

Fast inference engine for Decision Trees

*Zampieri Luca** supervised by *Wunsch Stefan**[†], *Sitong An**[‡], *Kim Albertsson **[§] and *Lorenzo Moneta**
CERN, Meyrin, Switzerland

Abstract

Decision trees and derivatives such as Boosted Decision Trees, Adaboost, XGboost, Random forest are widely used in the world, and are now part of the High Energy Physics toolbox. However, High Energy Physics has some special requirements, it needs the inference of such tree to be as fast as possible, to be used during reconstruction of events or even trigger. In this work, we design and implement a fast inference engine for decision trees in C++ and benchmark it against XGBoost C API, showing a 15x improvement event-by-event and a 3x improvement for batched versions. The code is freely available on github. This is realized in the context of the CERN summer student program for the ROOT-TMVA project.

Keywords

CERN Summer Student; Decision Trees; XGBoost; TMVA (Toolkit for MultiVariate Analysis), ROOT, CLING, C++

1 Introduction

Decision trees have existed from the early stages of Artificial Intelligence, but have a recent explosion in popularity due to the numerous data science competitions won with them, as can be witnessed on Kaggle¹. There are many different algorithms that uses variants of decisions tree, but the one that is the most discussed these day is XGBoost [1] which has been the dominant algorithm for building accurate models on tabular or structured data.

What all successful Decision Trees implementation have in common is the tendency to prefer many shallow trees (called weak learners) to few deep trees (called strong learners). Many weak learners can be aggregated together to form one strong learner. This method allows also to mitigate overfitting, which is very common in Decision Trees based models. For more details about the different flavors of decision trees algorithms, we refer an interested reader to [2]

There exist several libraries that focus on the training of decision trees, e.g. [1,2], but very few that focus on the inference of Decision Trees. This is of particular interest at CERN since there are very large volumes of data that need to be processed on the go. Moreover, for application such as triggering and reconstruction, the data need to be processed event-by-event, thus loosing the speed-up we could get from batching.

In this work ² we focus on the inference of Forests of decision trees, both event-by-event and batched. In particular we infer on Boosted Decision Trained with XGBoost. The code, written in C++,

*CERN

[†]Karlsruhe Institute of Technology (KIT)

[‡]Carnegie Mellon University

[§]Luleå University of Technology

¹<https://www.kaggle.com/>

²The developped code is at <https://github.com/LucaZampieri/root/tree/master/tmva/tmva/src/BDT> and should soon be integrated in the master branch of ROOT: <https://github.com/root-project/root>

can then be used both as a standalone or shall be integrated in ROOT-TMVA. We evaluate the performances of our algorithm against XGBoost C API.

Furthermore, we present 4 different inference strategies corresponding to 2 tree representations:

1. Branched: using pointers
2. BranchedJIT: Just In Time (JIT) compiled branched
3. Branchless: using topological ordering
4. BranchlessJIT: Just In Time (JIT) compiled branchless

JIT, does some compilation at run-time, thus having at disposal the optimization of a compiler to the price of a larger setup time. However, in High Energy Physics, we can easily sacrifice setup-time. JIT is thus translated into a faster application-phase. We give a taste of it in Figure 1.

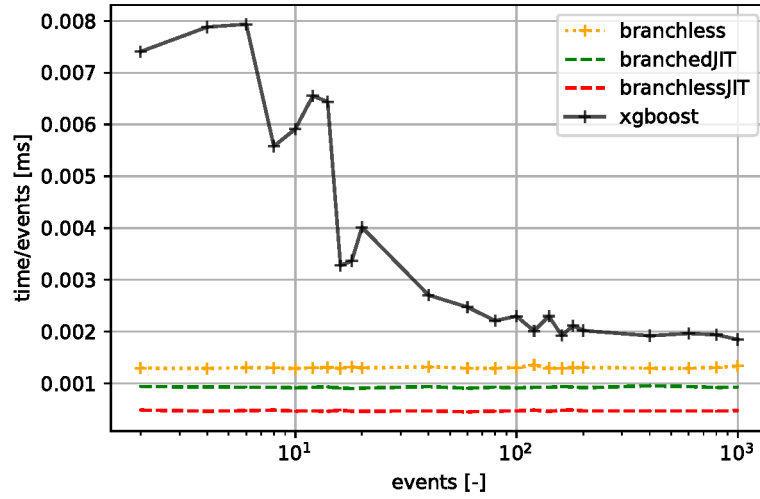


Fig. 1: XGBoost inference compared to our Branchless, BranchedJIT and BranchlessJIT version.

2 Tree representation

We depict in figure 2 a binary decision tree. We will use this decision tree as an example in the rest of this report. Note that this decision tree is **not** full.

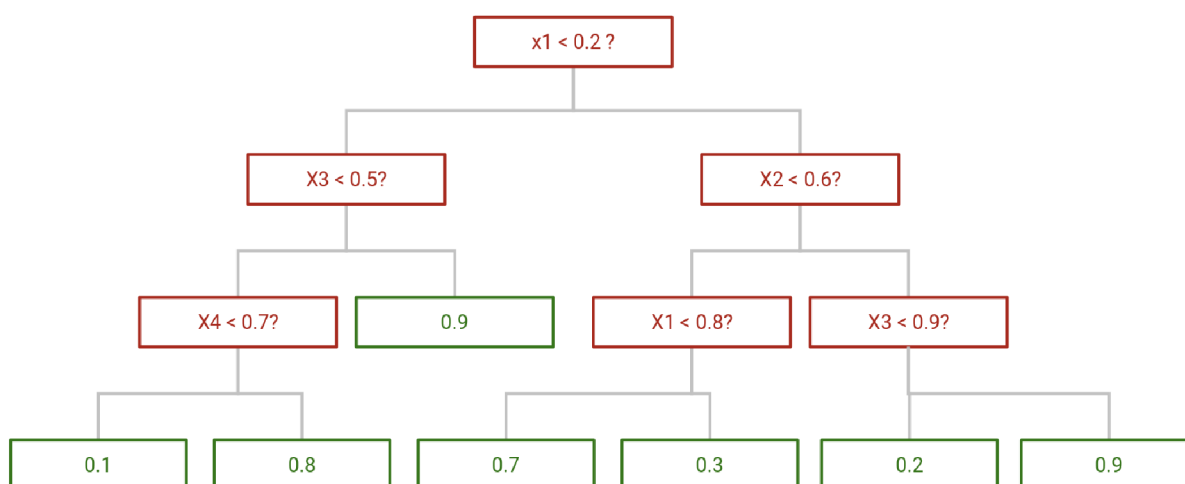


Fig. 2: Illustration of a binary decision tree, with depth of 4 and 4 variables (X1, X2, X3, X4). Leaf nodes are shown in green.

The maximal number of elements in a Tree increases exponentially with respect to the depth: $\text{size} = 2^{\text{depth}} - 1$. For inference we can go down the subsequent comparison statements until we reach a leaf node which gives us a score. For each tree in the forest we do similarly, then we aggregate the results and can perform either classification or regression.

2.1 Branched (Pointers)

The branched version is the most intuitive. It consist of creating Node objects that have pointers to the addresses of the two subsequent nodes and can redirect to one or the other node depending on the result of the comparison. A simple schematic representation is shown in Figure 3

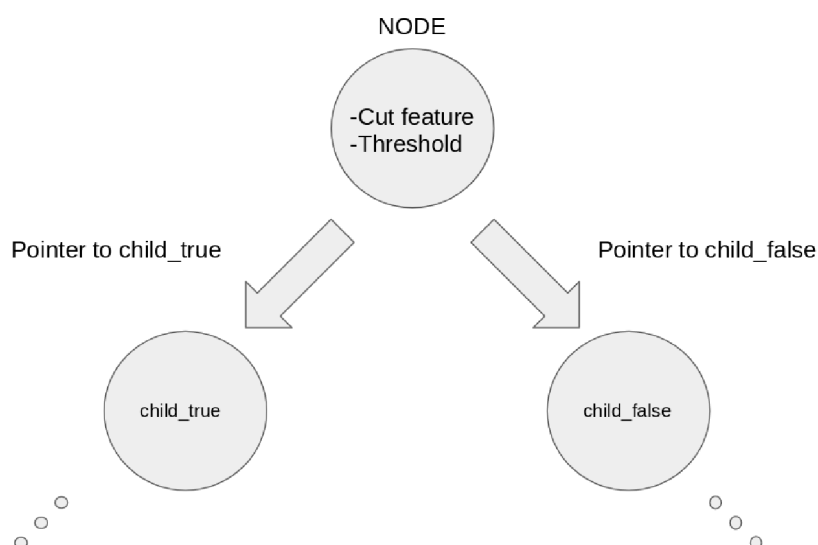


Fig. 3: Schematic visualization of branched tree.

Inference can then be done recursively through the node as depicted in Listing 2.1

```
if (event[split_variable] <= this->split_threshold) {
    if (child_true) child_true->inference(event);
    else return this->leaf_true;
}
else {
    if (child_false) child_false->inference(event);
    else return this->leaf_false;
}
```

2.2 BranchedJIT

Thanks to CLING and its Just In Time compilation, we can generate the code corresponding to the tree, as shown in Listing 2.2

```
if (event[2] < 2.290259) {
    if (event[2] < 1.974349) {
        if (event[1] < 1.840858) result += -0.001288;
        else result += -0.030341;
    } else {
        if (event[3] < 0.405312) result += 0.087805;
        else result += -0.016667;
    }
} else {
    if (event[0] < -0.536557) {
        if (event[2] < 2.361324) result += 0.111111;
        else result += -0.051852;
    } else {
        if (event[3] < -0.896920) result += 0.013333;
        else result += -0.127273;
    }
} // end here
```

2.3 Branchless (topological ordering)

Another possibility is to *unroll* the tree into a long array and exploit the topological ordering for the inference. We can keep one array for the features on which we apply the cut and one for the thresholds. Such an unrolling is shown in Figure 4. Note how we see in yellow the missing leaves due to the sparsity of the original tree (see figure 2).

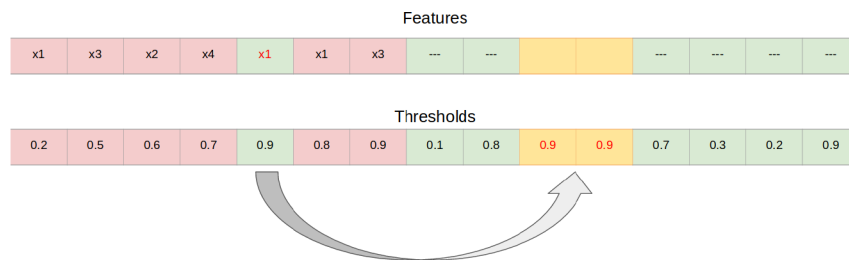


Fig. 4: Schematic visualization of branchless tree. In red we see artificially added value to render the tree full.

Now thanks to the natural ordering of the array we can iterate through the tree without if-then-else statements, as shown in Listing 2.3. Now, the operation `(event[features[index]] > thresholds[index])` is just a mathematical operation and cheaper than branching with if-statements. Note that this method is not possible for non-full trees, which is why we perform a "filling" of the tree as shown in Figure

4. We can expect thus the performances of this method to decrease when the trees are highly sparse. Thankfully, in general, the trees tends to be very shallow and thus poorly sparse.

```
index=0;
  for (iLevel = 0; iLevel < tree_depth; ++iLevel) {
    index = 2 * index + 1 + (event[features[index]] > thresholds[index]);
  }
return thresholds[index];
```

2.4 BranchlessJIT

Again, with JIT we can do some optimization such as unrolling the loop since we now at "JIT time" the depth of the trees, as shown in Listing 2.4

```
index = 0;
index = 2*index + 1 + (event[features[0+index]] > thresholds[0+index]);
index = 2*index + 1 + (event[features[0+index]] > thresholds[0+index]);
index = 2*index + 1 + (event[features[0+index]] > thresholds[0+index]);
result += thresholds[0+index];
```

3 Results and improvements

In this section, if not otherwise specified, we assume:

- # events: 10^5
- # features: 5
- maximum depth: 3
- # trees: 200

As anticipated in Figure 1, we can see our event-by-event implementations against the batched XGBoost. As we see, all our implementation outperform XGBoost, be it for a large number (3x) of events or event-by-event (15x).

How can we improve such a simple algorithm? In the next subsection we will do our best to give an answer to this questions, but meanwhile it can be useful to give a look to the cost of some common operations here: <http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/#rabbitref-NoBugs>

3.1 Branch Predictions

Modern processors do dynamic branch prediction, i.e. they will, at run time, predict the outcome of an if statement. Processors have many way to make such an educated guess. In the case of a *correct* guess the cost of the operation is around 1-2 CPU cycles, and in case of a *misprediction* it is of around 10-20 cycles³.

Now, How can we avoid branch mispredictions? We have to make it easier for the processor. One way of achieving this is by a simple ordering of the trees according to features and thresholds of the first node of the tree. In this way, we will group the true and wrong prediction of the first if, and it will be much easier to guess the outcome based on the previous outcomes. A 6x speedup has been reported for a simple array summation based on an if statement, just by previously ordering the array⁴.

How many branch mispredictions can we avoid this way? In 5 is a comparison of the two same forests, with vanilla ordering (not ordered) and one with ordered trees.

³According to wikipedia: https://en.wikipedia.org/wiki/Branch_predictor

⁴<https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array>

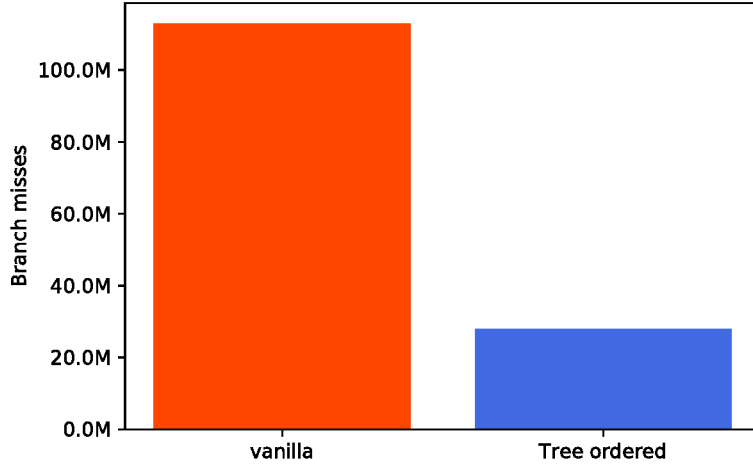


Fig. 5: Number of branch misprediction, for non-ordered trees [left] and ordered trees [right]

As we can see, the number of branch misses has been decrease by a factor of almost 4: from 113M to 29M. However, this does not guarantee an improvement in performances, so lets check:



Fig. 6: Performance gained by the ordering of trees (dotted lines w.r.t. dashed lines). Here we only show the inference done event-by-event, except XGBoost that can use a batch method.

As we see in the figure, ordering the trees thus decreasing the number of branch missed improves greatly the performances in our case.

3.2 Batch inference

When we have a bunch data, we have many ways to iterate though it make use of the fact that we have many events before looping, creating thus a "batch" inference.

We have been inspired by loop blocking, see e.g. https://en.wikipedia.org/wiki/Loop_nest_optimization.

To illustrate the concept, in Listing 3.2 sample code is shown on how it would be implemented in practice.

```
// "Normal"
for (i=0; i<num_events; i++)
    for (auto tree: trees)
        tree.inference(events[i]);

// Loop blocking for JIT since the forest is Jitted together
for (i=0; i<num_events; i+=batch_size)
    for (j=0; j<batch_size; j++)
        jitted_forest.inference(events[i+j]);

// Full Loop Blocking
for (i=0; i<num_events; i+=batch_size)
    for (auto tree: trees)
        for (j=0; j<batch_size; j++)
            tree.inference(events[i+j]);
```

In Figure 7 is shown

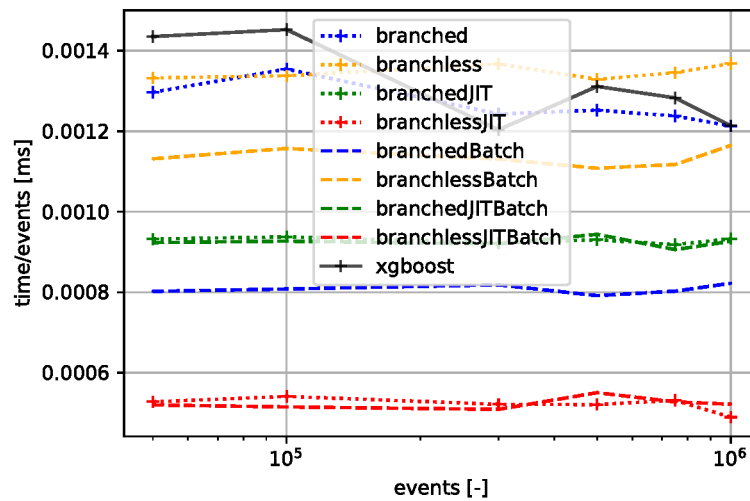


Fig. 7: Performance gain by using loop-blocking. Dotted lines [event-by-event] are higher, thus less performant than dashed lines [loop-blocked].

Another way to make use of the fact that we have the event before-hand is to fully loop over trees and then over events, allowing additional potential vectorization. Such a inference is shown in Listing 3.2, making use of storage of the prediction also for the accumulation.

```
void inference(const T *events_vector, const int rows, const int cols, T *preds)
{
    for (auto &tree : this->trees) {
        for (int j = 0; j < rows; j++) {
            preds[j] += tree.inference(events_vector + j * cols);
        }
    }
}
```

```

for (int j = 0; j < rows; j++) {
    preds[j] = this->objective_func(preds[j]);
}
};

```

3.3 Cleverer way to iterate trough the tree at JIT time

We thank Axel Naumann for the suggestion.

We could expect the time for inference to be linear wrt the depth of the tree. However, as depicted in Figure 8 for the branchless and JITTED version we witness a super-linear behavior. This is probably due to a decrease of the data locality.

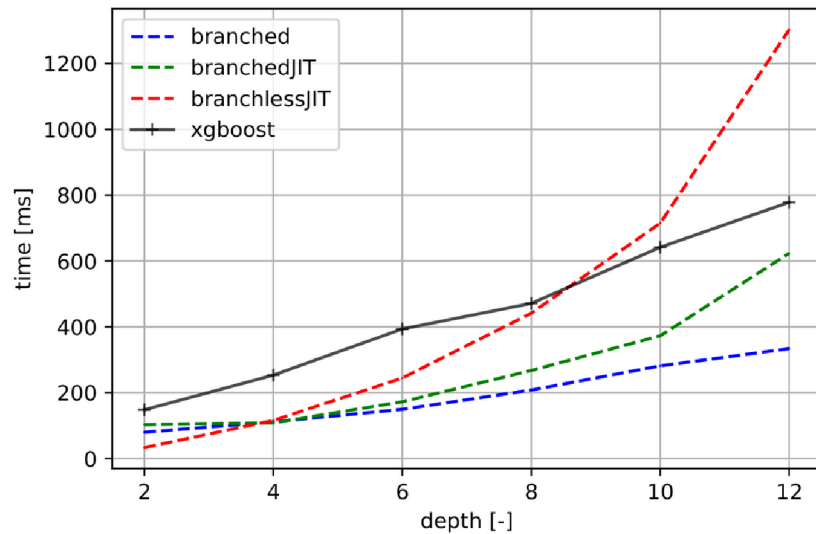


Fig. 8: Inference time as a function of the depth of the trees.

To improve the locality of the algorithm, we can re-order the trees as shown in Figure 9, which gives the arrays shown in Figure 10:

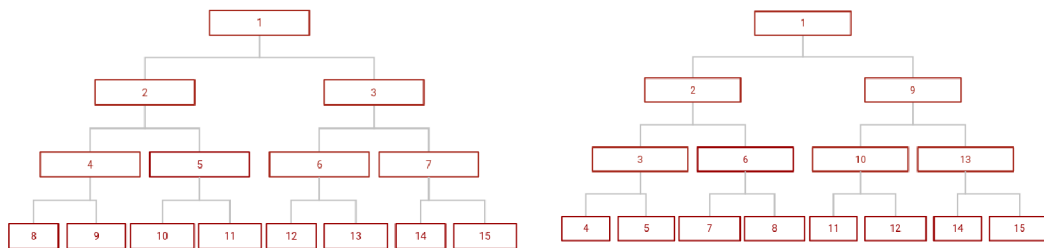


Fig. 9: Tree with nodes as they would be in an array topologically ordered[left] or transversely ordered[right].

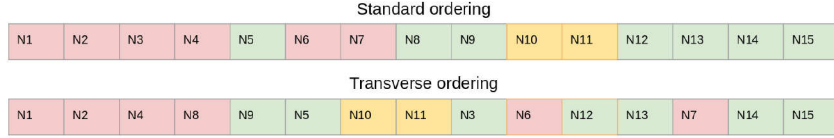


Fig. 10: Smarter ordering of an unrolled Tree. The idea is to start from the root node, order following the true statements, when we reach the bottom of the tree we go back to the last possible wrong statement re-iterate until filling completely the tree.

This leads to the inference shown in Listing 3.3. This shall improve performances as while the `if`-statement is true we will be contiguous in memory.

```
index = index + 1
+ std::pow(2, tree_depth - current_depth) - 1)
* (event[ this ->features[index]] > this ->thresholds[index]);
```

However, this implies the use of the power function that has very bad performances... **Or does it?** We can use the JIT to hard-code the power function, thus avoid the overhead of computing the powers. Already at a depth of 5 we witness an improvement of 40% wrt to standard topological ordering.

Conclusion

We implemented a fast inference engine for Decision Trees in C++, improving significantly the event-by-event inference (>15x) and the performances of batch inference, all using the tree already trained in XGBoost. The code has been the subject of a pull-request to root-master and will be integrated as soon as possible in the core of ROOT-TMVA. We had shown a use case in which JIT compilation can be used in C++ to improve the efficiency of an algorithm. We look forward to feedback from users, to see how it can improve their inference time.

Acknowledgements

I wish to thank my supervisors Kim Albertsson, Sitong An, Stefan Wunsch and Lorenzo Moneta for their warm welcome, their sympathy, talent and help. Thanks to them I really felt in-between friends. I also thank the whole ROOT team and SFT group for treating me as one of their own and for being so cheerful. I specially thank also Axel Naumann for the help he provided for Just In Time compilation with CLING, for maintaining CLING, a splendid tool. I thank Jesus Perales Hernandez and Emmanouil Michalainas for the discussions that led to improvements in our algorithms. At last but not at least, I thank my office mates: Manos, Jesus, Joana, Murat and Philipp for without them the Mondays would not have been that shiny.

References

- [1] T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’16, (New York, NY, USA), pp. 785–794, ACM, 2016.
- [2] T. Keck, “Fastbdt: A speed-optimized and cache-friendly implementation of stochastic gradient-boosted decision trees for multivariate classification,” *CoRR*, vol. abs/1609.06119, 2016.

Appendices

A Benchmark details

Google benchmark, release version, frequency scaling off, compiled with -O3
gcc version 7.4.0

Run on (4 X 3100 MHz CPU s)

CPU Caches:

L1 Data 32K (x2)

L1 Instruction 32K (x2)

L2 Unified 256K (x2)

L3 Unified 3072K (x1)

Load Average: 3.29, 3.17, 3.30