TF Boosted Trees: A scalable TensorFlow based framework for gradient boosting

Natalia Ponomareva, Soroush Radpour, Gilbert Hendry, Salem Haykal,
Thomas Colthurst, Petr Mitrichev, Alexander Grushetsky

Google, Inc.
`tfbt-public@google.com`

Abstract. TF Boosted Trees (TFBT) is a new open-sourced framework for the distributed training of gradient boosted trees. It is based on TensorFlow, and its distinguishing features include a novel architecture, automatic loss differentiation, layer-by-layer boosting that results in smaller ensembles and faster prediction, principled multi-class handling, and a number of regularization techniques to prevent overfitting.

Keywords: Distributed gradient boosting, TensorFlow

1 Introduction

Gradient boosted trees are popular machine learning models. Since their introduction in [3] they have gone on to dominate many competitions on real-world data, including Kaggle and KDDCup [2]. In addition to their excellent accuracy, they are also easy to use, as they deal well with unnormalized, collinear, missing, or outlier-infected data. They can support custom loss functions and are often easier to interpret than neural nets or large linear models. Because of their popularity, there are now many gradient boosted tree implementations, including scikit-learn [7], R gbm [8], Spark MLLib [5], LightGBM [6], XGBoost [2].
In this paper, we introduce another optimized and scalable gradient boosted tree library, **TF Boosted Trees (TFBT)**, which is built on top of the TensorFlow framework [1]. TFBT incorporates a number of novel algorithmic improvements to the gradient boosting algorithm, including new per-layer boosting procedure which offers improved performance on some problems. TFBT is open source, and available in the main TensorFlow distribution under `contrib/boosted_trees`.

2 TFBT features

In Table 1 we provide a brief comparison between TFBT and some existing libraries. Additionally, TFBT provides the following.
Layer-by-layer boosting. TFBT supports two modes of tree building: *standard* (building sequence of boosted trees in a stochastic gradient fashion) and novel *Layer-by-Layer* boosting, which allows for stronger trees (leading to faster convergence) and deeper models. One weakness of tree-based methods is the

fact that only the examples falling under a given partition are used to produce the estimator associated with that leaf, so deeper nodes use statistics calculated from fewer examples. We overcome that limitation by recalculating the gradients and Hessians whenever a new layer is built resulting in stronger trees that better approximate the functional space gradient. This enables deeper nodes to use higher level splits as priors meaning each new layer will have more information and will be able to better adjust for errors from the previous layers. Empirically we found that layer-by-layer boosting generally leads to faster convergence and, with proper regularization, to less overfitting for deeper trees.

Multiclass support. TFBT supports one-vs-rest, as well as 2 variations that reduce the number of required trees by storing per-class scores at each leaf. All other implementations use one-vs-rest (MLLib has no multiclass support).

Table 1: Comparison of gradient boosted libraries.

Lib	D?	Losses	Regularization
scikit-learn	N	R: least squares, least absolute dev, huber and quantile. C: logistic, Max-Ent and exp	Depth limit, shrinkage, bagging, feature subsampling
GBM	N	R: least squares, least absolute dev, t-distribution, quantile, huber. C: logistic, Max-Ent, exp, poisson & right censored observations. Supports *ranking*	Shrinkage, bagging, depth limit, min # of examples per node.
MLLib	Y	R: least squared and least absolute dev. C: logistic.	Shrinkage, early stopping, depth limit, min # of examples per node, min gain, bagging.
Light GBM	Y	R: least squares, least absolute dev, huber, fair, poisson. C: logistic, Max-Ent. Supports *ranking*.	Dropout, shrinkage, # leafs limit, feature subsampling, bagging, L1 & L2
XGBoost	Y	R: least squares, poisson, gamma, tweedie regression. C: logistic, Max-Ent. Supports *ranking* and **custom**.	L1 & L2, shrinkage, feature subsampling, dropout, bagging, min child weight and gain, limit on depth and # of nodes, pruning.
TFBT	Y	Any twice differentiable loss from tf.contrib.losses and **custom** losses.	L1 & L2, tree complexity, shrinkage, line search for learning rate, dropout, feature subsampling and bagging, limit on depth and min node weight, pre- post- pruning.

D? is whether a library supports distributed mode. *R* stands for regression, *C* for classification.

Since TFBT is implemented in TensorFlow, **TensorFlow specific features** are also available

- Ease of writing **custom loss** functions, as TensorFlow provides automatic differentiation [1] (other packages like XGBoost require the user to provide the first and second order derivatives).
- Ability to easily switch and compare TFBT with other TensorFlow models.
- Ease of debugging with TensorBoard.
- Models can be run on multiple CPUs/GPUs and on multiple platforms, including mobile, and can be easily deployed via TF serving.
- Checkpointing for fault tolerance, incremental training & warm restart.

3   TFBT system design

Finding splits. One of the most computationally intensive parts in boosting is finding the best splits. Both R and scikit-learn work with an exact greedy algorithm for enumerating all possible splits for all features, which does not scale. Other implementations, like XGBoost, work with approximate algorithms to build quantiles of feature values and aggregating gradients and Hessians for each bucket of quantiles. For aggregation, two approaches can be used [4]: either each of the workers works on all the features, and then the statistics are aggregated in Map-Reduce (MLLib) or All-Reduce (XGBoost) fashion, or a parameter server (PS) approach (TencentBoost [4], PSMART [9]) is applied (each worker and PS aggregates statistics only for a subset of features). The All-Reduce versions do not scale to a high-dimensional data and Map-Reduce versions are slow to scale.
TFBT Architecture. Our computation model is based on the following needs:

1. Ability to train on datasets that don't fit in workers' memory.
2. Ability to train deeper trees with a larger number of features.
3. Support for different modes of building the trees: standard one-tree-per-batch mode, as well as boosting the tree layer-by-layer.
4. Minimizing parallelization costs. Low cost restarts on stateless workers would allow us to use much cheaper preemptible VMs.

Fig. 1: TFBT architecture.
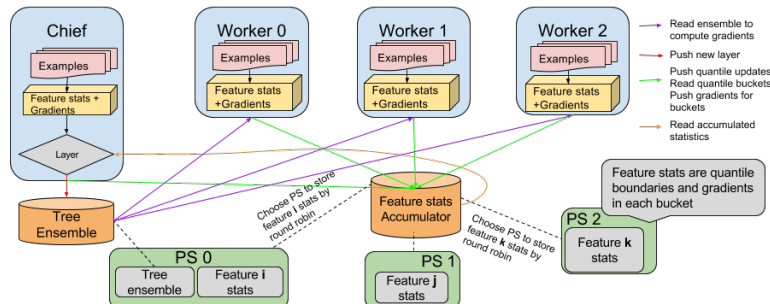
Our design is similar to XGBoost [2], TencentBoost [4] in that we build distributed quantile sketches of feature values and use them to build histograms, to be used later to find the best split. In TencentBoost [4] and PSMART [9] full training data is partitioned and loaded in workers' memory, which can be a problem for larger datasets. To address this we instead work on mini-batches, updating quantiles in an online fashion without loading all the data into the memory. As far as we know, this approach is not implemented anywhere else.
Each worker loads a mini-batch of data, builds a local quantile sketch, pushes it to PS and fetches the bucket boundaries that were built at the previous iteration. Workers then compute per bucket gradients and Hessians and push them back to the PS. One of the workers, designated as Chief, checks during each iteration if

---

Algorithm 1 Chief and Workers' work

```
 1: procedure CALCULATESTATISTICS(PS, MODEL, STAMP, BATCH_DATA, LOSS_FN)
 2:     predictions ← model.predict(BATCH_DATA)
 3:     quantile_stats ← calculate_quantile_stats(BATCH_DATA)
 4:     push_stats(PS,quantile_stats, stamp)                    ▷ PS updates quantiles
 5:     current_boundaries ← fetch_latest_boundaries(PS, stamp)
 6:     gradients, hessians ← calculate_derivatives(predictions,LOSS_FN)
 7:     gradients, hessians ← aggregate(current_boundaries,gradients, hessians)
 8:     push_stats(PS,gradients, hessians, size(BATCH_DATA), stamp)
 9: procedure DOWORK(PS, LOSS_FN, IS_CHIEF)              ▷ Runs on workers and 1 chief
10:     while true do
11:         BATCH_DATA ← read_data_batch()
12:         model ← fetch_latest_model(PS)
13:         stamp ← model.stamp_token
14:         CalculateStatistics(PS,model, stamp, BATCH_DATA,LOSS_FN)
15:         if is_chief & get_num_examples(PS, stamp) ≥ N_PER_LAYER then
16:             next_stamp ← stamp + 1
17:             stats ← flush(PS, stamp, next_stamp)         ▷ Update stamp, returns stats
18:             build_layer(PS, model, next_stamp, stats)    ▷ PS updates ensemble
```

---

the PS have accumulated enough statistics for the current layer and if so, starts building the new layer by finding best splits for each of the nodes in the layer. Code that finds the best splits for each feature is executed on the PS that have accumulated the gradient statistics for the feature. The Chief receives the best split for every leaf from the PS and grows a new layer on the tree.

Once the Chief adds a new layer, both gradients and quantiles become stale. To avoid stale updates, we introduce an abstraction called StampedResource - a TensorFlow resource with an int64 stamp. Tree ensemble, as well as gradients and quantile accumulators are all stamped resources with such token. When the worker fetches the model, it gets the stamp token which is then used for all the reads and writes to stamped resources until the end of the iteration. This guarantees that all the updates are consistent and ensures that Chief doesn't need to wait for Workers for synchronization, which is important when using preemptible VMs. Chief checkpoints resources to disk and workers don't hold any state, so if they are restarted, they can load a new mini-batch and continue.

References

1. Abadi, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. In: OSDI 16
2. Chen, T., et al.: XGBoost: A scalable tree boosting system. CoRR (2016)
3. Friedman, J.H.: Greedy function approximation: A gradient boosting machine. Annals of Statistics 29, 1189–1232 (2000)
4. Jiang, J., et al.: Tencentboost: A gradient boosting tree system with parameter server
5. Meng, X., Bradley, J., et al.: MLlib: Machine learning in Apache Spark (2016)
6. Microsoft: Microsoft/dmtk. https://github.com/microsoft/dmtk (2013)
7. Pedregosa, F., et al.: Scikit-learn: Machine learning in Python. JMLR 12 (2011)
8. Ridgeway, G.: Generalized boosted models: A guide to the gbm package (2005)
9. Zhou, J., et al.: PSMART: Parameter server based multiple additive regression trees system. WWW '17 Companion