EPJ Web of Conferences **337**, 01097 (2025) https://doi.org/10.1051/epjconf/202533701097
CHEP 2024

Zero-overhead training of machine learning models with ROOT data

Vincenzo Eduardo Padulano[1,*], *Kristupas* Pranckietis[1,2], and *Lorenzo* Moneta[1]

[1]CERN, Esplanade des Particules 1, 1211 Geneva 23, Switzerland
[2]Vilnius University, 3 Universiteto St., LT-01513 Vilnius

Abstract. The ROOT software framework is widely used in High Energy and Nuclear Physics (HENP) for storage, processing, analysis and visualization of large datasets. With the large increase in usage of ML for experiment workflows, especially lately in the last steps of the analysis pipeline, the matter of exposing ROOT data ergonomically to ML models becomes ever more pressing. This contribution presents the advancements in an experimental component of ROOT that exposes datasets in batches ready for the training phase. This feature avoids the need for intermediate data conversion and can further streamline existing workflows, facilitating direct access of external ML tools to the ROOT input data in particular for the case when it does not fit in memory. The goal is to keep the footprint of using this feature minimal, in fact it represents just an extra line of code in user application. The contribution demonstrates the usage of the tool in an ML model training scenario, also evaluating the performance with key metrics.

1 Introduction

There is a long-standing tradition of synergies between the field of Particle Physics and the research and literature coming from Machine Learning (ML). For example, Neural Networks can be found being employed in the early '90s to analyse and discover new insights from data collected by physics experiments such as ALEPH at LEP [1] and D0 at TevaTron [2]. Over the past few decades, the scope and number of applications of machine learning techniques to typical HEP problems has only grown larger. Physicists have successfully employed ML models for different purposes among which classification, reconstruction, simulation and others. Evidence of this can be found in literature, including publications from the four main LHC collaborations [3–6].

In many cases, one of the principal challenges that can be encountered during the training phase of the model that will be employed for further research is accessing the data. Models might need to access tens or hundreds of gigabytes, if not more, to be tuned properly for their task. Furthermore, the dataset itself might be the result of some previous data preparation step. Various examples can be found across the community, such as open data released by an experiment [7], a dataset created ad-hoc for jet tagging tasks [8], or the datasets being released as part of ML challenges [9]. The data formats can differ depending on the application, but

*e-mail: vincenzo.eduardo.padulano@cern.ch

usually they are either in the ROOT TTree data format [10] (widely used in the field, with more than 2 EB stored in this format), in HDF5 [11] or in Apache Parquet [12].

Depending on the specific case, data access may become an overhead in both the development of the model and its training. In its simplest form, it adds an extra burden on the researcher which is not related to the actual application at hand. This could be addressed by separating responsibilities and ensuring smooth data pipelines to streamline the development workflow. In Section 2, we analyse the scenarios that result in concrete data access challenges, showing currently used approaches and possible solutions. Then, Section 3 introduces a native data loading tool to read datasets written in the ROOT TTree format into ML models for training. A first performance evaluation is demonstrated in Section 4. Finally, conclusions and future work are discussed in Section 5.

2 Data loading pipelines for ML model training

Clearly the issue of loading data into the model does not pertain only to the High Energy Physics cases. In the general case, one can identify a set of common steps involved in the training phase. These are summarised in Figure 1.

```
raw_data ← LOAD_DATA                                  ▷ Load data in memory
prepared_data ← PREPROCESS_DATA(raw_data)             ▷ Run preprocessing steps
train_data, validation_data ← SPLIT_DATA(prepared_data)
for n = 1, ..., n_epochs do                           ▷ Train the model on multiple epochs
    for all train_batch ∈ train_data do MODEL_TRAIN(train_batch)
```

Figure 1. Typical ML model training workflow.

Some caveats apply to this type of workflow. For example, a model needs to read the same data multiple times (*n_epochs* variable in Figure 1) in order to learn its characteristics. Between epochs, data should also be shuffled so that the model does not incur in overfitting. Also, the function responsible for loading the data must take into account the nature of the data itself. Questions regarding the size and location of data become of paramount importance. Furthermore, proper in-memory representation of data on disk through tensors of the right shape for the model at hand is also important. Often, data must be further preprocessed to reach the right representation, which might need interacting with other components of the application or third-party libraries. Clearly then, data access represents the required condition for the rest of the training to proceed properly, but also something to take into account during the training itself. For example, if the required data needs to be accessed remotely, this operation might become costly and subject to network latency. In such a scenario, having to wait for each training batch (see the for loop in Figure 1) to be loaded sequentially from the remote location before being able to proceed with the iteration would be unfeasible. This could be for example tackled by introducing asynchronous I/O operations that do not block the model training while reading data from the source.

Taking into account all the challenges described above, we could then identify two main scenarios that may be encountered in a training workflow. In the first scenario, the whole dataset fits into one file on disk and its size is less than the total memory available on the machine performing the training. This is the simplest case, since data can be read from disk and the only challenge is providing a suitable in-memory representation as a tensor of the right shape. To provide a concrete example, a large part of the scientific community interested in ML workflows is using the Python programming language. Within a Python application,

a dataset fitting in memory may be read and piped directly into a tensor represented as a multi-dimensional array of the popular library `numpy` [13]. Other popular libraries for ML in Python are TensorFlow [14] and PyTorch [15]. They both offer public APIs to load a file into the tensors of the right shape and prepare them for training[1].

In the second scenario, the dataset is made up of multiple files, or it cannot fit in the memory of a single machine. In this scenario, the data loading engine must be able to connect multiple files to the same dataset schema, possibly reading only a portion of a file at a time to meet memory constraints. It is also quite likely that such a large dataset would be stored in a remote location, separated from the machine running the training. While in practice this might happen in many different scientific applications, there is no user-friendly native support coming from the popular Python libraries. While in some cases the user guides provide best practices to follow, these include long recipes that need anyway to be adapted to the specific user application [16]. As a consequence, the topic is often discussed in the ML community and it has opened a space for different data loading frameworks to emerge and overcome these limitations. These frameworks differ in scope and functionality. Some aim at being more generic and serve many communities in their ML needs [17], sometimes even providing databases of different datasets that can be shared and used on different models [18, 19]. Others focus on the specific needs of a certain scientific community, as it happens for example with the `weaver` package [20] which allows configurations that is relevant for HEP.

3 Native ML data loading with ROOT

Many of the datasets used in HENP to train ML model come from experiment-wide data production pipelines. Most Physics experiments in the field write their data in the ROOT TTree data format, in fact it is currently estimated that more than 2 EB of data are written in this format. So far, there has been no native solution that could address the challenges described in Section 1 and 2 in an efficient and, most importantly, user-friendly way. In this paper we introduce a new data loading abstraction native to ROOT that is devoted to making the data access phase of a training workflow as simple as possible for the user, while allowing configuration and customization according to the requirements of the model. This abstraction is comprised of two main components:

- A front-end component, representing the user-facing API. This is a series of Python functions that allow taking in a ROOT dataset and return multiple Python generators, that will lazily load parts of the dataset as batches at every iteration of the training for loop.
- A back-end component, which the users do not interact with. This is a C++ implementation of an asynchronous data loader which parallelises the I/O requests to read in new chunks of the dataset in a C++ thread separate from the main Python thread. This allows the model to be trained on the current batch while the I/O engine takes care of loading the next one. The data access pattern that results from this machinery is exemplified in Figure 2.

The front-end part is currently composed of three main functions that users can employ in their application. Listing 1 shows the signature and options for the function `CreateNumPyGenerators`. The other two available functions support the same options while adding a convenience for the user to get generators conforming to the data loading API of their favourite ML tool: respectively `CreateTFDatasets` for TensorFlow, returning `tensorflow.data.Dataset` and `CreatePyTorchGenerators` for PyTorch, returning generators of `torch.Tensor`. As a first argument, the functions accept a node of a ROOT

[1]TensorFlow offers the `tf.data` API, whereas PyTorch offers `torch.Dataset` and `torch.DataLoader`.

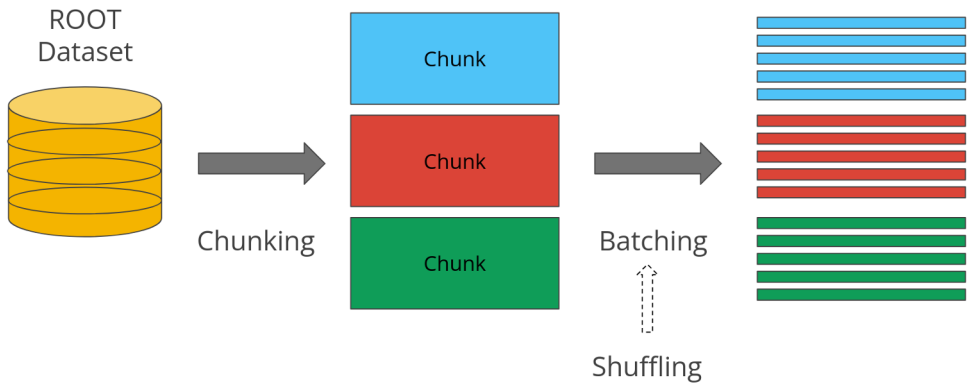

Figure 2. Schematic representation of the back-end component of the ROOT data loading abstraction for ML training.

```python
def CreateNumPyGenerators(
    rdataframe: ROOT.RDF.RNode,
    batch_size: int,
    chunk_size: int,
    columns: list[str] = list(),
    max_vec_sizes: dict[str, int] = dict(),
    vec_padding: int = 0,
    target: str | list[str] = list(),
    weights: str = "",
    validation_split: float = 0.0,
    max_chunks: int = 0,
    shuffle: bool = True,
    drop_remainder=True
) -> Tuple[TrainSetGenerator, ValidationSetGenerator]:
```

Listing 1: Signature of the Python API for the data loading abstraction. This function returns generators of (multi-dimensional) `numpy.array`.

RDataFrame computation graph. RDataFrame is the high-level interface to data analysis provided natively by ROOT [21]. The advantage given by RDataFrame is that any preprocessing steps required by the workflow can be encapsulated in a computation graph that will be run lazily by the RDataFrame object itself. Other parameters are more related to the shape of the tensors needed by the model, which columns of the dataset should be used as features or targets, the size in number of rows that each batch should be, as well as the split between training and validation or whether the batches should be shuffled or not between epochs. Once the customization options are chosen, what is left in the application is only the training of the model, thus abstracting away all the cumbersome details regarding data access. This can be appreciated by taking a look at the example shown in in Listing 2, where two Python

generators of `torch.Tensor` objects are created and used to feed the tensors to a PyTorch model in a seamless way.

```
# Returns two generators that return training
# and validation batches as PyTorch tensors.
gen_train, gen_validation = 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)
```

Listing 2: Example application training a PyTorch model and getting the input data as Python generators of `torch.Tensor` objects using the native ROOT abstraction.

4 Performance evaluation

A first evaluation of the abstraction layer has been carried out. The test setup is as follows:

- Dataset: The `JetClass` [8] dataset aimed at facilitating jet tagging tasks was used. This dataset includes one hundred million jets, scattered over one thousand files, with a total dataset size of one hundred forty two GB for training.
- Hardware: A single machine with an AMD Ryzen 9 5950X 16-Core CPU and 64 GB RAM.

The dataset is stored on the local machine. It is loaded via PyTorch `torch.Tensor` generators using the abstraction described in Section 3. It is then used to train a simple CNN with one layer. The process is constantly monitored for relevant statistics such as CPU and memory usage. The objective is verifying in practice that the back-end is capable of performing I/O operations without interfering with the main Python thread training the PyTorch model. Figure 3 and 4 show that this is the case. In particular, the first figure shows a consistently high CPU usage on the entire machine independently on the number of files used as input. This means that the main thread is always crunching data and never waiting for the next batches of dataset rows to arrive. The second figure instead shows that the entire process is never using more than 1 GB RAM in total for the entire duration of the training, irrespective of the number of files. This shows that the back-end is not loading the entire dataset in memory, but lazily traversing it one chunk at a time and always clearing the memory used by already processed batches in order not to hoard it as the training process continues.

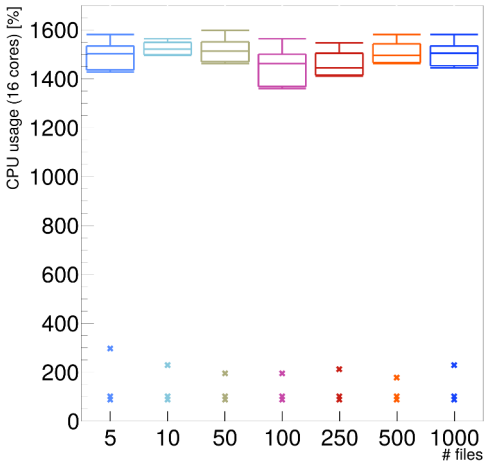

Figure 3. CPU usage in percentage (up to 16 cores) when loading an increasing number of files into a simple CNN using the ROOT abstraction. For every different number of file, the box represents the second and fourth quartile of the distribution, whereas whiskers represent five percent and ninety-five percent of the distribution. Outliers are indicated by crosses.

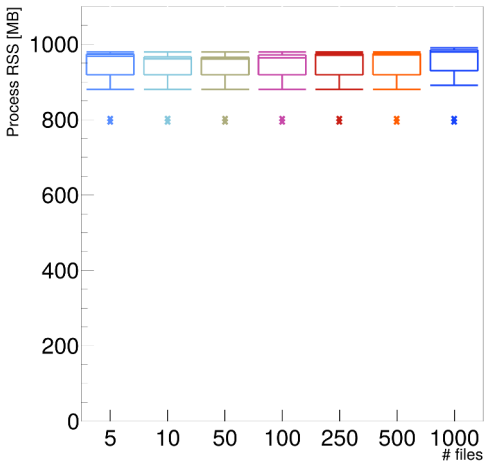

Figure 4. RSS used by the process when loading an increasing number of files into a simple CNN using the ROOT abstraction.

5 Conclusions

This contribution introduces a new data loading abstraction in ROOT aimed at simplifying the user experience in ML training workflows. It provides a public API which allows customization of the configuration and options required to prepare the data for the ingestion into the model, including preprocessing through RDataFrame. Behind the scenes, the abstraction leverages a powerful back-end that employs asynchronous I/O in a separate C++ thread to make sure the main thread is never blocked waiting for the next batch of rows to process. At

the time of publication, the feature is available as an experimental API in ROOT, with examples usages available in the documentation [2]. Next steps for this technology include testing with different datasets in the context of more realistic training scenarios for HEP experiments including reading large datasets stored remotely, ensuring its fitness for production use.

References

[1] V. Gaitan Alcalde, Ph.D. thesis, Autón. U. (1993)
[2] C.S. Lindsey, B.H. Denby, H. Haggerty, K. Johns, Real Time Track Finding in a Drift Chamber with a VLSI Neural Network, Nucl. Instrum. Meth. A **317**, 346 (1992). 10.1016/0168-9002(92)90628-H
[3] Łukasz Kamil Graczykowski, M. Jakubowska, K.R. Deja, M. Kabus, Using Machine Learning for Particle Identification in ALICE (2022), 2204.06900, https://arxiv.org/abs/2204.06900
[4] J. Gonski, Learning by machines, for machines: Artificial Intelligence in the world's largest particle detector (2024), https://atlas.cern/Updates/Feature/Machine-Learning
[5] D. Valsecchi, 2nd CERN IT ML workshop Report from CMS (2023), https://indico.cern.ch/event/1298990/#11-cms
[6] T. Likhomanenko, P. Ilten, E. Khairullin, A. Rogozhnikov, A. Ustyuzhanin, M. Williams, LHCb Topological Trigger Reoptimization, Journal of Physics: Conference Series **664**, 082025 (2015). 10.1088/1742-6596/664/8/082025
[7] CMS Collaboration, CMS releases open data for Machine Learning (2019), http://opendata.cern.ch/docs/cms-releases-open-data-for-machine-learning
[8] H. Qu, C. Li, S. Qian, Particle transformer for jet tagging (2024), 2202.03772, https://arxiv.org/abs/2202.03772
[9] G. Kasieczka, B. Nachman, D. Shih, R&D Dataset for LHC Olympics 2020 Anomaly Detection Challenge (2020)
[10] R. Brun, F. Rademakers, ROOT — An object oriented data analysis framework, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**, 81 (1997), New Computing Techniques in Physics Research V. https://doi.org/10.1016/S0168-9002(97)00048-X
[11] The HDF Group, Hierarchical Data Format, version 5, https://github.com/HDFGroup/hdf5
[12] D. Vohra, Apache Parquet (Apress, Berkeley, CA, 2016), pp. 325–335, ISBN 978-1-4842-2199-0, https://doi.org/10.1007/978-1-4842-2199-0_8
[13] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith et al., Array programming with NumPy, Nature **585**, 357 (2020). 10.1038/s41586-020-2649-2
[14] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin et al., TensorFlow: Large-scale machine learning on heterogeneous systems (2015), software available from tensorflow.org, https://www.tensorflow.org/
[15] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski et al., PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation, in *29th ACM International Conference on Architectural Support for Programming Languages and Operat-*

[2] https://root.cern/doc/v636/RBatchGenerator__NumPy_8py.html

ing Systems, Volume 2 (ASPLOS '24) (ACM, 2024), https://pytorch.org/assets/pytorch2-2.pdf

[16] TensorFlow developers, Better performance with the tf.data API, https://www.tensorflow.org/guide/data_performance

[17] Hugging Face developers, Hugging Face Datasets, https://huggingface.co/docs/datasets/index

[18] A. Hannun, J. Digani, A. Katharopoulos, R. Collobert, MLX: Efficient and flexible machine learning on apple silicon (2023), https://github.com/ml-explore

[19] Ray developers, Ray Data, https://docs.ray.io/en/latest/data/data.html

[20] weaver-core developers, weaver-core, https://github.com/hqucms/weaver-core

[21] D. Piparo, P. Canal, E. Guiraud, X. Valls Pla, G. Ganis, G. Amadio, A. Naumann, E. Tejedor Saavedra, RDataFrame: Easy Parallel ROOT Analysis at 100 Threads, EPJ Web Conf. **214**, 06029 (2019). 10.1051/epjconf/201921406029