FERMILAB-PUB-20-167-PPD-SCD

Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml

Jennifer Ngadiuba, Vladimir Loncar [*], Maurizio Pierini, Sioni Summers
European Organization for Nuclear Research (CERN)
CH-1211 Geneva 23, Switzerland

Giuseppe Di Guglielmo
Columbia University
New York, NY 10027, USA

Javier Duarte
University of California San Diego
La Jolla, CA 92093, USA

Philip Harris, Dylan Rankin
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

Sergo Jindariani, Mia Liu, Kevin Pedro, Nhan Tran
Fermi National Accelerator Laboratory
Batavia, IL 60510, USA

Edward Kreinar
HawkEye360
Herndon, VA 20170, USA

Sheila Sagear
Boston University
Boston, MA 02215, USA

Zhenbin Wu
University of Illinois at Chicago
Chicago, IL 60607, USA

Duc Hoang
Rhodes College
Memphis, TN 38112, USA

March 16, 2020

Abstract

We present the implementation of binary and ternary neural networks in the `hls4ml` library, designed to automatically convert deep neural network models to digital circuits with FPGA firmware. Starting from benchmark models trained with floating point precision, we investigate different strategies to reduce the network's resource consumption by reducing the numerical precision of the network parameters to binary or ternary. We discuss the trade-off between model accuracy and resource consumption. In addition, we show how to balance between latency and accuracy by retaining full precision on a selected subset of network components. As an example, we consider two multiclass classification tasks: handwritten digit recognition with the MNIST data set and jet identification with simulated proton-proton collisions at the CERN Large Hadron Collider. The binary and ternary implementation has similar performance to the higher precision implementation while using drastically fewer FPGA resources.

arXiv:2003.06308v1 [cs.LG] 11 Mar 2020

[*] Also at Institute of Physics Belgrade, Serbia.

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

1 Introduction

Field-programmable gate arrays (FPGAs) are an efficient and flexible processing solution to perform low latency and high bandwidth inference of deep neural networks (DNNs). Their design is extremely functional to parallelize the mathematical operations typical of DNN inference tasks, namely matrix multiplication and activation function application. FPGAs can be reprogrammed, which offers unquestionable advantages in terms of flexibility with respect to application-specific integrated circuits (ASICs). At the same time, they share some of the advantages offered by ASICs, such as low power consumption and speed.

Typically, FPGAs are used to emulate generic digitial circuits as a preliminary step toward the design of custom ASICs or as an alternative to them. For instance, hundreds of FPGAs are used as custom electronic logic to process in real time the proton-proton collisions at the CERN Large Hadron Collider (LHC). With beams colliding every 25 ns and thanks to a built-in buffering system, a typical LHC experiment has $\mathcal{O}(1)$ μs to decide whether to keep or discard a given event. This real-time decision-taking system, referred to as the "L1 trigger", consists of a set of digital circuits implementing physics-motivated rule-based selection algorithms. Currently, these algorithms are emulated on FPGAs, mounted on custom boards.

The severe L1 latency constraint prevents the LHC experimental collaborations from deploying complex rule-based algorithms on the L1 FPGA boards. Machine Learning (ML) solutions, and in particular DNNs, are currently being investigated as fast-to-execute and parallelisable approximations of rule-based algorithms. For instance, the CMS collaboration has deployed a Boosted Decision Trees in the L1 trigger electronic logic [1]. Following this approach, one could train a DNN to process a given input (e.g., energy deposits in a calorimeter) and return the output of an event reconstruction algorithm (e.g., to regress the energy of the incoming particle that caused these energy deposits or to identify its nature). Since the complexity of LHC collision events is going to increase, we expect this approach to become popular in the near future.

In order to facilitate the deployment of DNNs in the L1 triggers of high energy physics (HEP) experiments, we developed a software library, `hls4ml`, to convert a DNN model into FPGA firmware through an automatic workflow [2]. In HEP, the deployment of deep learning (DL) models on FPGAs has been discussed in the context of the online data-selection system of the LHC experiments. Alternative solutions based on VHDL [3] have been explored. Similar studies and comparable results have been shown in Ref. [4].

The `hls4ml` design is characterized by two aspects: (i) it relies on high-level synthesis (HLS) back-ends, in order to allow a fully automatized workflow from a trained model to an FPGA firmware; (ii) it is designed so that the final outcome is a fully-on-chip logic, which enables the latency to be within typical values of $\mathcal{O}(1)$ μs. Our ultimate goal is to support the most popular DNN model ingredients (layers, activation functions, etc.) and an interface to the most popular DL training libraries, directly (e.g., for `TensorFlow` [5], `Keras` [6], and `PyTorch` [7]) or through the `ONNX` [8] interface. The library is under development and many of these ingredients are already supported. While `hls4ml` was initially conceived for LHC applications, its potential use cases go well beyond HEP. In general, `hls4ml` provides a user-friendly interface to deploy custom DNN models on FPGAs, used as co-processing accelerators or as digital circuits in resource-constrained, low-latency computing environments.

The main challenge in deploying a DNN model on an FPGA is the limited computational resources. Typically, one would reuse resources for the inference operations across multiple clock cycles, at the price of a larger latency. The *reuse factor* quantifies how many times a resource is reused and is equal to the initiation interval (II) for that operation. A complementary approach consists of compressing the model, e.g., by reducing the number of operations needed in the inference step (pruning) or their cost (e.g., quantizing the network to a fixed point numerical representation). In a previous publication [2], we showed that pruning and quantization allow one to execute simple fully-connected DNN models with state-of-the-art performance on a specific LHC problem within a latency of $\mathcal{O}(100)$ ns, while using only a fraction of the FPGA resources. In this paper, we investigate how a similar result can be obtained with binary and ternary networks [9, 10, 11], following closely the studies presented in Refs. [9, 12, 13].

This paper is structured as follows: Section 2 introduces the benchmark problems and data sets. The implementation of binary and ternary networks in `hls4ml` is described in Section 3. Section 4 describes the different model architectures considered in this study, while their application to the two benchmark classification problems is discussed in Section 5. A summary and outlook is given in Section 6.

2 Benchmark models and data sets

We consider two benchmark classification tasks: a digit recognition task with the MNIST data set [14] and the LHC jet tagging task discussed in Ref. [2].

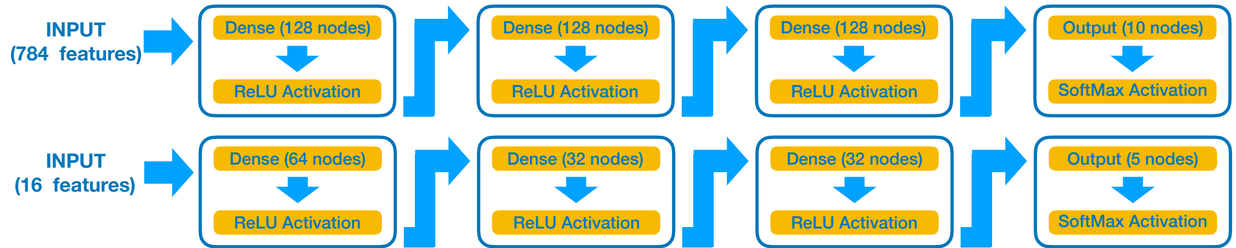

Figure 1: Network architecture for the baseline MNIST (top) and LHC jet (bottom) classifiers used as benchmark models in this study.

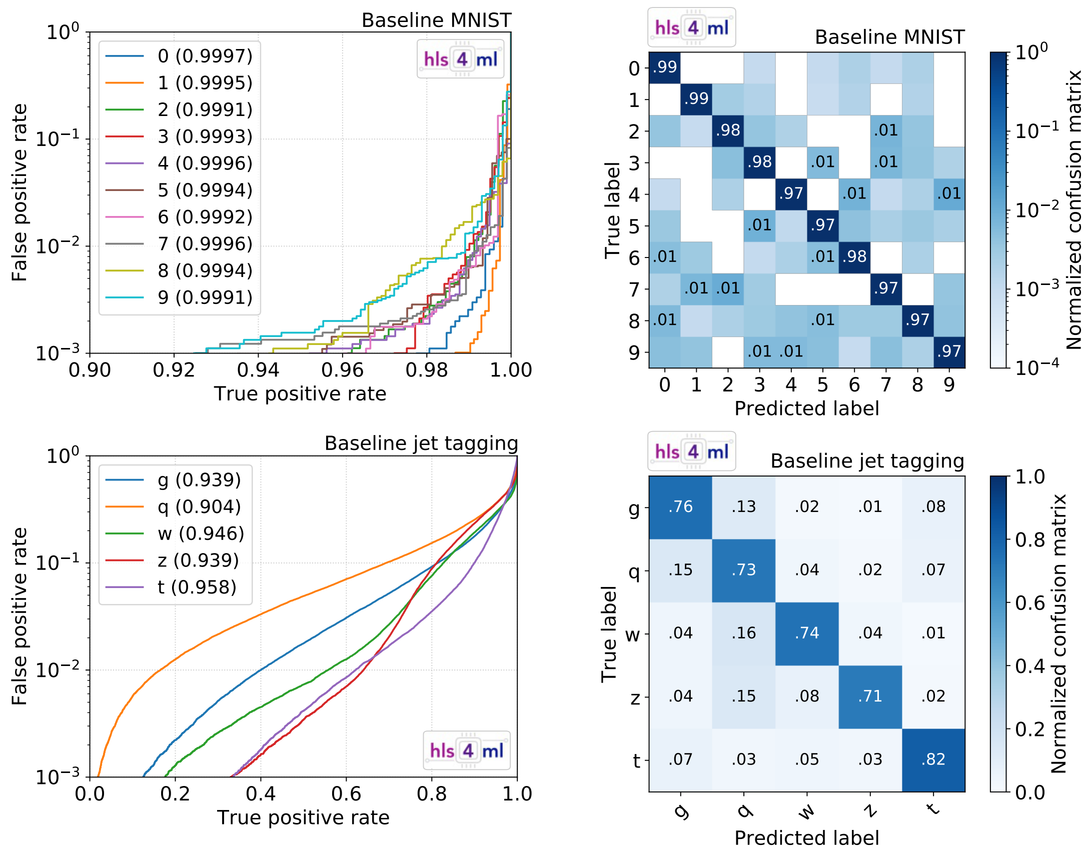

Figure 2: Classification performance evaluated on the testing sample of the baseline MNIST (top) and LHC jet (bottom) classifiers used as benchmark models in this study: ROC curves (left) and normalized confusion matrices (right). On the left, numbers in parentheses correspond to the AUC of each class. On the right, the text is omitted for bins corresponding to a false positive rate below 1%.

The MNIST data set consists of training and validation samples (with 60,000 examples) and a testing sample (with 10,000 examples). Each image is represented as a 28×28 pixel array, storing the gray-scale content of each pixel in the original image. For our purpose, we flatten the 2D array to a 1D array, concatenating each row of the image to the right to the previous one. The derived 1D array is passed as input to a multilayer perceptron (MLP) [15] with an input (output) layer of 784 (10) nodes and three hidden layers with 128 nodes each. Rectified linear unit (ReLU) activation functions [16] are used for the hidden layer nodes, while a softmax activation function is used for the output layer.

The other benchmark task consists of classifying jets from a set of 16 physics-motivated high-level features, as described in Ref. [2]. The network receives as input a vector of 16 quantities and processes them through a MLP with three hidden layers of 64, 32, and 32 nodes with ReLU activation functions. The output layer consists of five nodes with softmax

activation. The five output values correspond to the probability of a given jet to belong to one of the jet classes (light quark q, gluon g, W boson, Z boson, or top quark t). The utilized data set is available on the Zenodo repository [17].

The architectures of the baseline MNIST and LHC jet classifiers are illustrated in Fig. 1. Both are implemented and trained with `Keras` in floating point precision (FPP). Their performance is shown in Fig. 2 in terms of receiver operating characteristic (ROC) curves and normalized confusion matrices. The area under the curve (AUC) of each ROC curve is quoted in the figure, as well as in Table 1, where the corresponding accuracy values are also given. Following convention, we define the accuracy as the fraction of correctly labeled examples. In practice, this is done applying an $\mathrm{Arg\,Max}$ function to the array of scores returned by the network and comparing it to the corresponding target array. The total accuracy of the MNIST and LHC jet classifiers, computed across all categories, are found to be 98% and 75%, respectively.

Table 1: Classification performance evaluated on the testing sample of the baseline MNIST and LHC jet classifiers used as benchmark models in this study: AUC and per-class accuracy.

Class	MNIST		Class	Jet tagging	
	AUC	Accuracy [%]		AUC	Accuracy [%]
0	0.9997	99.7	g	0.939	89
1	0.9995	99.8			
2	0.9991	99.6	q	0.904	85
3	0.9993	99.6			
4	0.9996	99.6	W	0.946	91
5	0.9994	99.6			
6	0.9992	99.6	Z	0.939	92
7	0.9996	99.6			
8	0.9994	99.4	t	0.958	93
9	0.9991	99.5			

3 Implementing binary and ternary networks in `hls4ml`

Binary and ternary networks are extreme examples of quantized neural networks. A network is quantized when the numerical representation of its parameters is a fixed precision. This precision could be constant across the full network or specific for each components (e.g., for different layers). Quantization allows one to reduce the computing resources of a given model for inference and it can be tuned so that it comes with little or no loss in terms of performance. In the case of binary (ternary) networks, each weight assumes a value of $+1$ or -1 ($+1$, 0, or -1). Two- and three-valued activation functions are used after each layer, acting as discrete versions of the $\tanh$ function. As alternatives, we also investigate a standard ReLU function as well as its clipped version [18], defined as $\min(\mathrm{ReLU}(x), y_{\max})$, with $y_{\max}$ being a positive hyperparameter. In our study, we fix $y_{\max} = 1$. The four functions are represented in Fig. 3.

In order to convert the models described in Sections 2, we rely on the MLP-related functionalities offered by the `hls4ml` library, discussed at length in Ref. [2]. In addition to that, we exploit a set of custom implementations [12], specific to binary and ternary networks, that allow one to speed up the execution of the building-block architecture shown in Fig. 4. The implementation of these solutions is integrated in recent versions of the `hls4ml` library, starting with the `v0.1.6` tag of the GitHub repository [19]. With respect to the work presented in Ref. [2], this version provides a special support for large dense layers, which allows one to deal with large number of nodes as in the models we consider in this study. This functionality will be described in more detail in a future publication.

Binary networks use 1-bit representations for both weights and activations. In this case, the product between two quantities can be optimised to an extremely lightweight operation. By encoding an arithmetical value of '-1' as '0', the product can be expressed as an XNOR operation, as shown in Table 2. For models using ternary weights or greater than 1-bit for activations, FPGA logic is always used rather than DSPs.

The binary and ternary $\tanh$ activation functions are implemented by testing the sign (in the case of binary $\tanh$) or sign and magnitude (for ternary $\tanh$) of the input and yielding the corresponding value ± 1 or 0 as seen in Fig. 3. A binary or ternary $\tanh$ activation layer preceded by a BN layer can be further optimized. The usual BN transformation is:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta,$$

given the mean μ, variance σ^2, scale γ, and shift β computed during the network training. For a BN followed by a binary $\tanh$ activation, the sign of y is enough to determine a node output value. To avoid calculating the scaling of x

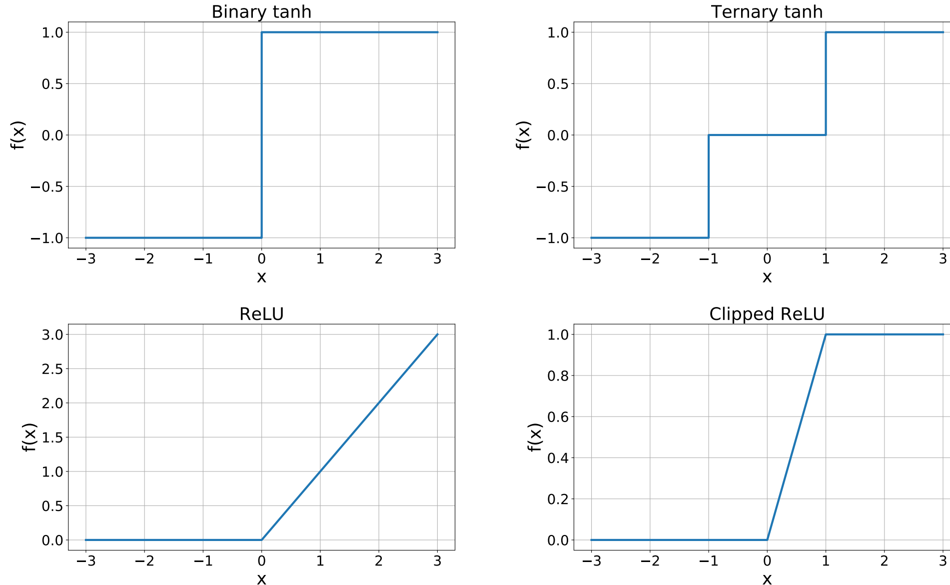

Figure 3: Activation functions used to define the models described in Section 4: binary $\tanh$ (top-left), ternary $\tanh$ (top-right), ReLU (bottom-left) and clipped ReLU (bottom-right).

Table 2: Left: All possible products between A and B with values constrained to ± 1. Right: The corresponding truth-table when the quantities A and B are each encoded with 1-bit, and the XNOR operation is used for the product.

A	B	$A \times B$
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

using FPGA logic, the four BN parameters are used to compute the value of x at which y flips sign. This calculation is performed at compile-time, when the model is converted to HLS firmware using `hls4ml`. Similarly, the two values of x around which the output of the ternary $\tanh$ activation changes are also calculated at compile-time. In the FPGA, each node output is then simply compared against these pre-computed thresholds, outputting the corresponding ± 1, or 0. An additional optimization step sets the type of x in the HLS implementation to integer with a bitwidth corresponding to the largest integer expected for each binary/ternary layer which is computed at compile-time. This procedure allows one to further save FPGA resources.

The binary and ternary layers considered for this work are fully integrated and compatible with the `hls4ml` package. While not explored here, the package also supports models mixing binary/ternary layers with higher precision layers for fully customised networks.

4 Binarization and ternarization strategies

Given a full-precision model, one could follow different strategies to turn it into a binary or ternary model. One could just replace each full-precision component by the corresponding binary/ternary element, in order to minimize resource utilization. This might result in a loss of accuracy. As an alternative, one could train a binary/ternary model with arbitrarily large architecture, in order to match the accuracy obtained at full precision, at a cost of a larger latency and resource consumption. The ultimate strategy to follow depends on the use case. In this work, we present a few options, covering these two extremes and intermediate solutions.

In this work, we focus on binary/ternary MLPs. The basic structure for the adopted architectures is shown in Fig. 4. Each model consists of a sequence of blocks, each composed of a dense, batch normalization (BN) [20], and activation layer. The BN layer shifts the output of the dense layers to the range of values in which the activation function is nonlinear, enhancing the network's capability of modeling nonlinear responses. For binary and ternary $\tanh$, a

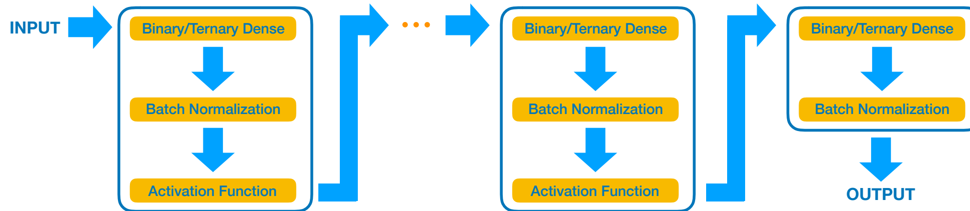

Figure 4: The MLP architecture used in this study, consisting of a sequence of repeating blocks. Each block, fully connected to the previous and following one, consists of a dense layer, a batch normalization layer, and an activation layer. The last block does not have an activation layer.

BN+activation layer sequence can be implemented at small resource cost (see Section 3), which makes this choice particularly convenient for fast inference on edge devices.

The binarization/ternarization of a given model can be done in different ways, e.g., preserving the model architectures or its performance. As a consequence, for each benchmark problem we consider seven models:

- *Baseline*: the three-layer MLP described in Section 2.
- *Binarized (BNN)*: a binary version of the baseline model, built preserving the model architecture (number of layers and nodes) while applying the following changes: use a binary representation (± 1) for the weights; replace the inner-layer ReLU activation functions with a binary $\tanh$ (see Fig. 3); introduce BN layers in between the binary dense layers and the activation functions; remove the softmax activation function in the output layer.
- *Ternarized (TNN)*: a ternary version of the baseline model, built preserving the model architecture (number of layers and nodes) while applying the following changes: use a ternary representation ($-1, 0, +1$) for the weights; replace the inner-layer ReLU activation functions with a ternary $\tanh$ (see Fig. 3); introduce BN layers in between the ternary dense layers and the activation functions; remove the softmax activation function in the output layer.
- *Best BNN*: same structure as the BNN model, but with more nodes in each layer to improve performance. We obtain this model with a Bayesian optimization, finalized to minimize the validation loss in the training process.
- *Best TNN*: same structure as the TNN model, but with the number of nodes per layer chosen through a Bayesian optimization of the architecture, as for the best BNN model.
- *Hybrid BNN*: same as the BNN model, but with ReLU or clipped ReLU activation functions rather than the binary $\tanh$ of Fig. 3.
- *Hybrid TNN*: same as the TNN model, but with ReLU or clipped ReLU activation functions rather than the ternary $\tanh$ of Fig. 3.

The baseline model is taken as a benchmark of ideal performance and the other models represent different strategies toward a more resource-friendly representation. The BNN and TNN models are simple translations of the baseline model. They are designed to reduce the resource consumption, at the potential cost of a performance drop. The best models are designed to match (as close as possible) the performance of the baseline model, which might result in a larger resource consumption with respect to what the BNN and TNN models achieve. The hybrid models are a compromise between the two approaches. The fixed-precision conversion is applied only to the weights and biases of the nodes in the dense layers, while ReLU or clipped ReLU activation functions are used. Given the relatively small resources used by the ReLU/clipped ReLU activations, the hybrid models allow one to reach performance closer to the baseline model without inflating the number of nodes and, consequently, numerical operations. The best BNN and TNN models are only presented for the LHC jet problem, since in that case the simple binarization or ternarization of the baseline model result in a substantial performance loss. The effect is much milder for the MNIST classification problem, so that the binary and ternary architectures are not re-optimized for in that case.

One should notice that not all the components of a binary (ternary) model come in binary (ternary) precision, e.g., the output of a ReLU activation function in a hybrid model. For this reason, in the following we discuss bit precision and network quantization even in the context of binary and ternary models.

All models are implemented in `Keras` [6], with `TensorFlow` [5] as a back-end using the implementation in [13] for binary and ternary layers, which we also cross-checked with `QKeras` [21] with similar results. The network training was performed on a NVIDIA V100 GPU. During training, binary/ternary precision is employed during forward propagation, while full precision is used during backward propagation. The baseline models of Section 2 are trained minimizing a categorical cross entropy. The binary and ternary models are trained minimizing a hinge loss function [22].

5 Experiments

The results presented below are synthesized with the Vivado HLS version 2018.2 for a Xilinx Virtex Ultrascale 9+ FPGA with part number xcvu9p-flga2104-2L-e. The clock frequency is fixed at 200 MHz, which is typical for the LHC L1 triggers. Unless otherwise specified, the quoted results are derived after the HLS compilation step. The network implementation is further refined by the logic synthesis. We verified that this final step does not affect the accuracy while it reduces the resource consumption.

5.1 Handwritten digits classification

We first evaluate the performance of the HLS neural network implementation for the models described in Section 4 with different fixed-point precisions by scanning the number of both integer (I) and fractional (F) bits. In the following, a given choice of fixed-point precision is specified as $\langle \mathrm{T}, \mathrm{I} \rangle$, where $\mathrm{T} = \mathrm{I} + \mathrm{F}$ is the total number of allocated bits. For each case, the minimum number of bits yielding an accuracy above 90% after quantization is considered. We then study the latency and resource utilization in these configurations. Table 3 shows a comparison of the performance obtained for the baseline, binary, and ternary models, in terms of accuracy and AUCs, before and after quantization.

Table 3: Accuracy and AUCs of the different MNIST-classification models described in Section 4 before and after quantization, for the fixed point precision settings chosen for this study. Both the numbers of integer (I) and fractional (F) bits are specified, using the notation $\langle I+F, I \rangle$. For each case, we report the range spanned by the 10 AUC values and the model accuracy evaluated on the testing sample.

Model	Floating point precision		Fixed point precision		
	AUC	Accuracy [%]	Number of bits	AUC	Accuracy [%]
Baseline	0.9991–0.9997	98	$\langle 18, 8 \rangle$	0.9919–0.9959	95
BNN	0.9869–0.9979	93	$\langle 16, 8 \rangle$	0.9860–0.9976	93
TNN	0.9921–0.9992	95	$\langle 16, 6 \rangle$	0.9918–0.9992	95
Hybrid BNN (ReLU)	0.9953–0.9990	95	$\langle 16, 10 \rangle$	0.9956–0.9989	95
Hybrid TNN (ReLU)	0.9970–0.9993	96	$\langle 16, 10 \rangle$	0.9971–0.9993	96
Hybrid BNN (clipped ReLU)	0.9827–0.9983	95	$\langle 16, 10 \rangle$	0.9828–0.9983	95
Hybrid TNN (clipped ReLU)	0.9857–0.9989	96	$\langle 16, 10 \rangle$	0.9859–0.9988	96

For binary and ternary models, the `hls4ml` library applies a further level of per-layer customization of the fixed-point representation, to match the numerical precision of each layer separately, as discussed in Section 3. The outcome of this optimization is shown in the right plot of Fig. 5 for the BNN model, where the gray areas cover different numerical ranges for different layers, despite the common precision specified at compilation ($\langle 16, 8 \rangle$ in this case). During the optimization, the inputs and the outputs are still represented by the fixed-point precision specified by the user, while the precision of the other network components is optimized.

When quantizing a model, one should allocate I and F bits so that the range of values one can cover overlaps with the range of values returned by the network layers, in order to reduce the impact on accuracy. This is shown in the left plot of Fig. 5, where the profile of output values returned by each layer of the baseline model is compared to the range covered by the allocated fixed-point precision. For each layer, we consider the distribution of the output values obtained running the network on a test sample. In the figure, the box represents the quartiles of the distribution, while the line inside the box shows the median. The lines extending beyond the box show the minimum and maximum values. The gray area represents the numerical range covered by the allocated precision. Overall, the optimized precision matched the bulk of the output values at each layer. The only exception is observed for the output layer. In this case, the allocated precision (gray area in the last row of the left plot in Fig. 5) does not cover the bulk of values returned by the layer (red box in the figure). This happens whenever a given example is associated to a specific class with a score close to 1, so that the other values are pushed close to 0 and out of the supported range. In practice, this fact would not alter the classification outcome in inference [2].

[2] For instance, this would not be a problematic aspect when operating this algorithm through the $\mathrm{Arg\,Max}$ function, associating a given example to the class with the largest output.

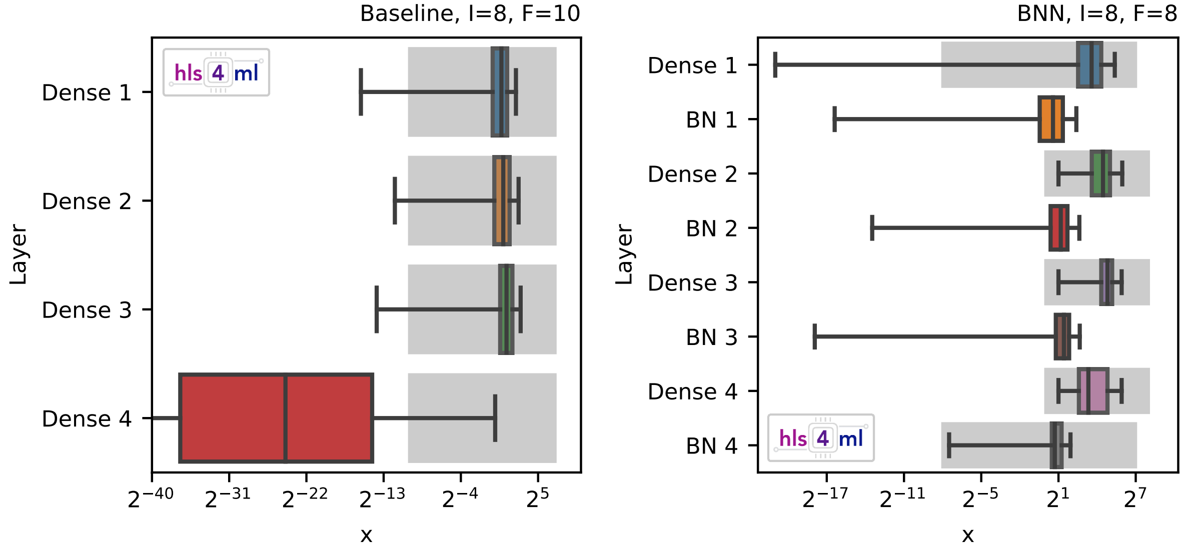

Figure 5: Profile of the range of output values of each layer, sampled during inference on the test dataset, for the baseline (left) and BNN (right) MNIST models. For each layer, the box represents the quartiles of the distribution, while the line shows the median. The lines extending beyond the box show the minimum and maximum values. The gray shaded areas represent the range covered by the allocated fixed point precision for each layer. In the left plot, these ranges correspond to the precision specified at compilation ($\langle 18, 8 \rangle$). On the right plot, an optimization procedure implemented in `hls4ml` for binary and ternary networks automatically adapts the precision of each layer to match the range covered by the output distribution; as the BN layer is merged with the binary $\tanh$ in the HLS implementation, its output precision is 1 bit. Dense, batch normalization (BN), and activation layers are presented in order from the input (top) to the output (bottom).

For the baseline model, the quantization from floating-point precision to $\langle 18, 8 \rangle$ results in an accuracy drop from 98% to 95%. This is almost entirely induced by the softmax activation function applied to the last layer and it results from the limited precision of the LUT implementing the $\exp$ functions in the softmax. This parameter is hard-coded in the version of `hls4ml` used for this study. One could avoid this accuracy loss removing the softmax function at the end, as long as there is interest only on which class has the biggest score and not on the individual scores. An alternative option is to further optimize the precision of the LUT implementing the softmax activation function. In this case, we verified that a $\langle 18, 8 \rangle$ quantization baseline with $\langle 22, 10 \rangle$ precision for the softmax LUT recovers an accuracy of 97% without affecting the resources. The ability to externally configure the precision of the softmax LUT will be implemented in future versions of `hls4ml`.

For the hybrid BNN/TNN models, the same number of bits used for the BNN/TNN cases allows one to achieve the FPP accuracy, at the condition of allocating more integer (10 instead of 6) and less fractional (6 instead of 10) bits. This behaviour can be understood from Figure 6, which shows the range of outputs returned by each hybrid BNN layer. While for I=10 the allocated precision spans the full range of outputs returned by each layer, frequent overflows are observed for the Dense 1, Dense 3 and Dense 4 layers when we set I=6.

Table 4 provides a comparison of the resource utilization and latency for the configurations presented in Tab. 3. For each configuration, we quote both the resource utilization estimated by the HLS compiler and those obtained by the logic synthesis. The logic synthesis transforms the Register Transfer Level (RTL) design created by the HLS compiler into a gate-level implementation, applying additional optimizations that result in a more accurate assessment of the resource utilization. In the table, the II represents the number of clock cycles needed before the algorithm may accept a new set of inputs. In our study, the II value is fixed by requiring that the resulting resource utilization is below the maximum allowed on the target FPGA. Lower II values would result in a network design that would not fit the device. Larger II values would result in higher latency.

At the very low latency values ($\mathcal{O}(100)$ ns) that we are targeting, BNN/TNN models allow one to reach competitive performance while saving most of the FPGA resources. About half of the observed accuracy loss can be recovered using hybrid BNN/TNN models, paying a small price in terms of DSPs utilization, induced by an explicit allocation of

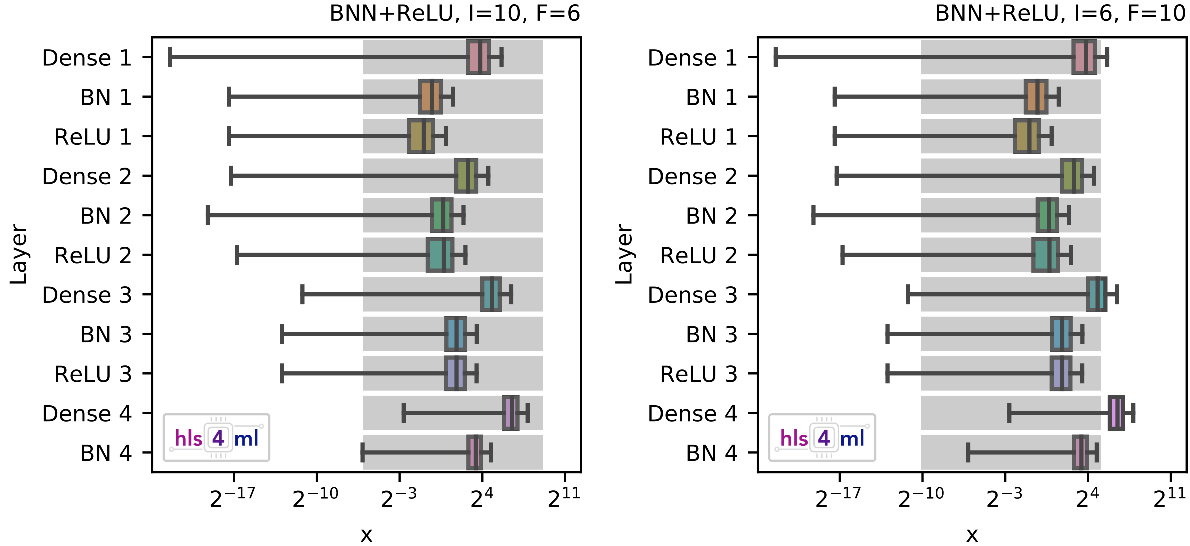

Figure 6: Profile of the range of output values of each layer, sampled during inference on the test dataset, for the hybrid BNN+ReLU model quantized to 16-bit precision, when 10 (left) or 6 (right) bits are used for the integer part. For each layer, the box represents the quartiles of the distribution, while the line shows the median. The lines extending beyond the box show the minimum and maximum values. The gray shaded areas represent the range covered by the allocated fixed-point precision for each layer. Dense, batch normalization (BN), and activation layers are presented in order from the input (top) to the output (bottom).

Table 4: Comparison of the resource utilization for the MNIST-classification models described in Section 4, together with timing information. Resources estimated by the HLS compiler (C) and obtained by the logic synthesis (S) are quoted for a chosen initiation interval (II).

Model	II	Latency [ns]	DSPs [%]		FFs [%]		LUTs [%]		BRAMs [%]	
			C	S	C	S	C	S	C	S
Baseline	28	315	130	100	18	8	69	54	126	61
BNN	14	200	0	0	5	7	155	18	46	16
TNN	14	190	0	0	6	7	174	22	52	16
Hybrid BNN (ReLU)	14	200	1	0.16	7	9	215	31	52	16
Hybrid TNN (ReLU)	14	200	1	1	7	10	217	35	52	16
Hybrid BNN (clipped ReLU)	14	200	1	2	7	8	215	29	52	16
Hybrid TNN (clipped ReLU)	14	200	1	1	7	9	215	31	52	16

a BN layers before the ReLU/clipped ReLU activation functions rather than the bit-shift implementation described in Section 3. A further optimization of the BN operations for hybrid models could in principle push the DSPs utilization closer to zero.

The LUTs usage is largely overestimated by the HLS compiler for all binary and ternary NN models, while it is found to be well within the available resources after the logic synthesis. Hybrid models require more LUTs with respect to the standard BNN/TNN, because of the wider data bitwidth at the input of each binary or ternary layer.

Figure 7 shows the dependence of the resource utilization on the maximum latency achieved by the design (controlled by the II) for the baseline and BNN models. Results for the TNN model are very close to the BNN ones. For all latency values, the resources used by the BNN/TNN models are typically reduced with respect to the baseline model. In particular, there is a large gain in using binary/ternary networks to obtain a large reduction in the number of DSPs up to a few-μs latency. For higher latency values, the II is large enough to allow a small usage of DSPs even for the baseline model. In that case, the advantage of using a binary or ternary quantization would be minor.

As a final test, we train a larger BNN model consisting of three dense layers with 256 nodes each, as in the study of Ref. [12], allowing for a direct comparison of our implementation of a binary architecture with what presented

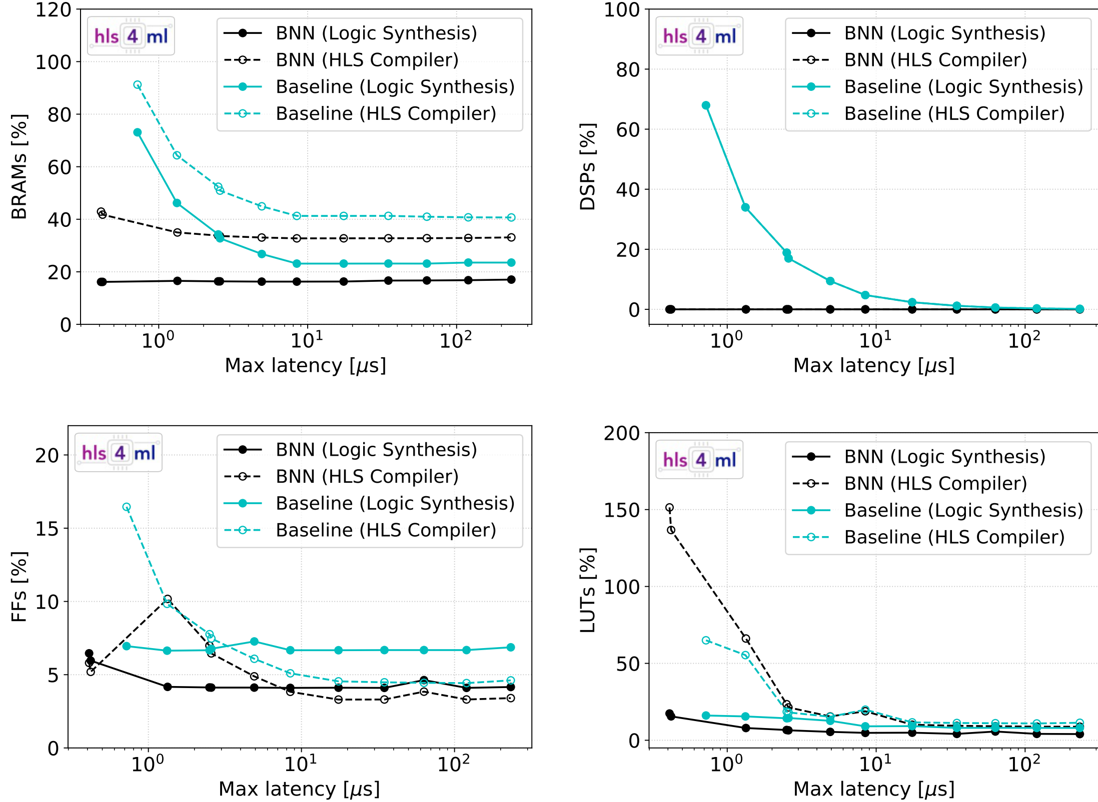

Figure 7: Comparison of the resource utilization estimated by the HLS compiler and obtained by the logic synthesis versus the maximum latency achieved by the design for the BNN and baseline MNIST-classification models. The TNN model gives similar resource utilization as the BNN and is omitted.

there. The `hls4ml` implementation of this model yields a total accuracy of 95% for both floating-point and fixed-point precision, where the latter is fixed to $\langle 16, 6 \rangle$. With an II of 28, we obtain a maximum latency of 0.31 μs with a resource utilization comparable to that in Ref. [12]. In particular, the deployed model obtained with `hls4ml` after the logic synthesis utilizes 0% DSPs, 7% FFs, 23% LUTs, and 16% BRAMs on a Xilinx Virtex Ultrascale 9+ FPGA card.

5.2 LHC jet identification

As a second benchmark example, we consider the LHC jet-tagging problem introduced in Section 2 and study all the binarization/ternarization strategies described in Section 4. For all models a fixed-point precision of $\langle 16, 6 \rangle$ is sufficient to reproduce the FPP accuracy after quantization, except for the hybrid binary/ternary models with ReLU for which a precision of $\langle 16, 8 \rangle$ with two more integer bits is needed. The AUCs and accuracy before and after quantization are reported in Table 5 for all models, while a comparison of the resource utilization is found in Table 6.

Unlike what is seen for the MNIST digit classification, the simple binarization/ternarization of the baseline model results in a big accuracy loss. This is partially mitigated by the use of ReLU and clipped ReLU activations. As an alternative approach, we also consider optimized binary and ternary architectures (best models in Table 5), fixed through a Bayesian optimization of the network hyperparameters. The result of the Bayesian hyperparameter optimization for BNN and TNN converges to architectures with about 40 and 4 times more parameters with respect to the baseline architecture, respectively. With these larger architectures, binary and ternary methods almost match, with a moderate loss in accuracy. Optimizing the architecture of the binary and ternary models yields comparable precisions, but with a different resource balance (e.g., DSPs vs. LUTs), offering an alternative that might better fit certain use cases.

The results of Tables 5 and 6 confirm that ternary networks generally offer a better resource vs. accuracy balance than binary networks, with a minimal (often negligible) additional resource cost and a comparable (sometimes smaller) latency. In terms of FPGA resources, even the large architecture of the best TNN model results in a limited resource

Table 5: Accuracy and AUCs of the different LHC jet tagging models described in Section 4 before and after quantization, for fixed-point precision $\langle I+F, I\rangle$ chosen for this study. For each case, we quote the range spanned by the 5 AUC values and the model accuracy evaluated on the testing sample.

Model	Architecture	Floating point precision		Fixed point precision		
		AUC	Accuracy [%]	Number of bits	AUC	Accuracy [%]
Baseline	16x64x32x32x5	0.904–0.958	75	$\langle 16,6\rangle$	0.900–0.955	75
BNN	16x64x32x32x5	0.794–0.891	58	$\langle 16,6\rangle$	0.794–0.891	58
TNN	16x64x32x32x5	0.854–0.915	67	$\langle 16,6\rangle$	0.854–0.915	67
Best BNN	16x448x224x224x5	0.886–0.937	72	$\langle 16,6\rangle$	0.884–0.938	72
Best TNN	16x128x64x64x64x5	0.886–0.931	72	$\langle 16,6\rangle$	0.886–0.930	72
Hybrid BNN (ReLU)	16x128x64x64x5	0.878–0.926	69	$\langle 16,8\rangle$	0.878–0.926	69
Hybrid TNN (ReLU)	16x128x64x64x5	0.876–0.939	71	$\langle 16,8\rangle$	0.876–0.939	71
Hybrid BNN (clipped ReLU)	16x128x64x64x5	0.866–0.917	68	$\langle 16,6\rangle$	0.866–0.917	68
Hybrid TNN (clipped ReLU)	16x128x64x64x5	0.870–0.933	70	$\langle 16,6\rangle$	0.870–0.933	70

usage, well below the baseline model. Instead, the largest best BNN model requires a higher II value to fit the FPGA resource boundaries. The latency is kept within the $\sim 1\mu$s boundary we target, but is significantly larger than what is achieved by the best TNN and the baseline models. The best TNN model gives the same accuracy as the best BNN model, with the same latency as the baseline model but with a drastic reduction of DSP utilization .

Table 6: Comparison of the resource utilization for the LHC jet-tagging models described in Section 4, together with timing information. Resources estimated by the HLS compiler (C) and obtained by the logic synthesis (S) are quoted for a chosen initiation interval (II).

Model	II	Latency [ns]	DSPs [%]		FFs [%]		LUTs [%]		BRAMs [%]	
			C	S	C	S	C	S	C	S
Baseline	1	60	60	57	1	1	7	5	0	0
BNN	1	40	0	0	0	0	3	1	0	0
TNN	1	40	0	0	0	0	4	1	0	0
Best BNN	16	205	0	0	1	3	128	8	12	0
Best TNN	1	55	0	0	0	0	14	3	0	0
Hybrid BNN (ReLU)	1	70	3	4	1	1	14	6	0	0
Hybrid TNN (ReLU)	1	65	3	4	1	1	22	6	0	0
Hybrid BNN (clipped ReLU)	1	60	3	4	0	1	14	6	0	0
Hybrid TNN (clipped ReLU)	1	60	3	4	1	1	22	6	0	0

6 Summary and Outlook

We presented the implementation of binary and ternary networks in the `hls4ml` library, designed to automatically convert a given neural network model into firmware of an FPGA card. Using two benchmark classification examples (handwritten digit recognition on the MNIST data set and jet identification at the LHC), we discuss different strategies to convert a given model into a binary or a ternary model. We showed how binary and ternary networks allow one to preserve competitive performance (in terms of accuracy) while drastically reducing the resource utilization on the card and, at the same time, keeping the inference latency at $\mathcal{O}(100)$ ns. When compared to binary models, ternary models reach accuracy values much closer to the original baseline models, at a typically smaller resource cost and comparable latency. Model binarization and ternarization are competitive alternatives to other compression approaches (e.g., pruning) and represent the ultimate resource saving in terms of network quantization. They offer a qualitative advantage of keeping DSP utilization at a minimum, and offer an interesting opportunity to deploy complex architectures on resource constrained environments, such as the L1 trigger system of a typical collider physics experiment.

Acknowledgement

We acknowledge the Fast Machine Learning collective (https://fastmachinelearning.org) as an open community of multi-domain experts and collaborators. This community was important for the development of this project.

M. P., S. S., V. L. and J. N. are supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement n° 772369). S. J., M. L., K .P., and N. T. are supported by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics. P. H. is supported by a Massachusetts Institute of Technology University grant. PH and DR thank support from NSF AWARD #190444, #1934700, #1931469, #1836650. Z. W. is supported by the National Science Foundation under Grants No. 1606321 and 115164.

References

[1] CMS Collaboration, "Boosted Decision Trees in the Level-1 Muon Endcap Trigger at CMS", *J. Phys. Conf. Ser.* **1085** (2018), no. 4, 042042, doi:10.1088/1742-6596/1085/4/042042.

[2] J. Duarte et al., "Fast inference of deep neural networks in FPGAs for particle physics", *JINST* **13** (2018), no. 07, P07027, doi:10.1088/1748-0221/13/07/P07027, arXiv:1804.06913.

[3] B. Schlag, "Jet Reconstruction in the ATLAS Level-1 Calorimeter Trigger with Deep Artificial Neural Networks",. Presented 20 Aug 2018.

[4] M. Wielgosz and M. Karwatowski, "Mapping neural networks to fpga-based iot devices for ultra-low latency processing", *Sensors* **19** (2019), no. 13, doi:10.3390/s19132981.

[5] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems", 2015. http://tensorflow.org/.

[6] F. Chollet et al., "Keras", 2015. https://github.com/fchollet/keras.

[7] A. Paszke et al., "Automatic differentiation in PyTorch", in *NIPS-W*. 2017.

[8] J. Bai et al., "Onnx: Open neural network exchange", 2019. https://github.com/onnx/onnx.

[9] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1", *CoRR* **abs/1602.02830** (2016) arXiv:1602.02830.

[10] M. Courbariaux, Y. Bengio, and J. David, "Binaryconnect: Training deep neural networks with binary weights during propagations", *CoRR* **abs/1511.00363** (2015) arXiv:1511.00363.

[11] F. Li and B. Liu, "Ternary weight networks", *CoRR* **abs/1605.04711** (2016) arXiv:1605.04711.

[12] Y. Umuroglu et al., "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, pp. 65–74. ACM, New York, NY, USA, 2017. doi:10.1145/3020078.3021744.

[13] B. Moons, K. Goetschalckx, N. V. Berckelaer, and M. Verhelst, "Minimum energy quantized neural networks", *CoRR* **abs/1711.00215** (2017) arXiv:1711.00215.

[14] Y. LeCun and C. Cortes, "MNIST handwritten digit database", 2010. https://www.bibsonomy.org/bibtex/2935bad99fa1f65e03c25b315aa3c1032/mhwombat.

[15] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain", *Psychological Review* (1958) 65–386.

[16] R. H. R. Hahnloser et al., "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit", *Nature* **405** (2000), no. 6789, 947–951.

[17] J. M. Duarte et al., "Hls4ml lhc jet dataset (150 particles)", January, 2020. https://doi.org/10.5281/zenodo.3602260. doi:10.5281/zenodo.3602260.

[18] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization", *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017) 5406–5414.

[19] V. Loncar et al., "hls4ml: v0.1.6", February, 2020. doi:10.5281/zenodo.3661292, https://github.com/hls-fpga-machine-learning/hls4ml.

[20] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", arXiv:1502.03167.

[21] C. Coelho, "Qkeras", 2019. https://github.com/google/qkeras.

[22] Y. Lin, G. Wahba, H. Zhang, and Y. Lee, "Statistical properties and adaptive tuning of support vector machines", *Mach. Learn.* **48** (September, 2002) 115–136, doi:10.1023/A:1013951620650.