Fast Inference for Machine Learning in ROOT/TMVA

Kim Albertsson[1,2], *Sitong* An[1,3], *Lorenzo* Moneta[1], *Stefan* Wunsch[1,4] and *Luca* Zampieri[5]

[1]CERN
[2]Lulea University of Technology
[3]Carnegie Mellon University
[4]Karlsruhe Institute of Technology
[5]École polytechnique fédérale de Lausanne

Abstract. ROOT provides, through TMVA, machine learning tools for data analysis at HEP experiments and beyond. However, with the rapidly evolving ecosystem for machine learning, the focus of TMVA is shifting. We present the new developments and strategy of TMVA, which will allow the analyst to integrate seamlessly, and effectively, different workflows in the diversified machine-learning landscape. Focus is put on a fast machine learning inference system, which will enable analysts to deploy their machine learning models rapidly on large scale datasets. We present the technical details of a fast inference system for decision tree algorithms, included in the next ROOT release (6.20). We further present development status and proposal for a fast inference interface and code generator for ONNX-based Deep Learning models.

1 Introduction

The Toolkit for Multivariate Analysis (TMVA) [1] is part of the ROOT Data Analysis Framework [2] since 2005. It provides a one-stop solution for the development, deployment and validation of machine learning methods for data analysis in the High Energy Physics community, long before the gain in popularity in machine learning in the recent years. One of the most popular machine learning methods supported by TMVA is the Boosted Decision Trees (BDTs), which is widely used in the community for a myriad of analyses [3-6]. There have been multiple efforts to implement efficient BDT inference at trigger level, such as by CMS at L1 Muon Endcap Trigger [7], Track Trigger [8], and the hls4ml group [9].

With the rapidly evolving machine learning landscape and the rise of popular machine learning frameworks supported by large technology companies, the focus of TMVA is shifting. Specifically, we aim to provide, within TMVA, methods that allow fast and convenient deployment of popular machine learning methods in the production environment.

This proceeding presents our recent development in fast inference engine for decision trees, discusses its technical features as well as a plan for supporting similar fast inference methods for neural network models for deep learning.

2 Fast Inference Engine for Decision Trees

2.1 Introduction

Decision tree-based algorithms have been widely popular both within the high energy physics community and in the data science industry for many years. This is a family of machine learning algorithms united by their use of tree-like model with a "test" at a certain "cut point" at each of the node of the tree. Some of the most popular decision tree algorithms include Boosted Decision Trees (BDTs) and XGBoost [10]. They use different strategies during training of the algorithms, but for deployment the structure of the model and the logic of the inference is the same. Their deployment thus can be done via a single inference engine.

The ability to deploy such algorithms in a fast and lightweight manner is of particular importance for applications in high energy physics. For example, low-latency inference is critical for some use cases applying decision-tree algorithms, such as BDT-driven high-level triggers [11]. Furthermore, in high energy physics the emphasis is usually put on event-loop inference rather than batch inference. This means that many of the existing inference engines for decision-tree based algorithms, developed by the data science industry and focusing on batch inference, might not suit the purpose of high energy physics community.

In the following sections we present the techniques used and the results achieved for a fast inference engine for decision trees within TMVA. This work was initiated as a CERN summer student project by Luca Zampieri [12].

2.2 Just-in-time (JIT) Compilation

Cling [13], the interactive compiler in ROOT, allows the use of just-in-time compiler for compilation. In practice, this means that we can compile hard-coded evaluation logic parsed from the decision-tree models. See figure 1 for a code snippet which demonstrates how this is done in the TMVA fast decision tree inference engine. Normally without the support of a just-in-time compiler, we have to code the generic logic for the inference of a decision-tree based algorithm for the C++ static compiler. However, with Cling, it is possible to read the model file, parse the tree structure, and compile and run the generated code snippet for inference at run time.

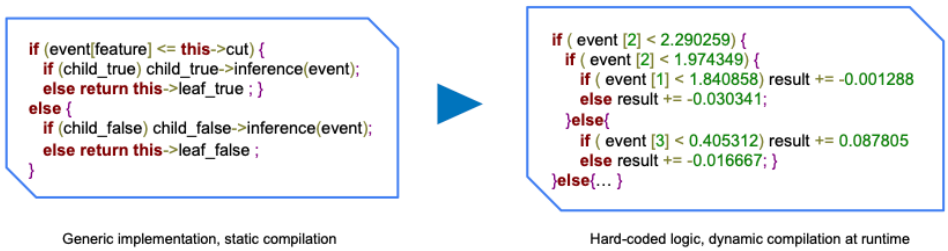

Fig. 1. Code snippet demonstrating the use of just-in-time compilation for the TMVA fast inference engine for decision-tree based algorithm. Conventional generic implementation is presented on the left, and the code snippet generated and compiled at runtime after reading the model file and parsing the tree is presented on the right.

The advantage of using the Cling just-in-time compiler in this case is that it allows us to exploit the powerful C++ compiler optimization dynamically, that is, at run time. With this technique, we are able to demonstrate significant speedup of the inference. The results are presented in figure 3, in the next section.

2.3 "Branchless" representation of trees

Another technique that we adopted is the "branchless" representation of trees [14]. Here, we "unroll" the tree from linked nodes into a long sequential array, using the array representation of binary tree in classical computer science. This of course assumes that the tree we are unrolling is a full binary tree, which is certainly not always true for the models trained with decision-tree based algorithms. To address this, we fill in the missing values in sparse trees to create full binary trees. See figure 2 for an example of this filling procedure, as well as a code snippet demonstrating how such an array can be traversed.

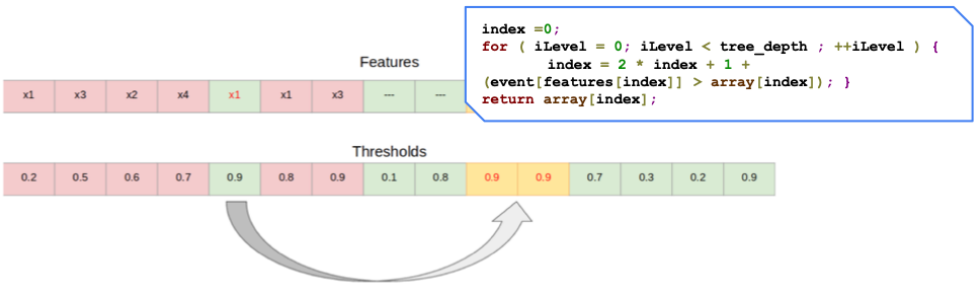

Fig. 2. Demonstration of the process that fills a sparse tree into a full tree and its corresponding array representation. On the top right is the code snippet for traversing the array representation of the tree.

While this seems to complicate the representation of the model, it has the advantage that tree traversal is now a mathematical operation (see the code snippet in figure 2), which is cheaper than an if operation. Together with JITting, we discovered that these two techniques combined gives rise to significant speedup in inference (see figure 3). While no conclusive study has been done on the exact cause of this speedup, we hypothesise that this might come from the better branch prediction of CPUs on mathematical operations.

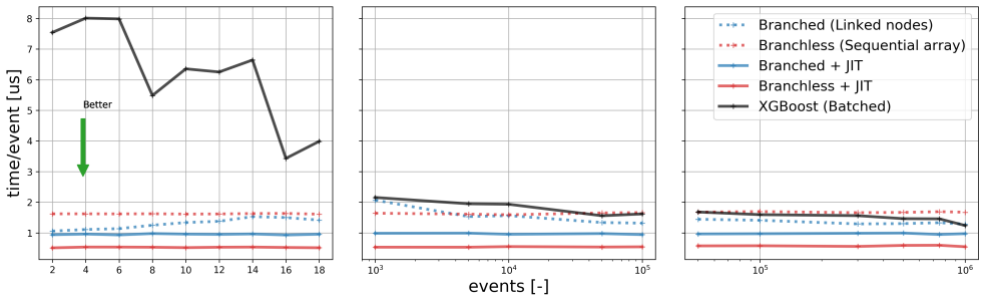

Fig. 3. Speedup in inference by adopting the JIT and the branchless representation techniques, in comparison to vanilla XGBoost implementation.

The branchless implementation assumes shallow, nearly full trees. If the tree is very deep and sparse, this implementation might not perform as well as the conventional "branched" (linked nodes) implementation, as it cannot stop early in case of a leaf at a very shallow level, having to traverse each level of the tree regardless of the model and the data. Figure 4 demonstrates this effect. Fortunately, most decision-tree based machine learning algorithms produce shallow, nearly full trees that branchless implementation works well on. This is partly due to the fact that with a deep tree the machine learning algorithm is usually more prone to overfitting. We will also integrate the branched (linked nodes) implementation for deep trees in the future so that the inference engine in TMVA will perform optimally regardless of the depth of the tree.

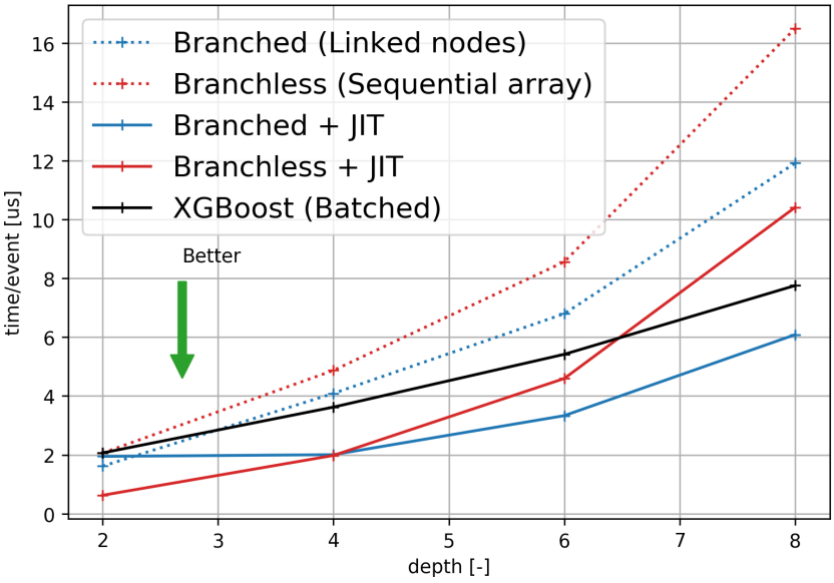

Fig. 4. Performance worsens for branchless implementation for greater depth of the tree, both for the jitted and non-jitted versions.

2.4 Tree ordering

Inspired by the exploitation of branch prediction in the branchless implementation, we also investigated the effect of changing the ordering of the trees. We ordered the trees such that they are evaluated in the order of feature and cut value of the root node. The rationale is that it could improve dynamic branch prediction and reduce branch misses. Again, this technique achieved significant speedup when coupled with JITting, as demonstrated in figure 5.

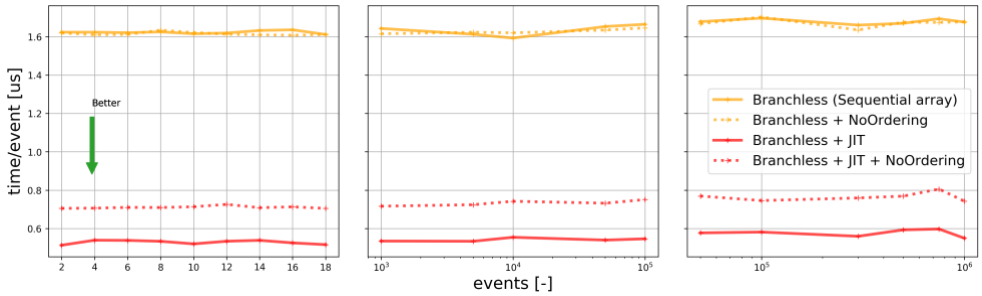

Fig. 5. Speedup in inference by adopting the JIT and the tree ordering techniques.

2.5 Loop nest optimization

Loop nest optimization is another classical optimization technique usually used in compiler design. Here, we chunk iteration space (over trees and events) into small blocks. The idea behind this technique is that it might improve data and instruction locality, thus potentially reducing cache misses. See figure 5 for an example code snippet that demonstrates how loop nest optimization is implemented in our decision tree inference engine, and figure 6 for its speedup with and without JITting.

```
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]);
```

Fig. 6. Speedup in inference by adopting the JIT and the tree ordering techniques, in comparison to vanilla XGBoost implementation.

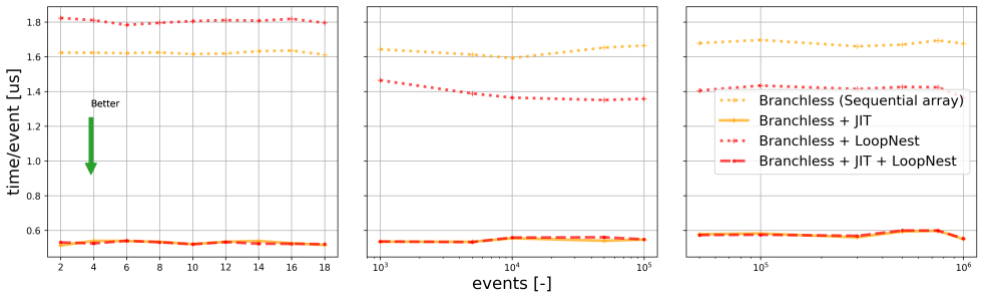

Fig. 6. Speedup in inference by adopting the JIT and the loop nest optimization.

Interestingly, here we find that if JITting is not used, then loop nest optimization can achieve a speedup. However, with JITting enabled, whether loop nest optimization is used does not make a difference on the inference speed, regardless of the batch size. This demonstrates that JITting is an effective way to exploit potential of compiler optimization at run time that is otherwise not possible. However, for implementations where JITting is not available, we demonstrated that loop nest optimization is still helpful in improving the inference speed.

2.6 Outlook on potential application to HLT

A preliminary study into the memory footprint of the Branchless method, running the inference of a 100-tree 3-depth XGBoost-trained model on the complete toy Iris dataset from scikit learn, suggested a result of 4.58 Mb. For future work, a more comprehensive study into the memory footprint from both the Python and C++ interface and for other inference implementations pending their integration into TMVA is warranted. A comparison study between current implementations used at LHC experiments and the methods reported in this proceeding should also be done once this integration is complete, so as to give better insights into the potential application of these methods to HLT.

3 Inference of ONNX Deep Learning Models

With the rapidly evolving landscape of modern machine learning software tools, TMVA is focusing on not only on supporting the inference of traditional machine learning algorithms like BDTs, but also deep learning models based on neural networks. Here, we outline a future proposal for how TMVA plans to achieve this.

ONNX [15] is an open format for deep learning models. It aims to create a set of open, future-proof rules and standards for the definition of deep learning models and interoperability of these models between different deep learning frameworks. It currently supports most of the popular deep learning operators and layers, and there are convertors available for the conversion of models files produced from major deep learning framework into ONNX format.

In parallel, an open source inference engine based on ONNX standard, named ONNX Runtime, is developed. It is supported by the Microsoft open source and in fast

development. Highly optimised for low-latency inference, it supports multiple backends and optimization methods.

While there has already been some success at integrating ONNX Runtime into the analysis frameworks of some of the large experiments at CERN, the goal of TMVA in this area is not to directly compete with it. Instead, we propose an inference engine based on the code generator model. It would take ONNX models as input and compiles it into a piece of static C++ code providing functional APIs for the inference of the model. See figure 7 for a demonstration of this process. The user would train their deep learning models in a Python-based, GPU-dominated environment, convert the trained model into ONNX Models. As the next step, TMVA will convert the ONNX Model into a snippet of C++ code, ready for deployment in the C++ production environment.

Fig. 7. Proposed work process for deploying deep learning models in C++ analysis frameworks with TMVA

Compared with the use of ONNX Runtime, this approach has the advantage of having minimal external dependency. In cases where the neural network is only a relatively small and simple part of the entire analysis workflow, it gives users the option to keep the inference code in-house instead of adding external dependency.

Currently, we have completed the ONNX operator-based infrastructure. We are able to explore and manipulate ONNX models, paving the road for potential future customized optimization of the models specifically for high energy physics purposes. We plan to further develop this with the code generation component and trial it in a few months. This is an ongoing work, and the ROOT/TMVA team warmly welcomes any request, suggestion, recommendation or complaint over this topic.

Luca Zampieri is a CERN summer student in 2019 working on the decision tree inference engine project under the supervision of Kim Albertsson, Sitong An, Lorenzo Moneta and Stefan Wunsch.
Sitong An gratefully acknowledges the support of the Marie Sklodowska-Curie Innovative Training Network Fellowship of the European Commission Horizon 2020 Programme, under contract number 765710 INSIGHTS.

References

1. A. Hoecker, P. Speckmayer, J. Stelzer, J. Therhaag, E. von Toerne, H. Voss, M. Backes, T. Carli, O. Cohen, A. Christov et al., TMVA - Toolkit for Multivariate Data Analysis (2007), physics/0703039
2. R. Brun, F. Rademakers, ROOT - An object oriented data analysis framework (1997)
3. V. M. Abazov et al. (The D0 Collaboration), Evidence for production of single top quarks, Phys. Rev. D **78**, 012005 (2008)
4. G. Aad et al. (ATLAS Collaboration), Phys. Lett. B **717** 89-108 (2012)
5. The CMS Collaboration, CMS-PAS-HIG-13-001 (2013)
6. The ATLAS Collaboration, ATL-PHYS-PUB-2015-022 (2015)
7. D. Acosta et al. (CMS Collaboration), J.Phys.Conf.Ser. 1085 (2018) 042042, Boosted Decision Trees in the Level-1 Muon Endcap Trigger at CMS
8. S. Summers et al., JINST 15 P05026 (2020), Fast inference of Boosted Decision Trees in FPGAs for particle physics
9. J. Duarte et al., JINST 13 P07027 (2018), Fast inference of deep neural networks in FPGAs for particle physics
10. T. Chen, C, Guestrin, XGBoost: A Scalable Tree Boosting System, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16, (2016)
11. V.V.Gligorov, M. Williams, JINST **8** P02013 (2013)
12. L. Zampieri, CERN-STUDENTS-Note-2019-183 (2019)
13. V. Vasilev, Ph. Canal, A Naumann, P Russo, Cling – The New Interactive Interpreter for ROOT 6, J. Phys.: Conf. Ser., **396** 052071 (2012)
14. T. Keck, FastBDT: A speed-optimized and cache-friendly implementation of stochastic gradient-boosted decision trees for multivariate classification, 1609.06119 [cs.LG] (2016)
15. B. Junjie, L. Fang, Z. Ke et al, ONNX: Opern Neural Network Exchange (2019)