Istituto Nazionale di Fisica Nucleare
SEZIONE DI FIRENZE

UNIVERSITÀ DEGLI STUDI FIRENZE

landerlini/scikinC

scikinC

a tool for deploying machine learning as binaries

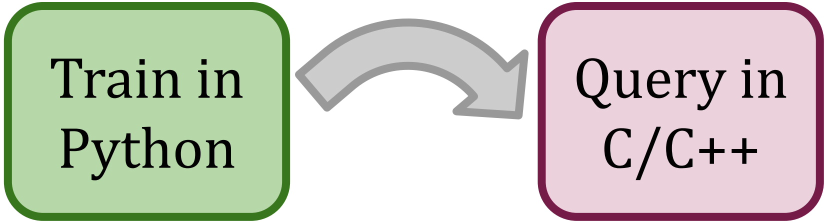
Lucio Anderlini [1], Matteo Barbetti [1,2]

[1] Istituto Nazionale di Fisica Nucleare – Sezione di Firenze
[2] Università degli Studi di Firenze – Dipartimento di Ingegneria dell'Informazione

Computational Tools for High Energy Physics and Cosmology 2021-11-26

Motivation

Wider and wider usage of machine learning algorithms in HEP C++ applications.



Several options for deployment exist, but come with some practical limitation.

For example,

- Require **external dependencies** sometimes difficult to integrate in the build system of large HEP applications
- Expect vectorized inputs introducing **overhead for branched flows**, as for example Geant4-based simulations
- Introduce limits in the interplay between the **preprocessing** and **algorithmic** steps
- Often require **compiling with the framework** large part of the algorithm.

The crib of scikinC: *the parametric simulation of LHCb*

Speeding up the simulation of the collision events at the LHC is a clear priority:

the current model is unsustainable for future Runs of the LHC.

Among the options under investigation, ***ultra-fast simulation*** aims at ***replacing detector simulation and the subsequent reconstruction with machine learning algorithms*** or simpler parametrizations for the higher level quantities used in physics analysis.

Sim applications are **long pipelines of tens of ML algorithms**, each simulating response and reconstruction of a part of the detector.

Generator → **Geometrical Acceptance** → **Reconstruction Efficiency** → **Tracking Efficiency** → **Charged Particle Identification** → **Calorimeters Simulation** → …

Goal: switch from BDT model for the efficiency to a NN without touching the framework.

The idea: *dynamically link to compiled models*

The input and the expected output for **each model is defined within the framework**, but then the ML model can be defined as a plug-in and dynamically linked to the main application as a shared object.

Models can be developed and released independently of the framework application as long as retaining the same list of input and output features.
$\longrightarrow$ development cycle of ML algorithms is much faster than for framework applications

Preprocessing steps must also be included in the shared object as part of the main algorithm (*e.g.* BDTs and NNs have different requirements in terms of preprocessing).

Need a tool to transparently compile into ***shared objects*** ML algorithms trained in Python.

scikinC: *a transpiler of ML models from Python to C*

Train a model using scikit-learn

→ Transpile the model to a C file with **`scikinC`**

→ Compile the C file to shared object with gcc

→ Link the shared object to your C/C++ application

→ Run your application

1. Train a model with scikit-learn

```
import numpy as np
import pickle

from sklearn.preprocessing import MinMaxScaler

minmax = MinMaxScaler()
minmax.fit ( np.random.normal(0,5, (2,1000) )

with open("example_scaler.pkl", 'wb') as f:
  pickle.dump (minmax, f)
```

Models supported by `scikinC` include several preprocessing steps, BDTs and some keras Deep Neural Networks.

Training happens as usual, independently of `scikinC`.

2. Transpile the model to a C file with `scikinC`

Trained models can be transpiled directly in Python,

```
import scikinC
```

```
c_string = scikinC.convert({
  'myMinMaxScaler': minmax
})
```

or stored with pickle and converted with `scikinC` CLI.

```
scikinC example_scaler.pkl > Cfile.C
```

3. Compile with gcc

```
gcc -o deployed_scaler.so Cfile.C -shared -fPIC -Ofast
```

For most applications, we recommend the options `-Ofast` enabling several optimizations in the gcc compilation.

```
nm -D deployed_scaler.so

0000000000203028 B __bss_start
                 w __cxa_finalize
0000000000203028 D _edata
0000000000203030 B _end
0000000000000894 T _fini
                 w __gmon_start__
0000000000000560 T _init
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
00000000000006b0 T myMinMaxScaler
00000000000007b0 T myMinMaxScaler_inverse
                 U __stack_chk_fail
```

gcc

gcc

```
c_string = scikinC.convert({
  'myMinMaxScaler': minmax
})
```

scikinC

scikinC

```
/*******************************************************************************/
/* File automatically generated with scikinC (github.com/landerli/scikinC) */
/*                                                                          */
/*                         D O   N O T   E D I T   ! ! !                    */
/*                                                                          */
/*  File generated on 2021-11-23 11:12
/*  by ColabExample
/*  using MinMaxScalerConverter
/*
/*******************************************************************************/
#define FLOAT_T float

    extern "C"
    FLOAT_T* myMinMaxScaler (FLOAT_T* ret, const FLOAT_T *input)
    {
      int c;
      FLOAT_T input_min[] = {-0.22432032741316995650, -0.60533476137983122101, -8.3757818989091
      FLOAT_T input_max[] = {2.23350308399643626700, -0.08018045562135850401, 2.536716236579525
      FLOAT_T output_min = 0.000000;
      FLOAT_T output_max = 1.000000;

      for (int c = 0; c < 1000; ++c)
        ret [c] = (input[c] - input_min[c]) / (input_max[c] - input_min[c])
                * (output_max - output_min) + output_min;

      return ret;
    }


    extern "C"
    FLOAT_T* myMinMaxScaler_inverse (FLOAT_T* ret, const FLOAT_T *input)
    {
      int c;
      FLOAT_T input_min = 0.000000;
      FLOAT_T input_max = 1.000000;
      FLOAT_T output_min[] = {-0.22432032741316995650, -0.60533476137983122101, -8.375781898909
      FLOAT_T output_max[] = {2.23350308399643626700, -0.08018045562135850401, 2.53671623657952

      for (int c = 0; c < 1000; ++c)
        ret [c] = (input[c] - input_min) / (input_max - input_min)
                * (output_max[c] - output_min[c]) + output_min[c];

      return ret;
    }
```

4. Link the shared object to your C/C++ application

All models converted by scikinC share the prototype:

```
float *(*mlfunc)(float *, const float*);
```

Output tensor   Input tensor

Load the file ***by path***, and link to the function ***by name.***

Allocate some memory for your input and output tensors, and evaluate the model calling a function.

Finally, release the library.

```c
// C Library for dynamic linking
#include <dlfcn.h>

// Define the type for generic machine learning functions
typedef float *(*mlfunc)(float *, const float*);

void somewhere_in_your_code (void)
{
  // Open the shared object library
  void *handle = dlopen ("./deployed_scaler.so", RTLD_LAZY );
  if (!handle)
    exit(1);

  // Load the scaler by name (as from Python dictionary key)
  mlfunc minmax = mlfunc(dlsym (handle, "myMinMaxScaler"));

  // Prepares the input and output buffer and evaluate the function
  float *inp [] = { /* your input goes here */ };
  float *out [ /*output n_features goes here*/ ];
  minmax ( out, inp );

  // Optionally, closes the linked library file
  dlclose(handle);
}
```

4. Link the shared object to your C/C++ application

The path of the shared object and the name of the symbol are strings and can be defined at runtime, without recompiling anything.

```
// C Library for dynamic linking
#include  <dlfcn.h>

// Define the type for generic machine learning functions
typedef float *(*mlfunc)(float *, const float*);

void somewhere_in_your_code (const char libpath, const char* funcname)
{
  // Open the shared object library
  void *handle = dlopen (libpath, RTLD_LAZY );
  if (!handle)
    exit(1);

  // Load the scaler by name (as from Python dictionary key)
  mlfunc minmax = mlfunc(dlsym (handle, funcname));

  // Prepares the input and output buffer and evaluate the function
  float *inp [] = { /* your input goes here */ };
  float *out [ /*output n_features goes here*/ ];
  minmax ( out, inp );

  // Optionally, closes the linked library file
  dlclose(handle);
}
```

Implemented algorithms (scikit-learn)

Scikit-Learn preprocessing

Model	Implementation	Test	Notes
`MinMaxScaler`	Available	Available	
`StandardScaler`	Available	Available	
`QuantileTransformer`	Available	Available	
`Pipeline`	Available	Partial	Pipelines of pipelines break

Scikit-Learn models

Model	Implementation	Test	Notes
`GradientBoostingClassifier`	Available	Available	

A few other preprocessing steps in the pipeline…

Implemented algorithms (keras)

Keras Models

Model	Implementation	Test	Notes
`Sequential`	Available	Available	

Keras Layers

Model	Implementation	Test	Notes
`Dense`	Available	Available	
`PReLU`	Available	Available	
`LeakyReLU`	Available	Available	

Keras Activation functions

Model	Implementation	Test	Notes
`tanh`	Available	Available	
`sigmoid`	Available	Available	
`relu`	Available	Available	

tvm is a very promising (and ambitious) project aiming at compiling deep models through LLVM.
We are evaluating offloading to tvm models not natively-supported by `scikinC`.

Known general issues

- Programming in C is fun, everything is simple and lean. At least until you get a Segmentation Fault.
- Distribution of binaries may hinder computer security and limit portability of the applications.
- Compilation of very large models may require several hours (especially for BDTs).

Conclusion

`scikinC` is a small stand-alone tool to convert Python-trained ML algorithms in C functions.

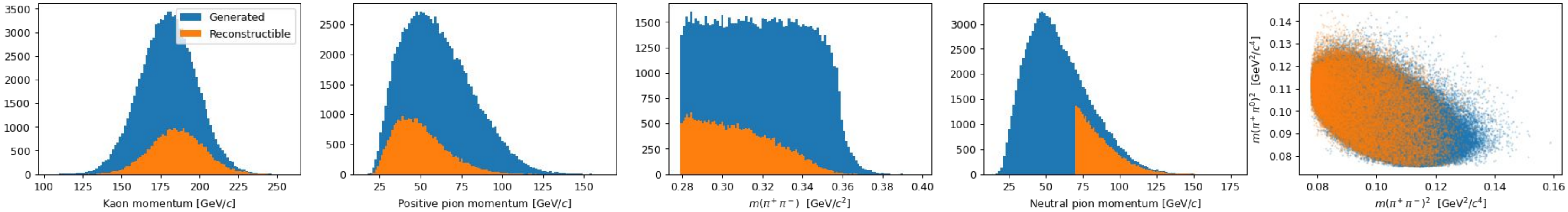
C functions can then be easily compiled into shared objects and dynamically linked to other applications.

While not yet a mature package, **`scikinC` is rather modular and not difficult to extend**. The few **models it currently includes** are sufficient to cover a large variety of applications, including **several parametrizations for the ultra-fast simulation of the LHCb experiment**.

If applying `scikinC` to your own tool sounds interesting, don't hesitate to get in touch!

In the tutorial

https://colab.research.google.com/drive/1E0jWfS7aQJqvdArDYibwWe3QAov_qgDj?usp=sharing



- Mock your own detector simulation
- Model the experimental efficiency with a Gradient Boosting Decision Tree
- Model the resolution with a Neural Network in keras
- Deploy everything with scikinC into a binary shared object
- Compile, link and validate the deployed model in Python and C applications.