From ROOT to PyTorch: Seamless Data Pipelines with HDF5

PyHEP 2025 - "Python in HEP" Users Workshop

Jan Gavranovic, 28 October 2025

Institut
"Jožef Stefan"
Ljubljana, Slovenija

UNIVERSITY OF LJUBLJANA
FMF
Faculty of Mathematics and Physics

Introduction

- ML is becoming a crucial tool in a lot of HEP analyses

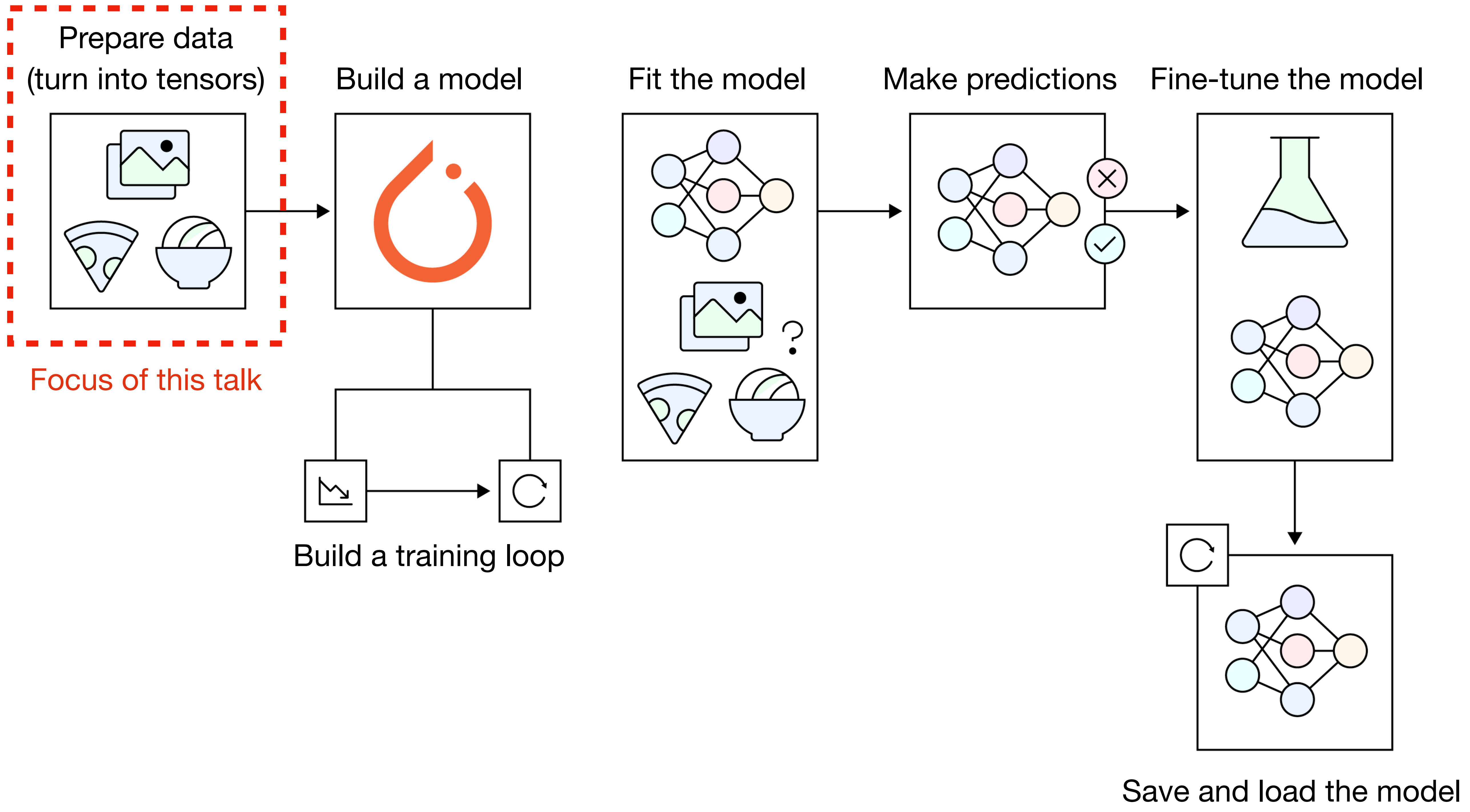
- There is a need to integrate the ML workflow inside a HEP analysis

- Challenges in integrating ML into HEP analyses:

    - Reliance on large-scale datasets stored in ROOT format (not ML friendly)

    - ML frameworks expect matrix-like data while HEP data is often jagged in structure

A PyTorch workflow



- Focus of this talk is on how to *bridge the gap* between HEP data stored in ROOT format and *PyTorch* ecosystem for data loading

Common workflow scenario

1. Central analysis framework that produces ROOT files (in most cases C++)

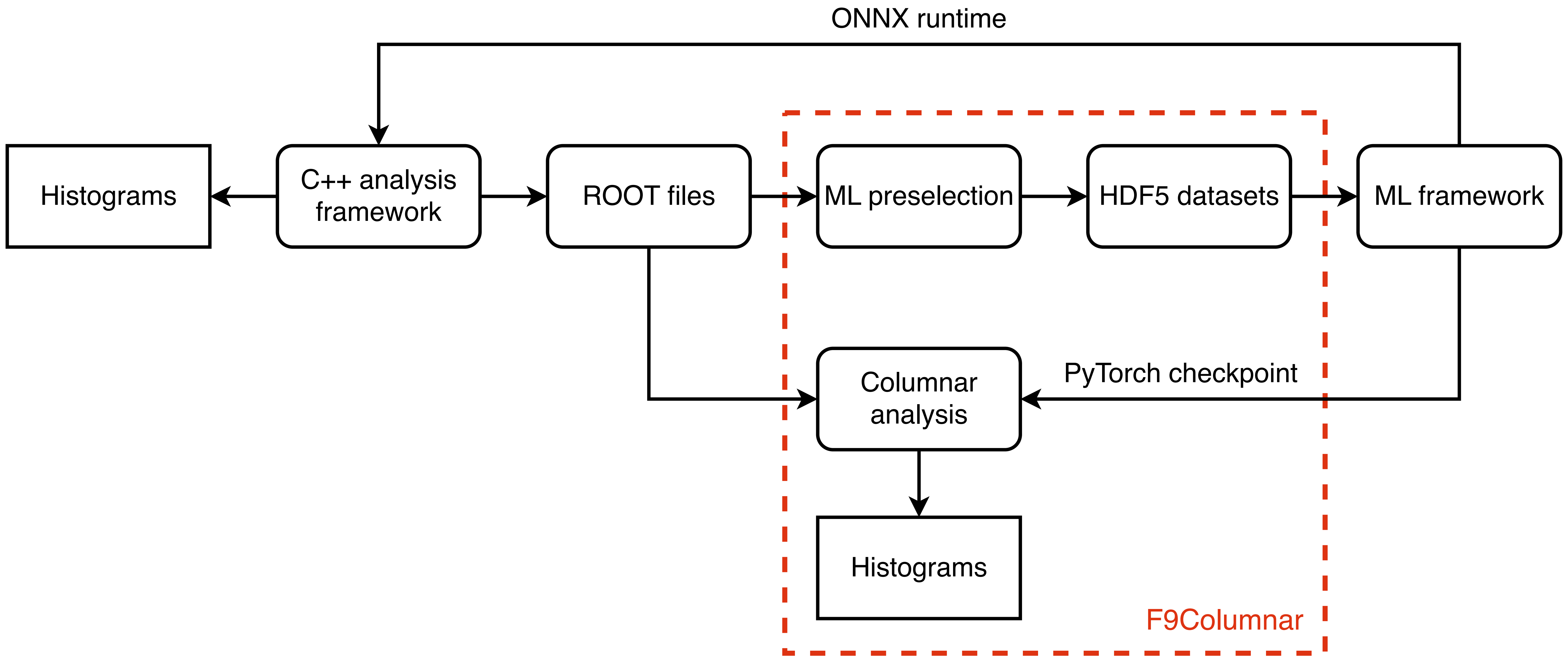
2. **Preprocess events** saved in input ROOT files (preselection)

3. **Save preprocessed events** ready for training to HDF5 files

4. Train using HDF5 files

5. Export model back to the framework that produced ROOT files or to a columnar analysis

6. Plot histograms



- I will present a PyTorch based library (***F9Columnar***) that preprocesses events from ROOT files and saves them to HDF5 files for ML training

```
pip install f9columnar
```

Preparing input data for ML

- HEP datasets come in ROOT files and are usually separated by MC and data
- Data is split on data taking periods
- MC is split for different processes
- Two challenges:
  1. **Large datasets** → solved by an *iterable dataset*
  2. **Shuffling data** → solved by *externally shuffling* events to HDF5

Z+jets MC example from ATLAS Open Data

```
ODEO_FEB2025_v0_1LMET30_mc_700320.Sh_2211_Zee_maxHTpTV2_BFilter.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700321.Sh_2211_Zee_maxHTpTV2_CFilterBVeto.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700322.Sh_2211_Zee_maxHTpTV2_CVetoBVeto.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700323.Sh_2211_Zmumu_maxHTpTV2_BFilter.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700324.Sh_2211_Zmumu_maxHTpTV2_CFilterBVeto.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700325.Sh_2211_Zmumu_maxHTpTV2_CVetoBVeto.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700901.Sh_2214_Ztt_maxHTpTV2_Mll10_40_BFilter.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700902.Sh_2214_Ztt_maxHTpTV2_Mll10_40_CFilterBVeto.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700903.Sh_2214_Ztt_maxHTpTV2_Mll10_40_CVetoBVeto.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700358.Sh_2211_Zee2jets_Min_N_TChannel.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700359.Sh_2211_Zmm2jets_Min_N_TChannel.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700360.Sh_2211_Ztt2jets_Min_N_TChannel.1LMET30.root
ODEO_FEB2025_v0_1LMET30_mc_700361.Sh_2211_Znunu2jets_Min_N_TChannel.1LMET30.root
```

Run 2 year 2015/16 data example from ATLAS Open Data

```
ODEO_FEB2025_v0_1LMET30_data15_periodD.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data15_periodE.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data15_periodF.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data15_periodG.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data15_periodH.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data15_periodJ.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodA.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodB.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodC.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodD.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodE.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodF.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodG.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodI.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodK.1LMET30.root
ODEO_FEB2025_v0_1LMET30_data16_periodL.1LMET30.root
```

Iterable datasets and data loaders

- PyTorch provides classes to deal with data:
  - **torch.utils.data.DataLoader**: combines a dataset and a sampler, and provides an iterable over the given dataset
  - **torch.utils.data.Dataset**: implements the `__getitem()__` and `__len__()` protocols, and represents a map from indices to data samples
  - **torch.utils.data.IterableDataset:** implements the `__iter__()` protocol, and represents an iterable over data samples

- The code uses uproot's `TTree.iterate` inside an `IterableDataset`

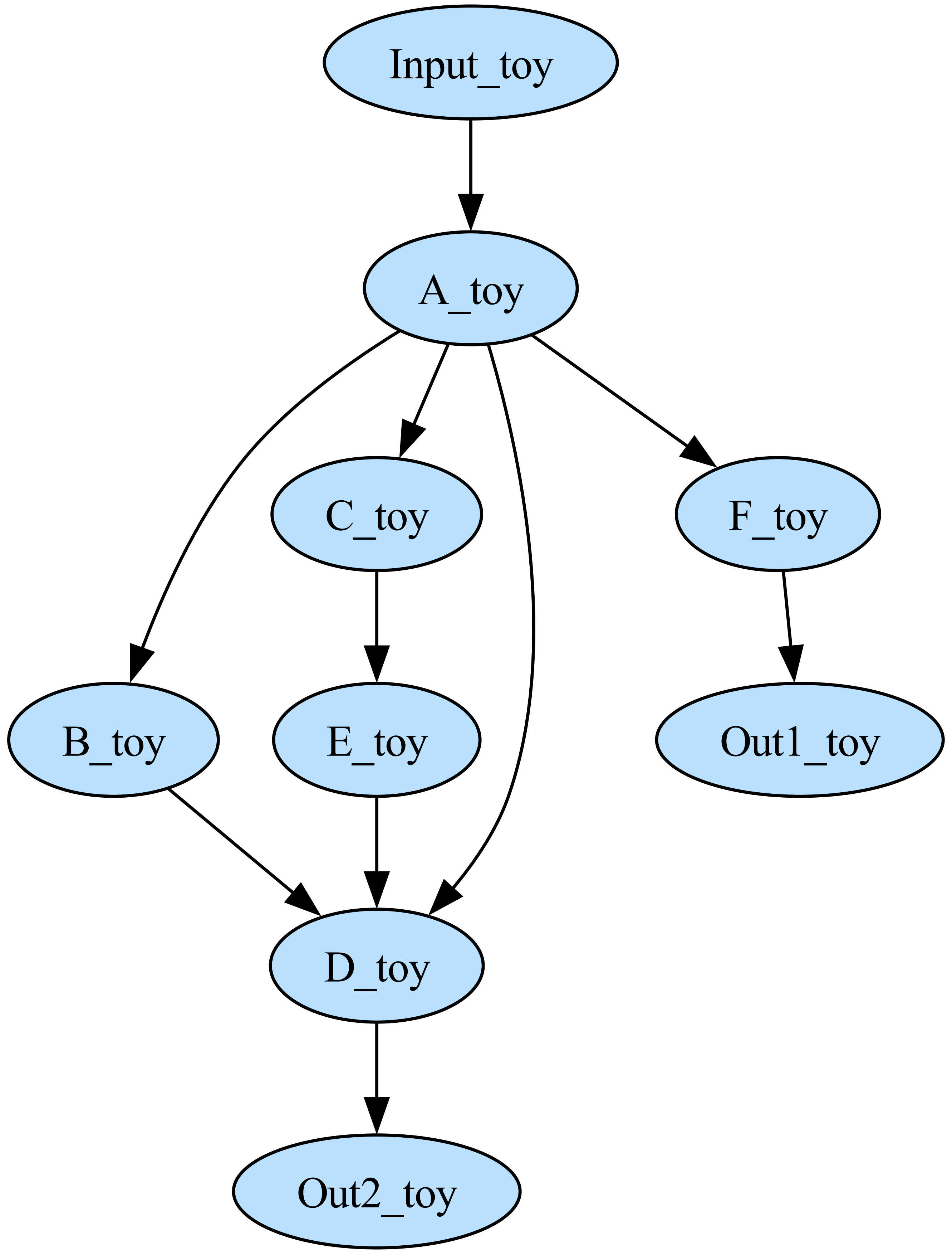
```
from f9columnar.root_dataloader import get_root_dataloader

# root_dataloader is an instance of a torch DataLoader that uses an IterableDataset
root_dataloader, total = get_root_dataloader(
    name="data", # arbitrary name identifier
    files=files, # list of root files
    key="NOMINAL", # root file tree name
    step_size=10**5, # number of events per array batch to read into memory
    num_workers=-1, # number of workers for parallel processing
    processors=None, # arbitrary calculations on arrays
    filter_name=None, # filter branches
    dataloader_kwargs=None, # additional arguments for DataLoader
)

# loop over batches of events from .root file(s), each batch is an awkward array
for events in root_dataloader:
    arrays, report = events
    # ... do something with the arrays
```

Next slide!

- Supports multi-process data loading with the specified number of loader worker processes
  - Dataset access together with its internal IO runs in the worker process

ML preselection as a DAG

- Data loader takes an optional argument called `processors` that is a `ProcessorsGraph` object
- This is a **NetworkX** directed acyclic graph (DAG) that describes the steps how awkward arrays are transformed
- The DAG describes a minimal columnar analysis that is performed inside a PyTorch data loader (i.e. each worker is running a copy of the graph on different inputs)
- The awkward array result of DAG computation can be histogrammed or saved to a HDF5 file as a ML preselection step using **h5py**




```python
from f9columnar.processors import ProcessorsGraph, Processor

class ToyProcessor(Processor):
    def __init__(self, name: str) -> None:
        super().__init__(name)

    def run(self, **kwargs: Any) -> dict[str, int]:
        result = random.randint(0, 10)
        print(f"Node {self.name}({kwargs}) with {result}.")
        return {"result": result}

toy_graph = ProcessorsGraph(
    prune_results=True,
    global_attributes=None,
    identifier="_toy",
)

I = ToyProcessor(name="Input")
A = ToyProcessor(name="A")
B = ToyProcessor(name="B")
C = ToyProcessor(name="C")
D = ToyProcessor(name="D")
E = ToyProcessor(name="E")
F = ToyProcessor(name="F")
Out1 = ToyProcessor(name="Out1")
Out2 = ToyProcessor(name="Out2")

toy_graph.add(I, A, B, C, D, E, F, Out1, Out2)

toy_graph.connect(
    [
        ("Input", "A"),
        ("A", "B"),
        ("A", "C"),
        ("C", "E"),
        ("E", "D"),
        ("B", "D"),
        ("A", "D"),
        ("A", "F"),
        ("F", "Out1"),
        ("D", “Out2"),
    ],
)

toy_graph.style_graph("#BAE0FF")
toy_graph.draw(file_path="toy_graph.pdf")

processors = toy_graph.fit()
```

How to shuffle?

- To improve model convergence, it is common practice to randomly shuffle the training dataset before feeding it into GPU trainer to avoid bias on the order of the data

- Fisher-Yeats shuffle does not work for arrays that do not fit into memory

- Have to use external shuffling

- **Each newly shuffled dataset (*saved as a HDF5 file called a pile*) should accurately reflect the underlying distribution of all MC processes or all data years**

Fisher-Yeats algorithm

```
-- To shuffle an array a of n elements (indices 0,...,n − 1):
for i from n − 1 down to 1 do
     j ← random integer such that 0 ≤ j ≤ i
     exchange a[j] and a[i]
```

External shuffle algorithm

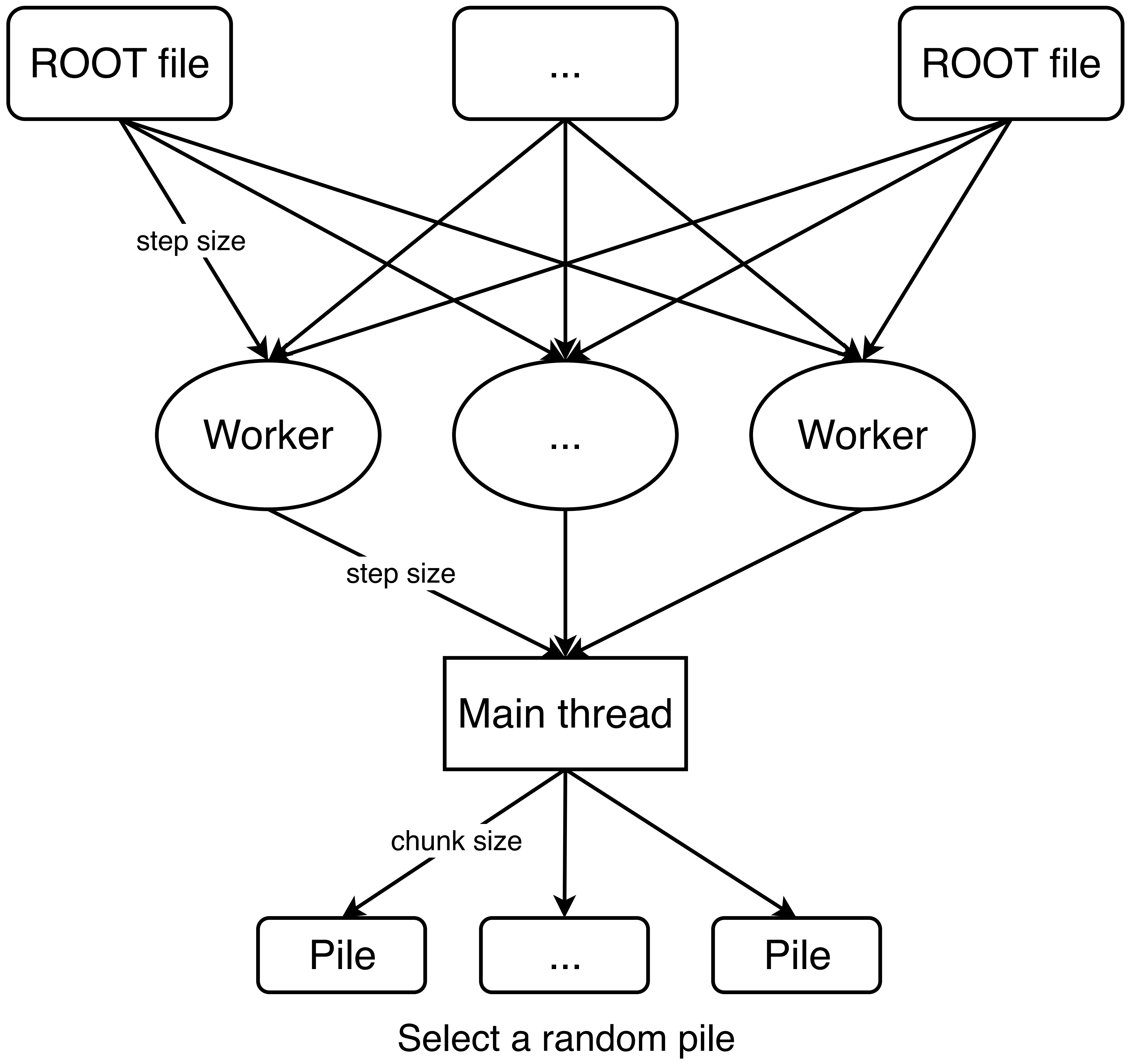
```
create empty datasets called piles p[0], p[1], ..., p[n - 1]
for step size of events x from a root file i do
    for chunk size c of events x do
        j ← random integer in [0, n - 1]
        append c to p[j]
```

```
for j in [n1, n2] where 0 ≤ n1 < n2 and n1 < n2 ≤ n do
    read all events from p[j] into memory
    shuffle p[j] in memory
    for batch size of events x in p[j] do
        yield x to DataLoader
```

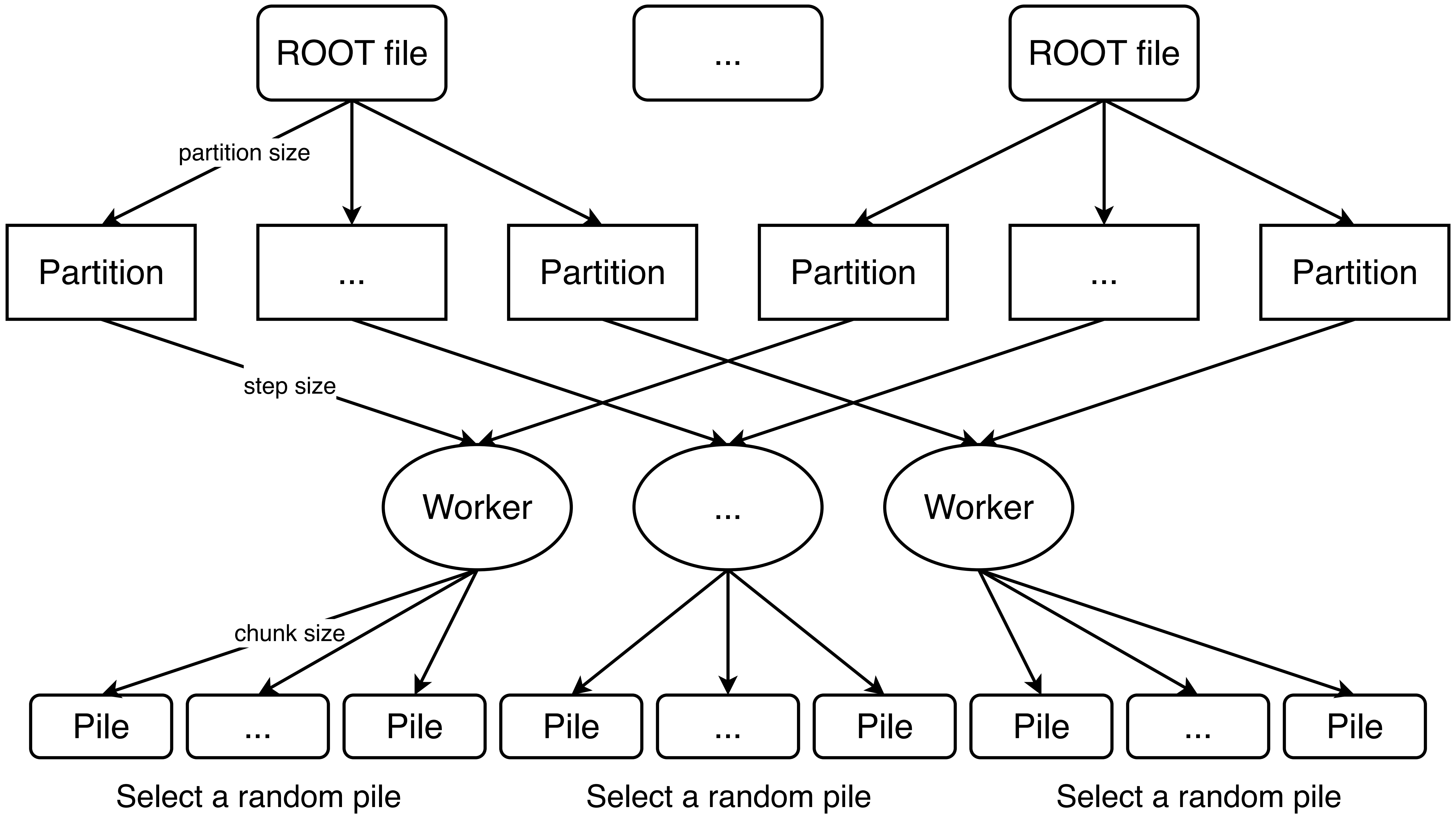
Per-epoch reading from saved shuffled piles

Two versions of external shuffling

- Each worker yields a step size of events to the main thread that holds all piles and shuffles chunks to piles



- Each worker holds some number of piles and shuffles partitions of events in chunks to its piles



External shuffling implementation

- Shuffling is performed in the main thread
- Better event pile coverage

```
from f9columnar.ml.hdf5_writer import Hdf5WriterPostprocessor

class Hdf5WriterPostprocessor(Postprocessor):
    def __init__(
        self,
        file_path: str,
        postprocessor_name: str = "hdf5WriterPostprocessor",
        flat_column_names: list[str] | None = None,
        jagged_column_names: dict[str, list[str]] | None = None,
        default_chunk_shape: int = 1024,
        custom_chunk_shape_dct: dict[str, int] | None = None,
        max_lengths: dict[str, int] | int = 10,
        pad_values: dict[str, float] | float | None = 0.0,
        n_piles: int | None = None,
        pile_assignment: str = "random",
        save_node: str = "output",
        merge_piles: bool = True,
        enforce_dtypes: dict[str, str] | None = None,
        full_metadata: bool = False,
        **hdf5_kwargs: Any,
    ) -> None
```

- Each worker shuffles to piles assigned to it
- Faster execution

```
from f9columnar.ml.hdf5_parallel_writer import Hdf5WriterProcessor

class Hdf5WriterProcessor(Processor):
    def __init__(
        self,
        file_path: str,
        postprocessor_name: str = "hdf5WriterProcessor",
        flat_column_names: list[str] | None = None,
        jagged_column_names: dict[str, list[str]] | None = None,
        default_chunk_shape: int = 1024,
        custom_chunk_shape_dct: dict[str, int] | None = None,
        max_lengths: dict[str, int] | int = 10,
        pad_values: dict[str, float] | float | None = 0.0,
        n_piles: int | None = None,
        pile_assignment: str = "random",
        enforce_dtypes: dict[str, str] | None = None,
        full_metadata: bool = False,
        max_workers: int | None = None,
        **hdf5_kwargs: Any,
    ) -> None
```

Putting it all together in code

```python
# import necessary classes
from f9columnar.dataset_builder import RootPhysicsDataset
from f9columnar.ml.hdf5_parallel_writer import Hdf5WriterProcessor
from f9columnar.processors import (
    CheckpointProcessor,
    Processor,
    ProcessorsGraph,
)
from f9columnar.processors_collection import ProcessorsCollection
from f9columnar.run import ColumnarEventLoop

# define processors and add them to a dictionary-like collection
analysis_collection = ProcessorsCollection(“analysisCollection")
analysis_collection += BranchesProcessor(input_branches)
analysis_collection += WeightsProcessor()
analysis_collection += LabelsProcessor(labels)
analysis_collection += SameSignPairProcessor()


# define a HDF5 writer to save awkward arrays
hdf5_writer = Hdf5WriterProcessor(
    file_path=output_file,
    flat_column_names=[...],
    jagged_column_names={"electrons": [...], "muons": [...], "jets": [...]},
    max_lengths={"electrons": 2, "muons": 2, "jets": 5},
    pad_values=999,
    n_piles=512,
    pile_assignment="random",
    merge_piles=False,
    max_workers=-1,
)
analysis_collection += hdf5_writer

# make a graph and add processors to it
analysis_graph = ProcessorsGraph()
analysis_graph.add(
    CheckpointProcessor("input"),
    *analysis_collection.as_list(),
    CheckpointProcessor("output"),
)
analysis_graph.chain() # build a linear graph
```

```python
class SameSignPairProcessor(Processor):
    def __init__(self) -> None:
        super().__init__(name="sameSignProcessor")

    @staticmethod
    def same_sign_pairs_mask(lep1: ak.Array, lep2: ak.Array) -> ak.Array:
        charges = ak.concatenate((lep1, lep2), axis=1)

        # count number of +1 and -1
        count_pos = ak.sum(charges == 1, axis=1)
        count_neg = ak.sum(charges == -1, axis=1)

        # count how many full same-sign pairs we can form
        same_sign_pairs = (count_pos // 2) + (count_neg // 2)

        # total number of full pairs we can possibly form from all charges
        total_pairs = ak.num(charges) // 2

        # it's valid only if all pairs we can form are same-sign pairs
        return same_sign_pairs == total_pairs

    def run(self, arrays: ak.Array) -> dict[str, ak.Array]:
        if "el_charge" in arrays.fields and "mu_charge" in arrays.fields:
            ss_pair_mask = self.same_sign_pairs_mask(arrays["el_charge"], arrays["mu_charge"])
            arrays = arrays[ss_pair_mask]

        return {"arrays": arrays}
```

Save to HDF5

```python
# wrapper for get_root_dataloader
dataset = RootPhysicsDataset("MC", files, is_data=False)
dataset.setup_dataloader(**dataloader_config)
dataset.init_dataloader(processors=analysis_graph)

# wrapper for data loader iteration
event_loop = ColumnarEventLoop(
    mc_datasets=[dataset],
    data_datasets=None,
    postprocessors_graph=None,
    fit_postprocessors=False,
    cut_flow=False,
)
# run the pipeline
event_loop.run(mc_only=True)
```

- Example of jagged ML preselection for pairs of same sign leptons in the final state
- A modular and extensible framework of classes designed for easy composition and future scalability

HDF5 file structure

Jet jagged features

	jet_e	jet_pt	jet_eta	jet_phi	jet_ftag_quantile
0	1045.4	279.31	1.9944	-2.1202	1
1	257.69	225.58	0.51694	0.9058	1
2	104.25	67.337	1.0017	-2.0874	1
3	999	999	999	999	999
4	999	999	999	999	999

Padding

Muon jagged features

	mu_charge	mu_pt	mu_eta	mu_phi
0	1	189.57	0.57925	1.0714
1	999	999	999	999

Padding

Electron jagged features

	el_charge	el_pt	el_eta	el_phi
0	1	77.827	0.089789	-2.1156
1	999	999	999	999

Padding

Flat features

	ptl1	ptl2	etal1	etal2	phil1	phil2
0	189.57	77.827	0.57925	0.089789	1.0714	-2.1156
1	213.06	50.502	1.9385	0.90365	0.28439	-0.51964
2	145.43	108.53	-0.38379	-1.9789	2.6666	-0.18607
3	249.84	68.331	1.636	-0.4379	1.9068	-0.8002
4	312.12	102.41	-0.90688	0.62145	1.0827	-1.9444
5	91.313	87.262	-1.917	-0.6375	2.671	-1.0829
6	293.41	61.099	-0.31475	0.84198	-0.75723	1.4277
7	830.87	212.22	-0.82804	1.02	2.4422	-0.52818
8	1473.5	133.59	-0.28962	1.7645	-3.0071	-0.41944
9	1360.4	833.58	0.38636	0.19496	2.9189	1.4792
10	1117.8	750.34	-0.88311	0.041773	-0.041482	1.0496

- Example of 2 lepton and 5 jet saved ML preselection
- Each physics object is saved as a HDF5 dataset

Datasets inside one HDF5 pile

Pile0.hdf5

- electrons *(jagged)*
- events *(flat)*
- jets *(jagged)*
- metadata *(dictionary)*
- muons *(jagged)*

Data loading example

- *PyTorch Lightning* example on how to use a `get_ml_hdf5_dataloader` function
- It can also be used standalone in the same way as `get_root_dataloader`

User defined per-worker setup function

```python
def user_setup_func(
    stacked_datasets: StackedDatasets,
    ml_iterator: MLHdf5Iterator,
) -> WeightedDatasetBatch | None:
    weighted_datase_batch = WeightedDatasetBatch()

    for ds_name in dataset_keys:
        ds = stacked_datasets[ds_name]

        X = ds.X # features

        if X.shape[0] == 0:
            return None # nothing to process

        y = ds.get_extra("label_type", None) # optional labels
        w = ds.get_extra("weights", None) # optional MC/data weights
        y_aux = ds.get_extra("aux_type", None) # optional auxiliary labels

        weighted_datase_batch[ds_name] = WeightedBatch(X, y, w, y_aux)

    return weighted_datase_batch
```

Dataclass container for structured arrays

Dataclass container for event batches of features, labels and weights

Any additional processing can happen here before the result is returned to the training loop

Returned to the collate function

User defined collate function

```python
def user_collate_fn(
    batch: tuple[WeightedDatasetBatch, dict[str, Any]],
) -> tuple[dict[str, BatchType], dict[str, Any]]:
    datasets_batch, reports = batch[0].make_batch(), batch[1]
    return datasets_batch, reports
```

Tensor and dictionary return value to the training loop

```python
import lightning as L
from torch.utils.data import DataLoader

from f9columnar.ml.dataloader_helpers import ColumnSelection
from f9columnar.ml.hdf5_ml_dataloader import get_ml_hdf5_dataloader


class LightningHdf5DataModule(L.LightningDataModule):
    def __init__(
        self,
        dl_name: str,
        files: str | list[str],
        column_names: list[str],
        stage_split_piles: dict[str, list[int] | int],
        shuffle: bool = False,
        collate_fn: Callable | None = None,
        dataset_kwargs: dict[str, Any] | None = None,
        dataloader_kwargs: dict[str, Any] | None = None,
    ) -> None:
        super().__init__()
        self.dl_name = dl_name
        self.files = files
        self.column_names = column_names
        self.stage_split_piles = stage_split_piles
        self.shuffle = shuffle
        self.collate_fn = collate_fn
        self.ds_kwargs = dataset_kwargs
        self.dl_kwargs = dataloader_kwargs

        self._selection: ColumnSelection | None = None

    @property
    def selection(self) -> ColumnSelection:
        if self._selection is None:
            raise ValueError("DataModule not yet setup, selection is not available.")
        return self._selection

    def _get_dataloader(self, stage: str) -> DataLoader:
        dl, self._selection, _ = get_ml_hdf5_dataloader(
            f"{stage} - {self.dl_name}",
            self.files,
            self.column_names,
            stage_split_piles=self.stage_split_piles,
            stage=stage,
            shuffle=self.shuffle,
            collate_fn=self.collate_fn,
            dataset_kwargs=self.ds_kwargs,
            dataloader_kwargs=self.dl_kwargs,
        )
        return dl

    def train_dataloader(self) -> DataLoader:
        return self._get_dataloader("train")

    def val_dataloader(self) -> DataLoader:
        return self._get_dataloader("val")

    def test_dataloader(self) -> DataLoader:
        return self._get_dataloader("test")
```

```python
dataset_kwargs = {
    "imbalanced_sampler": None,
    "drop_last": False,
    "scaler_type": None,
    "scaler_path": None,
    "setup_func": user_setup_func,
}
```

```python
dataloader_kwargs = {
    "batch_size": 128,
    "prefetch_factor": 2,
    "num_workers": -1,
}
```

Dataclass container for selected columns bookkeeping

```python
stage_split_piles = {
    "train": 128,
    "val": 64,
    "test": 64,
}
```

Build data loaders for training, validation and test stages depending on how many piles each stage uses

Integrated feature scaling

- The package also includes a utility to scale the entire HDF5 dataset using sci-kit learn scalers
- This is particularly useful for ML where feature scaling can improve model performance
- Split on continuous (numerical) and discrete (categorical) features
- Scalers are saved as pickle files and loaded and applied by the HDF5 data loader automatically if enabled

```
feature_scaling_kwargs = {
    "scaler_type": scaler_type,
    "scaler_path": save_path, # where the scaler was saved
    "scalers_extra_hash": "",
}
dataset_kwargs = dataset_kwargs | feature_scaling_kwargs
```

```
from f9columnar.ml.dataset_scaling import DatasetScaler

ds_scaler = DatasetScaler(
    files, # list of hdf5 files
    scaler_type, # minmax, maxabs, standard, robust, quantile, power, logit, standard_logit
    features, # names of the features to scale
    scaler_save_path=save_path, # where to save the scaler
    n_max=None, # maximum number of events to use for fitting the scaler
    extra_hash="", # extra string to add to the hash of the scaler
    scaler_kwargs=scaler_kwargs, # kwargs for the scaler
    dataloader_kwargs=dataloader_kwargs, # kwargs for the dataloader
)
ds_scaler.feature_scale()
```

Internally this happens

```
def feature_scale(self) -> None:
    progress = get_batch_progress()
    progress.start()
    bar = progress.add_task("Processing batches", total=None)

    for batch in self.hdf5_dataloader:
        ds_batch, _ = batch

        break_fit = self._scale(ds_batch)

        if break_fit:
            logging.info("Breaking feature scaling loop!")
            break

        progress.update(bar, advance=1)

    progress.stop()
```

- Supports partial fitting of standard, minmax and label encoder scalers
- In the case of a jagged dataset, each object has its own scaler

Conclusion

- Presented a modular Python library designed to aid ML analyses in HEP
- Demonstrated the library's structure and workflow
- Implements DAG-based columnar computation for efficient ML preselection
- Supports external shuffling and data storage in HDF5 format
- Provides *seamless* data loading utility for PyTorch
- Built with modularity and transparency in mind to support future extensions

Project description

F9 Columnar

Ruff | Python 3.12 | PyTorch 2.8 | license MIT | pypi v0.3.9

A lightweight Python package for processing of ROOT and HDF5 event data in high energy physics.

Project description

This package is designed for efficient handling of large datasets. Built on PyTorch, Awkward Array, and Uproot, it utilizes PyTorch's DataLoader with an IterableDataset to enable parallel processing. It implements a columnar event loop, returning batches of events in a format compatible with standard PyTorch training loops over multiple epochs.

It is optimized for machine learning applications, the package provides `RootDataLoader` and `Hdf5DataLoader` classes for data loading from ROOT and HDF5 files. Additionally, it supports parallel data processing through a modular pipeline of processor classes, allowing users to chain operations for complex computations and histogramming.

```
pip install f9columnar
```