CERN Summer Student Programme Report
August 2023

Development and optimisation of inference for FastCaloGAN with ONNX and SOPHIE

Submitted by

David Attard

Supervised by

Dr. Michele Faucci Giannelli
Dr. Dalila Salamani

Contents

Chapter 1

Introduction

Located at the Large Hadron Collider (LHC), the ATLAS detector stands as the most extensive volume detector utilized in particle colliders, comprising the inner detector, calorimeter, magnet system, and muon spectrometer [2]. Due to the stochastic nature of quantum processes, the analyses of the data of the LHC experiments is carried out using sophisticated Monte Carlo simulation of the processes that occur when two protons collide. The products of these interactions are propagated into the ATLAS detector that is simulated with GEANT [6]. However, the largest fraction of the simulation time is consumed by the calorimeters. While GEANT4 serves as a highly precise toolkit for conducting a full detector simulation, it remains inherently time-consuming and challenging to substantially accelerate [6]. Until now, the available computing capabilities of the GRID allowed us to use GEANT to simulate a large fraction of all the processes needed by ATLAS. However, this will no longer be possible when LHC increases the luminosity in the coming years. Hence, there arises a pressing need for a rapid calorimeter simulation, such as FastCaloGAN, to generate a large number of simulated events that correspond to the escalating luminosity for the LHC physics program [3].

1.1 Calorimetry

A calorimeter is an essential tool for measuring the energy left behind by charged and neutral particles. The energy absorption within the detector's material is crucial to this measurement. Calorimeters are necessary instruments in experimental physics for measuring energy from photons, electrons, positrons, jets, and deducing transverse energy. This is dependent on cascading processes, which are started by incoming particles and produce secondary particles to form a shower. Energy is deposited throughout this cascade un-

til the calorimeter completely absorbs it. The energy of the incident particle and the number of particles in the cascade are directly correlated. Numerous mechanisms, including heat, ionization, atom excitation, Čerenkov light, and nuclear interactions, are involved in the absorption process with the goal being to measure a signal that is inversely correlated with the particle's energy [1].

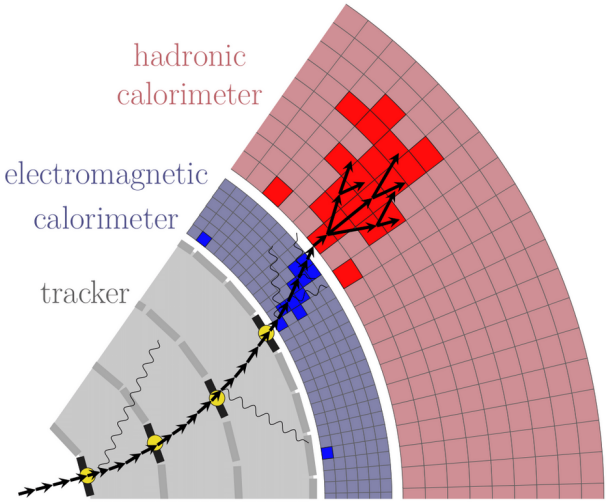

Figure 1.1: An illustration of the ATLAS detector cross-section showing the tracker, electromagnetic calorimeter, which is used to measure the energy deposited by electromagnetic showers developed from light electroweak particles (e.g. photons and electrons) and the hadronic calorimeter which is used to measure the energy deposited from hadrons (e.g. pions). The coloured blue and red squares represent the signal created due to an energy deposit. Image-credit: [1]

1.2 Simulations

1.2.1 Full Simulation

In order to be able to design detectors and perform physics analyses we require simulations. This allows us to compare our experimental measurements to the theoretical predictions. The Geant4 is a simulation toolkit used by large-scale experiments and projects including the ATLAS experiment [6]. It simulates particle interactions with matter using Monte Carlo techniques, where such simulations are referred to as detailed simulations or full sim-

ulations. This is because such simulations are considered the baseline for the model interactions. However, using such simulations comes with its own challenges. The Geant4 toolkit requires a lot of CPU and memory to model the in-depth and step-by-step processes. Additionally, because the entire simulation depends entirely on the detector, the simulation must change if any components of the detector are upgraded. This is why we need fast simulations [6].

1.2.2 Fast Simulation

As interactions of particles with detector material are often the slowest component of the full simulation, in a fast simulation approach, instead of simulating in detail the cascade of particles in the calorimeter, one can parametrize the energy deposition. All hits are created in a single step. The ATLAS detector employs two different fast simulations, FastCaloSimV2 and FastCaloGAN [3].

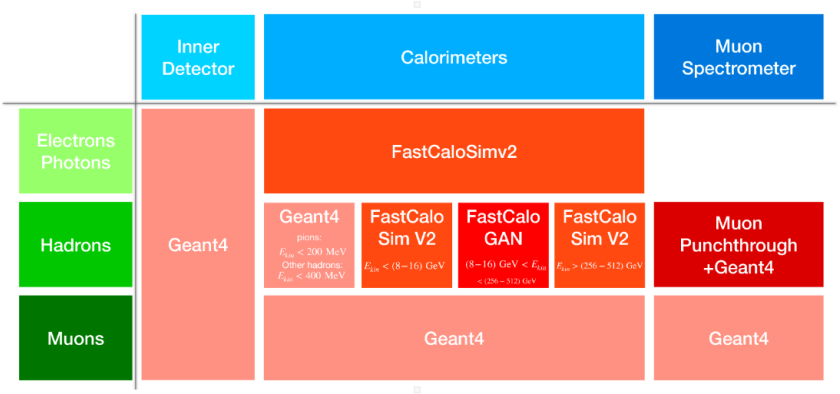

Figure 1.2: An illustration of which simulations are used to simulate different modules and events in the ATLAS detector. Image-credit: [3]

FastCaloSimV2 parameterizes the calorimeter's longitudinal and lateral development of showers. On the other hand, FastCaloGAN is an innovative approach that performs fast simulation using generative adversarial networks (GANs). GANs are machine learning models that can learn to generate realistic data distributions. In the context of particle physics, FastCaloGAN leverages GANs to generate approximate calorimeter response patterns based on the characteristics of input particles. This approach offers the advantage of faster simulations while retaining a high level of realism. FastCaloGAN can provide valuable insights into particle interactions and detector behaviour,

even when detailed simulations are impractical due to time or resource constraints.

Up until the current LHC run, Run3, the GAN models trained to generate showers in FastCaloGAN are implemented in the ATLAS Athena software framework using the Lightweight Tensor Neural Network (LWTNN) [7]. In this project, we test out two other inference libraries ONNX and SOFIE. First, the ONNX inference library and the associated optimization techniques are discussed [8]. Later we describe the implementation of the SOFIE inference library followed by a brief comparison of the three inference libraries [4].

Chapter 2

The new inference libraries

2.1 ONNX

Although the LWTNN inference library works quite well, the models produced are quite large in size ($\sim$ 37 MB) and also make use of relatively large amounts of memory. ONNX, which stands for Open Neural Network Exchange is another machine learning framework which can be used instead of LWTNN [8].

ONNX is an open-sourced artificial intelligence ecosystem that provides standards for representing a machine learning model and algorithms for their inference in a universally standardized way. Microsoft also developed ONNXRuntime which provides the environment required for the inference of an ONNX model of a particular machine learning method [5]. Although an ONNX implementation for FastCaloGAN has already been carried out in the past none of the models had yet been further optimized. Below I will describe the optimization process tested out and their respective performance.

2.1.1 ONNX Runtime

ONNX Runtime is a cross-platform machine-learning model accelerator. It serves as a runtime engine that accelerates the execution of ONNX models across a wide range of hardware and software configurations. The ONNX Runtime offers impressive speed and efficiency gains, allowing models to be deployed with minimal overhead, low latency, and improved performance. Three model optimization techniques were tested out in this project [5].

1. Quantization
2. Float16 model conversion

3. Graph Optimization

2.1.2 Quantization

The first optimization technique that was tested out was Quantization. ONNX Runtime employs quantization as a method to convert the values within an ONNX model to an 8-bit linear quantization format. During the quantization process, the original floating-point values are transformed into an 8-bit quantization space using the equation:

$$fp32\_value = scale * (val\_quantized - zero\_point) \tag{2.1}$$

In this equation, *scale* is a positive real number that serves to map the floating-point values into the quantization space. Additionally, *zero_point* represents the value zero within the quantization space. If the representation of zero is not distinct in the quantization process, it can lead to inaccuracies that impact the overall model accuracy.

Before a model can be quantized, pre-possessing of the float32 model has to be carried out which consists of symbolic shape inference, model optimization and ONNX shape inference. This is implemented using the *quant_pre_process* function found in the *onnxruntime.quantization* python package:

```
quant_pre_process(model_path, opt_path)
```

where *model_path* is the file location of the original ONNX model and the *opt_path* is the file path where the optimized model is saved.

Following the pre-processing of the ONNX model, quantization is carried out using the *quantize_dynamic* function found in the *onnxruntime.quantization* library, implemented as follows:

```
quantize_dynamic(opt_path, quant_path, weight_type =
    QuantType.QUInt8,   nodes_to_quantize=None)
```

where, *quant_path* is the file path where the quantized model will be saved and *weight_type = QuantType.QUInt8* specifies that the weights are saved as unsigned 8 bits.

When the model for pions at $20 < \eta < 25$ was quantized, the file size of the model was reduced by almost 3/4, where the original model was 5.1 MB in size whilst the new quantized model was only 1.3 MB. However, this resulted in the model becoming less accurate as shown in Figure 2.1. Hence we do not consider quantization any further in this project.

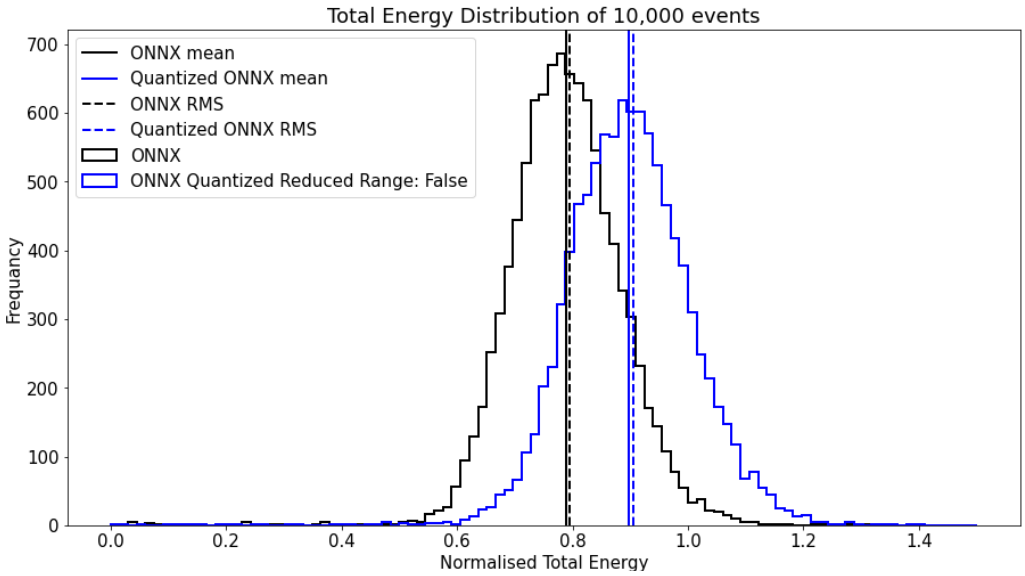

Figure 2.1: A plot comparing the distribution of the total energy from 10000 events using the original ONNX model and the quantized ONNX model.

2.1.3 Float16 Model Conversion

The second optimization technique that was considered was to convert our float32 ONNX models to a float16 model. Changing a model to utilize float16 instead of float32 has the potential to reduce the model's size by up to half and enhance performance on specific GPUs. While there might be a slight compromise in accuracy, for numerous models, the resultant accuracy remains satisfactory. Unlike quantization, float16 conversion doesn't require adjusting data, making it a preferable choice. The model conversion to float16 was implemented by making use of the *float16.convert_float_to_float16* function found in the *onnxconverter_common* library. The code implementation was as follows:

```
# load the original ONNX model
model = onnx.load("path/to/model.onnx")

# carry out the conversion
model_fp16 = float16.convert_float_to_float16(model)

# save the float16 model
onnx.save(model_fp16, "path/to/model_fp16.onnx")
```

By running an inference test on 10,000 generated events we compared the results obtained when using the original ONNX model and the float16 ONNX model in Figure 2.2. The events produced for comparison were carried out

for pions at 64 GeV and in etas, 20-25. In such events, it was observed that the distribution of the total energy from each event is comparable to that from the original model. Moreover, the float16 conversion reduced the file size of the model by approximately half, where the new float16 model was 2.6 MB in size as compared to the 5.1 MB original ONNX model. The new float16 model also made use of less memory for the session creation (look at section 2.3.1).

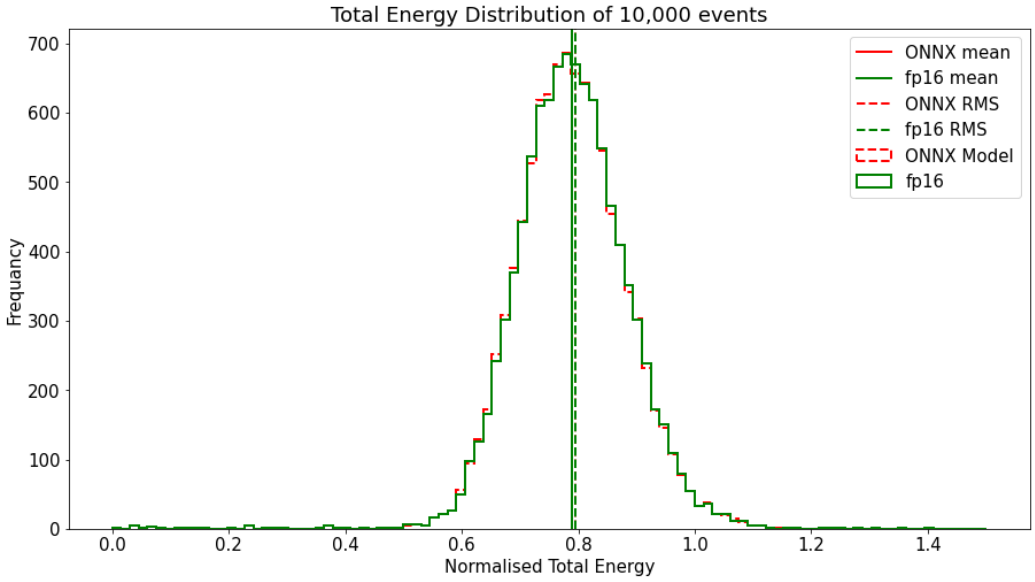

Figure 2.2: A plot comparing the distribution of the total energy from 10,000 events using the original ONNX model and the float16 ONNX model.

2.1.4 Graph Optimization

The third and final optimization test carried out on the ONNX model was graph optimization. Graph optimizations encompass a variety of transformations at the graph level, spanning from minor simplifications and the removal of nodes to more intricate processes like merging nodes and enhancing layout. Below is a graph optimization implementation in Python which makes use of the *onnx_runntime* package:

```
sess_options = onnx_runtime.SessionOptions()

# Set graph optimization level
sess_options.graph_optimization_level = onnx_runtime.
    GraphOptimizationLevel.ORT_ENABLE_EXTENDED
```

```
# set path to where the optimized model will be saved
sess_options.optimized_model_filepath = opt_model

# Creation of ONNX session
session = onnx_runtime.InferenceSession(opt_model,
    sess_options)
```

By carrying out inference on 10,000 events and looking at the distribution of the total energy from each event, we compare the accuracy of the graph-optimized model to the original ONNX model. Same as with the other tests, the events generated were for pions at 64 GeV and in etas, 20-25. As can be observed in Figure 2.3 the distributions of the original ONNX model and the graph-optimized ONNX model match perfectly and hence we can conclude that no accuracy was lost from this operation.

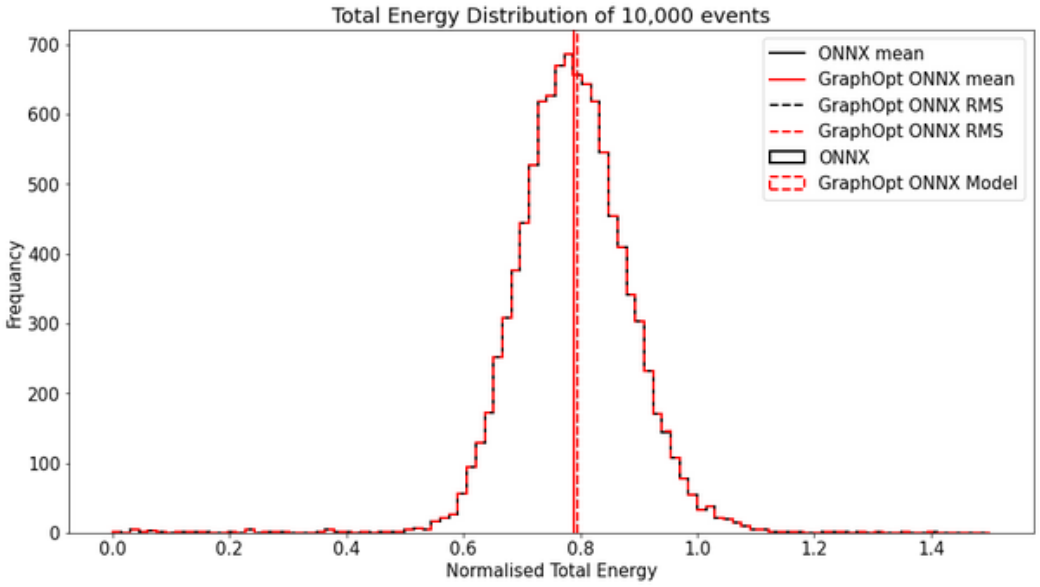

Figure 2.3: A plot comparing the distribution of the total energy from 10,000 events using the original ONNX model and the Graph Optimized ONNX model. The two distributions match perfectly.

2.2 SOFIE

Another inference library that was tested was SOFIE (System for Fast Inference code Emit) [4]. The two main advantages of using SOFIE are that it has minimal dependencies and also, currently the model files produced

by LWTNN and ONNX need to be stored in a string that is converted to the appropriate object in the TBuffer method. This is because fast simulations cannot access the disc (or else they would not be fast enough) during the simulation, thus all objects need to be stored in memory. In order to maximise efficiency, AF3 store the full detector parametrisation in a single ROOT file. However, since ROOT cannot store ONNX or LWTNN C++ objects, the current solution is to store the weights of the network (a .onnx or .json file) as a stringstream that is then converted into the C++ object when the parametrisation file is loaded at the start of a simulation job. The advantage of SOFIE is that it is ROOT native, so it would allow us to store the weights in the parametrisation file without the need to do the above-explained conversion process and hence SOFIE would simplify the long-term code maintenance.

The disadvantage with SOFIE is that the model conversion generates two files, one C++ header file and a .dat file in which the weights are stored. Hence, after the conversion an extra operation would have to be carried out to merge the two files as a single header file.

2.2.1 Model Conversion

The first task was to convert the generator tf.Keras models using SOFIE. This was done using the below Python implementation:

```
ROOT.TMVA.PyMethodBase.PyInitialize()

# path to the generator keras model
modelFile = 'keras_model.h5'

# Parse the generated Keras model into RModel object
model = ROOT.TMVA.Experimental.SOFIE.PyKeras.Parse(modelFile)

generatedHeaderFile = modelFile.replace(".h5",".hxx")

# Generate the .hxx and .dat files
model.Generate()
model.OutputGenerated(generatedHeaderFile)
```

In the above implementation we pass as input the file path of the generator keras model and we get back two files, a C++ header file (.hxx file) and a file with the weights stored (.dat file). When such conversion was carried out for a generator Keras model of pions at $20 < \eta < 25$ the .hxx file was 12.3 kB whilst the weights file was 16.8 MB. This means that files generated using SOFIE are more than half the size of the files produced by LWTNN,

but still significantly larger than the ONNX header files.

2.2.2 Validation with SOFIE

In order to check the consistency of the SOFIE-generated header files with LWTNN and ONNX, as well as to evaluate the performance of SOFIE, we created a stand-alone script which takes as input two tensors, one contains the noise and the other the input energy, and returns back the energy in each voxel for n events. First the session is created using:

```
session = ROOT.TMVA_SOFIE_checkpoint_pions_eta_20_25_All.
    Session(input_dat_file)
```

where *input_dat_file* is the file path to the .dat file which stores the weights. The inference is then carried out using the *infer* function,

```
result = session.infer(noise,conditions)
```

where *noise* is a tensor having dimensions [1, *ganParameters.latent_dim*] and is populated by random numbers having a mean and standard deviation of 0.5. Whereas, *conditions* is a tensor of the form [0, *input_energy*].

2.3 Comparison of Inference Libraries

Using the created C++ stand-alone script for LWTNN and the Python scripts for ONNX and LWTNN the values of energy in each voxel produced from inference were compared for pions and High and Low energy photons by inputting the same noise tensor into the model. These were found to match perfectly with each other, thus confirming that all three inference libraries can be used without any loss of information.

The stand-alone C++ code to run the inference can be found here. By changing the particle i.d., η, LWTNN file path and input energy one is able to generate a .csv file with the normalised energy values in each voxel.

Similarly, stand-alone Python scripts for ONNX and SOFIE were written to perform validation of the converted models. For easier use, the Python inference tests can be run using the *inference_production.sh* bash file.

2.3.1 Performance Testing

In this subsection the performance of the different ONNX optimized models and SOFIE are compared. It is important to note that the specific values

for these tests are machine-dependent, however, in order to obtain the fairest and most accurate comparison the below tests were all performed on the same lxplus node.

In Table 2.1 we compare the amount of memory used in order to generate the session and the inference of 10000 events. We observe that float16 converted ONNX model uses the least amount of memory for the session creation, with SOFIE using a up a similar but slightly larger amount of memory. On the other hand, the original ONNX model and the graph-optimized ONNX model were using the most memory. When looking at the amount of memory used for the inference it was found that the original ONNX model was using the least memory. However, memory usage is negligible when compared to the session creation and hence we conclude that the float16 ONNX model and SOFIE are the most efficient in terms of memory usage.

	ONNX	ONNX fp16	ONNX GraphOpt	ONNX fp16 +GraphOpt	SOFIE
Session Creation: Memory (MB)	17.496	8.223	29.45	13.63	10.88
Inference 10000 events: Memory (MB)	0.352	1.871	0.24	5.15	2.16
Session Creation: Time (s)	0.018	0.029	0.49	0.19	0.96
Inference 10000 events: Time (s)	2.3	19.36	2.1	18.85	19.92

Table 2.1: A table comparing the memory usage and the time taken for the different optimized ONNX models and SOFIE.

When looking at the time taken to create a session and then carry out the inference on 10000 events we find that the original ONNX model and the graph-optimized model take the least amount of time. The float16 ONNX model was found to take $\sim$ 19 s and this was attributed to the fact that the machine on which this computation was carried out was a 32-bit machine. Hence, further operations would need to be carried out to make the inference.

Chapter 3

Conclusion and Further Works

In this project the ONNX and SOFIE inference libraries were tested as an alternative to LWTNN. It was found that the models produced by ONNX are significantly smaller in size than LWTNN models and can even be further optimized through float16 model conversion to make the models run more efficiently whilst also preserving accuracy. Also, we managed to convert the generator Keras models using SOFIE to produce the model file (*.hxx*) and the weights file (*.dat*).

By creating stand-alone inference scripts for LWTNN, ONNX and SOFIE the three inference libraries validation testing was also carried out to check that they predict the same amount of energy in each voxel. This was a crucial part of the project as it resolved any ambiguities and confirmed that any of the three libraries can be used to produce identical results.

To conclude, future works should try to store the weights as ROOT objects as such functionality has only been made recently available. Moreover, the model file generated from SOFIE and the ROOT weights file will eventually have to be merged as a single header file in order to be later implemented into ATHENA.

References

[1] URL: https://g4fastsim.web.cern.ch/docs/intro/.

[2] G. Aad et al. "The ATLAS Experiment at the CERN Large Hadron Collider". In: *JINST* 3 (2008), S08003. DOI: 10.1088/1748-0221/3/08/S08003.

[3] G. Aad et al. "AtlFast3: The Next Generation of Fast Simulation in ATLAS". en. In: *Computing and Software for Big Science* 6.1 (Mar. 2022), p. 7. ISSN: 2510-2044. DOI: 10.1007/s41781-021-00079-7.

[4] Sitong An and Lorenzo Moneta. "C++ Code Generation for Fast Inference of Deep Learning Models in ROOT/TMVA". en. In: *EPJ Web of Conferences* 251 (2021). Ed. by C. Biscarat et al., p. 03040. ISSN: 2100-014X. DOI: 10.1051/epjconf/202125103040.

[5] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: x.y.z. 2021.

[6] S. Guatelli et al. "Introduction to the Geant4 Simulation toolkit". In: *AIP Conference Proceedings*. AIP, 2011. DOI: 10.1063/1.3576174. URL: https://doi.org/10.1063/1.3576174.

[7] Daniel Hay Guest et al. *lwtnn/lwtnn: Version 2.9*. June 2019. DOI: 10.5281/zenodo.3249317. URL: https://zenodo.org/record/3249317.

[8] Onnx. *Onnx/onnx: Open standard for machine learning interoperability*. URL: https://github.com/onnx/onnx.