80 threads, and found that using 40 threads results in the shortest execution time for XGBoost in the eight data sets. Similarly for CatBoost, the number of CPU threads is chosen automatically by the libraries. We have noticed that LightGBM has an issue running tasks in our workstation, when the number of threads is set to 40, which results in significant downgrade on efficiency. Therefore, we tuned the number of threads for LightGBM, and found that setting the number of threads to 20 achieves the best efficiency. Hence, we set the number of threads to 20 for LightGBM in all the experiments.

The results of the three libraries on CPUs are shown in Figure 3, in comparison with ThunderGBM running on the GPU. Among the three libraries, LightGBM is more reliable compared with XGBoost and CatBoost. XGBoost runs out of memory on the *log1p* data set, while CatBoost runs out of memory on *e2006* and *news20* besides *log1p*. ThunderGBM

data set	covtype	e2006	higgs	ins	log1p	news20	real-sim	susy
XGBoost-GPU	3.45	oom	129.40	188.29	oom	oom	41.42	40.41
XGBoost-CPU	4.66	oom	222.63	215.18	oom	oom	118.13	54.50
LightGBM-GPU	0.84	11.98	11.98	9.84	270.92	14.26	3.92	4.49
LightGBM-CPU	0.55	16.60	19.05	16.52	202.44	12.74	3.70	7.45
CatBoost-GPU	5.11	oom	63.56	80.21	oom	oom	oom	23.15
CatBoost-CPU	6.01	oom	91.62	115.36	oom	oom	44.5	34.57
ThunderGBM	0.87	7.37	14.51	30.47	35.57	7.30	2.28	5.60

Table 4: End-to-end efficiency comparison among libraries.

data set	training time			accuracy		
	epsilon	higgs	svhn	epsilon	higgs	svhn
XGBoost-GPU	91.43	8.70	18.69	0.87	0.74	0.93
XGBoost-CPU	403.89	122.92	67.11	0.87	0.74	0.92
LightGBM-GPU	50.96	14.30	27.57	0.87	0.74	0.90
LightGBM-CPU	70.40	18.20	30.96	0.87	0.74	0.90
CatBoost-GPU	18.35	19.14	12.18	0.84	0.71	0.82
CatBoost-CPU	34.70	82.18	17.67	0.84	0.71	0.82
ThunderGBM	35.10	6.91	10.07	0.87	0.73	0.93

Table 5: Efficiency and predictive accuracy on binary classification.

is faster than or achieves competitive efficiency on all the data sets. For the *real-sim* data set, ThunderGBM achieves 10x times speedup over XGBoost and CatBoost.

C.1.3 End-to-end efficiency comparison among libraries

To compare the end-to-end efficiency of different libraries, we use the "fit" function provided in the Python scikit-learn interface by all the four libraries. We measured the total time for executing the "fit" function. Table 4 gives the results. As we can see from the results, ThunderGBM is significantly faster than XGBoost and CatBoost. ThunderGBM is also significantly faster than LightGBM in high dimensional problems and achieves similar efficiency as LightGBM in low dimensional problems. Another finding in our experiment is that LightGBM produces very large training error on the *ins* data set, although LightGBM is faster than the other libraries on the data set.

C.1.4 Additional results on binary classification and regression

To further investigate the performance of ThunderGBM on binary classification problems, we provide additional results here on *epsilon*, *higgs* and *svhn* from the LibSVM website. Table 5 shows the results. As we can see from the results, ThunderGBM is faster than the other libraries while producing better or similar model quality. We have noticed that the model quality of CatBoost is worse than the other libraries, when the tree depth and the number of trees are the same as the other libraries. This is because CatBoost trains the so-called "oblivious" decision trees.

techniques	covtype	higgs	real-sim	susy
memory pooling	46.41%	43.68%	57.11%	71.54%
RLE compression	28.28%	26.49%	2.27%	18.79%
fast histogram building	12.16%	5.47%	13.92%	6.64%

Table 6: Impact of each individual techniques

C.1.5 Impact of individual techniques

We present the results of improvement brought by individual techniques. The main techniques evaluated here include "memory pooling" which is designed to reduce the memory allocation operations on GPUs, RLE compression and fast histogram building. Other important techniques (e.g., the GPU-oriented optimization) cannot be easily switched on and off. Existing libraries are CPU-oriented, while ThunderGBM is GPU-oriented. We implemented a baseline version of ThunderGBM which can individually switch off the use of memory pooling, the RLE compression and the fast histogram building. The fast histogram building includes optimizations such as building the histograms of root nodes by parallel reduction, the composition of histogram bin IDs and histogram starting positions on the GPU memory, and avoiding locking to the histograms when the gradient or the second order derivative is zero.

Table 6 shows the impact of enabling memory pooling, RLE compression and histogram building. The impact of a technique t is computed by

$$\frac{(\text{total training time without } t) - (\text{total training time with } t)}{\text{total training time without } t}.$$

The results show that using memory pooling is important for improving the overall efficiency, and the RLE compression and fast histogram building also have notable impact on the overall training efficiency.

C.2 Varying tree depth and the number of trees

In this set of experiments, we study the effect of tree depth and the number of trees. When we varied the tree depth from 6 to 16, the number of trees was fixed at 40; when we varied the number of trees from 40 to 1280, the tree depth was fixed at 6. As the existing libraries run out of memory when handling data sets such as *news20* and *e2006*, we use *susy*, *higgs*, *covtype* and *ins* which all the libraries can handle in this set of experiments.

C.2.1 Effect of tree depth

Figures 4 to 7 show the efficiency results of varying tree depth. As we can see from the figures, ThunderGBM outperforms XGBoost on the GPU and is competitive when comparing with LightGBM and CatBoost on the GPU. An observation from Figure 6 is that LightGBM and CatBoost are not stable as they crashed when the tree depth increases in the *ins* data set. ThunderGBM consistently outperforms XGBoost. We noticed that CatBoost is the fastest when the tree depth goes beyond 14. However, we have found that CatBoost produces larger errors than the other libraries. Table 7 shows the differences of RMSE among different libraries on *covtype*. The RMSE of CatBoost is much larger than

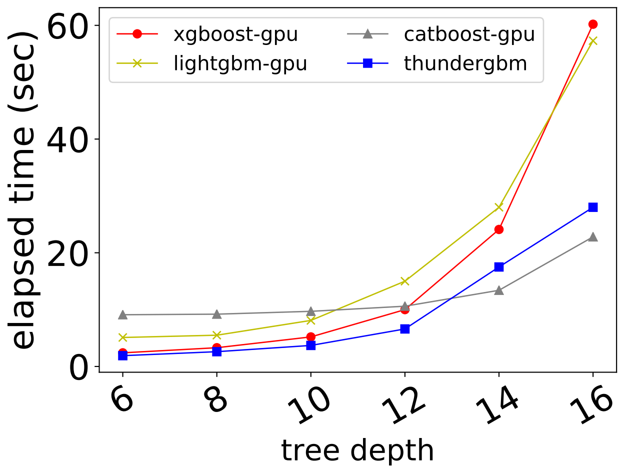

Figure 4: Efficiency on the *susy* data set.

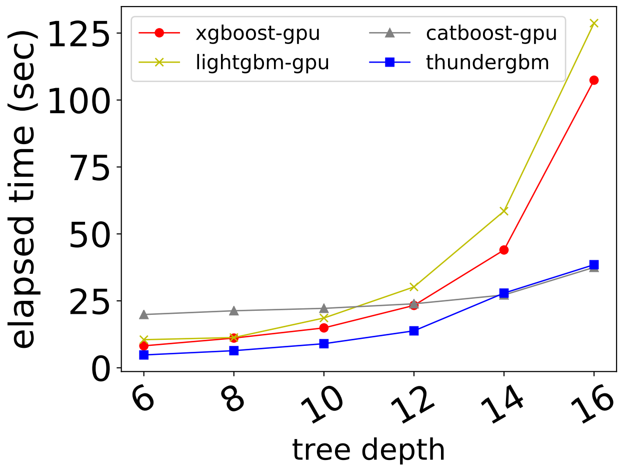

Figure 5: Efficiency on the *higgs* data set.

the other libraries, since CatBoost trains the so-called "oblivious" decision trees. We have observed that LightGBM also produces larger error than XGBoost and ThunderGBM.

depth	6	8	10	12	14	16
XGBoost (GPU)	0.73	0.57	0.41	0.31	0.24	0.21
LightGBM (GPU)	0.73	0.58	0.48	0.39	0.34	0.29
CatBoost (GPU)	0.85	0.78	0.70	0.62	0.56	0.49
ThunderGBM	0.73	0.59	0.44	0.32	0.26	0.23

Table 7: RMSE comparison among libraries.

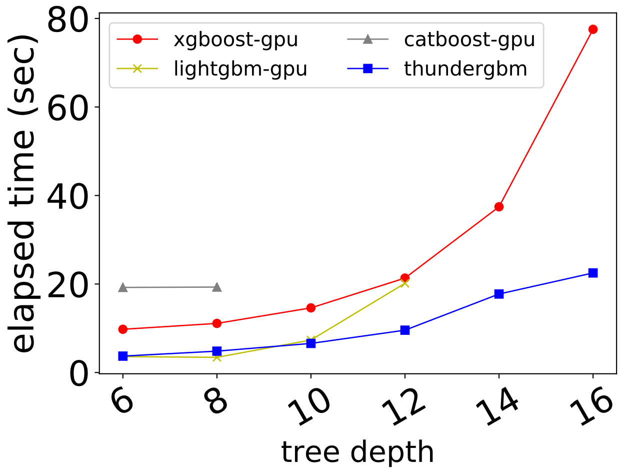

Figure 6: Efficiency on the *ins* data set.

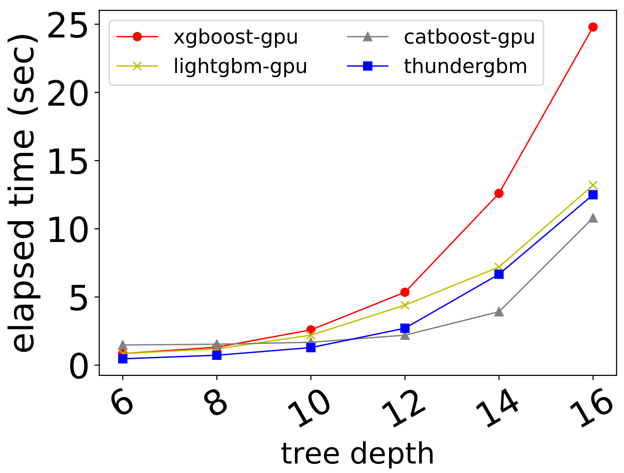

Figure 7: Efficiency on the *covtype* data set.

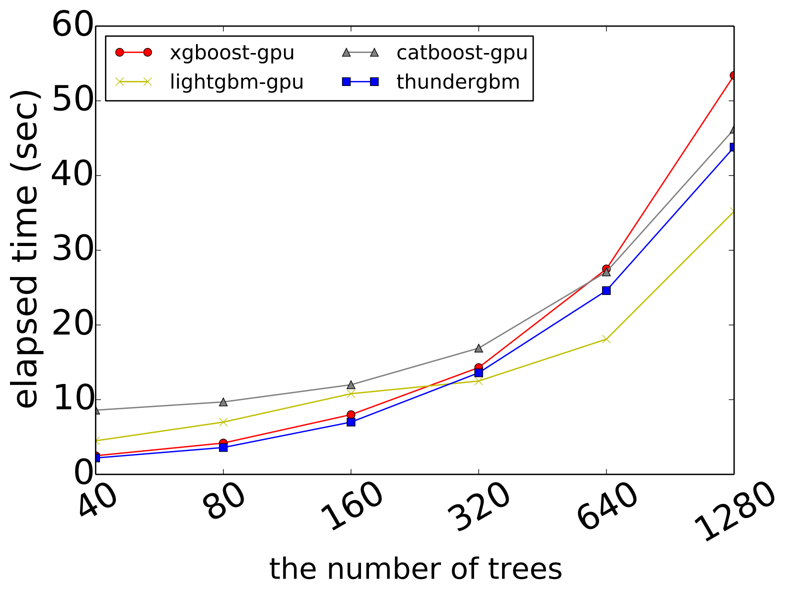

Figure 8: Efficiency on the *susy* data set.

data set	higgs	log1p	cifar10	news20	yahoo
XGBoost-GPU	0.18	oom	3.10	2.73	0.06
XGBoost-CPU	0.79	oom	13.0	7.18	0.47
LightGBM-GPU	0.16	6.53	lerr	0.41	0.74
LightGBM-CPU	0.31	4.73	lerr	0.39	0.28
CatBoost-GPU	0.18	oom	lerr	oom	n.a.
CatBoost-CPU	0.84	oom	lerr	oom	n.a.
ThunderGBM	0.11	0.64	2.04	0.15	0.06

Table 8: Average training time per tree.

C.2.2 Effect of the number of trees

We also study the effect of the number of trees on different libraries. The results are shown in Figures 8 to 11. Similar to that of varying tree depth, the key finding here is that ThunderGBM outperforms XGBoost and achieves similar efficiency as LightGBM and CatBoost. Please recall that CatBoost and LightGBM generally produce much larger error than XGBoost and ThunderGBM (cf. Table 7), which makes them run faster when the number of trees is above 100. On the *ins* data set, LightGBM is producing very different tree models and hence its results on the data set are not shown in Figure 10.

C.2.3 Average training time per tree

We also computed the average training time per tree. The results are shown in Table 8. As we can see from the results, ThunderGBM is faster than the other libraries.

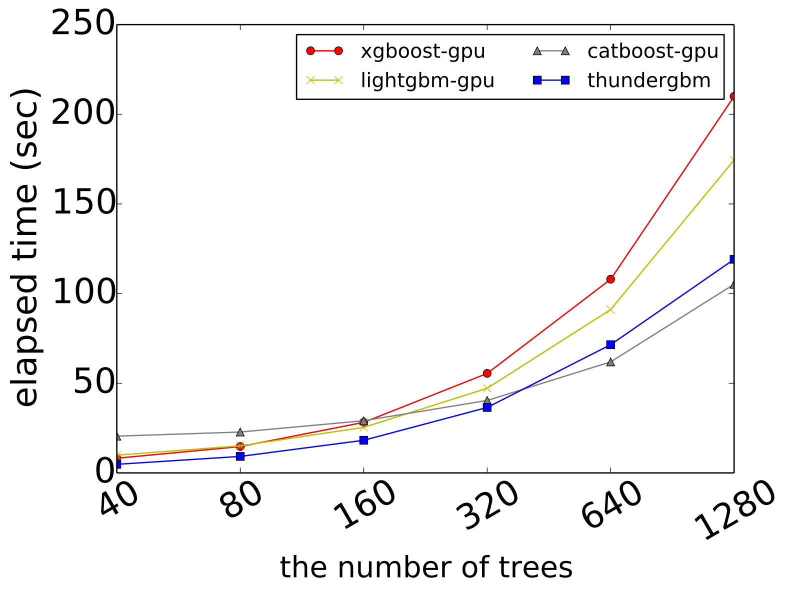

Figure 9: Efficiency on the *higgs* data set.

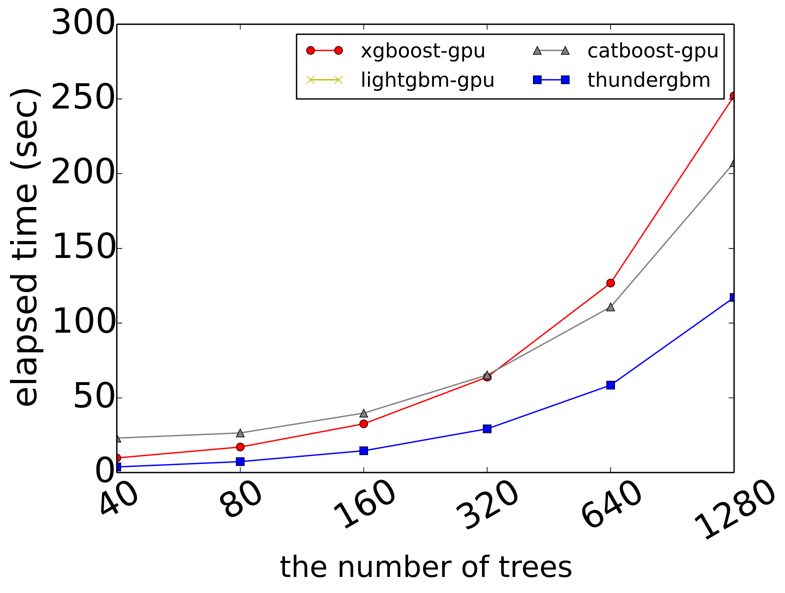

Figure 10: Efficiency on the *ins* data set.

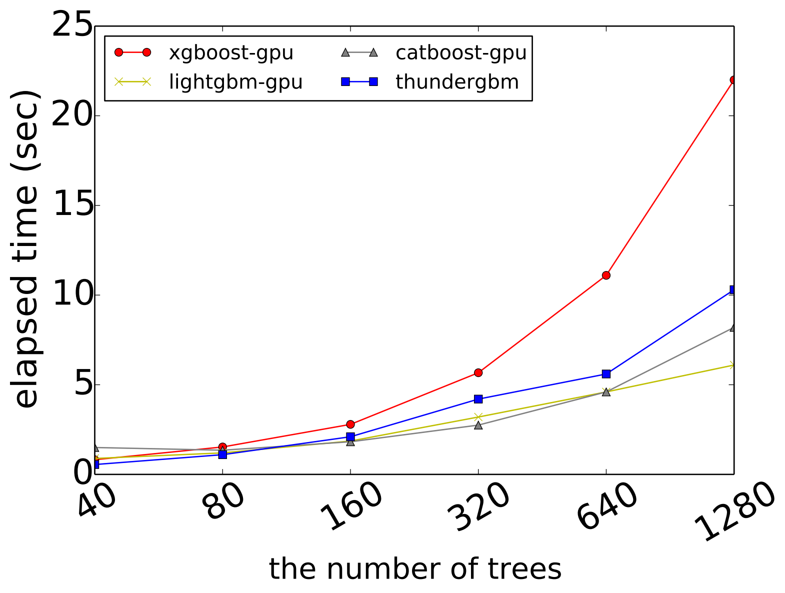

Figure 11: Efficiency on the *covtype* data set.

data set	XGBoost	LightGBM	CatBoost	ThunderGBM
e2006	9GB	76MB	9GB	76MB
log1p	256GB	0.4GB	256GB	0.4GB
news20	101GB	4.9MB	101GB	4.9MB

Table 9: Memory consumption comparison.

C.3 Handling high dimensional data

Here, we further investigate the memory consumption of ThunderGBM in comparison with XGBoost, LightGBM and CatBoost on *e2006*, *news20* and *log1p*. As XGBoost and CatBoost run out of memory for the data sets, we analyze the memory consumption of ThunderGBM and the other libraries. The memory for storing the training data sets are shown in Table 9. XGBoost and CatBoost require much more memory than LightGBM and ThunderGBM, because XGBoost and CatBoost use dense data representation while LightGBM and ThunderGBM use sparse data representation. Although LightGBM is as memory efficient as ThunderGBM, ThunderGBM is significantly faster than LightGBM on the high dimensional data sets as shown in Figure 2 and 3.

C.4 Training error comparison

We study the quality of trees learnt by different libraries in this set of experiments. In this set of experiments, we first investigate the training errors for the data sets listed in the main text. Then, we study the training errors using more data sets shown in Table 2 beyond the listed data sets.

We compare the training errors for the data sets shown in the main text. The results are shown in Table 10, where "n.a." stands for "not applicable". We used RMSE to

data set			xgboost	lightgbm	catboost	thundergbm	measure
name	card.	dim.					
higgs (reg)	11M	28	0.42	0.42	0.43	0.42	rmse
log1p (reg)	16,087	4,272,228	n.a.	0.29	n.a.	0.24	
cifar10 (clf)	50,000	3,072	0.83%	2.18%	45.22%	0.82%	prediction error rate
news20 (clf)	15,935	62,061	4.04%	5.68%	n.a.	3.78%	
yahoo (rnk)	473,134	700	0.882	0.903	n.a.	0.884	ndcg

Table 10: Training result comparison.

data set	xgboost	lightgbm	catboost	thundergbm
covtype	0.72	0.64	0.86	0.72
e2006	0.25	0.29	n.a.	0.25
higgs	0.42	0.42	0.43	0.42
ins	38.70	38.70	38.80	38.70
log1p	n.a.	0.29	n.a.	0.24
news20	0.49	0.35	n.a.	0.49
real-sim	0.47	0.37	0.52	0.47
susy	0.37	0.37	0.37	0.37

Table 11: RMSE comparison with XGBoost, LightGBM and CatBoost.

measure the regression tasks (marked with "reg"), used prediction error rate to measure the classification tasks (marked with "clf"), and used NDCG to measure the ranking task (marked with "rnk"). ThunderGBM obtains better or comparable results to the existing libraries as we can see from the table.

Next, we study the differences among different libraries using eight data sets in total. We used RMSE to measure the training error of different models. Table 11 shows the results. ThunderGBM produces similar RMSE as XGBoost, while CatBoost tends to have higher training errors.

C.5 Training Random Forests

Only XGBoost and ThunderGBM support RFs. Therefore, we only compare ThunderGBM with XGBoost in terms of RMSE and execution time. The results are shown in Table 12, where "lerr" stands for "large error". ThunderGBM produces similar or better RMSE than XGBoost on CPUs. XGBoost on GPUs results in large RMSE in some data sets (e.g., RMSE is 0.9 on the *susy* data set). In terms of efficiency, ThunderGBM is 1.5 to 10 times faster than XGBoost on CPUs. The improvement of ThunderGBM is more notable for large data sets such as *higgs* and *ins*. ThunderGBM is generally faster and more stable than XGBoost on GPUs as we can see from the results.

C.6 Prediction

We also evaluated the execution time of prediction of our library in comparison with the existing libraries. Table 13 provides the results of prediction time of ThunderGBM, existing libraries running on CPUs or running on the GPU. ThunderGBM is faster than XGBoost

data set	elapsed time (sec)			RMSE	
	xgboost (cpu)	xgboost (gpu)	thundergbm	xgboost	thundergbm
covtype	1.97	0.48	1.29	1.09	1.04
higgs	54.61	9.21	6.54	0.46	0.45
ins	52.99	10.47	5.3	38.99	38.98
susy	21.59	lerr	3.16	0.39	0.39

Table 12: Comparison on training Random Forests.

data set	covtype	e2006	higgs	ins	log1p	news20	real-sim	susy
XGBoost-GPU	3.00	oom	125	293	oom	oom	1.58	38.39
XGBoost-CPU	3.12	oom	130	300	oom	oom	1.55	39.95
LightGBM-GPU	0.10	0.36	2.76	2.78	2.69	0.33	0.02	1.19
LightGBM-CPU	0.10	0.30	2.86	3.47	2.50	0.32	0.02	1.26
CatBoost-GPU	3.52	oom	37.40	52.83	oom	oom	oom	11.87
CatBoost-CPU	3.56	oom	37.78	53.18	oom	oom	oom	11.79
ThunderGBM	0.18	0.12	4.18	5.60	0.57	0.07	0.07	1.68

Table 13: Efficiency: prediction time comparison among different libraries.

and CatBoost in prediction, and is faster than LightGBM on high dimensional data. For the smaller data sets, ThunderGBM achieves similar efficiency as LightGBM. We have noticed that XGBoost is much slower than other libraries in prediction. We have found that XGBoost requires construction of "DMatrix" which is very time consuming. The RMSE of the prediction results are identical to those shown in Table 11.

C.6.1 Model generality comparison

Here we compare the generalization quality of different libraries. We used five data sets from the LibSVM website where both training and test sets are available. The models were trained using the training data sets, and the prediction was performed on the test sets. The results are shown in Table 14. The results show that ThunderGBM has similar generality quality as the existing libraries.

data set	accuracy			rmse	
	svmguide1	letter	mnist	YearPredictionMSD	eunite2001
XGBoost-GPU	0.96	0.92	0.97	9.59	70.82
XGBoost-CPU	0.97	0.92	0.97	9.61	70.46
LightGBM-GPU	0.97	0.92	0.90	9.61	25.91
LightGBM-CPU	0.97	0.92	0.90	9.61	25.91
CatBoost-GPU	0.97	0.89	0.93	9.28	24.37
CatBoost-CPU	0.97	0.86	0.93	9.43	36.01
ThunderGBM	0.97	0.92	0.96	9.56	24.17

Table 14: Generalization: predictive accuracy comparison on test sets.

Appendix D. Loss functions of regression, classification and ranking

ThunderGBM supports the following loss functions: mean square error, logistic loss (Collins et al., 2002), cross-entropy loss (Moher and Gulliver, 1998), pairwise loss and NDCG loss (Ravikumar et al., 2011). Please note that for providing more information about the loss functions and to keep consistent with XGBoost, we use a special case of cross-entropy for logistic regression, and call the corresponding loss "logistic loss" similar to XGBoost (Chen and Guestrin, 2016).

D.1 Mean square error

The mean square error is used in regression in ThunderGBM. The option for the objective function is "reg:linear" following the convention of XGBoost which is arguably the most popular library for GBDTs and Random Forests. The goal of the training is the same as linear regression. The mean square error is defined as follows.

$$l(y_i, \hat{y}_i) = \frac{1}{2}(y_i - \hat{y}_i)^2$$

where y_i and $\hat{y}_i$ are the true target value and the predicted target value of the i-th training instance, respectively. Then, the gradient and second order derivative are $(y_i - \hat{y}_i)$ and 1, respectively.

D.2 Logistic loss

Logistic loss can be used in binary classification or regression for applications with target values between 0 and 1 (i.e., $y_i \in [0, 1]$). The option for the objective function is "reg:logistic" in ThunderGBM similar to XGBoost. The goal of the training is the same as logistic regression, and aims to minimize the logistic loss which is defined as follows.

$$l(y_i, \hat{y}_i) = -y_i \log(p_i) - (1 - y_i)\log(1 - p_i)$$

where $p_i = \frac{1}{1+e^{-\hat{y}_i}}$. The derivative of the loss function is derived as follows.

$$\frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i} = -y_i \cdot \frac{1}{p_i} \cdot p_i' - (1 - y_i) \cdot \frac{-1}{1 - p_i} \cdot p_i'$$

By substituting the derivative of p_i into the above equation, we obtain $\frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i} = p_i - y_i$. The second order derivative of the loss function is $\frac{\partial (p_i - y_i)}{\partial \hat{y}_i} = (1 - p_i)p_i$.

D.3 Cross-entropy loss

This loss is similar to logistic loss, but is used in the multi-class classification problems in ThunderGBM. The option for choosing this objective function is "multi:softmax" or "multi:softprob". The "multi:softprob" option has the same objective function as "multi:softmax", but the predicted value is a probability instead of a class label. Next, we present the loss function and derive its derivative. The cross-entropy loss is defined as follows.

$$l(y_i, \hat{y}_i) = -\sum_k y_i^k \log(p_i^k) \tag{1}$$

where k is the identifier of the k-th class and i is the identifier of the i-th training instance. Similar to the logistic loss, to help derive the gradient and second order derivative of the loss function, we first derive the derivative for the softmax function defined below.

$$p_i^k = \frac{e^{\hat{y}_i^k}}{\sum_{m=1}^{K} e^{\hat{y}_i^m}}$$

where K is the total number of classes. For ease of presentation, we ignore the subscript i which is the identifier of the i-th training instance. Then, we can write the above softmax function as $p^k = \frac{e^{\hat{y}^k}}{\sum_{m=1}^{K} e^{\hat{y}^m}}$. The derivative of the function is shown below.

$$\frac{\partial p^k}{\partial \hat{y}^m} = \frac{\partial \frac{e^{\hat{y}^k}}{\sum_{m=1}^{K} e^{\hat{y}^m}}}{\partial \hat{y}^m}$$

Now, we derive the gradient for the cross-entropy loss.

$$\frac{\partial l(y, \hat{y}^m)}{\partial \hat{y}^m} = -\sum_k y^k \cdot \frac{\partial log(p^k)}{\partial \hat{y}^m} = -\sum_k y^k \cdot \frac{1}{p^k} \cdot \frac{\partial p^k}{\partial \hat{y}^m}$$

As $\sum_k y^k = 1$, so we have $\frac{\partial l(y,\hat{y}^m)}{\partial \hat{y}^m} = p^m - y^m$. The second order derivative is $\frac{\partial^2 l(y,\hat{y}^m)}{\partial^2 \hat{y}^m} = p^m(1-p^m)$. In our implementation, a common normalization technique is used in computing p_i.

$$p_i^k = \frac{e^{\hat{y}_i^k}}{\sum_{m=1}^{K} e^{\hat{y}_i^m}} = \frac{N e^{\hat{y}_i^k}}{N \sum_{m=1}^{K} e^{\hat{y}_i^m}} = \frac{e^{\hat{y}_i^k + \log(N)}}{\sum_{m=1}^{K} e^{\hat{y}_i^m + \log(N)}}$$

The term $\log(N)$ is computed by $\log(N) = -\max(\hat{y}_i^k)$.

D.4 Pairwise loss and NDCG loss

Here, we present the loss functions used for ranking in ThunderGBM. In order to describe the loss functions, we define the true probability of a pair of training instances with indices i and j as follows.

$$P_{ij} = \frac{1}{2}(1 - S_{ij})$$

where $S_{ij} = -1$ if the i-th training instance is less relevant than the j-th training instance, $S_{ij} = +1$ if the i-th training instance is more relevant than the j-th training instance, and $S_{ij} = 0$ if the two training instances are the same.

We define the predicted probability of the pair of instances as follows.

$$\hat{P}_{ij} = \frac{1}{1 + e^{-\sigma(s_i - s_j)}}$$

where σ is a hyper-parameter, and s_i and s_j are the predicted scores of the ranking functions for the i-th and j-th training instance, respectively.

Following the existing literature (Burges, 2010), ThunderGBM also uses the cross-entropy loss defined below for ranking problems.

$$C_{ij} = -P_{ij}\log(\hat{P}_{ij}) - (1 - P_{ij})\log(1 - \hat{P}_{ij}) = \frac{1}{2}(1 - S_{ij})\sigma(s_i - s_j) + \log(1 + e^{-\sigma(s_i - s_j)}) \quad (2)$$

The derivatives over s_i and s_j are shown below.

$$\frac{\partial C}{\partial s_i} = -\frac{\partial C}{\partial s_j} = \sigma(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}})$$

The second order derivative of the loss function (cf. Equation 2) is as follows.

$$\lambda'_{ij} = \frac{\partial \lambda_{ij}}{\partial s_i} = \frac{-\sigma \cdot e^{\sigma(s_i - s_j)} \cdot \sigma}{(1 + e^{\sigma(s_i - s_j)})^2} \cdot |\Delta Z_{ij}| = -\sigma^2 \cdot |\Delta Z_{ij}| \cdot \frac{1}{1 + e^{\sigma(s_i - s_j)}} \cdot (1 - \frac{1}{1 + e^{\sigma(s_i - s_j)}})$$

Then the gradient of the i-th instance is computed as follows.

$$g_i = \sum_{\{i,j\} \in I} \lambda_{ij} - \sum_{\{j,i\} \in I} \lambda_{ji}$$

where I is the set of all pairs of the training instances. Similarly, we can compute the second order derivative for the i-th instance $h_i = 2 \cdot \sum_{\{i,j\} \in I} \lambda'_{ij}$. In ThunderGBM, $|\Delta Z_{ij}|$ equals to 1 when the objective function is pairwise loss; $|\Delta Z_{ij}|$ equals to the change of NDCG when the objective function is NDCG loss.

References

Samy Bengio, Jason Weston, and David Grangier. Label embedding trees for large multi-class tasks. In *NeurIPS*, pages 163–171, 2010.

Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *ICML*, pages 129–136. ACM, 2007.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *SIGKDD*, pages 785–794. ACM, 2016.

Michael Collins, Robert E Schapire, and Yoram Singer. Logistic regression, adaboost and bregman distances. *Machine Learning*, 48(1-3):253–285, 2002.

Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005.

Cristian Dittamo and Antonio Cisternino. GPU White paper, 2008.

Katherine E Goodman, Justin Lessler, Sara E Cosgrove, Anthony D Harris, Ebbing Lautenbach, Jennifer H Han, Aaron M Milstone, Colin J Massey, and Pranita D Tamma. A clinical decision tree to predict whether a bacteremic patient is infected with an extended-spectrum β-lactamase–producing organism. *Clinical Infectious Diseases*, 63(7):896–903, 2016.

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. In *NeurIPS*, pages 3149–3157, 2017.

Guosheng Lin, Chunhua Shen, and Jianxin Wu. Optimizing ranking measures for compact binary code learning. In *ECCV*, pages 613–627. Springer, 2014.

Michael Moher and T Aaron Gulliver. Cross-entropy and iterative decoding. *IEEE Transactions on Information Theory*, 44(7):3097–3104, 1998.

Sebastian Nowozin, Carsten Rother, Shai Bagon, Toby Sharp, Bangpeng Yao, and Pushmeet Kohli. Decision tree fields: An efficient non-parametric random field model for image labeling. In *Decision Forests for Computer Vision and Medical Image Analysis*, pages 295–309. Springer, 2013.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. CatBoost: unbiased boosting with categorical features. In *NeurIPS*, pages 6637–6647, 2018.

Pradeep Ravikumar, Ambuj Tewari, and Eunho Yang. On NDCG consistency of listwise ranking methods. In *AISTATS*, pages 618–626, 2011.

Zeyi Wen, Bingsheng He, Ramamohanarao Kotagiri, Shengliang Lu, and Jiashuai Shi. Efficient gradient boosted decision tree training on GPUs. In *International Parallel and Distributed Processing Symposium*, pages 234–243. IEEE, 2018.

Zeyi Wen, Jiashuai Shi, Bingsheng He, Jian Chen, Kotagiri Ramamohanarao, and Qinbin Li. Exploiting GPUs for efficient gradient boosting decision tree training. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2706–2717, 2019.