CHAPTER 1

Machine Learning for Monte Carlo simulations

COMPUTING techniques loosely based on mimicking the behavior of the human brain are becoming more and more important in a vast range of applications. Their utilization is not new, with studies based on simple Neural Networks [**?**, **?**, **?**] dating back at least to the 80s; what is instead quite recent is the possibility to deploy efficient computing architectures, often specifically tailored to the tasks. At the same time, the capability to deploy larger and larger system has triggered theoretical studies, driving to more solid bases and to the definition of more complex and specialized models.

In these chapter we will start with an introduction to the model most relevant for Monte Carlo simulations, followed by a selection of applications. In the last part of the chapter, we will review the strong and weak points about the utilization of Neural Networks applications for Monte Carlo simulations.

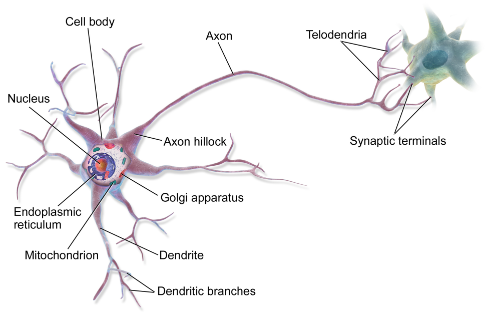
1.1 INTRODUCTION TO NEURAL NETWORKS

Neural Networks are a specific branch of the Artificial Intelligence (AI) domain in computer science. They get their inspiration from the fact that humans are evidently able to fulfill complex tasks; hence, by replicating the low-level mechanisms of the human brain on computing systems, one can potentially construct high level algorithms with similar capabilities.

1.1.1 The human brain

Neglecting any functional description, the human brain can be described as an organ composed by neurons, glial cells, neural stem cells and blood vessels (Figure 1.1a). With our current understanding, the neurons are the units

(a) A pictorial view of the human brain (from Wikipedia).



(b) The human neuron (from Wikipedia, by BruceBiaus, CC BY 3.0).

performing basic "operations" within the human brain, and their aggregated response is responsible for the high-level behaviour typical of humans. A neuron, as sketched in Figure 1.1b, is composed of three main units: a number of dendrites, the soma (the cell body), and an axion; the total size largely varies between different types of neurons; the neurons used for cognitive functions (as those in the grey matter of the brain) are usually short, XX μm. Functionally, a neuron is able to generate an electric response on the axion (*output*), depending on the electrical potential present at the synapses (*inputs*) present on the dendrites, generating a quite low-level response mechanism. Neurons are *chained* by connections between axions and dendrites, generating a mesh in which N neurons are connected via M synapses. The high-level response of the human brain to stimuli is understood to come from the complexity of such mesh, with a standard human brain featuring 10^{11} neurons each with 7000 synapses, for a total of 10^{15} "connections".

In literature various models of the neuron behavior have been proposed [**?**, **?**, **?**], here we will focus on the simplest yet most simple to implement in computer systems [**?**] (see Figure 1.2):

in this model, the *output* y signal at the axion is assumed to be a function

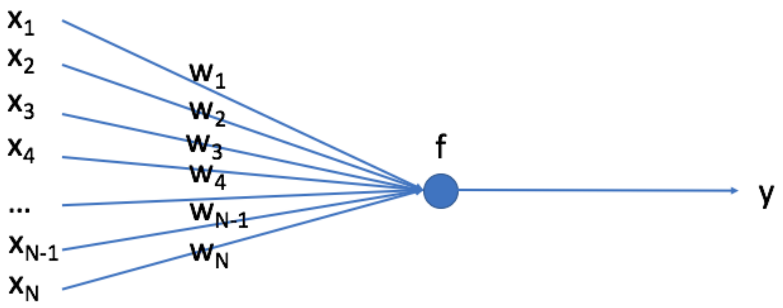


FIGURE 1.2: The artificial neuron.

of the *inputs* x_i via

$$y = f(\sum_{i=1}^{N} w_i x_i) \tag{1.1}$$

where w_i are weights defined by chemical potentials at the synapses, and the function f wants to model the non linearity of response of biological neurons with the *inputs*; on top of this, the function f is needed in the mathematical model in order to allow the description of non linear phenomena [**?**]. The perceptron [**?**], one of the first models used in literature for Neural Networks, uses a very similar model, with a simplified f function which is simply

$$f(\vec{x}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{N} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

Today, two small modifications are typical when using Neural Networks:

- the addition of a further synapse x_0 which is always 1, as a bias to the system; its weight is referred to as x_0 or b (as in *bias*).
- the use of continuous f non linear functions, as the logistic [**?**] or the hyperbolic [**?**] functions.

Neural networks are obtained by combining multiple neurons in *networks*, usually in a layered structure: one layer is used to map the inputs, a few/many layers are *hidden*, and a single layer used to to map the outputs. On top of that, more complex neurons an be used, for example including a ”memory” cell, or presenting a recurrent behavior by reusing its output as one of the inputs. A full description of all the type of neurons and networks is beyond the scope of this chapter; in the following, the ones most relevant to Monte Carlo simulations will be presented wit more detail. For reference, still, a complete classification of currently relevant neural networks is shown in Figure 1.3. As clearly visible in the figure, some network topologies have an high number of hidden layers. While the Universal Approximation Theorem [?] states that under quite generic conditions, a single hidden layer between inputs and outputs should be enough in all cases, networks used in science during the last decade tend to be "deep" (i.e. with many hidden layers). This has multiple motivations: on one side, the theorem states that it is possible to have just one hidden layers, but does not state with how many neurons (and it tends to be a very large number); on the other side, a deep structure tend to be better human readable, with cascade sub networks with a more identifiable and logical role. Hence, relevant networks in today's science tend to be deep.

The typical utilization pattern for a majority of network topologies is to feed them as input a large set of data representing the problem of interest, be it a medical image, a set of features or any output from the instrumentation, and at the same time provide the "expected output" from a so-called training set.

In the simplest type of networks, the response from the neurons in the hidden and the final layers are considered in sequence (inputs to outputs), with each layer *feeding* the following layers; hence the name Feed Forward Neural Networks (FFNNs).

During the training, the network adjusts its internal free degrees of freedom (the weights w_i^j in equation 1.1, extended to the j neurons in the various layers) to better reproduce the desired answers, via minimization procedures which can be either numerical or analytical.

What has just been described is the training process for *supervised* Neural networks, which rely on an externally provided "truth" to adjust for optimal performance, without having any a-priori knowledge of the physical process they want to reproduce. Other topologies describe instead the *unsupervised* Neural Networks, in which the training process just implies the utilization of datasets without the need to provide the correct answers". Examples of such networks will be provided in Section 1.1.4.1.

1.1.2 Convolutional networks

Convolutional networks (CNNs) are a useful subset of neural networks, which exhibit peculiar characteristics of being space invariant with respect of the inputs.

They are particularly interesting in the realm of Monte Carlo simulations, since the space invariance is a valuable characteristics: for example, a shower from an hadronic particle into a material, when far from the edges, does not depend on the specific entry point nor on its direction. CNNs are used with success in categorization problems, where typical structures must be discovered into a set of input data: a typical example is the identification of lesions in medical imaging, trained on reported images by clinicians [?], as depicted in Figure **??**. Technically, CNNs use basic neurons as explained in Section 1.1.1, with those in the hidden layers fed by small portions of the inputs per iteration, thus realizing spatial independence, for example. Multiple application of several convolutional layers drive to an overall analysis on the inputs, and to the final outputs.

1.1.3 Recurrent networks

The CNNs just described are a type of *stateless* network, in which the output depends only on the inputs presented on the first layer. In Recurrent Neural Networks (RNNs), the response to a specific input feature set also depends on the history of the network, i.e. the inputs presented at prior times; as such, the network presents a sort of "memory", which can be used to correlate multiple inputs in sequence. A typical utilization pattern is in the presence of inputs of variable length, which cannot be estimated *a priori*, like in the analysis of text or speech sequences. In simulation processes, the feature is often utilized when the signals from a non fixed number of inputs (i.e. the incoming particles to a volume) must be piled up in a coherent way: the inputs per each particle are presented to the network, with a specific input pattern which may or may not be used to signal the end of the sequence.

1.1.4 Generative Models

The networks popular up to 10 years ago were mostly useful during a decision process, such has categorizing inputs (signal vs background, for example) or counting and defining specific regions inside it (segmentation, counting of lesions, etc).

In order to be applied to Monte Carlo simulations, instead, the capability to produce ("generate") an output as close as possible to reality, or to a more standard algorithm. In order to do this, different network topologies and strategies for training are relevant.

1.1.4.1 Auto-Encoders and Variational Auto-Encoders

The autoencoders are a family of unsupervised Neural Networks designed to learn a lower dimensional representation (say N dimensions) out of a set of data (with M dimensions, M >N). The N dimension representation ("latent space") can be seen as coming from an "understanding" of internal patterns and correlations, and thus as a Neural Network which has been "discovering" these underlying patterns in the input data.

The easiest form of autoencoder is one in which the number of inputs and outputs equals to M, and where there is a layer at dimensionality N. The training is obtained by forcing the network to reproduce as close a possible the input features at the output layer, thus requesting the N-dimension space to be as sufficient as possible to represent the M-dimension inputs (see Figure 1.4).

Autoencoders in such form are used for two distinct purposes:

- auto-discover in the inputs hidden symmetries or underlying correlations, which can be used, for example, in lossy compression [?] scheme or to drive understanding on the inputs themselves;

- since the network is trained on a specific data sample, it will minimize the difference outputs vs inputs on that specific dataset. Once the same network is presented with "different" data, it is expected to fail to reproduce the inputs at the outputs; hence, it can be used to detect anomalous inputs, as important for the detection of not expected features, for example in medical imaging [?] or High Energy Physics events [?].

Variational AutoEncoders (VAEs) are autoencoders which can be used to *generate* realistic data, matching as close as possible the training set. The naive way to generate data from a standard autoencoder is to generate random configurations in the latent space of dimensionality N, and decode it with the rightmost part of Figure 1.4 in order to obtain a M dimensional configuration which is "generated" by the network. In practice, there is no guarantee that the latent space gets organized in a way that all its points are used in the decoding/encoding process. This is where Variational Autoencoders differ: it is explicitly trained in such a way to ensure that the latent space is apt for a generative process.

... ...

1.1.4.2 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a recent [?] class of networks designed to reproduce the behaviour of complex systems without an explicit programming.

The methodology (see Figure 1.5 is to put in competition the reference algorithm (or real data, A) with a generative network (B), which produces an output with the same structure and whose response, initially, is random. A second network (C) is trained on the capability to distinguish between the algorithm's output and the generated one. The two networks are put in competition (hence the term *adversarial*), with B winning if it can convince C its response is the real one; in the opposite case, C wins. The tension between the two networks pushed the network B to generate outputs, which are indistinguishable from the real data / the output algorithms. B and C are generic networks, with Convolutional neural structures mostly used.

GANs are more and more used when in the need for replicating the behaviour of a known data source, with improved computing performance. Upon successful training, in fact, the network B has the typical speed of (say) a CNN, and is able to reproduce the output of A, which can be a complex and time consuming algorithm. Successful examples are the capability to reproduce the showering of particles in calorimeters [?, ?], the famous generation of realistic faces [?], or for the automatic segmentation of medical images [?].

1.1.4.3 Graph Networks

In a category of problems, the task is to discover the relations between objects; think for example of the segmentation of medical images, where it has to be understood whether certain areas are part of a specific organs, or of clusterization algorithms in many science realms. The most used network architecture for these use cases is a Graph Neural Network (GNN, [?]). In a GNN, nodes (representing objects) and edges (representing connections between nodes) are the basic entities, and the training process aims to correctly categorize the nodes, while at the same time assessing the strength of each connection. GNNs are finding large applications in cases where one wants to replace a complex, combinatorial algorithms with a Neural Network: typical examples in literature are Jet clustering in High Energy Physics [?], and tracking in dense particle environments [?].

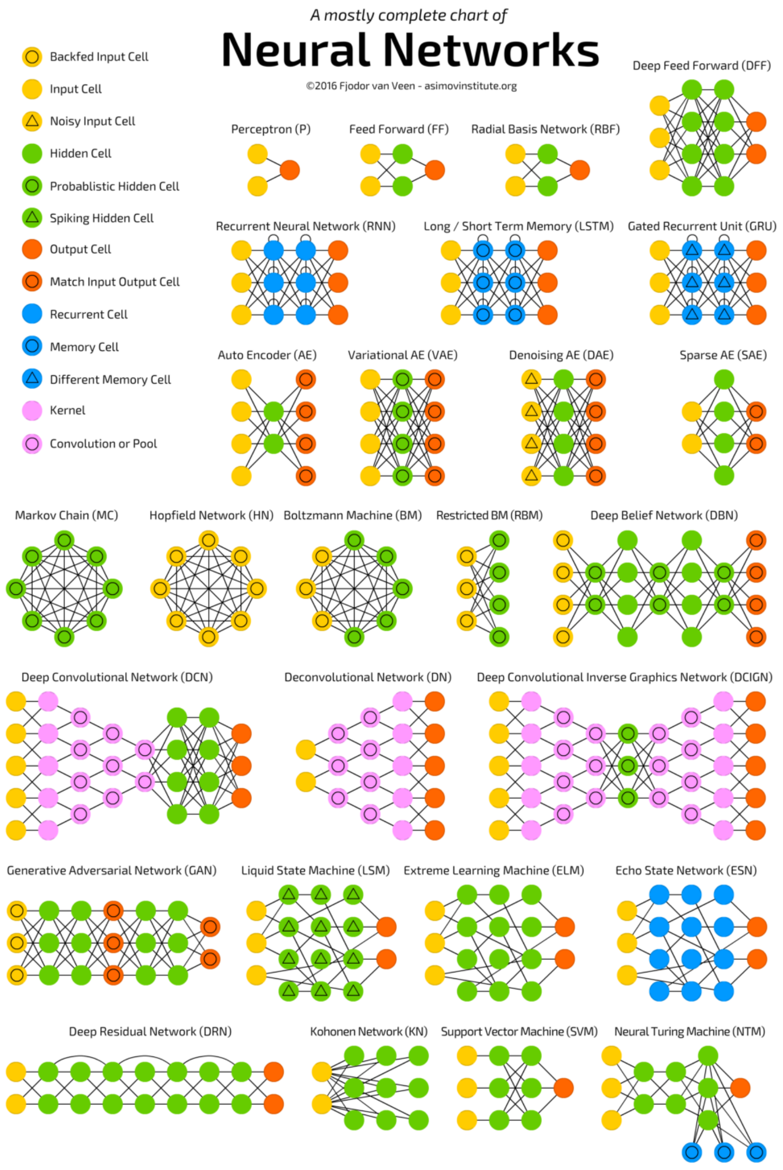

A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

FIGURE 1.3: Types of neurons and neural networks currently relevant in literature (Copyright F. van Veen 2016).

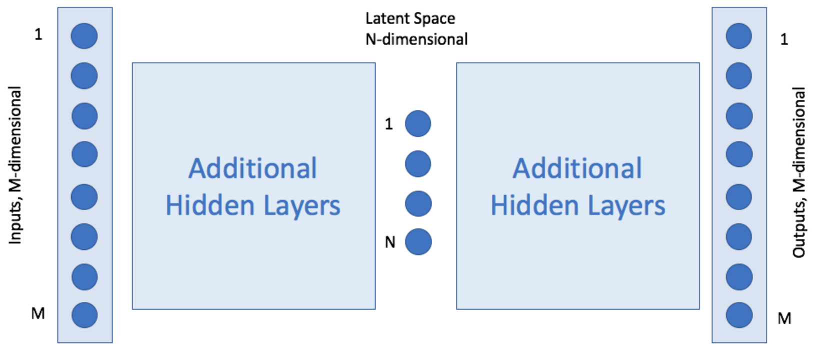


FIGURE 1.4: An autoencoder in its simplest form.

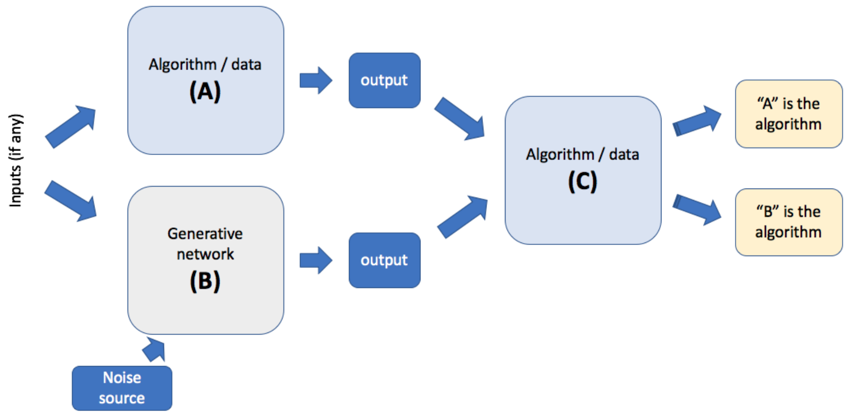


FIGURE 1.5: The structure of a simple GAN.

1.2 EXAMPLE OF APPLICATIONS FOR MONTE CARLO SIMULATIONS

1.2.1 Emulation of electromagnetic interaction models

1.2.2 Emulation of low energy nuclear interaction models

1.2.3 Emulation of radiation-matter interactions

DOES THIS GO TO THE INITIAL PART OF THE SECTION??? One of the most complex tasks in Monte Carlo simulations involving the use of detectors (medical apparatuses, particle physics detectors) lies in the dual need of being able to optimize the design before detector construction, and to simulate the behavior under working conditions after the setup has been prepared. In both cases, unless the setup is very similar to existing detectors, extensive Monte Carlo simulations of the expected detector capabilities are the widely used solutions. Various such tools exist (Geant4[**?**], Fluka[**?**], MCNPX[**?**]), with different application regimes and specific utilization patterns. as a general rule, these implement iteratively basic radiation-matter low-level processes to a knowledge of the detector setup, including materials, geometry and XXX; as such, they incur into two general limits:

- a scarce capability to be tuned to experimental results, by changing the basic modelling of the processes;
- a varge to very large need for computational resources, given the iteration oriented approach and the need to increase the level of iterations in order to obtain a better precision and adherence to data.

Both limitations can in principle be overridden via the use of Artificial Intelligence oriented tools. In presence of experimental data, the response of the AI system can be tuned to that without any explicit modelling of the physics processes; speed can be vastly improved by the change from iterative-based computations to standard Deep Learning matrix algebra, with intrinsic capabilities for high performance processing on, for example, GPU systems.

As an example we want to consider here CaloGan[**?**], an attempt to reproduce the details of radiation-matter interactions in the complex setup of segmented (3 layers) electromagnetic calorimeters. A generative Adversarial Network, as those presented in Section **??**, is used in conjunction with an as much-as-accurate as possible Geant4 simulation of the detector setup. The generator side accepts in input input particle 4-momentum, ad after the passage through quite standard convolutional (matching the detector response as 2-D images) and dense layers, the output is compared with Geant4 simulations of a particle with the same parameters. The training optimizes the energy deposition per layer and per 2-D transverse cell, in a way in principle suited also for using real data in input. Results are very encouraging, even in a detector scenario which can be considered as extreme: not only the quantities of direct training are well reproduced, but also secondary and derived quantities like shower shapes are in most cases well described.

ADD DISCUSSION and PLOT

A second similar attempt, applied to the not-yet existing CLIC proposed electromagnetic and hadronic calorimeter, is presented in [**?**], with the goal to directly reproduce 3-D signals in a high granular calorimeter. The reference dataset, in absence of real data, is has the form of Geant4 generated showers sampled in a 25x25x25 cells around the impinging particle. Figure **??** shows the longitudinal shower shapes for 100 GeV electrons in the electromagnetic part of the calorimeter compared with detailed Geant4 simulations. The level of agreement is very satisfactory.

In sections 1.3.1 and 1.3.3 we will discuss about the speed gain with respect to standard methods, and solutions and needs to prevent unphysical results.

1.2.4 Emulation of detector responses

Monte Carlo tools like Geant4 are designed to simulate, as accurately as possible, the energy deposition (in keV, for example) happening because of the passage on particle / radiation in the material of a detector. In real life, what a scientist measures is instead the response, in forms of analog or digital signals, of a measuring device in which energy deposition is read and processed by some electronics. So, in classical systems, the simulation of radiation / matter must be followed by an ad-hoc simulation of the electronic readout system, in order to be compared with actual readings from a detector. In the case of AI inspired tools, this can be avoided by completely bypassing the ”energy deposition” output results, and training the system directly with real or realistic (from the above ad-hoc simulation) signals from the electronic back-end. In [**?**], for example, images from a real MRI apparatus are used for the training the system (see Figure 1.6); the use of more classical approaches would be indeed more problematic since there is no practical way to measure (and validate) the output in terms of energy deposition in a running system.

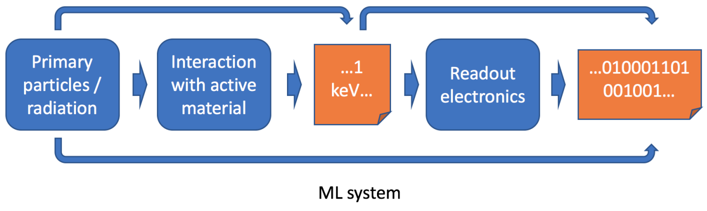

Primary particles / radiation
Interaction with active material
...1 keV...
Readout electronics
...010001101 001001...
ML system

FIGURE 1.6: Difference between classical and AI inspired simulation of experimental setups.

1.3 STRONG AND WEAK POINTS ABOUT NEURAL NETWORKS APPLICATIONS FOR MONTE CARLO SIMULATIONS

1.3.1 Speed, accuracy and reliability

As described in Section 1.2.3, one of the main advantage of AI inspired systems, when compared to more standard simulation algorithms, is a potential speed-up without necessarily impacting performance. The improvement comes from the nature of feed-forwards Neural Networks used at inference time, which are described as a series of matrix multiplication of fixed length[1], without access to any data structure apart from the inputs and the weights. Today's processors, from CPUs to GPUs to TPUs, are very efficient in processing matrix algebra calculations; this reflects diretly to short processing times. In order to give quantitative examples, we can refer to the two complex GANs described in Section 1.2.3, which are fed by Geant4-processed events in the training phase. Table 1.3.1 reports on the absolute time needed to process Geant4 vs the GAN for events with similar input; the speedups are of order 100x using CPUs, and an additional factor 100x with GPUs[2].

System	Software	Speed
Intel Xeon E5-2683	Geant4	1 min
Nvidia RTX 1080	3d-GAN	0.04 msec
AWS `p2.8xlarge`	Geant4	1.7 sec
AWS `p2.8xlarge`	CaloGan	O(10) msec
AWS `p2.8xlarge` + Nvidia K80	CaloGan	O(0.01) msec

Fig 1.3.1: Geant4 vs GAN performance under various setups, as extracted from [**?**] and [**?**].

1.3.2 Response to unexpected (untrained) signals

1.3.3 Unphysical responses and how to impose physical constraints

AI inspired tools are a mathematical solution to problems we have problems solving algorithmically, either because too difficult or too slow. An important difference with respect to standard "human-written" code is impossibility to impose at algorithmic level precise conservation laws, such as the conservation of energy and momentum. There is no guarantee a ML system will conserve them, since their validity is not implemented in any explicit form.

In most of the cases shown in Section 1.2 the strict conservation (for example) of energy in particle showering into calorimeters does not need to be exact, since in the transfer between the energy in the impinging particle and

[1]This is not strictly valid for some types of recurrent networks, where the recurrence can introduce an indetermination in the sequence of mathematical operations.

[2]It can be said that it is an unfair comparison since Geant4 cannot currently use GPUs; still, it shows how AI inspired tools offer a path to the use of more performing technologies.

the final products is in any case approximate due to effects of leaking, and smoothed by the detector resolution. In cases like this, while the training tries to match the "cell by cell" energy deposition, there is no explicit request that that the total energy deposited in the calorimeter matches the precise simulation. In principle, a loss function designed to have the minimum in when all the cells have the "expected" energy would suffice, but experience (and somehow common sense) shows that by adding to the loss function an explicit term tending to the conservation of total energy, like in Eqn. (2) in [**?**], constrains can implemented even if not in exact form. This solution is called "soft" precisely for this reason. . The relative weight of the various terms in the loss functions are somehow arbitrary, and an higher weight to the part tending to the constraint (not really imposing it!) represents the developer desire to have it more precisely maintained, even if never exactly.

A different approach, as presented in [**?**], tries to implement instead "hard" and exact constraints on the outputs during the training phase, the mathematical complexity and the time needed increase, such as to advice the use of (eventually up-weighted) "soft" constraints in any case.

Bibliography

[1]