ORIGINAL ARTICLE

MLaaS4HEP: Machine Learning as a Service for HEP

Valentin Kuznetsov[1] · Luca Giommi[2] · Daniele Bonacorsi[2]

Abstract

Machine Learning (ML) will play a significant role in the success of the upcoming High-Luminosity LHC (HL-LHC) program at CERN. An unprecedented amount of data at the exascale will be collected by LHC experiments in the next decade, and this effort will require novel approaches to train and use ML models. In this paper, we discuss a Machine Learning as a Service pipeline for HEP (MLaaS4HEP) which provides three independent layers: a data streaming layer to read High-Energy Physics (HEP) data in their native ROOT data format; a data training layer to train ML models using distributed ROOT files; a data inference layer to serve predictions using pre-trained ML models via HTTP protocol. Such modular design opens up the possibility to train data at large scale by reading ROOT files from remote storage facilities, e.g., World-Wide LHC Computing Grid (WLCG) infrastructure, and feed the data to the user's favorite ML framework. The inference layer implemented as TensorFlow as a Service (TFaaS) may provide an easy access to pre-trained ML models in existing infrastructure and applications inside or outside of the HEP domain. In particular, we demonstrate the usage of the MLaaS4HEP architecture for a physics use-case, namely, the $t\bar{t}$ Higgs analysis in CMS originally performed using custom made Ntuples. We provide details on the training of the ML model using distributed ROOT files, discuss the performance of the MLaaS and TFaaS approaches for the selected physics analysis, and compare the results with traditional methods.

Keywords BigData · LHC · Data management · Machine learning

Introduction

With the CERN LHC program underway, we started seeing an acceleration of data growth in the HEP field. By the end of Run II, the CERN experiments were already operating at the Peta-Byte (PB) level, producing $O(100)$ PB of data each year. The new HL-LHC program will extend it further, to the Exa-Byte scale, and the usage of ML in HEP will be critical [1]. ML techniques have been successfully used in online and offline reconstruction programs, and there is a huge gain in applying them to detector simulation, object reconstruction, identification, Monte Carlo (MC) generation, and beyond [2]. As was pointed out in the ML in HEP Community White Paper [1] the lack of engagement from Computer Science experts to address HEP ML challenges is partly due to the fact that HEP data are stored in ROOT data format, which is mostly unknown outside of the HEP community. The ROOT data format [3] is used to store HEP events in tree-based data structures, where the size of individual events cannot be determined a-priory, e.g., the number of electrons varies in each physics event. On a contrary, the existing ML frameworks rely on fixed-size data representation of individual events, usually stored in CSV [4], NumPy [5], HDF5 [6] data formats. Therefore, the event-based data structures cannot be fed directly to existing ML frameworks and special care should be taken either at the framework or at the data input level discussed in this paper. This and other reasons led to an artificial gap between ML and HEP communities. For example, in recent Kaggle challenges [7–9] the HEP data was presented in CSV data format to allow non-HEP ML practitioners to compete.

Here, we discuss the Machine Learning as a Service (MLaaS) architecture for HEP, referred to as MLaaS4HEP in this paper, which consists of two individual parts. The first part, the MLaaS4HEP framework [10], provides a way

✉ Luca Giommi
  luca.giommi3@unibo.it

  Valentin Kuznetsov
  vkuznet@gmail.com

  Daniele Bonacorsi
  daniele.bonacorsi@unibo.it

[1] Cornell University, Ithaca, USA

[2] University of Bologna and INFN, Bologna, Italy

to read HEP ROOT-based data natively into the Python ML framework of user choice. And, the second part, the TensorFlow as a Service (TFaaS) framework [11], can be used to host pre-trained ML models and obtain predictions via HTTP protocol.

This approach can be used by physicists or experts outside of HEP domain, because it only relies on Python libraries. It provides access to local or remote data storage, and does not require any modification or integration with the experiment's specific framework(s). Such modular design opens up a possibility to train ML models on PB-size data sets remotely accessible from the WLCG sites without requiring data transformation, i.e., from ROOT data format to flat data format and subsequent storage of these data sets to be used by underlying ML framework(s). Therefore, an existing gap between HEP and ML communities can be easily reduced using the discussed MLaaS architecture.

The organization of this paper is the following. Section 2 provides a summary of related works and the key aspects of the proposed solution. Section 3 presents the details of the MLaaS4HEP architecture and its workflow. Section 4 shows performance results and validation of MLaaS4HEP for a physics use-case. Section 5 outlines possible future directions, and Sect. 6 presents the summary.

Related Works and Solutions

Machine Learning as a Service is a well-known concept in industry, and major IT companies offer such solutions to their customers. For example, Amazon ML, Microsoft Azure ML Studio, Google Prediction API and ML engine, and IBM Watson are prominent implementations of this concept (see [12]). Usually, Machine Learning as a Service is used as an umbrella of various ML tasks such as data pre-processing, model training and evaluation, and inference through REST APIs. Even though providers offer plenty of interfaces and APIs, most of the time these services are designed to cover standard use-cases, e.g., natural language processing, image classifications, computer vision, and speech recognition. Although a custom ML codebase can be supplied to these platforms, its usage for HEP is quite limited for several reasons. For instance, the HEP ROOT data format cannot be used directly in any service provider's APIs. Therefore, the operational cost, e.g., data transformation from ROOT files to data format used by MLaaS provider APIs, data management, and data pre-processing, can be significant for large data sets. The data flattening from dynamic size event-based tree format to fixed-size data representation does not exist. Therefore, we found that out-of-the-box commercial solutions most often are not applicable or ineffective for HEP use-cases (costwise and functionalitywise). This might change in the future, as various initiatives, e.g., CERN

OpenLab [13], continue to work in close cooperation with almost all aforementioned service providers.

At the same time, various R&D activities within HEP are underway. For example, the hls4ml project [14] targets ML inference on FPGAs, while the SonicCMS project [15] is designed as Services for Optimal Network Inference on Co-processors. Both are targeted to the optimization of the inference phase rather than the whole ML pipeline, i.e., from reading data to training models and serving predictions.

Another solution uses the Spark platform for data processing and ML training [16]. Although it seems very promising, it requires data ingestion into the CERN EOS filesystem or the HDFS/Spark infrastructure. As such, there is no easy way to access data located at WLCG sites or from outside of such dedicated infrastructure. Besides, a Spark-based library (Analytics Zoo, BigDL) may be required on top of Keras API, and flexibility of ML framework choice is limited on the user side.

As introduced in the previous section, quite often additional transformations are required either at the data or software framework level. For example, in CMS a Deep Neural Network (DNN) used for a jet tagging algorithm relies on the TensorFlow (TF) queue system with a custom operation kernel for reading ROOT trees and feeding them to ML models like TensorFlow [17]. Although it represents an interesting approach for a specific use-case, it is not an "as a Service" solution.

In the end, we found that there is no final product that can be used as Machine Learning as a Service for distributed HEP data without additional efforts which can provide transparent integration with existing Python-based ML frameworks to perform ML training over HEP data, and this work aims to close this gap.

Novelty of the Proposed Solution

The original contribution of the proposed solution is the following.

1. We provide a transparent access to HEP data sets stored in the event tree-based ROOT data format into existing Python-based ML frameworks of user's choice. Usually, they are designed to operate with row-based data structures like NumPy arrays, CSV files and alike. The proposed solution discussed in Sects. 3.1 and 3.2 relies on the uproot library [18] and XrootD protocol [19] for reading small or large tree-based ROOT files from local filesystem or remote sites. It transforms the Jagged Arrays[1] representation of ROOT data, and feeds

---

[1] Jagged Array is an array of arrays of which the member arrays can be of different sizes, see Sect. 3.2 for more details.

it into ML framework via vector or matrix-based transformations applied to the I/O stream. This opens up a possibility to use favorite non-HEP ML frameworks like PyTorch [20], Keras [21], TensorFlow [22], fast.ai [23], etc., train ML models using distributed data sets, and, therefore, attracting non-HEP ML practitioners to be engaged in HEP ML activities.

2. In Sect. 4.2, we show that the proposed solution can be used as a valid alternative to a traditional HEP analysis based on custom flat tuples (derived from the production ROOT files). Then, in Sect. 4.3, we provide performance studies for a specific analysis, and in Sect. 4.4, we discuss the performance projection for very large data sets[2].

3. We developed an independent Tensor as a Service framework [11], as a part of this work, which provides access to any kind of Tensor-based ML models via HTTP protocol. Although similar functionalities exist in various industry solutions most of them are integrated as a part of their service stack which may not be affordable or accessible to research communities, where an efficient, scalable open-source alternative is desired to have.

4. Finally, the proposed modular architecture can be easily adapted among any HEP experiment either as an entire pipeline or be used partially, without requiring changes in existing frameworks or infrastructure.

MLaaS4HEP Architecture

A typical ML workflow consists of three steps: acquire the data necessary for training, use a ML framework to train the model, and utilize the trained model for predictions. In our Machine Learning as a Service solution, MLaaS4HEP [10], this workflow can be abstracted as data streaming, data training, and inference phases, respectively. Each of these components can be either tightly integrated into the application design, or composed and used individually. The choice is mostly driven by particular use-cases. We can define these layers as follows (see Fig. 1).

– **Data Streaming Layer:** it is responsible for reading local and/or remote ROOT files, and streaming data batches into the Data Training Layer. The implementation of this layer requires the ROOT I/O layer with the support of remote I/O file access;

– **Data Training Layer:** it represents a thin wrapper around standard ML libraries such as TensorFlow,

PyTorch, and others. It receives data from the Data Streaming Layer in chunks, transforms them from the ROOT TTree-based representation to the format suitable for the underlying ML framework, and uses it for training purposes;

– **Data Inference Layer:** it refers to the inference part of the service architecture. We implemented it as TFaaS [11] which allows to upload pre-trained models, and query them from a client side application using HTTP protocol.

Even though the implementation of these layers can differ from one experiment to another (or other scientific domains), it can be easily generalized and be part of the foundation for a generic Machine Learning as a Service framework. The MLaaS4HEP framework [10] implements the Data Streaming and Data Training layers, and we provide their details in Sects. 3.1 and 3.2, respectively. In Sect. 3.3, we provide technical details of the ML training workflow implemented in the MLaaS4HEP framework and used for our studies presented in Sect. 4. The data inference layer is implemented as independent TFaaS [11] framework, since it can be used outside of HEP, and its details are discussed in Sect. 3.4.

Data Streaming Layer

The Data Streaming Layer is responsible for streaming data from local or remote data storage. Originally, the reading of ROOT files was mostly possible from C++ or PyRoot frameworks, but the recent development of ROOT I/O significantly simplifies and speed up access to ROOT data from Python. The main development was done in the uproot [18] framework supported by the DIANA-HEP initiative [25]. The uproot library uses NumPy [5] calls to rapidly cast data blocks in ROOT file as NumPy arrays. It allows, among the implemented features, a partial reading of ROOT TBranches, non-flat TTrees, non TTrees histograms, and more. It relies on data caching and parallel processing to achieve high throughput. The data can be read from local ROOT files or remotely via XrootD protocol [19].

In our implementation of machine learning as a service (see Sect. 3.5) this layer is composed as a Python Generator [26] which is capable of reading chunk of data either from local or remote file(s). The output of this Python Generator is a NumPy array with flat and Jagged Array attributes. Such implementation provides efficient access to large data sets, since it does not require loading the entire data set into the RAM of the training node. In addition, it can be used to parallelize the data flow into the ML workflow pipeline. The choice of chunk size should be driven by complexity of the processed events, available network bandwidth and hardware resources, see discussion in Sect. 4.3.

---

[2] Even though it is possible to apply this approach for data sets at TB to PB scale an additional fine-tuning may be required to parallelize the data flow into the ML pipeline, see Sect. 4.3 and 4.4 for further details.
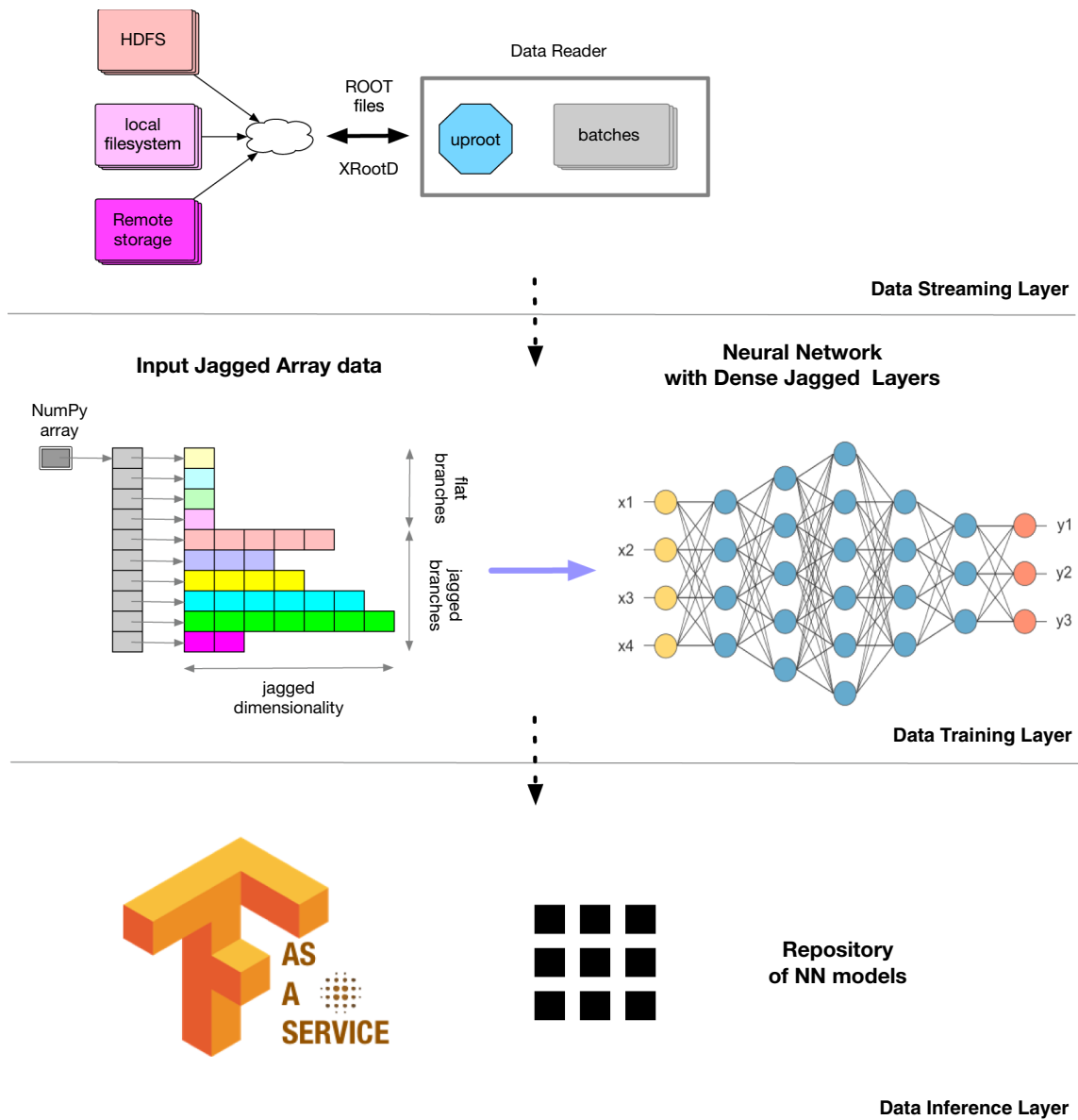
Fig. 1 MLaaS4HEP architecture diagram representing three independent layers: a Data Streaming Layer (top) to read local or remote ROOT files, a Data Training Layer (middle) to feed tree-based HEP data into ML framework, and a Data Inference Layer (bottom) via TensorFlow as a Service [24]

Data Training Layer

This layer transforms HEP ROOT data presented by the Data Streaming Layer as Jagged Array into a flat data format used by the application [1, 17]. The Jagged Array (see Fig. 2) is a compact representation of variable size event data produced in HEP experiments.

The HEP tree-based data representation is optimized for data storage but it is not directly suitable for ML frameworks. Therefore, a certain data transformation is required to feed tree-based data structures into the ML framework as a flat data structure. We explored two possible transformations: a

vector representation with padded values (see Fig. 3) and a matrix representation of data within the phase space of user choice (see Fig. 5).

The HEP events have different dimensionality across event attributes. For instance, a single event may have a different number of particles. Therefore, proper care should be done to flatten and padding ROOT events in the Jagged Array representation. For that, we use a two-passes procedure. In the first pass across all the events we determine the dimensionality of each attribute and its min/max values. Even though this procedure may not be feasible for very large data sets, i.e., at Tera or Peta-Byte scale, it can be
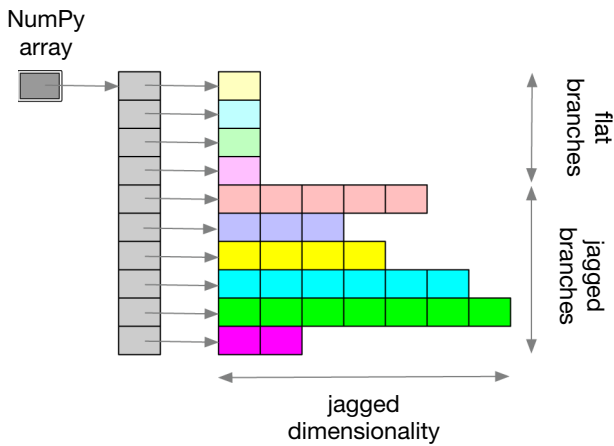
Fig. 2 Jagged Array data representation. It consists of flat attributes followed by Jagged attributes whose dimensions vary event by event [24]

easily replaced by alternative approaches with approximated min/max and clipping procedures. In the second pass we map Jagged Array attributes into a single vector representation with proper size and padding (see Fig. 3). In addition, we provide a proper normalization of each attribute during this phase. This layer can be easily abstracted as a Python decorator to allow multiple implementations of normalization procedure that can be provided directly by the user.

We also keep a separate masking vector (see Fig. 4) to distinguish assigned padded (e.g., NaN or zeros) values from the real values of the attributes. This may be important in certain kinds of Neural Networks, e.g., AutoEncoders (AE) [27], where the location of padded values in the input vector can be used in the decoding phase.

Alternatively, a matrix representation can be obtained from a Jagged Array (see Fig. 5). For example, the spatial coordinates are often part of HEP data sets and, therefore,

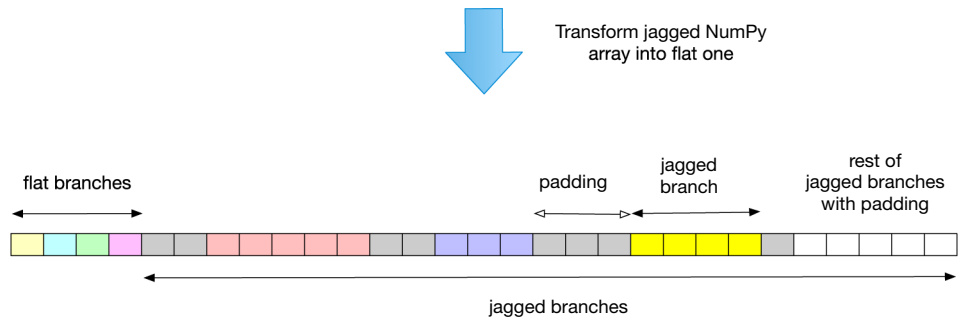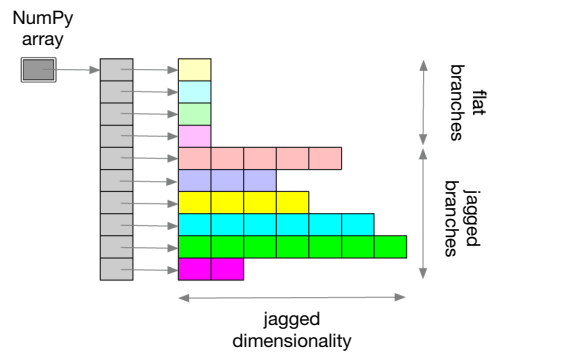**Fig. 3** Vector representation of Jagged Array with padded values [24]

Fig. 4 Vector representation of Jagged Array along with corresponding mask vector [24]
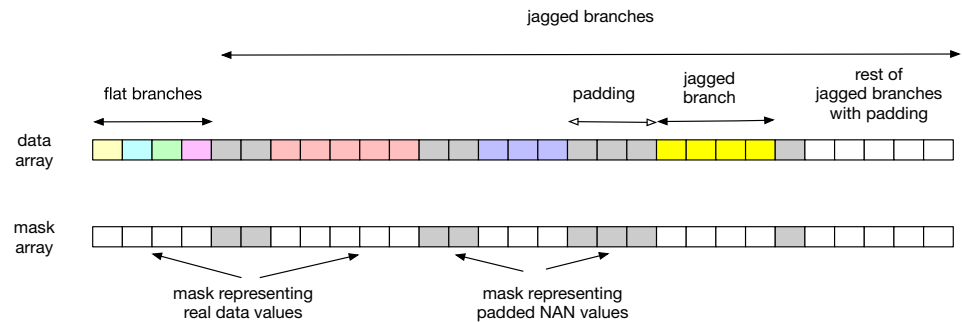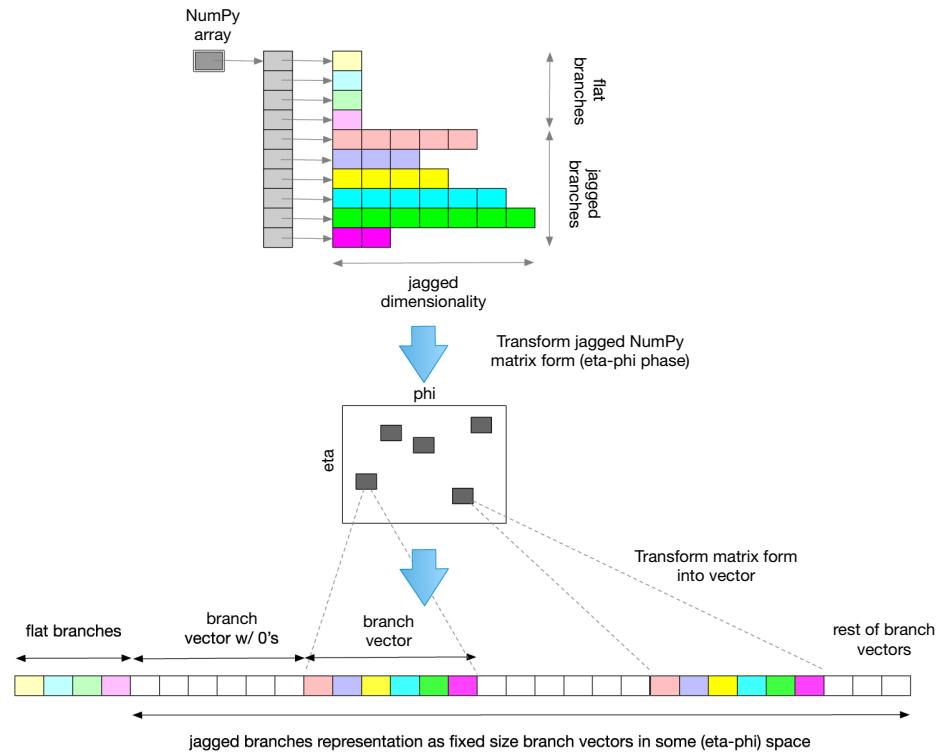
Fig. 5 Matrix representation of Jagged Array into certain phase space, e.g., eta-phi [24]

can be used for matrix representation of the events. This approach can resolve the issue of vector representation (of having to make a choice on the representation size) but it has its own problem with the choice of granularity of space matrix. For example, in the simplest case, a 2D matrix representation[3] (see Fig. 5) can be used in some X–Y phase space (where X and Y refer to an arbitrary pair of attributes). However, the cell size of this image is not known a-priory.

A choice of cell size may introduce a data collision issue within an event, e.g., different particles may have values of (X,Y) pair within the same cell. Such ambiguity may be resolved either by increasing matrix granularity or using an additional attribute, e.g., via higher dimensions of the cell space. But such changes will increase the sparsity of matrix representation and the matrix size and, therefore, will require more computing resources at training time.

Below we provide details of the MLaaS4HEP workflow used in the Data Streaming and Data Training layers using a vector representation for the results presented in Sect. 4.

ML Training Workflow Implementation

We implemented the Data Streaming and Data Training layers using the Python programming language and we made

them available in the MLaaS4HEP repository [10] under MIT license. The Data Training Layer was abstracted to support any kind of Python-based ML frameworks: TensorFlow, PyTorch, and others[4].

We used two parameters to control the data flow within the framework. The N_{chunk} parameter controls the chunk size of data read by the Data Streaming Layer from local or remote storage. And, the N_{batch} parameter defines a batch size, namely, the number of events used by the underlying ML framework in each training cycle. We refer to chunk as a set of events read by the Data Streaming Layer while batch as a set of events used by the ML training loop.

To train the ML models defined by the user code (provided externally) the MLaaS4HEP framework uses data chunks with the proper proportion of events presented in the ROOT files. The schematic of the data flow used in the Data Streaming and Data Training layers is shown in Fig. 6.

The first pass (denoted by ① in Fig. 6) represents the reading part of the MLaaS4HEP pipeline to create a specs file. This part is performed by reading all the ROOT files in chunks (which size is fixed a priori by the user) so that the information stored in the specs file is updated chunk by chunk. The specs file contains all the information about the ROOT files: the dimension of Jagged branches, the

---

[3] In the general case the matrix representation can have any number of dimensions.

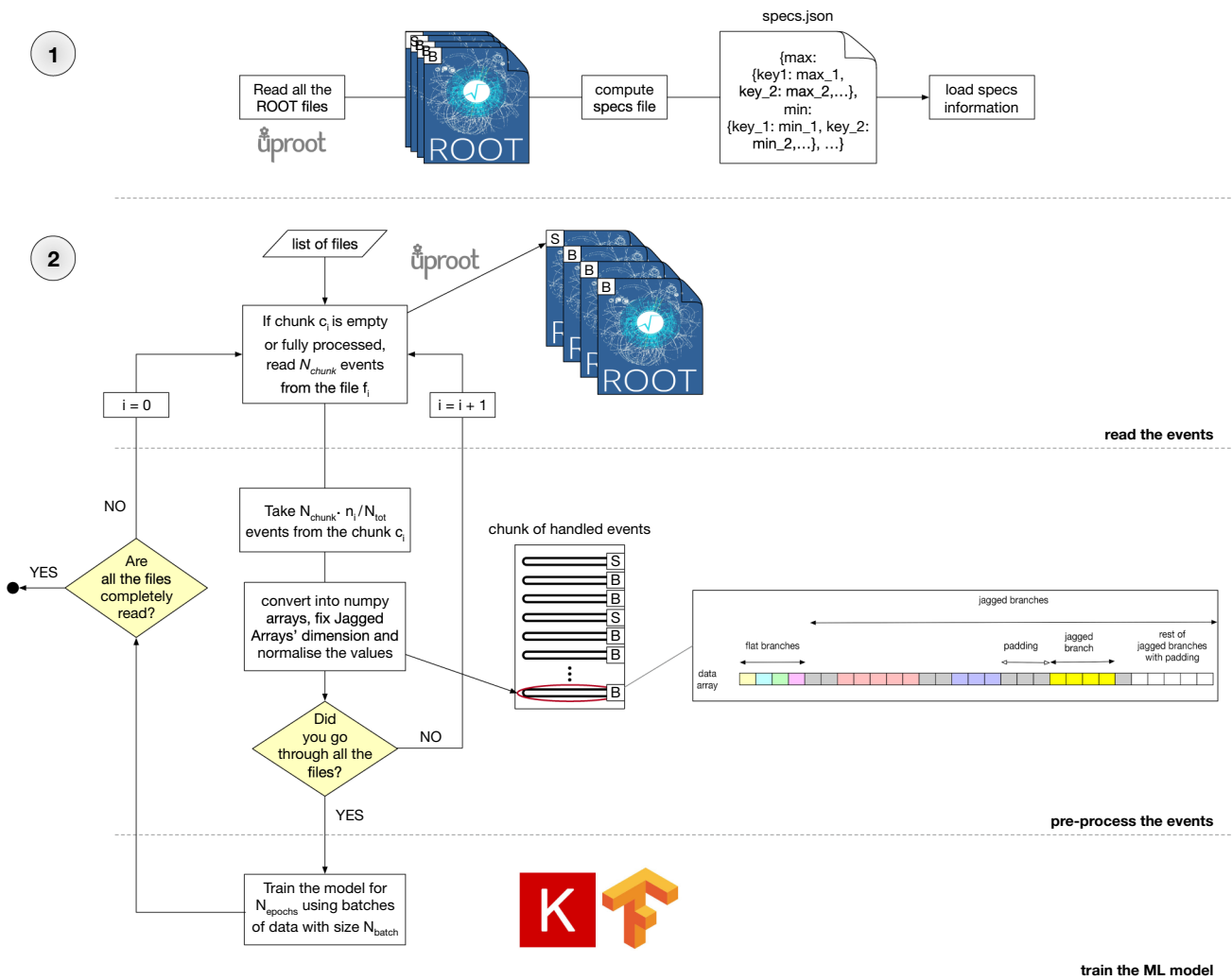[4] In all our tests we used Keras and PyTorch frameworks to define our ML models.

Fig. 6 Schematic representation of the steps performed in the MLaaS4HEP pipeline, in particular those inside the Data Streaming and Data Training layers (see text for details)

minimum and the maximum for each branch, and the number of events for each ROOT file[5].

The second part of the flowchart shown as ② represents the ML training phase. In the first loop of the cycle, when the events are not read yet, we read N_{chunk} events from the ith file f_i that we store into the ith chunk c_i. Then $N_{chunk} \cdot n_i/N_{tot}$ events are taken from it, where n_i is the number of events from file f_i and N_{tot} is the whole amount of events from all files. These events are converted into NumPy arrays, with the necessary transformation of the Jagged Arrays dimensions and normalization of the values (based on the

information computed during step ①). The reading of the events and their pre-processing is performed for all the files f_i. After having created a chunk of N_{chunk} events properly mixed from the different files, the events are used to train the ML model. The training phase is performed using batches of data taken from the created chunk, and run for a certain number of epochs. The batch size N_{batch} and the number of epochs N_{epochs} are fixed a priori by the user. In case N_{chunk} is not multiple of N_{batch}, the last batch used to train the model contains less than N_{batch} events. Then we come back at the beginning of the cycle, and if all the events stored in the chunk c_i have been already read, we read N_{chunk} events from the file f_i, otherwise we read the proper amount of events ($N_{chunk} \cdot n_i/N_{tot}$) from the chunk c_i. The training process continues and if files are not completely read the entire pipeline is restarted from the beginning of point ② until all events

---

[5] Once the specs file is produced, either through the aforementioned procedure or by studying Monte Carlo distributions (for large data sets) to determine attribute dimensions and their min/max values, it can be reused for all files from the given data set during the ML training phase.

are read, creating at each cycle a new chunk of events that is used to train the ML model for N_{epochs} epochs. At the end of the cycle, i.e., when we read all the events from all files and we completed the ML training for all the individual epochs, all the events contained in all files are read, and the training process of the model is completed, producing a model that can be used in physics analysis.

The discussed training procedure can be applied to a variety of use-cases. And, even though we used it in Sect. 4, it should not be viewed as the only way to train data sets using the MLaaS4HEP framework. We left to the end user the final choice of ML strategy for concrete use-cases, where appropriate steps should be taken to check the convergence of the model, a proper set of metrics to monitor the training cycle, etc. For instance, when a data set does not fit into the RAM of the training node other solutions can be adopted, e.g., using an SGD [28] model. In such case, the ML training workflow should be adapted to use the entire data set during each epoch. In this particular situation, the concept of batch and chunk would coincide.

Data Inference Layer

A data inference layer can be implemented in a variety of ways. It can be either tightly integrated with application frameworks (for example both CMS and ATLAS experiments followed this approach in their CMSSW-DNN [29] and LTNN [30] solutions, respectively) or it can be developed as a Service (aaS) solution. The former has the advantage of reducing latency of the inference step per processing event, but the latter can be easily generalized and become independent from internal infrastructure. For instance, it can be easily integrated into cloud platforms, it can be used as a repository of pre-trained models, and also serve models across experiment boundaries. However, the speed of the data inference layer, i.e., throughput of serving predictions, can vary based on the chosen technology. A choice of HTTP protocol guarantees easy adaptation, while gRPC protocol can provide the best performance but will require dedicated clients. We decided to implement the Data Inference Layer as a TensorFlow as a Service architecture [11] based on HTTP protocol.

We evaluated several ML frameworks and we decided to use TensorFlow graphs [22] for the inference phase. The TF model represents a computational graph in a static form, i.e., mathematical computations, graph edges, and data flow are well-defined at run time. Reading TF model can be done in different programming languages thanks to the support of APIs provided by the TF library. Moreover, the TF graphs are very well optimized for GPUs and TPUs. We opted for the Go programming language [31] to implement the inference part of the MLaaS4HEP framework based on the following factors: the Go language natively supports concurrency via *goroutines* and *channels*; it is the language developed and used by Google, and it is very well integrated with the TF library; it provides a final static executable which significantly simplifies its deployment on-premises and to various (cloud) service providers. We also opted out in favor of the REST interface. Clients may upload their TF models to the server and use it for their inference needs via the same interface. Both Python and C++ clients were developed on top of the REST APIs (end-points) and other clients can be easily developed thanks to HTTP protocol. The TFaaS framework can be used outside of HEP to serve any kind of TF-based models uploaded to TFaaS service via HTTP protocol[6].

MLaaS4HEP: Proof-of-Concept Prototype

When all layers of the MLaaS4HEP framework were developed, we successfully tested a working prototype of the system using ROOT files accessible through XrootD servers. The data were read in chunks of 1k events, where the single chunk was approximately 4 MB in size. We tested this prototype on a local machine as well as successfully deployed it on a GPU node. To further validate the MLaaS4HEP framework we decided to apply it to a real physics analysis, see Sect. 4, where we explored local and remote data access, usage of different data chunks, random access to files, etc.

Real Case Scenario

To validate the MLaaS4HEP approach, we decided to test the infrastructure on a real physics use-case. This allowed us to test the performances of the MLaaS4HEP framework, and validate its results from the physics point of view. We decided to use the $t\bar{t}$ Higgs analysis ($t\bar{t}H(b\bar{b})$) in the boosted, all-hadronic final state [32, 33] due to affinity with the analysis group. In the following sub-sections we discuss:

– the $t\bar{t}H(b\bar{b})$ all-hadronic analysis strategy (Sect. 4.1);
– MLaaS4HEP validation (Sect. 4.2);
– MLaaS4HEP performance results using the physics use-case (Sect. 4.3);
– MLaaS4HEP projected performance (Sect. 4.4);
– TFaaS performance results (Sect. 4.5).

---

[6] For instance, we tested the TFaaS functionality using non-HEP models such as image recognition ML models.
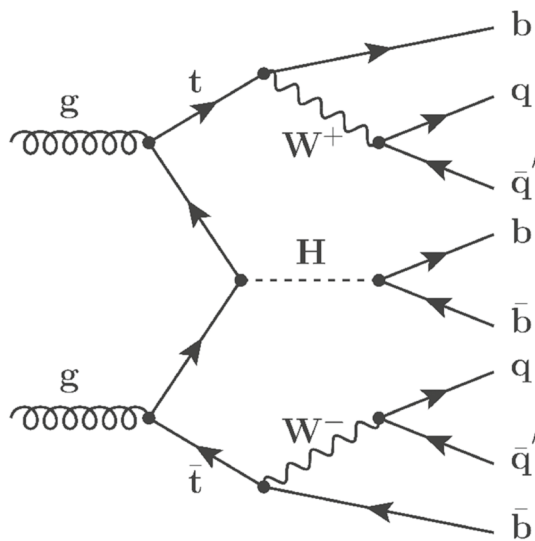
Fig. 7 Feynman diagram for the $t\bar{t}H(b\bar{b})$ decay

$t\bar{t}H(b\bar{b})$ all-hadronic Analysis Strategy

In this subsection, we provide details of the $t\bar{t}$ Higgs analysis we used to test the MLaaS performance and to validate its functionality on a real physics use-case.

The Higgs boson is considered the most relevant discovery of the last few years in High Energy Physics. After almost 50 years from its prediction, it was discovered by the ATLAS and CMS collaborations in 2012 at the CERN Large-Hadron Collider (LHC) [34, 35]. Since then, many analyses have been performed to measure its properties with higher precision.

In the Standard Model framework, the Higgs boson is predicted to couple with fermions via Yukawa-like interaction, which gives the mass to fermions proportionally to the coupling. The heaviest top quark is responsible for coupling to the Higgs boson. Direct measurement of the top-Higgs coupling exploits tree-level processes. The $t\bar{t}H$ production plays an important role in the study of the top-Higgs Yukawa coupling, as other production mechanisms (such as gluon–gluon fusion) involve loop-level diagrams in which contributions from Beyond Standard Model (BSM) physics could enter the loops unnoticed. The highest branching ratio ($\approx 25\%$) is represented by the all-hadronic decay channel with $H(b\bar{b})$ and all-hadronic $t\bar{t}$. The W bosons produced by the $t\bar{t}$ pair decay into a pair of light quarks, while the Higgs boson decays into a $b\bar{b}$ pair (see Fig. 7). In the final state, there are at least eight partons (more might arise from the initial and final state radiation) where four of them are bottom (b) quarks. Despite the highest branching ratio, the all-jets final state is very challenging. It is dominated by the large QCD

multi-jet production at LHC, and there are large uncertainties in this channel due to the presence of many jets. At the same time, it represents the unique possibility to fully reconstruct the $t\bar{t}H$ as all decay products are observable.

At the 13 TeV center-of-mass energy, top quarks with a very high p_T can be produced via $t\bar{t}H$. If their Lorentz boost is sufficiently high, their decay products are very collimated into a single, wide jet, named boosted jet. In particular, we are interested in the $t\bar{t}H(b\bar{b})$ analysis with all-jets final state, where at least one of the jets of the final state is a boosted jet, and where the Higgs boson decays in a pair of well resolved jets identified as a result of the hadronization of bottom quarks.

For identification of the $t\bar{t}H(b\bar{b})$ events containing a resolved-Higgs decay a Machine Learning model based on Boosted Decision Tree (BDT) was used by CMS in the analysis [32, 33] and the training was done within TMVA [36] framework.

The Monte Carlo simulation provides events used for training, where events are selected among the $t\bar{t}H$ sample and the two dominant background samples, namely, QCD and $t\bar{t}$, respectively. The $t\bar{t}H$ events with the resolved Higgs-boson matching to the system of two b-tagged jets are considered as signal events. On the contrary, unmatched $t\bar{t}H$ events, and all the QCD and $t\bar{t}$ events are considered as background events. Both signal and background events are required to pass some selection criteria, such as to have at least a boosted jet, to contain no leptons, to pass the signal trigger, etc. This selection is aimed to select boosted, all-jets-like events.

MLaaS4HEP Validation

To validate the MLaaS4HEP functionality against standard BDT-based procedure, we decided to use a set of ROOT files from the resolved-Higgs analysis discussed in the previous section. The goal of this exercise was to demonstrate that the MLaaS4HEP framework can provide a valuable alternative and deliver comparable results with respect to the traditional analysis based on a pre-defined set of metrics. For our purposes, we decided to use a generic ML model and compare the results obtained inside and outside MLaaS4HEP. In particular, we explored the following approaches:

- use MLaaS4HEP to read and normalize events, and to train the ML model;
- use MLaaS4HEP to read and normalize events, and use a Jupyter notebook to perform the training of the ML model outside MLaaS4HEP;
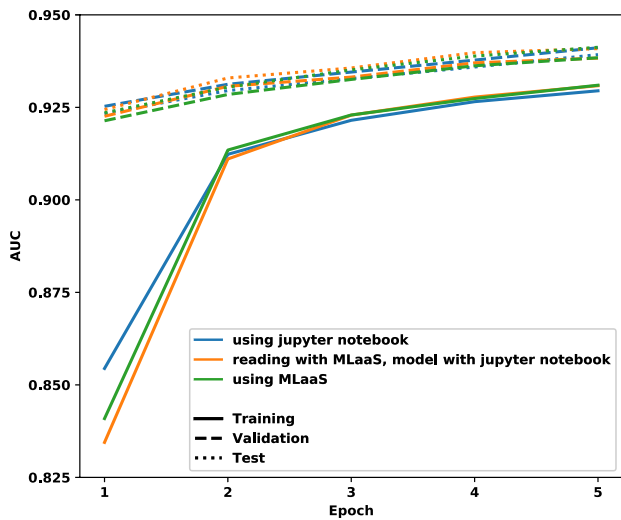- use a Jupyter notebook to perform the entire pipeline without using MLaaS4HEP.

Fig. 8 Comparison of the AUC score for the training, validation, and test set for three different cases: (i) using MLaaS4HEP to read and normalize events, and to train the ML model; (ii) using MLaaS4HEP to read and normalize events, and using a Jupyter notebook to perform the training of the ML model outside MLaaS4HEP; (iii) using a Jupyter notebook to perform the entire pipeline without using MLaaS4HEP

Initially, we performed the analysis using the ROOT files that passed the selection criteria discussed in the previous section. The final data set consisted of eight ROOT files containing background events, and one file containing signal events. Each file had 27 branches, with 350k events in total, and the total size of this data set was 28 MB. The ratio between the number of signal events and background events was approximately 10.8%. The data set was split into three parts, 64% for training, 16% for validation, and 20% for test purposes, respectively. We used a Keras [21] sequential Neural Network with two hidden layers made by 128 and 64 neurons, and with a 0.5 dropout regularization

between layers. Finally, we trained the model for 5 epochs with a batch size of 100 events.

The results of this exercise are shown in Fig. 8, and demonstrate that different approaches have similar performance. We did not target to reproduce and/or match exact AUC numbers obtained in the standard physics analysis, and we found that our result (in terms of AUC score) is comparable with the BDT model used in the physics analysis.

When the aforementioned ML model is trained using chunks of the data through the training workflow strategy described in Sect. 3.3, the convergence of the model is still valid. However, as we pointed out in Sect. 3.3, once the user chooses a specific physics use-case and a ML model, the convergence of the model should be verified and the appropriate ML training workflow should be adapted, if necessary.

MLaaS4HEP Performance

In this section, we provide details of the MLaaS4HEP performance testing: the scalability of the framework and its benchmarks using different storage layers. For that purpose, we used all available ROOT files without any physics cuts. This gave us a data set with 28.5M events with 74 branches (22 flat and 52 Jagged), and a total size of about 10.1 GB.

We performed all tests running the MLaaS4HEP framework on macOS (laptop), 2.2 GHz Intel Core i7 dual-core, 8 GB of RAM, and on CentOS 7 Linux, 4 VCPU Intel Core Processor Haswell 2.4 GHz, 7.3 GB of RAM CERN Virtual Machine. The ROOT files are read from three data centers: Bologna (BO), Pisa (PI), and Bari (BA). The average available bandwidth was approximately 129 Mbit/s with the Standard Deviation of Mean (SDOM) parameter equal to 4 Mbit/s and 639 (SDOM = 39) Mbit/s using macOS and CERN VM, respectively (in both cases the values are obtained after 10 trials).

Table 1 Performances of reading and specs computing phase with chunk size fixed to 100k events, using the macOS system and the CERN VM. Each value shown in the table cells represents the arithmetic mean of five trials with the corresponding SDOM reported inside the round brackets. The mean value of each event throughput is the mean of the values obtained chunk by chunk in the step ① in

Fig. 6. In local storage cases, the files are stored in a SSD 500 GB in the macOS case and in a Virtual Disk 52 GB in the CERN VM case, respectively. Moreover, BO, BA, and PI stand for various Italian storage facilities with different WAN configurations (see text for more details)

	Reading time (s)	Specs comp. time (s)	Time to complete Step ① (s)	Event throughput for Reading + specs comp. (evts/s)
macOS with local files	1633 (9)	958 (2)	2599 (11)	11055 (49)
macOS with remote files (BO)	2365 (49)	974 (10)	3353 (57)	8585 (149)
VM with local files	1131 (3)	963 (2)	2102 (5)	13690 (34)
VM with remote files (BO)	2455 (68)	959 (2)	3427 (67)	8396 (158)
VM with remote files (BA)	2304 (88)	961 (2)	3279 (89)	8801 (241)
VM with remote files (PI)	2129 (41)	1044 (78)	3186 (83)	9047 (228)

Fig. 9 Average event throughput for reading the data as a function of the chunk size for different trials of step ① in Fig. 6. The data points represent the arithmetic mean of five trials and the error bars are the corresponding SDOM
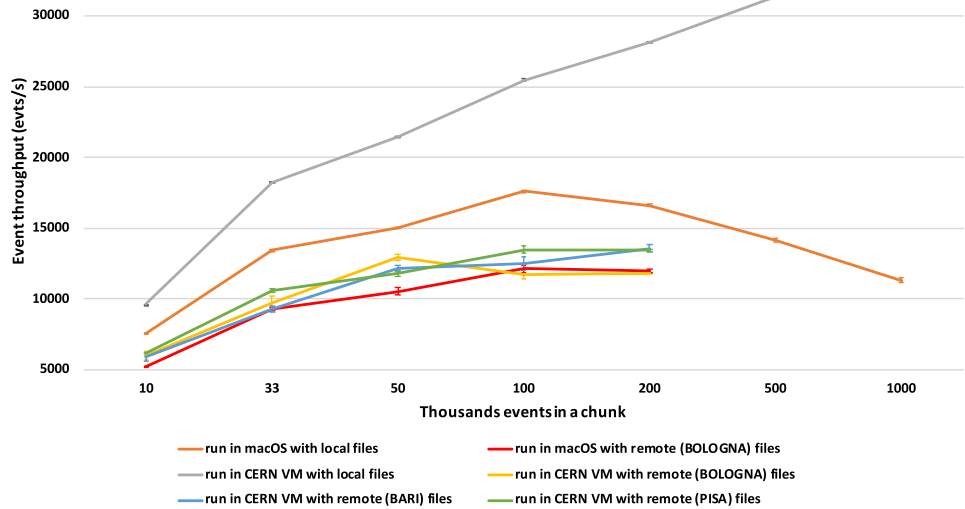

Table 2 Event throughput for the chunk creation and for the pre-processing step with a chunk size of 100k events computed as the ratio of the number of events over the time spent on chunk creation. The difference between the two steps is based on the reading part, i.e., the time for creating a chunk, as the sum of times for reading events from the ROOT files, and the time for the pre-processing step. Shown values represent the arithmetic mean of ten trials with the corresponding SDOM reported inside the round brackets

	Event throughput for Creating a chunk (evts/s)	Event throughput for Pre-processing a chunk (evts/s)
macOS with local files	1102 (11)	1157 (7)
macOS with remote files (BO)	1057 (17)	1138 (4)
VM with local files	1209 (11)	1247 (2)
VM with remote files (BO)	1110 (32)	1243 (5)
VM with remote files (BA)	1071 (19)	1153 (4)
VM with remote files (PI)	1152 (18)	1234 (5)

Table 1 summarizes the I/O numbers we obtained in the first step of the MLaaS4HEP pipeline (① in Fig. 6) using various setups and a chunk size of 100k events. It provides the values of time spent for reading the files, the time spent for computing specs values, the total time spent for completing the step ①, and the event throughput for the reading and specs computing step.

In Fig. 9, we show the event throughput for reading the data as a function of chunk size for different trials. In all cases, we find no significant peaks. The larger chunk sizes can lead to certain problems, as in the case of the CERN VMs, where we may reach a limitation of the underlying hardware, e.g., big memory footprint. We found lower reading times and higher event throughput using local files, while in the case of remote files, the results are mostly influenced by the available bandwidth (the link connectivity between processing node and sites hosting the data).

In the performance studies of the second step of the MLaaS4HEP pipeline (② in Fig. 6), we are interested in the data reading part, the data pre-processing step (which include data transformation), and the time spent in the MLaaS4HEP training step.

As already mentioned in Sect. 3.3, there is a loop over files that allows building the chunk used to train the ML model with the adequate proportion of the events. If the chunk that contains the events of the ith ROOT file is empty or fully processed, a new chunk of events from the ith file is read, and the time for reading is added to the whole time spent for creating the chunk (see Fig. 6). In other words, the time spent for creating a chunk is made by the sum of n reading actions, and of the time to pre-process the events. The event throughput for creating a single data chunk and the event throughput for pre-processing a single data chunk are reported in Table 2. In Fig. 10, we show the event throughput for creating a chunk as a function of the chunk size for different trials.

We found that the time spent for creating a chunk was almost the same using macOS or CERN VM, and similar using local or remote files. Obviously, for remote files, the reading time increased consequently, and the time for creating the chunk increased, but this difference was quite

Fig. 10 Event throughput for creating a chunk as a function of the chunk size for different trials. Shown values represent the arithmetic mean of ten trials with the SDOM error bars
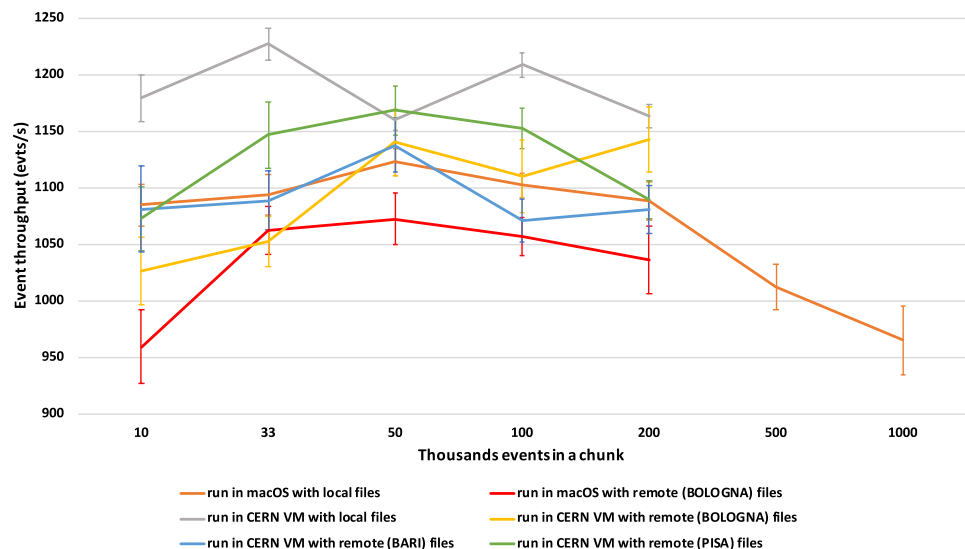
negligible. For instance, we spend around 90 s to create a chunk of 100k events, which translates into an event throughput of about 1.1k etvs/s as reported in Table 2.

The choice of the chunk size is left to the user and there is no pre-defined "best" value for it. We suggest that users should start with a lower value of chunk size, e.g., 1k, and increase it gradually based on their resource availability. For instance, in our initial proof-of-concept implementation, see Sect. 3.5, we used 1k events as a chunk data size, while within performance studies discussed in this section, we extended the chunk size to 100k events.

The actual ML training time is independent from the MLaaS4HEP framework, since it is determined by usage of the underlying ML framework, e.g., Keras or PyTorch, the complexity of used ML model and available hardware resources. In particular, using the simple ML model introduced in Sect. 4.2 and a chunk size of 100k events, we found that for each chunk the time spent to split properly the data for training, validation and test purpose is about 1s (and almost equal for MacOS and CERN VM), and the training time for 5 epochs is about 11s and 13s for MacOS and CERN VM, respectively.

During the implementation of the MLaaS4HEP framework, we resolved few bottlenecks with respect to the results obtained in [24]. For example, we improved the reading time by a factor of 10. This came from better handling of Jagged Arrays via flattening the event arrays and computing of min/max values of each branch. Moreover, we also obtained a factor of 2.8 improvements in the data pre-processing step using lists comprehensions instead of loops within the event. On MacOS the performance of the MLaaS4HEP framework is about 86s to pre-process 100k events with 36%, 26%, 27%, and 6% breakdown used to extract and convert each event in a list of NumPy arrays,

the normalization step, fixing the dimensions, and creating the masking vectors, respectively.

In conclusion, we demonstrated that MLaaS4HEP approach can be applicable to the discussed physics analysis. Using 10 GB of data (approximately 28.5M events) we obtained the following results:

– MLaaS4HEP framework is capable to work with local and remote files;
– its throughput reaches about 13.7k evts/s for reading local ROOT files (with specs computing), and about 9k evts/s for remote files;
– the throughput of the pre-processing step is peaked at 1.2k evts/s.

MLaaS4HEP Performance Projection

Based on our studies presented in the previous section, we found that to process 28.5M events (or 10 GB of data) MLaaS4HEP takes about 35 min for the first step of the pipeline (① in Fig. 6), i.e., to obtain min/max boundaries of all attributes across the processed events. The second step of the MLaaS4HEP pipeline (② in Fig. 6) takes about 7 h. This time includes reading all data chunks from the ROOT files, pre-process the events (data transformation from Jagged Array to flat NumPy arrays with fixing of the Jagged Arrays dimensions, data normalization), and feeding the data to the ML framework. The actual ML training time depends on the user-provided model and does not represent MLaaS4HEP performance. In our studies reported in Sect. 4.2, it adds an additional hour to the total time. Therefore, we estimate that using the same hardware resources the step ① will take $O(100)$ hours and $O(100k)$ hours for data sets at TB and PB scale, and the time for step ② will be $O(1k)$ hours and $O(1M)$ hours, respectively, plus time

required to train the ML model. These estimates suggest that further optimization of the MLaaS4HEP pipeline will be required to process TB or PB scale data sets and it may involve parallelization of I/O, distributed ML training, and other optimization techniques which we discuss further in Sect. 5.

At this stage, our goal was mainly to prove the feasibility of the MLaaS4HEP pipeline, and validate its usage within the context of a real physics use-case rather than perform real ML training at TB/PB scale. In Sect. 5, we discuss further improvements which can be done.

TFaaS Performance

The performance testing of the TFaaS service was done using a variety of ML models, from simple image classification to the ML model developed and discussed in Sect. 4.2. In particular, we performed several benchmarks using the TFaaS server running on CentOS 7 Linux, 16 cores, 30 GB of RAM. The benchmarks were done in two modes: using 1k calls with 100 concurrent clients and 5k calls with 200 concurrent clients. We tested both JSON and ProtoBuffer [37] data formats while sending and fetching the data to/from the TFaaS server. In both cases, we achieved a throughput of ∼ 500 req/sec. These numbers were obtained by serving the mid-size pre-trained model with 27 features and 1024x1024 hidden layers used in the physics analysis discussed in Sect. 4.1. A similar performance was found for image classification data sets (MNIST). The actual performance of TFaaS will depend on the complexity of the served ML model and the available hardware resources. Even though a single TFaaS server may not be as efficient as an integrated solution, it can be easily horizontally scaled, e.g., using Kubernetes or other cluster orchestrated solutions, and may provide the desired throughput for concurrent clients. It also decouples the application layer/framework from the inference phase which can be easily integrated into any existing infrastructure using the HTTP protocol.

Future Directions

In the previous section, we discussed the usage of MLaaS4HEP in the scope of a real HEP physics analysis. We found the following:

– the usage of MLaaS4HEP is transparent to the chosen HEP data set, i.e., data can be read locally or from remote storage;
– the discussed architecture is HEP experiment agnostic and can be used with any existing ML (Python-based)

framework as well as easily integrated into existing infrastructure;
– the data can be read in chunks from remote storage, and this allows continuous ML training over large data sets, and further parallelization.

These observations open up a possibility to train ML models over large data sets, potentially at Peta-Byte scale while using existing Python-based open-source ML frameworks. Therefore, we foresee that the Machine Learning as a Service approach can be widely applicable in HEP. For example, future directions of this work might include the exploitation of this architecture to streamline the access to cloud and HPC resources for training and inference tasks. It can represent an attractive option to open up HPC resources for large scale ML training in HEP along with required security measurements, resource provisioning, and remote data access to WLCG sites. To move in this direction additional work will be required. Below, we discuss a possible set of improvements that can be explored.

Data Streaming Layer

To improve the Data Streaming Layer a multi-threaded I/O layer can be implemented. This can be achieved by wrapping up the data reader code-base into a service that will deliver the data chunks in parallel upon requests from the upstream layer. In addition, the chunks can be pre-fetched from XrootD servers into a local cache to improve the I/O throughput. In particular, there are several R&D's underways to demonstrate intelligence smart caching [38] for Dynamic On-Demand Analysis Service (DODAS) at computer centers, such as HPC, national Tier centers, etc. Such a DODAS facility can reduce the time spent on the Data Streaming Layer by pre-fetching ROOT files into local cache and use them for ML training.

Data Training Layer

If data I/O parallelism can be achieved, further improvements can be made via implementation of distributed training [39]. There are several R&D developments in this direction, from adapting the Dask Python framework [40], or using distributed Keras [41], to using MPI-Based Python framework for distributed training [42], or using MLflow framework [43] on an HDFS+Spark infrastructure, which explores both task and data parallelism approaches.

The current landscape of ML frameworks is changing rapidly, and we should adjust MLaaS4HEP to existing and future ML framework and innovations. For instance, Open Network Exchange Format [44] opens up the door to migration of models from one framework into another. This may open up a possibility to use MLaaS4HEP for the

next generation of Open-Source ML frameworks and ensure that end-users will not be locked into a particular one. For instance, we are working on the automatic transformation of PyTorch [20] and fast.ai [23] models into TensorFlow which later can be uploaded and used through TFaaS service [11].

As discussed in Sect. 3.2, there are different approaches to feed Jagged Arrays into ML framework and R&D in this direction is in progress. For instance, for AutoEncoder models, the vector representation with padded values should always keep around a cast vector which later can be used to decode back the vector representation of the data back to Jagged Array or ROOT TTree data structures. We also would like to explore matrix representation of Jagged Array data and see if it can be applied to certain types of use-cases, e.g., in calorimetry or tracking, where image representation of the objects can be used.

Data Inference Layer

On the inference side, several approaches can be used. As discussed above, the TFaaS [11] throughput can be further improved by switching from HTTP to a gRPC-based solution such as SONIC [45] which can provide a fast inference layer based on FPGAs and GPUs-based infrastructures.

The current implementation of TFaaS can be used as a repository of pre-trained models which can be easily shared across experiment boundaries or domains thanks to serving ML models via HTTP protocol. For instance, the current implementation of TFaaS allows visual inspection of uploaded models, versioning, tagging, etc. We foresee the next logical step is towards a repository of pre-trained models with flexible search capabilities, extended model tagging, and versioning. This can be achieved by providing a dedicated service for ML models with proper meta-data description. For instance, such meta-data can capture model parameters, details of used software, releases, data input, and performance output. With a proper search engine in place, users may search for available ML models related to their use-case.

MLaaS4HEP Services

The proposed architecture allows us to develop and deploy training and inference layers as independent services. The separate resource providers can be used and dynamically scaled if necessary, e.g., GPUs/TPUs can be provisioned on-demand using the commercial cloud(s) for training purposes of specific models, while inference TFaaS service can reside elsewhere, e.g., on a dedicated Kubernetes cluster at some computer center. For instance, the continuous training of complex DL models would be possible when data produced by the experiment will be placed on WLCG sites. The training service will receive a set of notifications about newly available data, and re-train specific model(s). When a new ML model is ready it can be easily pushed to TFaaS and be available for end-users immediately without any intervention on the existing infrastructure as part of CD/CI (Continuous Development and Continuous Integration) workflows. The TFaaS can be further adapted to use FPGAs to speed up the inference phase. We foresee that such an approach may be more flexible and cost-effective for HEP experiments in the HL-LHC era. As such, we plan to perform additional R&D studies in this direction and evaluate further MLaaS4HEP services using available resources.

Summary

In this paper, we presented a modern approach to train HEP ML models using the native ROOT data-format either from local or remote storage. The MLaaS4HEP consists of three layers: the Data Streaming and Data Training layers as part of the MLaaS4HEP framework [10], and the Data Inference Layer implemented in the TFaaS framework based on the TensorFlow library. All three layers are implemented as independent components. The Data Streaming Layer relies on the uproot library for reading data from ROOT files (local or remote) and yielding NumPy (Jagged) arrays. The Data Training Layer transforms the input Jagged Array into a vector representation and passes it into the ML framework provided by the user. Finally, the Data Inference Layer was implemented as an independent HTTP service. We foresee that it can be useful in a variety of use-cases such as quick evaluation of ML models in physics analysis, or online applications, where new models can be built periodically. The TFaaS implementation allows to use itself as a repository of ML pre-trained models, and it can be a valuable component in the agile ML development cycle of any group, from small physics analysis group(s) to cross-experiment collaborations.

The flexible architecture we implemented allows performing ML training over a large set of distributed HEP ROOT data without physically downloading data into local storage. We demonstrated that such architecture is capable of reading local and distributed data sets, available via XrootD protocol on WLCG infrastructure. We validate the MLaaS4HEP architecture using an official CMS $t\bar{t}$ Higgs analysis ($t\bar{t}H(bb)$) in the boosted, all-hadronic final state, and we obtained comparable ML model performance with respect to a traditional physics analysis based on data extraction from ROOT files into custom Ntuples.

many other) packages which open up a possibility to work on the MLaaS4HEP implementation. We would like to thank Fabio Iemmi for the helpful discussions we had on the aspects of the physics use-case.

Data Availibility Statement This manuscript has associated data in a data repository. [Authors' comment: The details about data sets used in current study are available from the corresponding author on reasonable request. The availability of CMS data set itself is a subject of CMS policy].

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Code availability The code is available at [10, 11] under MIT license.

References

1. Albertsson K et al (2018) Machine Learning in High Energy Physics Community White Paper. arXiv:1807.02876 [physics.comp-ph]
2. A Living Review of Machine Learning for Particle Physics. https://iml-wg.github.io/HEPML-LivingReview/
3. A modular scientific toolkit used in HEP for analysis and as a data-storage format. https://root.cern.ch
4. Comma Separated Values (CSV) data-format. https://www.wikiwand.com/en/Comma-separated_values
5. Scientific package to represent data as multi-dimensional arrays. http://www.numpy.org
6. Hierarchical Data Format. https://www.wikiwand.com/en/Hierarchical_Data_Format
7. Higgs Boson Machine Learning Challenge used by ATLAS experiment to identify Higgs boson. https://www.kaggle.com/c/higgs-boson
8. High Energy Physics particle tracking in CERN detectors. https://www.kaggle.com/c/trackml-particle-identification
9. Flavours of Physics: Finding $\tau \to \mu\mu\mu$. https://www.kaggle.com/c/flavours-of-physics/data
10. Kuznetsov V, Giommi L (2018) MLaaS4HEP is set of MLaaS components for HEP. DOI: https://doi.org/10.5281/zenodo.1481785, https://github.com/vkuznet/MLaaS4HEP
11. Kuznetsov V (2018) TensorFlow as a Service. https://doi.org/10.5281/zenodo.1308049, http://github.com/vkuznet/TFaaS
12. Yao Y et al (2017) Complexity vs. performance: empirical analysis of machine learning as a service. In: Proceedings of the 2017 Internet Measurement Conference, pp 384-397. https://doi.org/10.1145/3131365.3131372
13. CERN Openlab. https://home.cern/science/computing/cern-openlab
14. A package for machine learning inference in FPGAs. https://hls-fpga-machine-learning.github.io/hls4ml
15. Services for Optimal Network Inference on Coprocessors. https://github.com/hls-fpga-machine-learning/SonicCMS
16. Machine Learning Pipelines for High Energy Physics Using Apache Spark with BigDL and Analytics Zoo. https://db-blog.web.cern.ch/blog/luca-canali/machine-learning-pipelines-high-energy-physics-using-apache-spark-bigdl
17. CMS Collaboration (2020) A deep neural network to search for new long-lived particles decaying to jets. Mach. Learn.: Sci. Technol. 1: 035012. DOI: https://doi.org/10.1088/2632-2153/ab9023
18. DIANA-HEP Scikit-hep uproot library. Minimalist ROOT I/O in pure Python and NumPy. https://github.com/scikit-hep/uproot
19. A high performance, scalable fault tolerant access to data repositories of many kinds. http://xrootd.org
20. PyTorch AI library. https://www.pytorch.org
21. Keras AI library. https://keras.io
22. Tensor Flow AI library. http://www.tensorflow.org
23. Fast AI library. https://www.fast.ai
24. Kuznetsov V (2018) Machine Learning as a Service for HEP. arXiv:1811.04492 [hep-ex]
25. An umbrella organization for bringing state-of-the art for HEP experiments. http://diana-hep.org
26. Python Generator concept. https://wiki.python.org/moin/Generators
27. Géron A (2019) Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly, ISBN: 9781492032632
28. Stochastic Gradient Descent. https://scikit-learn.org/stable/modules/sgd.html
29. DNN/TensorFlow interface for CMSSW. https://github.com/mharrend/CMSSW-DNN
30. ATLAS Lightweight Trained Neural Network. https://doi.org/10.5281/zenodo.4299114, https://github.com/lwtnn/lwtnn
31. Go programming language. http://www.golang.org
32. CMS Collaboration (2016) Search for $t\bar{t}$H production in the H $\to b\bar{b}$ decay channel with $\sqrt{s} = 13$ TeV pp collisions at the CMS experiment. CMS PAS HIG-16-004
33. Iemmi F (2020) $t\bar{t}H$ associated production in the all-jets final state with the CMS experiment. Nuovo Cim. C 43(2–3):77. https://doi.org/10.1393/ncc/i2020-20077-4
34. Collaboration ATLAS (2012) Observation of a New Particle in the Search for the Standard Model Higgs Boson with the ATLAS Detector at the LHC. Phys Lett B 716(1):1–29. https://doi.org/10.1016/j.physletb.2012.08.020
35. Collaboration CMS (2012) Observation of a New Boson at a Mass of 125 GeV with the CMS Experiment at the LHC. Phys Lett B 716(1):30–61. https://doi.org/10.1016/j.physletb.2012.08.021
36. Hoecker A, Speckmayer P, Stelzer J, Therhaag J, von Toerne E, Voss H (2007) TMVA: Toolkit for Multivariate Data Analysis. arXiv:physics/0703039 [physics.data-an]
37. ProtoBuffer library. https://github.com/protocolbuffers/protobuf
38. Tracolli M et al (2020) Using DODAS as deployment manager for smart caching of CMS data management system. J. Phys.: Conf. Ser. 1525: 012057. https://doi.org/10.1088/1742-6596/1525/1/012057
39. Ben-Nun T, Hoefler T (2018) Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. arXiv:1802.09941 [cs.LG]
40. Scalable Analytics framework in Python. https://dask.org
41. Distributed Keras framework. https://github.com/cerndb/dist-keras

42. Anderson D et al (2017) An MPI-Based Python Framework for Distributed Training with Keras. arXiv:1712.05878 [cs.DC]
43. An open source platform for the machine learning life cycle. https://www.mlflow.org
44. Open Neural Network Exchange format. http://www.onnx.ai
45. Duarte J et al (2019) FPGA-accelerated machine learning inference as a service for particle physics computing. Comput Softw Big Sci 3:13. https://doi.org/10.1007/s41781-019-0027-2