Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Machine Learning for Monitoring of Complex Detector Systems

Borislav Pavlov[a], Leandar Litov[a], Peicho Petkov[a], Nevena Ilieva[b,c,*]

[a]Faculty of Physics, Sofia University "St. Kl. Ohridski", Sofia, Bulgaria
[b]Institute of Information and Communication Technologies, Bulgarian Academy of Sciences, Sofia, Bulgaria
[c]Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Sofia, Bulgaria

Abstract

The experiments running at the Large Hadron Collider (LHC) at CERN work in challenging conditions and produce an enormous amount of data to be analyzed. The smooth operation of the detectors is vital for data quality. The detector performance monitoring and the certification of the experimental data as usable for physics analysis are of key importance in ensuring the reliability of the obtained results. These human-conducted operations could be largely automated with the use of Machine Learning (ML) techniques. We report the results of a proof-of-concept study addressing this challenge by developing a customized ML tool for detector monitoring and testing its efficiency on a simulated setup of 512 individual detector modules, mimicking detector systems at the ongoing experiments at the LHC accelerator at CERN.

1. Introduction

The increasing complexity of modern Nuclear and Particle Physics experiments and the huge amount of collected data leads to practical problems related to the monitoring of the experimental setups and the data quality certification. For the large-scale experiments at the Large Hadron Collider (LHC) at CERN, these problems are even more pressing. The four detector complexes – ATLAS[1], CMS[2], ALICE[3] and LHCb[4] – collect data for years in harsh conditions in order to accumulate the necessary statistics for producing reliable scientific results, such as the discovery of new particles or phenomena, like the Higgs boson discovery 10 years ago by the CMS [5] and ATLAS[6] experiments. Monitoring of the detector systems and quality certification of the experimental data is of primary importance for the reliability of the physics results, both these tasks being human-conducted by now. The objective of the implemented pilot project was the development of a machine-learning (ML) toolkit based on a dedicated fine-grained massively parallel neural network (NN) for big data analysis to facilitate monitoring and improve the accuracy of data certification, with the resistive-plate chambers (RPC) subsystem of the CMS detector as a model target. This is one of the biggest subsystems of the CMS experiment covering approximately 4000 square meters of particle detectors with some 115 000 channels readout every 25 ns. Per run (a data taking session), the RPC system generates several files of output data with a typical size of several GB each, and the whole amount of data to be analysed is accumulated during over 350 000 runs. However, it is not the data certification but the NN training stage that requires substantial computational resources due to the huge amount of input data in terms of both the number of files and volume, and the vast number of inputs of the first layer of the network.

A successful machine learning algorithm for detector monitoring can be developed following different strategies. One approach is to build an ML model stepping on the detailed knowledge about the physics processes taking place in a particular detector type [7]. This type of model is applicable to individual detectors. In this paper we take a more general approach by using cross-correlations between different detector modules for the ML tool development, thus making the tool applicable for detector systems consisting of a large number of detector modules. As a *proof-of-concept*, we develop an autoencoder-based ML algorithm for monitoring the detector parameters. Two parameters were selected as important indicators of the detector stability – the current and the counting rate. During the training phase, the autoencoder is supposed to "learn" the collective behavior of all detector modules.

*Corresponding author, e-mail: nevena.ilieva@iict.bas.bg

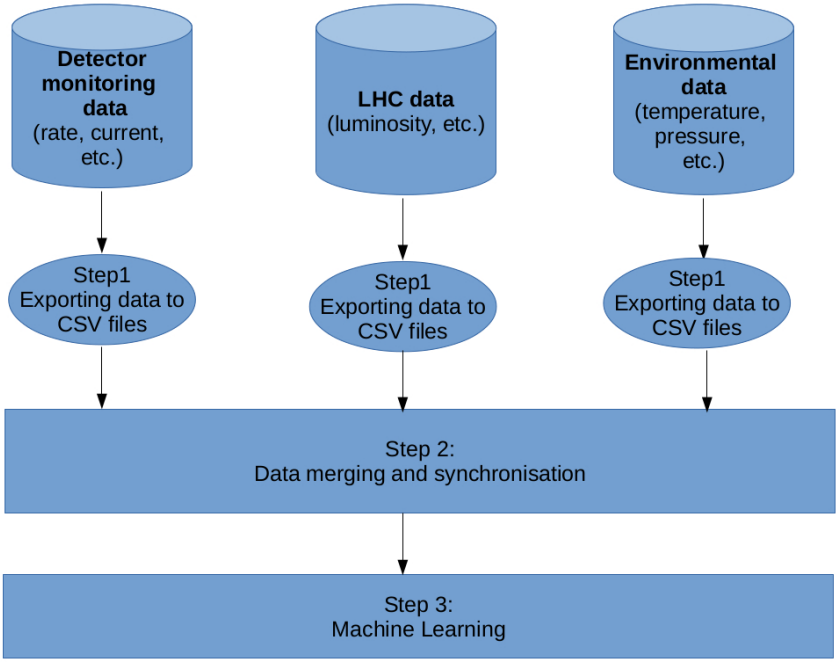

Figure 1. The inclusive three-step protocol: synthetic data generation by Monte Carlo simulations (Step 1); Data merging and synchronisation (Step 2); essential Machine Learning (Step 3). Steps 2 and 3 are performed using HPC infrastructure.

Such an autoencoder can then be used to spot signs of abnormal behaviour of a single or multiple detector modules. The autoencoder was trained and tested on synthetic data. The data sets were obtained by a custom-designed Monte Carlo tool developed for the simulation of realistic accelerator's instantaneous luminosity, detectors' counting rates, and currents. Different simulations were stored in different files in order to mimic the real detector monitoring data. The data synchronisation (rate and luminosity) and ML step were performed at the Marconi100 supercomputer at CINECA [8], within a dedicated Preparatory Access project.

2. Workflow

The LHC experiments store and analyze the data suitable for physics analysis on the GRID infrastructure. In addition to the data samples, the experiments record auxiliary data as well, used to monitor the stability of each detector module, like the detector's supply voltages, currents, counting rates, electronics thresholds, etc. Usually, this data is stored temporarily in different databases and after some time is even discarded. This data is of particular interest for training the monitoring ML algorithm. The Monte-Carlo generated synthetic data fully resembles the actual data sets, with all their specific properties, recording and storing pathways.

As a first step, the relevant data has to be extracted from all those different databases in an ML-suitable format. Note that the data is stored asynchronously. Thus, the detector's rate is stored every few seconds, and instantaneous luminosity is stored every ≈ 23 seconds (the so-called Lumi section), but the data for the applied voltage is stored only if a change in the latter has occurred. Therefore, an additional step of data synchronisation and/or data averaging is needed, a dedicated data-synchronisation tool being used, prior to proceeding to the essential ML process as a third step (Fig. 1).

The ML step is computationally more challenging, so it was designed to use Horovod [9] and TensorFlow [10] to allow the simultaneous use of GPUs on multiple nodes. A built-in autoencoder predicts the detector current. The difference between the predicted and the "measured" current is used to spot possible inconsistencies in the detector operation. Both tools were originally developed on Avitohol supercomputer at the Bulgarian Academy of Sciences [11] and then validated on CINECA's supercomputer Marconi100 [8] showing good scalability.

3. Synthetic-data generation: Monte Carlo simulations

Because of the restricted real-data access, the ML software for detector monitoring was developed, tested, and validated on MC-simulated data. In particular, by introducing certain malfunction manifestations in the synthetic data samples, the reliability of the ML-based monitoring can be probed. Another advantage of this approach is that the MC simulations can easily be expanded to emulate also other types of detectors or to include additional information not available otherwise.

In order to emulate the response of each detector module a dedicated tool was developed and implemented in Python. The simulation steps on a very general physics model, which is adequate for simulations of various detector types and can be described by the following formula:

$$R(t) = E\big(X(t),x(t)\big)L(t) + N\big(X(t),x(t)\big), \tag{1}$$

where $R(t)$ is the simulated detector rate, $L(t)$ is the accelerator's instantaneous luminosity, $E\big(X(t),x(t)\big)$ is the detector's effective cross-section. It accounts for the detector's efficiency, geometry coverage, etc. In Eq. (1), $N\big(X(t),x(t)\big)$ is the detector dark-counting rate, i.e. the counting rate when the accelerator is off, so at zero luminosity. $X(t)$ denotes a set of global parameters which may affect the response of all detectors. Such parameters are, e.g. the environment temperature, atmospheric pressure, humidity, etc. $x(t)$ denotes a set of local parameters accounting for the performance of each detector, e.g. supply voltage, detector thresholds, etc. The current $I(t)$ is supposed to be proportional to the detector rate

$$I(t) = q_{av}R(t), \tag{2}$$

where the proportionality factor q_{av} is the average charge developed in the detector per single hit.

The model described above is linear with respect to the instantaneous luminosity under normal conditions. However, if the detector is subjected to an enormous radiation flux, it could saturate. Although such an anomaly should never happen in the real runs, we can account for it in the simulated data by introducing either a cut-off to $R(t)$ or a luminosity dependence to the detector effective cross-section, i.e. $E = E\big(X(t),x(t),L(t)\big)$. To emulate real data taking, the samples are grouped in "runs". During each run, the luminosity is simulated to decrease exponentially with time. Sudden luminosity drops and recoveries are also simulated in order to mimic the so called van der Meer scans, which normally occur during the LHC operation. The chambers are represented by two parameters – the effective cross-section and the dark counting rate. Both parameters can be varied in time to account for the change of the working conditions, environmental parameters, spurious events, possible problems or malfunctions, etc.

In the present study, the sample data – accelerator instantaneous luminosity, detectors counting rates, and currents – was generated along a two-step Monte Carlo protocol:

1. The accelerator instantaneous luminosity and global parameters are simulated using their proper time constants within several hundred runs to emulate a one-year data acquisition. In each run the instantaneous luminosity is simulated by means of as exponentially decreasing function and a discrete value is stored at intervals t=23 s (the time corresponding to one accelerator Lumi section);
2. The individual response of each chamber (512 in total) is simulated independently. This step is run on a cluster and the scaling is linear. The data for each detector is stored in a separate file in order to emulate the real data.

4. Data synchronization

In real experiments, the environmental parameters, the accelerator luminosity, and the detector parameters are stored in different databases. After their extraction, all the parameters have to be synchronised using the time stamp of each measurement and merged together in an ML-convenient format (for example, CSV). In order to emulate similar behavior, the synthetic data – environmental parameters, accelerator luminosity, and detector parameters – were stored separately for different time periods and then a synchronisation-and-merging tool was applied to aggregate them. In Fig. 2 the performance of the data synchronisation tool is shown, demonstrating excellent scalability. The super-linear speedup observed is due to the increased availability of faster memory. With the increase of the computer nodes not only does the computational power change but also the size of total caches from different nodes. With the larger accumulated cache size, more or even all of the data could be loaded into caches and the memory access time reduces dramatically, which causes the extra speedup in addition to the one from the actual computation.

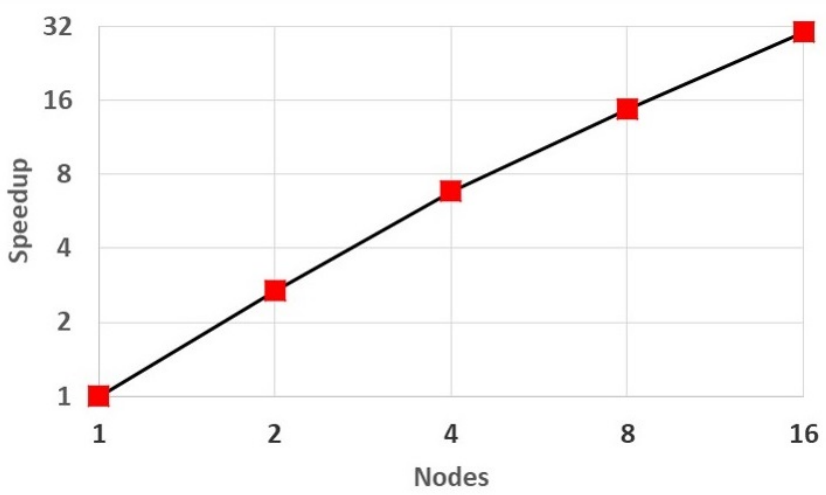

Figure 2. Performance result (speedup) obtained for the data synchronization tool.

5. Autoencoder ML tool

The monitoring ML tool we developed uses Horovod and TensorFlow to allow the use of GPUs on multiple nodes simultaneously. The instrument is based on an autoencoder with 512 inputs and the same number of outputs. The autoencoder uses five layers with the following architecture: 512-128-64-128-512 and predicts the detector parameters. Monitoring consists in continuously comparing the prediction with the measurement (in this case, the synthetic data), a deviation indicating possible irregularities in the operation of the detector. The tool is available on GitHub [12].

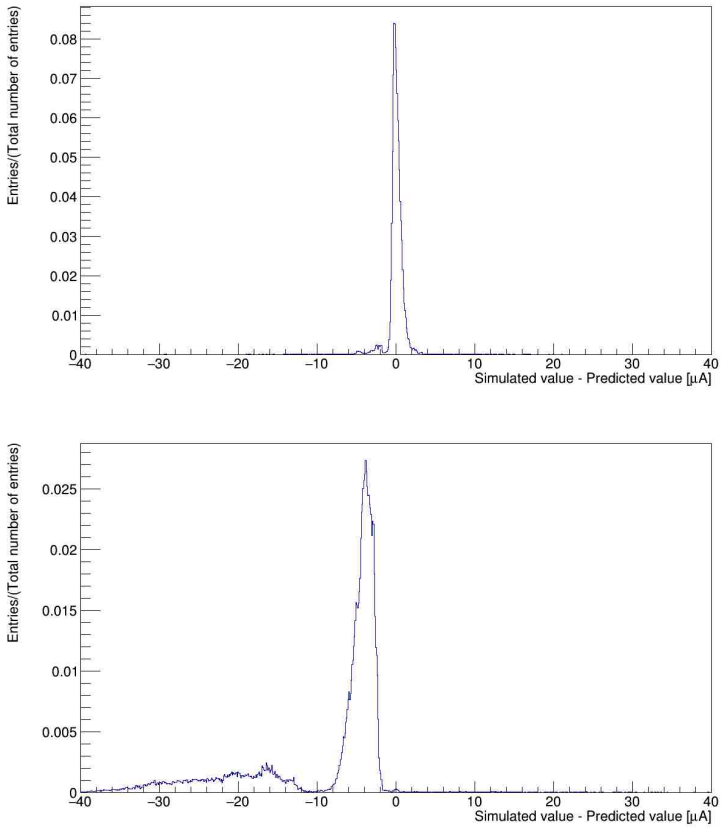

Figure 3. Distributions of the difference between the "measured" (in our case, MC-generated) and the predicted by the ML autoencoder tool detector current for a properly functioning sample (top panel) and for a test sample with simulated problematic behavior (bottom panel).

A sample distribution of the differences between the simulated detector rate and the one predicted by the encoder is shown in Fig. 3. In the case of normal detector behavior, the distribution should be rather sharp and centered at the origin – like the one shown in the top panel for a properly functioning training set. In case of abnormal detector behavior, the distribution will exhibit deviations from this shape and/or drift away from the origin. The distribution in the bottom panel is obtained when running the ML tool over a sample simulating a defective detector module. The shift and the non-Gaussian form of the distribution confirm the ability of the autoencoder ML tool to spot the existing irregularity.

The ML tool was originally written in Python using TensorFlow for the machine-learning part. In order to run it on GPUs installed on multiple nodes, the tool was modified to use in addition to TensorFlow also Horovod, the latter employing an MPI back-end. In the appendix, the section of the code, elaborating on the TensorFlow – Horovod interface is given.

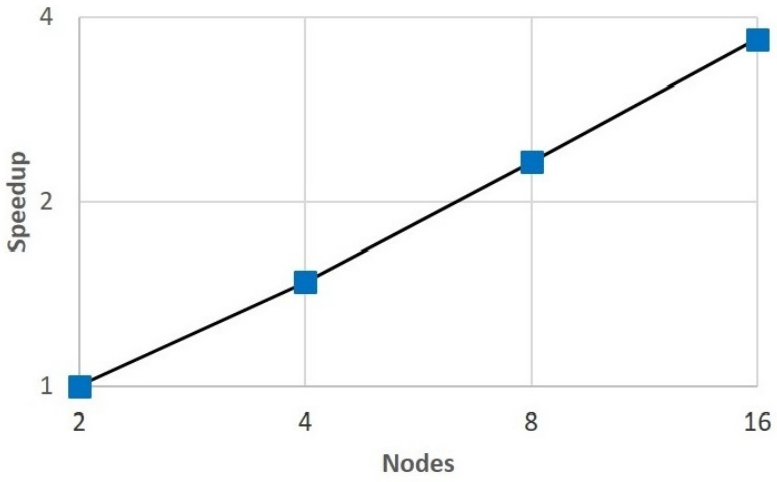

Figure 4. Performance of the autoencoder ML tool.

In Fig. 4 the speed-up achieved by the multiple-node execution of the autoencoder ML tool compared to the two-node execution is shown. The time used to calculate the speedup is estimated by employing the Horovod timeline as a profiling tool. With its help, performance problems and possible bottlenecks can be detected by analysing the output timeline files, which are in JSON format and can be visualised using the Chrome browser tracing facility [13]. In particular, it is thus possible to exclude the time used by Horovod for initial broadcasting and initialisation and to account for the actual parallel code execution only.

Fig. 5 illustrates Horovod timeline plots for the autoencoder ML tool execution on two and eight nodes respectively. As seen, the execution of parallel routines compared to the initialisation step takes less time in the case of eight nodes (bottom panel) than in the case of two nodes (top panel), justifying the above-outlined speed-up evaluation approach. It should be noted that Horovod timelines store information about the function calls but no hardware-related information (number of nodes, etc.). The profiling information is saved in a JSON file. According to the instructions the files are viewed by the chrome://tracing tool. The Chrome tracing tool does not allow for the selection or alignment of the timing of the plots. However, in order to make the comparison possible we have highlighted and measured the duration of a typical parallel step on the plots. The duration is clearly visible on the plots and is approximately 2.527 s in the case of 2 nodes and 0.968 s in the case of 8 nodes.

6. Summary

We conducted a *proof-of-concept* study on Machine Learning potential in monitoring complex detector systems. As a testbed, we used an emulation of a real detector complex like the ones used in the experiments at the LHC accelerator at CERN, consisting of 512 detector units. Both the training and the validation data were generated with a custom-design Monte Carlo tool. For detector monitoring, an autoencoder ML tool was developed, its ability to run on GPUs at different nodes being ensured by using TensorFlow. The tool performs a continuous comparison between the predicted and the measured values of a set of parameters, essential for the detector operation diagnostics. Both data-synchronisation and ML tools were installed and validated on CINECA's Marconi100 machine. The tool successfully detected all simulated irregularities. Performance results in terms of speedup with respect to the one-node execution of the code are encouraging. Though the toolkit validation was performed on accumulated

Figure 5. Horovod timeline plots for the ML tool execution on two (top panel) and eight (bottom panel) nodes.

data, the toolkit is designed to work online as well, for real-time detector monitoring and data certification. Further improvement of the ML tool can be possibly achieved by including supervised learning algorithms.

Acknowledgments

This work was supported in part by the PRACE-6IP Project funded under Horison 2020 Program, Grant agreement ID 823767. The reported results were obtained using Avitohol (BAS, Bulgaria) [11] and the PRACE Research Infrastructure, in particular Marconi100 (CINECA, Italy) [8].

References

[1] ATLAS Collaboration, “The ATLAS Experiment at the CERN Large Hadron Collider", JINST 3 (2008) S08003, DOI:10.1088/1748-0221/3/08/S08003

[2] CMS Collaboration, “The CMS Experiment at the CERN LHC", JINST 3 (2008) S08004, DOI:10.1088/1748-0221/3/08/S08004

[3] ALICE Collaboration, “The ALICE experiment at the CERN LHC", JINST 3 (2008) S08002, DOI:10.1088/1748-0221/3/08/S08002

[4] LHCb Collaboration, “The LHCb Detector at the LHC", JINST 3 (2008) S08005, DOI:10.1088/1748-0221/3/08/S08005

[5] CMS Collaboration, “Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC", Phys. Lett. B 716, 30–61 (2012). DOI:10.1016/j.physletb.2012.08.021

[6] ATLAS Collaboration, “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, 1–29 (2012). DOI:10.1016/j.physletb.2012.08.020

[7] A. Samalan et al, “A new approach for CMS RPC current monitoring using Machine Learning techniques“, 2020 JINST 15 C10009, DOI:10.1088/1748-0221/15/10/C10009

[8] Marconi100 Supercomputer, https://www.hpc.cineca.it/hardware/marconi100

[9] Horovod, https://github.com/horovod/horovod

[10] TensorFlow, https://www.tensorflow.org/

[11] Avitohol HPC System, http://www.hpc.acad.bg/system-1/

[12] The ML Monitoring Tool,
https://github.com/bapavlov/ML_with_TensorFlow_and_Horovod/blob/main/tensorflow2_keras_GPUs_v0.py

[13] Horovod Timeline, https://horovod.readthedocs.io/en/stable/timeline_include.html

[14] Horovod–TensorFlow interface, https://horovod.readthedocs.io/en/stable/tensorflow.html

Appendix

A useful sample code interfacing TensorFlow to Horovod, based on [14]:

```
import tensorflow as tf
import horovod.tensorflow.keras as hvd
import numpy as np

# Horovod: initialize Horovod.
hvd.init()

# Horovod: pin GPU to be used to process local rank (one GPU per process)
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

# Load data, etc.
.....
.....
......
.......................................................
input_img = tf.keras.Input(shape=(x_train.shape[-1],))
encoded = tf.keras.layers.Dense(512, activation='relu')(input_img)
encoded = tf.keras.layers.Dense(128, activation='relu')(encoded)
encoded = tf.keras.layers.Dense(64, activation='relu')(encoded)
decoded = tf.keras.layers.Dense(128, activation='relu')(encoded)
decoded = tf.keras.layers.Dense(512, activation='relu')(decoded)
decoded = tf.keras.layers.Dense(x_train.shape[-1], activation='relu')(decoded)

loss_fn = tf.keras.losses.MeanSquaredError()

# Horovod: adjust learning rate based on number of GPUs.
scaled_lr = 0.001 * hvd.size()
opt = tf.optimizers.Adam(scaled_lr)

# Horovod: add Horovod DistributedOptimizer.
opt = hvd.DistributedOptimizer(opt)

# Horovod: Specify `experimental_run_tf_function=False` to ensure TensorFlow
# uses hvd.DistributedOptimizer() to compute gradients.
autoencoder = tf.keras.Model(input_img, decoded)
```

```
autoencoder.compile(optimizer=opt, loss=loss_fn)
.....................................................
callbacks = [
# Horovod: broadcast initial variable states from rank 0 to all other processes.
# This is necessary to ensure consistent initialization of all workers when
# training is started with random weights or restored from a checkpoint.
hvd.callbacks.BroadcastGlobalVariablesCallback(0),
# Horovod: average metrics among workers at the end of every epoch.
# Note: This callback must be in the list before the ReduceLROnPlateau,
# TensorBoard or other metrics-based callbacks.
hvd.callbacks.MetricAverageCallback(),
# Horovod: using `lr = 1.0 * hvd.size()` from the very beginning leads to worse final
# accuracy. Scale the learning rate `lr = 1.0` ---> `lr = 1.0 * hvd.size()` during
# the first three epochs. See https://arxiv.org/abs/1706.02677 for details.
hvd.callbacks.LearningRateWarmupCallback(initial_lr=scaled_lr,
warmup_epochs=3, verbose=1),
]

# Horovod: save checkpoints only on worker 0 to prevent other workers
from corrupting them.
# Horovod: write logs on worker 0.
verbose = 1 if hvd.rank() == 0 else 0

autoencoder.fit(x_train, x_train, steps_per_epoch=500 // hvd.size(),
callbacks=callbacks, epochs=6, validation_steps=10, validation_data=(x_test, x_test),
verbose=verbose)
............
```