EM shower simulator with Neural Network

Cafasso Dario, Passaro Daniele

University of Pisa, Physics Department "E. Fermi"

ABSTRACT

Recent developments in calorimeter physics are leading to a new paradigm in calorimeters' design. The aim is to reconstruct the whole spatial distribution of the shower instead of extracting information only about the energy deposition inside calorimeter cells. Many collaborations are designing highly-segmented calorimeters (for instance, CMS High-Granularity CAL and ALICE FoCAL) to reach this goal. In this project, our goal is to build a neural network to simulate electromagnetic shower energy deposition inside a toy segmented calorimeter. Taking inspiration from recent works in this field, like the CaloGAN network, we have built a GAN with auxiliary conditions based on total energy deposition and particle IDs. The neural network was trained with a dataset created on purpose with the Geant4 simulation toolkit. Simulations involve electrons, positrons and photons with energies from 1 to 30 GeV that strike on a CsI calorimeter with 12 layers and 25×25 cells per layer. The results are further analyzed with ROOT to evaluate GAN's performances. Due to time, dataset and hardware constraints, this project must be considered an exploratory work whose results can even be improved with more resources.

Physics and reasons for our work

Electromagnetic showers usually generate when a sufficiently energetic $e^{\pm}$ or photon interacts with matter. The main processes through which the shower develops are bremsstrahlung and pair prodution: in the first process, an $e^{\pm}$ is deflected by an atomic nucleus (or an atomic electron) emitting a photon; in the latter one, a photon interacts with an atomic nucleus and breaks producing a pair of $e^{\pm}$, with a threshold energy of $E_\gamma = 2 \cdot m_e \approx 1.022\,\text{MeV}$. An electromagnetic shower develops after the incoming particle (electron, for instance) produces a photon by bremsstrahlung; then, if it is energetic enough, it goes pair production, and the process continues until photons energy is below the threshold and bremsstrahlung becomes disfavoured in respect to ionization. Because of the physical processes involved in EM shower, their development is in first approximation exponential in the primary particle energy, so increasing the first particle's energy produces a shower with growing complexity in terms of generated particles. It's particularly relevant while simulating EM showers with very high energy. The physical parameters which describe these processes are radiation lenght X_0, the characteristic length for energy loss by bremsstrahlung, and the photon absorption lenght λ_γ; they are related by $\lambda_\gamma = \frac{9}{7} X_0$, which essentially means that a shower produced by a primary photon will be, on average, deeper than ones with a primary $e^{\pm}$.

EM showers are frequent in particle physics, mainly in accelerator experiments and cosmic rays experiments. In accelerator physics, highly energetic γ and $e^{\pm}$ are produced by the decay of massive particles ($H \to \gamma\gamma$ or $W \to e^- \overline{\nu_e}$ for instance); EM calorimeters are suited to detect these particles and to measure their energy by measuring the energy deposition within it. Historically, calorimeters were designed so that each detector module could completely absorb the incoming particle's energy. In this way, the energy resolution for EM calorimeters is optimal, but the position resolution is bound to the module's dimensions. Nowadays, many experiments are re-designing their calorimeters, to measure both the shower's energy and the spatial development of the shower (to resolve multi-jet events generated by the decays of vector bosons). Particularly challenging are events simulations with this kind of detector topology, where the main problems are related to the computation time. Inspired by that, our goal has been to build a neural network capable of simulating EM showers inside a toy segmented calorimeter.

Geant4 simulations

We have developed our physics simulations using Geant4 software. So, we could have full control over the simulation dataset, which we treat as the "real" dataset. We have used a pre-compiled version of Geant4, called "GEARS" (https://github.com/jintonic/gears), in which we only need to specify the geometry of the detector, the physics list to use during the computation, and primary particle source. Our toy detector is composed of 12 layers of 25×25 cells per layer, where each cell is $1 \times 1 \times 5\,\text{cm}^3$ wide; the material used is cesium iodide CsI. We decided to use this material since its radiation length is $X_0 \approx 1.9\,\text{cm} \ll 25\,\text{cm}$ (lateral side of the detector), and the longitudinal depth for 98% containment $t^{98\%} \sim 40.80\,\text{cm} < 60\,\text{cm}$ (total depth of the calorimeter): the shower would have been fully contained inside the detector.

We performed simulations of shower development for $e^{\pm}, \gamma$ primary particle with initial energy $1\,\text{GeV} < E_0 < 30\,\text{GeV}$. Potentially Geant4 simulates higher energy too, but we restrained the energy range due to hardware memory limitation. The output of GEARS is a .root file containing a tree whose branches contain the simulated events. The branches are particle ID, track ID, particle energy, XYZ position, and many more. Into them, the data are stored in various step points (discrete points where Geant4 simulation is performed). This file is processed using the ROOT toolkit, to format the data from the simulation in a suitable form for the multivariate analysis. In particular, we produce a .root file containing a tree, whose branches are primary particle ID, primary particle energy, total energy deposited inside the calorimeter, and a $12 \times 25 \times 25 \times 1$ vector containing the log-energy deposited inside each cell, normalized to the absolute maximum of log-energy deposition in the whole simulation dataset, while empty cells are set to -1 as default. So, the training dataset ($\mathcal{I}$) for the neural network - as a vector of 12 images with 25×25 pixels - is already normalized to $[-1; 1]$ and we have a well-defined go-back algorithm to reconstruct the true energy of the shower.

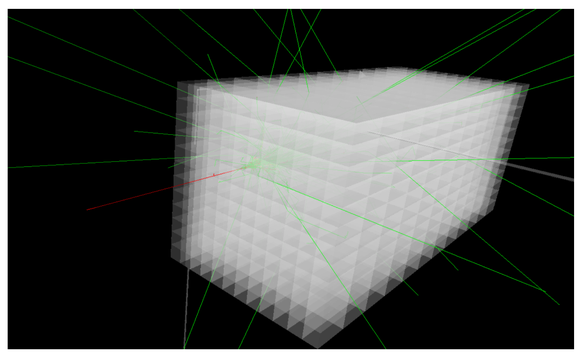


Figure 1. 3D visualization of a 2 GeV electron striking on the calorimeter, simulated using Geant4 and GEARS.

cafasso.dario@gmail.com d.passaro1@studenti.unipi.it

EM shower simulator with Neural Network

Cafasso Dario, Passaro Daniele

University of Pisa, Physics Department ”E. Fermi”

GAN structure

To randomly simulate the EM showers we decided to build a GAN (generative adversarial network) with features implemented ad hoc for our purpose. We used Keras layers and the TensorFlow package to build and manage our network.

A GAN consists of a generator network and a discriminator one, which are trained against each other: the generator is awarded if it manages to cheat the discriminator, while the latter is awarded if it succeeds to recognize fake samples coming from the generator.

The generator network has 3 inputs: normally distributed random noise from a latent space $\mathbb{R}^{1024}$, integer primary particle ID (P_{in} = 0 for electrons, 1 for photons and 2 for positrons) and primary particle energy (E_{in}, from 1.0 to 30.0) . Primary particle ID is connected to an embedding layer to create a discrete category for each particle, and then it is concatenated with the other inputs. A succession of Conv3DTranspose, BatchNormalization and LeakyRelu layers is present. Its final output (hereon said $\mathcal{G}$) is a $(\text{None}, 12, 25, 25, 1)$ tensor whose activation function is tanh, so that each pixel takes value in $[-1; 1]$. Conv3DTranspose layers deconvolve inputs maintaining high levels of connectivity between feature maps, BatchNormalization is important to help with weight initialization and gradient stability, and LeakyRelu activation is useful to prevent vanishing gradient and to provide a higher level of sparsity.

The discriminator network takes as only input a $(\text{None}, 12, 25, 25, 1)$ tensor, which is the images vector from the generator or one from the Geant4 dataset. It works as a classifier: it decides (outputs a decision ”$\mathcal{D}$”) whether the input is fake ($\mathcal{G}$) or real ($\mathcal{I}$). Moreover, it decides the primary particle generating the shower ($P^{label}_{GAN/GEANT}$) and has to recognize the primary particle energy ($E^{label}_{GAN/GEANT}$) (actually, this label helps to relate shower’s shape to primary particle energy). For this purpose, the input tensor is concatenated with a computed tensor whose elements are the non-normalized energies deposited in each detector’s layer. This operation is performed with a costume Lambda layer. In this way, we observed that the discriminator better recognizes fake from original showers, enhancing the generator to produce images with the ”correct” energy deposition per layer. Then a succession of Conv3D, Pooling3D and LeayRelu layers is applied to the input tensor. Pooling layers help to resolve spatial characteristics of the shower, like their maximum and shape. After a convolutive stage, a Flatten layer that passes the output tensor to 3 different paths is present. Each one represents one of the decisions above. Dense layers are present to increase the deepness of the network and increase connection among neurons. To avoid ”mode collapse”, which is an often common evolution of GANs, we inserted between convolutive layers a costume Lambda layer that performs ”minibatch standard deviation discrimination”. This technique has been conceived by NVIDIA researchers while developing a GAN to create ultra-realistic deep-fake faces. Its role is to compute the standard deviation of a group of input features. It concatenates the result back to the input tensor and passes the resulting tensor to a convolutive layer. In this way, the discriminator should be able to distinguish fake from original samples by looking at the standard deviation among features: for a diversified real dataset it will be high, while for a fake dataset where the generator has undergone mode collapse it will be small.

Recognizing it, the discriminator would enhance the generator to produce diversified samples. We adapted this idea to our case computing the standard deviation for each image (layer of the calorimeter) instead of the whole images vector, thus introducing a ”sparsity” metrics for each layer of the detector.

Losses implementation

We gave great importance to losses implementation. We managed the whole training step for the training of the GAN, directly handling the gradients of the computation. Besides the Binary Cross Entropy losses used for the generator and discriminator training (”gener_loss” and ”discr_loss”), we computed some auxiliary losses to make them able to learn how to generate EM showers. In particular, the terms that make up the losses are:

Generator:

- gener_loss $= \mathcal{L}_{\text{BCE}}\left(\mathbb{1}, \mathcal{D}(\mathcal{G})\right)$
  It quantifies the capability of the generator to cheat the discriminator.
- comp_en $= \lambda_E \cdot \mathcal{L}_{\text{MSE}}\left(E_{in}, E^{meas}_{GAN}\right)$
  It quantifies how much the energy deposition from the generator output is close to the primary particle energy. It is a kind of energy conservation implementation. Here E^{meas}_{GAN} is computed through the go-back algorithm said above.
- fake_label_en $= \lambda_E \cdot \mathcal{L}_{\text{MSE}}\left(E_{in}, E^{label}_{GAN}\right)$
  It quantifies how much the energy label in output to the discriminator, when applied to fake showers, is close to the initial energy. This helps the generator to produce showers with physical shapes.
- fake_part_id $= \mathcal{L}_{\text{BCE}}\left(P_{in}, P^{label}_{GAN}\right)$
  It quantifies how much the generator is capable to generate showers compatible with the primary particle ID.

Discriminator:

- discr_loss $= \mathcal{L}_{\text{BCE}}\left(0, \mathcal{D}(\mathcal{G})\right) + \mathcal{L}_{\text{BCE}}\left(\mathbb{1}, \mathcal{D}(\mathcal{I})\right)$
  It represents how good the discriminator is to identify generator showers as fake and Geant4 showers as real.
- real_label_en $= \lambda_E \cdot \mathcal{L}_{\text{MSE}}\left(E_{in}, E^{label}_{GEANT}\right)$
  This helps the discriminator to relate its energy label to the primary particle energy, checking only on Geant4 samples.
- real_part_id $= \mathcal{L}_{\text{BCE}}\left(P_{in}, P^{label}_{GEANT}\right)$
  This helps the discriminator to relate its PID label to the actual primary particle ID, checking only on Geant4 samples.

Here λ_E is a hyper-parameter used to normalize energy losses to the losses computed via Binary Cross Entropy. The total losses are constructed through the sum of the above terms. These are used as a basis to apply the gradients to the trainable variables of the whole network.

Usually, it is hard to define metrics for GANs to quantify their goodness in reproducing real samples. For this reason, we introduced some ”unbiased metrics” to keep track of the GAN’s evolution. These reflect the physical properties of EM showers. At each epoch, we compute these quantities over a generated batch and make a comparison with the ones computed over the whole GEANT4 dataset. The unbiased metrics we used are (see ”Analysis of performances”): 1) shower depth : $\hat{d}(E_{in}, P_{ID})$; 2) lateral depth : $\hat{\sigma}(E_{in}, P_{ID})$. We compute their mean values and standard deviations summing on energies and primary particle ID.

cafasso.dario@gmail.com d.passaro1@studenti.unipi.it

EM shower simulator with Neural Network

Cafasso Dario, Passaro Daniele

University of Pisa, Physics Department "E. Fermi"

Analysis of performances

Geant4 and GAN simulations are analyzed to infer the performances of the neural network. We did this through a C++ code, in which we also used the ROOT toolkit. Using $n = \#P_{ID}$, the physical parameters that we analyzed are:

- Deposited energy vs initial energy
- Mean energy deposition per layer
- Mean energy deposition per cell in each layer
- Shower mean depth per primary particle ID

$$\hat{d}(E_{in}, P_{ID}) = \frac{1}{n}\sum_{n}\left(\sum_{l=0}^{11}\frac{l \cdot E_l}{E_{in}}\right)$$

It represents the mean barycenter of showers in units of layer number.

- Shower depth's mean width per primary particle ID

$$\hat{w}_{long}(E_{in}, P_{ID}) = \frac{1}{n}\sum_{n}\left(\sqrt{\sum_{l=0}^{11}\frac{l^2 \cdot E_l}{E_{in}} - \left(\sum_{l=0}^{11}\frac{l \cdot E_l}{E_{in}}\right)^2}\right)$$

It represents the mean longitudinal width of showers in units of layer number.

- Shower mean lateral width per primary particle ID

$$\hat{\sigma}(E_{in}, P_{ID}) = \frac{1}{n}\sum_{n}\frac{1}{12}\sum_{l=0}^{11}\left(\sqrt{\frac{l^2 \cdot E_l^w}{E_{in}} - \left(\frac{l \cdot E_l^w}{E_{in}}\right)^2}\right)$$

where $$E_l^w = \sum_{n_x, n_y=0}^{24} E_{l,n_x,n_y} \cdot (n_x - 12)$$

It represents the mean standard deviation of shower lateral development, averaged over layers, in units of cell number, assuming center of the shower lying over the central cell.

Package structure

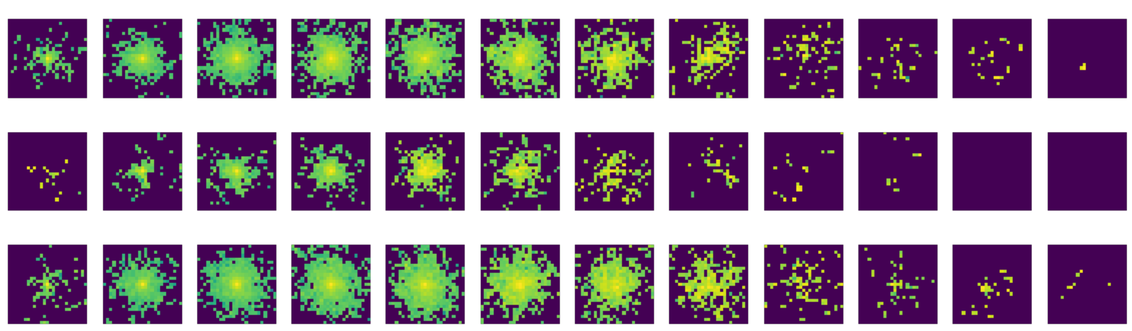
We built the project using GitHub as code hosting platform. The documentation has been created using Sphinx and is hosted by "Read the Docs". The repository has a package structure: it can be cloned and installed in development mode to generate customs showers. The setup.py file loads the package's metadata from the setup.cfg file, and then it installs the command-line interface (CLI) of the shower simulator and the analogue Python module. The versioneer module is used for automatic version detection. The unittest module is used for test writing and execution, while Travis CI has been chosen as continuous integration service.

It is possible to generate showers with specific or random features from bash or a Python interpreter. The generator loads the weights of the best training available checkpoint, and then the shower and associated information are showed. The repository basical structure is:

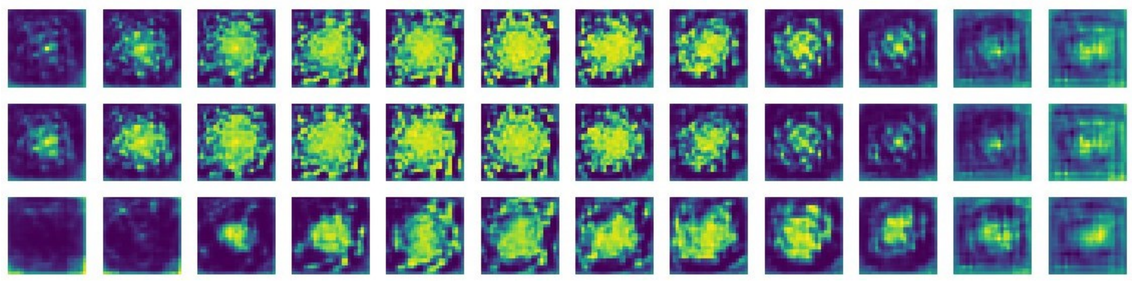
```
EM-shower-simulator-with-NN/
├ docs/
├ EM_shower_simulator/
│  ├ checkpoints/
│  ├ shower_simulator.py
│  ├ ...
├ ...
├ tests/
├ ...
├ ...
├ LICENSE
├ README.rst
├ requirements.txt
├ setup.cfg
├ setup.py
├ versioneer.py
```

Some examples

Geant4 simulations:



GAN simulations:



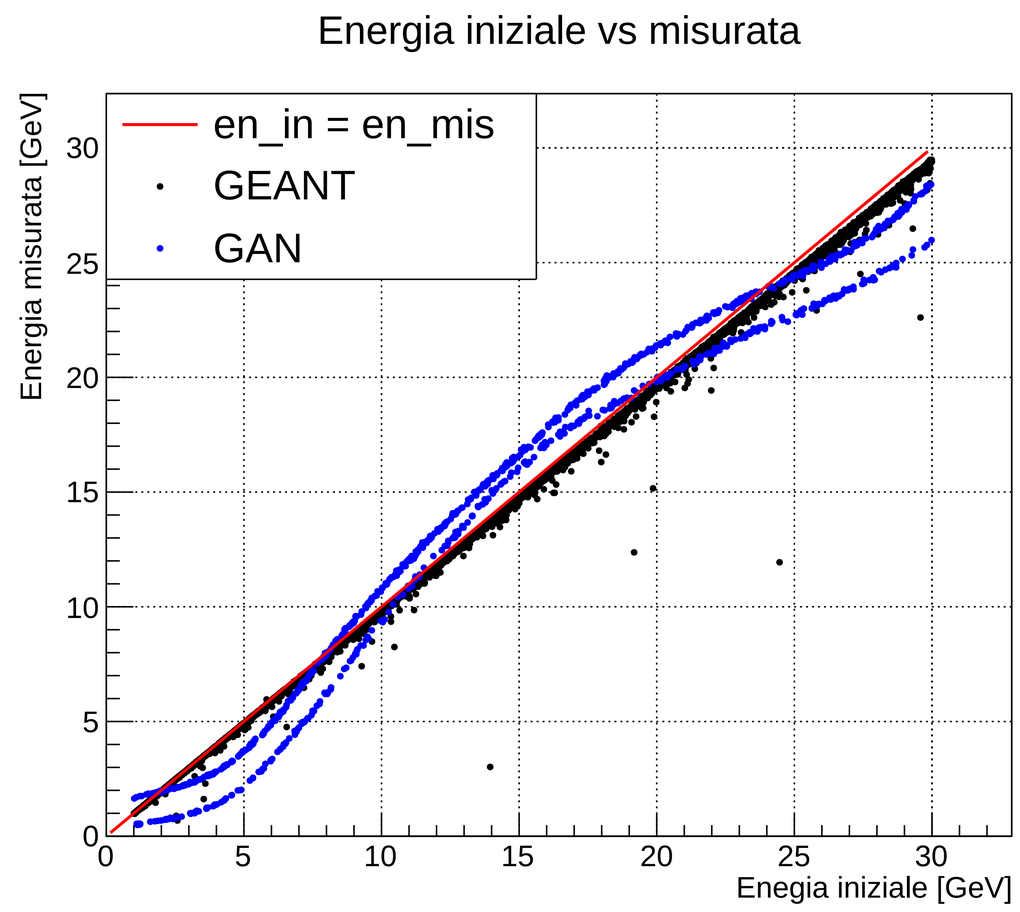


Figure 2. Comparison between GAN and GEANT simulations of deposited energy against primary particle energy

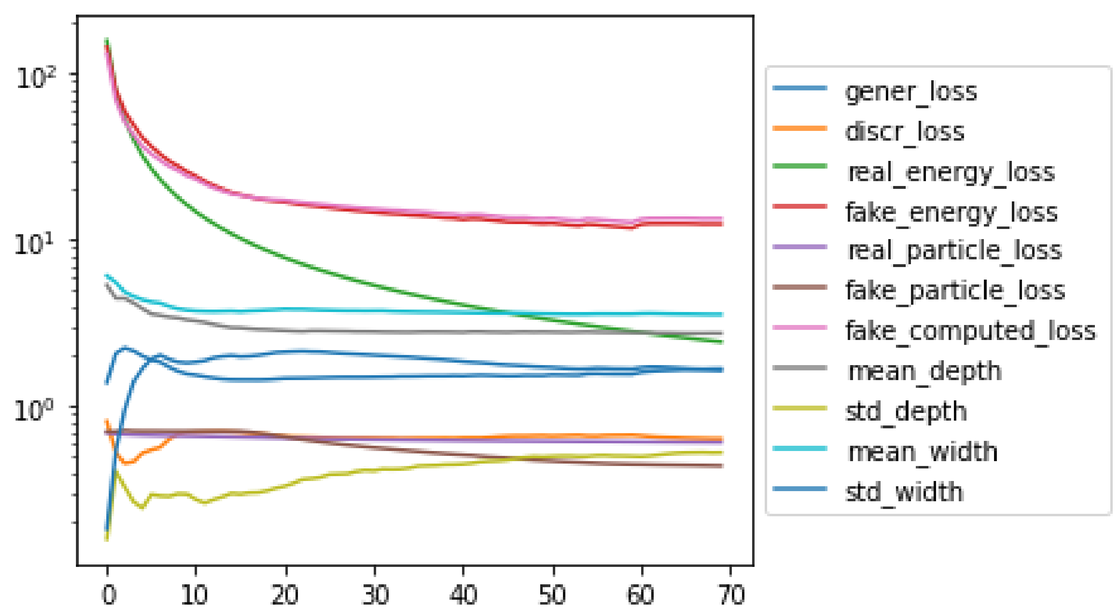


Figure 3. Evolution of losses and unbiased metrics for a training of 70 epochs

cafasso.dario@gmail.com d.passaro1@studenti.unipi.it