COMPRESSING DEEP NEURAL NETWORKS ON FPGAS TO BINARY OR TERNARY PRECISION

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 at 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

January 8, 2020

ABSTRACT

We present the implementation of binary and ternary neural networks in the `hls4ml` library, designed to convert Deep Neural Network Models in electronic circuits, emulated in an FPGA firmware. Starting from benchmark models trained with full precision, we investigate different strategies to reduce the layer precision to binary or ternary and we discuss the trade-off between model accuracy and resource consumption. In addition, we show how an optimal balance between latency and accuracy can be obtained by retaining full precision on a selected subset of network layers. As an example, we consider two multi-class classification tasks: digit recognition on the MNIST dataset and jet identification on simulated proton-proton collisions at the Large Hadron Collider.

***K*eywords** Deep Learning · FPGA · Binary

1 Introduction

Field-programmable gate arrays (FPGAs) are one of the most interesting solutions to perform fast inference of Deep Neural Networks (DNNs). Their design is extremely functional to parallelize the kind of mathematical operations typical of DNN inference tasks. FPGAs can be programmed, which offers unquestionable advantages in terms of flexibility, with respect to ASICs. At the same time, they share many advantages offered by ASICs, such as low power consumption and speed.

Traditionally, FPGAs are used to emulate generic electronic circuits, as a preliminary step towards 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 nsec and thanks to a built-in buffering system, a typical LHC experiments has $\mathcal{O}(10)$ μsec to decide to keep or discard a given event. This real-time decision-taking system, usually referred to as "L1 trigger", consists in a set of electronic circuits implementing physics-motivated rule-based selection algorithms. Currently, these algorithms are emulated on FPGAs, mounted on custom boards. Recently, the possibility of deploying machine learning (ML) algorithms in the L1 trigger electronic logic was explored [1].

In the future, the complexity of LHC collision events is expected to increase, making the L1 selection task much harder. Since the severe L1 latency constraints prevent the LHC experimental collaborations from deploying complex rule-based algorithms on the L1 FPGA boards, ML solutions, and in particular DNNs, are currently investigated as a fast-to-execute approximation of complex rule-based algorithms: 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). In order to facilitate the implementation of this strategy and the deployment of DNNs in the L1 triggers of the LHC experiments, we started developing a software library, `hls4ml`, to convert a DNN model into an FPGA firmware through an automatic workflow. In High Energy Physics, the deployment of DL models on FPGAs has been discussed in the context of the online data-selection system of the LHC experiments. Solutions based on HLS [2] and VHDL [3] have been considered. 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 allows to keep the latency within typical values of $\mathcal{O}(10)$ μsec. 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 High Energy Physics. In general, `hls4ml` provides a user-friendly solution to deploy custom DNN models on FPGAs, used as accelerators or as electronic-circuit emulators on low-resource computing environments (e.g., drones, self-operating vehicles, etc.).

The main challenge in deploying a DNN model on an FPGA is the limited computational resources. Typically, one would use large reuse factors to dilute the inference operations across multiple clock cycle, at the price of a larger latency. A complementary approach consists in compressing the model, e.g., by reducing the amount 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 to execute simple fully-connected DNN models with state-of-the-art performance on specific LHC problems within a latency of $\mathcal{O}(\infty\prime\prime)$ nsec, while using only a small fraction of the FPGA resources. In this paper, we investigate how a similar result could be obtained with binary and ternary networks [9, 10, 11]. In our development, we followed closely the studies presented in Ref. [12, 10, 13].

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

2 Benchmark models and datasets

We consider two benchmark classification problems: a digit recognition problem on the MNIST dataset and the LHC jet tagging problem discussed in Ref. [2].

The MNIST dataset consists of training+validation (with 60K examples) and a test (with 10K examples) data sets. Each image is represented as a gray-scale array of 28x28 pixels. To our purpose, we flatten the 2D array to a 1D array, concatenating each line of the image right to the previous one. The derived 1D array is passed as input to a MLP with

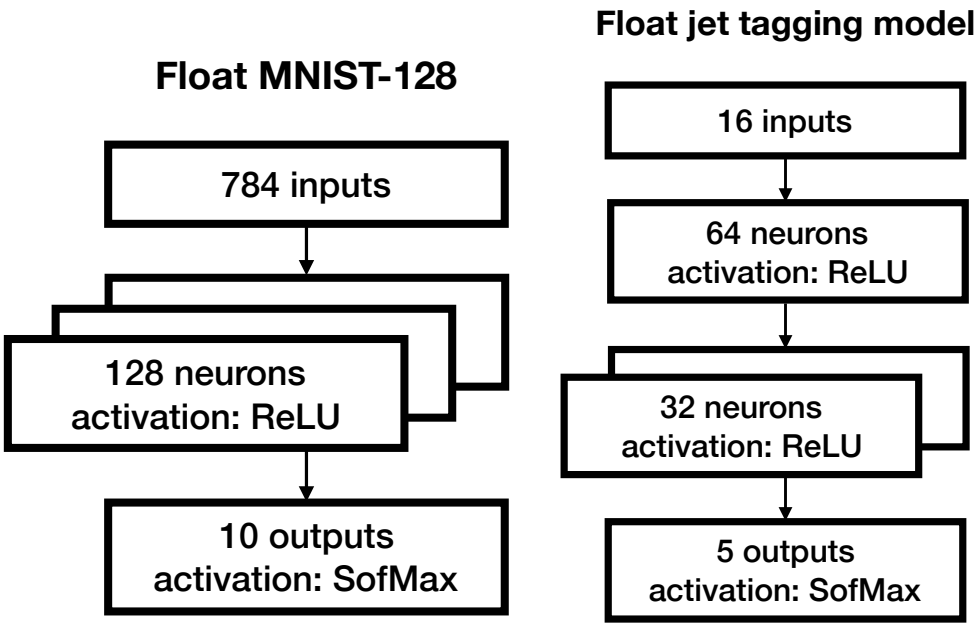

Figure 1: Network architecture for the MNIST (left) and jet flavour (right) classifiers used as benchmark model in this study.

an input (output) layer of 784 (10) nodes and three inner layers with 128 nodes each. ReLU activation functions [14] are used for each inner layer, while the softmax activation function is used for the output layer.

The other benchmark problem consists in classifying jets from a set of 16 physics-motivated high-level features, as described in Ref. [2]. The model receives as input a vector of 16 quantities and processes them through three layers of 64, 32, and 32 nodes with ReLU activation function. The output layer consists of five nodes, corresponding the five classes of jets (light quark q, gluon, W boson, Z boson, or top quark).

The architectures of the benchmark MNIST and jet flavour classifiers are illustrated in Fig. 1. Their performance are shown in Fig. 2, in terms of receiver operating characteristic (ROC) curves and confusion matrices. In addition, we quote the area under the curve (AUC) for each class. The total accuracy of the MNIST and jet flavour classifier is found to be 98% and 75%, respectively. The classification performance for the two models are summarized in Table **??**.

Class	MNIST		Class	Jet Tagging	
	AUC	Accuracy [%]		AUC	Accuracy [%]
0	0.9996	99.29	gluon	0.934	76
1	0.9997	99.12			
2	0.9996	97.77	quark	0.898	73
3	0.9994	97.62			
4	0.9996	97.45	W	0.942	74
5	0.9993	97.42			
6	0.9994	98.12	Z	0.934	71
7	0.9994	97.37			
8	0.9991	96.82	top	0.954	82
9	0.9991	96.63			

Table 1: Performance of the MNIST and LHC Jet classifiers used as benchmark models in this study. The by-class accuracy is defined as the fraction of examples of a given class to be correctly classified, after applying an ArgMax function to the network output. **AUC in MNIST not in percentage and with 4 digits**

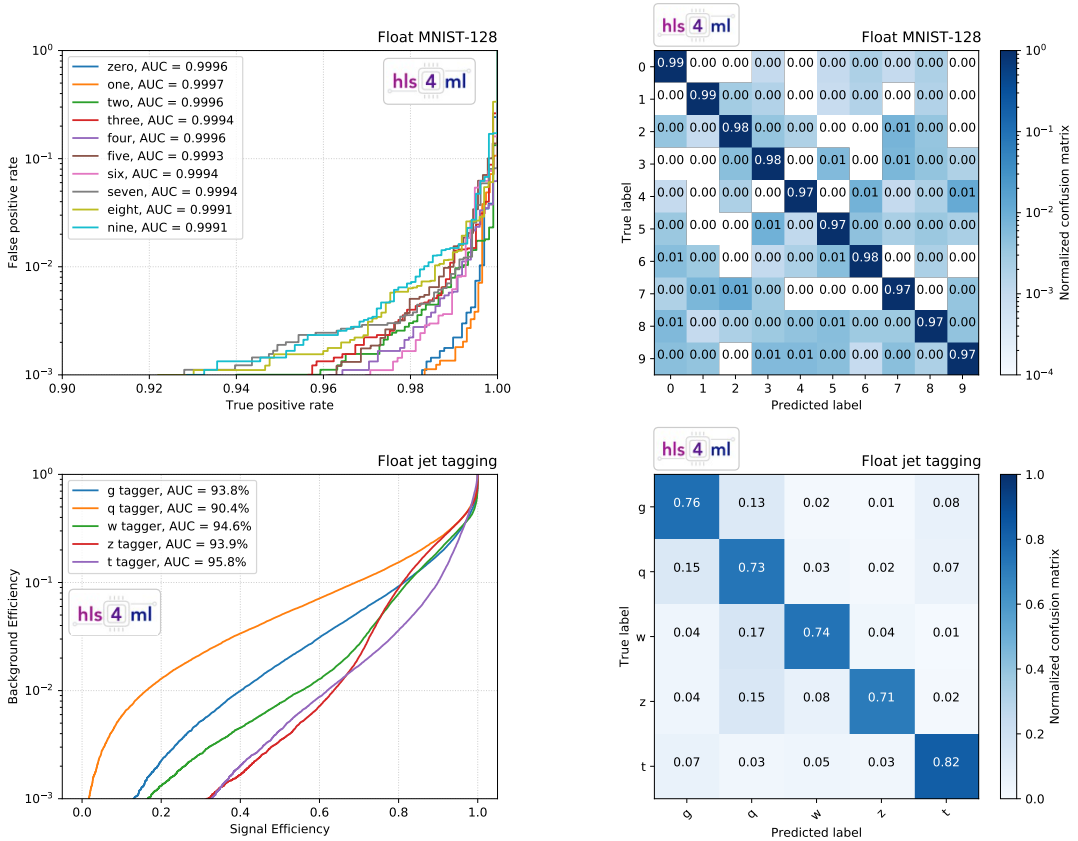

Figure 2: Classification performance of the MNIST (top) and jet-flavour (bottom) classifiers used as benchmark model in this study: ROC curves (left) and normalized confusion matrix (right). **AUC in MNIST not in percentage and with 4 digits; show FPR (in log scale) vs TPR (linear) for both. Maybe put in log scale the temperature scale of the MNIST confusion matrix**

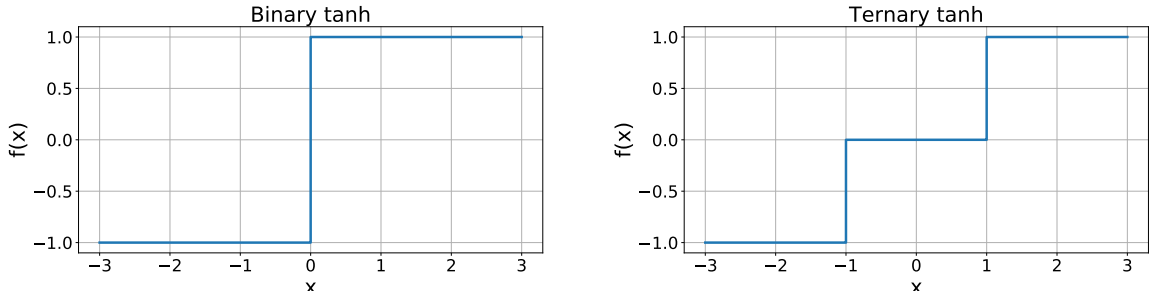

Figure 3: Binary (left) and ternary (right) tanh activation functions, used in the models described in Section 5.2. **the x axis label is cut. TO BE FIXED**

3 Binary and ternary architectures

Binary and ternary networks are extreme examples of quantized neural networks. A network is quantized when the numerical representation of its parameters is done with a fixed-precision. This precision could be fixed across the full network or customized for specific components. Quantization allows to reduce the computing resources required to use a given model for inference and it usually implies little or no cost in terms of performance. In the case of binary and ternary networks this concept is pushed to extreme. Each element of a binary (ternary) network is forced to assume values ± 1 (± 1 or 0). Two- and three-values activation functions (see Fig. 3) are used after each layer, acting as discrete

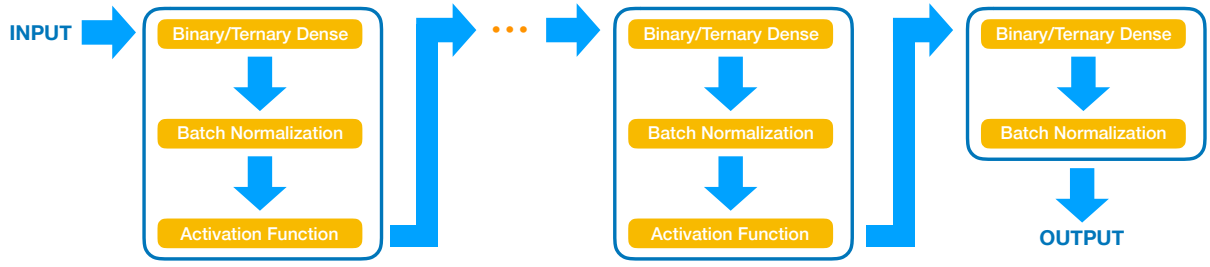

Figure 4: The MLP architecture used in this study, consisting in a sequence of repeating blocks, Each block, fully connected to the previous and following one, consists of a dense layer, a BatchNormalization layer, and the activation layer. The last block does not have an activation layer.

versions of the $\tanh$ function. To help the model training, a BatchNormalization [15] layer is used between the dense layer and its activation function.

In this work, we apply different binarization/ternarization strategies on multilayer perceptrons (MLPs). The adopted architecture is shown in Fig. 4. Each model consists of a sequence of blocks, each made of a dense, a BatchNormalization, and an activation layer. We use of fixed-precision numerical representation and specific activation functions (see Fig. 3), working in conjunction with the BatchNormalization layers [15]. The latter shifts the output of the dense layers to the range of values in which the activation function is non-linear, enhancing the network's capability of modeling non-linear responses. At the same time (see Section 4, the sequence of BatchNormalization+activation layers can be implemented at small resource cost, which makes this choice particularly convenient for fast inference on edge.

The full benchmark models, defined before any reduction of the numerical representation of the network parameters, are designed to solve two multi-class classification tasks: a digit recognition problem on the MNIST dataset; and a jet tagging problem typical of a particle physics experiment at the LHC. These full models are trained minimizing a categorical cross entropy. The two benchmark classification tasks are solved minimizing a hinge loss function [**?**]. 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:

- *Full model:* the three-layers MLP
- *Full Binarized*: a binary version of the Full model, built preserving the model architecture (number of layers and nodes) while applying the following changes: use a binary representation (± 1) for weights and biases; replace the inner-layer ReLU activation functions with a *binary tanh* (see Fig. 3); introduce Batch Normalization layers in between the binary dense layers and the activation functions; remove the soft-max activation function in the output layer.
- *Full Ternarized*: a binary version of the Full model, built preserving the model architecture (number of layers and nodes) while applying the following changes: use a ternary representation ($-1,\ 0,\ 1$) for weights and biases; replace the inner-layer ReLU activation functions with a *ternary hard-max* (see Fig. 3); introduce Batch Normalization layers in between the ternary dense layers and the activation functions; remove the soft-max activation function in the output layer.
- *Best Binary*: same structure as the Full Binarized model, but with different number of nodes in each layer. This model, relevant for the LHC-specific jet classification problem described in Section 2, was optimized with a Bayesian optimization, finalized to minimize the validation loss in the training process.
- *Best Ternary*: same structure as the Full Ternarized model, but with the number of nodes per layer chosen through a Bayesian optimization of the architecture, as for the Best Binary model.
- *Hybrid Binary*: same as the Full Binary model, but with ReLU activation functions rather than the binary tanh of Fig. 3.
- *Hybrid Ternary*: same as the Full Ternary model, but with ReLU activation functions rather than the ternary tanh of Fig. 3.

The *Full* model is taken as as a benchmark of ideal performance and the other models represent different strategies towards a more resource-sparse representation. The *Full Binarized* and *Full Ternarized* models are simple translations of the full model. They are optimal in terms of resource reduction, at the cost of a performance drop. The *best* models are designed to match (as close as possible) the performance of the Full model, resulting in a larger resource consumption

than the *Full Binarized* and *Full Ternarized* models. 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 activation functions are used. Given the relatively small resources used by the ReLU activation, the *Hybrid* models allow to reach the same performance of the Full model without inflating the number of nodes.

4 Optimization for FPGA deployment with `hls4ml`

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, specific of binary and ternary networks, that allow to speed up the execution of the building-block architecture shown in Fig. 4. **did we take this somewhere or is this our idea? Do we need to cite someone?**

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

Binary networks use 1-bit 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 $\tanh$ - and yielding the corresponding value ± 1 or 0 as seen in Fig. 3. A binary or ternary $\tanh$ activation layer preceded by a Batch Normalization (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 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 an 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.

5 Experiments

5.1 Handwritten digits classification

We first evaluate the performance of the HLS neural network implementation for the models described in Section 5.2 with different fixed point precision in steps of 2 bits fixing the number of fractional bits to 10 **So in practice I just scan the number of integer bits**. For each case, the minimum precision yielding an accuracy above 90% after quantization is then chosen to study the latency and resource utilization. The number of bits for each model is reported in Table ?? **Maybe put AUCs/accuracy with number of bits in a separate table?**. For the full and BNN models, a larger number of bits would yield a higher accuracy, hence closer to the one obtained in floating point precision with Keras, however, such choice would also result in an increase in resource utilization: a factor 2 increase in LUTs for the BNN, and a factor 2 increase in DSPs for the full model.

1) How about having two tables here: one with latency/resources at fixed II/RF and one with the minimum achievable reuse factor and corresponding latency/resources. However, the minimum achievable reuse factor needs to be defined (as breakdown point of HLS compiler?)
2) For these tables it might be useful to indicate also the LS resources and the AUC/accuracy after quantization. Explain what we mean with logic synthesis in the text.
3) Add in the text info on Vivado version, targeted clock (5 ns) and xilinx part (xcvu9p-flga2104-2L-e, virtex

ultrascale 9 +).
4) Need to describe MaxReLu somewhere.
5) We have never run the bayesian optimization for the MNIST. Shall we? Alternatively, add results for more neurons (256,512, etc...)?

Table 3: Accuracy and AUCs of the different models before and after quantization for the fixed point precisions chosen for these studies.

Model	Number of bits	AUC	Accuracy [%]	AUC	Accuracy [%]
		Floating point		Quantized	
Full	18	0.9991–0.9997	98	0.9922–0.9965	95
BNN	16	0.9869–0.9979	93	0.9823–0.9973	91
TNN	16	0.9921–0.9992	95	0.9918–0.9985	95
Hybrid BNN+ReLU (20 bits)	20	0.9953–0.9990	95	0.9953–0.9990	95
Hybrid TNN+ReLU (20 bits)	20	0.9970–0.9993	96	0.9970–0.9993	96
Hybrid BNN+MaxReLU (18 bits)	18	0.9827–0.9983	95	0.9829–0.9982	95
Hybrid TNN+MaxReLU (18 bits)	18	0.9857–0.9989	96	0.9859–0.9989	96

Table 4: Comparison of accuracy, latency, and resource utilization at fixed II=128 for the models described in Section5.2. The architecture here is fixed at 784x128x128x128x10.

Model	Latency [ns]	DSPs [%]	FF [%]	LUTs [%]	BRAMs [%]
Full	2580	16 (17)	7 (7)	18 (14)	50 (33)
BNN	2575	0 (0)	6 (4)	21 (6)	33 (16)
TNN	2560	0 (0)	6 (4)	22 (7)	33 (18)
Hybrid BNN+Relu	2585	4 (5)	5 (8)	27 (17)	41 (20)
Hybrid TNN+Relu	2600	4 (5)	5 (8)	27 (17)	41 (22)
Hybrid BNN+MaxRelu	2590	4 (5)	6 (7)	27 (15)	37 (18)
Hybrid TNN+MaxRelu	2590	4 (5)	6 (7)	27 (16)	37 (20)

Table 5: Comparison of accuracy, timing, and resource utilization from the HLS estimate for the minimum achievable reuse factor for the models described in Section 5.2. The architecture here is fixed at 784x128x128x128x10. Numbers in parentheses for the accuracy and AUC correspond to the values obtained after quantization. Numbers in parentheses for the resources correspond to the values obtained from the logic synthesis. **Why the TNN gives two clock cycles less for same RFs? Because of the zeros? Why the quantized accuracy is higher for the TNN wrt BNN for same number of bits (16)?**

Model	II	Latency [ns]	DSPs [%]	FF [%]	LUTs [%]	BRAMs [%]
Full	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	210	4 (5)	10 (13)	222 (71)	60 (20)
Hybrid TNN+ReLU	14	215	5 (5)	10 (14)	215 (74)	60 (20)
Hybrid BNN+MaxReLU	14	210	4 (5)	8 (10)	215 (63)	56 (18)
Hybrid TNN+MaxReLU	14	210	4 (5)	8 (10)	216 (64)	56 (18)

After fixing the precision, we perform a scan of the resources as a function of the latency. The latency is controlled by the reuse factor which can be different among the layers.

5.2 Jet substructure at the LHC

As a second benchmark example, we consider the particle-physics application described in Ref. [2]: given a set of jets produced in proton-proton collisions at the LHC and a set of 16 physics-motivated quantities, we train a multi-layer perceptron (MLP) to associate each jet to one of five mutually exclusive classes: quark (q), gluon (g), W-, Z, or top (t) jets. To this purpose, we train a multi-class classifier with 16 input and 5 output nodes.

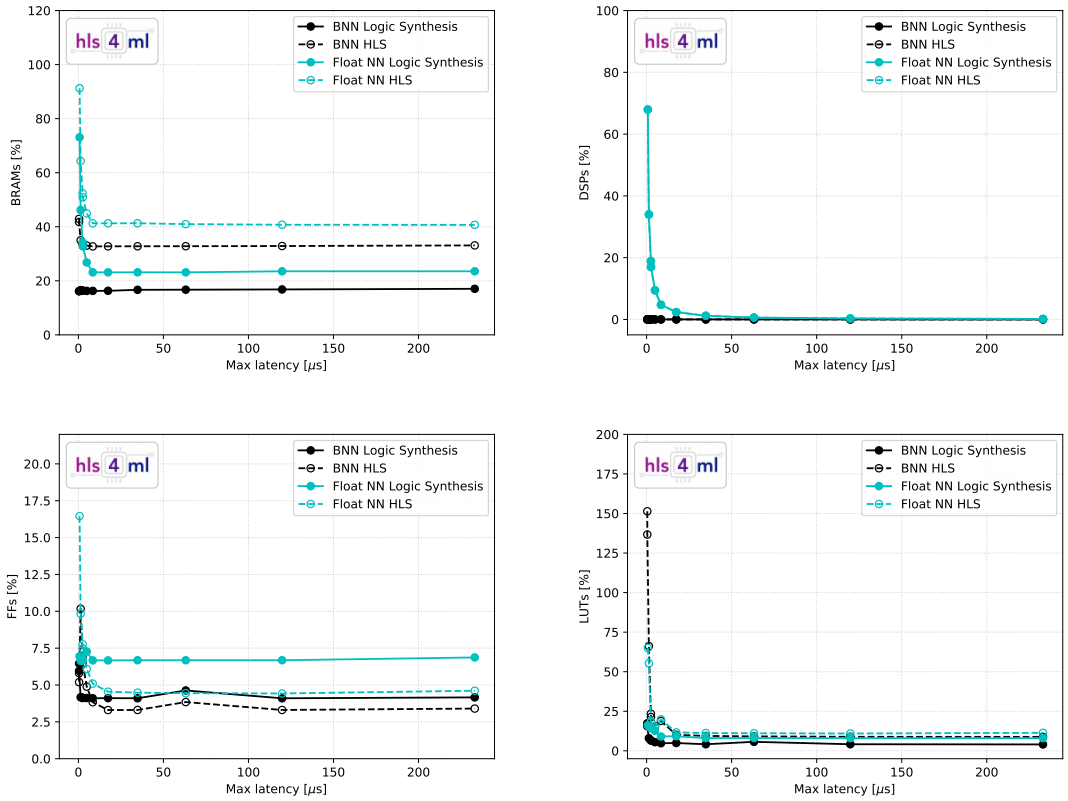

Figure 5: Scan of the resource utilization estimated by the HLS compiler nd by the LS versus latency for the BNN and full model cases. The TNN gives similar results as the BNN and they are omitted from this plot.

Table 6: Comparison of accuracy, latency, and resource utilization for the models described in Section5.2.

Model	Architecture	AUC	Accuracy	Latency	DSPs	FF	LUTs
Full	16x64x32x32x5	90–96%		110 nsec	15%	1%	5%
Full Binarized	16x64x32x32x5	-%			-%	-%	-%
Full Ternarized	16x64x32x32x5	-%			-%	-%	-%
Best Binary	16x448x224x224x5	-%			-%	-%	-%
Best Ternary	16x128x64x64x64x5	89-93%		125 ns	0%	1%	27%
Hybrid Binary	16x128x64x64x5	88-93%			-%	-%	-%
Hybrid Ternary	16x128x64x64x5	88-93%			-%	-%	-%

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 a firmware of an FPGA card.

Using two benchmark classification examples (hand digit recognition on the MNIST dataset and jet-flavor identification for the LHC experiments), we discuss different strategies to convert a given model into a binary or a ternary model.

We showed how binary and ternary networks allow 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 $\mathcal{O}(\text{nsec})$.

Model binarization and ternarization are competitive alternatives to alternative 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.

7 Acknowledgement

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.

J. D., B. K., S. J., R. R., 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. 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^o 772369). Z. W. is supported by the National Science Foundation under Grants No. 1606321 and 115164.

References

[1] CMS Collaboration, D. Acosta et al., *Boosted Decision Trees in the Level-1 Muon Endcap Trigger at CMS*, *J. Phys. Conf. Ser.* **1085** (2018) 042042.

[2] J. Duarte et al., *Fast inference of deep neural networks in FPGAs for particle physics*, *JINST* **13** (2018) P07027 [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) .

[5] M. Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* http://tensorflow.org/.

[6] F. Chollet et al., *Keras* 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* https://github.com/onnx/onnx.

[9] M. Courbariaux, Y. Bengio and J. David, *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*, *CoRR* **abs/1511.00363** (2015) [1511.00363].

[10] M. Courbariaux and Y. Bengio, *BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*, *CoRR* **abs/1602.02830** (2016) [1602.02830].

[11] F. Li and B. Liu, *Ternary Weight Networks*, *CoRR* **abs/1605.04711** (2016) [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, (New York, NY, USA), pp. 65–74, ACM, 2017 DOI.

[13] B. Moons, K. Goetschalckx, N. V. Berckelaer and M. Verhelst, *Minimum Energy Quantized Neural Networks*, *CoRR* **abs/1711.00215** (2017) [1711.00215].

[14] R. H. R. Hahnloser et al., *Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit*, *Nature* **405** (2000) 947.

[15] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, http://dl.acm.org/citation.cfm?id=3045118.3045167.