Zero-overhead training of machine learning models with ROOT data

Vincenzo Eduardo Padulano[1], *Kristupas Pranckietis*[2], *Nopphakorn Subsa-Ard*[3], *Lorenzo Moneta*[1]

[1] CERN, EP-SFT
[2] Vilnius University (LT)
[3] King Mongkut's University of Technology Thonburi (KMUTT) (TH)
21.10.2024, CHEP ‘24, Kraków, Poland

Outline

- HEP data loading for ML training
- Native data loading for ML with ROOT
- Performance evaluation

HEP data loading for ML training

- Long-standing synergy
  - NNs used at LEP (link) and Tevatron (link)
  - 2014 Higgs Boson ML challenge (link)
  - H -> bb confirmation using BDT (link)
- Different purposes
  - Classification, reconstruction, simulation, …
- ML techniques widely employed in the field
  - Different experiments (e.g. ALICE, ATLAS, CMS, LHCb)
  - Track 9 at CHEP'23, various talks this year

ML datasets

Plenty of datasets employed, many Open Data:

- CMS Open Data release for ML (link)
- Particle Transformer for Jet tagging data set (link)
- LHC Olympics 2020 (link)
- Fast Calorimeter Simulation Challenge (link)

Main data formats seen: TTree, HDF5

Data loading

Data loading is a crucial aspect in the model training step

- Disk-to-memory or memory-to-memory representation of tensors
- Further data manipulation before training
- Need to provide a fast data pipeline, asynchronous to the model fitting
- Shuffle inputs, avoid overfitting

Scenario 1: 1 input file, data fits in memory

- Simplest: import dataset in memory and convert to numpy arrays
- TensorFlow: Some options with native compatibility (tf.data API)
- PyTorch: torch.Dataset and torch.DataLoader (docs)

Data loading

Scenario 2: >1 file or data does not fit in memory

- Need to implement custom data loading machinery
- Frequently discussed, asked on support forums (1, 2, 3)
- Performance fine-tuning needs increasingly more time and care (TF best practices)

Some **functionality** is **missing**, giving space for different communities to propose **data loading frameworks**
HuggingFace Datasets, Weaver (HEP-specific),
mlx-data, Ray Data, …

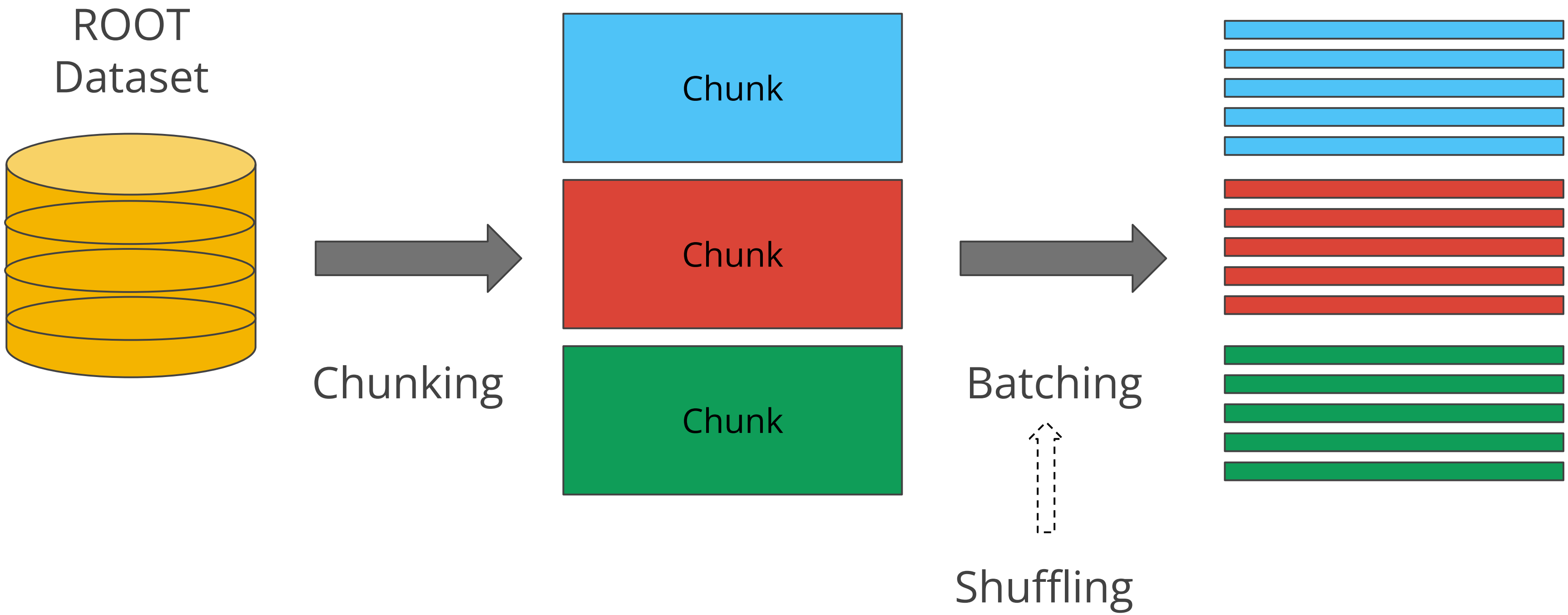
Native ML data loading with ROOT

Native ROOT data loading

- **ROOT**: storage, I/O, processing, scientific analysis of **structured** data
- **EBs** data stored in **ROOT** format
- Missing a **native**, coherent and **simple data loading** end-user experience

Native ROOT data loading

Provide a native data loading abstraction to pipe ROOT data (TTree, RNTuple) into ML training workflows (e.g. PyTorch, TF)



Native ROOT data loading

Provide a native data loading abstraction to pipe ROOT data (TTree, RNTuple) into ML training workflows (e.g. PyTorch, TF)

- **Asynchronous** loading (C++ thread)
- Supports **scalar** inputs as well as **collections**
- **Native** ROOT I/O: can read **any** HEP **EDM, local** or **remote** files
- **No** need for **pre-conversion** step to other data formats thus **no duplication**
- Integrated with **RDataFrame** for batch **preprocessing**

Usage

```python
# Returns two generators that return training
# and validation batches as PyTorch tensors.
gen_train, gen_validation =
ROOT.TMVA.Experimental.CreatePyTorchGenerators(
    rdataframe, batch_size, chunk_size,
    columns=features+labels, target=labels,
    max_vec_sizes=100, validation_split=0.3,
)
# [...] Create PyTorch model
for x_train, y_train in gen_train:
    # Make prediction and calculate loss
    pred = model(x_train)
    loss = loss_fn(pred, y_train)
```

Usage

```python
# Returns two generators that return training
# and validation batches as numpy arrays.
gen_train, gen_validation =
ROOT.TMVA.Experimental.CreateNumpyGenerators(
    rdataframe, batch_size, chunk_size,
    columns=features+labels, target=labels,
    max_vec_sizes=100, validation_split=0.3,
)
# [...] Create model
for x_train, y_train in gen_train:
    # Make prediction and calculate loss
    pred = model(x_train)
    loss = loss_fn(pred, y_train)
```

Usage

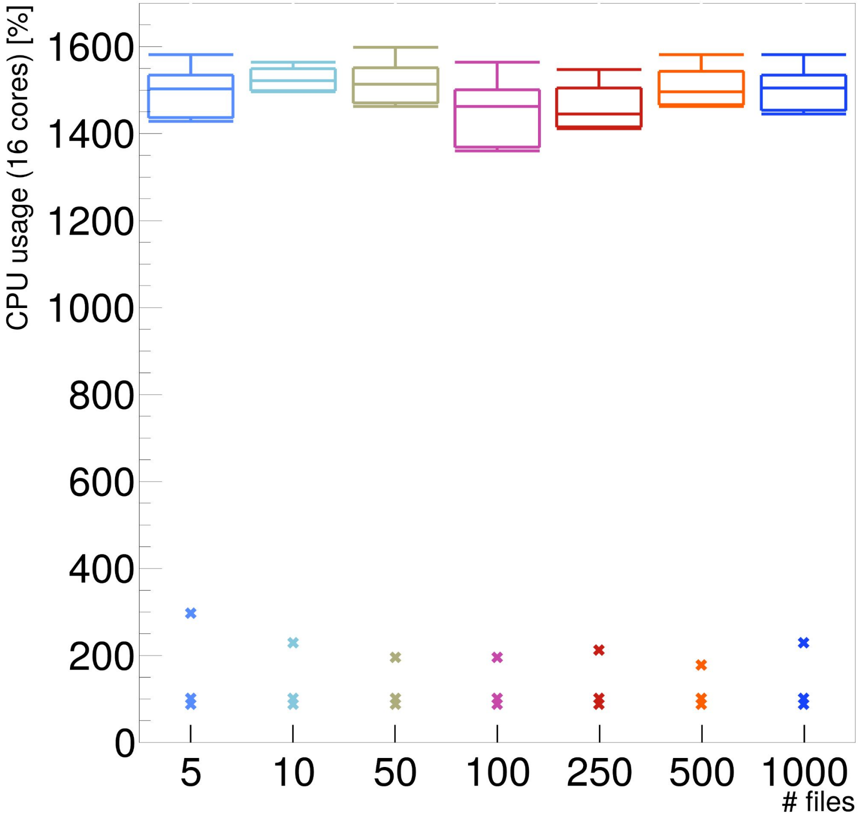
```python
# Returns two generators that return training
# and validation batches as TensorFlow Datasets.
gen_train, gen_validation =
ROOT.TMVA.Experimental.CreateTFDatasets(
    rdataframe, batch_size, chunk_size,
    columns=features+labels, target=labels,
    max_vec_sizes=100, validation_split=0.3,
)
# [...] Create TensorFlow model
for x_train, y_train in gen_train:
    # Make prediction and calculate loss
    pred = model(x_train)
    loss = loss_fn(pred, y_train)
```

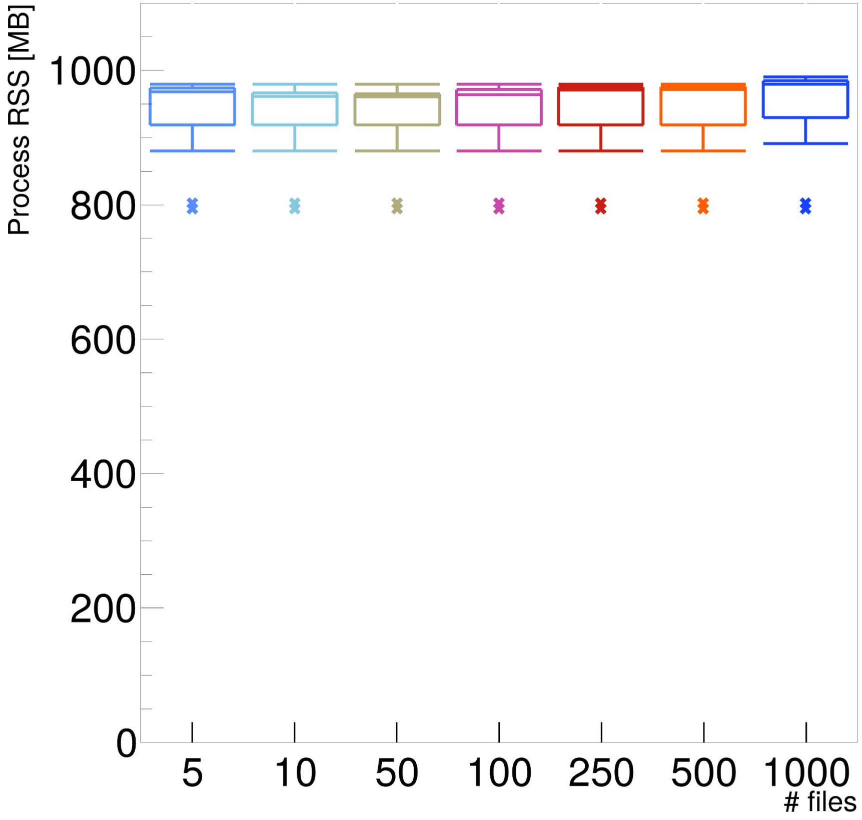
Performance evaluation

Benchmark setup

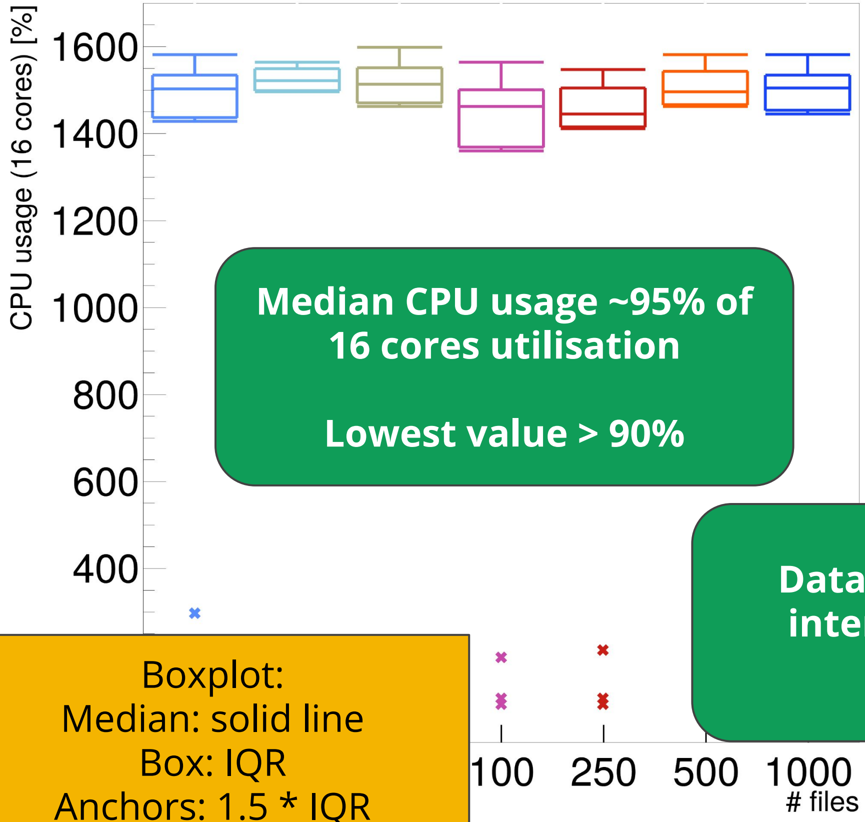
- Dataset: [JetClass]
  - 100 M jets, 1000 files, 142 GB
- Hardware
  - AMD Ryzen 9 5950X 16-Core, 64 GB RAM
- Benchmark
  - Load data via `CreatePyTorchGenerators`
  - Feed the tensors into a `torch.DataLoader`
  - Train a simple CNN with loaded data
  - Monitor CPU and Memory usage (sampling rate 1 Hz)
- Goal
  - Assess data loading performance, not the ML results

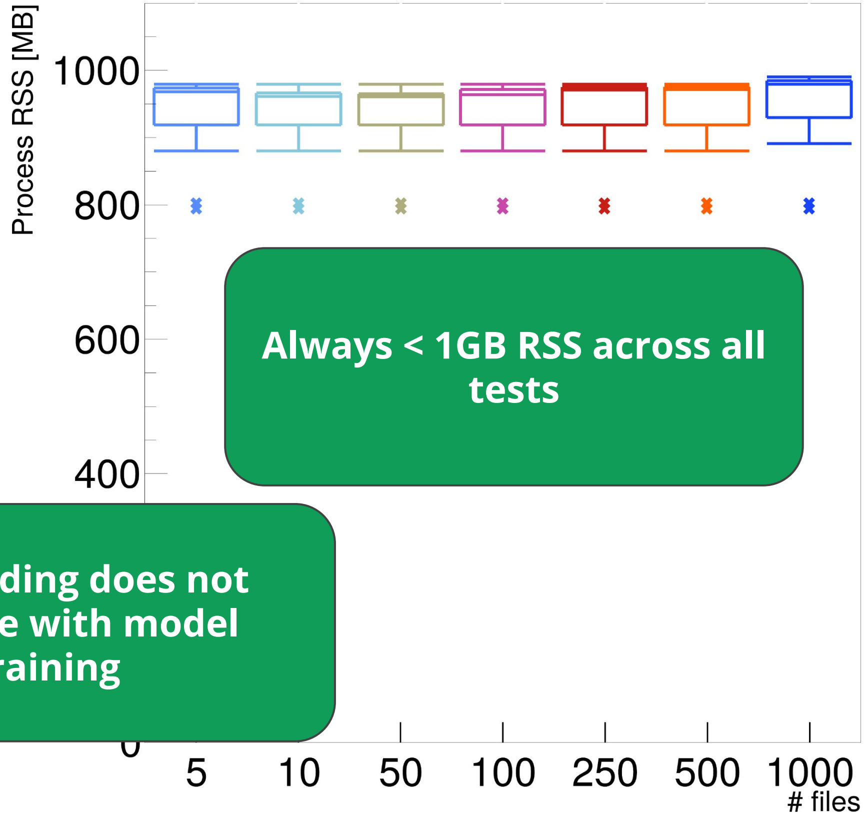
Data loading with multiple files





Data loading with multiple files





Median CPU usage ~95% of 16 cores utilisation

Lowest value > 90%

Always < 1GB RSS across all tests

Data loading does not interfere with model training

Boxplot:
Median: solid line
Box: IQR
Anchors: 1.5 * IQR

Conclusions

Conclusions

- Introduced a **native data loading** pipeline from **ROOT** data to **ML** model **training** workflows
- **Asynchronous loading**, native ROOT I/O
- **Easy** output to **PyTorch** tensors, **TensorFlow** Dataset, **numpy** arrays.

Want to know more? Eager to try it out? Do you have suggestions for improvements? Would you like to contribute?

Meet me around CHEP! And feel free to contact me at

`vincenzo.eduardo.padulano@cern.ch`