Machine Learning as a Service for HEP on heterogeneous computing resources

Luca Giommi[1] (luca.giommi3@unibo.it), Valentin Kuznetsov[2], Daniele Bonacorsi[1], Daniele Spiga[3]

[1] University of Bologna and INFN Bologna, Italy

[2] Cornell University, USA

[3]INFN Perugia, Italy

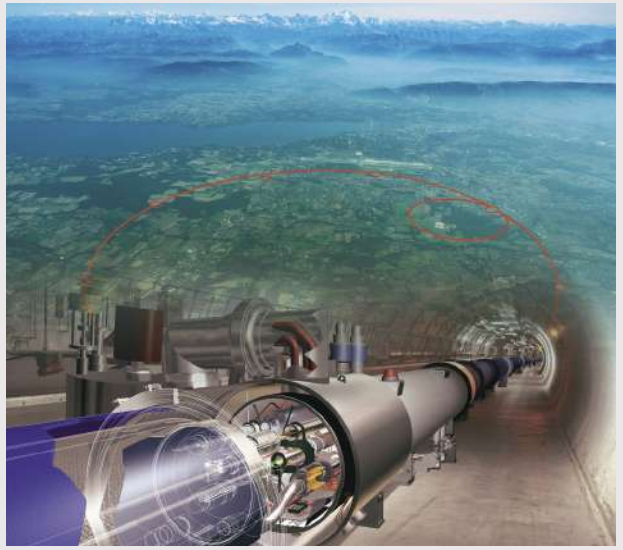
International Symposium on Grids & Clouds 2021, March 25th

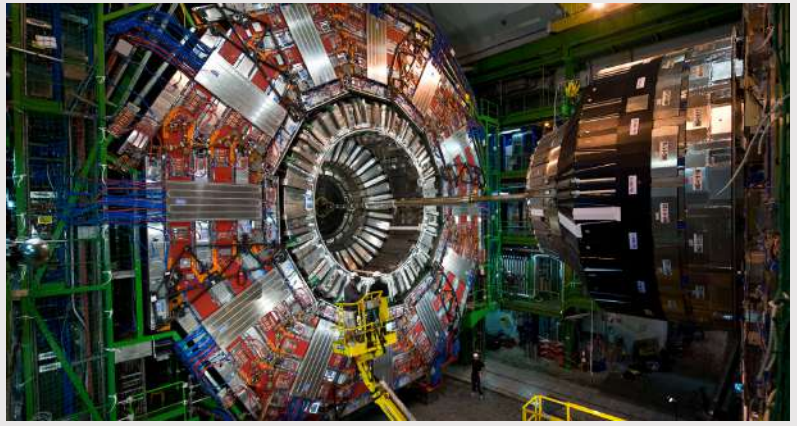
High Energy Physics at CERN

- The European Organization for Nuclear Research (CERN) was born in 1954 and it is based in the northwest suburb of Geneva on the Franco–Swiss border.

- At CERN is located the world's largest and most powerful particle accelerator called Large Hadron Collider (LHC).

- Inside the accelerator, two high-energy particle beams travel at close to the speed of light before they are made to collide at four locations, corresponding to the positions of four particle detectors: ATLAS, CMS, ALICE and LHCb.





Why Machine Learning as a Service?

- Machine Learning techniques in the HEP domain are ubiquitous (e.g. detector simulation, object reconstruction, particles identification, Monte-Carlo generation) and will play a significant role also in the upcoming High-Luminosity LHC upgrade.

- One of the main obstacles in this scenario is the real gap among HEP physicists and ML experts, caused by the specificity of some parts of the HEP typical workflows and solutions.

- To close this gap and ease the physicists not ML-practitioners in the usage of ML techniques in their analyses, we propose a Machine Learning as a Service for HEP (MLaaS4HEP) solution as a product of R&D activities within the CMS experiment.

Machine Learning in High Energy Physics Community White Paper

May 17, 2019

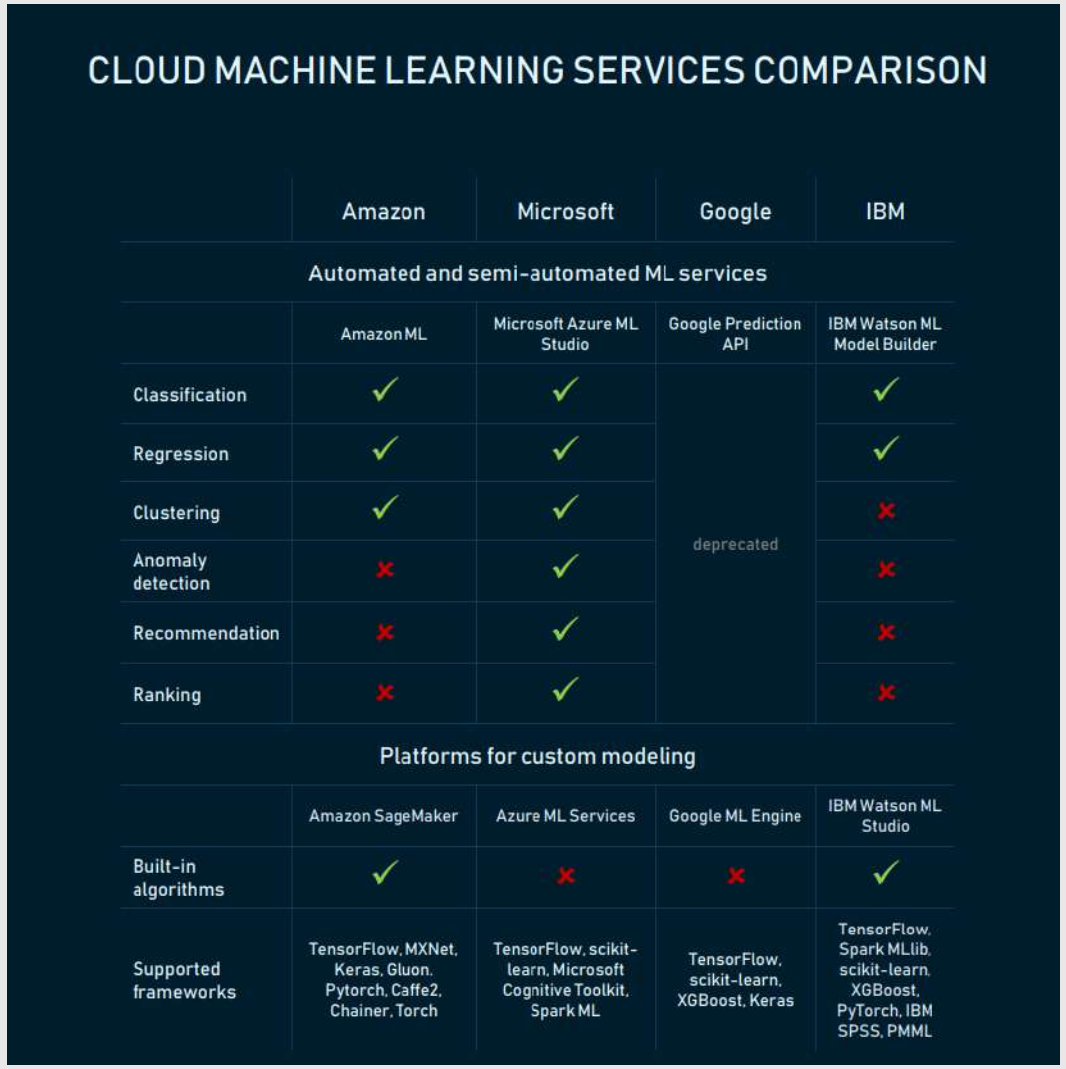
Abstract: Machine learning has been applied to several problems in particle physics research, beginning with applications to high-level physics analysis in the 1990s and 2000s, followed by an explosion of applications in particle and event identification and reconstruction in the 2010s. In this document we discuss promising future research and development areas for machine learning in particle physics. We detail a roadmap for their implementation, software and hardware resource requirements, collaborative initiatives with the data science community, academia and industry, and training the particle physics community in data science. The main objective of the document is to connect and motivate these areas of research and development with the physics drivers of the High-Luminosity Large Hadron Collider and future neutrino experiments and identify the resource needs for their implementation. Additionally we identify areas where collaboration with external communities will be of great benefit.

Editors: Sergei Gleyzer[30], Paul Seyfert[13], Steven Schramm[32]

Contributors: Kim Albertsson[1], Piero Altoe[2], Dustin Anderson[3], John Anderson[4], Michael Andrews[5], Juan Pedro Araque Espinosa[6], Adam Aurisano[7], Laurent Basara[8], Adrian Bevan[9], Wahid Bhimji[10], Daniele Bonacorsi[11], Bjorn Burkle[12], Paolo Calafiura[10], Mario Campanelli[9], Louis Capps[2], Federico Carminati[13], Stefano Carrazza[13], Yi-Fan Chen[4], Taylor Childers[14], Yann Coadou[15], Elias Coniavitis[16], Kyle Cranmer[17], Claire David[18], Douglas Davis[19], Andrea De Simone[20], Javier Duarte[21], Martin Erdmann[22], Jonas Eschle[23], Amir Farbin[24], Matthew Feickert[25], Nuno Filipe Castro[6], Conor Fitzpatrick[26], Michele Floris[13], Alessandra Forti[27], Jordi Garra-Tico[28], Jochen Gemmler[29], Maria Girone[13], Paul Glaysher[18], Sergei Gleyzer[30], Vladimir Vava Gligorov[31], Tobias Golling[32], Jonas Graw[2], Lindsey Gray[21], Dick Greenwood[33], Thomas Hacker[34], John Harvey[13], Benedikt Hegner[13], Lukas Heinrich[17], Ulrich Heintz[12], Ben Hooberman[35], Johannes Junggeburth[36], Michael Kagan[37], Meghan Kane[38], Konstantin Kanishchev[8], Przemysław Karpiński[13], Zahari Kassabov[39], Gaurav Kaul[40], Dorian Kcira[3], Thomas Keck[29], Alexei Klimentov[41], Jim Kowalkowski[21], Luke Kreczko[42], Alexander Kurepin[43], Rob Kutschke[21], Valentin Kuznetsov[44], Nicolas Köhler[36], Igor Lakomov[13], Kevin Lannon[45], Mario Lassnig[13], Antonio Limosani[46], Gilles Louppe[17], Aashrita Mangu[47], Pere Mato[13], Helge Meinhard[13], Dario Menasce[48], Lorenzo Moneta[13], Seth Moortgat[49], Meenakshi Narain[12], Mark Neubauer[35], Harvey Newman[3], Sydney Otten[50], Hans Pabst[40], Michela Paganini[51], Manfred Paulini[5], Gabriel Perdue[21], Uzziel Perez[52], Attilio Picazio[53], Jim Pivarski[54], Harrison Prosper[55], Fernanda Psihas[56], Alexander Radovic[57], Ryan Reece[58], Aurelius Rinkevicius[44], Eduardo Rodrigues[7], Jamal Rorie[59], David Rousseau[60], Aaron Sauers[21], Steven Schramm[32], Ariel Schwartzman[37], Horst Severini[61], Paul Seyfert[13], Filip Siroky[62], Konstantin Skazytkin[43], Mike Sokoloff[7], Graeme Stewart[63], Bob Stienen[64], Ian Stockdale[65], Giles Strong[6], Wei Sun[4], Savannah Thais[51], Karen Tomko[66], Eli Upfal[12], Emanuele Usai[12], Andrey Ustyuzhanin[67], Martin Vala[68], Sofia Vallecorsa[69], Justin Vasel[56], Mauro Verzetti[70], Xavier Vilasís-Cardona[71], Jean-Roch Vlimant[3], Ilija Vukotic[72], Sean-Jiun Wang[30], Gordon Watts[73], Michael Williams[74], Wenjing Wu[75], Stefan Wunsch[29], Kun Yang[4], Omar Zapata[76]

arXiv:1807.02876v3 [physics.comp-ph]
DOI: 10.1007/s41781-018-0018-8

Machine Learning as a Service



CLOUD MACHINE LEARNING SERVICES COMPARISON

	Amazon	Microsoft	Google	IBM
Automated and semi-automated ML services				
	Amazon ML	Microsoft Azure ML Studio	Google Prediction API	IBM Watson ML Model Builder
Classification	✓	✓	deprecated	✓
Regression	✓	✓		✓
Clustering	✓	✓		✗
Anomaly detection	✗	✓		✗
Recommendation	✗	✓		✗
Ranking	✗	✓		✗
Platforms for custom modeling				
	Amazon SageMaker	Azure ML Services	Google ML Engine	IBM Watson ML Studio
Built-in algorithms	✓	✗	✗	✓
Supported frameworks	TensorFlow, MXNet, Keras, Gluon, Pytorch, Caffe2, Chainer, Torch	TensorFlow, scikit-learn, Microsoft Cognitive Toolkit, Spark ML	TensorFlow, scikit-learn, XGBoost, Keras	TensorFlow, Spark MLlib, scikit-learn, XGBoost, PyTorch, IBM SPSS, PMML

- MLaaS is a set of tools and services including: data visualization, pre-processing, model training and evaluation, serving predictions, etc.
- Many of the world's leading cloud providers provide different types of MLaaS services, including Amazon, Microsoft, Google and IBM
- MLaaS service providers offer pre-defined models that can be used to cover standard use-cases, e.g. classification, regression, natural language processing, facial recognition, DeepLearning

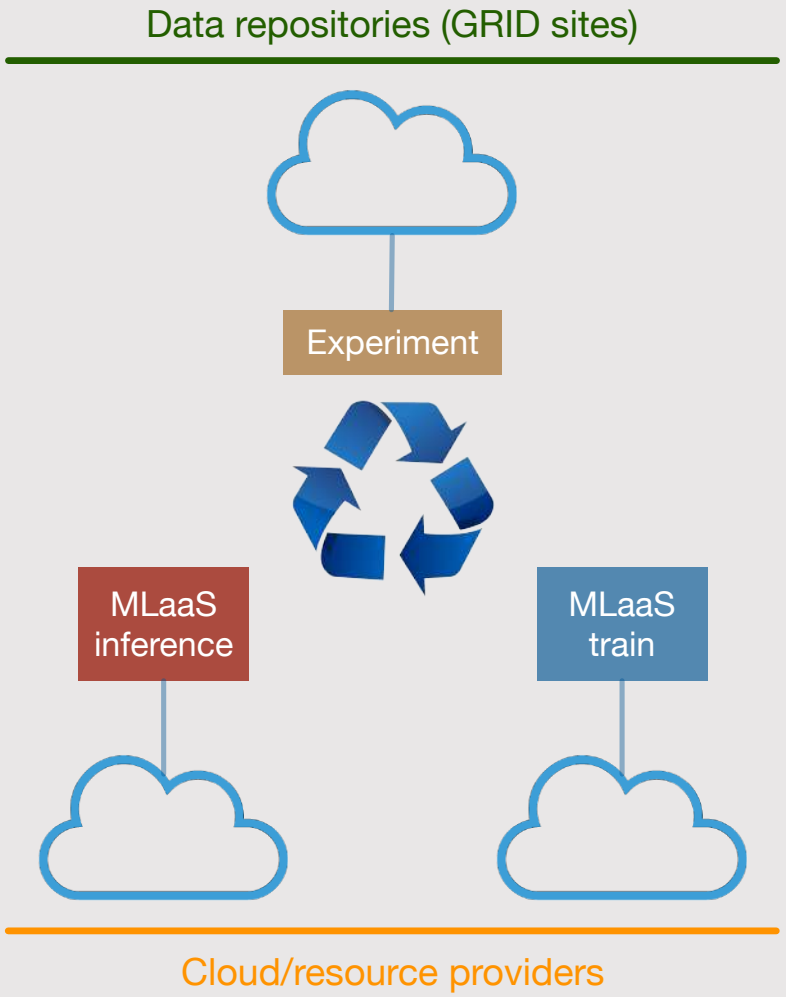
Issues with using existing solutions

- Existing MLaaS services can't read HEP data directly in the **ROOT data-format**: most of the cases ML deal with either CSV or NumPy arrays representing tabular data
  - We don't use ROOT data directly in the ML framework, we need a conversion step
  - Pre-processing operations may be more complex than offered by service providers
- R&D for specialized solutions to speed-up inference on FPGAs, e.g. HLS4ML
  - These solutions are designed for optimization of the inference phase rather than targeting the whole ML pipeline from reading data, to training and serving predictions
- Custom solutions adopted in specific CMS analysis (e.g DOI: 10.1088/2632-2153/ab9023) cannot easily generalized and do not represent ''as a Service'' solutions
- Recent solution with Spark platform for data processing and ML training (DOI: 10.1007/s41781-020-00040-0). Here data are read from the CERN EOS storage system, not allowing access to data stored in WLCG sites

MLaaS for HEP

MLaaS for HEP aims at providing the following:

- **natively read HEP data**, e.g. be able to read ROOT files of arbitrary size from local or remote distributed data-sources via XrootD
- **use heterogeneous resources** both for training and inference, like local CPU, GPUs, farms, cloud resources, etc.
- **use different ML libs and frameworks** (Keras, TF, PyTorch, etc.)
- **serve pre-trained HEP models**, like a models repository, and access it easily from any place, any code, any framework.

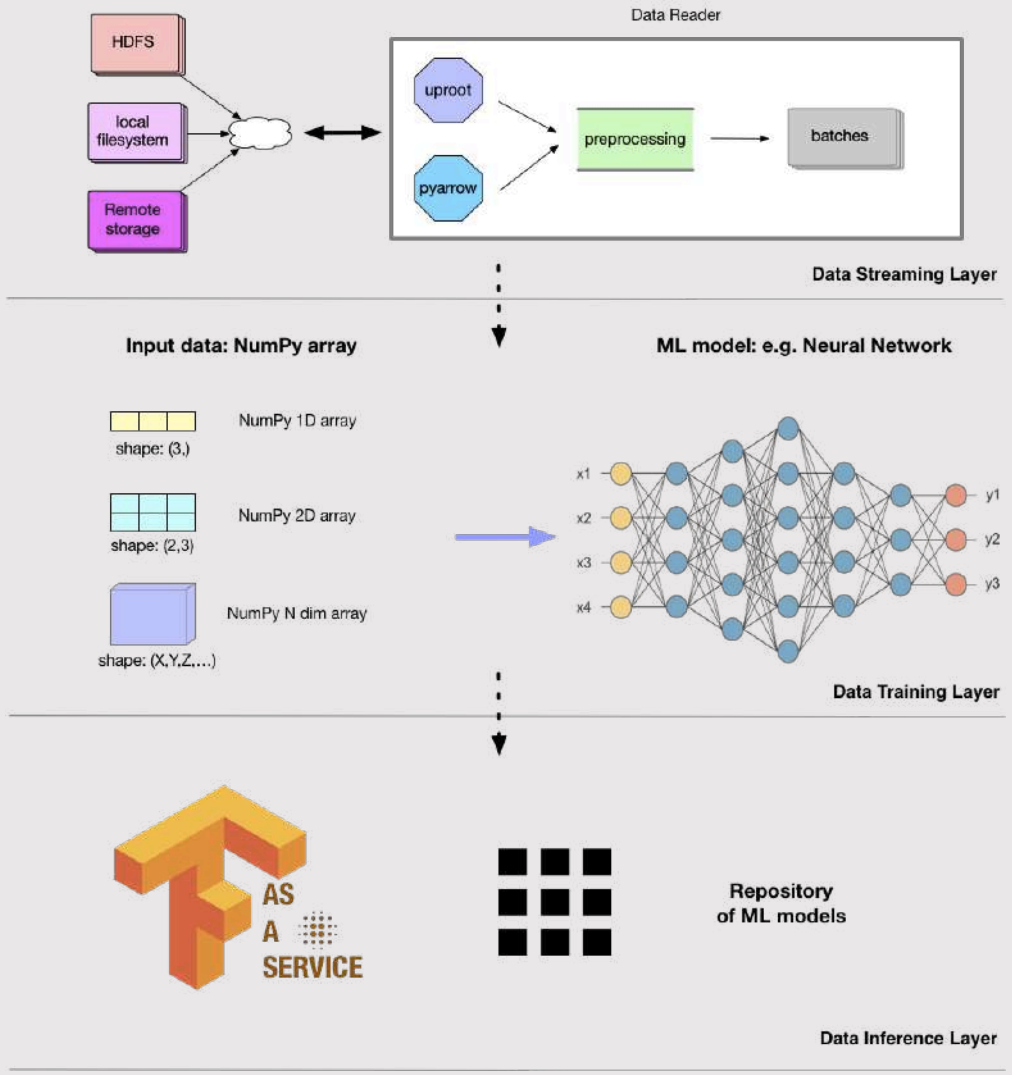


MLaaS4HEP R&D

- **Data Streaming Layer** is responsible for local and remote data access of HEP ROOT files
- **Data Training Layer** is responsible for feeding HEP ROOT data into existing ML frameworks
- **Data Inference Layer** provides access to pre-trained HEP model for HEP users

All three layers are independent from each other and allow independent resource allocation

Data streaming and training tools: github.com/vkuznet/MLaaS4HEP
Data inference tool: github.com/vkuznet/TFaaS
Paper on arXiv (arXiv:2007.14781v2 **[hep-ex]**) and submitted to the CSBS journal



Data Streaming Layer

- The development of the DIANA-HEP **uproot** library provides the ability to read ROOT data in Python, access them as NumPy arrays, and implements XrootD access to read remote files

- MLaaS4HEP extends uproot library and provide APIs to feed data read from local and remote distributed ROOT files into existing ML frameworks

  - a Python Generator is created to read ROOT files and deliver them as chunks

    - such implementation provides efficient access to large datasets since it does not require loading the entire dataset into the RAM of the training node

  - read from multiple files is supported, taking the right proportion of data from each file

- The non-flat ROOT branches are read and represented by uproot as Jagged Arrays

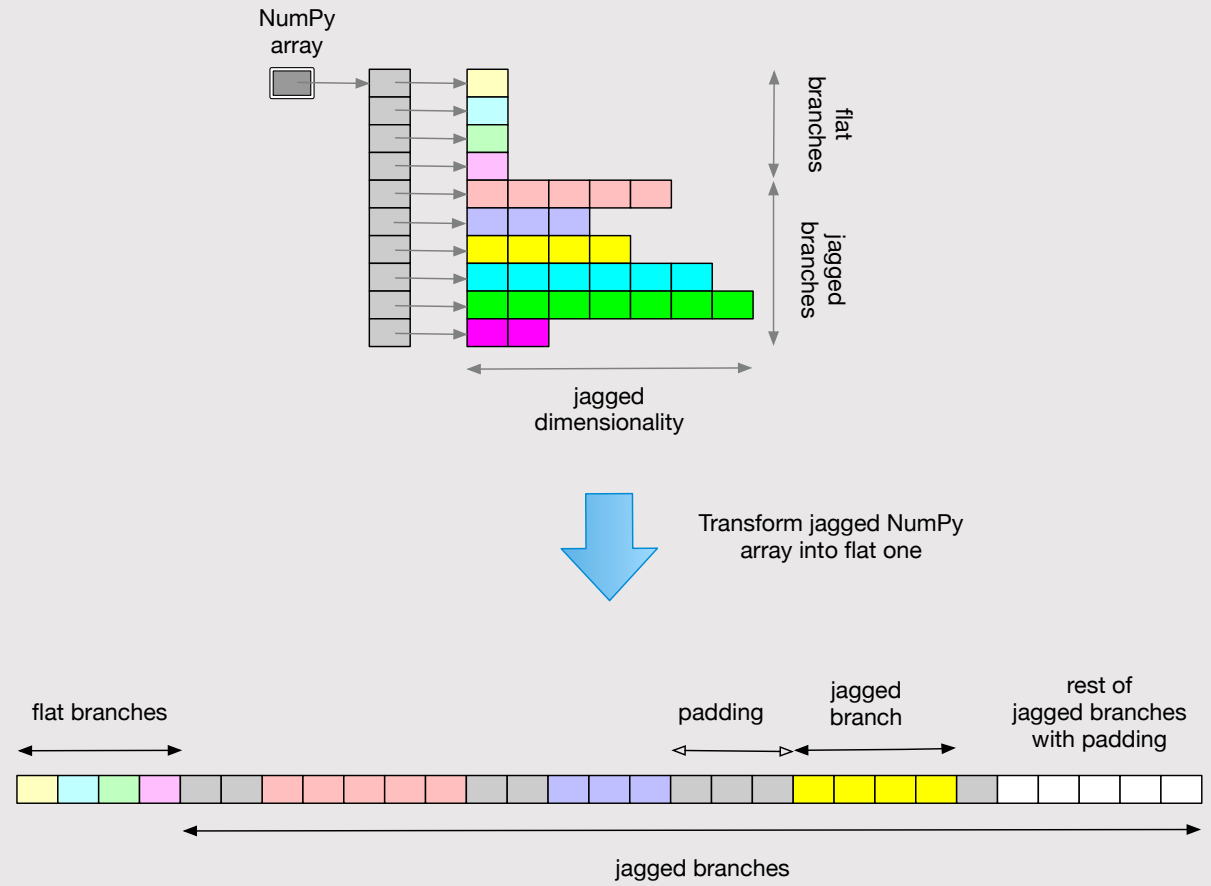
Data Training Layer

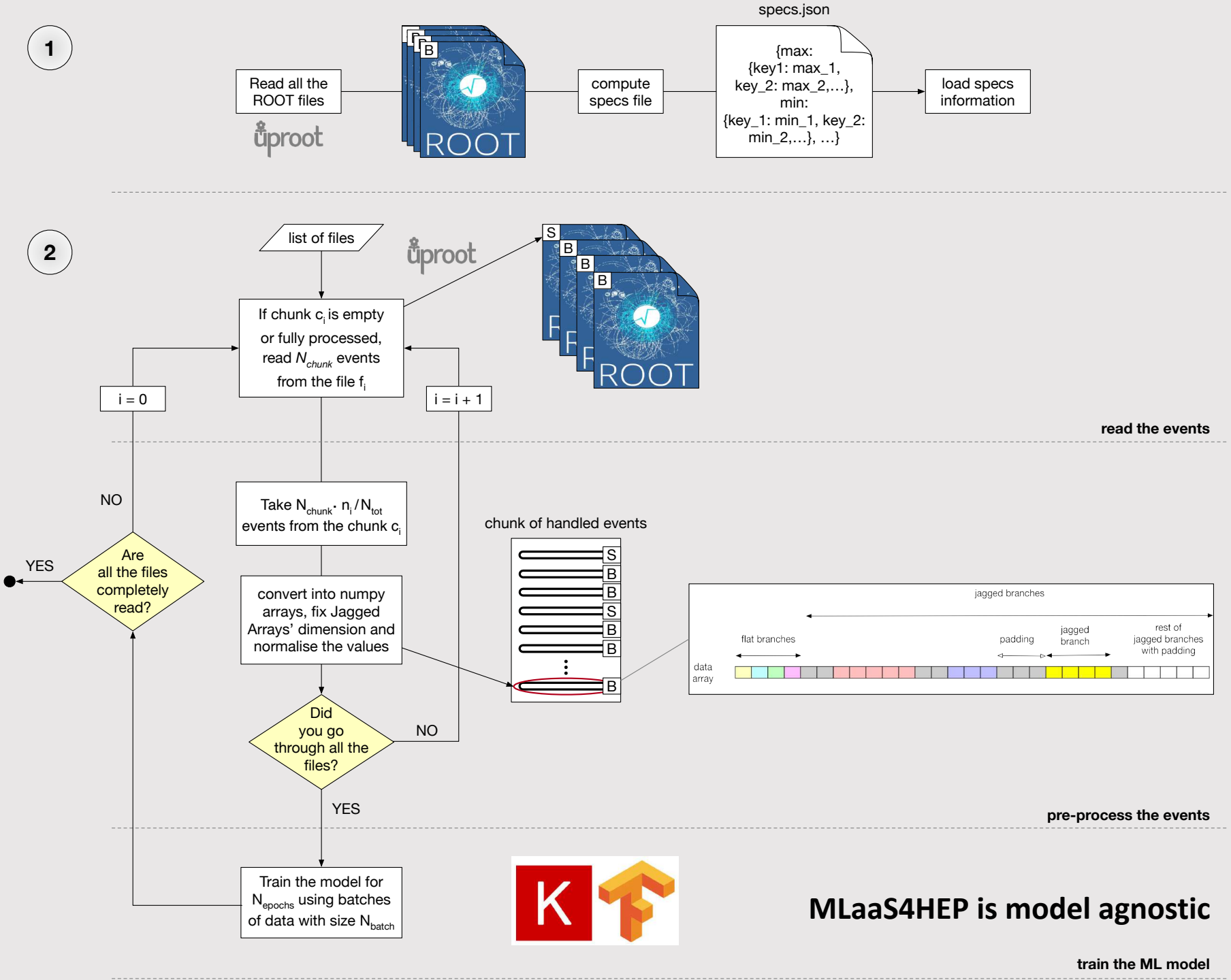
- Each event is a composition of flat and Jagged Arrays
  - Such data representation is not directly suitable for ML (dynamic dimension of Jagged Arrays across events)

- To feed these data into ML we need to resolve how to treat Jagged Arrays. We opted to flatten Jagged Arrays into fixed-size array with padding values through a two-step procedure:
  - know the dimensionality of every Jagged Array attribute;
  - update the dimension of jagged branches using padding values, which should be assigned as NANs since all other numerical values can represent attribute spectrum.

- Keep the mask array with padding values location

- We provide a proper normalization of each attribute




1
Read all the ROOT files
uproot
B
ROOT
compute specs file
specs.json
{max:
{key1: max_1,
key_2: max_2,…},
min:
{key_1: min_1, key_2:
min_2,…}, …}
load specs information
2
list of files
uproot
If chunk c_i is empty or fully processed, read N_{chunk} events from the file f_i
S
B
B
B
ROOT
i = 0
i = i + 1
read the events
NO
Take $N_{chunk} \cdot n_i / N_{tot}$ events from the chunk c_i
chunk of handled events
YES
Are all the files completely read?
convert into numpy arrays, fix Jagged Arrays' dimension and normalise the values
S
B
B
S
B
B
jagged branches
flat branches
padding
jagged branch
rest of jagged branches with padding
data array
Did you go through all the files?
NO
YES
pre-process the events
Train the model for N_{epochs} using batches of data with size N_{batch}
K
MLaaS4HEP is model agnostic
train the ML model

```
./workflow.py --files=files.txt --labels=labels.txt --model=model.py --params=params.json
```

MLaaS workflow | Input ROOT files | Labels of ROOT files | User model | MLaaS parameters

Keras model (model.py)

```
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout

def model(idim):
    "Simple Keras model for testing purposes"
    ml_model = Sequential([Dense(128,
activation='relu',input_shape=(idim,)),
                           Dropout(0.5),
                           Dense(64, activation='relu'),
                           Dropout(0.5),
                           Dense(1, activation='sigmoid')])
    ml_model.compile(optimizer=keras.optimizers.Adam(lr=1e-3),
                     loss=keras.losses.BinaryCrossentropy(),
                     keras.metrics.AUC(name='auc')])
```

MLaaS parameters (params.json)

```
{
    "nevts": 30000,
    "shuffle": true,
    "chunk_size": 10000,
    "epochs": 5,
    "batch_size": 100,
    "identifier":["runNo", "evtNo", "lumi"],
    "branch": "boostedAk8/events",
    "selected_branches":"",
    "exclude_branches": "",
    "hist": "pdfs",
    "redirector": "root://xrootd.ba.infn.it",
    "verbose": 1
  }
```

Input ROOT files (files.txt)

```
PATH/flatTree_ttHJetTobb_M125_13TeV_amcatnloFXFX_madspin_pythia8.root
PATH/flatTree_TT_TuneCUETP8M2T4_13TeV-powheg-pythia8.root
```

Labels of ROOT files (labels.txt)

```
1
0
```

MLaaS parameters

Read remote root files

Write and load the specs

```
./workflow.p  --files=files.txt --labels=  bels.  xt –model=model.py --params=params.json
DataGenerato . <MLaaS4HEP.generator.Roo ataGen  ator object at 0x7f0cb58d7fd0> [29/Jun/2020:17:53:  4] 1593445994.0
model parameters: {"nevts": 30000, "s  ffle": t ue, "chunk_size": 10000, "epochs": 2, "batch_size"  500, "identifier": ["runNo", "evtNo", "lumi"],
"branch": "boostedAk8/events", "sel  ted_branc  s": "", "exclude_branches": "", "hist": "pdfs", "r director": "root://xrootd.ba.infn.it", "verbose": 1}


Reading root://xrootd.ba.infn.it//PATH_FILES/ latTree_ttHJetTobb_M125_13TeV_amcatnloFXFX_madspin  ythia8.root
# 10000 entries, 77 branches, 9.522203445434  7 MB, 1.0169336795806885 sec, 9.36364252323795 MB/s  c, 9.833482950553169 kHz
# 10000 entries, 77 branches, 9.53391551971  55 MB, 1.2977769374847412 sec, 7.346343770133804 M  sec, 7.705484441248654 kHz
# 10000 entries, 77 branches, 9.53866767883  08 MB, 1.4104814529418945 sec, 6.7627033726234735  B/sec, 7.089777734505208 kHz
--- first pass: 948348 events, (22-flat, 5  jagged) branches, 328 attrs
<MLaaS4HEP.reader.RootDataReader object at  0x7f840dbf4d50> init is complete in 4.8529925346374  1 sec


Reading root://xrootd.ba.infn.it//PATH_FILES/flatTree_TT_TuneCUETP8M2T4_13TeV-powheg-pythia8  oot
# 10000 entries, 77 branches, 8.875920295715332 MB, 0.9596493244171143 sec, 9.24912889518941  MB/sec, 10.42047313071777 kHz
# 10000 entries, 77 branches, 8.868906021118164 MB, 1.2938923835754395 sec, 6.8544386949790  MB/sec, 7.728618026459661 kHz
# 10000 entries, 77 branches, 8.869449615478516 MB, 1.1267895698547363 sec, 7.8714338974773  9 MB/sec, 8.874771534572496 kHz
--- first pass: 1003980 events, (22-flat, 52-jagged) branches, 312 attrs
<MLaaS4HEP.reader.RootDataReader object at 0x7f8410e15f90> init is complete in 4.535124778  47559 sec


write global-specs.json
load specs from global-specs.json for root://xrootd.ba.infn.it//$PATH_FILES/flatTree_ttHJetTobb_M125_13TeV_amcatnloFXFX_madspin_pythia8.root
load specs from global-specs.json for root://xrootd.ba.infn.it//$PATH_FILES/flatTree_TT_TuneCUETP8M2T4_13TeV-powheg-pythia8.root
init RootDataGenerator in 11.186564683914185 sec
```

```
label 1, file <flatTree_ttHJetTobb_M125_13TeV_amcatnloFXFX_madspin_pythia8.root>, going to read 4858 events
read chunk [0:4857] from /$PATH_FILES/flatTree_ttHJetTobb_M125_13TeV_amcatnloFXFX_madspin_pythia8.root
# 10000 entries, 77 branches, 9.52220344543457 MB, 1.3816642761230469 sec, 6.891835889507034 MB/sec, 7.237648228164387 kHz
total read 4858 evts from /$PATH_FILES/flatTree_ttHJetTobb_M125_13TeV_amcatnloFXFX_madspin_pythia8.root


label 0, file <flatTree_TT_TuneCUETP8M2T4_13TeV-powheg-pythia8.root>, going to read 5142 events
read chunk [4858:9999] from /$PATH_FILES/flatTree_TT_TuneCUETP8M2T4_13TeV-powheg-pythia8.root
# 10000 entries, 77 branches, 8.875920295715332 MB, 1.7170112133026123 sec, 5.169401473297779 MB/sec, 5.8240737873606205 kHz
total read 5142 evts from /$PATH_FILES/flatTree_TT_TuneCUETP8M2T4_13TeV-powheg-pythia8.root
```

Create the chunk

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               49152
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 57,473
Trainable params: 57,473
Non-trainable params: 0
_________________________________________________________________
```

Init the ML model

Perform training cycle

```
Train on 7000 samples, validate on 3000 samples
Epoch 1/2
7000/7000 [==============================] - 2s 220us/sample - loss: 1.5275 - auc: 0.7845 - accuracy: 0.7307 - val_loss: 2.5731e-04 -
val_auc: 1.0000 - val_accuracy: 1.0000
Epoch 2/2
7000/7000 [==============================] - 0s 20us/sample - loss: 0.1406 - auc: 0.9883 - accuracy: 0.9543 - val_loss: 8.8477e-06 - val_auc:
1.0000 - val_accuracy: 1.0000
```

Data Inference Layer

- The Data Inference Layer is implemented as TensorFlow as a Service (TFaaS), written in the **Go** programming language
  - The Go programming language natively supports concurrency and it is very well integrated with the TF library
- TFaaS is capable of serving any TensorFlow model and it can be used as a global repository of pre-trained HEP models (it is experiment agnostic and can work with any HTTP based client)
- Both Python and C++ clients were developed on top of the REST APIs (end-points) and other clients can be developed thanks to HTTP protocol used by the TFaaS Go RESTful implementation
  - C++ client library talks to TFaaS using ProtoBuffer data-format, all others use JSON (see examples)
- TFaaS allows a rapid development or continuous training of TF models and their validation: clients can test multiple TF models at the same time
  - A demo server hosted by CERN is online: https://cms-tfaas.cern.ch/
- From R&D to a production service: expanding the team on code maintainance and ops support

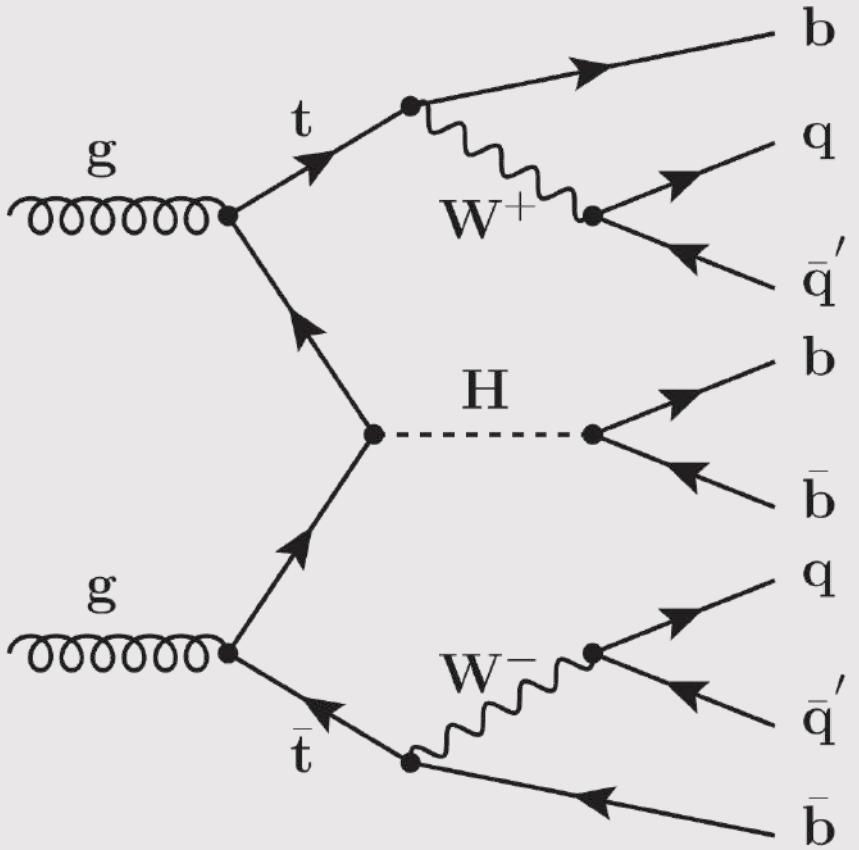
Real case scenario: $t\bar{t}H(bb)$ analysis in the boosted, all-hadronic final states

- We performed a proof-of-concept of the entire pipeline using CMS NANOAOD.

- We recently validated the MLaaS framework, namely tested the infrastructure on real physics use-case. We chose a signal vs background discrimination problem in a $t\bar{t}H$ analysis. This allows us to:

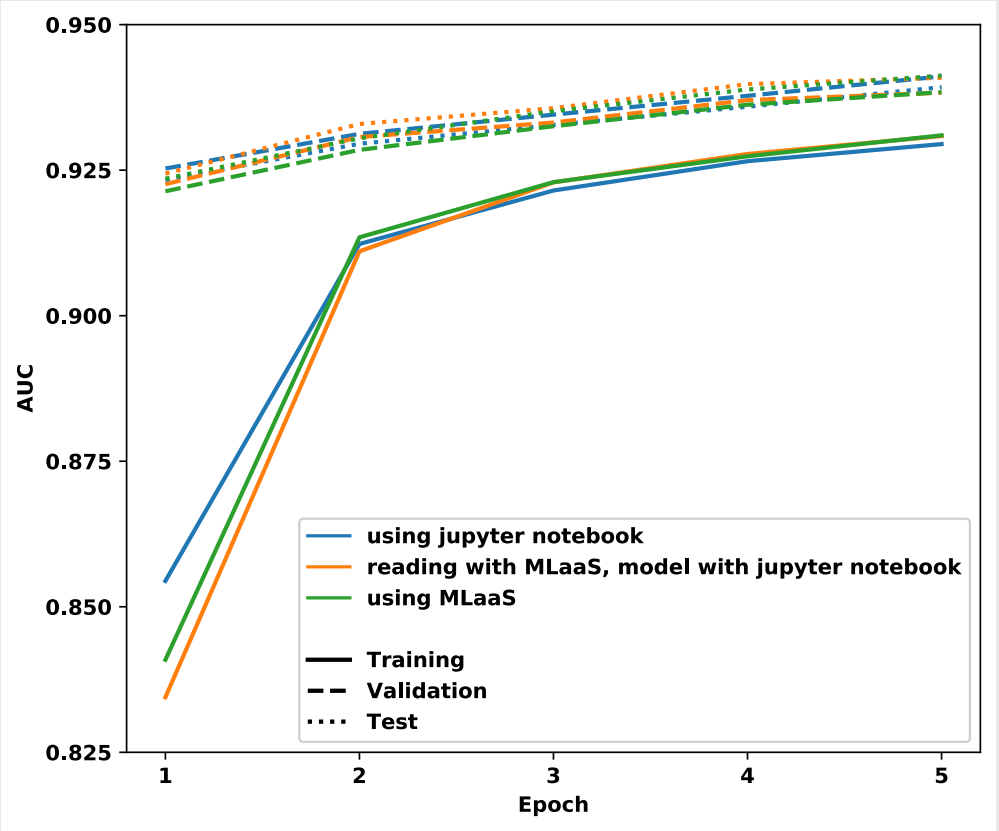
    1. validate MLaaS results from the physics point of view
    2. test performances of MLaaS framework

For the phase of validation we used 9 ROOT files, 8 of background and 1 of signal. Each file has 27 branches, with 350 thousand events for the whole pool of files and a total size of almost 28 MB. The ratio between signal and background is 10.8%.



MLaaS4HEP validation

- Validate the MLaaS4HEP approach by comparing it with alternative methods on the reference use-case
  - We used a simple NN with Keras in all methods
- Validation successful: physics results are not impacted
- The AUC score is also comparable with the BDT-based analysis, performed within the TMVA framework by a subgroup of the CMS HIG PAG



AUC score

MLaaS4HEP performance

- For this phase we used all available ROOT files without any physics cut. This gave us a dataset with 28.5M events with 74 branches (22 flat and 52 Jagged), and a total size of about 10.1 GB.

- We performed all the tests running MLaaS framework on
  - macOS, 2.2 GHz Intel Core i7 dual-core, 8 GB of RAM
  - CentOS 7 Linux, 4 VCPU Intel Core Processor Haswell 2.4 GHz, 7.3 GB of RAM CERN Virtual Machine

- The average available bandwidth was approximately 129 Mbit/s and 639 Mbit/s using macOS and CERN VM, respectively.

- The ROOT files are read from local file-systems (SSD storages) and remotely from the Grid sites. In particular, we read files remotely from three different data-centers located at
  - Bologna (BO)
  - Pisa (PI)
  - Bari (BA)

MLaaS4HEP performance results

- Based on the resource we used and if the ROOT files were local or remote, we obtained:
  - **specs computing phase (chunk size = 100k events)**
    - Event throughput: 8.4k – 13.7k evts/s
    - Total time using all the 28.5M events: 35 – 57 min
  - **chunks creation in the training phase (chunk size = 100k events)**
    - Event throughput: 1.1k – 1.2k evts/s
    - Total time using all the 28.5M events: 6.5 – 7.5 hrs

PRELIMINARY

- The time to train the ML model is not included in the performance. It is independent from the MLaaS4HEP framework but depends on the underlying ML framework, the complexity of the used ML model, and the available hardware resources.

- We estimate that projecting these results for datasets at the **TB scale** and using the same hardware resources, the specs computing phase will take O(100) hours and the training phase will take O(1k) hours (plus the time required to train the ML model).
  - Further optimization of the MLaaS4HEP pipeline will be required to process TB or PB scale datasets and it may involve parallelization of I/O, distributed ML training, etc.

TFaaS performance

- We did TFaaS benchmarks on CentOS 7 Linux, 16 cores, 30 GB of RAM in two modes:
  - using 1k calls with 100 concurrent clients,
  - using 5k calls with 200 concurrent clients.

- We tested both JSON and ProtoBuffer 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 a mid-size pre-trained NN model with 27 features and 1024x1024 hidden layers used in the physics analysis discussed. Similar performance was found for image classification datasets (MNIST).

- The actual performance of TFaaS will depend on the complexity of served ML model and available hardware resources.

- Even though a single TFaaS server may not be as efficient as an integrated solution, it can be horizontally scaled, e.g. using Kubernetes or other cluster orchestrated solutions, and may provide the desired throughput for concurrent clients.

Towards MLaaS4HEP cloudification

- We created a service performing the ML pipeline using local and remote ROOT files
  - The performance strictly depends on the available hardware resources

- How to improve the performance?
  - Adopt new solutions in the code
  - Invest in better and more expensive on-premise resources
  - Move to the cloud

- The operation of cloudification has two benefits:
  - opens us to potentially more performing resources
  - provides a real ‘’as a Service’’ solution for the user

- We started to work for a MLaaS4HEP cloudification using DODAS

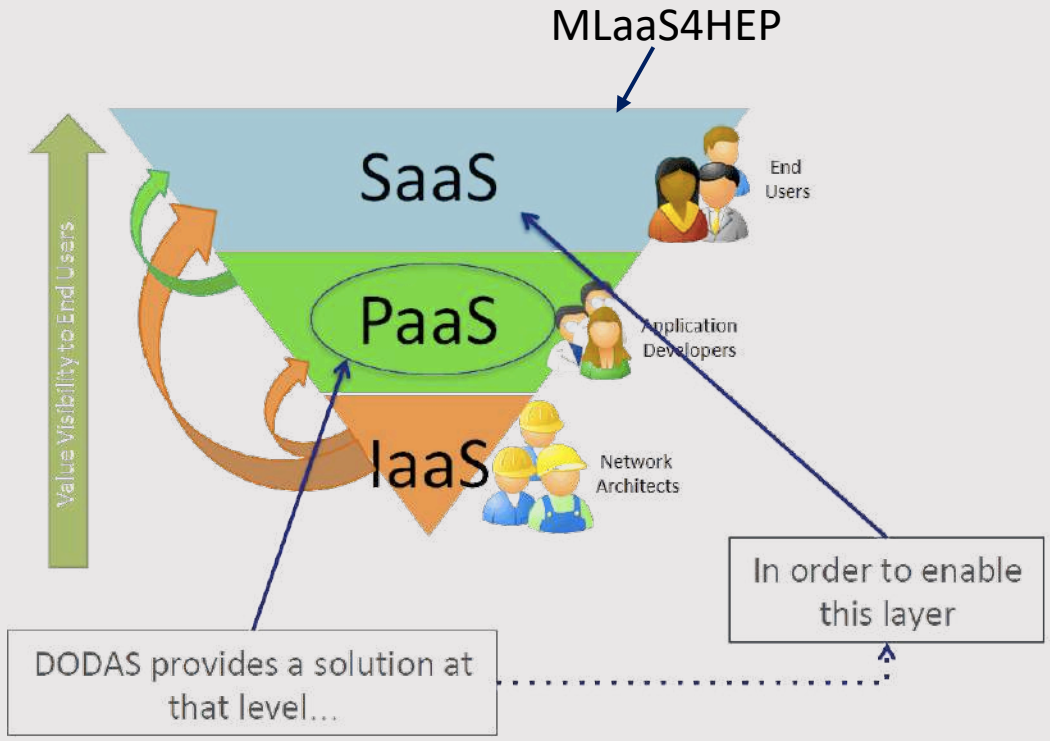
Dodas

Dynamic On Demand Analysis Service (DODAS) is a Platform as a Service tool for generating over cloud resources and on-demand, container based solution.

- DODAS completely automates the process of provisioning, creating, managing and accessing a pool of distributed and heterogeneous computing and storage resources.

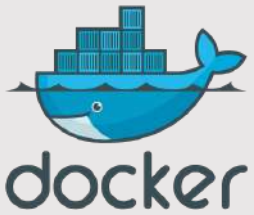
- DODAS has a high level of modularity, a key to a generic applicability.
  - Being modular, the architecture provides the ability to easily customize the workflow depending on the community computational requirements.
  - Implements services composition model based on templates

- Both HTCondor batch system and platform for the Big Data analysis based on Spark, Hadoop etc, can be deployed using ‘’any cloud provider’’ with almost zero effort.



PoS(ISGC 2018 & FCDD)024

MLaaS4HEP cloudification with DODAS



Creation of a docker image able to run the workflow.py script

Create an Ansible playbook to automatize the configuration and deployment of the container with dependencies

Convert the Ansible playbook into an Ansible role

Creation of a Tosca template to define the resource requirements and the input parameters for the creation of the docker container

Create the deployment from command line

Run workflow.py interactively or with jupyterhub

```
dodas create lgiommi-template.yml
   dodas login <infID> <vmID>
```

Functional test

- We tested MLaaS4HEP using an infrastructure created with DODAS
  - Ubuntu 18 Linux, 8 AMD Opteron 62xx class CPU 2.6 GHz, 16 GB RAM VM
  - We used both local and remote (stored in the Pisa data-center) ROOT files

		MacOS and CERN VM	DODAS infrastructure
specs computing phase	Event throughput	8.4k – 13.7k evts/s	5.8k – 7.3k evts/s
	Total time using all the 28.5M events	35 – 57 min	66 – 82 min
chunks creation in the training phase	Event throughput	1.1k – 1.2k evts/s	0.5k evts/s
	Total time using all the 28.5M events	6.5 – 7.5 hrs	16 hrs

RESULTS OF THE FUNCTIONAL TEST

MLaaS4HEP using Jupyterhub

- We provide a SaaS solution for a sharable jupyter notebook
- Token-based access to the jupyterhub, with the support for a customizable environment

DEMO

- Integrate cloud storage for managing the required files (ROOT files, ML model, etc.)

```
# . ./shared/setup_local
(base) # cd /workarea/shared/folder_test
(base) # ../../workarea/MLaaS4HEP/src/python/MLaaS4HEP/workflow.py --files=files_test.txt --labels=labels_test.txt --
model=keras_model.py --params=params_test.json
model parameters: {"nevts": -1, "shuffle": true, "chunk_size": 10000, "epochs": 5, "batch_size": 100, "identifier": ["runNo", "evtNo",
"lumi"], "branch": "events", "selected_branches": "", "exclude_branches": "", "hist": "pdfs", "redirector": "root://gridftp-storm-
t3.cr.cnaf.infn.it:1095", "verbose": 1}
Reading ttH_signal.root
# 10000 entries, 29 branches, 1.10626220703125 MB, 0.034181833267211914 sec, 32.364039645948566 MB/sec, 292.5530623775014 kHz
# 10000 entries, 29 branches, 1.10626220703125 MB, 0.022344589233398438 sec, 49.50917626973965 MB/sec, 447.53563807084936 kHz
```

Summary

We built a MLaaS solution for HEP where:

- local and remote ROOT files can be directly read
- complexity of data transformation from ROOT I/O to ML frameworks is hidden from the user
- resources can be used dynamically and independently for training and inference layers
- Python based ML framework of user choice can be used (e.g. TensorFlow, Keras, Pytorch)
- customization via JSON configuration: total number of events to read; chunk size of data to read; select or exclude branches to read, choice of XrootD redirector
- we validated the MLaaS framework with a physics use-case and achieved similar results as BDT analysis
- we did performance tests for the streaming and training layers

And …

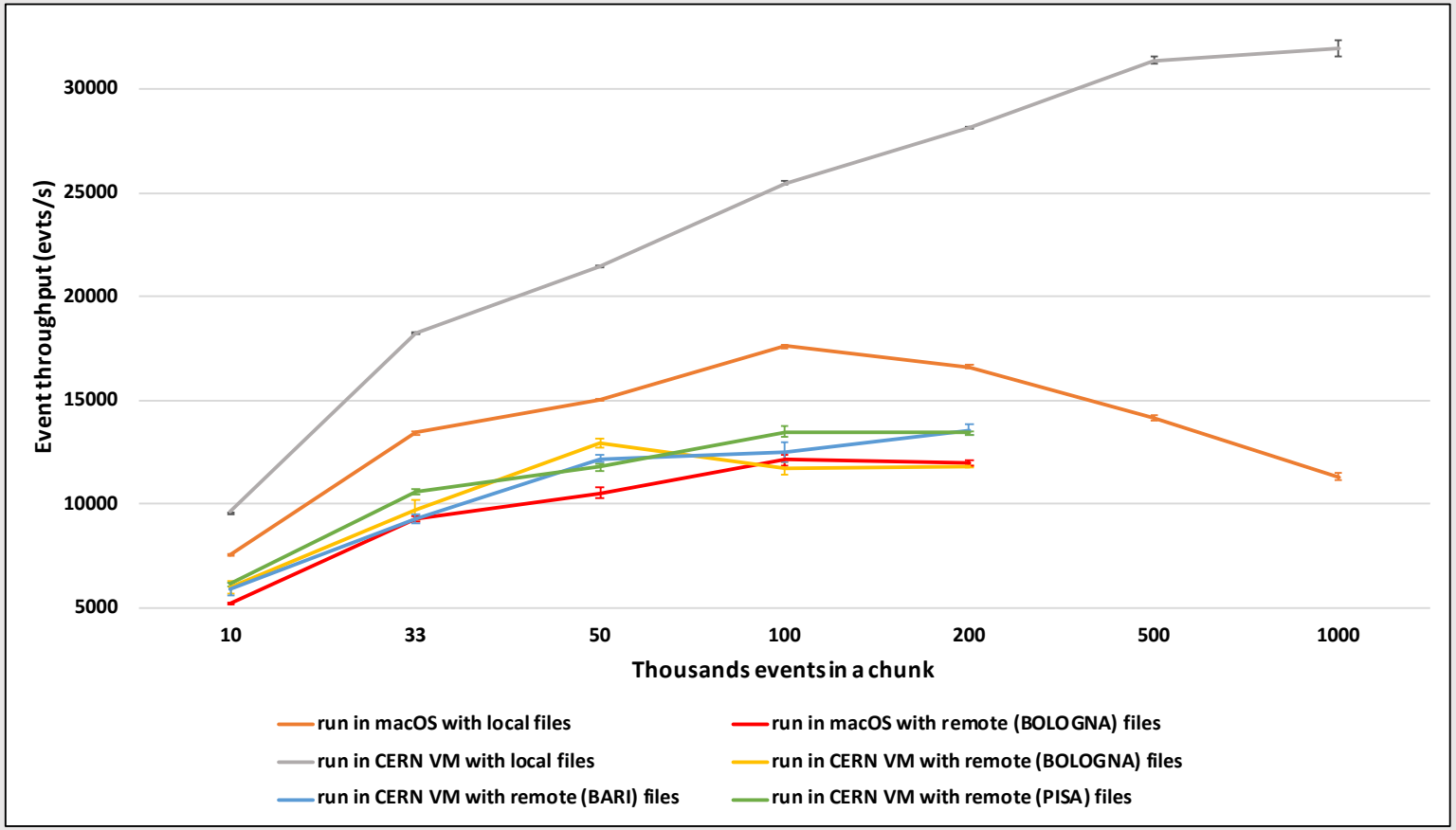
- we moved towards the MLaaS4HEP cloudification using DODAS
  - provides interactive and jupyterhub access for an user-friendly platform where run MLaaS4HEP
  - provides compliance with the INFN-Cloud portfolio of services

Planning: dynamically load user-based pre-processing functions; try FPGAs to speed up the inference phase, I/O parallelization and distributed ML training, …

Thanks for the attention

Backup slides

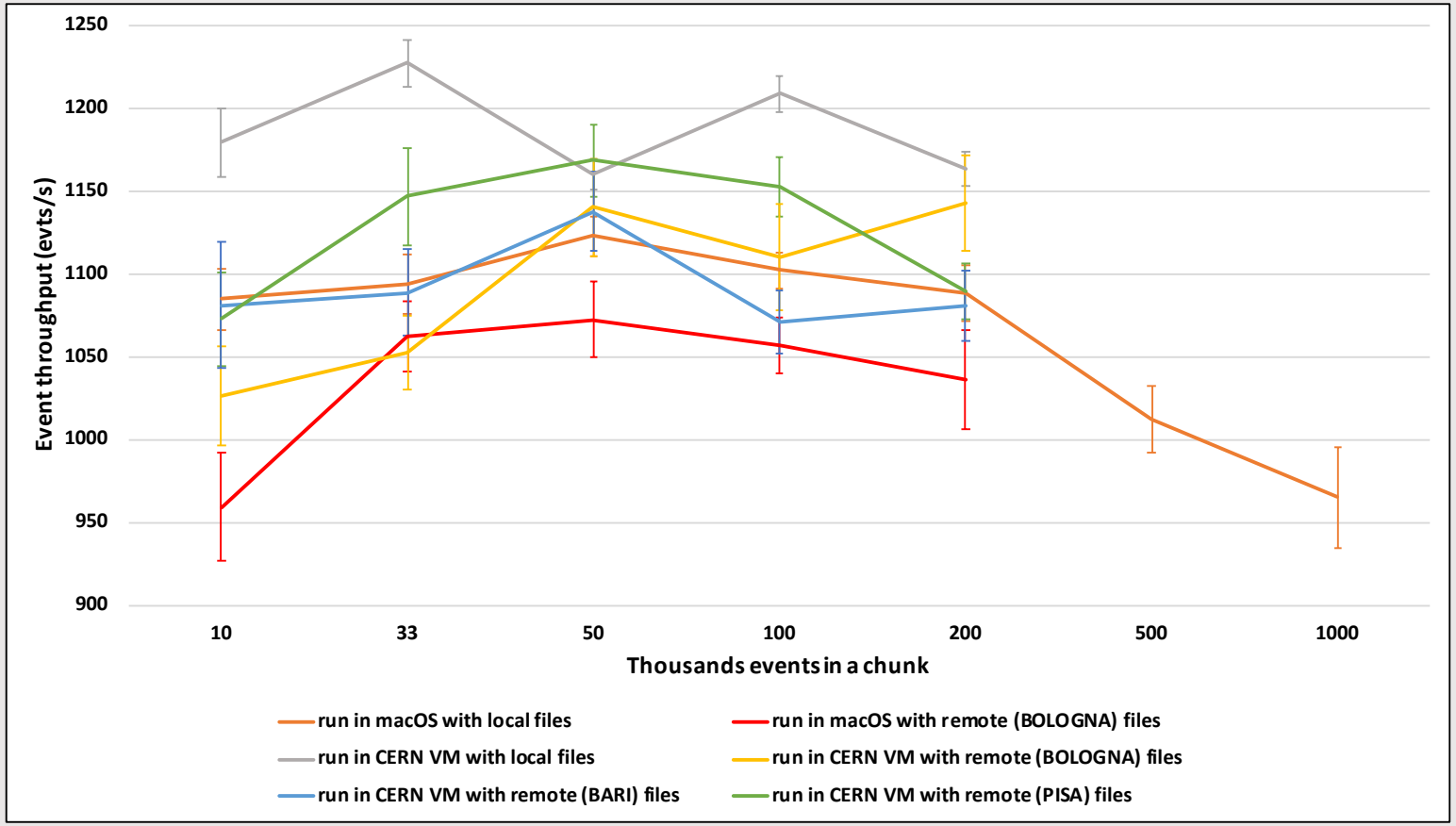
1



	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)
	~ 19-41 min	~ 16-17 min	~ 35-57 min	~ 8.4k – 13.7k evts/s

Values for chunk size fixed to 100 thousands events

2



	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)

~ 1.1k – 1.2k evts/s

Values for chunk size fixed to 100 thousands events

More on MLaaS4HEP performance

- The training step takes about 7 hours for both MacOS and CERN VM, using all the 28.5M events (plus the time required to train the ML model), where:
  - 36% is used to extract and convert each event in a list of NumPy arrays;
  - 27% is used for fixing the Jagged Arrays' dimension;
  - 26% is used for the normalization step;
  - 6% is used for creating the masking vectors.

- We estimate that, projecting these results for datasets at the **TB scale** and using the same hardware resources, the specs computing phase will take O(100) hours and the training step will take O(1k) hours (plus the time required to train the ML model)
  - Further optimization of the MLaaS4HEP pipeline will be required to process TB or PB scale datasets and it may involve parallelization of I/O, distributed ML training, etc.

- The time to train the ML model is not included in the performance shown. This time is independent from the MLaaS4HEP framework since it is determined by usage of the underlying ML framework, the complexity of used ML model and the available hardware resources.
  - In our case the training time for a chunk of 100k events is about 11 seconds and 13 for MacOS and CERN VM, respectively.

How to use TFaaS (1)

- Follow the installation instructions here
- Setup url to point to your TFaaS server (e.g url=http://localhost:8083 or url=https://cms-tfaas.cern.ch/)
- If necessary convert the model into a TensorFlow one (e.g using this solution)
- create upload json file, which should include:
  - fully qualified model file name
  - fully qualified labels file name
  - model name you want to assign to your model file
  - fully qualified model parameters json file name

Example of an upload.json file:
{ "model": "/path/model.pb",
"labels": "/path/labels.txt",
"name": "model_name",
"params":"/path/params.json"}

Example of a params.json file:
{ "name": "model_name", "model": "model.pb", "description": "my model description", "labels": "labels.txt", "inputNode": "dense_1_input", "outputNode": "output_node0"}

- the model parameters json file is used on a TFaaS server side. It context should be the following:
  - model name (how your model will be named in TFaaS server, e.g. the one in the upload.json file)
  - model file name (name of your model file will be used in TFaaS server, e.g. the one in the upload.json file)
  - model labels file name (similar to model file name but used for labels file)
  - description string (provide details about your model)
  - inputNode of your TF model (can be found by inspecting pbtxt)
  - outputNode of your TF model (can be found by inspecting pbtxt)

How to use TFaaS (2)

- We provide pure python client to perform all necessary actions against TFaaS server. Here is short description of available APIs

- upload a model to the server
  - tfaas_client.py --url=$url --upload=upload.json

- list existing models in TFaaS server
  - tfaas_client.py --url=$url --models

- delete given model in TFaaS server
  - tfaas_client.py --url=$url --delete=model_name

- prepare input json file for querying model predictions
  - e.g. {"keys":["attribute1", "attribute2"], values: [1.0, -2.0]}

- get predictions from TFaaS server
  - tfaas_client.py --url=$url --predict=input.json

- get image predictions from TFaaS server
  - tfaas_client.py --url=$url --image=/path/file.png --model=ImageModel

INFN Cloud

- INFN is offering to its users a comprehensive and integrated set of Cloud services through its dedicated INFN Cloud infrastructure

- The current operational state of INFN Cloud is pre-production but already serving several INFN experiments and collaborations, with full production state and general availability expected in 2021
  - The access to the INFN Cloud services is currently reserved to INFN personnel or personnel with whom INFN has established formal collaborations, such as research associates

- The INFN Cloud portfolio, available by an easy to use web interface but also exploitable via command line interfaces, is defined upon clear user requirements

- The INFN Cloud services are based on modular components and span the IaaS, PaaS and SaaS
  - All services are described by TOSCA templates (which can refer internally to other components such as Ansible playbooks)

- For more information visit the official web page