



Electronic Communications of the EASST
Volume 86 (1) Year 2026

deRSE26 - Selected Contributions of the 5th Conference for Research Software Engineering in Germany (Part 1)

Edited by Bernd Flemisch, René Caspart, Bernadette Fritzsch, Florian Goth, Jean-Noël Grad, Florian Thiery

Quantization Effects of Artificial Neural Networks for Embedded Edge-Computing Applications

Alperen Aksoy, Ilja Bekman, Vesselin Dimitrov, Qader Dorosti, Chimezie Eguzo, Sarah Fleitmann, Fabian Hader, André Zambanini, Stefan van Waasen

DOI: 10.14279/eceasst.v86.1.2799

License: This article is licensed under a CC-BY 4.0 License.

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Quantization Effects of Artificial Neural Networks for Embedded Edge-Computing Applications

Alperen Aksoy [1], Ilja Bekman [1], Vesselin Dimitrov [1,2], Qader Dorosti [2], Chimezie Eguzo [1], Sarah Fleitmann [1], Fabian Hader [3], André Zambanini [1], Stefan van Waasen [1,4]

Peter Grünberg Institute (PGI), Integrated Computing Architectures (ICA | PGI-4), Forschungszentrum Jülich GmbH, Germany[1]
Center for Particle Physics Siegen, Department für Physik, Universität Siegen, Germany[2]
Faculty of Engineering, JARA-FIT Institute for Quantum Information, Forschungszentrum Jülich GmbH and RWTH Aachen University, Germany[3]
University Duisburg-Essen, Germany[4]

Abstract: This paper examines the use of Quantized Neural Networks (QNNs) for two resource-constrained scientific applications: automated calibration of semiconductor quantum bits (qubits) and scientific particle detectors. We evaluate the trade-offs between Post-Training Quantization (PTQ), Quantization-Aware Training (QAT), and ultra-low-bit Binary Neural Networks (BNNs) with respect to latency and resource usage. Our results demonstrate that PTQ and QAT are easily implementable solutions achieving a four-fold reduction in memory usage for U-shaped CNN (U-Net) architectures, whereas BNNs are better suitable for use cases with challenging latency requirements. For the training of non-differentiable custom BNNs, we propose a novel, hardware-constrained learning approach using Genetic Algorithms (GAs). We showcase a Look-Up Table (LUT)-based BNN architecture suitable for direct conversion to Very High-Speed Integrated Circuit Hardware Description Language (VHDL) via the HCL4BNN framework. This method achieves nanosecond-scale inference latencies at 10 ns to 15 ns without requiring specialized Digital Signal Processor (DSP) or Block RAM (BRAM) resources.

Keywords: Signal Processing, Triggering Systems, Binary Neural Networks, Neural Network Quantization, Quantum Computing, Edge Computing

Abbreviations

ADC Analog-to-Digital Converter

ANN Artificial Neural Network

BNN Binary Neural Network

BRAM Block RAM

CAM Content-Addressable Memory

CNN Convolutional Neural Network

DSP Digital Signal Processor

EAS Extensive Air Shower

FPGA Field Programmable Gate Array

eFPGA Embedded Field Programmable Gate Array

FF Flip-Flop

FP Floating Point

GA Genetic Algorithm

HDL Hardware Description Language

HLS High-Level Synthesis

HCL4BNN Hardware-Constrained Learning for Binary Neural Networks

LLM Large Language Model

LUT Look-Up Table

MAC Multiply-Accumulate

PTQ Post-Training Quantization

QAT Quantization-Aware Training

QNN Quantized Neural Network

ReLU Rectified Linear Unit

RFI Radio Frequency Interference

SiPM Silicon Photo-multiplier

U-Net U-shaped CNN

VHDL Very High-Speed Integrated Circuit Hardware Description Language

1 Introduction

Machine learning has proven to be an essential tool for the identification of patterns in complex datasets. A field, where this becomes increasingly important is experimental physics because it is dominated by measuring and processing data in large quantities. This leads to an increased use of machine learning algorithms for feature extraction. As of now, it is mostly used in large data centers for offline data processing [1], but thanks to recent advancements of hardware and algorithms, it can be used more and more for edge processing applications [2]. However, many deployment environments operate under severe energy, latency, and memory constraints [3, 4, 5, 6], requiring dedicated efforts for processing hardware.

Field Programmable Gate Arrays (FPGAs) (as well as embedded FPGAs or eFPGAs)[1] are frequently used for these edge computing tasks, since they offer low and deterministic latency and parallelism suitable to tackle the data rates of sensors in edge environments. As processing units like CPUs and GPUs optimized for floating point operations differ fundamentally from FPGAs, full-precision neural networks cannot be executed verbatim on such resource-constrained platforms.

To address this challenge, this work focuses on Quantized Neural Networks (QNNs), which reduce numerical precision to enable efficient edge computing while minimizing negative effects of the reduced dynamic range of values.

The choice of quantization strategy is heavily influenced by the specific constraints and processing demands of the application environment. Therefore, we demonstrate the selected basic quantization methods for neural networks and their implementation for two distinct domains:

Automated Qubit Tuning via PTQ and QAT The precise calibration of semiconductor spin qubits requires detecting charge transitions in charge stability diagrams like in Figure 5. In order to automate this task, we utilize U-Net architectures, a segmented Convolutional Neural Network (CNN) consisting of a contracting and an expansive part [3, 7]. The primary constraint in the physical setup is the limited cooling power within a cryostat, where heat dissipation from control electronics must be minimized. Ealier works show that U-Net-like neural networks executed on specialized hardware accelerators would be compatible with the power budget available at the millikelvin to 4 kelvin stages of cryostats used for qubit experiments [8]. We evaluate how Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) affect the model performance in terms of detection quality of charge transitions in charge stability diagrams for U-Net architectures with different numbers of parameters.

Real-Time Particle Detection via BNNs In high-energy physics, detectors search for rare phenomena within a huge number of irrelevant events, hence generating massive data volumes that require immediate processing. Even though the experiments differ fundamentally in their research question, the detection technologies and thus data processing needs are often comparable. In this work, we demonstrate data processing for the SHiP experiment at CERN [4]. Real-time filtering there requires inference latencies in the micro- to nanosecond range, while a simple amplitude threshold cannot reliably separate relevant pulses from distorted or overlapping ones, so

[1] Some of the hardware-specific terms are explained in the appendix in 7.3.

the trigger must make this decision itself. Consequently, we focus on ultra-low-bit Binary Neural Network (BNN), trained via nature-inspired genetic optimization and translate the hardware constrained network directly to FPGA fabric to achieve the necessary throughput and energy efficiency.

To further investigate performance in terms of complexity trade-offs at the limit of numerical precision, we systematically evaluate the impact of extreme 1-bit quantization on the classification accuracy of highly compressed CNNs. Utilizing the MNIST dataset [9] as a benchmark, we analyze models trained through conventional methods — distinct from the Genetic Algorithm (GA) approach used for hardware-native logic — to demonstrate that carefully designed micro-architectures have comparable predictive performance.

Paper structure: After an introduction to different paradigms of quantization of neural networks and a discussion of the used implementation methods, we present their application to mentioned examples. This paper serves two purposes: first, it demonstrates the use of established PTQ/QAT strategies to a resource-constrained qubit-tuning task; second, it proposes a new method with a genetic-algorithm-trained, LUT-native BNN framework for a use case where the first methods are not sufficient.

2 Quantization Strategies

This section summarizes the quantization strategies considered in this work and motivates their use in the two application domains mentioned above. Quantization maps high-precision floating-point values to lower-precision representations requiring fewer bits, ranging from 8-bit integers down to single-bit values. Depending on the bit width and model architecture, the resulting information loss may have little effect on task performance, or it may require adjusted training methods to recover accuracy.

2.1 Post-Training Quantization

PTQ serves as a strategy to convert a pre-trained floating-point model into a lower-precision representation without the requirement of retraining [10]. Within the field of experimental physics, particularly for the automated calibration of semiconductor spin qubits, PTQ provides a rapid and practical pathway for model compression. This technique enables significant reductions in memory usage while maintaining a segmentation accuracy that is comparable to full-precision models. In favorable cases, PTQ can preserve model performance making PTQ a useful first step for deploying QNNs in energy-restricted edge environments. In this work, PTQ is implemented directly in `PyTorch` [11, 12].

2.2 Quantization-Aware Training

QAT is a quantization strategy that simulates quantization effects during the training process itself by inserting fake quantization operators, like rounding, low resolution, or integer wrap-

ping, into the model. For implementation of QAT on top of `PyTorch` we use `QKeras` [13], a framework providing quantized layer versions for deep neural network models.

This approach is designed to enhance the robustness of the QNN against the precision loss typically associated with lower-bit representations [14].

2.3 Extreme Quantization: Binary Neural Networks

At extreme levels, BNNs constrain weights and activations to low-bit representations, thereby departing entirely from multi-level floating point training and inference. Although this generally requires a slightly larger network, it allows multiplications to be replaced by lightweight and fast logic operations and accumulations to become (unary) pop-counting procedures [15]. This makes them an ideal candidate for Look-Up Table (LUT)-based FPGA implementations. Multiple approaches in recent years have addressed the challenge of training these structures [16, 17, 18, 19], since standard gradient-based methods are hindered by the non-differentiability of the binary design.

Training with Genetic Algorithms We are proposing an evolutionary optimization with GAs for the training of LUT-based BNNs, bypassing the gradient requirement of standard backpropagation and making them directly applicable to binary-weight networks where gradients are undefined. GAs are nature-inspired optimization methods published in [20] that mimic the processes behind the biological evolution [21, 22].

Given a `population` of `individuals` (each representing a unique BNN) with a set of `genes` (weights), the individuals' `fitness`-es (BNN accuracy) are determined and the next `generation` (iteration) is constructed by `selecting` more successful individuals, `mutating` (randomly changing few weights) and `crossing-over` (swapping sections of weights) their `genes`. Each new generation is evaluated again and consists by construction of more and more "fit" individuals. The condition for loop breakout may be that an individual achieves the target accuracy or that a certain number of iterations has been reached.

With these three strategies established, the next section contrasts the first two and details how the third proposed BNNs architecture is mapped onto FPGA hardware.

3 Hardware Implementation and Deployment Flows

This section summarizes the hardware deployment flows used in this work. We first describe conventional FPGA workflows based on quantization and High-Level Synthesis (HLS), and then introduce the proposed hardware-constrained approach for directly mapping LUT-based BNNs to Very High-Speed Integrated Circuit Hardware Description Language (VHDL).

3.1 Conventional FPGA Workflows

Conventional workflows for QNN implementation in FPGAs typically develop from floating-point model design and training to quantization, optimization, and hardware synthesis as illustrated in Figure 1. The steps in detail are:

(a) Network definition in Floating Point (FP) representation, including input normalization;
(b) training via iterative back-propagation with weight adjustment using gradient-based optimization; quantization to map the FP values to FPGA-compatible fixed point or integer arithmetic; pruning and compression aiming to remove low impact connections and nodes;
(c) expressing the necessary QNN operations in synthesizable C++ and conversion to a hardware-description language (HLS), *e.g.* via `Vitis_HLS` [23];
(d) integration into the target FPGA device firmware.

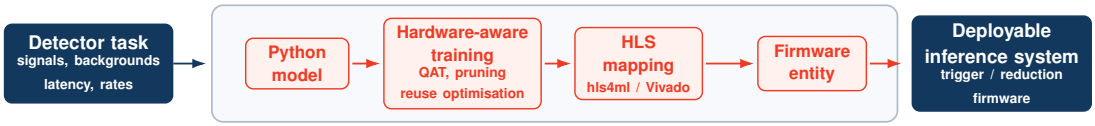


Figure 1: Conventional workflow to deploy a QNN on an FPGA based on gradient training, quantization, and HLS conversion.

During inference, the floating point preprocessing and Multiply-Accumulate (MAC)-heavy neuron evaluations can require significant latency and FPGAs resources, particularly Digital Signal Processors (DSPs) utilization. Furthermore, step d) depends strongly on the optimization quality of the HLS conversion tool-chain with the relevant tools being introduced here.

`hls4ml` [24] is an open-source framework that converts trained machine learning models into FPGA firmware using HLS. It is designed to deploy neural networks with low latency and low power consumption, especially for real-time applications.

`FINN` [25] is an open-source experimental framework from AMD/Xilinx for accelerating the inference of QNNs on FPGAs. It focuses on generating highly optimized FPGA implementations for low-precision networks such as binary and integer-quantized models.

`hls4ml` targets a broader set of machine-learning models and emphasizes simple, rapid conversion from frameworks such as `PyTorch` and `TensorFlow` [26] into FPGA implementations. `FINN`, on the other hand, specializes in low-bit quantized neural networks and generates customizable FPGA dataflow architectures.

In this work for BNN baseline comparison both QATs flows. First, we use the `Brevitas` library [27] with `PyTorch`, whose quantized models are exported to `FINN`. Second, we use `QKeras` [13] for quantized layers in models converted by `hls4ml`.

These flows provide reference implementations against which the proposed LUT-based BNN approach is compared.

3.2 Proposed Hardware-Constrained BNN Flow

3.2.1 Quantization Considerations

To achieve the least latency for the signal processing at the inference time, we are realizing our network using LUTs on an FPGA fabric, taking care to avoid DSP or Block RAM (BRAM) clocked structures and seeking to use combinatorial logic, parts of which can be executed in sub-clock speed of the standard FPGA fabric.

We propose a hardware constrained approach: to design the binary network using only FPGA-appropriate operations implementable with LUT-building blocks (*e.g.* Content-Addressable Mem-

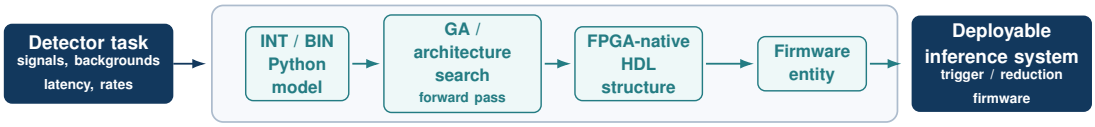


Figure 2: Proposed hardware-constrained learning approach with direct HDL generation.

ory (CAM) or adders), through routing (*e.g.* division by powers of 2, *i.e.* bit shifting), or other combinatorial logic (*e.g.* carry chains, multiplexers) that avoid clocked structures as in [16]. This constraint eliminates the quantization and conversion steps but requires training suitable for non-differentiable operations.

Rather than approximating conventional floating-point multiplication, the proposed architecture replaces arithmetic neuron operations with a constrained set of FPGA-native logical transformations that are optimized for low-resource inference.

Unlike conventional 1-bit binary weights, which can only encode a sign flip (*e.g.*, {+1,-1}), our synapse model requires 2 bits per weight for four distinct operations: blocking, passing through unchanged, increasing, and negating the input. This choice preserves the coarse, non-arithmetic character of binary-style weights while adding the "Block" state needed for the sparsity objective of the genetic algorithm (Sec. 3.2.2) and the "Pass" state that avoids unnecessary sign inversion.

The same reasoning extends to the choice of 2-bit neuron values (rather than 1-bit): four activation levels allow a neuron to be selectively increased or decreased rather than only switched on/off, while remaining small enough for efficient LUT mapping, although higher bit widths are usable as well and are used for multi-bit input for example (see below).

- $w = 0 \Rightarrow$ `Block`: Blocking operation, output is set to 0 regardless of input, which auto-prunes this synapse (also see 3.2.2).
- $w = 1 \Rightarrow$ `Pass`: Passing the input value through unchanged.
- $w = 2 \Rightarrow$ `Incr.`: Increasing the input value, settling on binary shift left ($<< 1$) with saturation safeguard (limiting to respective `MAX_INT`).
- $w = 3 \Rightarrow$ `Neg.`: Negation of the value, which is represented by bit-wise inversion, avoiding classical $+1$ correction for the implementation efficiency.

This allows a multiplication operation to be replaced by a very hardware-efficient LUT-operation in a 4×4 CAM, illustrated in the following Eq. 1, fitting efficiently into a single standard LUT4/LUT6 [28] primitive:

2-bit input				weight		2-bit output			
0	1	2	3	$w = 0$	Block	0	0	0	0
0	1	2	3	$w = 1$	Pass	0	1	2	3
0	1	2	3	$w = 2$	Incr.	1	2	3	3
0	1	2	3	$w = 3$	Neg.	3	2	1	0

(1)

While this is viable for the hidden layers, the input layer in edge applications is often connected to multi-bit sensor outputs, *e.g.* 12-bit Analog-to-Digital Converter (ADC) samples, so it is most useful to use integer values. The network input is reduced without normalization from 12 bit to 7 bit, which is a trivial operation in an FPGA.

We adapt the activation function to represent one of the four operations at inference time:

INT input	weight		INT output
v	$w = 0$	Block	0
v	$w = 1$	Pass	v
v	$w = 2$	Incr.	$\min(v << 1, \text{MAX_INT})$
v	$w = 3$	Neg.	$\sim v$

(2)

An integer summation of N weighted inputs per neuron is performed, allowing for the sum bit width of at least $\log_2(3N)$ to prevent an overflow. This step introduces a long carry chain, which is depending on the layout and is optimized in hardware by pairwise tree addition, *e.g.* $((1+2)+(3+4))$. This reduces the carry chain length logarithmically while maintaining a fully combinatorial implementation without pipeline stages. The network structure is constrained to 2^n neurons per layer for efficient summation.

We use an activation function inspired by a Rectified Linear Unit (ReLU) [29], which maps the integer sum back to 2-bit neuron values using three thresholds to separate the sum into four bins. As our network is not considering biasing, the thresholds are calculated during training based on the number of inputs to the neuron, as more inputs yield higher accumulation values. During training, some of the weights to these inputs might be set to zero (`Block`), effectively deactivating this input. Hence, this needs to be taken care for the threshold values as well. This way, the full range of the activation is available also to neurons, which inputs turn out to be heavily pruned.

The ReLU-like quantized ”staircase” activation function uses the discrimination with three fixed thresholds. Those are set at training time, depend on number of non-zero weights going into the neuron, and act directly on integer sums to avoid normalization step at runtime.

To give an example of the activation function for four non-zero neurons, we need to divide the maximum sum of $3 \cdot 4$ into 4 output values 0, 1, 2, 3 via inclusive thresholds: 2, 6, 10.

The final network output is threshold encoded, with neuron values 0 and 1 as ”off” and 2 and 3 as ”on”. The proposed approach assumes that the target classification tasks remain separable under ultra-low-bit representations and coarse logical activation transformations.

3.2.2 Training with Genetic Algorithms

To implement the GA, we have used the `deap` Python package [30] and its `eaSimple` procedure is derived to include `elitism`, which transfers the n best individuals unchanged to the next generation. This accommodates for the stochastic dips of the noisy fitness function described in more detail in the following.

Fitness Evaluation For training, reference data is required. In the case of the Silicon Photomultiplier (SiPM) readout in the application described in section 4.3.2, we use empirical double exponential functions. All SiPM training and evaluation data was generated by `HCL4BNN` [31] using the `SiPMDataset` Python class. The relevant distinction criterion is clean waveform based on a single input pulse (”good”) versus distorted waveforms with multiple inputs (”ugly”), as depicted in Figure 3. Alternatively, dedicated simulation frameworks may also be used [32].

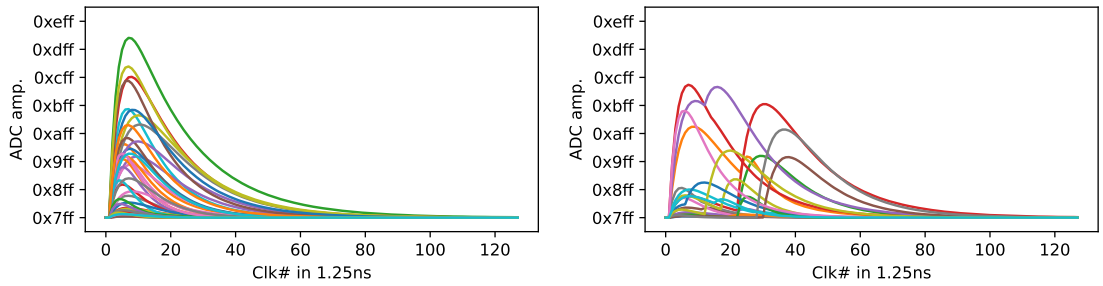


Figure 3: Left: isolated simulated SiPM pulses (”Good” class). Right: two overlapping or distorted simulated SiPM pulses (”Ugly” class).

For each individual a set of typically 300 ”good” and 300 ”ugly” waveforms are generated. The BNN predicts tuples: a two-element binary class vector with $(1,0)$ for ”good” and $(0,1)$ for ”ugly”. Due to both outputs being calculated for themselves, a consequential classification of neither ”good” nor ”ugly” would be encoded as $(1,1)$ or $(0,0)$, respectively. This can be used to indicate an abstention. Since the training set is regenerated for each evaluation, the fitness exhibits erratic behavior. The impact is discussed further in 4.3.3 together with the results.

The accuracy score measures how well the predicted tuples match the target tuples, assigning credits for partially correct predictions, as shown in Eq. 3 for tuples (left) and MNIST targets (right). It is then normalized by the tuple length and size of the training set. To avoid trivial ”broken clock” classifiers from achieving 50 % accuracy by always predicting the same class, goodness is set to 0 % whenever all predictions are identical. Otherwise, training stagnates, since complex models in early stages cannot out-compete the trivial variants in the same generation prior to further optimization and are selected for reproduction less frequently.

$$
\begin{array}{lll}
\text{for tuple target} = (1,0) & & \text{with digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9} \\
(1,0) & \hat{=} \quad 1.0 & \text{for MNIST target} = (0,0,0,0,0,0,0,0,0,1) \\
(1,1) & \hat{=} \quad 0.5 & (0,0,0,0,0,0,0,0,0,1) \quad \hat{=} \quad 10/10 \\
(0,0) & \hat{=} \quad 0.5 & (0,0,0,0,0,0,1,0,0,1) \quad \hat{=} \quad 9/10 \\
(0,1) & \hat{=} \quad 0.0 &
\end{array}
\tag{3}
$$

When calculating the fitness of an individual f we see an opportunity to combine two objectives of higher accuracy a and more sparse i.e. smaller network, measured by the fraction of zero-weights w_{zero}. Rather than performing a Pareto-based multi-objective optimization, we use a weighted sum of these two criteria with scales of 10 and 1 respectively.

The resulting composite fitness can be maximized within `deap` [33] as:

$$f = a \cdot 10 + \frac{w_z}{w_{\text{tot}}} \cdot 1 \tag{4}$$

The relative scales of 10 and 1 were chosen to reflect the typically logarithmic shape of the accuracy improvement over generations (see Figure 4 on the right): accuracy tends to increase rapidly in early generations before entering a slow, near-stagnant phase. Weighting accuracy an order of magnitude above the sparsity term ensures that the composite fitness remains dominated

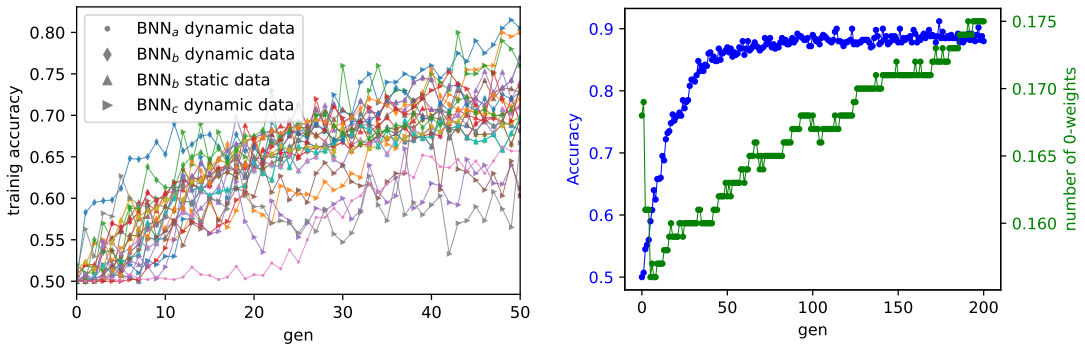


Figure 4: Left: Best training accuracy progression across generations for several GA runs, see Table 3 for validation results. Right: Progression of best accuracy and network sparsity (measured by the number of zero-weights) across generations for a longer GA run.

by a and the search is not diverted toward smaller but less accurate individuals. In the later slow-growth phase, individuals of comparable accuracy are then further differentiated by the sparsity term, favoring smaller networks among near-equal performers. The specific values were not individually tuned; a decade of separation was chosen to establish this priority ordering robustly rather than to optimize the trade-off precisely. The training is typically performed using populations above 300 individuals or even 1000 individuals if computation allows.

3.2.3 Hardware Description Language Conversion and Inference

Our Python code generates directly usable VHDL `entity` code from the weights (LUT-constants) and sum threshold constants, using the results of the training. A custom VHDL `package` supports this step by defining functions with operator overloading, making the Hardware Description Language (HDL) from Python output easier to read.

The FPGA implementation software can optimize across layers and neurons so that further compression of up to 20 % of the network footprint can be achieved without functional changes, *e.g.* via LUT packing [34]. Optionally, the VHDL `attribute KEEP` [35] can be used to keep the layers separated for ease of debugging.

In contrast to the related approaches `LUTNet` [17] and `LogicNets` [18], which also map trained network components directly onto FPGAs, our approach does not rely on gradient-based training and differentiable synapse behavior. This makes our GA-based approach applicable to explicitly hardware-mapped, non-differentiable structures without needing a differentiable surrogate for them. The resulting training pipeline becomes simpler and does not depend on backpropagation infrastructure like `PyTorch` or `TensorFlow`.

Having established the quantization paths, the next section applies them to two use cases from experimental physics with varying constraints.

4 Domain-Specific Evaluation and Results

This section evaluates the quantization strategies introduced above in two application domains.

4.1 Motivation for Quantization

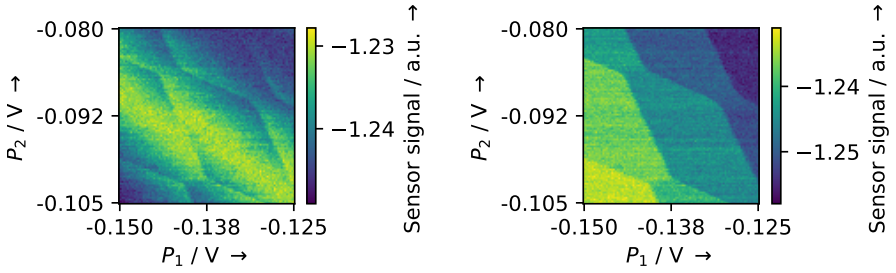
The first case study on segmentation for automated qubit tuning focuses on an energy-efficient implementation of a neural network. When porting to FPGA hardware, this directly leads to memory-efficiency. As the latency requirements are secondary, only conventional PTQ and QAT are investigated.

The second case study on signal classification for particle detectors requires ultra-low-latency data processing as new data is continuously taken with high rates and proceeding processing steps need to be activated. Thus, the more agressive BNN inference is pursued.

4.2 Memory-Efficient Qubit Tuning using PTQ and QAT Analysis

Our research on the trade-offs between quantization strategies and detection quality evaluated different U-Net architectures with varying parameter counts: the compact UNet-38k and the extremely lightweight UNet-447, both architectures described in [8].

The inputs for the U-Net are so called charge stability diagrams similar to Figure 5, where a charge sensor signal is shown in dependency on the gate voltages of the qubit structure. The processing task with these diagrams is to identify transitions between regions, *i.e.* identifying the edges. To be able to use lightweight neural networks for the detection of the charge transition lines in these diagrams, the unwanted response of the sensor to the change of the applied voltages is first compensated like shown in Figure 5b. In the following, compensated charge stability diagrams are used and simulated with the help of SimCATS [36].



(a) Without sensor compensation (b) With sensor compensation

Figure 5: Simulated charge stability diagram with and without compensation of the sensor response to the voltage changes for gates P_1 and P_2 [8]. Further data examples can be found in [37].

The results presented in Table 1 are achieved with the help of the quantization module of `PyTorch` following the tutorial in [38]. A description of the training process and the used hyperparameters is available in appendix 7.1. The loss function used for training is the sum of the

binary cross entropy [39] and the dice similarity coefficient [40], which balances overall overlap of ground truth and predicted mask with pixel-wise accuracy. During training a dataset with 10000 randomly sampled SimCATS parameter configurations and 100 charge stability diagrams per configuration is used [41]. The performance evaluation is done with the help of a separate test set which was generated by using 1000 randomly sampled SimCATS parameter configurations and the dice similarity coefficient as accuracy metric.
The results demonstrate that PTQ serves as a rapid and practical pathway for model compression, achieving significant memory reduction while maintaining segmentation accuracy comparable to full-precision models. The results achieved for QAT are only marginally better when looking at the mean dice similarity coefficient. However, differences can be seen in the standard deviation, where QAT achieves a similar standard deviation as the unquantized models and PTQ has a slightly higher standard deviation. Performance differences for PTQ and QAT between the two U-Net models could not be identified.

By reducing the numerical precision of weights and activations from 32-bit floating-point to 8-bit integers, we can reduce memory consumption 4× and computational cost 16× when the models are executed on appropriate hardware, for instance FPGAs.

Model	Unquantized in %	PTQ in %	QAT in %
UNet-447	85 ± 28	83 ± 29	85 ± 28
UNet-38k	96 ± 9	96 ± 9	96 ± 9

Table 1: Mean dice similarity on the test set with standard deviation on the test set.

4.3 Ultra-Low-Bit BNN Evaluation for Particle Detector Signal Classification

4.3.1 Preliminary MNIST Benchmark

We used MNIST as a compact benchmark to test the performance vs. complexity trade-off of highly compressed binary networks, before applying the BNN to the detector signals.

This benchmark is not intended to model particle-detector data, but to provide a simple reference for how much classification accuracy can be retained with very small binary architectures. To systematically evaluate performance-complexity trade-offs, we trained highly compressed CNNs with binary weights and activations on the MNIST dataset [9] containing hand-written digits. While a better benchmark would be another time-series dataset, MNIST offers well-known accuracy target.

Models with as few as 1013 binary parameters achieved an accuracy above 70 % on CPU for the classification of the MNIST-digits, illustrating that carefully designed micro-architectures can preserve performance even at extreme compression levels (see Table 2 and Figure 6). Because some training runs may get stuck in local minima, we use an accuracy threshold to identify runs that should be restarted. Specifically, a run is considered to have become stuck if its accuracy is below 70 % after 10 epochs. Without the knowledge of absolute reachable accuracy for a given model a near-normal distribution of converged runs is considered as estimate and so not the highest accuracy but the mean over converged runs' accuracies is taken. The network with

the higher number of parameters is able to achieve higher accuracy, although on average a larger spread indicates difficulty to converge to its global optimum. Study of convergence behavior and especially comparison to the genetic algorithm has to be performed at a future point.

# Parameters	9,112	4,268	3,147	2,062	1,013
Accuracy in %	88.7 ± 3.7	89.1 ± 1.2	87.5 ± 1.7	85.3 ± 1	78.3 ± 1.2
Non-convergent	18 %	29 %	20 %	15 %	18 %

Table 2: Results of compressed CNNs on MNIST dataset with small amount of parameters, binary weights and activations. Accuracy mean with standard deviation over normal fraction of runs is given in percent for converged runs. A fraction of training runs do not reach accuracy of 70 % after 10 epochs and are defined as stuck/non-convergent.

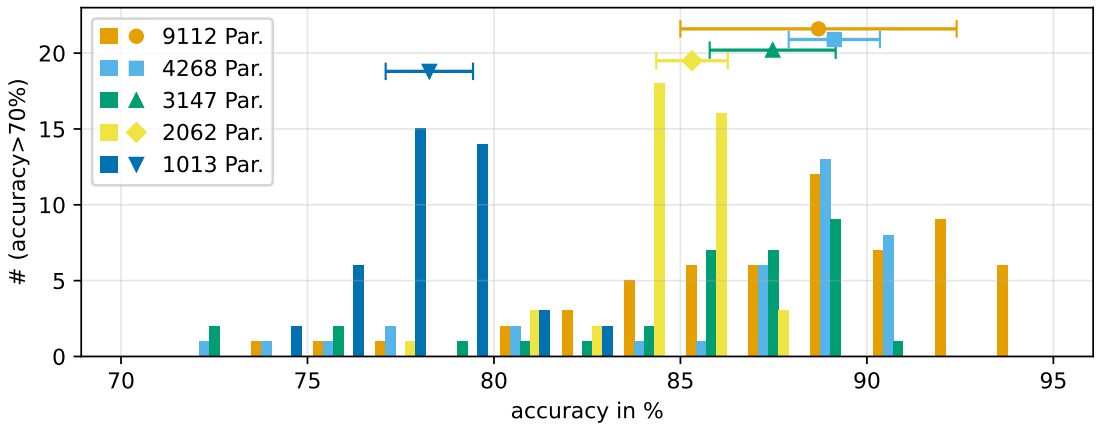


Figure 6: Illustration of the accuracy distributions of 5 compressed CNNs (next to each other per accuracy bin) for training reruns on MNIST dataset after 10 epochs. Largest CNN (orange color, circle) distribution has the mean at 89%, Smallest CNN (purple color, down-triangle) peaks at 78%. Means and standard deviation points are calculated over the normal part of the distributions.

The following investigations into detector triggering are also based on this extreme compression approach, but are evaluated using application-specific waveform data rather than image classification.

4.3.2 SiPM Pulse Classification

The Search for Hidden Particles (SHiP) experiment aims to detect particles interacting feebly with ordinary matter [42]. In some of the experiment's detectors, data is generated by photosensitive Silicon Photo-multiplier (SiPM) that convert single photons into small electrical signals. Real-time filtering of digitized SiPM signals is essential to reduce transmitted data volume while preserving relevant detector signals. To this end, we implemented `FINN`-based 2D CNNs, `hls4ml` CNNs as conventional baselines, and two LUT-based 2-bit BNN variants.

All models are targeting a ZCU104 FPGA with 230k LUTs, 640k Flip-Flops (FFs), 1,728 DSPs, 624 (18k)BRAMs [43].

In our case, the `FINN` workflow uses the `Brevitas` framework to quantize CNN. The FINN model is a 3-layer 1D-style QuantConv2d network (channels 1-4-6-8, kernel 5×1, padding 2×0, stride 1, 8-bit weights/activations, bias enabled) followed by an 8-bit-activation QuantLinear classifier ($1024 \rightarrow 2$, 8-bit weights), totaling 2452 parameters. The input signal has a frame length of 128 samples, with 12 bits per sample.

The `hls4ml` workflow used the same dataset and model structure but a true 1D convolutional (Conv1d) layers trained in `PyTorch`. Precision is fixed to 8-bit only at HLS-conversion time via the `hls4ml` configuration.

The trained model was converted into HLS, which was then synthesized into FPGA module using the `Vitis_HLS` tool. The generated module is subsequently integrated and verified using a testbench for functional validation and an example design for timing validation. The resulting inference latency achievable with QAT or PTQ techniques was higher than few μs and would be limiting the throughput of the design without further parallelization.

To overcome these limitations, we implemented BNNs trained via GAs to navigate the non-differentiable search space inherent in ultra-low-bit structures. In this architecture, weights and activations are constrained to 2-bit representations, allowing complex multiplication operations to be replaced by efficient LUT operations within a 4×4 CAM. The training process utilizes the `eaSimple` procedure with elitism, evolving a population of neural networks through mutation and crossover to maximize a fitness function derived from simulated air-shower pulses and measured noise traces. This evolutionary approach further optimizes the network by while maximizing accuracy also simultaneously increasing network sparsity with lower priority.

For deployment, the resulting models are synthesized through the HCL4BNN framework, which maps the logic directly to FPGA fabric using asynchronous combinatorial primitives such as LUTs, carry chains and multiplexers. By avoiding clocked DSP and BRAM operations, this hardware-constrained learning approach achieves nanosecond-scale inference latencies while significantly reducing the hardware footprint.

Three alternative BNN models are tried: BNN_a, BNN_b, and BNN_c with $[16, 64]$, $[32, 32]$, and $[64, 128]$ neurons in two hidden layers respectively, leaving the input (128) and output (2) widths

Model	Accuracy in %	Latency in ns	LUTs $\times 10^3$	FFs $\times 10^3$	DSPs	BRAMs 18 k	Training Time min $\times$ cores
`FINN`	74 ± 4	24850	30	20	106	5	$\sim 2 \times 16$
`hls4ml`	93 ± 2	3050	186	112	556	120	$\sim 2 \times 16$
BNN_a	63 ± 3	15	58	1.5	0	0	$\sim 300 \times 90$
BNN_b	74 ± 5	10	23	1.5	0	0	$\sim 105 \times 90$
BNN_b (static)	72 ± 3						
BNN_c	64 ± 8	10	18	1.5	0	0	$\sim 60 \times 90$

Table 3: Results and comparison of `FINN` 2DCNN ((128-4-6-8-2) , Kernel [5,1], padding 2.0, int8) implementation, `hls4ml` CNN, and BNN_a (128-64-128-2), BNN_b (128-32-32-2), BNN_c (128-16-64-2) with INT7 input quantization. BNN_b is also run with a static training dataset.

		Predicted	
		Good	**Ugly**
True	**Good**	947	53
	Ugly	410	590

		Predicted	
		Good	**Ugly**
True	**Good**	984	16
	Ugly	86	914

Table 4: Examples of confusion matrices for `FINN` (left) and `hls4ml` (right) for small validation sample. For the accuracies averaged over 30 runs see 3.

		Predicted		
		Good	**Ugly**	**Either**
True	**Good**	3052	1948	0
	Ugly	1595	3400	5
	Noise	0	0	5000

		Predicted		
		Good	**Ugly**	**Either**
True	**Good**	4883	117	0
	Ugly	1533	3430	37
	Noise	0	4999	1

Table 5: Example confusion matrices for one BNN_a 128-64-128-2 (left) and one BNN_b 128-32-32-2 (right); Either (or undecided) encodes $(1,1)$ or $(0,0)$ tuples for prediction and random input value test for truth.

same. With this a short-hand for architecture, *e.g.* (128-16-32-2) is used in captions.

BNN_a contains 4.06 kB of tunable parameters, BNN_b 1.27 kB, and BNN_c 0.78 kB. The `FINN` model results in 2.4 kB parameter space, while the `hls4ml` one has 2.39 kB. Since BNN weights encode combinatorial logic operations rather than multiplicative coefficients, a comparison based on matching layer topology or channel width across the three implementations would not yield a fairer comparison. We therefore match the comparison on task, dataset, and overall parameter budget rather than architecture, and report resource and latency figures (Table 3) for judging the trade-off directly. A resource-matched (*e.g.*, iso-LUT or iso-latency) comparison across paradigms is left for future work.

Since an abstention prediction is explicit for BNN an out-of-distribution (OOD) detection test is preformed, where a random vector of input size and bit-width is evaluated by the network.

In about 74 % of trainings the noise is predicted mostly as "ugly" (with less than 0.1 % misclassified as "good" and 0.4 % as "either"), about 22 % mostly as "either" (with less than 0.1 % misclassified as "good" and 0.8 % as "ugly"). Although, this is not evenly distributed: most networks produced no false positives at all, while a small number of runs accounted for the majority of the (still rare) misclassifications. In 4 % of cases noise is classified as "good" with similar purity. Such runs should be discarded even if they have similar accuracy of predicting the main "good" and "ugly" classifiers. It is thinkable to add noise samples to the training to obtain more robust networks, and should be studied further. Example confusion matrices from a validation dataset are shown in Tables 4 and 5.

The LUT-based BNN achieves low inference latencies around 10 ns to 15 ns while requiring no DSPs or BRAMs (see Eq. 3).

Although classification accuracy is reduced compared to `hls4ml` implementations, the resource efficiency makes BNNs attractive for first-stage filtering under strict power budgets.

4.3.3 Static vs. Regenerated Training Datasets for GA

To assess the effect of regenerating the training set at each fitness evaluation, we performed a dedicated control run for BNN_b using a single, seed-controlled, fixed set of 200 waveforms per class ("good" and "ugly") throughout training, rather than resampling a new set each generation. The result is added to the Table 3, showing similar performance within uncertainty. This is consistent with the expected effect of training on a fixed dataset: without regeneration, the GA can partially over-fit to the specific set of training waveforms, whereas continual resampling acts akin to unlimited data augmentation and reduces this risk. Essentially the accuracy noise for the latter most strongly depends on the size of the set (smaller for training, larger for validation). An effect is visible in Figure 7 where the network seeing validation data and dynamically regenerated training data shows similar accuracy (right), while for static training data the over-fitting is visible as discrepancy between the two (left). For the GA optimization, where the inference of multiple individuals dominates the computation time, time needed for the regeneration is negligible, while for backpropagation-based optimization the data regeneration has a significant impact on the overall processing time.

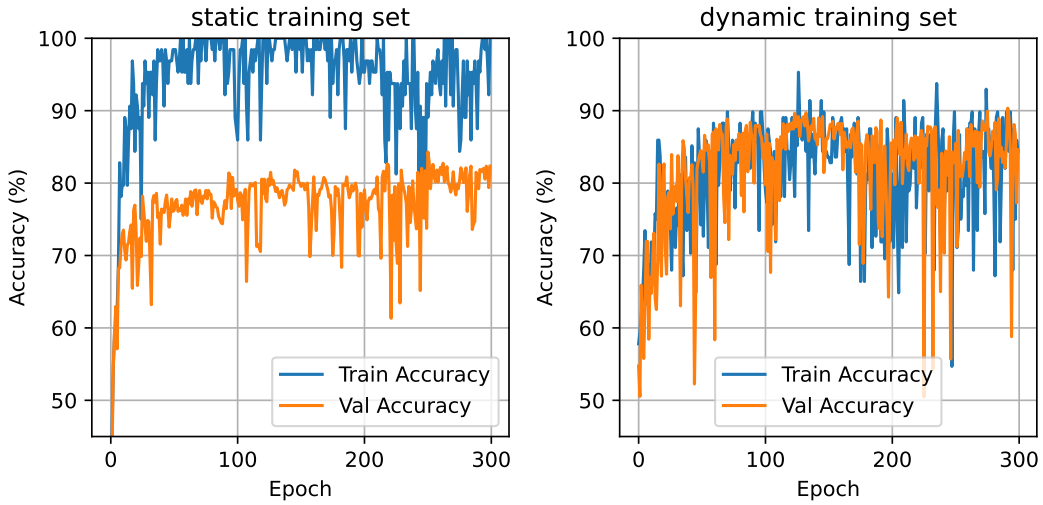


Figure 7: Illustration of the difference for static (left) and dynamic (right) SiPM training datasets of large (21k 4bit parameters) compressed CNNs. The training dataset here is 32+32 waveforms and 640+640 for validation. Static training data leads to exaggerated over-fitting in this illustration.

5 Discussion of Quantization Trade-offs

Our results confirm that quantization is a key enabler for the deployment of neural networks within energy-restricted scientific edge environments, such as cryostats for quantum computing or distributed detector arrays. The investigation across different architectures and scientific domains leads to the following key conclusions:

1. **PTQ** provides a rapid and practical pathway for compressing existing models with minimal implementation effort. By reducing precision from 32-bit floating-point to 8-bit integers, we achieve a four-fold reduction in memory usage while maintaining detection quality.

2. **QAT** requires additional training in contrast to PTQ and, thus, achieves slightly better results for the evaluated use case. For small neural network architectures, which are easy to train, this can be beneficial. However, for large network architectures with time-consuming training, PTQ might be easier to apply and achieves comparable results when the quantization aims at 8-bit integers. A more aggressive quantization might yield a stronger benefit from QAT but this has not yet been investigated.

3. **BNNs** offer an unmatched hardware efficiency, particularly for FPGA-based inference. By constraining weights and activations to 1–2 bits, computationally intensive MAC operations are replaced by efficient LUT operations. This allows for nanosecond-scale latencies in the range of around 10 ns to 15 ns and the elimination of specialized DSP or BRAM blocks, which is critical for the first-level triggers in experiments like SHiP.

Several techniques are still to be investigated in order to improve the efficiency of the BNN approach. Most prominently, frame data, *i.e.* short sequences, is typically stored in BRAM structures, which have limited port width for the retrieval. On the other hand, a typical neural network input is expecting the whole dataset to perform a classification. This creates an access bottleneck and implementing a segment-wise evaluation with several simpler networks allows for faster classifications, when a decision can be made before the the entire frame has been received.

Moreover, GA-based training of BNNs could incorporate a history-informed mutation bias for a fraction of operations: by tracking whether increasing or decreasing a given weight's operation code has previously correlated with fitness improvements across the population, mutation could be biased toward the empirically favorable direction rather than applied uniformly at random. This is conceptually related to estimation-of-distribution or self-adaptive mutation strategies, and could smooth the GA's exploration behavior. This remains speculative and untested at present, but the option only exists because of the extra state granularity of the 2-bit encoding, unlike a 1-bit weight which has no intermediate state to traverse.

Importantly, accuracy degradation is application-dependent and is often acceptable for early-stage filtering or triggering tasks where the primary objective is the significant reduction of data rates. For instance, at the Pierre Auger Observatory, AI-based triggers achieved a signal efficiency of 68 % compared to just 16.8 % for traditional methods, even after quantization to 13 bits [5].

Outlook: Cosmic-Ray Radio Detection Another relevant use case for aggressive quantization is in the field of radio detection of cosmic-ray-induced Extensive Air Showers (EASs). The target signals are short radio pulses embedded in a complex and time-varying background dominated by transient noise and Radio Frequency Interference (RFI). This makes simple threshold-based triggering extremely challenging: thresholds low enough to retain weak air-shower signals can lead to unacceptable false-positive rates while stricter thresholds suppress relevant events. The same background complexity also complicates the training of machine-learning triggers, since

the model must distinguish rare air-shower pulses from a broad range of non-stationary backgrounds.

The Pierre Auger Observatory is one prominent experimental setting where this problem appears in practice, using large-scale radio antenna arrays to detect EASs induced by ultra-high-energy cosmic rays. In this context, autonomous self-triggering is especially important because external triggers from particle detectors can limit the detection of highly inclined events, where the particle cascade is largely absorbed in the atmosphere while the radio signal remains measurable.

For the methodology considered here, radio traces can be represented in a waveform format comparable to the SiPM case, using 128 samples per frame, with 12-bit signed integer amplitudes.

Previous work on ML-driven triggerring achieved a signal efficiency of approximately 68 % at a false-positive rate of 10^{-4}, demonstrating the potential of learned triggers in high-interference environments [5]. Real-time neural network inference is therefore a promising route toward autonomous self-triggering of the radio arrays. Applying the BNNs approach studied above could further reduce the previously achieved latency of approximately 2 µs.

6 Conclusion

This work demonstrates that QNNs serve as a critical bridge between the high-performance requirements of modern Artificial Neural Networks (ANNs) and the severe resource constraint environments of experimental physics. The constraints in the particular environment have to be considered, as they determine the approaches to be taken.

Our results indicate that PTQ provides a rapid and practical pathway for model compression, when latency is not of major concern. It achieves substantial reductions in memory usage (4×) and computational cost (16×) for 8-bit integer representations while maintaining detection quality. Using the same quantization strength, the advantage of QAT turned out to be minimal in our use case of quantum dot calibration. For the two models investigated, a UNet-447 and a UNet-38k, the prediction quality only increases from 83.5 % to 84.6 % and from 96.0 % to 96.1 %, respectively, with a cost of additional training steps.

For applications demanding extreme energy efficiency and ultra-low latency, BNNs implemented via LUTs on FPGA hardware represent a promising solution due to nanosecond-scale inference latencies. Our results in the use case of SiPM readout yield to 10 ns to 15 ns latency without requiring specialized DSP or BRAM resources, which are often the limiting factor in FPGA designs. The primary task at this stage in the readout is the reduction of irrelevant data. With a prediction quality of around 74 % for irrelevance (”ugly”) in the most extreme case of only using $\sim$ 1000 parameters, the data transmission is reduced significantly.

Also the training of BNNs has been investigated based on a Genetic Algorithm (GA) for our hardware-constrained, LUT-native BNN architecture. With this, we could extend prior work to a forward-evaluation-only 2-bit weight/activation scheme, allowing the training of non-differentiable network architectures. The open-source `HCL4BNN` framework supports reproducible hardware-constrained learning by utilizing GAs to navigate non-differentiable search spaces. This framework successfully bridges the gap between Python-based experimentation

and VHDL-based hardware synthesis, enabling the creation of autonomous, intelligent scientific instrumentation at the edge.

Collectively, these findings contribute to a new generation of autonomous, intelligent scientific instrumentation. By integrating hardware-embedded AI directly at the edge, experimental systems can achieve real-time, on-device data processing, thereby enhancing the scalability and scientific reach of next-generation detectors.

7 Appendix

7.1 Hyperparameters Used to Train the Qubit Tuning Approach

For training of the U-Nets that are evaluated for the use case of charge transition detection in charge stability diagrams, the PyTorch implementation of the AdamW optimizer [44] is used with the parameters `lr = 0.1` and `weight_decay = 0.0001`. Also the `OneCycleLR` scheduler is used to adjust the learning rate during training. The unquantized versions are trained for 4 epochs with a total of 1.000.000 charge stability diagrams and a batch size of 128. For the implementation of PTQ and QAT the default qconfigs for the `qnnpack` backend given in PyTorch are used. For the training of the QAT versions the quantization parameters (zero point and scale) are fixed after 3 epochs and another 3 epochs are executed to fine tune the weights with fixed quantization parameters.

7.2 Hyperparameters of Genetic Algorithm Runs

Table 6 lists the hyperparameters used for GA training with dynamic and fixed random seeds for training and validation dataset generation.

Parameter	Value	Description
`pop_size`	300	Number of individuals per generation
`ngen`	50	Number of generations the GA is evolved for = stopping criterion
`nmutbit`	50	Expected number of mutated bits per genome (sets per-bit mutation probability, see Sec. 3.2.2)
`tourn_size`	5	Tournament size used for parent selection
`cxpb`	0.5	Crossover probability per individual
`cxpb_bit`	1	Crossover probability per bit, given that crossover occurs
`elite_size`	2	Number of top individuals carried over unchanged (elitism)
`train_n_frames`	300	Number of ”good”/”ugly” waveforms regenerated per fitness evaluation during training
`train_static_seed`	123	Random seed fixed for the training data generator
`validation_n_frames`	5000	Number of waveforms used for the independent validation/test evaluation
`validation_static_seed`	1	Random seed fixed for the validation data generator

Table 6: Genetic algorithm hyperparameters and dataset generation settings used for BNN training.

7.3 Glossary of Relevant Hardware-Specific Expressions

FPGA A Field Programmable Gate Array (FPGA) is a programmable integrated circuit for logic operations. It consists of repeating elements, especially LUTs, DSPs, and memory blocks. By combining these fundamental elements together, logic functions can be programmed onto the FPGA. The instructions are usually given in a HDL and then synthesized for the specific hardware.

LUT A Look-Up Table (LUT) is a fundamental building block of an FPGA. It provides a configurable function based on full adders and single bit storage.

DSP A Digital Signal Processor (DSP) is a larger fundamental building block inside of an FPGA that allows more complex signal operations, for instance multiplications. It is a much scarcer resource than LUTs on FPGAs.

BRAM Data storage in an FPGA depends on the amount of data and accessible hardware components. Usually available inside the FPGA fabric and most flexible is storing data in a memory segment called Block RAM (BRAM). Denser and more efficient storage may be available with external components but is not part of the fabric and requires dedicated memory controllers.

HDL A Hardware Description Language (HDL) is a hardware-compatible description of the logic that can be ported onto integrated circuits. Similar to programming languages, the instructions have to be converted to fundamental instructions that are compatible with hardware. The major difference to programming languages is that HDL describes the connection of physical systems and electrical signals. Common languages are **VHDL** and **Verilog**.

HLS The hardware description in a HDL can be done manually or through an automized process. By using a High-Level Synthesis (HLS), the user describes the intended function in typical programming language like C++ and converts it via a framework into HDL.

7.4 Declarations

Data availability: Source code was published in Zenodo under [31]. SiPM training data for BNNs was generated by this code using the `SiPMDataset` Python class.

Declaration on the use of AI: Large Language Model (LLM) tools were used solely for linguistic editing, including wording, grammar, and style. All scientific content, analyses, interpretations, and conclusions were developed by the authors. All code for the framework was authored and reviewed by humans, and all changes were merged only after human review and verification.

References

[1] Valentin Kuznetsov, Luca Giommi, and Daniele Bonacorsi. MLaaS4HEP: Machine Learning as a Service for HEP. arXiv:2007.14781 [hep-ex], 2020.

[2] Sanjiban Sengupta and Lorenzo Moneta. Challenges and Implementations for ML Inference in High-energy Physics. *Python in Science Conference, 2025*, 2025.

[3] Fabian Hader, Fabian Fuchs, Sarah Fleitmann, Karin Havemann, Benedikt Scherer, Jan Vogelbruch, Lotte Geck, and Stefan van Waasen. Automated Charge Transition Detection in Quantum Dot Charge Stability Diagrams. *IEEE Transactions on Quantum Engineering*, 6:1–14, 2025.

[4] F. Rössing, D. Arutinov, A. Brignoli, H. Fischer, C. Grewing, H. Lacker, F. Lyons, A. Zambanini, and S. van Waasen. Design space exploration for particle detector read-out implementations in Matlab and Simulink on the example of the SHiP SBT. *Journal of Instrumentation*, 20(02):C02022, feb 2025.

[5] Qader Dorosti. AI-enhanced self-triggering for extensive air showers: performance and FPGA feasibility. *Journal of Instrumentation*, 20(10):P10010, October 2025.

[6] Heiko Augustin et al. New Frontiers in Muon-Spin Spectroscopy Using Si-Pixel Detectors. arXiv:2504.12993 [physics.ins-det], 2025.

[7] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[8] Fabian Hader. *On Scalable Integrated Charge State Tuning for Semiconductor Quantum Dot Devices*. PhD thesis, Forschungzentrum Jülich, 2025.

[9] Yann LeCun. The MNIST database of handwritten digits. *ATT Labs [Online]*, 1998. http://yann.lecun.com/exdb/mnist/ - Mirror: https://huggingface.co/datasets/ylecun/mnist - https://cir.nii.ac.jp/crid/1571417126193283840. Accessed: 2026-10-13.

[10] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[11] Adam Paszke et al. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[12] Jason Ansel et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[13] Claudionor N. Coelho Jr., Aki Kuusela, Shan Li, Hao Zhuang, Thea Aarrestad, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Adrian Alan Pol, and Sioni Summers. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. arXiv:2006.10159 [physics.ins-det], 2021.

[14] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: training neural networks with low precision weights and activations. *J. Mach. Learn. Res.*, 18(1):6869–6898, January 2017.

[15] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 525–542, Cham, 2016. Springer International Publishing.

[16] Alan T. L. Bacellar, Zachary Susskind, Mauricio Breternitz Jr, Eugene John, Lizy K. John, Priscila M. V. Lima, and Felipe M. G. França. Differentiable Weightless Neural Networks. arXiv:2410.11112 [cs.LG], 2025.

[17] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. LUTNet: Rethinking Inference in FPGA Soft Logic. arXiv:1904.00938 [cs.LG], 2019.

[18] Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. arXiv:2004.03021 [eess.SP], 2020.

[19] Olivia Weng et al. Greater than the sum of its LUTs: Scaling up LUT-based neural networks with AmigoLUT. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 25–35. ACM, 2025.

[20] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, 1992.

[21] Gregory S. Hornby, Al Globus, Derek S. Linden, and Jason D. Lohn. Automated Antenna Design with Evolutionary Algorithms. In *AIAA Space 2006*, San Jose, CA, USA, September 2006. American Institute of Aeronautics and Astronautics. NASA Technical Reports Server document ID 20060024675.

[22] Marta Meloni, Achim Stahl, and Livia Ludhova. *Optimization of a neutrino beam for the study of CP violation with the LENA and JUNO detector*. PhD thesis, RWTH Aachen University, 2016.

[23] AMD Inc. *Vitis High-Level Synthesis User Guide (UG1399)*, September 2025. Accessed: 2026-10-13.

[24] Farah Fahim et al. hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices. arXiv:2103.05579 [cs.LG], 2021.

[25] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 65–74. ACM, February 2017.

[26] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems. 10.5281/zenodo.4724125, 2015. Software available from tensorflow.org.

[27] Giuseppe Franco, Alessandro Pappalardo, and Nicholas J Fraser. Xilinx/brevitas. 10.5281/zenodo.3333552, 2024.

[28] Marco Milenkovic, Lukas Bauer, and Michael Schäferling. FPGA-Hardwareentwicklung mit Open-Source-Tools. Talk presented at the 20th Linux-Infotag Augsburg (LIT) - https://www.luga.de/static/LIT-2024/assets/talks/LIT_Pr%C3%A4sentation.pdf, 2024. Accessed 2026-10-13.

[29] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU). arXiv:1803.08375 [cs.NE], 2019.

[30] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[31] Ilja Bekman, Alperen Aksoy, and Sarah Fleitmann. HCL4BNN - Hardware-Constrained Learning for binary neural networks. https://github.com/fzj-ica/HCL4BNN. doi:10.5281/zenodo.17542690.

[32] J. Peña-Rodríguez, J. Förtsch, C. Pauly, and K. H. Kampert. A simulation framework for sipms. arXiv:2411.16710 [physics.ins-det], 2025. https://github.com/JesusPenha/SiPM-APD-MPPC.

[33] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006.

[34] AMD Inc. *Vivado Design Suite User Guide: Implementation (UG904)*, May 2025. Accessed: 2026-10-13.

[35] AMD Inc. *Vivado Design Suite User Guide: Synthesis (UG901)*, June 2025. Accessed: 2026-10-13.

[36] Fabian Hader, Sarah Fleitmann, Jan Vogelbruch, Lotte Geck, and Stefan van Waasen. Simulation of charge stability diagrams for automated tuning solutions (simcats). *IEEE Transactions on Quantum Engineering*, 5:1–14, 2024.

[37] Fabian Hader. SimCATS_gaas_v1_random_variations_v2. https://zenodo.org/doi/10.5281/zenodo.13903285.

[38] Martin Ferianc. Quantisation example in pytorch. https://gist.github.com/ibekman/285836bb537f6905d49945217db29594, 2020. Accessed: 2026-10-13.

[39] PyTorch Foundation. BCEWithLogitsLoss. https://docs.pytorch.org/docs/2.13/generated/torch.nn.BCEWithLogitsLoss.html. Accessed: 2026-10-13.

[40] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[41] Fabian Hader, Fabian Fuchs, Karin Havemann, Sarah Fleitmann, and Jan-Friedrich Vogelbruch. SimCATS-Datasets. 10.5281/ZENODO.13862231 - https://github.com/f-hader/SimCATS-Datasets, 2024.

[42] SHiP Experiment — ship.web.cern.ch. https://ship.web.cern.ch/. Accessed 2026-05-08.

[43] Xilinx Inc. now AMD Inc. *ZCU104 Evaluation Board User Guide (UG1267)*. Advanced Micro Devices, Inc., v1.1 edition, 2018. Accessed: 2026-10-13.

[44] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.