HEP ML Lab: An end-to-end framework for signal vs background analysis in high energy physics

Jing Li, Hao Sun

Dalian University of Technology

15th July, 2023

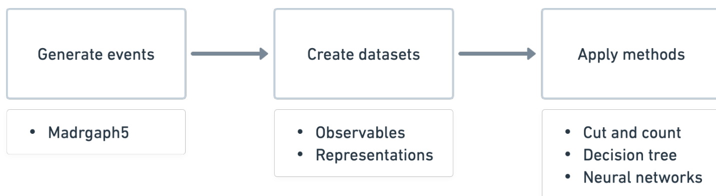
Table of Contents

1. Introduction: why we need an end-to-end framework?
2. Three core parts: generate events, create datasets, apply methods
3. Future: share and cooperate

Table of Contents

1. Introduction: why we need an end-to-end framework?
2. Three core parts: generate events, create datasets, apply methods
3. Future: share and cooperate

Introduction: why we need an end-to-end framework?



HEP ML Lab (HML) is an end-to-end framework for signal vs background analysis. It combines the machine learning (ML) techniques with the high energy physics (HEP) researches.

In a complete study of signal vs background analysis, a lot of details come into play:

- What phase space cuts are applied to the events?
- What kind of data processing is performed?
- How is such data utilized by different classifiers?
- ...

Introduction: why we need an end-to-end framework?

Even though physicists describe every step as clearly as possible, it is inevitable subsequent researchers may face difficulties, or even fail to reproduce the results:

- The exact version of the packages used in the study may be not available nowadays.
- Different implementations of the same model may lead to different results.
- ...

The unreliable reproduced results lead to difficulties evaluating the new proposed physics processes, and thus slow down the progress of lowering the upper limit of new physics discoveries.

Introduction: why we need an end-to-end framework?

That's why we need an end-to-end framework:

- It covers every step of the research processes;
- It provides certain assurance for reproducibility;
- It facilitates researchers to adopt new methods and compare them quickly.

So, why do we need the HEP ML Lab? Is it simply another one of such frameworks?

Let's now take a moment to review the existing frameworks currently available. Please note that the following list maybe incomplete due to our limit knowledge and the comparison we made does not cover highlights of every framework, which may lead to unfairness.

Introduction: why we need an end-to-end framework?

Name	Data generation	Model training	Style
hml ml	✗	✓	sklearn
weaver	✗	✓	CLI + config
JetNet	✗	✗	custom
pd4ml	✗	✓	modified keras
MLAnalysis	✗	✓	custom
mapyde	✓	✗	CLI + config

Data generation refers to obtaining raw data from event generators. While only the last one is capable of this, others offer quite good support for existing datasets.

Model training includes providing built-in models and corresponding training methods. Most of the frameworks offer comprehensive support for this, despite their different implementations and usage styles.

Introduction: why we need an end-to-end framework?

As you can see, for the two core components - **data** and **models**, there currently isn't a framework that completely covers both. This is the motivation of the HEP ML Lab.

Another point of interest is the data processing that connects these two parts: how raw event data is processed into inputs that various models can use. Our framework has also made efforts in this area, enabling researchers to smoothly obtain results from start to finish.

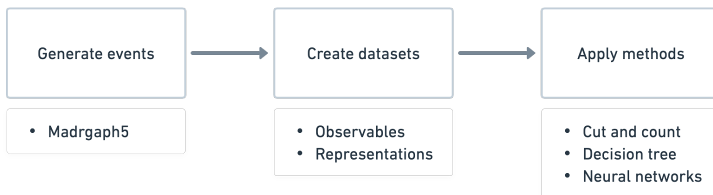


Table of Contents

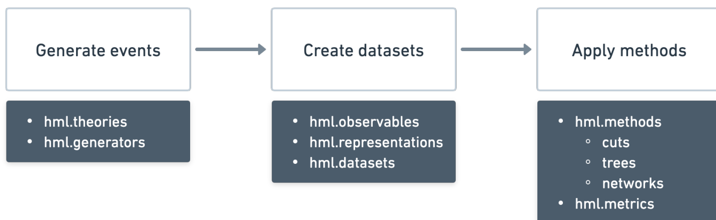
1 Introduction: why we need an end-to-end framework?

2 Three core parts: generate events, create datasets, apply methods

3 Future: share and cooperate

Three core parts: module overview

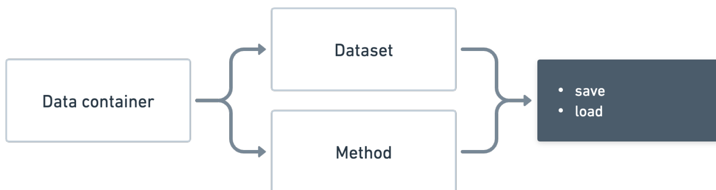
Modules of HEP ML Lab (HML) are in charge of the three core parts:



Note here "model" is a quite general term, which can be a physics theory model, a machine learning model. To avoid confusion, we use "theories" to refer to theory models and "methods" to refer to cuts, trees and neural networks.

Three core parts: data containers

The design concept of the data containers (datasets, models) is to facilitate users in saving and reading data. We save the parameters used to instantiate data containers as metadata in YAML format, while the data itself is saved as files named after the corresponding container, depending on its format.



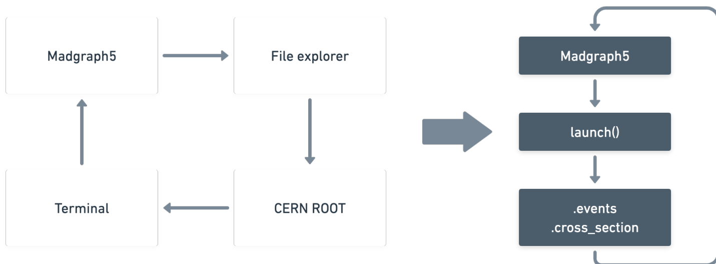
Three core parts: generate events

Let's first generate some events. We use the `Madgraph5` module to generate events. The following code snippet shows how to generate 10000 events of $pp \to ZZ \to jj\nu_e\bar{\nu}_e$ with the `mg5_aMC` event generator. We also generate background events $pp \to jj/Z$ for comparison:

```
>>> from hml.generators import Madgraph5
Welcome to JupyROOT 6.24/02
>>> signal_generator = Madgraph5(
...     executable="mg5_aMC",
...     processes="p p > z z, z > j j, z > ve ve~",
...     output_dir="./data/pp2zz",
...     shower="Pythia8",
...     detector="Delphes",
...     settings={
...         "nevents": 10000,
...         "iseed": 42,
...         "htjmin": 400,
...     },
... )
```

Generate events: Madgraph5 API

Madgraph5 Class is a simple wrapper of the `mg5_aMC` executable.



Initializing a `Madgraph5` object is equivalent to running the `mg5_aMC` executable with the given parameters.

Generate events: launch the generator

The method `launch` starts the generation immediately. The generator will monitor the run log and print the progress like this:

```
>>> signal_generator.launch()
    Generating events...
    Running Pythia8...
    Running Delphes...
    Storing files...
    Done
    >>> background_generator.launch()
    ...
```

Generate events: check the output

After the generation is finished, the generator will automatically read the cross section from the log file and the output root file.

```
>>> sig_run = signal_generator.runs[0]
... bkg_run = background_generator.runs[0]
... print(f"cross section (pb): {sig_run.cross_section}")
... print(f"cross section (pb): {bkg_run.cross_section}")
... print(f"number of events: {sig_run.events.GetEntries()}")
... print(f"number of events: {bkg_run.events.GetEntries()}")
cross section (pb): 0.00077034
cross section (pb): 56849.047629999994
number of events: 10000
number of events: 10000
```

Here we use `PyROOT` as the backend to read the root file.

As now we can easily access the raw events data, let's move on to creating datasets.

Create datasets: represent an event in a proper way

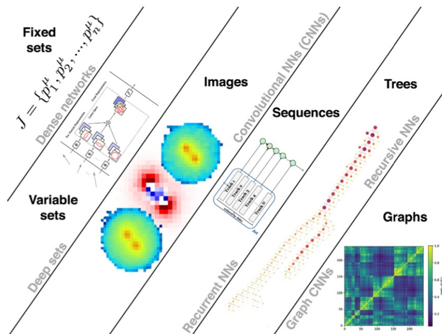


Figure: Jet representations from 1709.04464

An event can be represented in different ways based on its constituent particles, or it can be saved in different data formats. There are mainly three representations: Set, Image, and Graph. This time, let's use Set as an example for explanation.

Create datasets: represent an event in a proper way

```
>>> from hml.generators import MG5Run
>>> from hml.representations import Set
>>> from hml.observables import Pt, M, DeltaR
>>> sig_run = MG5Run("./data/pp2zz/Events/run_01/")
>>> bkg_run = MG5Run("./data/pp2jj/Events/run_01/")
>>> representation = Set([Pt("Jet1"),
...                       Pt("Jet2"),
...                       DeltaR("Jet1", "Jet2"),
...                       M("FatJet1")])
```

- `MG5Run` is used to link the generated events and other information, such as cross section, together.
- `Set` is used to represent an event as a set of observables.
- `Pt`, `M`, `DeltaR` are observables fetched from particles in the event.

We declare a representation with four observables: the transverse momentum of the first jet and the second jet, the ΔR between them, and the mass of the first fat jet.

Create datasets: fill the dataset

```python
import numpy as np

data, target = [], []

for event in sig_run.events:
    if event.Jet_size >= 2 and event.FatJet_size >= 1: # preselection
        representation.from_event(event)
        data.append(representation.values)
        target.append(1)

# Do the same for background events
...

data = np.array(data, dtype=np.float32)
target = np.array(target, dtype=np.int32)
```

Loop over the events and fill the data and target arrays.

Create datasets: save the dataset

```python
from hml.datasets import Dataset

dataset = Dataset(
    data,
    target,
    feature_names=representation.names,
    target_names=["pp2jj", "pp2zz"],
    description="This is a demo dataset for Z vs QCD jets.",
    dataset_dir="./data/z_vs_qcd",
)

dataset.save()
```

Complement the dataset with other information and save it to disk.

The way we create the dataset is similar to the `scikit-learn` API, which is quite straightforward and easy to use.

Apply methods: method overview

Currently, three types of methods are supported:

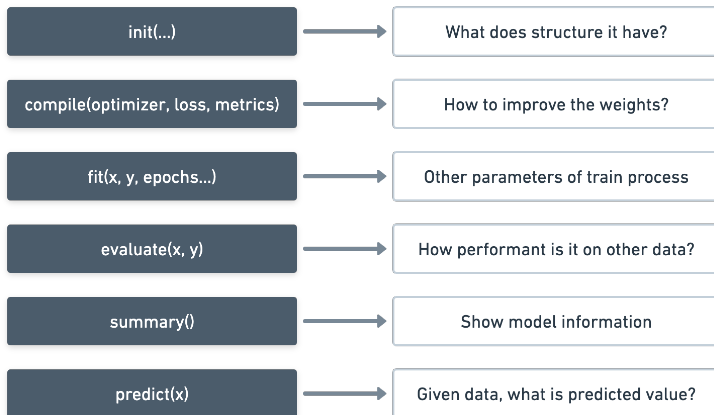
- Cuts: `hml.methods.cuts`
- Trees: `hml.methods.trees`
- Networks: `hml.methods.networks`

```
from hml.methods import CutAndCount, BoostedDecisionTree, ToyMLP
```

Users done't have to remember the exact module names, we create shortcuts for them. Just import them from `hml.methods` and use them directly.

Although the methods are implemented in different ways, they all follow the `Method` protocol, which means they all have the same API.

Apply methods: Keras style protocol - Method



`Method` is the minimum wrapper of the original Keras to make it compatible with other methods.

Apply methods: use diffent methods in unified style

```
>>> from hml.datasets import Dataset
>>> from sklearn.model_selection import train_test_split
>>> from keras.utils import to_categorical
>>> # Load the dataset and split it into train and test sets
>>> dataset = Dataset.load("./data/z_vs_qcd")
>>> x_train, x_test, y_train, y_test = train_test_split(
...     dataset.data, dataset.target, test_size=0.2, random_state=42
... )
>>> # Convert the labels to categorical
>>> y_train = to_categorical(y_train, dtype="int32")
>>> y_test = to_categorical(y_test, dtype="int32")
>>> # Show the shape of the training and testing sets
... print("x_train shape:", x_train.shape, "y_train shape:", y_train.sh
... print("x_test shape:", x_test.shape, "y_test shape:", y_test.shape)
x_train shape: (12943, 4) y_train shape: (12943, 2)
x_test shape: (3236, 4) y_test shape: (3236, 2)
```

Apply methods: monitor training with custom metrics

```
>>> from hml.methods import CutAndCount, BoostedDecisionTree, ToyMLP
>>> from keras.losses import CategoricalCrossentropy
>>> from keras.metrics import CategoricalAccuracy
>>> from hml.metrics import MaxSignificance, RejectionAtEfficiency
```

- `MaxSignificance` is a custom metric that calculates the maximum significance under uniform distributed thresholds.
- `RejectionAtEfficiency` is a custom metric that calculates the background rejection at a given signal efficiency.

Apply methods: train the models

```
>>> m1 = BoostedDecisionTree(n_estimators=10)
>>> m2 = CutAndCount()
>>> m3 = ToyMLP(input_shape=(x_train.shape[1],))

>>> m1.compile(
...     metrics=[
...         CategoricalAccuracy(name="acc"),
...         MaxSignificance(name="max_sig"),
...         RejectionAtEfficiency(name="r50"),
...     ]
... )
>>> m2.compile(...) # Same as m1
>>> m3.compile(...) # Same as m1

>>> _ = m1.fit(x_train, y_train)
>>> _ = m2.fit(x_train, y_train)
>>> _ = m3.fit(x_train, y_train, epochs=10, batch_size=256, verbose=2)
```

Apply methods: train the models

```
# m1: BoostedDecisionTree
Iter 1/10 - loss: 1.2112 - acc: 0.8960 - max_sig: 72.5789 - r50: 187.94
Iter 2/10 - loss: 1.0756 - acc: 0.9256 - max_sig: 103.2460 - r50: 158.7
Iter 3/10 - loss: 0.9628 - acc: 0.9376 - max_sig: 126.6023 - r50: 824.0
...
# m2: CutAndCount
Cut 1/4 - loss: 1.9140 - acc: 0.8812 - max_sig: 66.1910 - r50: 8.5635
Cut 2/4 - loss: 2.1843 - acc: 0.8729 - max_sig: 91.5682 - r50: 16.6093
Cut 3/4 - loss: 3.7870 - acc: 0.8369 - max_sig: 105.4932 - r50: 24.9139
Cut 4/4 - loss: 4.3486 - acc: 0.8102 - max_sig: 115.8835 - r50: 33.2185
# m3: ToyMLP
Epoch 1/10
51/51 - 4s - loss: 0.7794 - acc: 0.8726 - max_sig: 66.8790 - r50: 31.88
Epoch 2/10
51/51 - 1s - loss: 0.5577 - acc: 0.8940 - max_sig: 69.6916 - r50: 73.62
```

These training processes can also be obtained as the return value of the `fit` method.

Apply methods: show the results

```
>>> from tabulate import tabulate
>>> results1 = m1.evaluate(x_test, y_test)
>>> results2 = m2.evaluate(x_test, y_test)
>>> results3 = m3.evaluate(x_test, y_test)
>>> results = {}

>>> results['name'] = [m1.name, m2.name, m3.name]
>>> for k in results1.keys():
...     results[k] = results1[k] + results2[k] + results3[k]

>>> print(tabulate(results, headers="keys", floatfmt=".4f"))
name                      loss     acc     max_sig       r50
---------------------  ------  ------  ----------  --------
boosted_decision_tree  0.2529  0.9615    238.2441  882.6083
cut_and_count          4.3782  0.8011    120.5472   37.4138
toy_mlp                0.1659  0.9475     35.9336  163.9975
```

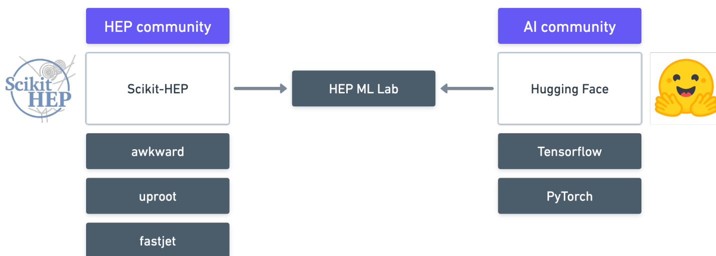
Table of Contents

1 Introduction: why we need an end-to-end framework?

2 Three core parts: generate events, create datasets, apply methods

3 Future: share and cooperate

Future: expectations



From the HEP side, we hope to further use the existing tools of scikit-hep to enhance the efficiency and robustness of the framework;

From the machine learning community's perspective, we embrace the API design of Keras and the open-sharing philosophy of Hugging Face;

By combining both, we aim to make research in the high-energy community also open, reusable, and reproducible.

Future: the next steps

The HEP ML Lab is still under development. Its first version v0.1.0 is just a scaffold of the framework. v0.2.0 is about to release in the next few days.

We are currently working on the following features:

- More observables;
- Image and Graph representations;
- More available networks;
- CLI support;
- ...

Please follow and support us.
`https://github.com/Star9daisy/hep-ml-lab`
`https://pypi.org/project/hep-ml-lab`