PROCEEDINGS OF SCIENCE

scikinC: a tool for deploying machine learning as binaries

Lucio Anderlini[a,*] **and Matteo Barbetti**[a,b]

[a] *Istituto Nazionale di Fisica Nucleare - Sezione di Firenze, via G. Sansone, 1, Sesto Fiorentino, Italy*

[b] *Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Firenze, via Santa Marta, 3, Firenze, Italy*

E-mail: Lucio.Anderlini@fi.infn.it, Matteo.Barbetti@fi.infn.it

Machine Learning plays a major role in Computational Physics providing arbitrarily good approximations of arbitrarily complex functions, for example with Artificial Neural Networks and Boosted Decision Trees. Unfortunately, the integration of Machine Learning models trained with Python frameworks in production code, often developed in C, C++, or FORTRAN, is notoriously a complicated task. We present scikinC, a transpiler for scikit-learn and Keras models into plain C functions, intended to be compiled into shared objects and linked to other applications. An example of application to the parametrization of a detector is discussed.

Computational Tools for High Energy Physics and Cosmology (CompTools2021)
22-26 November 2021
Institut de Physique des 2 Infinis (IP2I), Lyon, France

*Speaker

PoS(CompTools2021)034

1. Introduction

The idea of approximating computational-intensive operations with simple parametric formulae is not new in Computer Science. The Remez algorithm has been commonly used to determine polynomial approximations to transcendental functions such as the exponential, the logarithm and the trigonometric functions. Machine Learning extends the opportunities of parametrizing complex tasks as sequences of simple operations to functions of many variables, possibly known only through a set of examples. For example, Artificial Neural Networks (ANNs) are often trained on large datasets to approximate some relation between different variables as a sequence of matrix multiplications, whereas Boosted Decision Trees (BDTs) can approximate similar relations with a sequence of conditional statements. In Machine Learning, the training dataset, used to determine the matrix elements used by the ANNs or variables and thresholds for the conditions tested by the BDTs, is often considered as a set of examples collected from reality, but it is more generally a set of evaluations of the function to be approximated. If evaluating that function is more expensive than inferring the result through the trained algorithm, then Machine Learning may provide a powerful tool to speed-up CPU-intensive operations. When the training dataset is generated querying a Machine Learning algorithm itself, then this approximation procedure is known as *Knowledge Distillation* and has been a subject for intense research for the last decade [1].

While the application of Machine Learning algorithms in the field of High Energy Physics has a long and glorious tradition for what concerns data analysis and statistical inference, serious attempts of employing Machine Learning algorithms to speed up the code are relatively recent and mainly flourishing in the field of Fast Detector simulation. Traditionally, simulating the response of large and complex detectors to a traversing particle requires an accurate computation of the quantum interactions of each particle with the material the detector is built with, which include the generation of new particles, and results in cascade phenomena known as particle showers. Fast Simulation approaches aim at replacing parts of the computation with statistical models, possibly defined as Machine Learning algorithms trained on the Standard Simulation or using acquired calibration data [2].

The deployment of Machine Learning techniques in such computational intensive, highly branched scenario, however, is not a trivial task. Different particles in the same collision event, for example, may undergo different physics processes and may require the evaluation of different models to provide a parametrized simulation. In the other hand, modern Machine Learning frameworks are designed for high-level languages, such as Python, and achieve CPU performance by defining batches of data where the same sequence of operation is performed on multiple data entries, which is rarely the case in the context of HEP simulation.

A further challenge is related to the slow development cycle of the large applications used for Detector Simulation, and their distribution through the computing grids where they are executed. The development and tuning of Machine Learning parametrizations of parts of the detector requires the ability to plug new models in the Simulation at runtime, without a release of the full software stack. This can be hardly achieved if the Machine Learning model is compiled together with the main application, even if retaining the possibility of loading optimized parameters from files.

Finally, long dependency chains are considered dangerous for HEP software projects that are intended to serve large communities of researchers for several decades.

In this work we propose a simple tool, named scikinC, to transpile Machine Learning models trained in Python, in plain C functions taking as input a single data entry. The transpiled functions can then be compiled as shared objects and dynamically linked to the main applications, with negligible overhead. The transpiled functions are designed to run in the same thread where they are called, and being stateless they are thread-safe by design.

In this document, we will briefly review the state of the art and related projects in Section 2, then we will provide a description of the implementation of scikinC in Section 3, and finally, in Section 4, we will present a simplified application of scikinC to Fast Simulation. Section 5 concludes the document with a short summary.

2. State of the art and related projects

The to-go deployment option for Machine Learning is the ONNX runtime [3]. ONNX, originally an open format for neural network exchange, provides today extensions to deploy models trained with several frameworks, including scikit-learn [4] and Keras [5]. Since the focus of ONNX is on interoperability and performance, it provides a runtime optimized for various platforms including multiple CPUs, GPUs and other hardware accelerators. The ONNX runtime has APIs for several languages including C and C++, but unfortunately the need for a runtime designed to get the most from parallel computation on multiple threads introduces a small overhead and sets some minor thread-safety issues. These considerations make ONNX ideal for a number of applications in HEP as long as the model is complex enough or the batches can be large enough to make the overhead negligible, this is especially true for reconstruction or analysis tasks, and for some special task in Fast Detector Simulation such as simulating the energy deposits in a calorimeter in a collision event.

In order to avoid the need for a runtime and aiming at a drastical reduction of the amount of external dependencies for long-term scientific applications, the HEP community developed the LWTNN project [6], providing a format to exchange Machine Learning algorithms alternative to ONNX, which can then be interpreted and executed as part of the main application. LWTNN is popular in the HEP community because of its simplicity.

A more drastic approach to reduce the complexity of the dependency tree is to separate the applications in charge of running the main task and the Machine Learning inference infrastructure in two different processes communicating via sockets. The inference application acts as a server specialized in Machine Learning applications, providing optimal access to hardware accelerators and building batches possibly combining the requests from several concurrent instances of the main application. The server application does not even need to run on the same machine and can be written in Python in the exact same environment as used for training. Machine Learning as a Service (MLaaS) is a fascinating opportunity for extremely computing-intensive tasks, where the large overhead due to the inter-process communication becomes negligible in front of the speed-up obtained offloading the inference to specialized hardware accelerators [7].

At the opposite end, the HEP community is developing tools to compile Machine Learning models to optimize the inference of small models on tiny batches, which still cover an impressing amount of use cases in our simulation, reconstruction and analysis applications. Compiling the models also helps to reduce the memory footprint of the inference routines, which is especially

important when multiple instances of the application run in parallel as different threads or processes on the same machine. The most advanced transpiler for machine learning models is being developed by the CERN ROOT team as part of the Toolkit for MultiVariate Analysis (TMVA), and is recently presented as SOFIE (System for Optimized Fast Inference code Emit) [9]. SOFIE extends TMVA by introducing the ability to parse ONNX files and convert them into C++ code. The generated code can be compiled with a C++ compiler with some Basic Linear Algebra Subprogram (BLAS) library as the only dependency. While an extremely promising tool providing conversion templates for a large variety of ONNX operators, SOFIE comes as part of the ROOT framework, which sets serious limitations in terms of portability, especially when targeting the cloud-based Python environments commonly used for training.

Outside of the HEP community, there are several active projects to develop compiler for Deep Neural Networks built on top of LLVM, for example TVM [10], nGraph [11], Tensor Comprehensions [12], Glow [13] and Extended Linear Algebra (XLA) [14]. A comprehensive review of the different approaches adopted in the various projects can be found in Ref. [15]. Each of these projects produces compiled and optimized representations of Deep Neural Networks that can be linked to applications developed in several languages, which will then depend on runtimes or at least on external kernels implementing the actual computation.

Interesting experiences were reported in the field of Machine Learning deployment on microcontrollers and real-time processing, requiring extremely low latency and being usually designed for tiny batches. We report in particular on two packages, `emlearn` [17] and `keras2c` [16], providing transpilers for several `scikit-learn` and Keras models to C. Conceptually, these recent packages are similar to `scikinC`, with minor differences in the implementation choices rather than in the design principles. In particular, `keras2c` is more mature and complete than `scikinC` on the Keras models, which is more focused on `scikit-learn` models and their distribution, as discussed in the next Section.

3. `scikinC` implementation

As mentioned in the introduction, `scikinC` was primarily intended for offering a simple solution to integrate small to medium-sized Machine Learning models within a Fast Simulation framework.

The models, trained with `scikit-learn` (or Keras), are converted to C code using the parser made available by `scikinC`. The generated code can be immediately compiled into a dynamically linkable library using a C compiler, such as GCC, and then dynamically linked to the main application using Standard C libraries. Producing the C code, `scikinC` includes the weights of the models avoiding dependencies on configuration files. Dynamic linking enables selecting the model to use for inference at runtime via the option files or the command-line arguments used to configure the main application, enabling frequent updates of the models (for example for validation purpose) without releasing the whole software stack to production. Figure 1 depicts the simple procedure to compile and link a model in a C/C++ application.

Internally, `scikinC` fills template C implementations of the forward pass, or inference, for the supported `scikit-learn` and Keras models by accessing the attributes and properties exposed

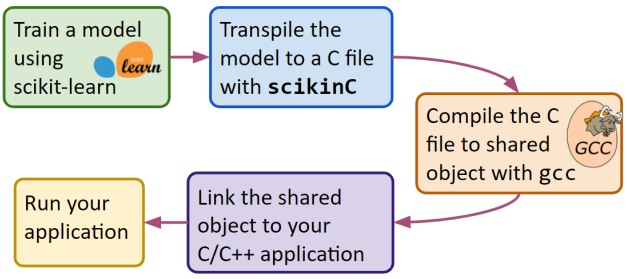

Figure 1: Sketch of the working principle of scikinC.

by the class. The modular design, involving a separate template for each model, simplifies the extension of the package to support additional models.

Currently, template implementations for the following scikit-learn models are provided:

- Preprocessing steps
  - MinMaxScaler
  - StandardScaler
  - QuantileTransformer
- Model steps
  - GradientBoostingClassifier
- Other steps
  - Pipeline

The support for Keras is minimal and limited to Sequential models of Dense layers. The supported activation functions are tanh, sigmoid, relu, PReLU, and LeakyReLU.

All the functions defined by scikinC are intended for single entries as no support is provided for batch inference, and share the same signature:

```
float *functionname (float* output, const float* input);
```

where input and output are arrays containing the inputs features of the model and the predictions obtained. The pointer to the output array is also returned to ease inline operations on the result of the computation. The number of input features and produced output variables is fixed by the model and is supposedly known by the client application. Passing the size of the arrays as additional arguments is therefore unnecessary.

During code generation, scikinC disables the name mangling of the linker symbols assigning to the functions representing the entry point for the evaluation of each model user-defined names and symbols. Combining the standard function signature and user-defined linker symbol, the function can be accessed easily by using the function dlopen which is part of standard C libraries.

Each one of the available template implementations is associated to a test entry defined within the pytest framework[1]. For each test, a simple model is trained using scikit-learn (or Keras),

[1] see pytest.org

it is transpiled to C, compiled with GCC, and linked to a test C application designed to compare the output of the inference of the same model on the same data as obtained with Python and C. If the difference is not much larger than the floating-point precision the test succeeds.

The scikinC codebase is released under MIT license and its Git repository is publicly available on GitHub[2].

4. Example application to a simplified Fast Simulation

Consider the following scenario. We have access to a small sample of simulated data from some experiment. We would like to model the response of the detector in terms of efficiency and resolution. The models will then be integrated in the C++ framework we developed to model new-physics effects, enabling quick checks on whether the experiment would be sensitive to our model.

We consider the decay of 180 MeV/c neutral kaons into three pions $K^0 \to \pi^+\pi^-\pi^0$. The experiment tags these decays using a calorimeter to reconstruct the π^0, but this requires neutral pions with a momentum higher than 70 GeV/c to reject some background. We will assume the efficiency of this requirement is zero for neutral pions below 70 GeV and 100% for neutral pions above 70 GeV.

This requirement introduces an efficiency term on the Dalitz plot that might compromise the sensitivity of the experiment to our model, which will be tested studying the invariant mass of the two charged pions: $m_{\pi^+\pi^-}$.

The latter variable is reconstructed with a rudimental spectrometer reconstructing the momentum of the charged pions with an error

$$\frac{\delta_p}{p} \approx 1\,\mathrm{GeV}^{-1} \cdot p$$

For simplicity, we will assume the same error on the neutral pion.

The plots in Figure 2 present the effect of the requirement on the momentum of the neutral pion on the other variables of the system. Clearly, we observe a suppression of part of the Dalitz plot corresponding to higher values of $m_{\pi^+\pi^-}$.

To model the efficiency in a quick and reliable way we train a Gradient Boosting Decision tree to predict the probability that an event will be reconstructed given the coordinates of the Dalitz plot: $m_{\pi^+\pi^-} \perp m_{\pi^+\pi^0}$. The performance of the trained model is shown in Figure 3.

Similarly, the resolution function associated to each entry in the Dalitz plots can be modeled with a Neural Network. As the uncertainty on the momentum depends on the momentum itself, a variation of the uncertainty on the invariant masses through the Dalitz plot is expected, and interesting to model.

To simplify the setup, a simple Neural Network is trained to predict the standard deviation of the distribution of the experimental errors on the reconstructed $\pi^+\pi^-$ mass as a function of the Dalitz plot coordinates. What we obtained from this simplified parameterization is reported in Figure 4.

In real experiments, Generative Adversarial Networks, Variational Autoencoders or Gaussian Mixture Models are used to model resolution functions with neural networks. The training of these

[2] see https://github.com/landerlini/scikinC

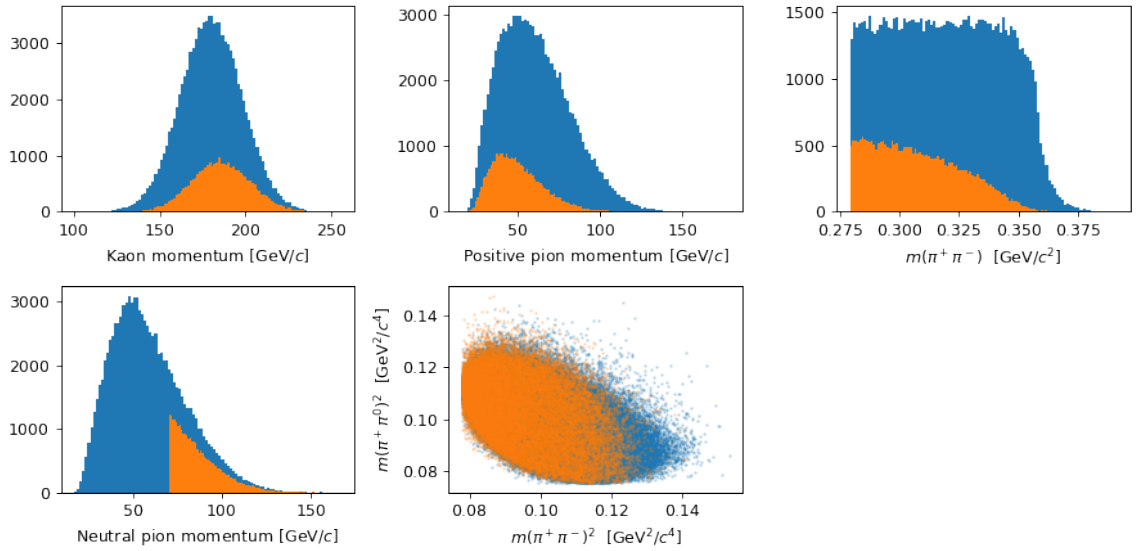

Figure 2: Representation of simplified simulated detector effects that we wish to model with Machine Learning algorithms. Blue (darker) histograms and markers represent the generated events prior of any reconstruction and selection, while the orange (lighter) histograms and markers represent the reconstructed and selected events.

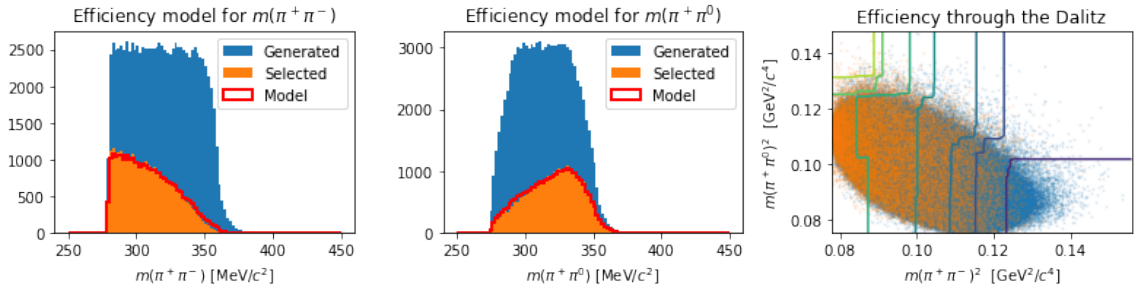

Figure 3: Representation of the detector efficiency parametrization using a Boosted Decision Tree trained with `scikit-learn`.

advanced algorithms, however, is beyond the scope of this example, as their deployment would happen exactly the same way as discussed below for this simpler model.

In order to ease the training of the neural network we will use a prescaling step, based on the MinMaxScaler class of `scikit-learn`.

Once the efficiency BDT, the resolution ANN and its preprocessing steps are defined, `scikinC` can produce the corresponding C code:

```
import scikinC

convertedcode = scikinC.convert (dict(
  efficiency_model=efficiency_model,
  resolution_preprocessing=preprocessing,
  resolution_model=resolution_model))
```

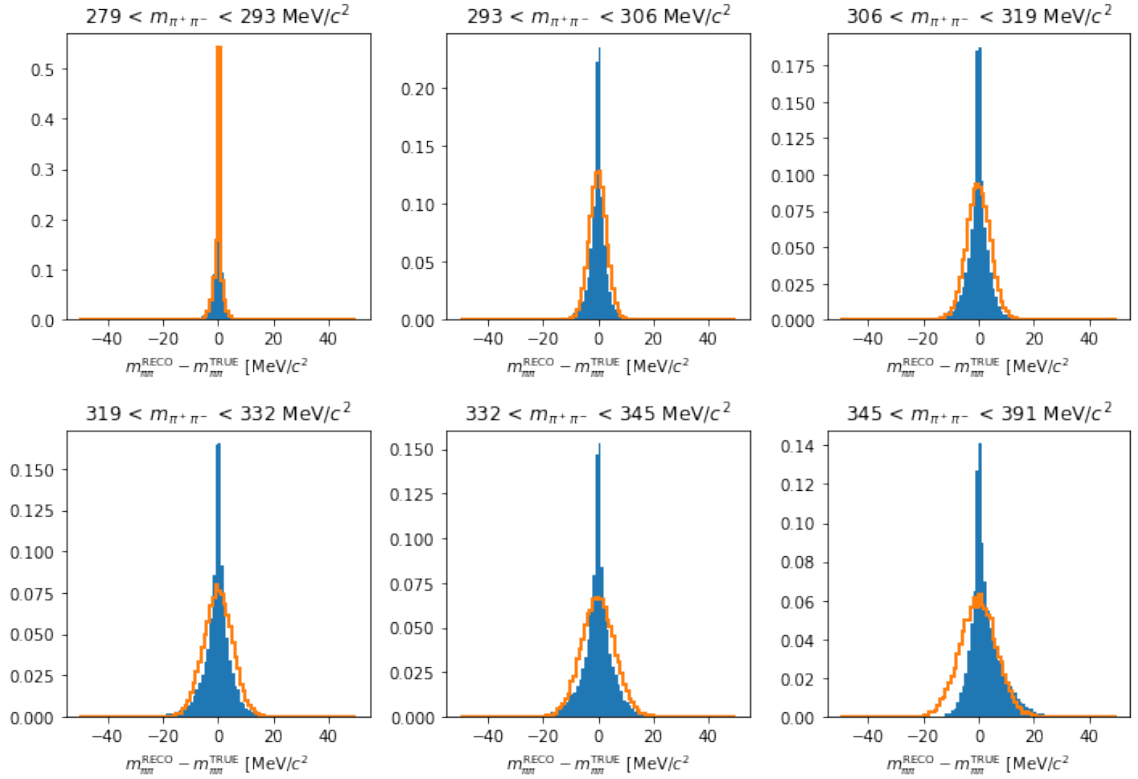

Figure 4: Simplified model for the resolution, with a Neural Network predicting the standard deviation of the difference between the reconstructed and real $\pi^+\pi^-$ masses as a function of the mass itself. The ground-truth resolution model is represented as a blue filled histogram, while the Gaussian prediction is shown as an orange line.

Here, the dictionary keys, placed at left of the equal signs, will be used as linker symbols in the compiled shared library, while the dictionary values, placed at right of the equal signs, are the `scikit-learn` and Keras objects defined the trained models.

The content of `convertedcode` is C code containing the instructions to evaluate the converted models. Saving `§convertedcode` to a file, for example `converted.C` it can be immediately compiled to a shared object as

```
gcc -o converted.so --shared -fPIC -Ofast converted.C
```

The binary version of the models representing the resolution and efficiency of the simulated detector are now stored in a shared library that can be dynamically linked to other applications, for example using the `dlopen` function. Listing 1 reports the listing of a minimal C application employing the parametrization for detector efficiency and resolution as modeled with Machine Learning algorithms.

Note that the both the name of the shared object file and of the linker symbol are strings that can be taken as inputs at runtime. On the other hand, changing the parametrization of the efficiency from a BDT to an ANN could be achieved without recompiling the main application.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// --> Import the dlfcn header
#include  <dlfcn.h>

// --> Define the signature for scikinC functions and name it mlfunc
typedef float *(*mlfunc)(float *, const float*);

int main (int argc, char* argv[])
{
  float m_pipi = atof(argv[1]);
  float m_pipi0 = atof(argv[2]);

  // --> Open the converted.so shared object
  void *handle = dlopen ("./converted.so", RTLD_LAZY);

  // --> Load the functions from the shared object based on their names
  mlfunc efficiency_model = (mlfunc) dlsym (handle, "efficiency_model");
  mlfunc resolution_model = (mlfunc) dlsym (handle, "resolution_model");
  mlfunc resolution_preprocessing = (mlfunc) dlsym (handle, "resolution_preprocessing");

  float efficiency [1];
  float input[]  = {m_pipi, m_pipi0};
  // --> Once loaded, trained models can be called as normal functions.
  efficiency_model (efficiency, input);
  printf ("Efficiency: %.3f\n", efficiency[0]);

  float preprocessed [2], resolution [2];
  resolution_preprocessing (preprocessed, input);
  resolution_model (resolution, preprocessed);

  // ...

  return 0;
}
```

Listing 1: Listing of a minimal C application dynamically linking to the library generated with `scikinC`.

5. Conclusion

Deploying trained Machine Learning algorithms in environments independent of the setup used for training is critical to many applications. Academia and Industry have developed a number of solutions targeting for example edge devices and the Internet-Of-Things. Most of the approaches are intended for the inference of large models on large input data performing task such as image and speech recognition, introducing significant overhead in the computation, exploiting multithreading and involving important dependency trees. In the field of Physics research these are important limitations for a number of applications, including real-time data processing, deployment on micro-controllers and deployment in large, highly-branched software applications processing data through

the grid.

The proposed solution is to compile in C or C++ language the operations to be performed during the inference of the trained models, with minimal dependencies on external projects. We presented scikinC a simple software tool for transpiling selected Machine Learning models trained using scikit-learn and Keras to compatible C code. We discussed how the generated code can be immediately compiled to shared libraries and then dynamically linked to large applications.

During the development of scikinC, we became aware of two ongoing projects with similar, but not completely overlapping, goals: emlearn targeting microcontrollers and keras2c targeting real-time data processing. The latter introduces a more general signature for the transpiled C functions including the shape of the input and output tensors. In the future, scikinC will share the same signature in order to make switching from a keras2c model to a scikinC model transparent for the client application. On the same line, we will consider offloading the conversion of Keras models to keras2c focusing on models trained with scikit-learn, instead, aiming at a complete toolbox of C transpilers for Machine Learning models with the lowest latency.

References

[1] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, *"Distilling the Knowledge in a Neural Network"*, arXiv preprint 1503.02531

[2] Lucio Anderlini, *"Machine Learning for the LHCb Simulation"*, arXiv preprint 2110.07925

[3] ONNX Runtime developers, *"ONNX Runtime"*, onnxruntime.ai (2021)

[4] F. Pedregosa *et al.*, *"Scikit-learn: Machine Learning in Python"*, JMLR 12 pp. 2825-2830, 2011

[5] François Chollet *et al.*, *"Keras"*, github.com/fchollet/keras

[6] Daniel H. Guest *et al.* *"LWTNN"*, doi:10.5281/zenodo.597221

[7] Valentin Kuznetsov *et al.*, *"MLaaS4HEP: Machine Learning as a Service for HEP"*, arXiv preprint 2007.14781

[8] Andreas Hocker *et al.*, *"TMVA - Toolkit for Multivariate Data Analysis"*, CERN-OPEN-2007-007, arXiv preprint physics/0703039

[9] Sitong An, Lorenzo Moneta, *"C++ Code Generation for Fast Inference of Deep Learning Models in ROOT/TMVA"*, EPJ Web of Conferences 251, 03040 (2021)

[10] Tianqi Chen *et al.*, *"TVM: An automated end-to-end optimizing compiler for deep learning"*, arXiv preprint 1802.04799

[11] Scott Cyphers *et al.*, *"Intel nGraph: An Intermediate Representation, Compiler and Executor for Deep Learning"*, arXiv preprint 1801.08058

[12] Nicolas Vasliache *et al.*, *"Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions"*, arXiv preprint 1802.04730

[13] Nadav Rotem *et al.*, *"Glow: Graph Lowering Compiler Techniques for Neural Networks"*, arXiv preprint 1805.00907

[14] Google XLA team, *"XLA - TensorFlow, compiled"*, Google developers, March 6th, 2017

[15] Mingzhen Li *et al.*, *"The Deep Learning Compiler: A Comprehensive Survey"*, IEEE Transactions on Parallel & Distributed Systems **32** (2021) 03, arXiv preprint 2002.03794

[16] Rory Conlin *et al.*, *"Keras2c: A library for converting Keras neural networks to real-time compatible C"*, Volume 100, April 2021, 104182

[17] Jon Nordby, *"emlearn: Machine Learning inference engine for microcontrollers and embedded devices"*, doi:10.5281/zenodo.2589394