nature machine intelligence

ARTICLES

https://doi.org/10.1038/s42256-021-00356-5

Check for updates

Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors

Claudionor N. Coelho Jr[1], Aki Kuusela[2], Shan Li[2], Hao Zhuang[2], Jennifer Ngadiuba[3], Thea Klaeboe Aarrestad[4]✉, Vladimir Loncar[4,5], Maurizio Pierini[4], Adrian Alan Pol[4] and Sioni Summers[4]

Although the quest for more accurate solutions is pushing deep learning research towards larger and more complex algorithms, edge devices demand efficient inference and therefore reduction in model size, latency and energy consumption. One technique to limit model size is quantization, which implies using fewer bits to represent weights and biases. Such an approach usually results in a decline in performance. Here, we introduce a method for designing optimally heterogeneously quantized versions of deep neural network models for minimum-energy, high-accuracy, nanosecond inference and fully automated deployment on chip. With a per-layer, per-parameter type automatic quantization procedure, sampling from a wide range of quantizers, model energy consumption and size are minimized while high accuracy is maintained. This is crucial for the event selection procedure in proton–proton collisions at the CERN Large Hadron Collider, where resources are strictly limited and a latency of $\mathcal{O}(1)$ μs is required. Nanosecond inference and a resource consumption reduced by a factor of 50 when implemented on field-programmable gate array hardware are achieved.

With edge computing, real-time inference of deep neural networks (DNNs) on custom hardware has become increasingly relevant. Smartphone companies are incorporating artificial intelligence (AI) chips in their design for on-device inference to improve user experience and tighten data security, and the autonomous vehicle industry is turning to application-specific integrated circuits (ASICs) to keep the latency low. Although the typical acceptable latency for real-time inference in applications like those above is $\mathcal{O}(1)$ ms (refs. [1,2]), other applications may require submicrosecond inference. For example, high-frequency trading machine learning (ML) algorithms are running on field-programmable gate arrays (FPGAs) to make decisions within nanoseconds[3]. At the extreme inference spectrum end of both the low latency (as in high-frequency trading) and limited area (as in smartphone applications) is the processing of data from proton–proton collisions at the Large Hadron Collider (LHC) at CERN[4]. In the particle detectors around the LHC ring, tens of terabytes of data per second are produced from collisions occurring every 25 ns. This extremely large data rate is reduced by a real-time event filter processing system—the trigger—which decides whether each discrete collision event should be kept for further analysis or be discarded. Data are buffered close to the detector while the processing occurs, with a maximum latency of $\mathcal{O}(1)$ μs to make the trigger decision. High selection accuracy in the trigger is crucial to keep only the most interesting events while keeping the output bandwidth low, reducing the event rate from 40 MHz to 100 kHz. In 2027, the LHC will be upgraded from its current state, capable of producing up to one billion proton–proton collisions per second, to the so-called High Luminosity-LHC (HL-LHC)[5]. This will involve increasing the number of proton collisions occurring every second by a factor of five to seven, ultimately resulting in a total amount of accumulated data one order of magnitude higher than what is possible with the current collider. With this extreme increase, ML solutions are being explored as fast approximations of the algorithms currently in use to minimize the latency and maximize the precision of tasks that can be performed.

Hardware used for real-time inference in particle detectors usually has limited computational capacity due to size constraints. Incorporating resource-intensive models without a loss in performance poses a great challenge. In recent years, many developments have aimed at providing efficient inference from an algorithmic point of view. This includes compact network design[6–10], weight and filter pruning[11,12] or quantization. In post-training quantization[13–17], the pre-trained model parameters are translated into lower-precision equivalents. However, this process is, by definition, lossy, and it sacrifices model performance. Therefore, solutions to do quantization-aware training have been suggested[18–27]. In these, a fixed numerical representation is adopted for the whole model, and the model training is performed enforcing this constraint during weight optimization. More recently[28–31], it has been argued that some layers may be more accommodating for aggressive quantization, whereas others may require more expensive arithmetic. This suggests that per-layer heterogeneous quantization is the optimal way to achieve higher accuracy at low resource cost, but it may require further specialization of hardware resources.

In this Article, we introduce a novel workflow for finding the optimal heterogeneous quantization per layer and per parameter type for a given model, and deploy that model on FPGA hardware. Through minimal code changes, the model footprint is minimized while retaining high accuracy, and then

[1]Palo Alto Networks, Palo Alto, CA, USA. [2]Google LLC, Mountain View, CA, USA. [3]California Institute of Technology (Caltech), Pasadena, CA, USA. [4]European Organization for Nuclear Research (CERN), Geneva, Switzerland. [5]Institute of Physics, Belgrade, Serbia. ✉e-mail: thea.aarrestad@cern.ch

translated into low-latency firmware. This Article makes the following contributions:

- We implement a range of quantization methods in a common library, providing a broad base from which optimal quantizations can easily be sampled.
- We introduce a novel method for finding the optimal heterogeneous quantization for a given model, resulting in minimum area or minimum power DNNs while maintaining high accuracy.
- We have made these methods available online in easy-to-use libraries, called QKeras and AutoQKeras[60], where simple drop-in replacement of Keras[32] layers makes it straightforward for users to transform Keras models to their equivalent deep heterogeneously quantized versions, which are trained quantization-aware. Using AutoQKeras, a user can trade off accuracy by model size reduction (for example, area or energy).
- We have added support for quantized QKeras models in the library, hls4ml[13], which converts these pre-trained quantized models into highly parallel FPGA firmware for ultralow-latency inference.

To demonstrate the substantial practical advantages of these tools for high-energy physics and other inference on the edge applications:

- We conduct an experiment consisting of classifying events in an extreme environment, namely the triggering of proton–proton collisions at the CERN LHC, where resources are limited and a maximum latency of $\mathcal{O}(1)$ µs is imposed.
- We show that inference within 60 ns and a reduction of the model resource consumption by a factor of 50 can be achieved through automatic heterogeneous quantization, while maintaining similar accuracy (within 3% of the floating-point model accuracy).
- We show that the original floating-point model accuracy can be maintained for homogeneously quantized DNNs down to a bit-width of six while reducing resource consumption by up to 75% through quantization-aware training with QKeras.

The proposed pipeline provides a novel, automatic end-to-end flow for deploying ultralow-latency, low-area DNNs on chip. This will be crucial for the deployment of ML models on FPGAs in particle detectors and other fields with extreme inference and low-power requirements.

In the remainder of the Article we discuss previous work related to model quantization and model compression with a focus on work related to triggering in particle detectors, we uncover the novel library for training ultralow-latency optimally heterogeneously quantized DNNs (QKeras), we describe the procedure of automatic quantization for optimizing model size and accuracy simultaneously and, finally, we deploy these optimally quantized QKeras models on an FPGA and evaluate their performance.

Motivation

The hardware triggering system in a particle detector at the CERN LHC is one of the most extreme environments in which one can imagine deploying DNNs. Latency is restricted to $\mathcal{O}(1)$ µs, governed by the frequency of particle collisions and the number of on-detector buffers. The system consists of a limited amount of FPGA resources, all of which are located in underground caverns 50–100 m below the ground surface, where they work on thousands of different tasks in parallel. Because of the high number of tasks being performed, limited cooling capabilities, limited space in the cavern and the limited number of processors, algorithms must be kept as resource-economic as possible. To minimize the latency and maximize the precision of tasks that can be performed in the hardware trigger, ML solutions are being explored as fast approximations of the algorithms currently in use. To simplify the implementation of these, a general library for converting pre-trained ML models into FPGA or ASIC firmware has been developed—hls4ml[13]. The package comprises a library of optimized C++ code for common network layers, which can be synthesized through a high-level synthesis (HLS) tool. Converters are provided for multiple model formats, like TensorFlow[33], Keras[32], PyTorch[34] and ONNX[35].

Although there are other libraries for the translation of ML models to FPGA firmware, as summarized in refs. [36–39], hls4ml targets extreme low-latency inference to stay within the strict constraints of $\mathcal{O}(1)$ µs imposed by the hardware trigger systems. In addition, the unique aspect of hls4ml is the support for multiple HLS-vendor backends like Xilinx Vivado HLS, Intel Quartus HLS[40] and Mentor Catapult HLS[41], all of which are in use at the LHC experiments. The Vivado HLS backend is the most advanced and therefore the one used in this Article.

The hls4ml inference architecture is introduced in ref. [13]. A model-specific, layer-unrolled architecture is used to produce ultralow-latency, resource-efficient inference engines for particle physics. The computation for each NN layer is carried out in distinct hardware elements of the target device, which allows for high computational throughput through the layer pipeline, as well as a fine-grained configuration of each layer (including quantization). A simple handle, named 'Reuse Factor' enables users to control the parallelization of the computation, again at a per-layer level. In the fully parallel model, using a Reuse Factor of 1, each individual multiplication of the NN layers is carried out on different resources (whether FPGA digital signal processors (DSPs) or lookup tables (LUTs)). With a Reuse Factor greater than 1, multiplication elements are reused sequentially to reduce the resource cost, at the expense of latency and throughput. This simple handle enables rapid design space exploration as well as configurability to target-specific constraints in the available resources, latency and throughput.

In addition, data access at the NN input and output, as well as data movement between NN layers, can be configured to be fully parallel or fully serial. The former option is used to target ultralow-latency, high-throughput inference in the real-time processing of particle physics experiments, while the latter can be used to fit larger NN models within the available FPGA resources when ultralow latency is not as much of a constraint.

The hls4ml library is implemented as a Python package to facilitate ease of use for non-experts, as well as consistency with other popular deep learning libraries. The first step in the conversion into FPGA firmware consists of translating a given model into an internal representation of the network graph. During this conversion, user-specified optimization configurations are attached to the model, such as the choice of quantization and parallelization. The internal representation is written out into an HLS project, assigning the appropriate layers of the target NN and the user configuration. This HLS project can then be synthesized with the FPGA vendor tools, generating an IP core that can be used in the target application. Many commonly used NN layers are supported: Dense, Convolution, BatchNormalization and several Activation layers. In addition, domain-specific layers can be easily added, one example being compressed distance-weighted graph networks[42].

In hls4ml, the precision used to represent weights, biases, activations and other components is configurable through post-training quantization, replacing the floating-point values by lower-precision fixed-point ones. This allows compression of the model size, but to some extent sacrifices accuracy. Recently, support for binary and ternary precision DNNs[43] trained quantization-aware has been included in the library. This greatly reduces the model size, but requiring such an extremely low precision of each parameter type sacrifices accuracy and generalization.

As demonstrated in refs. [28–31], mixed-precision quantization (that is, keeping some layers at higher precision and some at lower precision) is a promising approach to achieve smaller models with high

Table 1 | Per-layer quantization for post-training quantized models

Model	Precision							
	Dense	ReLU	Dense	ReLU	Dense	ReLU	Dense	Softmax
BF/BP	⟨14, 6⟩	⟨14, 6⟩	⟨14, 6⟩	⟨14, 6⟩	⟨14, 6⟩	⟨14, 6⟩	⟨14, 6⟩	⟨14, 6⟩
BH[a]	w:⟨8, 3⟩ b:⟨4, 2⟩	⟨13, 7⟩	⟨7, 2⟩	⟨10, 5⟩	⟨5, 2⟩	⟨8, 4⟩	w:⟨7, 3⟩ b:⟨4, 1⟩	⟨16, 6⟩

When different precision is used for weights and biases, the quantization is listed as w and b, respectively.

accuracy. However, finding the optimal heterogeneous quantization per layer and per parameter type, here referred to as 'quantization configuration', is extremely challenging, with the search space increasing exponentially with the number of layers in a model[30]. A solution for finding the mixed quantization configuration that yields the best generalization and accuracy using the Hessian spectrum is proposed in ref. [30]. For ML applications in hardware triggering systems, the resources one has at disposal, as well as the minimum tolerable model accuracy, are usually known. Finding the best model for a given task is therefore a fine compromise between the desired model compression and accuracy with respect to the floating-point-based model. Both factors must be considered when tuning quantization. The goal of this work is thus to provide a method for finding the optimal mixed-precision configuration for a given model, accounting for both the desired model size and accuracy when optimizing the architecture, and to transform these into highly parallel firmware for ultralow-latency inference on chip.

Related work

Closely related to the work presented here are the FINN[44] and FINN-R[45] frameworks from Xilinx Research Labs, which aim to deploy quantized neural networks on Xilinx FPGAs. The same group have also developed a library for quantization-aware training, Brevitas[46], based on PyTorch model formats. The LogicNets design flow[47], also from Xilinx Research Labs, allows for the training of quantized DNNs that map to highly efficient Xilinx FPGA implementations. A comparison between the approach presented here and LogicNets is provided in the section 'Ultralow-latency, quantized model on FPGA hardware'. The FP-DNN[48] framework takes TensorFlow[33]-described DNNs as input and maps them onto FPGAs. The open-source alternative, DNNWeaver[49], automatically generates accelerator Verilog code using optimized templates. Other frameworks focusing on the mapping of convolutional architectures onto efficient hardware design include Snowflake[50], fpgaConvNet[51–53] and ref. [54]. For other work on FPGA DNN inference, we refer to refs. [36–39,55]. TensorFlow Lite[56] is a set of tools for on-device inference with low latency and small binary sizes, targeting mobile, embedded and Internet of Things (IoT) devices. Currently, TensorFlow Lite supports deployment on Android and iOS devices, embedded Linux and microcontrollers.

Our approach differs from those above with its emphasis on being a multi-backend tool, embracing a fully on-chip design to target the microsecond latency imposed in physics experiments. The hls4ml library is completely open-source, and aims to provide domain scientists with easy-to-use software for deploying highly efficient ML algorithms on hardware.

In HAQ[28], a hardware-aware automated framework for quantization is introduced. The automization procedure consists of computing the curvature of the weight space of a layer, assuming a low curvature will require a lower bit precision for the weights. Our approach differs from HAQ by combining reduced bit precision with filter or neuron unit tuning, where the number of filters or neurons can be automatically tuned during the scan. In this case, the problem becomes highly nonlinear, and we therefore take advantage of an AutoML-type of approach. A Bayesian optimization or randomized search is performed to find a solution that encompasses the precision used for the weights and activations, and the number of units or filters of the layer.

Particle identification in the hardware trigger

A crucial task performed by the trigger system that could be greatly improved by a ML algorithm, both in terms of latency and accuracy, is the identification and classification of particles coming from each proton–proton collision. In this Article, we analyse the publicly available dataset introduced in refs. [13,57]. Here, a dataset[58] for the discrimination of jets, a collimated spray of particles, stemming from the decay and/or hadronization of five different particles was presented. This consists of quark (q), gluon (g), W boson, Z boson and top (t) jets, each represented by 16 physics-motivated high-level features. In ref. [13], this dataset was used to train a DNN for deployment on a Xilinx FPGA. This model was compressed through post-training quantization to further reduce the FPGA resource consumption and provides a baseline to measure the benefits of quantization-aware training with heterogeneous quantization, over post-training quantization.

Adopting the same architecture as in ref. [13], we use a fully connected neural network consisting of three hidden layers (64, 32 and 32 nodes, respectively) with rectified linear unit (ReLU) activation functions. The architecture is shown in Extended Data Fig. 1. The output layer has five nodes, yielding a probability for each of the five classes through a softmax activation function. The model definition in TensorFlow Keras is given in Listing 1.

As in ref. [13], the weights of this network are homogeneously quantized post-training to a fixed-point precision yielding the best compromise between accuracy, latency and resource consumption. This is found to be a fixed-point precision, or bit-width, of 14 bits with 6 integer bits, in the following referred to as ⟨14, 6⟩. We refer to this configuration as the baseline full model (BF). We then train a second pruned version of the BF model, here referred to as baseline pruned (BP). This model has 70% of its weights set to zero through an iterative process where small weights are removed using the TensorFlow Pruning application programming interface[59], following ref. [13]. This reduces the model size and resource consumption considerably, as all zero-multiplications are excluded during the firmware implementation. We then create one heterogeneously quantized version of the BP model, where each layer is quantized independently post-training to yield the highest accuracy possible at the lowest resource cost. We start with an initial configuration of the model quantization using a wide bit-width, then iteratively reduce the bit-width until reaching a threshold in accuracy loss relative to the initial floating-point model, evaluated on the training set. We iterate over the model in layer order, finding the appropriate precision for weights, biases and output of a given layer, before moving to the next. We apply a more strict threshold in accuracy for earlier layers, because each round of precision reduction only degrades the accuracy. In this case we restrict to a 1% accuracy difference in the first layer, loosening to 2% for the final layer. This model is referred to as the baseline heterogeneous (BH) model. A summary of the per-layer quantizations for the baselines is provided in Table 1.

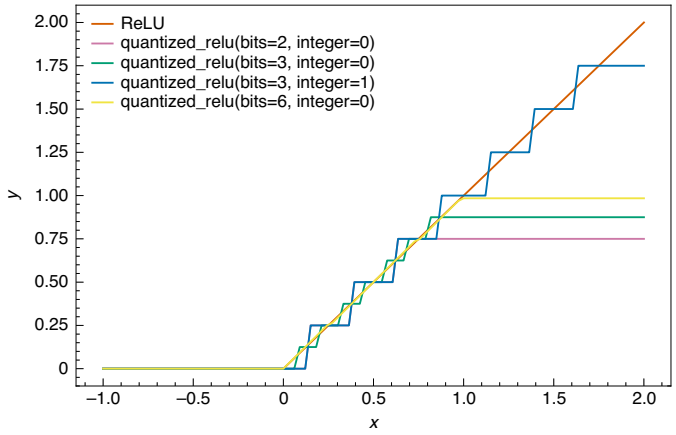


Fig. 1 | Quantized ReLU function in QKeras. The quantized_relu function as implemented in QKeras for 2-bit (purple), 3-bit (green and blue) and 6-bit (yellow) precision and for 0 or 1 integer bits. The unquantized ReLU function is shown for comparison (orange).

From ref. [13], we know that a post-training quantization of this model results in a degradation in model accuracy. The smaller the model footprint is made through post-training quantization, the larger the accuracy degradation becomes. To overcome this, we develop a novel library that, through minimal code changes, allows us to create deep heterogeneously quantized versions of the Keras model, trained quantization-aware.

In addition, as the amount of available resources on chip is known in advance, we want to find the optimal model for a given use-case allowing a trade-off between model accuracy and resource consumption. We therefore design a method for performing automatic quantization, minimizing the model area while maximizing accuracy simultaneously through a novel loss function. These solutions, simple heterogeneous quantization-aware training and automatic quantization are explained in the following sections.

Keras[32] is a high-level application programming interface designed for building and training deep learning models. It is used for fast prototyping, advanced research and production. To simplify the procedure of quantizing Keras models, we introduce QKeras[60]: a quantization extension to Keras that provides a drop-in replacement for layers performing arithmetic operations. This allows for efficient training of quantized versions of Keras models.

QKeras is designed using the design principle of Keras—that is, being user-friendly, modular, extensible and minimally intrusive to Keras native functionality. The code is based on the work of refs. [18,22], but provides a substantial extension to these. This includes providing a richer set of layers (for instance, including ternary and stochastic ternary quantization), extending the functionality by providing functions to aid the estimation of model area and energy consumption, allowing for simple conversion between non-quantized and quantized networks, and providing a method for performing automatic quantization. Importantly, the library is written in such a way that all the QKeras layers maintain a true drop-in replacement for Keras ones so that minimal code changes are necessary, greatly simplifying the quantization process. During quantization, QKeras uses the straight-through estimator[19], where the forward pass applies the quantization functions and the backward pass assumes the quantization as the identity function to make the gradient differentiable.

For the model in Listing 1, creating a deep quantized version requires just a few code changes. An example conversion is shown in Listing 2.

Listing 1. Defining a model in Keras: TensorFlow Keras model definition

```
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.layers import BatchNormalization
x = Input((16))
x = Dense(64)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = Dense(32)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = Dense(32)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = Dense(5)(x)
x = Activation('softmax')(x)
```

Obtaining optimal heterogeneous quantization

The necessary code modifications consist of typing Q in front of the original Keras data manipulation layer name and specifying the appropriate quantization type, for instance, the kernel_quantizer and bias_quantizer parameters in a QDense layer. We change only data manipulation layers that perform some form of computation that may change the data input type and create variables (trainable or not). Data transport layers, namely layers performing some form of change of data ordering, without modifying the data itself, remain the same, for example Flatten. When quantizers are not specified, no quantization is applied to the layer and it behaves as the unquantized Keras layer. The only exception is the QBatchNormalization layer. Here, when no quantizers are specified, a power-of-2 quantizer is used for the trainable parameters of the batch normalization layer, γ and β, as well as for the emperical variance σ, while the emperical mean μ remains unquantized. This has worked best when attempting to implement quantization efficiently in hardware and software (γ and σ become shift registers and β maintains the dynamic range aspect of the centre parameter)

Listing 2. Defining a model in QKeras: quantized QKeras model example.

```
from tensorflow.keras.layers import Input, Activation
from qkeras import quantized_bits
from qkeras import QDense, QActivation
from qkeras import QBatchNormalization
x = Input((16))
x = QDense(64, kernel_quantizer=quantized_bits(6,0,alpha=1),
bias_quantizer=quantized_bits(6,0,alpha=1))(x)
x = QBatchNormalization()(x)
x = QActivation('quantized_relu(6,0)')(x)
x = QDense(32, kernel_quantizer=quantized_bits(6,0,alpha=1),
bias_quantizer=quantized_bits(6,0,alpha=1))(x)
x = QBatchNormalization()(x)
x = QActivation('quantized_relu(6,0)')(x)
x = QDense(32, kernel_quantizer=quantized_bits(6,0,alpha=1),
bias_quantizer=quantized_bits(6,0,alpha=1))(x)
x = QBatchNormalization()(x)
x = QActivation('quantized_relu(6,0)')(x)
x = QDense(5, kernel_quantizer=quantized_bits(6,0,alpha=1),
bias_quantizer=quantized_bits(6,0,alpha=1))(x)
x = Activation('softmax')(x)
```

The second code change is to pass appropriate quantizers, for example quantized_bits. In the example above, QKeras is instructed to quantize the kernel and bias to a bit-width of 6 and 0 integer bits. The parameter alpha can be used to change the absolute scale of the weights while keeping them discretized within the chosen bit-width. For example, in a binary network, rather than using the representations ±1, one can use ±alpha. In QKeras, by setting

alpha='auto', we also allow for the value of alpha to be computed during training from the absolute scale of the weights in question. Further details are provided in the Methods and illustrated in Extended Data Fig. 2.

QKeras works by tagging all variables, weights and biases created by Keras, as well as the output of arithmetic layers, with quantized functions. Quantized functions are specified directly as layer parameters and then passed to QActivation, which acts as a merged quantization and activation function. Quantizers and activation layers are treated interchangeably. To minimize code changes, the quantizers' parameters have carefully crafted and pre-defined defaults or are computed internally for optimal set-up.

The quantized_bits quantizer used above performs mantissa quantization:

$$2^{\text{int}-b+1}\text{clip}(\text{round}(x \times 2^{b-\text{int}-1}), -2^{b-1}, 2^{b-1}-1), \qquad (1)$$

where x is the input, b specifies the number of bits for the quantization, and 'int' specifies how many bits of bits are to the left of the decimal point.

The quantizer used for the activation functions in Listing 2, quantized_relu, is a quantized version of ReLU[61]. Two input parameters are passed, namely the precision, in this case 6 bits, and number of integer bits, in this case zero, respectively. The class has further attributes, for instance allowing for stochastic rounding of the activation function, all of which are described in detail in ref. [60]. Figure 1 shows the quantized ReLU function for three different bit-widths and two different numbers of integer bits.

Through simple code changes like those above, a large variety of quantized models can be created. A full list of quantizers and layers is provided in the Methods and listed in Extended Data Fig. 3 or in the QKeras code repository[60].

We use QKeras to create a range of deep homogeneously quantized models, trained quantization-aware and based on the same architecture as the baseline model, which will provide a direct comparison between post-training quantization and models trained using QKeras. The model in Listing 2 is an example of such a homogeneously quantized model. Finally, we want to create an optimally heterogeneously quantized QKeras model with a considerably reduced resource consumption, without compromising the model accuracy. The search space for finding such a configuration is large and exponential in layers. We therefore attempt to automatize the process by allowing users to scan through all the available quantizers in QKeras to find the configuration that fits the available chip area while maintaining high accuracy.

Resource-aware automatic quantization

As described in the section 'Motivation', there are several methods for finding the optimal quantization configuration for a given model. These usually proceed by calculating the sensitivity of a given layer to quantization through evaluation of how small disturbances within that layer influence the loss function.

Often, as for example in refs. [29,30], only maximization of the model's accuracy and ability to generalize is considered. However, when doing inference on the edge, resources are often limited and shared between multiple applications. This is the case in particle detectors, where a single FPGA is used to perform multiple different tasks. The desired accuracy and size constraints of the model in question are known in advance, and it is desirable to optimize the precision configuration considering both model accuracy and size. Some methods, like HAQ[28], do perform such a hardware-aware optimization. However, only the weight precision per layer is considered. When models are strongly quantized, it is often the case that more or fewer filters in convolutional layers, or neurons in densely connected layers, are necessary. A fine-tuning of the number of units per layer is therefore crucial to achieve the highest possible accuracy at the lowest resource cost.

In this Article, we introduce a method for performing automatic quantization where the user can trade off model area or energy consumption by accuracy in an application-specific way. The per-layer weight precision as well as the number of neurons or filters per layer are optimized simultaneously. By defining a forgiving factor based on the tolerated drop in accuracy for a given reduction in resource cost, the best quantization configuration and number of units per layer, for a set of given size or energy constraints, can be found. We consider both energy minimization and bit-size minimization as a goal in the optimization.

Approximating relative model energy consumption. To target a reduction in model energy consumption, a high-level estimate of the model energy is needed. Here, we only concern ourselves with the difference in energy consumption when comparing models using different quantizations, and not the absolute energy, as this is highly hardware-specific. To this end, we assume an energy model where the energy consumption of a given layer is defined as

$$E_{\text{layer}} = E_{\text{input}} + E_{\text{parameters}} + E_{\text{MAC}} + E_{\text{output}}.$$

These correspond to the energy cost of reading inputs (E_{input}), parameters ($E_{\text{parameters}}$) and output (E_{output}) and the energy required to perform multiply-and-accumulate (MAC) operations (E_{MAC}). For the first three, in a similar way to compulsory accesses in cache analysis[62], we only consider the first access to the data, as only compulsory accesses are independent of the hardware architecture and memory hierarchy of an accelerator, when comparing models using the same architecture. We also assume a fully unrolled implementation on the hardware (as is the case with hls4ml). For the MAC energy estimation, we only consider the energy needed to compute the MAC. We do not include the energy usage of registers, or glue and pipeline logic that will affect the overall energy profile of the device. For a given architecture, this energy consumption is known, and here we assume a 45 nanometre processor and follow the energy table given in ref. [63].

Although this model provides a good initial estimate, it has high variance concerning the actual energy consumption one finds in practice, especially for different architectural implementations. However, when comparing the energy of two different models, or models of different quantizations, both implemented in the same technology, this simple energy model is sufficient. The reason for this is that one can assume that the real energy of a layer is some linear combination of the high-level energy model, that is, $E^{\text{Real}}_{\text{layer}} = k_1 \times E_{\text{layer}} + k_2$, where k_1 and k_2 are constants that depend on the architecture of the accelerator and the implementation process technology. The slope can be considered as a factor accounting for the additional storage needed to keep the model running, and the offset corresponds to logic that is required to perform the operations. When comparing the energy consumption of two layers with different quantizations, L1 and L2, for the same model architecture, we have that $E^{\text{Real}}_{\text{L1}} > E^{\text{Real}}_{\text{L2}}$ if, and only if, the estimated energy $E_{\text{L1}} > E_{\text{L2}}$.

For these reasons, only relative energy estimates are considered during the automatic quantization, and users cannot target a specific energy value.

To facilitate easy estimation of the relative energy consumption or model bit size when comparing different QKeras models, we have implemented a tool in the QKeras library, QTools, which performs both data type map generation and energy consumption estimation. A data type map for weights, biases, multipliers and so on, is generated for each layer, and includes operation types, variable sizes, quantizer types and bits. The output is an estimate of the per-layer energy consumption in picojoules, as well as a dictionary of data types per layer. Included in the energy calculation is a set of other tunable specifications, such as whether parameters and activations

Table 2 | Per-layer quantization and relative energy consumption for automatically quantized QKeras models, showing per-layer quantization configuration and the relative model energy consumption for the AutoQKeras energy optimized (QE) and AutoQKeras bits optimized (QB) models, compared to the simple homogeneously quantized model, Q6

Model	Accuracy (%)	Precision							$\frac{E}{E_{Q6}}$	$\frac{\text{Bits}}{\text{Bits}_{Q6}}$
		Dense	ReLU	Dense	ReLU	Dense	ReLU	Dense	Softmax		
QE	72.3	⟨4, 0⟩	⟨4, 2⟩	Ternary	⟨3, 1⟩	⟨2, 1⟩	⟨4, 2⟩	w: Stoc. bin. b: ⟨8, 3⟩	⟨16, 6⟩	0.27	0.18
QB	72.8	⟨4, 0⟩	⟨4, 2⟩	Stoc. bin.	⟨4, 2⟩	Ternary	⟨3, 1⟩	Stoc. bin.	⟨16, 6⟩	0.25	0.17
Q6	74.8	⟨6, 0⟩	⟨6, 0⟩	⟨6, 0⟩	⟨6, 0⟩	⟨6, 0⟩	⟨6, 0⟩	⟨6, 0⟩	⟨6, 0⟩	1.00	1.00

When different precision is used for weights and biases, the quantization is listed as w and b, respectively. Stoc. bin., stochastic binary quantization.

are stored on static random-access memory (SRAM) or dynamic random-access memory (DRAM), or whether data are loaded from DRAM to SRAM. The precision of the input can also be defined for a better energy estimate. A full list of options is available in ref. [60]. The QTools library provides an additional metric for model tuning when both accuracy and energy consumption, or model size, need to be considered.

Defining a forgiving factor. With the high-level estimate of a given layer's energy consumption provided by QTools, we define a forgiving factor (FF) to be targeted during automatic quantization of the model, providing a total loss function that combines energy cost and accuracy. The FF allows one to tolerate a degradation in a given metric, such as model accuracy, if the model gain in terms of some other metric, like model size, is considerably larger. Here, we allow the forgiving metric to be either minimization of the model bit size or minimization of the model energy consumption. The FF is defined as

$$\mathrm{FF} = 1 + \Delta_{\mathrm{acc}} \times \log_R \left(S \times \frac{C_{\mathrm{ref}}}{C_{\mathrm{trial}}} \right), \tag{2}$$

where Δ_{acc} is the tolerated reduction in accuracy in percent, R is the factor stating how much smaller energy the optimized model must have compared to the original model (as a multiplicative factor to the FF metric) and S is a parameter to reduce the reference size, effectively forcing the tuner to choose smaller models. Parameters C_{ref} and C_{trial} refer to the cost (energy or bits) of the reference model and the quantization trial model being tested, respectively. The FF can be interpreted in the following way: if we have a linear tolerance for model accuracy degradation (or any other performance metric), we should be able to find a multiple of that degradation in terms of the cost reduction of the implementation. This enables an automatic quantization procedure to compensate for the loss in accuracy when comparing two models, by acting as a multiplicative factor.

Automatic quantization and rebalancing are then performed by treating quantization and rebalancing of an existing DNN as a hyper parameter search in Keras Tuner[64] using random search, hyperband[65] or Gaussian processes. We design an extension to Keras Tuner called AutoQKeras, which integrates the FF defined in equation (2) and the energy estimation provided by QTools. This allows for simultaneously tuning of the model quantization configuration and the model architecture. For example, AutoQKeras allows for tuning of the number of filters in convolutional layers and the number of neurons in densely connected layers. This fine-tuning is critical, as when models are strongly quantized, more or fewer filters might be needed. Fewer filters might be necessary in cases where a set of filter coefficients are quantized to the same value.

Consider the example of quantizing two sets of filter coefficients, [−0.3, 0.2, 0.5, 0.15] and [−0.5, 0.4, 0.1, 0.65]. If we apply a binary quantizer with scale $= \lceil \log_2(\frac{\sum |w|}{N}) \rceil$, where w are the filter coefficients and N is the number of coefficients, we will end up with the same filter binary([−0.3, 0.2, 0.5, 0.15]) = binary([−0.5, 0.4, 0.1, 0.65]) = [−1, 1, 1, 1] × 0.5. In this case, we are assuming a scale is a power-of-2 number so that it can be efficiently implemented as a shift operation. On the other hand, more filters might be needed as deep quantization drops information. To recover some of the boundary regions in layers that perform feature extraction, more filters might be needed when the layer is quantized. Finally, certain layers are undesirable to quantize, often the last layer of a network. In principle, we do not know if by quantizing a layer we need more or fewer filters or neurons and, as a result, there are advantages to treating these problems as co-dependent problems, as we may be able to achieve a lower number of resources. Note that AutoQKeras does not completely remove model layers.

In AutoQKeras, one can specify which layers to quantize by specifying the index of the corresponding layer in Keras. If attempting to quantize the full model in a single shot, the search space becomes very large. In AutoQKeras, there are two methods to cope with this: grouping layers to use the same choice of quantization or quantization by blocks. For the former, regular expressions can be provided to specify layer names that should be grouped to use the same quantization. In the latter case, blocks are quantized sequentially, either from inputs to outputs or by quantizing higher energy blocks first. If blocks are quantized one by one, assuming each block has N choices and the model consists of B blocks, one only needs to try $N \times B$, rather than N^B options. Although this is an approximation, it is a reasonable trade-off considering the explosion of the search space for individual filter selections, weight and activation quantization.

Whether to quantize sequentially from inputs to outputs or starting from the block that has the highest energy impact depends on the model. For example, for a network like ResNet[66], and if filter tuning is desirable, one needs to group the layers by the ResNet block definition and quantize the model sequentially to preserve the number of channels for the residual block. A few optimizations are performed automatically during model training. First, we dynamically reduce the learning rate for the blocks that have already been quantized so that they are still allowed to train, but at a slower pace. Also, we dynamically adjust the learning rate for the layer we are trying to quantize as opposed to the learning rate of the unquantized layers. Finally, we transfer the weights of the model blocks we have already quantized, whenever possible (when shapes remain the same).

We then use AutoQKeras to find the optimal quantization configurations for the baseline model for extremely resource-constrained situations, one targeting a minimization of the model's footprint in terms of model energy (QE) and one minimizing the footprint in terms of model bit size (QB), using the different available targets in AutoQKeras. We want to reduce the resource footprint by at least a factor of four while allowing the accuracy to drop by at most 5%. We also allow for tuning of the number of neurons for each dense layer, for the same reason given above for model filter tuning.

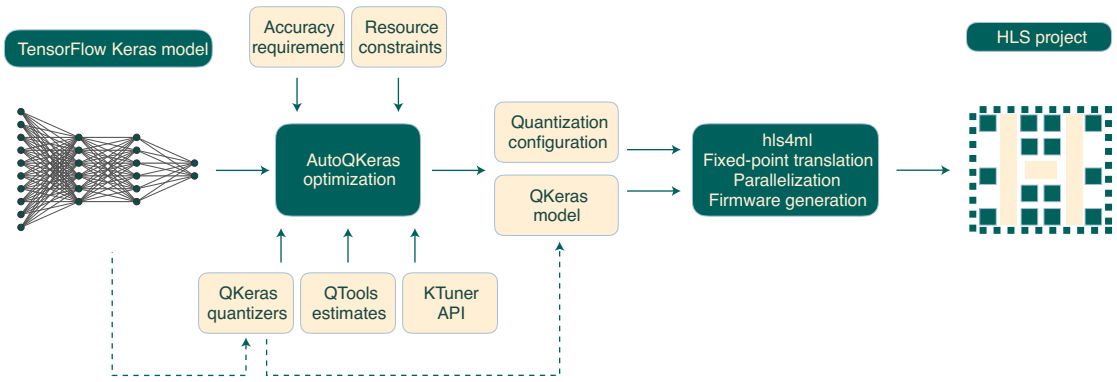


Fig. 2 | The QKeras and hls4ml workflow. The full workflow starting from a baseline TensorFlow Keras Model, which is then converted into an optimally quantized equivalent through QKeras and AutoQKeras. This model is then translated into highly parallel firmware with hls4ml.

The model is quantized sequentially per block, where one block consists of a Dense layer and a ReLU layer. The resulting quantization configuration is listed in Table 2. A very aggressive quantization configuration is obtained for both optimizations, with both binary and ternary quantizers and a bit-width of four at maximum for kernels. Despite the large search space, the obtained configurations are very similar, as is to be expected due to the strong correlation between model energy and bit size. Whenever an input or the kernel has one (binary) or two (ternary) bits, we can completely eliminate multiplication operations in an implementation, saving valuable multiplier resources.

The preferred number of neurons per layer is half that of the original (32, 16, 16 rather than 64, 32, 32).

We then compare the relative energy consumption and bit size of the QE and QB models as computed with QTools with respect to the simple homogeneously quantized model using a 6-bit precision in Listing 2, hereby referred to as Q6.

The QE and QB model energy consumption is reduced by 75% when compared to the Q6 model and, despite the aggressive quantization and reduction in neurons per layer, only a ~3% degradation in accuracy is observed for both. The total bit size is reduced by 80%. The QB model obtains a slightly smaller energy footprint than the QE model, alluding to some degree of randomness when scanning such a large search space. The relative power consumption when implemented on FPGA hardware will be discussed in the section 'Ultralow-latency, quantized model on FPGA hardware'.

All the models presented so far are trained minimizing the categorical cross-entropy loss[67] using the Adam optimizer[68]. A learning rate of 0.0001 is set as the starting learning rate. If there is no improvement in the loss for ten epochs, the learning rate is reduced by 50% until a minimum learning rate of 10^{-6} is reached. The batch size is 1,024 and the training proceeds for 100 epochs. The training time for the models trained quantization-aware with QKeras is increased by ×1.5 with respect to the Keras equivalent.

For particle detector trigger applications, it is often desirable to operate the algorithm at very low false positive rates (FPRs), ensuring that only the most interesting events are kept while staying within the available trigger bandwidth. In Extended Data Fig. 4, the classification performances of the BF, Q6, QE and QB models for two different target classes, top (t) and gluon (g), are compared. These classes were chosen as the ones where the original network, introduced in ref. [13], had the highest and lowest area under the curve (AUC) scores, respectively. Specifically, the receiver operating characteristic (ROC) curves of FPR versus true positive rate (TPR), and the corresponding AUC, are shown. The classification performance of the Q6 model is almost identical to that of the BF model for FPRs down to 0.1%. The QE and QB models perform slightly worse, with AUC scores 0.02 points lower than for Q6 and BF. For a fixed FPR of 1%, the TPR for BF/Q6 is 60% and is 55% for QE/QB. No notable degradation at very low FPR, where typical trigger algorithms would be operated, is observed.

With AutoQKeras, we give the user full flexibility to optimize the quantization configuration for a given use-case. An estimate of the model size and energy consumption can be computed using QTools and the user can then proceed by instructing AutoQKeras as to how much energy or bits it is desirable to save, given a certain accuracy-drop tolerance. Going from a pre-defined Keras model to an optimally quantized version (based on available resources) that is ready for chip implementation is made extremely simple through these libraries.

The final, crucial step in this process is to take these quantized models and make it simple to deploy them in the trigger system FPGAs (or any hardware) while making sure the circuit layout is optimal for the ultralow-latency constraint. We will address this in the following section.

Ultralow-latency, quantized model on FPGA hardware

To achieve ultralow-latency inference of QKeras models on FPGA firmware, we introduce full integration of QKeras layers in the hls4ml library. The libraries, together, provide a streamlined process for bringing quantized Keras models into particle detector triggering systems, while staying within the strict latency and resource constraints and performing high-accuracy inference.

When converting a QKeras model to an HLS project, the model quantization configuration is passed to hls4ml and enforced on the FPGA firmware. This ensures that the use of specific, arbitrary precision in the QKeras model is maintained during inference. For example, when using a quantizer with a given alpha parameter (that is, scaled weights), hls4ml inserts an operation to rescale the layer output. For binary and ternary weights and activations, the same strategies as in ref. [43] are used. With binary layers, the arithmetical value of −1 is encoded as 0, allowing the product to be expressed as an XNOR operation. The full workflow starting from a baseline TensorFlow Keras model and up until FPGA firmware generation is shown in Fig. 2. This illustrates how, through two simple steps, Keras models can be translated into ultra-compressed, highly parallel FPGA firmware.

We now compare the accuracy, latency and resource consumption of the different models derived so far: the BF, BP and BH models derived without using QKeras, two models optimized using AutoQKeras minimizing the model energy consumption (QE) and model bit consumption (QB), as well as a range of homogeneously quantized QKeras models scanning bit-widths from 3 to 16. Each model is trained using QKeras version 0.7.4, translated into

Table 3 | Performance on a Xilinx VU9P FPGA (2), showing model accuracy, latency, resource utilization and relative energy estimate for six different models

Model	Accuracy (%)	Latency[c] (ns)	Latency (clock cycles)	DSP (%)	LUT (%)	FF (%)	$\frac{E_{QK}}{E_{QK}(Q6)}$	$\frac{P_{HLS}}{P_{HLS}(Q6)}$
BF	74.4	45	9	56.0 (1,826)	5.2 (48,321)	0.8 (20,132)	-	-
BP	74.8	70	14	7.7 (526)	1.5 (17,577)	0.4 (10,548)	-	-
BH	73.2	70	14	1.3 (88)	1.3 (15,802)	0.3 (8,108)	-	-
Q6	74.8	55	11	1.8 (124)	3.4 (39,782)	0.3 (8,128)	1.00	1.00
QE	72.3	55	11	**1.0 (66)**	**0.8 (9,149)**	0.1 (1,781)	0.27	0.30
QB	71.9	70	14	1.0 (69)	0.9 (11,193)	0.1 (1,771)	0.25	0.25
LogicNets JSC-M[47]	70.6	NA[a]	NA	0 (0)	1.2 (14,428)	0.02 (440)	-	-
LogicNets JSC-L[47]	71.8	13[b]	5	0 (0)	3.2 (37,931)	0.03 (810)	-	-

[a]Not evaluated. [b]Using a clock frequency of 384 MHz. [c]The latency is evaluated for a clock cycle of 200 MHz. Resources are listed as percentage of total, with absolute numbers quoted in parentheses. The energy is estimated relative to the Q6 model and correspond to the relative energy computed using QTools (second to last column) and the relative power estimate from the post place-and-route report from Vivado (last column).

firmware using hls4ml version 0.2.1, and then synthesized with Vivado HLS (2019.2), targeting a Xilinx Virtex Ultrascale 9+ FPGA with a clock frequency of 200 MHz. We compare the resource consumption and latency on chip for each model, to the model accuracy. The resources at disposal on the FPGA are DSPs, LUTs, block random access memory (BRAM) and flip-flops. In this case, the BRAM is only used as a LUT read-only memory for calculating the final softmax function and is the same for all models, namely 1.5 units, corresponding to a total of 54 kb. For larger NNs using a higher reuse factor and longer latency, BRAM may also be used to store model weights. The estimated resource consumption and latency from logic synthesis, together with the model accuracy, are listed in Table 3. A fully parallel implementation is used, with an initiation interval—the number of clock cycles between new data inputs—of 1 in all cases. Resource utilization is quoted in the percentage of total available resources, with absolute numbers quoted in parentheses.

The most resource-efficient model is the AutoQKeras QE model, reducing the DSP usage by ~98%, LUT usage by ~80% and flip-flop usage by ~90%. The accuracy drop is less than 3%, despite using half the number of neurons per layer and the overall lower precision. The extreme reduction of DSP utilization is especially interesting as, on the FPGA, DSPs are scarce and usually become the critical resource for ML applications. DSPs are used for all MAC operations, but, if the precision of the incoming numbers is much lower than the DSP precision (which, in this case, is 18 bits) MAC operations are moved to LUTs. This is an advantage, as a representative FPGA for the LHC trigger system has $\mathcal{O}(10^3)$ DSPs compared to $\mathcal{O}(10^6)$ LUTs. If the bulk of multiplication operations is moved to LUTs, this allows for deeper and more complex models to be implemented. In our case, the critical resource reduces from 56% of DSPs for the baseline to 3.4% of LUTs for the 6-bit QKeras trained model with the same accuracy. The latency is $\mathcal{O}(10)$ ns for all models.

In the final two columns of Table 3, we compare the relative energy estimation from QTools with the post place-and-route power report from Vivado for the three QKeras models, in both cases relative to the Q6 model. Because the target clock frequency and model initiation interval are identical across these models, the inference rate is the same and taking the ratio of the power is equivalent to taking the ratio of the energy. Very good agreement between the QTools relative energy estimates and the Vivado relative power estimates is observed for the QE and QB models, and the energy ordering is the same for all models.

We compare the results obtained using the QKeras and hls4ml workflow to LogicNets[47], another work on extreme low-latency, low-resource, fully unfolded (initiation interval = 1) FPGA implementations. The metrics are those quoted in Table 3. Two LogicNets models have been evaluated: one using the same architecture as in this Article, JSC-M and another using a larger architecture (32, 64, 192, 192, 16 numbers of neurons), JSC-L. For JSC-M, an accuracy of 70.6% is quoted, 1.7 points lower than the most resource-efficient model using QKeras and hls4ml, QE. In addition, QE uses 1.2× fewer LUTs than JSC-M. No DSPs are used in LogicNets, compared to the 66 DSPs in use by the QE model.

The latency has only been evaluated for JSC-L and is quoted to be 13 ns, using a clock frequency of 384 MHz. The final softmax function has been removed from this estimate. In high-energy physics experiments, the final softmax layer is crucial because trigger thresholds are usually set based on an algorithm's FPR. The threshold on the FPR is usually set as high as the trigger bandwidth allows, maximizing the TPR while staying within the bandwith-budget.

For a clock period of 5 ns, the QE model has a latency of 55 ns, reduced to 45 ns when ignoring the final softmax layer. The JSC-L model has a latency of 13 ns for a clock period of 2.6 ns.

Finally, we compare the accuracy and resource consumption of a range of homogeneously quantized QKeras models, scanning bit-widths from 3 to 16. In Fig. 3 (left) the accuracy relative to the baseline model evaluated with floating-point precision is shown as a function of bit-width. This is shown for the accuracy as evaluated offline using TensorFlow QKeras (green line) and the accuracy as evaluated on the FPGA (orange line). We compare this to the performance achievable using the baseline model and post-training quantization (purple dashed line). The markers represent the accuracy of the baseline, baseline pruned, baseline heterogeneous and AutoQKeras optimized models (again emphasizing that the AutoQKeras models use half as many neurons per layer as the baseline Keras model). Models trained with QKeras retain performance very close to the baseline using as few as 6 bits for all weights, biases and activations. Accuracy degrades slightly down to 98% of the baseline accuracy at a precision of 3 bits.

Post-training homogeneous quantization of the baseline model shows a much more notable accuracy loss, with accuracy rapidly falling away below 14 bits. The model resource utilization as a function of bit-width for homogeneously quantized QKeras models is shown in the right plot in Fig. 3. The switch from DSPs to LUTs mentioned above is clearly visible: below a bit-width of ~10, MAC operations are moved from the DSPs to the LUTs and the critical resource consumption is considerably reduced. For example, in this case, using a model quantized to 6-bit precision will maintain the same accuracy while reducing resource consumption by ~70%. The symbols in Fig. 3 show the resource consumption of the heterogeneously quantized models. The only model comparable in accuracy and resource consumption to the AutoQKeras optimized models,

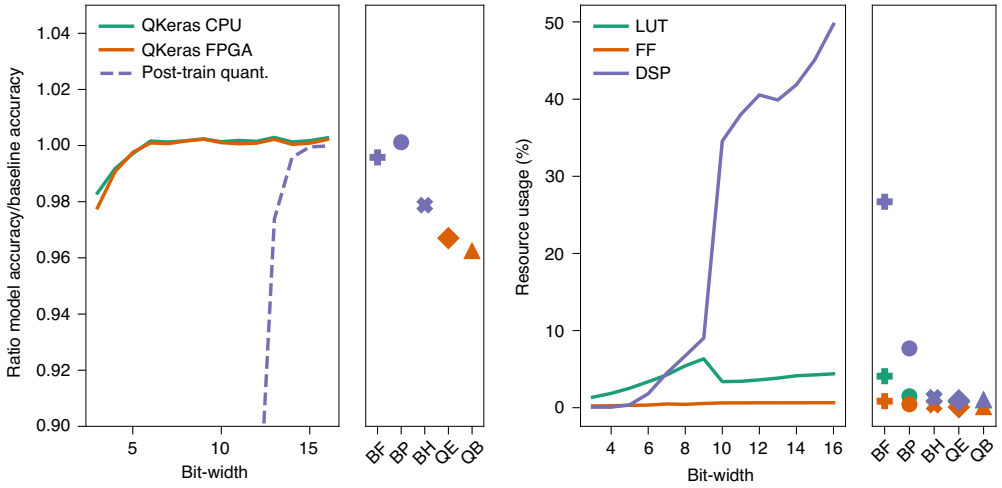


Fig. 3 | Performance on a Xilinx VU9P FPGA. Relative accuracy (left) and resource utilization (right) as a function of bit-width. The right-hand panel shows the metrics for the heterogeneously quantized models. The relative accuracy is evaluated with respect to the floating-point baseline model. Resources are expressed as a percentage of the targeted FPGA, a Xilinx VU9P.

QE and QB, is the baseline heterogeneous (BH). However, in contrast to the QKeras models, BH has been pruned to a weight sparsity of 70%, which further reduces the resource consumption (all zero multiplications are removed). In addition, the process of manually quantizing a model post-training is time-consuming and cumbersome, and not guaranteed to always succeed due to its lossy nature. AutoQKeras and hls4ml allow us to quantize automatically through quantization-aware training, with specific tolerances in terms of accuracy and area, greatly simplifying the process.

In ref. [69], the QKeras and hls4ml workflow has been demonstrated on convolutional architectures benchmarked on the Streetview House Numbers dataset[70], both on large FPGAs and small system-on-chip FPGAs. High accuracy matching the floating-point model accuracy can be maintained down to 6-bit precision with QKeras, executed with a latency of 5 µs. For larger convolutional architectures like ResNet[66], hls4ml does not scale due to the very low latency target and the fully on-chip implementation used to obtain this. Our main application is the efficient implementation of smaller, custom models targeting latencies of $\mathcal{O}(10)$ ns to $\mathcal{O}(1)$ µs.

Conclusion and future work

We have introduced a novel library, QKeras, providing a simple method for uncovering optimally heterogeneously quantized DNNs for a set of given resource or accuracy constraints. Through simple replacement of Keras layers, models with heterogeneous per-layer, per-parameter type precision, chosen from a wide range of novel quantizers, can be defined and trained quantization-aware. A model optimization algorithm that considers both model area and accuracy is presented, allowing users to maximize the model performance given a set of resource constraints, crucial for high-performance inference on edge. Support for these quantized models has been implemented in hls4ml, providing the necessary chip layout instruction components to enable ultrafast inference of these tiny-footprint models on a chip. We have demonstrated how on-chip resource consumption can be reduced by a factor of 50 without much loss in model accuracy while performing inference within $\mathcal{O}(10)$ ns. The methods presented here provide crucial tools for inference on the extreme low-area and low-latency edge, like that in particle detectors where a latency of $\mathcal{O}(1)$ µs is enforced. Taking a pre-trained model and making it suitable for hardware implementation on the edge, both in terms of latency and size, is one of the bottlenecks for bringing ML applications into extremely constrained computing environments (for example, a detector at a particle collider), and the workflow presented here will allow for a streamlined and simple process, ultimately resulting in a great improvement in the quality of physics data collected in the future.

The generality and flexibility of the QKeras+hls4ml workflow opens up a wide array of possible future work. This includes integration with other quantization libraries targeting non-FPGA hardware, such as TensorFlow Lite, as well as those targeting FPGA synthesis, such as FINN (and the quantization library Brevitas) and HAQ. In addition, while the energy estimator provides a good baseline for relative energy consumption, as demonstrated, we hope to extend the library to provide more device-specific absolute energy estimates. We also plan to explore using a combination of block energy and the curvature of the weight space, as done in HAQ, when quantizing a network one block at a time. Finally, work is ongoing to use the QKeras+hls4ml workflow to deploy ML algorithms for the next data-taking period at CERN LHC both on FPGAs and ASICs.

Methods

Additional layers, quantizers and methods in QKeras. In this section, we will give an overview of the available layers, quantizers and methods in QKeras. A summary of available layers in QKeras is listed in Extended Data Fig. 3.

For several quantizers (including quantized_bits), a parameter called keep_negative can be set.

If keep_negative is true, negative numbers are not clipped. With a lower number of bits, the rounding adds more bias to the number system. Reference [71] suggested using stochastic rounding, which uses the fractional part of the number as a probability to round the number up or down.

Stochastic rounding for quantized_bits quantizers can be turned on by setting use_stochastic_rounding=True. However, when an efficient hardware or software implementation is considered, this flag should be avoided in activation functions as it may affect the implementation efficiency.

Activations have been migrated to QActivation, but activation parameters passed directly in convolutional and dense layers will be recognized as well.

The bernoulli and stochastic functions rely on stochastic versions of the activation functions, so they are best suited for weights and biases. They draw a random number with uniform distribution from sigmoid of the input x, adding additional regularization. The result is based on the expected value of the activation function. The temperature parameter determines the steepness of the sigmoid function.

The quantizers quantized_relu and quantized_tanh are quantized versions of ReLU[61] and tanh functions, respectively.

The quantized_po2 and quantized_relu_po2 quantizers perform exponent quantization, as defined in ref. [72]. The main advantage of this quantizer is that it provides a representation that is very efficient for multiplication. The parameter max_value defines maximum value.

It should also be noted that the QSeparableConv2D layer is implemented as a depthwise, followed by pointwise quantized expansions, which is an extended form of the SeparableConv2D implementation of MobileNet[73]. The reason we chose to use this version is that MobileNet's SeparableConv2D has an activation between the depthwise convolution and the pointwise convolution, where we need to at least apply some form of quantization.

Besides the drop-in replacement of Keras layers, we have written a few utility functions.

The model_quantize function converts a non-quantized model into a quantized version, by applying a specified configuration for layers and activations. The method model_save_quantized_weights saves the quantized weights in the model compatible with an inference or writes the quantized weights in the file filename for production. The method load_qmodel loads and compiles the quantized Keras model. The methods print_model_sparsity and print_qstats print sparsity for the pruned layers in the model and statistics of the number of operations per operation type and layer. Meanwhile, quantized_model_debug allows for debugging and plotting model weights and activations. Finally, extract_model_operations estimates which operations are required for each layer of the quantized model, for example xor, mult, adder and so on.

Variance shift handling in QKeras. A critical aspect when training quantized versions of tensors and trainable parameters is the variance shift. During training with very few bits, the variance may shift a lot from its initialization. With popular initialization methods, such as glorot_normal, during the initial steps of the training, all of the output tensors will become zero. Consequently, the network will not be trained. For example, in a VGG network[74], the fully connected layers have 4,096 elements, and any quantized representation with fewer than 6 bits will turn the output of these layers to 0, as $\log_2(\sqrt{(4,096)}) = 6$. For layer i and minimum quantization threshold Δ, the weights w_i are quantized by quantizer(w_i) operation. When the gradient is computed, the quantized weights will appear as a result of the chain rule computation, as depicted in Extended Data Fig. 2. With the absolute values of all weights below Δ, the gradient will vanish in all layers that transitively generate the inputs to layer i. This applies to any large DNN.

QKeras mitigates this challenge by rescaling the initialized weights appropriately. The parameter alpha is used as a scaling factor. It can be considered as a way to compute a shared exponent when used in weights[75]. It can be set to a given value manually, or overridden by setting it to auto or auto_po2. With alpha = 'auto', we compute the scale as $\sum q(x)x/\sum q(x)q(x)$ as in ref. [24] for the quantization function q, with a different value for each output channel or output dimension of tensor x. This provides a learned scaling factor that can be used during training. With alpha = 'auto_po2'[19], the scaling factor is set to be a power-of-2 number.

For the ternary and stochastic_ternary quantizers, we iterate between scale computation and threshold computation, as presented in ref. [76], which searches for the threshold and scale tolerant to different input distributions. This is especially important when we need to consider that the threshold shifts depending on the input distribution, affecting the scale as well, as pointed out by ref. [77]. When computing the scale in these quantizers with alpha = 'auto', we compute the scale as a floating-point number. With alpha = 'auto_po2', we enforce the scale to be a power of 2, meaning that an actual hardware or software implementation can be performed by just shifting the result of the convolution or dense layer to the right or left by checking the sign of the scale (positive shifts left, negative shifts right), and taking the $\log_2$ of the scale. This behaviour is compatible with shared exponent approaches, as it performs a shift adjustment to the channel.

Data availability

The data used in this study are openly available at Zenodo[58] from https://doi.org/10.5281/zenodo.3602260.

Code availability

The QKeras library, which also includes AutoQKeras and QTools, is available from https://github.com/google/qkeras (the work presented here uses QKeras version 0.7.4). Examples on how to run the library are available in the notebook subdirectory. The hls4ml library is available at https://github.com/fastmachinelearning/hls4ml and all versions ≥0.2.1 support QKeras models (the work presented here is based on version 0.2.1). For examples on how to use QKeras models in hls4ml, the notebook part4_quantization at https://github.com/fastmachinelearning/hls4ml-tutorial serves as a general introduction.

Received: 23 November 2020; Accepted: 6 May 2021;
Published online: 21 June 2021

References

1. Lin, S.-C. et al. The architectural implications of autonomous driving: constraints and acceleration. *ACM SIGPLAN Notices* **53**, 751–766 (2018).
2. Ignatov, A. et al. AI benchmark: running deep neural networks on Android smartphones. In *Computer Vision – ECCV 2018 Workshops. ECCV 2018 Lecture Notes in Computer Science* Vol. 11133 (eds Leal-Taixé, L. & Roth, S.) 288–314 (Springer, 2018); https://doi.org/10.1007/978-3-030-11021-5_19
3. Leber, C., Geib, B. & Litz, H. High frequency trading acceleration using FPGAs. In *2011 21st International Conference on Field Programmable Logic and Applications* 317–322 (IEEE, 2011).
4. The LHC Study Group. *The Large Hadron Collider, Conceptual Design*. Technical Report CERN/AC/95-05 (CERN, 1995).
5. Apollinari, G., Béjar Alonso, I., Brüning, O., Lamont, M. & Rossi, L. *High-Luminosity Large Hadron Collider (HL-LHC): Preliminary Design Report*. Technical Report (Fermi National Accelerator Laboratory, 2015).
6. Iandola, F. N. et al. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5-MB model size. Preprint at https://arxiv.org/pdf/1602.07360.pdf (2016).
7. Howard, A. G. et al. MobileNets: efficient convolutional neural networks for mobile vision applications. Preprint at https://arxiv.org/pdf/1704.04861.pdf (2017).
8. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. & Chen, L.-C. MobileNetV2: inverted residuals and linear bottlenecks. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition* 4510–4520 (IEEE, 2018).
9. Ma, N., Zhang, X., Zheng, H.-T. & Sun, J. ShuffleNet V2: practical guidelines for efficient CNN architecture design. In *Proc. European Conference on Computer Vision (ECCV) Lecture Notes in Computer Science* 116–131 (Springer, 2018).
10. Howard, A. et al. Searching for MobileNetV3. In *Proc. IEEE International Conference on Computer Vision* 1314–1324 (IEEE, 2019).
11. Ding, X. et al. Global sparse momentum SGD for pruning very deep neural networks. In *Advances in Neural Information Processing Systems* (eds Wallach, H. et al.) 6382–6394 (NIPS, 2019).
12. He, Y., Zhang, X. & Sun, J. Channel pruning for accelerating very deep neural networks. In *Proc. IEEE International Conference on Computer Vision* 1389–1397 (IEEE, 2017).
13. Duarte, J. et al. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum.* **13**, P07027 (2018).
14. Nagel, M., van Baalen, M., Blankevoort, T. & Welling, M. Data-free quantization through weight equalization and bias correction. In *Proc. IEEE International Conference on Computer Vision* 1325–1334 (IEEE, 2019).
15. Meller, E., Finkelstein, A., Almog, U. & Grobman, M. Same, same but different: recovering neural network quantization error through weight factorization. In *Proc. 36th International Conference on Machine Learning* (eds Chaudhuri, K. & Salakhutdinov, R.) 4486–4495 (PMLR, 2019).
16. Zhao, R., Hu, Y., Dotzel, J., De Sa, C. & Zhang, Z. Improving neural network quantization without retraining using outlier channel splitting. Preprint at https://arxiv.org/pdf/1901.09504.pdf (2019).
17. Banner, R., Nahshan, Y. & Soudry, D. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems* (eds Wallach, H. et al.) 7950–7958 (NIPS, 2019).
18. Moons, B., Goetschalckx, K., Van Berckelaer, N. & Verhelst, M. Minimum energy quantized neural networks. In *51st Asilomar Conference on Signals, Systems, and Computers* 1921–192 (ACSSC, 2017).
19. Courbariaux, M., Bengio, Y. & David, J.-P. BinaryConnect: training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems 28* (eds Cortes, C. et al.) 3123–3131 (Curran Associates, 2015).
20. Zhang, D., Yang, J., Ye, D. & Hua, G. LQ-Nets: learned quantization for highly accurate and compact deep neural networks. In *Proc. European Conference on Computer Vision (ECCV)* 365–382 (Springer, 2018).
21. Li, F. & Liu, B. Ternary weight networks. Preprint at https://arxiv.org/pdf/1605.04711.pdf (2016).
22. Zhou, S. et al. DoReFa-Net: training low bitwidth convolutional neural networks with low bitwidth gradients. Preprint at https://arxiv.org/pdf/1606.06160.pdf (2016).
23. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R. & Bengio, Y. Quantized neural networks: training neural networks with low precision weights and activations. *J. Mach. Learn. Res.* **18**, 6869–6898 (2017).
24. Rastegari, M., Ordonez, V., Redmon, J. & Farhadi, A. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *Computer Vision – ECCV 2016. Lecture Notes in Computer Science* Vol. 9908 (eds Leibe, B. et al.) 525–542 (Springer, 2016).
25. Micikevicius, P. et al. Mixed precision training. In *International Conference on Learning Representations* (ICLR, 2018).
26. Zhuang, B., Shen, C., Tan, M., Liu, L. & Reid, I. Towards effective low-bitwidth convolutional neural networks. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* 7920–7928 (IEEE, 2018).
27. Wang, N., Choi, J., Brand, D., Chen, C.-Y. & Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems* 7675–7684 (NIPS, 2018).

28. Wang, K., Liu, Z., Lin, Y., Lin, J. & Han, S. HAQ: hardware-aware automated quantization with mixed precision. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (IEEE, 2019).
29. Dong, Z., Yao, Z., Gholami, A., Mahoney, M. & Keutzer, K. HAWQ: Hessian AWare Quantization of neural networks with mixed-precision. In *Proc. 2019 IEEE/CVF International Conference on Computer Vision* (*ICCV*) 293–302 (IEEE, 2019).
30. Dong, Z. et al. HAWQ-V2: Hessian AWare trace-weighted Quantization of neural networks. In *Advances in Neural Information Processing Systems* Vol. 33 (eds Larochelle, H. et al.) 18518–18529 (Curran Associates, 2020).
31. Wu, B. et al. Mixed precision quantization of ConvNets via differentiable neural architecture search. Preprint at https://arxiv.org/pdf/1812.00090.pdf (2018).
32. Chollet, F. et al. *Keras* https://github.com/fchollet/keras (2015).
33. Abadi, M. et al. *TensorFlow*: *Large-Scale Machine Learning on Heterogeneous Systems* http://tensorflow.org/ (2015).
34. Paszke, A. et al. PyTorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32* (eds Wallach, H. et al.) 8024 (Curran Associates, 2019); https://arxiv.org/pdf/1912.01703.pdf
35. Open Neural Network Exchange Collaboration https://onnx.ai/ (2017).
36. Venieris, S. I., Kouris, A. & Bouganis, C.-S. Toolflows for mapping convolutional neural networks on FPGAs: a survey and future directions. *ACM Comput. Surv.* **51**, 56 (2018).
37. Guo, K., Zeng, S., Yu, J., Wang, Y. & Yang, H. A survey of FPGA-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.* **12**, 2 (2018).
38. Shawahna, A., Sait, S. M. & El-Maleh, A. FPGA-based accelerators of deep learning networks for learning and classification: a review. *IEEE Access* **7**, 7823–7859 (2019).
39. Abdelouahab, K., Pelcat, M., Serot, J. & Berry, F. Accelerating CNN inference on FPGAs: a survey. Preprint at https://arxiv.org/pdf/1806.01683.pdf (2018).
40. Intel. *Intel High Level Synthesis Compiler* https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html (2020).
41. Mentor/Siemens. *Catapult High-Level Synthesis* https://www.mentor.com/hls-lp/catapult-high-level-synthesis (2020).
42. Iiyama, Y. et al. Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction in high energy physics. *Front. Big Data* **3**, 598927 (2021).
43. Ngadiuba, J. et al. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. *Mach. Learn. Sci. Technol.* **2**, 015001 (2020).
44. Umuroglu, Y. et al. FINN: a framework for fast, scalable binarized neural network inference. In *Proc. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* 65–74 (ACM, 2017).
45. Blott, M. et al. FINN-R: an end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfigurable Technol. Syst.* **11**, 16 (2018).
46. Alessandro, F. G. & Nickfraser, U. Y. *Xilinx/brevitas*: *Release version 0.2.1* https://doi.org/10.5281/zenodo.4507794 (2021).
47. Umuroglu, Y., Akhauri, Y., Fraser, N. J. & Blott, M. LogicNets: co-designed neural networks and circuits for extreme-throughput applications. In *30th International Conference on Field-Programmable Logic and Applications* 291–297 (IEEE, 2020).
48. Guan, Y. et al. FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines* (*FCCM*) 152 (IEEE, 2017).
49. Sharma, H. et al. From high-level deep neural models to FPGAs. In *Proc. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture* 1 (IEEE, 2016); https://doi.org/10.1109/MICRO.2016.7783720
50. Gokhale, V., Zaidy, A., Chang, A. X. M. & Culurciello, E. Snowflake: an efficient hardware accelerator for convolutional neural networks. In *Proc. 2017 IEEE International Symposium on Circuits and Systems* (*ISCAS*) 1–4 (IEEE, 2017).
51. Venieris, S. I. & Bouganis, C.-S. fpgaConvNet: a toolflow for mapping diverse convolutional neural networks on embedded FPGAs. In *Proc. NIPS 2017 Workshop on Machine Learning on the Phone and other Consumer Devices* (NIPS, 2017); https://arxiv.org/pdf/1711.08740.pdf
52. Venieris, S. I. & Bouganis, C.-S. fpgaConvNet: automated mapping of convolutional neural networks on FPGAs. In *Proc. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* 291 (ACM, 2017).
53. Venieris, S. I. & Bouganis, C.-S. fpgaConvNet: a framework for mapping convolutional neural networks on FPGAs. In *Proc. 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines* (*FCCM*) 40 (IEEE, 2016).
54. Huimin Li et al. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *Proc. 2016 26th International Conference on Field Programmable Logic and Applications* (*FPL*) 1–9 (IEEE, 2016).
55. Zhao, R. et al. Hardware compilation of deep neural networks: an overview. In *2018 IEEE 29th International Conference on Application-Specific Systems, Architectures and Processors* (*ASAP*) 1–8 (IEEE, 2018).
56. Google. *TensorFlow Lite* https://www.tensorflow.org/lite (2020).
57. Moreno, E. A. et al. JEDI-net: a jet identification algorithm based on interaction networks. *Eur. Phys. J. C* **80**, 58 (2019).
58. Pierini, M., Duarte, J. M., Tran, N. & Freytsis, M. *HLS4ML LHC Jet Dataset* (*150 particles*) https://doi.org/10.5281/zenodo.3602260 (2020).
59. Zhu, M. & Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. Preprint at https://arxiv.org/pdf/1710.01878.pdf (2017).
60. Coelho, C. *Qkeras* https://github.com/google/qkeras (2019).
61. Nair, V. & Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proc. 27th International Conference on International Conference on Machine Learning* 807–814 (ICML, 2010).
62. Hennessy, J. L. & et al. *Computer Architecture*: *a Quantitative Approach* 6th edn (Morgan Kaufmann, 2016).
63. Horowitz, M. Computing's energy problem (and what we can do about it). In *Proc. 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers* (*ISSCC*) 10–14 (IEEE, 2014).
64. O'Malley, T. et al. *Keras Tuner* https://github.com/keras-team/keras-tuner (2019).
65. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A. & Talwalkar, A. Hyperband: a novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.* **18**, 6765–6816 (2017).
66. He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* 770–778 (IEEE, 2016).
67. Goodfellow, I., Bengio, Y. & Courville, A. *Deep Learning* (MIT Press, 2016).
68. Kingma, D. P. & Ba, J. Adam: a method for stochastic optimization. In *3rd International Conference on Learning Representations* (*ICLR*) *2015*, *Conference Track Proceedings* (eds Bengio, Y. & LeCun, Y.) (ICLR, 2015); https://arxiv.org/pdf/1412.6980.pdf
69. Aarrestad, T. et al. Fast convolutional neural networks on FPGAs with hls4ml. Preprint at https://arxiv.org/pdf/2101.05108.pdf (2021).
70. Netzer, Y. et al. Reading digits in natural images with unsupervised feature learning. In *Proc. NIPS 2011 Workshop on Deep Learning and Unsupervised Feature Learning* (NIPS, 2011); https://deeplearningworkshopnips2011.files.wordpress.com/2011/12/12.pdf
71. Gupta, S., Agrawal, A., Gopalakrishnan, K. & Narayanan, P. Deep learning with limited numerical precision. In *Proc. 32nd International Conference on Machine Learning* 1737–1746 (PMLR, 2015).
72. Kwan, H. K. & Tang, C. Z. A design method for multilayer feedforward neural networks for simple hardware implementation. In *Proc. 1993 IEEE International Symposium on Circuits and Systems* Vol. 4, 2363–2366 (IEEE, 1993).
73. Howard, A. G. et al. MobileNets: efficient convolutional neural networks for mobile vision applications. Preprint at https://arxiv.org/pdf/1704.04861.pdf (2017).
74. Simonyan, K. & Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations* (eds Bengio, Y. & LeCun, Y.) (ICLR, 2015); https://arxiv.org/pdf/1409.1556.pdf
75. Das, D. et al. Mixed precision training of convolutional neural networks using integer operations. In *International Conference on Learning Representations* (ICLR, 2018).
76. Hwang, K. & Sung, W. Fixed-point feedforward deep neural network design using weights +1, 0, and −1. In *Proc. 2014 IEEE Workshop on Signal Processing Systems* (*SiPS*) 1–6 (IEEE, 2014).
77. Li, F., Zhang, B. & Liu, B. Ternary weight networks. Preprint at https://arxiv.org/pdf/1605.04711.pdf (2016).

Acknowledgements

M.P. and S.S. are supported by, and V.L. and A.A.P. are partially supported by, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant no. 772369). V.L. is supported by Zenseact under the CERN Knowledge Transfer Group. A.A.P. is supported by CEVA under the CERN Knowledge Transfer Group. We acknowledge the Fast Machine Learning collective as an open community of multi-domain experts and collaborators. This community was important for the development of this project.

Author contributions

C.N.C., A.K., S.L. and H.Z. conceived and designed the QKeras, AutoQKeras and QTools software libraries. T.A., V.L., M.P., A.A.P., S.S. and J.N. designed and implemented

support for QKeras in hls4ml. S.S. conducted the experiments. T.A., A.A.P. and S.S. wrote the manuscript.

Competing interests
The authors declare no competing interests.

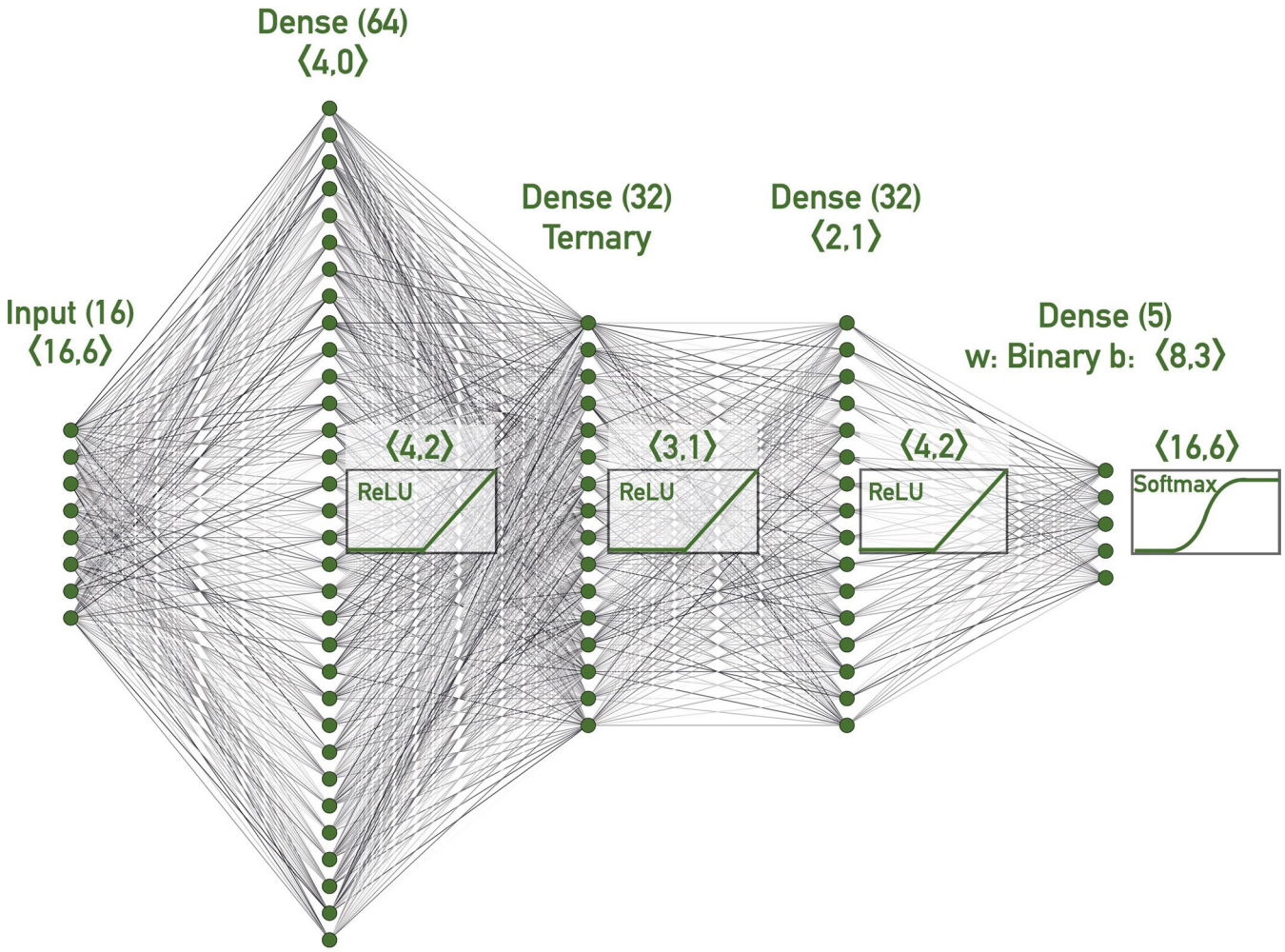
Additional information
Extended data is available for this paper at https://doi.org/10.1038/s42256-021-00356-5.

Correspondence and requests for materials should be addressed to T.K.A.

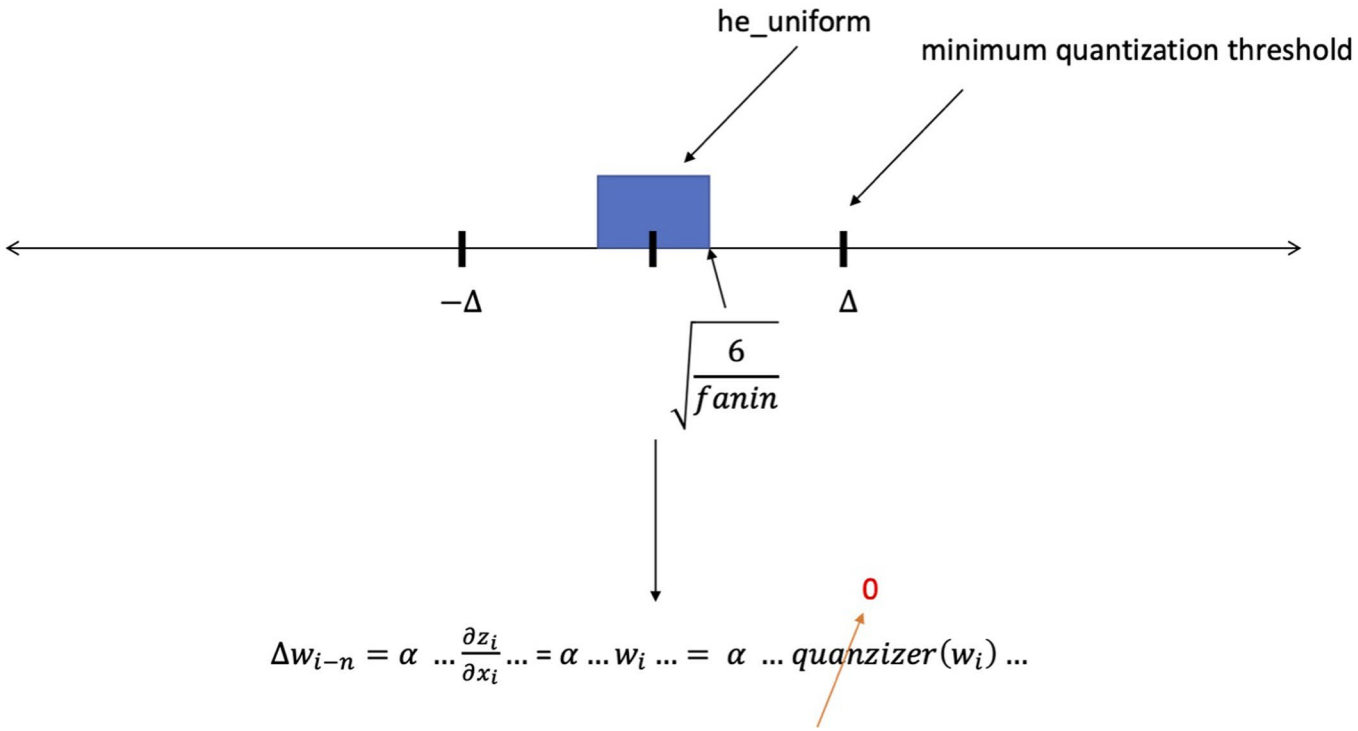
Peer review information *Nature Machine Intelligence* thanks Jose Nunez-Yanez, Stylianos Venieris and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



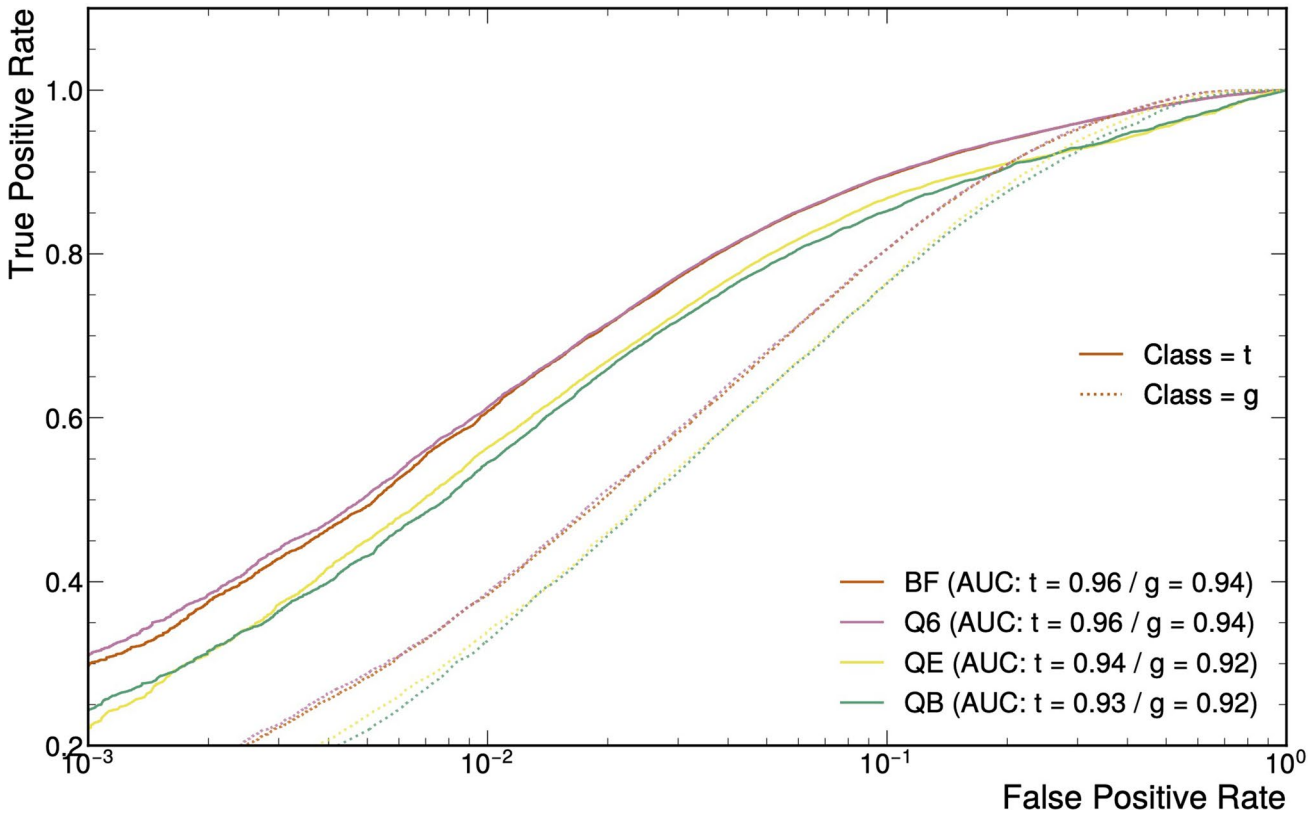
Extended Data Fig. 1 | Model architecture and quantization. Model architecture for the fully-connected NN architecture under study. The numbers in brackets are the precisions used for each layer, quoted as $\langle B, I \rangle$, where B is the precision in bits and I the number of integer bits. When different precision is used for weights and biases, the quantization is listed as w and b, respectively. These have been obtained using the per-layer, per-parameter type automatic quantization procedure described in Section VI.



Extended Data Fig. 2 | Variance shift. Variance shift and the effect of initialization in gradient descent.

Layers	Quantizers
QDense,	quantized_bits,
QConv1D,	binary,
QConv2D,	ternary,
QDepthwiseConv2D,	bernoulli,
QSeparableConv2D,	stochastic_ternary,
QActivation,	stochastic_binary,
QAveragePooling2D,	smooth_sigmoid,
QBatchNormalization,	hard_sigmoid,
QOctaveConv2D,	binary_sigmoid,
QSimpleRNN,	smooth_tanh,
QLSTM,	hard_tanh,
QGRU	binary_tanh,
	quantized_relu,
	quantized_ulaw,
	quantized_tanh,
	quantized_po2,
	quantized_relu_po2

Extended Data Fig. 3 | Layers and quantisers in QKeras. List of available layers and quantizers in QKeras.



Extended Data Fig. 4 | ROC curves for the models under study. ROC curves of false positive rate (FPR) versus true positive rate (TPR) for the Baseline Full (BF), quantized 6-bit (Q6), AutoQKeras Energy Optimized (QE) and AutoQKeras Bits Optimized (QB) models.