Growing Cache Friendly Decision Trees

Efficient Online Inference of Gradient Boosted Tree Models

Niloy Gupta
Yelp Inc.
San Francisco, California
niloyg@yelp.com

Adam Johnston
Yelp Inc.
San Francisco, California
adamj@yelp.com

ABSTRACT

This paper discusses a cache efficient compression and re-ordering strategy for decision trees in order to reduce the latency of online predictions. In our use case of ad click prediction, we saw latency improve by a factor of three when compared to a standard decision tree implementation. We also incorporate these optimizations into an open source Java library compatible with models trained with DMLC XGBoost [3, 4] for the benefit of the machine learning community.

1 INTRODUCTION

One of the key challenges of computational advertising is to ensure that online model inference is fast enough to meet service latency requirements without compromising on prediction accuracy. Yelp's Ad Targeting system uses Gradient Boosted Trees (GBTs) for predicting an ad's click-through-rate (CTR). Evaluating a GBT model has a time complexity of $O(Td)$ and a space complexity of $O(T2^d)$ where T is the number of trees and d is the max depth of each tree. A typical CTR model at Yelp has millions of nodes and on the order of a thousand trees in order to obtain a desired level of accuracy. This makes online prediction a challenge.

A basic implementation of a decision tree involves connecting the nodes of the tree with pointers to its children. The disadvantage of this implementation is that the nodes are not necessarily stored adjacently in memory. Due to the simplicity of the tree traversal logic, much of the model evaluation time is spent dereferencing subsequent nodes. A popular approach to help solve this problem is to use a "flat tree approach" [1, 2, 6] where each decision tree in the model is converted into an array with nodes placed next to each other in memory. This approach still runs into issues as the breadth-first ordering results in non-adjacent reads, wasting cycles on memory stalls. Another approach is the "compiled tree" implementation [1, 6] where the decision tree is written as nested ternary expressions with the split condition and feature index as constants.

Both of these approaches can benefit from an optimized model structure. In this paper, we discuss a tree layout and node compression strategy which improves the speed of online inference for large GBT models by improving cache utilization.

2 CACHE OPTIMIZATIONS

2.1 Pre-Order Layout with *Cover* Statistic

A key aspect of the ad CTR prediction problem is that the dataset is heavily skewed towards non-clicks. As a result, during the model training phase, the split points partition the samples at each node into highly skewed subsets. In other words, we expect some branches to be taken much more frequently than others.

The DMLC XGBoost [3, 4] library has a *cover* statistic associated with each node defined as the sum of the second order gradient of the training data at that node. Therefore, a node's *cover* is correlated to the number of training samples seen at that node. Under the assumption that our training distribution reflects our online distribution, we can then express the probability of a node's conditional being evaluated as true as simply:

$$P[Conditional] = \frac{TrueChild_{Cover}}{Parent_{Cover}}$$

We then restructure the tree by performing a pre-order traversal and swapping the order of the child nodes at each step when the "False" child is more likely than the "True" child. This results in a pre-order indexing of the tree where the adjacent child node has a higher cover than its sibling and is therefore more likely to be visited.

In practice, the combination of the skewed CTR dataset with the process of boosting producing more specialized trees later in the ensemble results in very hot paths [5]. Our metric for evaluating the effectiveness of the pre-order layout with respect to *cover* is:

$$CoverBias = \frac{\sum max(TrueChild_{Cover}, FalseChild_{Cover})}{\sum Parent_{Cover}} \approx 0.95$$

This illustrates the extent to which heavily skewed decision trees benefit from this reordering, since effectively ~95% of traversal steps take the hot path.

2.2 Compressed Node Representation

A typical structure for a decision tree node would store the following information: split feature index, split feature value, leaf value, left and right child addresses, and the default child (in the case of sparse feature vectors[1]).

It can be observed that every node has either a split condition or a leaf value but never both. We therefore store whichever value is present in a single 32-bit field. Since we always place one child node immediately after its parent node in the array, the parent only needs to store the address of the distant child (i.e., the child not in the hot path). Since no address or offset would ever be zero, we store a zero in that field to indicate when the node is a leaf. Bit flags for the default path and for if the child nodes were swapped during restructuring can be stored in the unused sign bit of the two indexing fields.[2] The new conditional is then written such that *true* is always the adjacent child in memory.

[1]XGBoost selects either the true or false child as default for missing values. [4]
[2]Many languages, including Java, index arrays with signed 32-bit values.

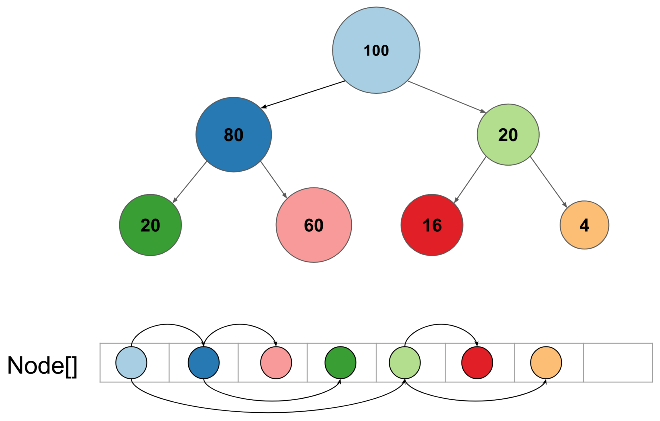

Figure 1: Each node is associated with a cover statistic that correlates with the number of samples at each node. Children with higher cover are placed next to their parent in memory which makes the tree ordering cache friendly. In the graphic, the number in each node indicates the cover for that node.

Compared to a basic implementation which would require 16 bytes just for the child pointers, this design compresses the contents of a node into 12 bytes without impacting the flexibility of the training or inference implementation. With a 64 byte cache line, this typically allows for 5 nodes along the current hot path to be loaded at once.

Split condition or leaf weight (32 bits)	
Distant child offset (31 bits)	True implies distant child (1 bit)
Feature index (31 bits)	Default path (1 bit)

3 OBSERVATION AND RESULTS

The limitations of this implementation require the following constraints: the number of nodes per tree cannot exceed 2^{31}, the number of features cannot exceed 2^{31}, and the online data must have the same skewed distribution as the offline training set. In practice, it is rare to train GBTs on more than 2 billion features or construct trees deeper than 31 levels. Additionally, models in production are often retrained to capture the changes in distribution to minimize the effects of feature drift. Hence these limitations do not restrict us from using these optimizations in online production environments.

For the benchmark results in Figure 2, we measured prediction latency against a random sample of production data with and without our described optimizations. Our implementation using compressed nodes with a cover based pre-order layout offered a 3.1x speedup compared to a flat implementation and a 2.6x speedup compared to an implementation using a compressed representation alone.

4 CONCLUSION AND FUTURE WORK

In this paper, we presented a cache efficient compression and layout of decision trees to make online prediction using GBT models faster. Our design shows significant speedups compared to standard implementations. We have incorporated these optimizations into

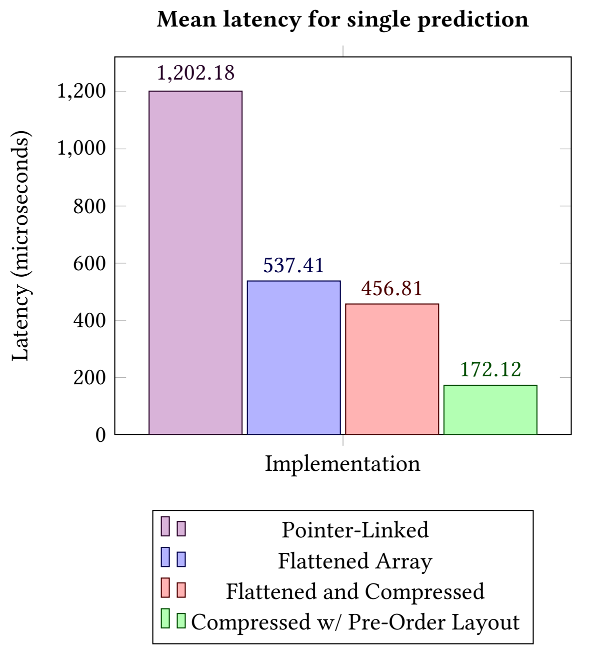

Figure 2: Prediction Latency Comparisons. The latency for the pointer-linked implementation is from an older benchmark and wasn't obtained with the rest of the results.

a Java library used in our production environment, which can be found at https://github.com/Yelp/xgboost-predictor-java.

Our decision to use pre-order indexing was based on an intuition that caching as much of the hot path as possible is always the best approach. However, in cases where any two child paths are followed with more equal probability, it might be better to cache both paths to less depth. Additionally, a compiled tree approach would benefit from our optimizations, which decrease memory stalls and take advantage of branch prediction. We plan to explore both of these improvements in the future.

5 ACKNOWLEDGEMENTS

We would like to extend our gratitude to Eric Liu, Joseph Malicki and Hossein Rahimi for their valuable insights during the design phase.

REFERENCES

[1] Oleksandr Kuvshynov Aleksandar Ilic. 2017. Evaluating boosted decision trees for billions of users. https://code.facebook.com/posts/975025089299409/evaluating-boosted-decision-trees-for-billions-of-users/. (27 March 2017).

[2] Komiya Atsushi. 2017. xgboost-predictor-java. https://github.com/komiya-atsushi/xgboost-predictor-java. (2017).

[3] Tianqi Chen. 2017. xgboost. https://github.com/dmlc/xgboost. (2017).

[4] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). arXiv:1603.02754 http://arxiv.org/abs/1603.02754

[5] K. V. Rashmi and Ran Gilad-Bachrach. 2015. DART: Dropouts meet Multiple Additive Regression Trees. *CoRR* abs/1505.01866 (2015). arXiv:1505.01866 http://arxiv.org/abs/1505.01866

[6] Andrew Tulloch. 2013. The Performance of Decision Tree Evaluation Strategies. http://tullo.ch/articles/decision-tree-evaluation/. (2 December 2013).