HEP ML LAB: An end-to-end framework for applying machine learning to phenomenology studies*

Jing Li (李靖)† Hao Sun (孙昊)

Department of Physics, Dalian University of Technology, Dalian 116024, China

Abstract: Recent years have seen the development and growth of machine learning in high-energy physics. However, additional effort is required to continue exploring the use of machine learning to its full potential. To simplify the application of the existing algorithms and neural networks and to advance the reproducibility of the analysis, we developed HEP ML LAB (`hml`), a Python-based, end-to-end framework for phenomenology studies. It covers the complete workflow from event generation to performance evaluation, and provides a consistent style of use for different approaches. We propose an observable naming convention to streamline the data extraction and conversion processes. In the KERAS style, we provide the traditional cut-and-count and boosted decision trees together with neural networks. We take the W^+ tagging as an example and evaluate all built-in approaches with the metrics of significance and background rejection. With its modular design, HEP ML LAB is easy to extend and customize, and can be used as a tool for both beginners and experienced researchers.

Keywords: framework, machine learning, phenomenological research, Jet, new physics

DOI: 10.1088/1674-1137/addcc9 **CSTR:** 32044.14.ChinesePhysicsC.49093106

I. INTRODUCTION

In recent years, the continuous accumulation of data from the Large Hadron Collider experiments has intensified the demand for new developments in physics. Because of their outstanding capabilities in data analysis and pattern recognition, machine learning techniques have been widely researched and applied in high-energy physics, such as jet tagging tasks [1−34] and rapid generation of simulated events [35−40]. Additional applications are discussed in a review paper on this topic [41].

Typically, the process of research involving machine learning models in high-energy physics comprises four steps: data generation, dataset construction, model training, and performance evaluation. In this process, cooperation between various types of software is often required. For instance, MADGRAPH5_AMC [42] is used for generating simulated events, PYTHIA8 [43] for simulating parton showering, DELPHES [44] for fast simulation of detector effects, ROOT [45] for data processing, and deep learning frameworks such as PYTORCH [46] and TENSORFLOW [47] for subsequently building the neural networks. Researchers new to high-energy physics find it challenging to learn and use these software tools, whereas experienced researchers find it tedious to switch between different types of software. Such a process inevitably increases the complexity of the computational results, which makes them potentially difficult to replicate, leading to difficulties in comparing the results in subsequent research.

Lately, some efforts have been made to simplify the entire process, as follows. PD4ML [48] includes five datasets — Top Tagging Landscape, Smart Background, Spinodal or Not, EoS, and Air Showers — and provides a set of concise application programming interfaces (APIs) for importing them. MLANALYSIS [49] can convert LHE and LHCO files generated by MADGRAPH5_AMC into datasets, and has three built-in machine learning algorithms: isolation forest (IF), nested isolation forest (NIF), and k-means anomaly detection (KMAD). MADMINER [50] offers a complete process for inference tasks [51], and internally encapsulates the necessary simulation software as well as neural networks based on PYTORCH. These frameworks significantly reduce the workload related to specific tasks but have scope for further improvement.

HEP ML LAB, developed in Python, encompasses an end-to-end process. All modules are listed in Fig. 1.

Received 28 March 2025; Accepted 23 May 2025; Published online 24 May 2025

* Hao Sun is supported by the National Natural Science Foundation of China (12075043)

† E-mail: jingliphd@mail.dlut.edu.cn

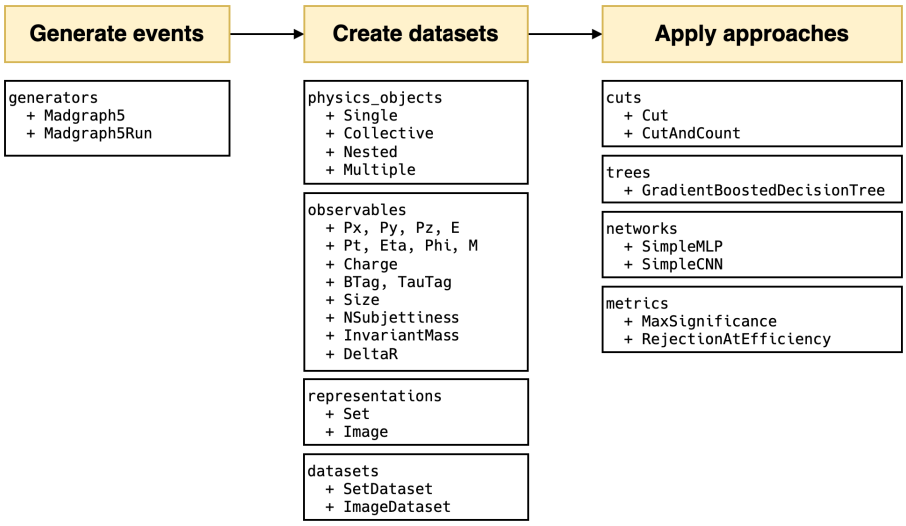


Fig. 1. (color online) All modules in the `hml` framework and main classes in each module.

MADGRAPH5_AMC is minimally encapsulated for event generation, such as defining processes, generating Feynman diagrams, and launching runs. In the transition from events to datasets, we introduced an *observable naming convention* that directly links physical objects with observables, facilitating users to directly use the names of the observables to retrieve the corresponding values. This convention can be further applied to the definitions of cuts. Inspired by the expression form of cuts in UPROOT [52], we expand the corresponding syntax to support filtering at the event level, using veto to define events that need to be removed and define custom observables of greater complexity. This naming rule also applies to the creation of datasets with different representations. In the current version, users can easily create set and image datasets. In addition, we offer a rich set of functions for preprocessing and displaying of images.

In the context of machine learning, we introduce two basic deep learning models: simple multi-layer perceptron and simple convolutional neural network. Both have fewer than ten thousand parameters, providing a baseline for classification performance. These models are implemented using KERAS [53] without any custom modifications, making it easy to expand to other existing models. Additionally, we integrated two traditional approaches, cut-and-count, and gradient boosted decision tree, ensuring compatibility with KERAS. After the different approaches were trained, we provided physics-based evaluation metrics — signal significance and background rejection rate — at a fixed signal efficiency to assess their performance.

This package is publicly available through the Python Package Index (PyPI) and can be installed using the standard pip package manager with the command `pip install hep-ml-lab`. It supports Python 3.9+ and is compatible with Linux, MacOS, and Windows operating systems. The source code is open-sourced on Github[1)].

The structure of the paper is as follows. Sec. II introduces the wrapper class of MADGRAPH5_AMC to generate events. In Sec. III, we describe the observable naming convention and demonstrate step by step its use in extracting data from events as well as its extension to filter data and to the creation of datasets. Three types of approaches — cut and count, decision trees, and currently available neural networks — are discussed in Sec. IV. Physics-inspired metrics are also discussed. In Sec. V, we demonstrate the effectiveness of the framework using a simple and complete W boson tagging as a case study. Finally, we conclude the paper and discuss the scope for future research in Sec. VI.

II. GENERATE EVENTS

All phenomenological studies generally begin by simulating collision events, for example, by using MADGRAPH5_AMC. The `generators` module provides a wrapper for specific parts of its core functionalities, aiming to facilitate its integration into Python scripts for customized setting requirements.

```
from hml.generators import Madgraph5

g = Madgraph5(executive="mg5_aMC", verbose=1)
```

Code example 1: Initialize `Madgraph5`.

In code example 1, users need to pass the executable path to the `executive` parameter to ensure that commands can be sent to it. The `verbose` parameter decides whether to display the intermediate outputs. The default

1) https://github.com/Star9daisy/hep-ml-lab

value is 1, indicating that the intermediate outputs are displayed, consistent with the output obtained when using MADGRAPH5_AMC in the terminal. After initialization, we can use its various methods to simulate commands entered in the terminal, as shown in code example 2.

```
g.import_model(model)
g.define(expression)
g.generate(*processes)
g.display_diagrams(diagram_dir="Diagrams",
    overwrite=True)
g.output(output_dir, overwrite=True)
```

Code example 2: Methods of `Madgraph5` to generate processes.

During the process generation, we first need to use the `import_model` method to import the model file. This method supports passing the path or name of the model (MADGRAPH5_AMC will search for the model in the models folder or download the model based on its name). Next, use the `define` method to define multi-particles, for example, `define "j = j b b~"`. Then, in the `generate` method, pass all the processes to be generated without having to input `add process` as in the case of the terminal. Here, the asterisk represents the unpacking operation in Python, and multiple processes can be directly entered, separated by commas, `g.generate("p p > w+ z", "p p > w- z")` without constructing a list with square brackets. Usually, to confirm that processes have been generated as expected, we need to view the Feynman diagrams, for which the `display_diagrams` method can be used. It saves the generated Feynman diagrams to the `diagram_dir` folder. Prior to this, it converts the default EPS files into the PDF format for convenience. Finally, use the `output` method to export the processes to a folder.

```
g.launch(
    shower="off",
    detector="off",
    madspin="off",
    settings={},
    decays=[],
    cards=[],
    multi_run=1,
    seed=None,
    dry=False,
)
```

Code example 3: Use the `launch` method and set up all possible parameters for generating the events.

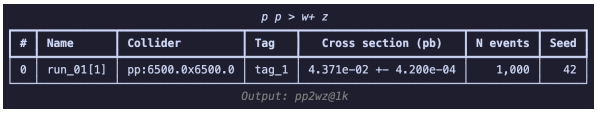
With the process folder ready, we can begin producing runs to generate the simulated events, as shown in code example 3. The `launch` method includes parameters that may need to be configured for the run, where `shower`, `detector`, and `madspin` represent switches for PYTHIA8, DELPHES, and MADSPIN, respectively, consistent with the options in the prompt of the terminal. `settings` includes parameters configured in the run card, for example, `settings = {"nevents":1000, "iseed": 42}`. Furthermore, `iseed` is the random seed used by MADGRAPH5_AMC to control the randomness of the sub-level events. It does not affect PYTHIA8 and DELPHES. You can specify the `seed` parameter to uniformly configure these three, ensuring the cross section, error, and events are fully reproducible. The `decays` method is used to set the decay of the particles; for example, `decays = ["w+ > j j", "z > vl vl~"]`. The `cards` parameter accepts the path to the pre-configured parameter files; for example, `cards=["delphes_card.dat", "pythia8_card.dat"]`. In this version, only Pythia8 and Delphes cards with "pythia8" and "delphes" in their file names can be recognized correctly. It currently does not support the cards that have external folders as dependencies, such as the muon collider delphes card. When numerous events need to be generated, the `multi_run` parameter can be set to create multiple sub-runs for a single run, for example, by setting `multi_run` = 2. The final event files will be named as `run_01_1`, `run_01_2`, which is controled by `MadEvent`. Note that because MADGRAPH5_AMC does not recommend generating more than one million events in a single run, the `nevents` parameter in `settings` should also be set appropriately, as the total number of events is the result of `nevents` multiplied by `multi_run`. `hml` will generate the corresponding valid commands based on the settings and send them to MADGRAPH5_AMC running in the background. To check the actual commands before the beginning of the run, set `dry = True`, which returns the generated commands instead of starting the run.

```
run = g.runs[0]
# run.name
# len(run.sub_runs)
# run.collider
# run.tag
# run.cross
# run.error
# run.n_events
# run.seed
```

Code example 4: All the information in the table can be accessed.

After generating the events, the `summary` method, *i.e.*, `g.summary()` can be used to print the results in a table, as shown in Fig. 2. The table includes the name of each run, number of sub-runs in brackets, colliding particle beam information, tags, cross-section, error, total number of events, and the random seed. The header displays the process information, and the footnote shows the relative path of the output; these are essentially consistent with the results seen on the web page.

To continue experimenting with different parameter combinations, the `launch` method can be used again, or the loop statements in Python can be employed to gener-


p p > w+ z

#	Name	Collider	Tag	Cross section (pb)	N events	Seed
0	run_01[1]	pp:6500.0x6500.0	tag_1	4.371e-02 +- 4.200e-04	1,000	42

Output: pp2wz@1k

Fig. 2. Output of the `summary` method.

ate a series of combinations to observe the differences in the cross-section under various conditions. When doing so, it is recommended to set short label names to facilitate subsequent search and analysis, as in code example 5.

```
for mass in [100, 200, 300]:
    g.launch(
        settings={
            "nevents": 1000,
            "run_tag": f"mass={mass}"},
            "mnh2": mass,
            "wnh2": "auto",
        seed=42,
    )
g.summary()
```

Code example 5: Use a loop to scan the mass of a particle called "nh2" and show the summary.

If output files are already available, `hml` can be used to extract the necessary information for subsequent use. The class method `Madgraph5.from_output` and `Madgraph5Run` will be of great assistance, as shown in code example 6. The former accepts the path to the output folder, which is the path entered in the `output` command of `MadGraph5`, as well as the path to the executable file. The latter requires the output folder path and name of the run to access information such as the cross section and error. The `events` method enables the retrieval of the paths to all event files under a run, including those of the sub-runs. Currently, it supports only files in the root format. `uproot` can be used to open these files for subsequent processing.

```
import uproot
from hml.generators import Madgraph5,
    Madgraph5Run

g = Madgraph5.from_output(output_dir, executive="
    mg5_aMC")
run = g.runs[0]

# Or
run = Madgraph5Run(output_dir, name)
print(run)
# Madgraph5Run run_01 (1 sub runs):
# - collider: pp:6500.0x6500.0
# - tag: 1k
# - seed: 42
# - cross: 0.04371
# - error: 0.00042
# - n_events: 1000,

# Open the first root file it finds
uproot.open(run.events(file_format="root")[0])
```

Code example 6: Use `Madgraph5.from_output` and `Madgraph5Run` to access the information.

III. CREATE DATASETS

The mass of the leading fat jet, angular distance between the primary and secondary jets, total transverse momentum of all jets, number of electrons, etc., demonstrate that observables are always connected to certain physical objects. Therefore, we propose the following observable naming convention: the name of an observable is a combination of the name of the physical object and type of the observable, connected by a dot, that is, `<physics object>. <observable type>`. In this section, starting from physical objects, we gradually refine this representation, eventually extending it to the acquisition of observables, construction of data representations, and definitions of cuts.

A. Physics objects

Physical objects in DELPHES are stored in different branches and represent a category rather than a specific instance. Considering that the calculation of multiple observables involves different types and numbers of physical objects, often utilizing their fundamental four-momentum information, we have categorized physical objects into four types based on their quantity and category:

1. `Single` physical objects, which precisely refer to a specific physical object. For example:
   – `"jet0"` is the leading jet.
   – `"electron1"` is the secondary electron.

2. `Collective` physical objects, representing a category of physical objects. For example:
   – `"jet"` or `"jet:"` represents all jets.
   – `"electron:2"` represents the first two electrons.

3. `Nested` physical objects, formed by free combinations of single and collective physical objects. It currently supports the combination of "FatJet/Jet" and "Constituents":
   – `"jet.constituents"` represents all constituents of all jets.
   – `"fatjet0.constituents:100"` represents the first 100 constituents of the leading fat jet.

4. `Multiple` physical objects, comprising the previous three types and separated by commas. For example:
   – `"jet0,jet1"` represents the leading and secondary jets.

This naming convention is inspired by the syntax of Python lists. To minimize the input cost for the user, we discard the original requirement to use square brackets for receiving indexes or slices: for single physical objects, the type name is directly connected to the index value; for collective physical objects, a colon is used to separate the start index from the end index, and the type name alone

represents the entire set of objects. The `parse_physics_object` method can be used to obtain the branch and the required index values based on the name of the physical objects, as shown in code example 7. This design makes users focus on the physical objects, rather than on how the corresponding classes should be initialized. In Table 1, we summarize all types of physical objects along with their initialization parameters and provide examples.

```
from hml.physics_objects import
    parse_physics_object

# Single
obj = parse_physics_object("jet0")
# obj.branch: "jet"
# obj.slices: [slice(0, 1)]

# Collective
obj = parse_physics_object("jet:10")
# obj.branch: "jet"
# obj.slices: [slice(None, 10)]

# Nested
obj = parse_physics_object("jet0.constituents:10"
    )
# obj.branch: "jet.constituents"
# obj.slices: [0, slice(None, 10)]

# Multiple
obj = parse_physics_object("jet0,jet1")
# obj.branch: ["jet0", "jet1"]
# obj.slices: [slice(0, 1), slice(1, 2)]
```

Code example 7: Use the `parse_physics_object` method to obtain the branch and slices of physics objects.

In this version, physical objects are merely tools for parsing the user input and do not contain any information about the observables. Unlike other software packages, we strictly separate the acquisition of observables from the physical objects. Physical objects store only the information on the connection between the observables and their data sources, and not the data.

B. Observable

After defining the physical objects, the task of the observables is to extract information from them. In code example 7, we store all useful information from a physical object in `branch` and `slices`: the former refers to the corresponding branch name, and the latter means specific parts of array-like data. The advantage of this is that when encountering certain physical objects, such as the hundredth jet, which does not exist, it returns a list of length zero instead of an error. An empty list will automatically be judged as `False` when applying cuts, thereby being skipped.

Table 2 lists all the currently available observables. To avoid remembering the exact name of an observable, its name is case-insensitive and common aliases are added. For example, `Mass` can be written as `mass` or `m`, and `NSubjettinessRatio` has the alias `taumn`, where the values of `m` and `n` are passed as parameters to the corresponding class. For the transverse momentum, considering the style in different types of software, we have assigned a greater number of aliases for its symbol representation. Moreover, different observables support different types of physical objects. For example, the `Size` observable supports collective physical objects, whereas the `AngularDistance` observable supports all combinations of multi-body objects.

In code example 8, we show how to use such an observable. First, initialize the corresponding observables using the `parse_observable` function from the `observables` module. Then, use the `read` method to extract the values from an event. As the `read` returns the object itself, method chaining can be used to define an observable, directly followed by the reading of an event. Additional information, namely, the observable name, shape, and data type, is added when the observable is printed. Internally, `awkward` [54] is used for manipulating variable-length jagged arrays. The question mark in the data type indicates missing values (`None`). The `var` appearing in the shape indicates inconsistent lengths; for example, each event has a varying number of jets and each jet has a varying number of constituents.

The first dimension of the observable value always represents the number of events, but the shape is generally determined by the related physical objects. For example, the shape of the transverse momentum and other kinematic variables is identical to its physics object. However, this also depends on the computation of the observable. For instance, the shape of the `Size` observable is the number of physics objects and the second dimension is always 1, whereas the shape of `AngularDistance` depends on the type of physical objects: when calculating the distance between all jets and the leading fat jet, we will obtain an array of shape `(n_events, var, 1)`, where `n_events` represents the number of events, `var` represents a variable number of jets, and 1 represents the leading fat jet; when calculating the distance between the

Table 1. Types of physics objects and their examples.

Type	Initialization parameters	Name examples
`Single`	`branch: str, index: int`	`"jet0", "muon0"`
`Collective`	`branch: str, start: int|None`	`"jet", "jet1:", "jet:3", "jet1:3`
`Nested`	`main: str|PhysicsObject, sub: str|PhysicsObject`	`"jet.constituents", "jet0.constituents:100`
`Multiple`	`all: list[str|Physicsobject]`	`"jet0,jet1", "jet0,jet"`

Table 2. Types of observables and their supported types of physical objects.

Type	Alias	Single	Collective	Nested	Multiple
MomentumX, Px	momentum_x, px	✓	✓	✓	
MomentumY, Py	momentum_y, py	✓	✓	✓	
MomentumZ, Pz	momentum_z, pz	✓	✓	✓	
Energy, E	energy, e	✓	✓	✓	
TransverseMomentum, Pt	transverse_momentum, pt, pT, PT	✓	✓	✓	
PseudoRapidity, Eta	pseudo_rapadity, eta	✓	✓	✓	
AzimuthalAngle, Phi	azimuthal_angle, phi	✓	✓	✓	
Mass, M	mass, m	✓	✓	✓	
Charge	charge	✓	✓		
BTag	b_tag	✓	✓		
TauTag	tau_tag	✓	✓		
NSubjettiness, TauN	n_subjettiness,tau_n, taun	✓	✓		
NSubjettinessRatio, TauMN	n_subjettiness_ratio, tau_mn, taumn	✓	✓		
Size	size		✓		
InvariantMass	invariant_mass, inv_mass, inv_m	✓			✓
AngularDistance, DeltaR	angular_distance, delta_r	✓	✓	✓	✓

first ten fat jets and all constituents of all the jets, we obtain an array of shape `(n_events, 10, var)`. The value of `var` now originates from the number of constituents and number of jets; the two dimensions are compressed into one. For events that do not have sufficient physics objects, the missing values are filled with `None`.

```
from hml.observables import parse_observable

obs = parse("jet0.pt")
obs.read(events)
print(obs)
# jet.pt: 100 * 1 * ?float32

obs = parse("jet.pt").read(events))
print(obs)
# jet.pt: 100 * var * float32

obs = parse("jet.constituents.pt")
obs.read(events)
# obs.name: "jet.constituents.pt"
# obs.shape: (100, var, var)
# obs.dtype: float32

obs = parse("jet:10.constituents:100.pt")
# obs.shape: (100, 10, 100)
```

Code example 8: Use `parse_observable` and `read` to obtain the values of the observables. `events` are opened by `uproot`.

The built-in observables are very basic and may not be sufficient for every use case. Therefore, we show three examples of building your own observables. In the first example 9, when the needed observable is already stored under a certain branch, only the name of this observable needs to be declared as a class that inherits from `Observable`. `hml` will search for the branch based on the physical object name and extract the corresponding value based on the slices. Here, we take the observable `MET` as an example, which is originally stored under the branch `MissingET`. To use the `parse_observable` function, the `register_ observable` function can be called to register an alias for it. Please note that this implementation requires a physical object, which means that only by entering `"missinget0.met"` or `"MissingET0.MET"` can the `parse_observables` function normally. Only `"MET"` or `"met"` without a physics object is not allowed. As each event has only one missing energy physical object, `"MissingET"` is followed by 0.

```
from hml.observables import (
    Observable,
    parse_observable,
    register_observable,
)

class MET(Observable): ...

register(MET, "MET", "met")

# Both lower and upper cases are acceptable
obs = parse_observable("missinget0.met")
obs = parse_observable("MissingET0.MET")
```

Code example 9: Inheriting from `Observable` will automatically retrieve the corresponding value if the physics object has it.

If a computation process has been already established and the physical objects need not be considered, we recommend referring to the second example 10. For this, the `read` method should be overwritten by specifying the process for computing the values of the observables. `events` is the return value of `uproot.open`. You may need to adjust the calculation because of `events` and the underlying `awkward` array. It is important to note that `_value` must be an iterable object, such as a list or array, to be correctly converted into an `awkward` array by `hml`. Additionally, `self` should be returned at the end, enabling chain calls similar to those of other observables.

```
class SumPtOfJet0Muon0(Observable):
    def read(self, events):
        value = []
        pts1 = events["Jet.PT"].array()[:, 0:1]
        pts2 = events["Muon.PT"].array()[:, 0:1]

        # Loop over the events (slow)
        for pt1, pt2 in zip(pts1, pts2):
            value.append(sum(list(pt1) + list(pt2))
                )

        # Or manipulate the arrays directly (fast)
        pt1 = ak.pad_none(pts1, 1)[:, 0]
        pt2 = ak.pad_none(pts2, 1)[:, 0]
        pt1 = ak.fill_none(pt1, 0)
        pt2 = ak.fill_none(pt2, 0)

        self._value = value
        return self

register_observable(SumPtOfJet0Muon0, "mine")
obs = parse_observable("mine").read(events)
print(obs)
# mine: 1000 * float64
```

Code example 10: Overwrite the `read` method to specify the process for calculating the value of the observable.

The third example 11 changes the initialization. We add constraints on the physical objects, i.e., it can be related only to a single physics object. In addition, a new parameter is introduced for greater flexibility. The `read` part is the same as that in the second example. This is the strictest observable but also the safest one.

Currently, the naming convention is built upon the output of DELPHES and does not support other formats yet. However, considering that different analyses require data at different levels and in different formats, we plan to gradually add support for other event formats in the future versions, such as HEPMC, LHE, etc.

C. Representation

To make high-energy physics data compatible with different analysis approaches, it is necessary to convert the data into various representations. The review paper [55] summarizes six representations of jets: ordered sets, images, sequences, binary trees, graphs, and unordered sets. Built upon the observable naming convention, we extend the representation to an event. Currently, `hml` supports the (ordered) set and image representations. In future versions, we will prioritize adding the graph representation and the corresponding neural networks.

```
class RestrictObservable(Observable):
    def __init__(self, new_parameter,
         physics_object, class_name=None):
        self.new_parameter = new_parameter
        super().__init__(physics_object,
            class_name, supported_objects = ["
            single"])

    def read(self, events):
        # Calculation of the observable
        return self
```

Code example 11: Define an observable with constraints on the type of physical objects and with a new parameter.

The ordered set is one of the most commonly used representations. It arranges physics-inspired observables in an arbitrary order to form a vector that describes an event. The vectors from all events are then assembled into one matrix by event, with the shape `(n_events, n_observables)`. By following the naming convention, such a set can be constructed in a straightforward and concise manner, as illustrated in code example 12.

```
from hml.representations import Set

r = Set(["jet0.pt", "muon0.charge", "fatjet0.mass
    "])
r.read(events)
# r.names
# ['jet0.pt', 'muon0.charge', 'fatjet0.mass']
# r.values
# [[189, None, 112],
# [229, None, 65.6],
# ...,
# [251, None, 63.5],
# [311, None, 58.1]]
# ------------------------
# type: 1000 * 3 * ?float64
```

Code example 12: Use `Set` to represent the ordered set of observables.

The observable names must be packaged into a list and passed to the `Set`. Next, the `read` method must be called to obtain the values from the events. The values will be stored in the `values` attribute. Here, it can be seen that the `awkward` arrays are used to store the data. For observables that have the correct physical object name but do not exist (for example, `muon0.charge`, when there are no muons produced in the event), the value is set to `None`. This approach of handling missing values allows us to follow the matrix operation sequence: first build the data matrix, then deal with the missing values.

For the image representation, the observable names are used to specify the method for fetching the data for its height, width, and channel, as shown in code example 13. The `read` method is used to read events as before. Here, we construct an image of the leading fat jet, along with the pseudorapidity, azimuthal angle, and transverse momentum of all its constituents. Considering that similar preprocessing processes have been employed in a large number of studies, we add them as the methods of the `Image` class. Because preprocessing often relates to sub-jets, it is necessary to add the information on sub-jets via the method `with_subjets`; its parameters include the name of the constituents, clustering algorithm, radius, and minimum momentum of the jet. The `translate` method moves the position of the leading sub-jet to the origin, which reduces the complexity of the position information and expedites the learning process. Next, the position of the sub-leading sub-jet can be rotated right below the origin, making the features of the entire image more pronounced. Lastly, the `pixelate` method is used to pixelate the data to obtain a real image. Because pixelation reduces the data precision, this step is removed. Further studies are needed to determine when this method should be applied and the effect of the order of its application.

```
from hml.representations import Image

r = Image(
    height="fatjet0.constituents.phi",
    width="fatjet0.constituents.eta",
    channel="fatjet0.constituents.pt",
)
r.read(events)
r.with_subjets("fatjet0.constituents", "kt", 0.3,
     0)
r.translate(origin="SubJet0")
r.rotate(axis="SubJet1", orientation=-90)
r.pixelate(size=(33, 33), range=[(-1.6, 1.6),
    (-1.6, 1.6)])
```

Code example 13: Use `Image` to represent a fat jet and preprocess it via sub-jets.

For convenience in displaying the images, the `Image` class contains the `show` method, which can directly plot it as an image. Code example 14 shows all the available parameters: the first two are used to show the image as dots; the last three parameters display a pixel-level grid, enable the grid by default, and apply normalization over the entire image, respectively. Figure 3 shows the image representations before and after the preprocessing steps. In the raw image, the observables used for "height" and "width" are directly plotted as a 2D scatter image. In addition, `norm="log"` enhances the features of the final pixelated image to make them more distinct. The data can be accessed via the `.values` property as a list of awkward arrays (before pixelation) or one awkward array (after pixelation). These can be converted and saved in formats such as a numpy array and JSON files to allow for their handling with common tools.

```
r.show(
    as_point=False,
    limits=None,
    show_pixels=False,
    grid=True,
    norm=None,
)
```

Code example 14: Use `show` to plot the image as a 2d heatmap if it has been pixelated or as a 2d scatter plot.

After acquisition, the original event data are filtered to obtain events that satisfy specific criteria. In the previous workflow, during the event loop, it was common to manually include the calculation of the observables and then apply conditionals to filter the events. We note that the (array) method in `uproot` supports the `cut` parameter. In `hml`, we utilize a matrix-oriented programming style to change the filtering procedure into Boolean indexing; furthermore, we add the logical operation syntax to the observable naming convention to make the definitions of cuts intuitive, as shown in code example 15. `Cut` continues to have a similar `read` method. The values form a one-dimensional Boolean matrix, the length of which is equal to the number of events, allowing its direct use to filter other observables via Boolean indexing.

For the extend syntax of the logical operations, *i.e.*,

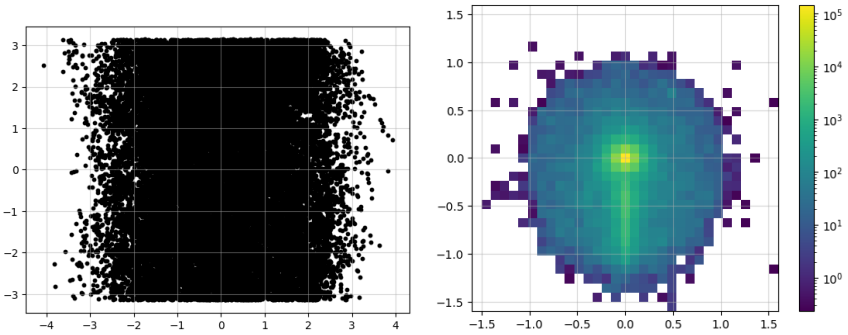


Fig. 3. (color online) Raw image and pixelated image after preprocessing.

how to combine multiple conditions, we refer to the implementation of `uproot`: it uses the bitwise logical operators of the matrices, & and |, to replace `and` and`or`, respectively, and adds parentheses to ensure priority. For example, `(pt1 > 50) & ((E1>100) | (E1<90))` expresses the condition that `pt1` is greater than 50 and `E1` is either greater than 100 or less than 90. The expression is then parsed directly by Python: it is purely a matrix operation, without considering the case of the DELPHES output. It cannot handle cuts such as "all jets are required to have a transverse momentum greater than 10 GeV," where the data have the shape `n_events, var`. It undoubtedly requires users to rearrange the original data to make the dimensions of the matrices consistent, as the number of jets is not necessarily the same among the events. This essentially filters only the values that meet the conditions, not the events that meet the conditions. In conjunction with the observable naming convention, we advocate simplifying the matrix logical operation syntax to facilitate the user input.

```
import uproot
from hml.approaches import Cut

tree = uproot.open("path/to/file.root")
events = tree["Delphes"]

cut = Cut("jet.size > 1 and fatjet.size > 0")
cut.read(events)
# cut.value
#[True,
# True,
# ...
# True,
# True]
# -----------------
# type: 1000 * bool

# Mask an observable
# obs.value[cut.value]
```

Code example 15: Cut values are Boolean arrays that can be used to filter other observables.

1. Logical AND and logical OR are represented by `and` and `or`, respectively. They are automatically converted to bitwise logical operators & and |, respectively, avoiding too many parentheses.

2. The result of the logical expression acts on the first dimension, that is, the dimension of the events, to filter the events.

3. The involved observables must have the same dimensions to ensure the correctness of the logical operations.

4. At the end of a cut, default `[all]` represents a logical AND operation on all values of all observables in each event. It can be ignored and need not be written; `[any]` represents a logical OR operation on all values of all observables in each event. This syntax is suitable for cases where all of a certain observable or any one observable in the events must meet the conditions.

5. Add support for `veto` at the beginning of a cut for cases where certain events need to be excluded.

Below, we demonstrate and explain the new syntax by referring to literature. The original text will be presented first, followed by the corresponding cut in the next line. We assume that the data are stored in the same units as indicated in the description. In [56]:

1. Muons $\mu^{\pm}$ are identified with a minimum transverse momentum $p_T^{\mu} > 10\,\text{GeV}$ and rapidity range $|\eta^{\mu}| < 2.4$...

```
muon.pt > = 10 and -2.4 < muon.eta < 2.4
```

Here, we take all the muons and simplify the syntax for pseudorapidity within a certain range, which can be written consecutively. Here, `and` represents the bitwise logical operator of the matrix. For each event, if `[any]` is absent at the end of the expression, all values must satisfy the condition;

2. Only events with reconstructed di-muons having the same sign are selected.

```
muon0.charge = = muon1.charge
```

Here, we only need to judge whether the charges of the two muons are the same, without determining whether the number of muons is two. When there are fewer than two muons, the charge of one muon will be `None`, and such a judgment will be automatically treated as `False`;

3. We identify the hardest fat-jet with the $W^{\pm}$ candidate jet (J), which is required to have $p_T^J > 100\,\text{GeV}$.

```
fatjet0.pt > = 100
```

In [57]:

1. C3: we veto the events if the OS di-muon invariant mass is less than 200 GeV.

```
muon0.charge ! = muon1.charge and muon0,muon1.inv
_mass > 200
```

2. C4: we apply a b-veto.

```
veto jet.b_tag = = 1 [any]
```

Use `veto` and `[any]` to indicate that for all jets, if any jet is b-tagged, then the event is excluded.

3. C5: we consider only events with a maximum MET of 60 GeV.

```
MissingET0.PT < 60
```

The case used (uppercase or lowercase) is inconsequential because there is only one missing energy; hence, it is represented by 0. The MET observable in this case also refers to the transverse momentum; hence, it can be represented by PT.

4. C8: we choose events with N-subjettiness $\tau_{21}^{J_0} < 0.4$.

```
fatjet0.tau21 < 0.4
```

The definition of this observable is provided by the DELPHES card. We have already defined its parsing method in the observables module; therefore, its name can be directly used here.

In [58], the cuts of CBA-I for $M_N = 600 - 900$ GeV:

1. Jet-lepton separation $2.8 < \Delta R(j,\ell) < 3.8$

```
2.8 < jet0,lepton0.delta__mass > 200r < 3.8
```

In [59]:

1. The basic selections in our signal region require a lepton (e or μ) with $|\eta_\ell| < 2.5$ and $p_{T,\ell} > 500$ GeV...

```
-2.5 < muon0.eta < 2.5 and muon0.pt > 500.
```

We take the example of the muon.

2. ... veto events that contain additional leptons with $|\eta_\ell| < 2.5$ and $p_{T,\ell} > 7$GeV

```
veto -2.5 < muon.eta < 2.5 and muon.pt >
7 [any]
```

In this cut, it is easier to use `veto` to exclude events with additional leptons.

3. ... and impose a jet veto on the subleading jets with $|\eta_j| < 2.5$ and $p_{T,j} > 30$GeV.

```
veto -2.5 < jet1.eta < 2.5 and jet1.pt >
30 [any]
```

In [60]:

1. Photon-veto: Events having any photon with $p_T > 15$ GeV in the central region, $|\eta| < 2.5$, are discarded.

```
veto -2.5 < photon.eta < 2.5 and photon.pt
> 15 [any]
```

2. τ and b-veto: No tau-tagged jets in $|\eta| < 2.3$ with $p_T > 18$ GeV and no b-tagged jets in $|\eta| < 2.5$ with $p_T > 20$ GeV are allowed.

```
veto jet.tau_tag = = 1 and -2.3 < jet.eta <
2.3 and jet.pt > 18 [any]
```

3. Alignment of MET with respect to the jet directions: Azimuthal angle separation between the reconstructed jet with the MET to satisfy $\min(\Delta\phi(\mathbf{p}_T^{\rm MET}, \mathbf{p}_T^j)) > 0.5$ for up to four leading jets with $p_T > 30$ GeV and $|\eta| < 4.7$.

```
jet:4,missinget0.min_delta_phi > 0.5 and
jet:4.pt >30 and -4.7 < jet:4.eta < 4.7
```

Users need to define the observable `MinDeltaPhi` in advance and register it with the alias `min_delta_phi`.

D. Dataset

With the previously defined data representation and cuts, we can now proceed to construct the dataset. Corresponding to the data representations, we currently offer twodatasets:`SetDataset`and`ImageDataset`.Codeexample 16 shows the use of the dataset of an ordered set. Its initialization requires the names of the observables. Next, use the `read` method to read the events. For this `read`, we introduced two additional parameters: `targets` is the integer label assigned to the event, which is the target of the convergence. Here, we assign 1 to denote the events as signals; `cuts` are the filtering criteria. Here, we require the number of jets to be greater than 1, and the number of leading fat jets to be greater than 0. When multiple cuts are used, the result of each cut is applied to the dataset sequentially, which means they are connected by a logical AND. The `split` method can be used to split the dataset. Its parameters are the ratios for the training, test, and validation sets. Here, we used 70% of the data as the training set, 20% as the test set, and 10% as the validation set. Before saving the dataset, `samples` and `targets` can be accessed to view the stored data, which have already been converted into `numpy` arrays. Finally, use the `save` method to save the dataset to a zip compressed file with the `.ds` suffix. Such a file can be loaded by the `load_dataset` function or `SetDataset.load` class method.

The `show` method can be used to quickly view the distributions of the entire dataset. Code example 17 shows all the available parameters: `n_feature_per_line` is the number of observables to display per line, `n_samples` is the number of events to display, and `target` is the label of the events to display.

```
from hml.datasets import SetDataset, load_dataset

cuts = ["jet.size > 1 and fatjet.size > 0"]
set_ds = SetDataset(["jet0.pt", "jet1.pt", "
    fatjet0.mass"])

set_ds.read(events, targets=1, cuts=cuts)
set_ds.split(train=0.7, test=0.2, val=0.1)
set_ds.save("my_set_dataset.ds")
```

Code example 16: Use `SetDataset` to build a dataset representing each event as a set of observables.

```
set_ds.show(
    n_feature_per_line=3,
    n_samples=-1,
    target=None,
)
```

Code example 17: Use `show` to display the distribution of observables of a set dataset.

The construction process of an image dataset is similar to that of an ordered set, as shown in code example 18. When initializing an `Image`, you can directly configure the necessary preprocessing steps in method chaining. When the dataset reads the events later, these steps will be performed in sequence.

```
from hml.representations import Image
from hml.datasets import ImageDataset

cuts = ["jet.size > 1 and fatjet.size > 0"]
image = (
    Image(
        height="fatjet0.constituents.phi",
        width="fatjet0.constituents.eta",
        channel="fatjet0.constituents.pt",
    )
    .with_subjets("fatjet0.constituents", "kt",
        0.3, 0)
    .translate(origin="SubJet0")
    .rotate(axis="SubJet1", orientation=-90)
    .pixelate(size=(33, 33), range=[(-1.6, 1.6),
        (-1.6, 1.6)])
)
ds = ImageDataset(image)

ds.read(events, targets=1, cuts=cuts)
ds.split(train=0.7, test=0.2, val=0.1)
ds.save("my_dataset.ds")
```

Code example 18: Use `ImageDataset` to build a dataset representing each event as an image

The `show` method of an image dataset can be used to display the events as an image, as shown in Example 19. By default, the images of the entire dataset are compressed into one image. If the original dimensions are `n_events,height, width, channel`, the compressed dimensions will be `height, width, channel`, respectively. Most parameters are the same as those in the `show` method of `Image` with the exception of two additional parameters: `n_samples` representing the number of events to display, and `target` representing the label of the events to display.

```
ds.show(
    limits=None,
    show_pixels=False,
    grid=True,
    norm=None,
    n_samples=-1,
    target=None,
)
```

Code example 19: Use `show` to plot the image of the dataset.

IV. APPLICATION OF THE APPROACHES

With well-prepared datasets, we can apply different approaches to identify rare new physics signals. The `approaches` module includes cut-and-count, trees, and neural networks. We will gradually enhance it in the future versions. The basic design principle of this module is ensuring minimal encapsulation to interface with current frameworks, such as SCIKIT-LEARN [61], TENSORFLOW, and PYTORCH. Considering simplicity, we adopted the Keras-style interface design: decide approach structure at initialization, `compile` for configuring the training process, `fit` for training the approach, and `predict` for prediction using new data. KERAS was originally a high-level encapsulation of TENSORFLOW. However, after the release of version 3, it supports multiple backends, thus offering unprecedented flexibility, which is one of the reasons for choosing it. Note that to date, we have tested only its compatibility with the TensorFlow backend.

A. CutandCount

Cut-and-count (or cut-based analysis) is fundamental and widely used when studying the effect of various observables on the final sensitivity. It provides evidence to support the discovery of new particles and verification of new theories.

As the name suggests, it involves two steps: applying a series of cuts to distinguish the signal from the background as much as possible, followed by counting the number of events that pass the cuts. The subsequent distribution, such as an invariant mass, is used to determine the nature of the particles involved. These cuts can be applied to the properties of specific particles, such as kinematic quantities, charge, other observables, or other characteristics associated with simulated collision events, such as the particle states in decay chains. Filtering the data allows focusing on the areas of interest, increasing

the possibility of discovering new physics.

Applying cuts requires a technique to make the final signal more evident. Typically, the distribution of the observables that reflect the signal characteristics would be plotted and the area with a higher signal ratio would be selected as the cut range based on manual judgment. There are two issues here: 1) manual judgment is subjective and cannot guarantee the effect of the cut; 2) the observable distributions observed by users are sometimes the source data without any cuts. If there is an unavoidable association between the observables, applying a cut will affect the distribution of the next observable. Therefore, a more rigorous method is to apply one cut first, plot the distribution of the next observable, and then determine the next cut.

Users can use `CutAndCount` to implement these two different strategies of applying cuts. In code example 20, we demonstrate how to initialize a `CutAndCount` method. The number of involved observables must be specified. Subsequently, an internal `CutLayer` is created to automatically search for the optimal cut values. For each observable, four possible conditions are considered: the signals on the left side, right side, middle, and both sides of the cut. Then, the user-specified loss function for each case is calculated, and the one with the minimum loss is selected as the final cut. `n_bins` sets the granularity of the data when searching, *i.e.*, the number of bins for the distribution of each observable. A higher number of bins can make the cut more precise but increases the cost of calculation, which also affects the data size and complexity of its distribution. The principle here is that as long as the data distributions remain stable, the number of bins can be appropriately reduced without affecting the final result. `topology` sets the order or strategy of applying the cuts: `parallel` indicates that all cuts are independent, and the distributions are considered to originate from the original data, whereas `sequential` considers the correlation among the cuts, with each cut applied on the basis of the previous one.

```
from hml.approaches import CutAndCount

approach = CutAndCount(
    n_observables: int,
    n_bins: int = 50,
    topology="parallel", # or "sequential"
)
```

Code example 20: Initialize the CutAndCount approach

In code example 21, we show how to configure and train the `CutAndCount` approach. In the `compile` method, you can specify the optimizer, loss function, and evaluation metrics. The optimizer is unnecessary for `CutAndCount` because it does not use the gradient descent methods internally but rather finds the optimal cut values through a search process. The loss function is used to evaluate the effectiveness of each cut, whereas the evaluation metrics will be used to reveal the performance scores during the training. It is better to evaluate the performances simultaneously after the training. The setting `run_eagerly = True` is necessary. By default, KERAS uses a computational graph for the calculations, which is very efficient for training neural networks. However, `CutAndCount` includes some custom calculations that are not yet fully compatible within the computational graph; hence, it needs to be set to `True`. In the `fit` method, the samples and targets of the training set must be input, where the batch size should be the minimum number that can reflect the distribution pattern of the data. If the user dataset is relatively small and can fit entirely into the GPU memory, the batch size can be set to the size of the entire training set. Moreover, the `epochs` parameter is unnecessary, and the `callbacks` parameter has not been implemented yet but will be gradually added in future versions.

```
approach.compile(
    optimizer=None,
    loss="crossentropy",
    metrics=["accuracy"],
    run_eagerly=True,
)

approach.fit(
    dataset.train.samples,
    dataset.train.targets,
    batch_size=len(dataset.train.samples),
)
```

Code example 21: Configure and train the `CutAndCount` approach

B. Trees and neural networks

Decision trees are a common method of classification, and several mature frameworks are available, such as TMVA [62], XGBOOST [63], and SCIKIT-LEARN. TENSORFLOW DECISION FORESTS [64] is also a good choice, as it too adopts the KERAS training style. Considering our preference for the multi-backend support, we modify parts of the `GradientBoostingClassifier` code from SCIKIT-LEARN to conform to the same style.

Firstly, the original `fit` method is enhanced to handle the input targets in a one-hot encoded format. Secondly, support for the `Keras` metrics, such as the commonly used `"accuracy"`, is provided during the training process. Thirdly, the output of its `predict` method is changed to predict the probabilities, aligning it with `Keras`. Despite these changes, many parameters are not supported yet: many of the original initialization parameters are related to early stopping, learning rate adjust-

ments, etc., which are implemented in `Keras`through the callback functions. Moreover, loss functions are not uniformly customizable in `scikit-learn`; hence, we do not support changing it in the `compile` method. In code example 22, we demonstrate the basic usage.

A starting point of `hml` is to provide researchers with existing deep learning models so that they can conduct benchmark tests on their datasets and select the optimal model. At this early development stage, we offer only two basic models: `SimpleMLP` (multi-layer perceptron) and `SimpleCNN` (convolutional neural network). In future versions, after thorough testing, we will gradually introduce additional existing models to provide a larger range of selection.

```
from hml.approaches import
    GradientBoostedDecisionTree

bdt = GradientBoostedDecisionTree(...)
bdt.compile(
    # optimizer=None,
    # loss="crossentropy",
    metrics=["accuracy"],
)
bdt.fit(
    x_train,
    y_train,
    # callbacks
)
```

Code example 22: Basic usage of the modified `GradientBoostedDecisionTree` approach.

In KERAS, a model can be built in three ways: 1. Using `Sequential` to stack the layers, 2. Using the *Functional* API to construct more complex topologies, 3. Inheriting from `Model`to declare a subclass for greater flexibility. Considering that the construction of many models is highly complex and requires exposing a certain number of hyper-parameters for tuning, we used the third approach to build the models. Fig. 4 shows the structures of the two models.`SimpleMLP` has 4386 parameters and the inputs are three observables. `SimpleCNN` has 5960 parameters and the inputs are images of the shape $(33, 33, 1)$. Both models are shallow and simple, resulting in low consumption of computing resources during the training and testing stages.

C. Metrics

After training an approach on a dataset, it is often necessary to use various metrics to assess its effectiveness. Unlike the classical accuracy score, which is commonly used in classification tasks, in high-energy physics, the scarcity of signals shifts the focus toward the signal significance, denoted by σ. A higher value indicates a lower probability that the observed signal is a result of background fluctuations alone. For instance, 3σ is often considered as an evidence of a signal, indicating that there is

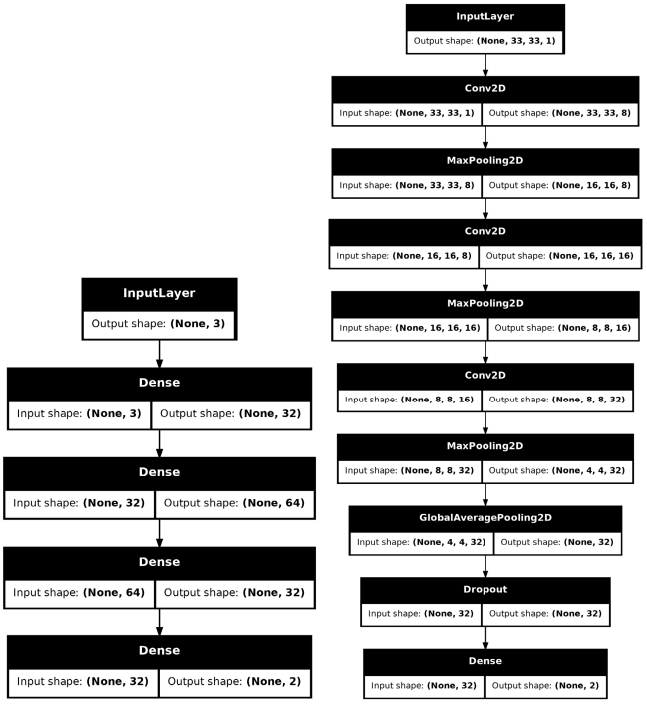


Fig. 4. Structures of the `SimpleMLP` and `SimpleCNN` models.

approximately a 0.27% chance of the signal being a statistical fluke. Further, 5σ is the gold standard in high-energy physics for claiming a discovery, corresponding to a probability of roughly 1 in 3.5 million that the observed signal is due to background noise. Equation 1 is the formula for calculating the significance in `hml`. Note that S represents the number of signals, and B represents the number of backgrounds, which indicates the number of simulated events when the cross section of the corresponding process and integrated luminosity are not specified. We demonstrate its use in code example 23.

$$\sigma = \frac{S}{\sqrt{S+B}}. \tag{1}$$

```
from hml.metrics import MaxSignificance

metric = MaxSignificance(
    cross_sections=[1, 1],
    luminosities=[1, 1],
    thresholds=None,
    class_id=1,
)

metric(y_true, y_test)
```

Code example 23: Use the `MaxSignificance` metric to evaluate the performance of a trained approach.

The outputs of the built-in approaches are the probabilities for the signal and background. By default, only when the probability exceeds 0.5 do we consider it as a

signal or background. In `MaxSignificance`, this threshold can be changed by setting `thresholds`. By default, data with `class_id = 1` are viewed as the signal and 0 as the background. This can be changed if the targets for the signal and background in the user dataset are different.

In addition to significance, some studies have also included the background rejection at a fixed signal efficiency as an evaluation metric. Hence, this metric was adopted in our study as well and is expressed as equation 2. The higher the background rejection rate, the fewer are the background events that are mistakenly classified as signals. Example 24 demonstrates its use. Note that multiple calls to both the metrics will result in the values being averaged. Therefore, the `reset_state` method should be called to calculate them from scratch.

$$\text{rejection} = 1/\left.\epsilon_b\right|_{\epsilon_s} . \tag{2}$$

```
from hml.metrics import RejectionAtEfficiency

metric = RejectionAtEfficiency(
    efficiency,
    num_thresholds=200,
    class_id=None,
)

metric(y_true, y_test)
```

Code example 24: Use the `RejectionAtEfficiency` metric to evaluate the performance of the model.

V. EXAMPLE: W BOSON TAGGING

To enable users to gain a complete understanding of the entire workflow, this section shows how to integrate various modules to complete the task of jet tagging. This example serves merely as a proof of concept; users must conduct a more personalized analysis on this basis.

A. Step 1: generate events

We simulated the production of highly boosted W^+ bosons that decay into two jets, resulting in a single fat jet during event reconstruction. This jet has distinct characteristics of mass and spatial distributions, making it easier to identify using all built-in approaches.

In code example 25, we first import the event generator module from `Madgraph5` using the `generate` method to create the signal process, and then use the `output` method to save it to a designated folder. To make W^+ bosons highly boosted, we leave the decay chain unfinished here and constrain its p_T range in code example 26. After

p p > w+ z

#	Name	Collider	Tag	Cross section (pb)	N events	Seed
0	run_01[1]	pp:6500.0x6500.0	tag_1	4.375e-02 +- 1.400e-04	10,000	42

Output: data/pp2wz

p p > j j

#	Name	Collider	Tag	Cross section (pb)	N events	Seed
0	run_01[0]	pp:6500.0x6500.0	tag_1	1.161e+04 +- 4.000e+01	10,000	42

Output: data/pp2jj

Fig. 5. Summary table of the signal (upper) and background (lower) events.

the `output` command is completed, the output Feynman diagrams can be viewed in the `Diagrams` folder within the output folder.

Then, the `launch` method in code example 26 is used to start the simulation, turning on the `shower` and `detector`. To boost the W^+ boson, we set the spin mode as "none" to apply the following cuts[1)] in the settings. Following [3], we set the p_T range for the W^+ boson to 250 to 350 GeV. When using the default CMS delphes card with $R = 0.8$ and the anti-k_T algorithm to cluster the jets, a fat jet is expected to originate from the decay of the W^+ boson. The `decay` method is used for further specific decays. The random seed (`seed`) is set to 42 to ensure the reproducibility of the results. When the simulation ends, the `summary` method is used to review the results (shown in Fig. 5). This is akin to viewing the results on the website of `Madgraph5`.

```
from hml.generators import Madgraph5

sig = Madgraph5(executable="mg5_aMC", verbose=0)
sig.generate("p p > w+ z")
sig.output("data/pp2wz")
```

Code example 25: Generate W^+ boson events using `Madgraph5`

```
sig.launch(
    shower="pythia8",
    detector="delphes",
    madspin="none",
    settings={
        "nevents": 10000,
        "pt_min_pdg": "{24: 250}",
        "pt_max_pdg": "{24: 300}",
    },
    decays=["w+ > j j", "z > vl vl~"],
    seed=42,
)

sig.summary()
```

Code example 26: Launch the simulation of W^+ boson events.

The generation process for the background events is

1) https://answers.launchpad.net/mg5amcnlo/+question/666825

similar to that in code example 27, with the difference lying in the p_T range settings. Because the jets in this case do not originate from the decay of a single particle, we directly restrict the p_T range of the jets.

```
bkg = Madgraph5(executable="mg5_aMC", verbose=0)
bkg.generate("p p > j j")
bkg.output("data/pp2jj")
bkg.launch(
    shower="pythia8",
    detector="delphes",
    settings={
        "nevents": 10000,
        "ptj": 250,
        "ptjmax": 300,
    },
    seed=42,
)

bkg.summary()
```

Code example 27: Generate background events using `Madgraph5`

After the event generation is complete, we begin preparing the dataset. First, in code example 28, we use `uproot` to open the root file output by DELPHES, which stores the branches categorized according to the physical objects. The `generators` module includes `Madgraph5Run`, which conveniently retrieves information about the run, such as the cross section and generated event files. Because it searches for files produced in all sub-runs of a given run, even though the files for the signal events are stored in the sub-runs, `run_01` can also retrieve the corresponding path correctly.

```
import uproot

from hml.generators import Madgraph5Run

sig_run = Madgraph5Run("./data/pp2wz", "run_01")
bkg_run = Madgraph5Run("./data/pp2jj", "run_01")

sig_events = uproot.open(sig_run.events()[0])
bkg_events = uproot.open(bkg_run.events()[0])
```

Code example 28: Use `uproot` to open the DELPHES output root file.

In code example 29, to avoid missing values of the desired observables, we use the previously mentioned extended logical operations to apply the cuts. For both types of datasets, that is, `SetDataset` and `ImageDataset`, it is required to have at least one fat jet and two regular jets. The `read` method supports entering multiple cuts. Thus, there is no need to use the `Cut` class to parse the expressions first. When reading the events, integer labels are assigned separately for the signal and background events. Before saving locally, the data are split into training and testing sets at a 7:3 ratio.

```
from hml.datasets import SetDataset

cut = "fatjet.size > 0 and jet.size > 1"
set_ds = SetDataset(
    [
        "fatjet0.mass",
        "fatjet0.tau21",
        "jet0,jet1.delta_r",
    ]
)
set_ds.read(sig_events, 1, [cut])
set_ds.read(bkg_events, 0, [cut])

set_ds.split(0.7, 0.3)
set_ds.save("./data/wjj_vs_qcd_set.ds")
```

Code example 29: Prepare the set dataset.

For the image dataset, the representation of the data is first decided, namely, which observables should constitute the images and which preprocessing steps should be taken. In code example 30, ϕ, η, and p_T of all constituents from the leading fat jet are used as the data source for the height, width, and channel of an image. Before preprocessing, `with_subjets` is used to recluster the constituents to add information about the subjets. Because the distance between the two sub-jets will not be less than 0.3 according to the previous equation, it is safe to use the k_T algorithm with $R = 0.3$. Then, `translate` and `rotate` are used to translate and rotate the image, aligning the information of the two sub-jets. Finally, `pixelate` is used to pixelate the image; the size here is 33×33, with a range of $(-1.6, 1.6)$ and an equivalent precision of around 0.1. This precision does not match the precision in the detector card. For simplicity, we take this fixed precision. In code example 31, we show how to prepare the image dataset.

```
from hml.representations import Image
from hml.datasets import ImageDataset

image = (
    Image(
        height="fatjet0.constituents.phi",
        width="fatjet0.constituents.eta",
        channel="fatjet0.constituents.pt",
    )
    .with_subjets(
        constituents="fatjet0.constituents",
        algorithm="kt",
        r=0.3,
        min_pt=0,
    )
    .translate(origin="SubJet0")
    .rotate(axis="SubJet1", orientation=-90)
    .pixelate(
        size=(33, 33),
        range=[(-1.6, 1.6), (-1.6, 1.6)],
    )
)
```

Code example 30: Construct the representation of the image dataset.

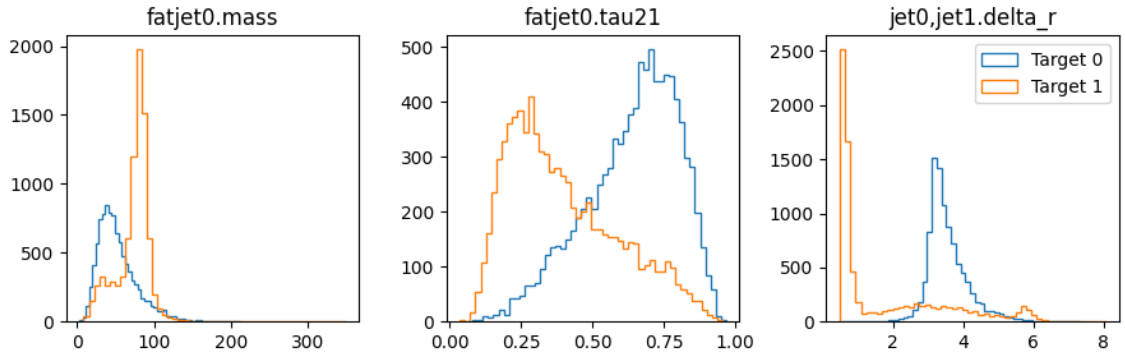


Fig. 6. (color online) Feature distributions of the set dataset.

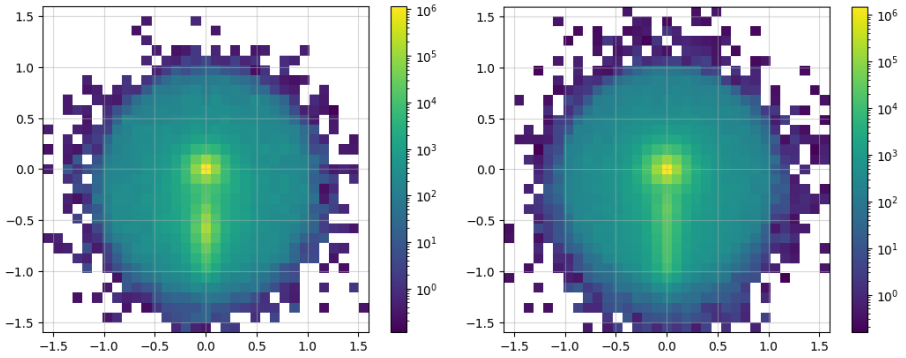


Fig. 7. (color online) Combined images of signal (left) and background (right) events.

After constructing the datasets, we count the signal and background samples in code example 32 to avoid introducing an artificial bias. Then, we use the `show` method of the dataset to display the distribution of the observables, as illustrated in Fig. 6 and Fig. 7.

```
image_ds = ImageDataset(image)
image_ds.read(sig_events, 1, [cut])
image_ds.read(bkg_events, 0, [cut])

image_ds.split(0.7, 0.3)
image_ds.save("./data/wjj_vs_qcd_image.ds")
```

Code example 31: Prepare the image dataset.

```
import numpy as np

print(np.unique(set_ds.targets, return_counts=
    True))
# (array([0, 1], dtype=int32), array([9782,
    8281]))

set_ds.show()
image_ds.show(norm="log", target=0, show_pixels=
    True)
image_ds.show(norm="log", target=1, show_pixels=
    True)
```

Code example 32: Count the number of signals and backgrounds and show the distribution of the datasets.

It is observed that the number of signals and number of background instances are approximately equal, and the observables used clearly reflect the distinct characteristics of each, which is beneficial for our subsequent classification tasks. In the display of the image dataset, we show a merged image of the signal and background. It can be seen that the fat jet image of the signal prominently features two sub-jets, whereas the sub-jets in the background are less distinct.

B. Step 2: apply the approaches

After preparing the datasets, all the available approaches are imported for training. These approaches learn the differences between the signal and background. Subsequently, a dictionary and the built-in metrics are used to assess their performance, which will then be presented as a benchmark test. First, import all the necessary packages, as shown in code example 33; the packages have been roughly categorized to facilitate the understanding of their purposes.

The selected evaluation metrics are accuracy, Area Under the Curve (AUC), signal significance, and background rejection rate at a fixed signal efficiency. These metrics are commonly used in high-energy physics and help us better understand the performance of the approaches. In code example 34, we define a function to retrieve the evaluation metrics of an approach.

The cut-and-count and decision tree approaches are

not sensitive to the scale of features. Hence, we can directly import the dataset (code example 35) and start the training (code example 36). Two different topologies of the cut-and-count are used to demonstrate how the order of applying the cuts affects the performance.

```
# General
import numpy as np
import matplotlib.pyplot as plt
from keras import ops
from rich.table import Table

# Dataset
from sklearn.preprocessing import MinMaxScaler
from hml.datasets import load_dataset

# Approaches
from hml.approaches import CutAndCount as CNC
from hml.approaches import
    GradientBoostedDecisionTree as BDT
from hml.approaches import SimpleCNN as CNN
from hml.approaches import SimpleMLP as MLP

# Evaluation
from keras.metrics import Accuracy, AUC
from sklearn.metrics import roc_curve
from hml.metrics import MaxSignificance,
    RejectionAtEfficiency

# Save and load
from hml.approaches import load_approach
```

Code example 33: Import necessary packages for the benchmark test.

```
results = {}

def get_result(approach, x_test, y_test):
    y_pred = approach.predict(x_test, verbose=0)
    fpr, tpr, _ = roc_curve(y_test, y_pred[:, 1])
    result = {
        approach.name: {
            "acc": Accuracy()(y_test, y_pred.argmax
                (axis=1)),
            "auc": AUC()(y_test, y_pred[:, 1]),
            "sig": MaxSignificance()(y_test, y_pred
                ),
            "r50": RejectionAtEfficiency(0.5)(
                y_test, y_pred),
            "r99": RejectionAtEfficiency(0.99)(
                y_test, y_pred),
            "fpr": fpr,
            "tpr": tpr,
        }
    }

    return result
```

Code example 34: Define a function to obtain the evaluation metrics of a model.

The input for the multilayer perceptron requires the use of `MinMaxScaler` to scale the features within the 0-1 range, as detailed in code example 38, which aids in the rapid convergence of the model. We trained the model for 100 epochs with a batch size of 128. Code example 39 illustrates the training process.

```
ds = load_dataset("./data/wjj_vs_qcd_set.ds")

x_train, y_train = ds.train.samples, ds.train.
    targets
x_test, y_test = ds.test.samples, ds.test.targets
```

Code example 35: Load the set dataset for training the cut-and-count and decision tree approaches.

```
cnc1 = CNC(
    n_observables=3,
    topology="parallel",
    name="cnc_parallel",
)
cnc1.compile(
    optimizer="adam",
    loss="crossentropy",
    metrics=["accuracy"],
    run_eagerly=True,
)
cnc1.fit(x_train, y_train, batch_size=len(x_train
    ))

cnc2 = CNC(
    n_observables=3,
    topology="sequential",
    name="cnc_sequential",
)
cnc2.compile(
    optimizer="adam",
    loss="crossentropy",
    metrics=["accuracy"],
    run_eagerly=True,
)
cnc2.fit(x_train, y_train, batch_size=len(x_train
    ))

results.update(get_result(cnc1, x_test, y_test))
results.update(get_result(cnc2, x_test, y_test))
```

Code example 36: Train the cut-and-count approaches with two different topologies.

```
bdt = BDT(name="bdt")
bdt.compile(metrics=["accuracy"])
bdt.fit(x_train, y_train)

results.update(get_result(bdt, x_test, y_test))
```

Code example 37: Train the decision tree approach.

For the convolutional neural network, we employ two different preprocessing methods. One scales each image by its maximum value, whereas the other applies a logarithmic transformation to each pixel value. Given the large variations in the pixel intensity in the jet images, scaling

directly by the maximum value might result in excessively small pixel values, whereas logarithmic transformation preserves the information better. In code example 40 and 42, we load the image dataset and demonstrate these two distinct preprocessing techniques.

```
ds = load_dataset("./data/wjj_vs_qcd_set.ds")

x_train, y_train = ds.train.samples, ds.train.
    targets
x_test, y_test = ds.test.samples, ds.test.targets

scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
```

Code example 38: Load the set dataset for training the multilayer perceptron approach.

```
mlp = MLP(name="mlp", input_shape=x_train.shape
    [1:])
mlp.compile(
    optimizer="adam",
    loss="crossentropy",
    metrics=["accuracy"],
)
mlp.fit(
    x_train,
    y_train,
    batch_size=128,
    epochs=100,
)
results.update(get_result(mlp, x_test, y_test))
```

Code example 39: Train the multilayer perceptron approach.

Finally, we present a performance comparison using code example 43. The results are shown in Table 3.

Table 3. Comparison of different approaches.

Name	ACC	AUC	Significance	R50	R99
cnc_parallel	0.750323	0.728121	33.660892	4.005174	1.000000
cnc_sequential	0.787784	0.769440	36.557026	4.712174	1.000000
bdt	0.902011	0.955063	44.368549	117.804291	2.146139
mlp	0.900904	0.956274	44.205276	117.804291	2.124265
cnn_max	0.806827	0.867769	38.444225	17.089737	1.188322
cnn_log	0.809452	0.876692	38.732323	19.042860	1.276514

The significance column shows that for the cut-and-count method, the sequential topology, which considers the impacts among the cuts, performs better than the parallel topology. For convolutional neural networks, the performance of logarithmic scaling is roughly equivalent to that using maximum value scaling. Multilayer perceptron and decision trees, which utilize features with clear distinctions, exhibit the best performance. For more practical problems, we can apply different approaches to the dataset and then select the most suitable one based on various performance metrics.

```
ds = load_dataset("./data/wjj_vs_qcd_image.ds")

x_train, y_train = ds.train.samples, ds.train.
    targets
x_test, y_test = ds.test.samples, ds.test.targets

non_zero_train = x_train.reshape(x_train.shape
    [0], -1).sum(1) != 0
non_zero_test = x_test.reshape(x_test.shape[0],
    -1).sum(1) != 0

x_train, y_train = x_train[non_zero_train],
    y_train[non_zero_train]
x_test, y_test = x_test[non_zero_test], y_test[
    non_zero_test]

x_train = (
    x_train.reshape(len(x_train), -1)
    / x_train.reshape(len(x_train), -1).max(1,
        keepdims=True)
).reshape(x_train.shape)
x_test = (
    x_test.reshape(len(x_test), -1)
    / x_test.reshape(len(x_test), -1).max(1,
        keepdims=True)
).reshape(x_test.shape)
x_train = x_train[..., None]
x_test = x_test[..., None]
```

Code example 40: Load the image dataset and normalize the pixel values with the maximum value.

```
cnn1 = CNN(name="cnn_max", input_shape=x_train.
    shape[1:])
cnn1.compile(
    optimizer="adam",
    loss="crossentropy",
    metrics=["accuracy"],
)
cnn1.fit(
    x_train,
    y_train,
    epochs=100,
    batch_size=128,
)
results.update(get_result(cnn1, x_test, y_test))
```

Code example 41: Train the convolutional neural network approach with the maximum value normalization.

VI. SUMMARY

In the current era of rapid evolution of machine learning models, it is worthwhile to explore methods to use them more conveniently in high-energy physics for searching new physical signals. In this paper, we introduced the `hml` Python package, which offers a stream-

lined workflow from event generation to performance evaluation. The simplified process and control over random seeds significantly enhance the reproducibility of the final analysis results.

```
...
x_train = np.log(x_train + 1)
x_test = np.log(x_test + 1)
...

cnn2 = CNN(name="cnn_log", input_shape=x_train.
    shape[1:])
...
```

Code example 42: Normalize the pixel values by using the logarithm. "..." indicates the same code as in code examples 40 and 41.

```
table = Table(
    "Name", "ACC", "AUC", "Significance", "R50", "
        R99", title="Approach Comparison"
)

for name, metrics in results.items():
    table.add_row(
        name,
        f"{metrics['acc']:.6f}",
        f"{metrics['auc']:.6f}",
        f"{metrics['sig']:.6f}",
        f"{metrics['r50']:.6f}",
        f"{metrics['r99']:.6f}",
    )

print(table)
```

Code example 43: Compare the performances of different approaches.

We proposed a naming convention for observables, which enables users to easily extract the required data from the events output by DELPHES. Additionally, we extended the cut expression syntax, originally in UPROOT, to make it more user-friendly and compatible with the DELPHES output formats. This convention is also utilized in our dataset construction process, helping users to quickly and conveniently build datasets. Based on this naming convention, we implemented a transformation from the outputs of event generators to datasets usable by various analysis approaches. Moreover, the `show` method included in the datasets enables users to display the data either as 1D distributions or 2D images, facilitating the adjustment of observable selections based on the observed differences.

We adopted the interface style of KERAS to standardize traditional methods such as the cut-and-count technique and decision trees. Furthermore, the cut-and-count approach supports automatic searching for the optimal cut positions, significantly reducing the workload for users. Additionally, we incorporated the commonly used evaluation metrics in high-energy physics, such as signal significance and background rejection rate at a fixed signal efficiency. These metrics help users to better understand the performance of the models.

We demonstrated the complete workflow through a practical example, which intuitively showcased the usage of `hml`. `hml` is continuously being updated. We plan to incorporate additional existing deep learning models and datasets, and extend it to graph representations of data to further enhance its capabilities.

References

[1] J. Cogan, M. Kagan, E. Strauss *et al.*, JHEP **02**, 118 (2015)
[2] L. G. Almeida, M. Backović, M. Cliche *et al.*, JHEP **07**, 086 (2015)
[3] L. de Oliveira, M. Kagan, L. Mackey *et al.*, JHEP **07**, 069 (2016)
[4] P. Baldi, K. Cranmer, T. Faucett *et al.*, Eur. Phys. J. C **76**(5), 235 (2016)
[5] P. T. Komiske, E. M. Metodiev, and M. D. Schwartz, JHEP **01**, 110 (2017)
[6] G. Kasieczka, T. Plehn, M. Russell *et al.*, JHEP **05**, 006 (2017)
[7] L. M. Dery, B. Nachman, F. Rubbo *et al.*, JHEP **05**, 145 (2017)
[8] G. Louppe, K. Cho, C. Becot *et al.*, JHEP **01**, 057 (2019)
[9] A. Butter, G. Kasieczka, T. Plehn *et al.*, SciPost Phys. **5**(3), 028 (2018)
[10] E. M. Metodiev, B. Nachman, and J. Thaler, JHEP **10**, 174 (2017)
[11] J. A. Aguilar-Saavedra, J. H. Collins, and R. K. Mishra, JHEP **11**, 163 (2017)
[12] L. Moore, K. Nordström, S. Varma *et al.*, SciPost Phys. **7**(3), 036 (2019)
[13] T. Heimel, G. Kasieczka, T. Plehn *et al.*, SciPost Phys. **6**(3), 030 (2019)
[14] P. T. Komiske, E. M. Metodiev, and J. Thaler, JHEP **01**, 121 (2019)
[15] H. Qu and L. Gouskos, Phys. Rev. D **101**(5), 056019 (2020)
[16] A. Butter *et al.*, SciPost Phys. **7**, 014 (2019)
[17] E. A. Moreno, O. Cerri, J. M. Duarte *et al.*, Eur. Phys. J. C **80**(1), 58 (2020)
[18] Y. C. J. Chen, C. W. Chiang, G. Cottin *et al.*, Phys. Rev. D **101**(5), 053001 (2020)
[19] V. Mikuni and F. Canelli, Eur. Phys. J. Plus **135**(6), 463 (2020)
[20] J. S. H. Lee, I. Park, I. J. Watson *et al.*, J. Korean Phys. Soc. **84**, 427 (2024)
[21] F. A. Dreyer and H. Qu, JHEP **03**, 052 (2021)
[22] L. Anzalone, T. Diotalevi, and D. Bonacorsi, (2022). DOI: 10.1088/2632-2153/ac917c
[23] S. K. Choi, J. Li, C. Zhang *et al.*, Phys. Rev. D **108**(11), 116002 (2023)
[24] A. Elwood, D. Krücker, and M. Shchedrolosiev, J. Phys. Conf. Ser. **1525**, 012110 (2020)
[25] P. Baldi, P. Sadowski, and D. Whiteson, Nature Commun.

5, 4308 (2014)
[26] A. Aurisano, A. Radovic, D. Rocco *et al*., JINST **11**(09), P09001 (2016)
[27] W. Bhimji, S. A. Farrell, T. Kurth *et al*., J. Phys. Conf. Ser. **1085**(4), 042034 (2018)
[28] P. Abratenko *et al*., Phys. Rev. D **103**(9), 092003 (2021)
[29] J. Li, T. Li, and F. Z. Xu, JHEP **04**, 156 (2021)
[30] Y. Zhu, H. Liang, Y. Wang *et al*., Eur. Phys. J. C **84**(2), 152 (2024)
[31] E. Buhmann, C. Ewen, G. Kasieczka *et al*., Phys. Rev. D **109**(5), 055015 (2024)
[32] S. Song, J. Chen, J. Liu *et al*., JINST **19**(04), P04033 (2024)
[33] C. L. Cheng, G. Singh, and B. Nachman, *Incorporating Physical Priors into Weakly-Supervised Anomaly Detection*, (2024), arXiv: 2405.08889
[34] C. Li *et al*., *Accelerating Resonance Searches via Signature-Oriented Pre-training*, (2024), arXiv: 2405.12972
[35] L. de Oliveira, M. Paganini, and B. Nachman, Comput. Softw. Big Sci. **1**(1), 4 (2017)
[36] M. Paganini, L. de Oliveira, and B. Nachman, Phys. Rev. Lett. **120**(4), 042003 (2018)
[37] M. Paganini, L. de Oliveira, and B. Nachman, Phys. Rev. D **97**(1), 014021 (2018)
[38] P. Baldi, L. Blecher, A. Butter *et al*., SciPost Phys. **13**(3), 064 (2022)
[39] C. Jiang, S. Qian, and H. Qu, SciPost Phys. **18**, 195 (2025)
[40] D. Kobylianskii, N. Soybelman, E. Dreyer *et al*., Phys. Rev. D **110**, 072003 (2024)
[41] M. Feickert and B. Nachman, *A Living Review of Machine Learning for Particle Physics*, (2021), arXiv: 2102.02770
[42] J. Alwall, R. Frederix, S. Frixione *et al*., JHEP **07**, 079 (2014)
[43] T. Sjöstrand, S. Ask, J.R. Christiansen *et al*., Comput. Phys. Commun. **191**, 159 (2015)
[44] J. de Favereau, C. Delaere, P. Demin *et al*., JHEP **02**, 057 (2014)
[45] R. Brun, F. Rademakers, P. Canal *et al*., root-project/root: v6.18/02, (2020), DOI: https://doi.org/10.5281/zenodo.3895860
[46] J. Ansel, E. Yang, H. He *et al*., in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Volume 2 (ASPLOS ' 24) (ACM, 2024). DOI: 10.1145/3620665.3640366. URL https://pytorch.org/assets/pytorch2-2.pdf
[47] M. Abadi, A. Agarwal, P. Barham *et al*., *TensorFlow: Large-scale machine learning on heterogeneous systems*, (2015). URL https://www.tensorflow.org/. Software available from tensorflow.org
[48] L. Benato *et al*., Comput. Softw. Big Sci. **6**(1), 9 (2022)
[49] Y. C. Guo, F. Feng, A. Di *et al*., Comput. Phys. Commun. **294**, 108957 (2024)
[50] J. Brehmer, F. Kling, I. Espejo *et al*., Comput. Softw. Big Sci. **4**(1), 3 (2020)
[51] J. Brehmer, K. Cranmer, I. Espejo *et al*., J. Phys. Conf. Ser. **1525**(1), 012022 (2020)
[52] J. Pivarski, P. Das, C. Burr *et al*., scikit-hep/uproot: 3.12.0, (2020). DOI: https://doi.org/10.5281/zenodo.3952728
[53] F. Chollet *et al*., Keras. https://keras.io (2015)
[54] J. Pivarski, I. Osborne, I. Ifrim *et al*., Awkward Array, (2018). DOI: https://doi.org/10.5281/zenodo.4341376
[55] A. J. Larkoski, I. Moult, and B. Nachman, Phys. Rept. **841**, 1 (2020)
[56] A. Das, P. Konar, and A. Thalapillil, JHEP **02**, 083 (2018)
[57] A. Bhardwaj, A. Das, P. Konar *et al*., J. Phys. G **47**(7), 075002 (2020)
[58] S. Chakraborty, M. Mitra, and S. Shil, Phys. Rev. D **100**(1), 015012 (2019)
[59] L. Buonocore, U. Haisch, P. Nason *et al*., Phys. Rev. Lett. **125**(23), 231804 (2020)
[60] V. S. Ngairangbam, A. Bhardwaj, P. Konar *et al*., Eur. Phys. J. C **80**(11), 1055 (2020)
[61] L. Buitinck, G. Louppe, M. Blondel *et al*., in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, (2013), pp. 108–122
[62] A. Hocker *et al*., *TMVA - Toolkit for Multivariate Data Analysis*, (2007)
[63] T. Chen and C. Guestrin, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (ACM, New York, NY, USA, 2016), KDD'16, pp. 785–794. DOI: 10.1145/2939672.2939785
[64] M. Guillame-Bert, S. Bruch, R. Stotz *et al*., in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, (2023), pp. 4068–4077. DOI: 10.1145/3580305.3599933