



farheenasif@email.arizona.edu

25th Oct 2021

Ultra-fast Machine Learning Inference for triggering at CMS Experiment

Farheen Asif

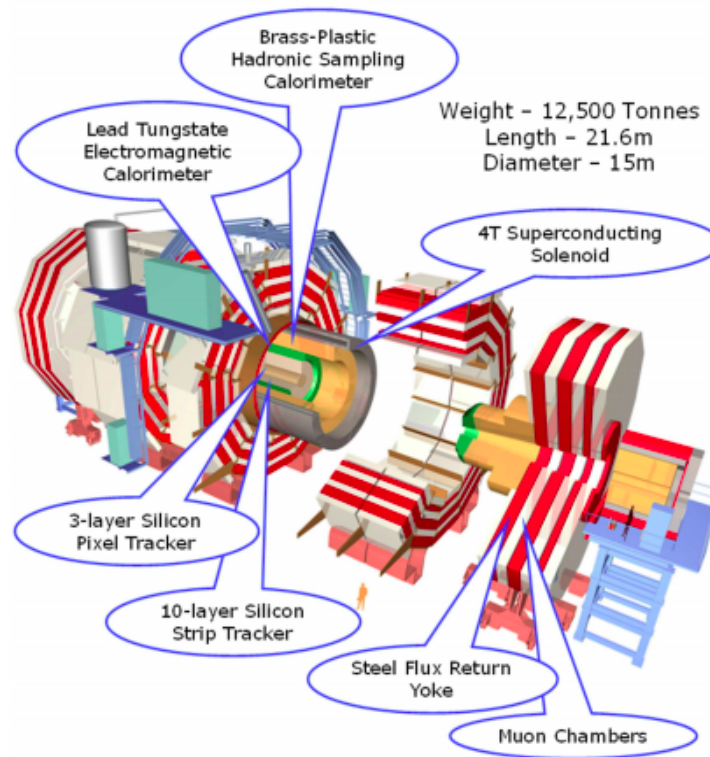
Supervised by: Sioni Paris Summers

Introduction:

I have been assigned a project related to the CMS detector at the Large Hadron Collider. Before I move towards the actual details and goals for ultra-fast machine learning inference at CMS, I will first describe the CMS detector, trigger system, triggering at CMS, the need for Machine Learning and FPGAs, high level synthesis and boosted decision trees briefly. The readers of this report are expected to have a little knowledge about these concepts before.

LHC and the CMS Experiment:

The world's largest and most powerful particle accelerator is the Large Hadron Collider. It is basically a 27 km in circumference at CERN on the border of France and Switzerland near Geneva. Protons are accelerated close to the speed of light, and they collide at 14 TeV centre of mass energy in order to search for new fundamental physics of the universe. In LHC, collisions happen at 4 points where there are detectors. One of these particle detectors is the Compact Muon Solenoid (CMS), a general purpose detector. The CMS Experiment corresponds to the triggering and collisions at this particle detector. The CMS detector is built around a huge solenoid magnet which is basically a cylindrical coil of superconducting cable generating a field of 4 tesla, about 100,000 times the magnetic field of the Earth. There are specialised sub-detectors to detect different types of signatures. It is basically 21 metres long, 15 metres in width and height. The CMS experiment is one of the largest international scientific collaborations in history, involving 5000 collaborations of physicists, technical staff and students.



Trigger System:

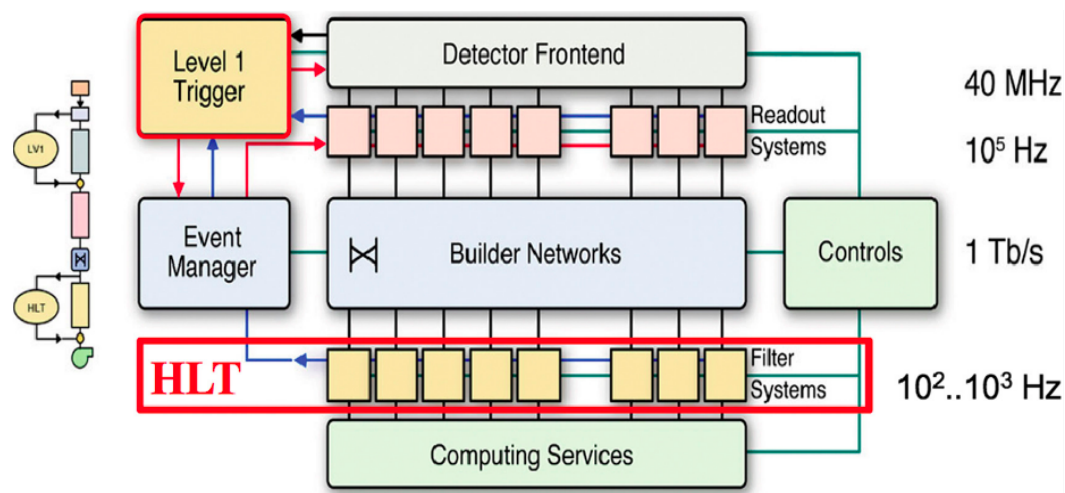
At LHC, protons collide at 40MHz of frequency and there are extreme data rates of 100TB/s. Due to the real world limitations of data storage capacity, computing power and data rates, trigger systems are necessary. Therefore, these trigger systems use some criteria to decide rapidly which specific events in the particle detector are to be recorded and kept while discarding others. So, when we say triggering at any detector, we mean filtering of the events to reduce the data rates to a manageable level. This system is specifically important because at this time approximately per event there are 40 collisions in LHC and by 2026 the number of collisions will increase up to 200 per event. Furthermore, there is a need for sophisticated techniques in order to maintain sensitivity in an increasingly complex collision environment because if any event remains untriggered, it is lost forever.

Triggering at the CMS:

Extreme data rates are not reduced at one stage only in the CMS detector, there is a multi-stage system for this purpose including L1 trigger, high level trigger etc. Level 1 is an extremely fast

and wholly automatic process. In this step we select the best 100 thousand events each second from the billions produced. The next is the higher level trigger in which the information from different parts of the detector is gathered and synchronised to recreate the entire event and send it to a computing farm for detailed analysis of the full event.

The PCs in the computing farm run very complex physics tests and they select 100 events per second and the remaining others are thrown out. Only those events are left which would tell us something new. There is a custom hardware for these triggers and they have terabytes of absorbing capacity. The L1 trigger environment is too extreme for conventional PCs therefore, typically field programmable arrays and many high speed optical links are used. The main task of the trigger is to choose events and therefore the decisions are to be made instantly in microseconds. Latencies are limited by buffering capacity on-detector.

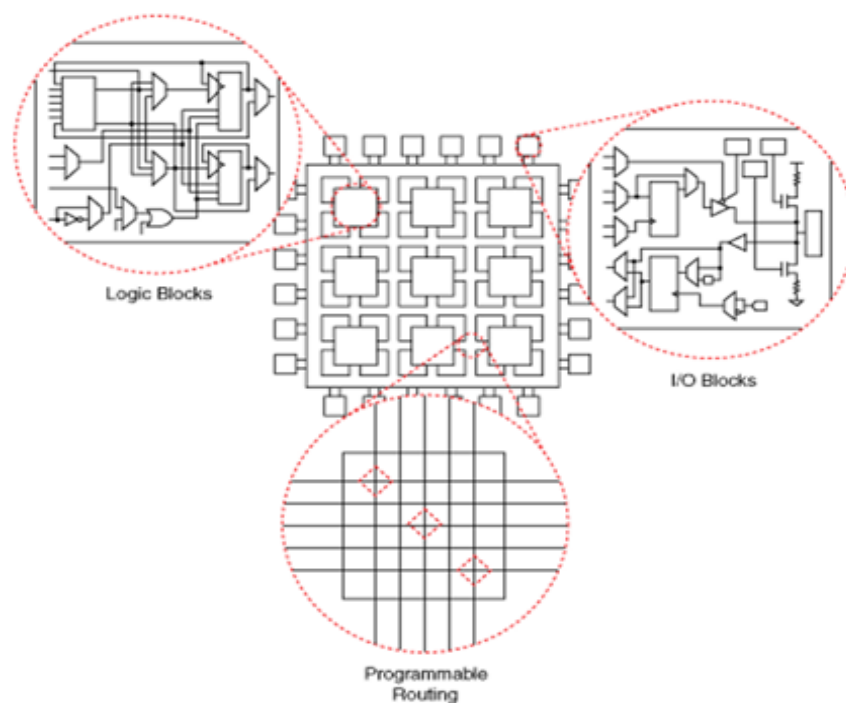


FPGAs:

Field programmable gate arrays are the integrated circuits. It has a programmable hardware fabric. Unlike graphics processing units (GPUs) or ASICs, the circuitry inside an FPGA chip is not hard etched, thus, it can be reprogrammed as needed. This capability makes FPGAs an best substitute of the ASICs, because they require long development time and a significant investment to design and fabricate.¹

¹ <https://www.intel.com/content/www/us/en/artificial-intelligence/programmable/fpga-gpu.html>

An FPGA is basically an ocean of gates with programmable wiring. It is an integrated circuit that consists of internal hardware blocks with user-programmable interconnects to customize operation for a specific application. The interconnection can quickly be reprogrammed, thus helping an FPGA to adapt changes to a design or even support a new application till the end time of the FPGA. It consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. The interface between the FPGA and external devices is Input/output (I/O) block.

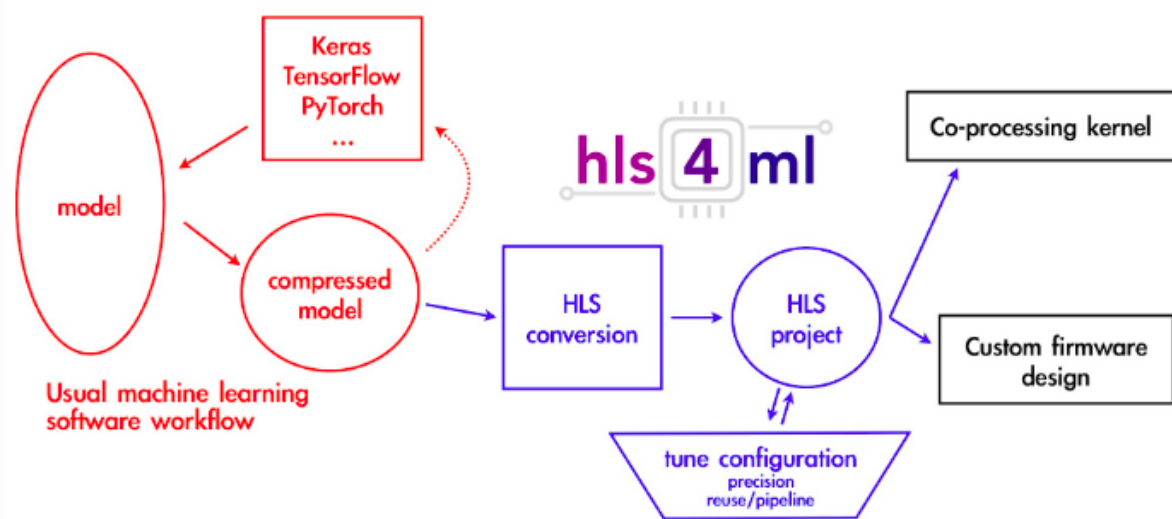


Machine Learning:

Machine Learning is a branch of Artificial intelligence which uses data and algorithms to copy the learning behavior of human beings and improves the accuracy gradually with the increase in learning rate. Machine Learning is used for the enhancement of signal and background discrimination. Machine Learning methods are often employed in offline analysis or for the trigger tasks with longer latency. These methods can help us to improve trigger selection and thus can be deployed on FPGA for fast inference.

High Level Synthesis for Machine learning:

hls4ml² is a python package to facilitate the implementation of machine learning models on edge devices such as FPGAs. Basically it is a firmware implementation of machine learning algorithms using high level synthesis language (HLS). The library translates the traditional open-source machine learning package models into HLS that can be configured for our use case. It creates a low-latency and optimized firmware implementation of machine learning algorithms using high level synthesis language (HLS). The workflow starts from creating a python-based machine learning model, which is transformed into HLS firmware. The model further goes through pruning and quantization for better CPU utilization. It can read input with standard ML libraries like Keras, PyTorch and onnx, and uses Xilinx HLS software. It comes with the implementation of common ingredients like layer types, activation functions and novel ingredients for fast, efficient inference like binary, ternary neural networks and network optimizations.

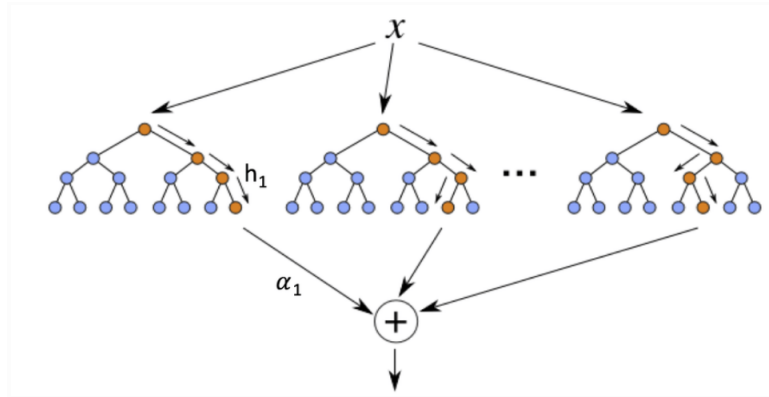


Boosted Decision Trees and conifer:

Boosted Decision Trees (BDTs) are one type of ML algorithm that have been around in particle physics for years. Boosting means each tree is dependent on the prior trees. They give good performances with low latency and less resources. In future BDTs would be used in LHC for

² <https://fastmachinelearning.org/hls4ml/index.html>

triggering soon. Therefore, the conifer package was recently created to support BDTs, similar to hls4ml for neural networks. For future use of BDTs, the conifer should be ready. Trained BDTs using scikit-learn, xgboost, TMVA are read in conifer to an internal representation of python dictionary with arrays of values and have backends that can write out an FPGA project so that user can deploy.



Existing support in conifer:

In the conifer package, there is an existing support for the scikit learn library for the Boosted Trees implementation and scikit learn converter for the conversion of python code and trees representation into the FPGA firmware for low latency inference. Currently, sklearn, xgboost and TMVA models are supported and FPGA firmware in XILINX high level synthesis or in VHDL can be produced.

INTERNSHIP TO-DO LIST

New Additions in the conifer package

As there is already an existing support of three ML libraries but a few other important Machine Learning libraries like ONNX, LightGBM, CatBoost and Google Yggdrasil are to be added at the frontend of conifer. Among these four, the most important is to start with the ONNX because LightGBM & CatBoost support can be added then using free onnxmltools.

Main Task

So the first and the main task is to write a converter that converts the onnx representation of BDTs to the FPGA firmware. Ultimately, I'll be using onnxmltools to check the translated BDTs in onnx graph structure. I need to figure out how the graph structure is related to the scikit learn or normal representation of trees and the graph structure is can be directly converted to the HLS or VHDL.

Bonus Tasks

There are two bonus tasks which are obviously dependent upon the completion of the main task to write a converter. Both are related to the testing of the converter. First on the server and the other one on the hardware directly.

For the bonus task 1, recently set up Jenkins server to run tests of conifer will be used. For the purpose, a gitlab tool for CI/CD will be used. They use pytest to run unit tests on the conversion, simulation and synthesis of some simple BDTs. Also, simple pytest code can be written and can be tested directly also.

For the bonus task 2, execution of the conifer models on Pynq is the goal. Inside CERN firewall, there is availability of Pynq-Z2 on CERN network and by following the Vivado GUI instructions of hls4ml pynq, we can implement conifer BDT.

Literature Review:

Overview of ONNX:

ONNX is the abbreviation of Open Neural Network Exchange and it is a format to represent learning models. ONNX is especially used in machine Learning models inference and with the help of specialized frameworks it enables fast inference. Now the benefit of adding ONNX support in the conifer package is clear because:

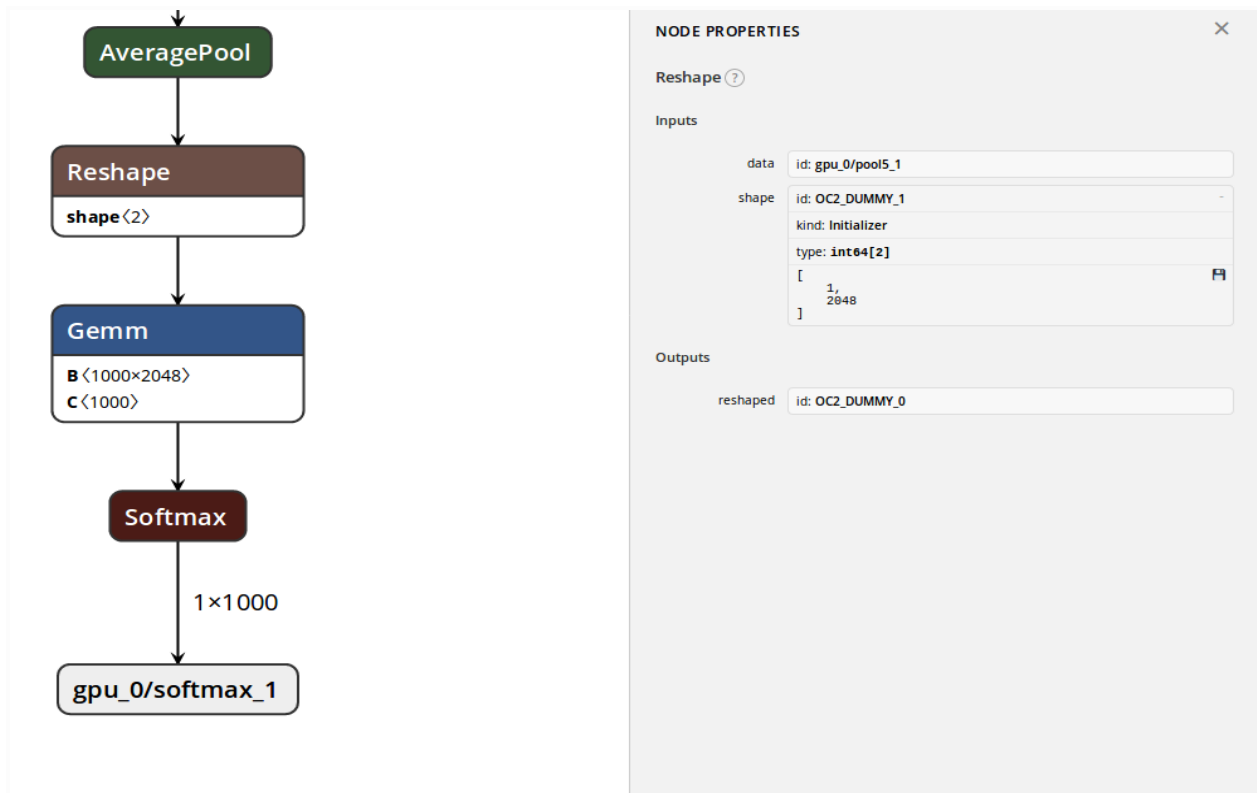
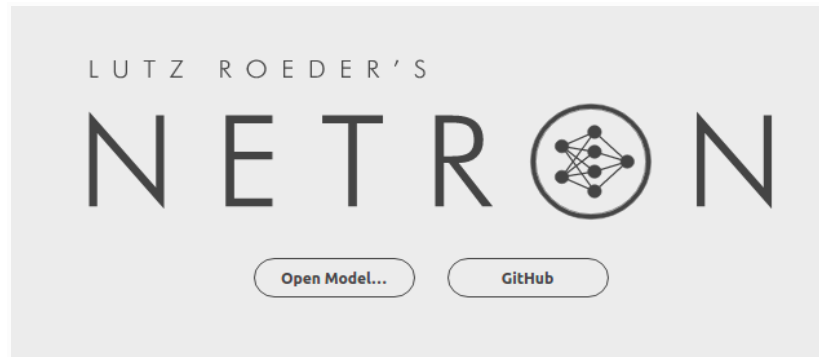
Fast inference through ONNX representation and Hardware Access: The fact is that different python frameworks of Machine Learning are used for training and are optimized for training purposes and not the fast inference and testing. Their conversion to ONNX with specialized inference engines help the developers to speed up the testing and inference part. In short, It makes it easier to access hardware optimizations to maximize performance across hardware.

Interoperability: The other benefit is that we do not have to care about the inference implications down the stream. And also, ONNX format is strictly well defined and will stay compatible in the future unlike other ML frameworks like PyTorch.

Various inference specific Software development kits like ONNX Runtime, ailia SDK and TensorRT can be used to perform inference tasks. The learning models and architectures can be imported from various Machine learning frameworks like Tensorflow/Keras and Pytorch. There are various operators pre-defined which are common across frameworks and a common file format for the building blocks of Machine Learning and Deep learning models.

ONNX Representation:

ONNX represents a computational graph. It is a serialization format for the machine learning model. More precisely it is a flat description of the learned model, its parameters and mathematical processing. It helps to describe the prediction function of the model for deep learning and Machine Learning models, through a list of mathematical functions. We can see the graph of the onnx representation through various tools. The models are saved with the extension of “.onnx”. A sample picture of a deep learning model in onnx format is shown below with the help of Netron tool.



The graph is a list of various nodes with nodes having individual properties. Some common properties of the nodes include the input type, output type, name of the node and various attributes inside the node. Each attribute inside the node has its name and values with the type of the values listed at the end of the node. Various other nodes like labels, probabilities and the model elements are also mentioned in different nodes. But to represent the total tree and model values, one node is used only.

ONNX Runtime:

It is an open source project which provides an accelerator for ML models with multi platforms support and for the integration of these models with the hardware specific libraries a flexible interface is provided. ONNX Runtime can be used with PyTorch, tensorflow, TFLite, scikit-learn and various other frameworks.

Implementation for the Boosted Decision Trees:

Comparison of scikit-learn and onnx:

The ONNX representation is even more flat i-e a simple list or array than the scikit-learn one. In the scikit-learn representation, there is an array of trees. In the ONNX one there is one array per attribute for the whole model, and the attribute contains the index of the whole model, flattened over tree nodes. For the scikit-learn representation, an array of trees exists in case of binary class and for multiple classes, multiple estimators exist and therefore multiple arrays exist. To access the information for each tree the thresholds, scores and the tree structure is used in the scikit-learn converter. In conifer we represent the BDT in this way too. The tree structure is flattened into arrays. Each array is one attribute of the tree, and each index represents one node. `children_left`, `children_right`, `feature`, `threshold` and `value` arrays exist in each of the tree estimators.

- The right and the left children arrays of the tree define the tree structure - the index of the left child of the root node is `children_left[0]`, and the index of its right child is `children_right[0]`.
- The feature array defines which input feature is used in that node. -2 means no feature is used: it is a leaf node.
- The threshold array are the thresholds that those features are compared against (again -2 means leaf node)
- The value array are the scores that we need to output to make the prediction.

Agenda:

The main agenda to approach this problem is to find one to one correspondence between scikit-learn representation and onnx representation of trees. Afterwards, implement the logic for binary class classification and then extend it further for multiclass problems. The correspondence will help us to write the code for the onnx converter.

1-1 Correspondence:

To find the correspondence between the two, first I need to analyze how a particular attribute of the scikit learn tree is shown in onnx graph node attribute. For the node representing the total model, The total attributes are listed below and their description is also given.

Attribute name	Type of Attribute	Description	Attribute name	Type of Attribute	Description
base_values	int	The consistent base value for all the nodes of all the trees.	nodes_hitrates	float	The probability of being true for the value of each node of the trees.
class_ids	int	Specific class id	nodes_missing_value_tracks_true	int	If any value of this attribute is 1 then it means that the value of the corresponding node is missing.
class_node_ids	int	Only those nodes ids are listed which are non leaf.	nodes_modes	string	It represents which node is the leaf node and which node is not the leaf node i-e somewhere inside the tree, a parent or root. In scikit-learn “-2” indicates that a particular node is the leaf of the tree. Here in onnx, if node_modes is “LEAF” then it is a leaf node, if it is “BRANCH_LEQ”, then it is not a leaf node.

class_treeids	int	Tree numbers written in ascending order, only wrt the non-leaf nodes	nodes_nodeids	int	For each tree, the node number is mentioned, so that if a person is looking at tree number 16, he/she can also know which node number it is. They are listed in ascending order i-e from 0th node id up to the last.
class_weights	float	Values of non leaf nodes of all the trees	nodes_treeids	int	Tree numbers written in the same sequence as the other attribute values to show that specific attribute value corresponds to this specific tree. Usually, tree ids and tree attributes are listed in ascending order i-e tree no 1 first and goes up to the last tree.
class_labels_int64s	int	All the integer values used as labels for different classes in the model. For the binary class model, -1 and 1 are the integers used. For multi-class they are 0,1,2 and so on up to n-1 classes.	nodes_tree_unodeids	int	Left children of all the trees in the scikit learn representation. If the value is 0, then it is a leaf node.
nodes_false_nodeids	int	Right children of all the trees in the scikit learn representation. If the value is 0, then it is a leaf node.	nodes_values	float	Values of each node of all the trees in the scikit learn representation. If the value is 0, then it is a leaf node.
nodes_featureids	int	It describes which input feature is used for each node of all the trees. If the value is 0,	post_transform	string	Output format, if binary then it is LOGISTIC.

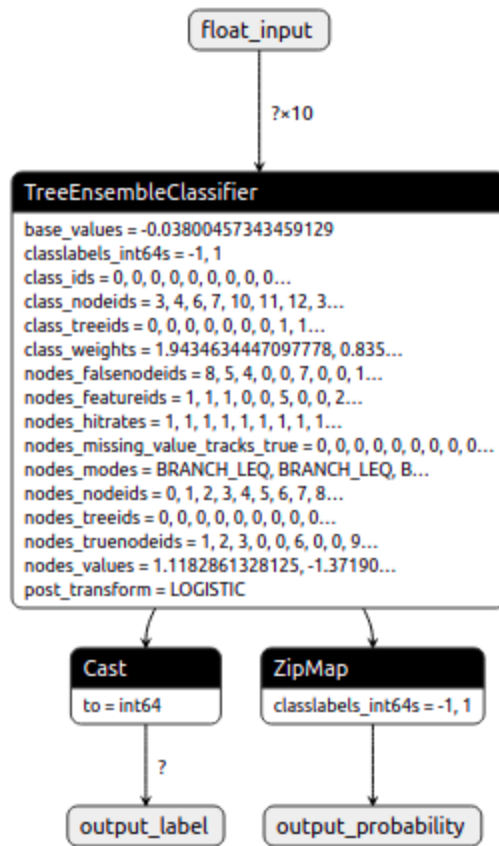
		then it is a leaf node.			
--	--	-------------------------	--	--	--

The onnx attributes, which represent scikit learn attributes and are useful for code computations are listed below:

Attribute names scikit learn	Attribute names ONNX
threshold	nodes_values
children_left	nodes_truenodeids
children_right	nodes_falsenodeids
feature	nodes_featureids
values(valid only for leaf nodes)	class_weights

Binary class Implementation:

The onnx graph for the binary class Boosted Decision Tree model is given below:



As we can see in this picture, for the two class classification only 1 base value and two class labels are given. At the same time, LOGISTIC activation function is for the post_transform.

base_values	-0.03800457343459129	+
classlabels_int64s	-1, 1	+
post_transform	LOGISTIC	

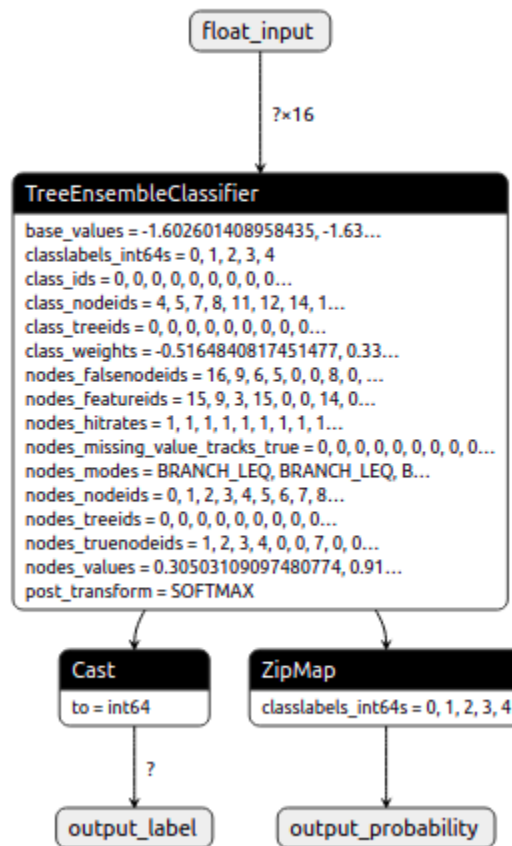
Sequential steps to write the onnx converter for the binary class are given below:

1. Write a convert_graph function to retrieve the attribute values of the graph and make a tree dictionary out of them to be used in the next bdt converter function.
2. Find the total number of trees by finding the length of the dictionary created.
3. Use the logarithmic function to find the maximum depth of the whole estimator.

4. Access the base values of the node from the graph to save as initial predictions.
5. Retrieve the features values of the nodes to find maximum features.
6. Make necessary type conversion for the compatibility with the converter.py file.
7. Extend the tree by adding parent and depth attributes in the tree dictionary.
8. Make an ensemble dictionary, in the convert_bdt function, to make a complete dictionary from all the estimators.

Extending the Implementation to multi-class classification

The onnx graph for the multi-class Boosted Decision Tree model is given below:



As we can see in this picture, for the 5 class classification, correspondingly 5 base values and five class labels are given. At the same time, SOFTMAX activation function is for the post_transform.

base_values	-1.602601408958435, -1.6396331787109375, -1.5976402759552002, -1.604270577430725, -1.6036189794540405	+
classlabels_int64s	0, 1, 2, 3, 4	+

post_transform

SOFTMAX

Additional step to make a multi class converter:

Treelist is the list of dictionary of attribute arrays, just rearranging the list for the multi-class model according to the number of classes would make the multiclass converter. The number of classes variable will be accessed through the onnx graph. Only if-and-else check is the only addition.

Performance Analysis:

For binary classification:

```
Accuracy sklearn: 0.8626  
Accuracy onnx: 0.8626  
Accuracy conifer(onnx-converter): 0.8628
```

For multi-class model:

```
Accuracy sklearn: 0.7562530120481927  
Accuracy onnx: 0.7562530120481927  
Accuracy conifer(onnx-converter): 0.7398192771084338
```

For multiclass, there is a drop in the accuracy of the onnx converter, but the similar accuracy drop was observed for the sklearn converter in case of multi-class classification. The reason is not the converter itself but the conifer library backend.

Testing:

For the purpose of testing, there were two methods:

Through the gitlab CI/CD

I have created a CERN gitlab mirror of my forked conifer repository. I have ported Jenkins le to .gitlab-ci.yml. It uses pytest to run unit tests on the conversion, simulation and synthesis of some simple BDTs.

Installing pytest in the local machine

I have installed pytest in my machine and tested the onnx-converter for rigorous continuous testing. I have written a pytest script for all the cases. The converter passed the test.

```
(base) fasif@yavin:~/conifer/tests$ conda activate hls4ml
(hls4ml) fasif@yavin:~/conifer/tests$ pytest test_onnx_to_hls.py
===== test session starts =====
platform linux -- Python 3.7.10, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/fasif/conifer, configfile: pytest.ini
collected 4 items

test_onnx_to_hls.py .... [100%]

===== 4 passed in 101.86s (0:01:41) =====
```

Completion Status:

Tasks	Status
Onnx-converter.py	Done
Testing	Done
Added in the github main branch	Done

Problems Faced:

Leaf node values only class weights

Class_weights attribute of the onnx graph only lists the values for the non leaf nodes. While working in a loop, an unequal number of attribute value errors occur because of this, so I used the nodes nodes to decide which nodes are leaf nodes and insert -2 in the values array if the value does not exist for the leaf node. But we do not need the inserted values, only the non leaf node values are useful for further processing, therefore any number inserted for this purpose won't play any role.

Array and list type error

When we rearrange the treelist into different shapes according to the multi-classes, list and array type mismatch errors occur. This is because the converter.py file is already made for all of the converters i-e scikit-learn, onnx. We cannot change the common file for our converter so this type change is necessary.

ONNX Version difference

While testing, an error arises to access the number of classes of the model. This happened due to the change of the onnx version. The issue has been resolved by adding an if-and-else block.

References:

- Summers, S. & Di Guglielmo, Giuseppe & Duarte, Javier & Harris, P. & Hoang, D. & Jindariani, S. & Kreinar, E. & Loncar, V. & Ngadiuba, J. & Pierini, Maurizio & Rankin, D. & Tran, N. & Wu, Zero. (2020). Fast inference of Boosted Decision Trees in FPGAs for particle physics. *Journal of Instrumentation*. 15. P05026-P05026. 10.1088/1748-0221/15/05/P05026.
- Duarte, Javier & Han, Song & Harris, Philip & Jindariani, Sergo & Kreinar, Edward & Kreis, Benjamin & Ngadiuba, Jennifer & Pierini, Maurizio & Tran, Nga & Wu, Zhenbin. (2018). Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*. 13. 10.1088/1748-0221/13/07/P07027.
- Di Guglielmo, Giuseppe & Duarte, Javier & Harris, Philip & Hoang, Duc & Jindariani, Sergo & Kreinar, Edward & Liu, Mia & Loncar, Vladimir & Ngadiuba, Jennifer & Pedro, Kevin & Pierini, Maurizio & Rankin, Dylan & Saguear, Sheila & Summers, Sioni & Tran, Nga & Wu, Zhenbin. (2020). Compressing deep neural networks on FPGAs to binary and ternary precision with HLS4ML.
- Coelho, Claudionor & Kuusela, Aki & Li, Shan & Zhuang, Hao & Ngadiuba, Jennifer & Aarrestad, Thea & Loncar, Vladimir & Pierini, Maurizio & Pol, Adrian & Summers, Sioni. (2021). Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Machine Intelligence*. 3. 1-12. 10.1038/s42256-021-00356-5.
- Aarrestad, Thea & Loncar, Vladimir & Pierini, Maurizio & Summers, Sioni & Ngadiuba, Jennifer & Petersson, Christoffer & Linander, Hampus & Iiyama, Yutaro & Di Guglielmo, Giuseppe & Duarte, Javier & Harris, Philip & Rankin, Dylan & Jindariani, Sergo & Pedro, Kevin & Tran, Nga & Liu, Mia & Kreinar, Edward & Wu, Zhenbin & Hoang, Duc. (2021). Fast convolutional neural networks on FPGAs with hls4ml.