



Autoencoders on field-programmable gate arrays for real-time, unsupervised new physics detection at 40 MHz at the Large Hadron Collider

Ekaterina Govorkova¹✉, Ema Puljak¹, Thea Aarrestad¹, Thomas James¹, Vladimir Loncar^{1,2}, Maurizio Pierini¹, Adrian Alan Pol¹, Nicolò Ghielmetti^{1,3}, Maksymilian Graczyk^{1,4}, Sioni Summers¹, Jennifer Ngadiuba^{5,6}, Thong Q. Nguyen⁶, Javier Duarte⁷ and Zhenbin Wu⁸

To study the physics of fundamental particles and their interactions, the Large Hadron Collider was constructed at CERN, where protons collide to create new particles measured by detectors. Collisions occur at a frequency of 40 MHz, and with an event size of roughly 1 MB it is impossible to read out and store the generated amount of data from the detector and therefore a multi-tiered, real-time filtering system is required. In this paper, we show how to adapt and deploy deep-learning-based autoencoders for the unsupervised detection of new physics signatures in the challenging environment of a real-time event selection system at the Large Hadron Collider. The first-stage filter, implemented on custom electronics, decides within a few microseconds whether an event should be kept or discarded. At this stage, the rate is reduced from 40 MHz to about 100 kHz. We demonstrate the deployment of an unsupervised selection algorithm on this custom electronics, running in as little as 80 ns and enhancing the signal-over-background ratio by three orders of magnitude. This work enables the practical deployment of these networks during the next data-taking campaign of the Large Hadron Collider.

Proton–proton collision events occur 40 million times per second at the particle detectors at the CERN Large Hadron Collider (LHC)¹. The largest general-purpose particle detectors at the LHC, ATLAS² and CMS³, discard most of the collision events with online selection systems, as a result of bandwidth limitations. These systems consist of two stages: the level-1 trigger (L1T)^{4–7}, where algorithms are deployed as programmable logic on custom electronic boards equipped with field-programmable gate arrays (FPGAs), and the high-level trigger (HLT), where selection algorithms asynchronously process the events accepted by the L1T on commercially available CPUs. The event rate is reduced from 40 MHz to around 100 kHz within a few microseconds at the first selection stage, L1T. When designing searches for collisions containing new physics (for example, dark matter production), physicists typically consider specific scenarios motivated by theoretical considerations. This supervised strategy has proven to be successful when dealing with theory-motivated searches, as was the case with the search for the Higgs boson^{8,9}. Conversely, this approach may become a limiting factor in the absence of a strong theoretical prior. For this reason, there are several community efforts to investigate unsupervised machine learning (ML) techniques for new physics searches^{10,11}. These investigate the use of autoencoders (AEs) and variational autoencoders (VAEs) for offline processing^{12,13}, and therefore do not consider constraints such as resource use and latency. Early suggestions to use AEs in HEP for anomaly detection^{14,15} are not easily adapted to an L1T environment. For instance, refs. ^{14,15} require access to the momenta of all jet particle constituents, something that is not available now and will only be partly available (for example, first eight candidates) in the future. Refs. ^{16,17} propose integrating unsupervised learning algorithms in

the online selection system of the CMS and ATLAS experiments, in order to preserve rare events that would not otherwise be selected, in a special data stream.

While the primary focus on online unsupervised learning so far has been for the HLT, this strategy could be more effective if deployed in the L1T, that is, before any selection bias is introduced. Due to the extreme latency and computing resource constraints of the L1T, only relatively simple, mostly theory-motivated selection algorithms are currently deployed. These usually include requirements on the minimum energy of a physics object, such as a reconstructed lepton or a jet, effectively excluding lower-energy events from further processing. Instead, by deploying a new-physics model agnostic algorithm that selects events based on their degree of abnormality, we can collect data in a signal-model-independent way. Such an anomaly detection (AD) algorithm is required to have extremely low latency because of the restrictions imposed by the L1T.

Many recent efforts for translating ML algorithms into FPGA firmware are reviewed extensively in refs. ^{18–20}. However, many of these toolflows result in implementations that are not optimized for the L1T systems or do not apply to HEP AE architectures. For example, FINN^{21,22} focuses on dataflow-style implementations of convolutional neural networks (CNNs), which may not achieve the low latency and high throughput required for L1T applications. It is by construction limited to Xilinx FPGAs, while hls4ml backends targeting different HLS libraries (Quartus for Intel and Katapult for ASIC design) are under development. Other efforts, Conifer²³ (also developed by the hls4ml team) and fwXmachina²⁴, feature a custom implementation of boosted decision trees on FPGAs, which achieves the desired L1T constraints, but does not extend to neural network implementations.

¹European Organization for Nuclear Research (CERN), Geneva, Switzerland. ²Institute of Physics Belgrade, Belgrade, Serbia. ³Politecnico di Milano, Milan, Italy. ⁴Imperial College London, London, UK. ⁵Fermi National Accelerator Laboratory, Batavia, USA. ⁶California Institute of Technology, Pasadena, USA. ⁷University of California San Diego, La Jolla, USA. ⁸University of Illinois at Chicago, Chicago, USA. ✉e-mail: katya.govorkova@cern.ch

Recent developments of the hls4ml library allow us to consider the possibility of deploying AE-based AD algorithms on the FPGAs mounted on the L1T boards. The hls4ml library is an open-source software, developed to translate neural networks^{25–29} and boosted decision trees³⁰ into FPGA firmware. A fully on-chip implementation of the ML model is used in order to stay within the 1 μ s latency budget imposed by a typical L1T system. Additionally, the initiation interval (II) of the algorithm should be within 150ns, which is related to the bunch-crossing time for the upcoming period of the LHC operations⁵. Since there are several L1T algorithms deployed per FPGA, each of them should use much less than the available resources. With its interface to QKERAS³¹, hls4ml supports quantization-aware training (QAT)³², which makes it possible to drastically reduce the FPGA resource consumption while preserving accuracy. Using hls4ml, we can compress neural networks to fit the limited resources of an FPGA.

The aim of this work is the development of a fast algorithm to define a dataset enriched in anomalies, without using physics-motivated expectations about new physics to define the anomaly. Once collected, these data could be visually inspected or analysed with model-agnostic techniques, for example, those proposed in refs. ^{33,34}, or even with traditional model-dependent searches (provided an understanding of the bias imposed by the online selection on the offline event distribution). We focus on AEs, with specific emphasis on VAEs^{12,13}. We consider both fully connected and convolutional architectures, and discuss how to compress the model through pruning^{35–37}, the removal of unnecessary operations, and quantization^{26,38–45}, the reduction of the precision of operations.

As discussed in ref. ¹⁶, one can train (V)AE on a given data sample by minimizing a measure of the distance between the input and the output (the loss function). This strategy, which is very common when using (V)AEs for anomaly detection⁴⁶, brings practical challenges when considering a deployment on FPGAs. The use of high-level features is not optimal because it requires time-consuming data preprocessing. The situation is further complicated for VAEs, which require a random sampling from a Gaussian distribution in the latent space. Furthermore, one has to buffer the input data on chip while the output is generated by the FPGA processing in order to compute the distance afterwards. To deal with all of these aspects, we explore different approaches and compare the accuracy, latency and resource consumption of the various methods.

In addition, we discuss how to customize the model compression in order to better accommodate for unsupervised learning. Previously, we showed that QAT can result in a large reduction in resource consumption with minor accuracy loss for supervised algorithms^{28,32}. In this paper, we extend and adapt that compression workflow to deal with the specific challenge of compressing autoencoders used for AD. Several approaches are possible:

- Post-training quantization (PTQ)^{25,36,47–50}, consists of applying a fixed-point precision to a floating-point baseline model. This is the simplest quantization approach, typically resulting in good algorithm stability, at the cost of losing performance.
- QAT, consists of imposing the fixed-point precision constraint at training time, for example, using the QKERAS or Brevitas⁵¹ libraries. This approach typically allows one to limit the accuracy loss when imposing a higher level of quantization, finding a better weight configuration than what one can get with PTQ.
- Knowledge distillation with QAT changes the quantized-model optimization strategy by reframing the problem as knowledge distillation^{52–55}.
- Anomaly classification with QAT; approximated loss regression with QAT could be turned into a classification problem.

In this paper, we focus on the first two approaches, leaving the investigation of the other approaches to future work.

Data samples

This study follows the setup of refs. ^{16,56}. The dataset (with its definition and limitations) are taken from ref. ¹⁶. We adapt the data format to make it more consistent with inputs received in the L1T (as opposed to the HLT) and show that one can do at L1T what ref. ¹⁶ proposed for the HLT. Perhaps surprisingly, this is indeed possible due to recent progress made on deploying neural networks on FPGAs. We use a data sample that represents a typical proton–proton collision dataset that has been pre-filtered by requiring the presence of an electron or a muon with a transverse momentum $p_T > 23$ GeV and a pseudo-rapidity $|\eta| < 3$ (electron) and $|\eta| < 2.1$ (muon). These requirements were introduced to reduce the dataset size to a manageable level, such that we could generate it with our limited computing resources. In a real-life application, no p_T requirement of this kind would be applied. The η requirements would stay since they are intrinsic consequences of the detector geometry. In addition to the background-like sample, we consider the four benchmark new physics scenarios discussed in ref. ¹⁶:

- A leptoquark (LQ) with a mass of 80 GeV, decaying to a b quark and a τ lepton⁵⁷,
- A neutral scalar boson (A) with a mass of 50 GeV, decaying to two off-shell Z bosons, each forced to decay to two leptons: $A \rightarrow 4\ell$ ⁵⁸,
- A scalar boson with a mass of 60 GeV, decaying to two tau leptons: $h^0 \rightarrow \tau\tau$ ⁵⁹,
- A charged scalar boson with a mass of 60 GeV, decaying to a tau lepton and a neutrino: $h^\pm \rightarrow \tau\nu$ ⁶⁰.

These four processes are used to evaluate the accuracy of the trained models. A detailed description of the dataset can be found in ref. ⁶¹. In total, the background sample⁶² consists of 8 million events. Of these, 50% are used for training, 40% for testing and 10% for validation.

Autoencoder models

We consider two classes of architecture: one based on dense feed-forward neural networks (DNNs) and one using CNNs. Both start from the (p_T, η, ϕ) values for 18 reconstructed objects (ordered as 4 muons, 4 electrons and 10 jets), the ϕ and magnitude of the missing transverse energy (MET), forming together an input of shape (19, 3) where MET η values are zero-padded by construction (η is zero for transverse quantities). For events with fewer than the maximum number of muons, electrons or jets, the input is also zero-padded, as commonly done in the L1T algorithm logic.

In order to account for resource consumption and latency of the data pre-processing step, we use a batch normalization layer⁶³ as the first layer for each model. For both architectures, CNN and DNN, we consider both a plain AE and a VAE. In the AE, the encoder provides directly the coordinates of the given input, projected in the latent space. In the VAE, the encoder returns the mean values $\vec{\mu}$ and the standard deviation $\vec{\sigma}$ of the N -dimensional Gaussian distribution that represents the latent-space probability density function associated with a given event.

For the DNN model (shown on the top plot in Extended Data Fig. 1), all of the inputs are batch-normalized and passed through a stack of three fully connected layers, with 32, 16 and 3 nodes. The output of each layer is followed by a batch normalization layer and activated by a leaky ReLU function⁶⁴. The decoder consists of a stack of three layers, with 16, 32 and 57 nodes. As for the encoder, we use a batch normalization layer between the fully connected layer and its activation. The last layer has no activation function, while leaky ReLU is used for the others.

The CNN AE architecture is shown on the bottom plot in Extended Data Fig. 1. The encoder takes as input the single-channel 2D array of three-vector including the two MET-related features

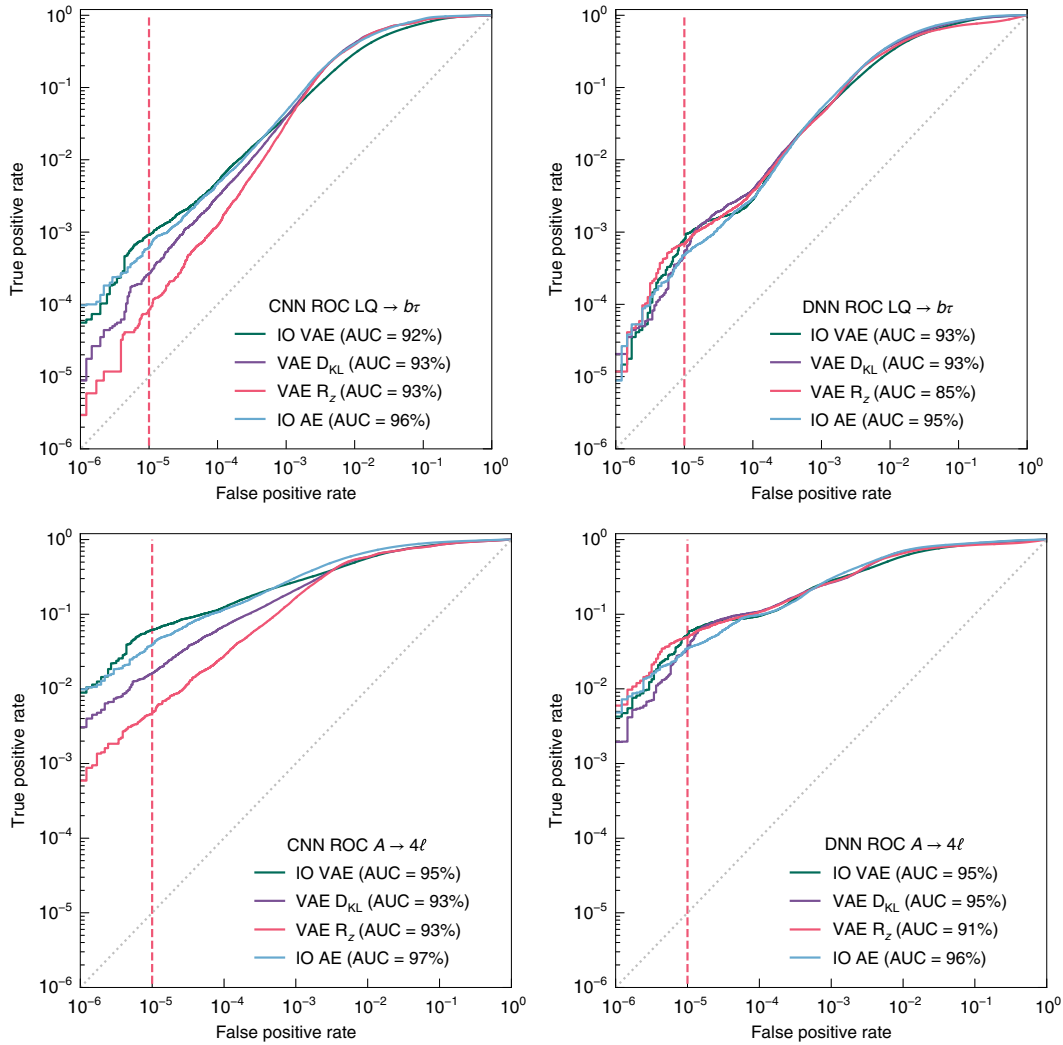


Fig. 1 | Model performance at floating-point precision. ROC curves of four AD scores (IO AD for AE and VAE models, R_z and D_{KL} ADs for the VAE models) for the CNN (left) and DNN (right) models, obtained from the two new physics benchmark models: LQ $\rightarrow b\tau$ (top) and A $\rightarrow 4\ell$ (bottom).

(magnitude and ϕ angle) and zeros for MET η , resulting in a total input size of $19 \times 3 \times 1$. It should be emphasized that we are not using image data, rather treating tabular data as a 2D image to make it possible to explore CNN architectures. The input is scaled by a batch normalization layer and then processed by a stack of two CNN blocks, each including a 2D convolutional layer followed by a ReLU⁶⁵ activation function. The first layer has $16 \ 3 \times 3$ kernels, without padding to ensure that p_T , η and ϕ inputs do not share weights. The second layer has $32 \ 3 \times 1$ kernels. Both layers have no bias parameters and a stride set to one. The output of the second CNN block is flattened and passed to a DNN layer, with eight neurons and no activation, which represents the latent space. The decoder takes this as input to a dense layer with 64 nodes and ReLU activation, and reshapes it into a $2 \times 1 \times 32$ table. The following architecture mirrors the encoder architecture with two CNN blocks with the same number of filters as in the encoder and with ReLU activation. Both are followed by an upsampling layer, in order to mimic the result of a transposed convolutional layer. Finally, one convolutional layer with a single filter and no activation function is added. Its output is interpreted as the AE-reconstructed input.

The CNN and DNN VAEs are derived from the AEs, including the $\vec{\mu}$ and $\vec{\sigma}$ Gaussian sampling in the latent space.

All models are implemented in TENSORFLOW, and trained on the background dataset by minimizing a customized mean squared

error (MSE) loss with the Adam⁶⁶ optimizer. In order to aid the network learning process, we use a dataset with standardized p_T as a target, so that all the quantities are $\mathcal{O}(1)$. To account for physical boundaries of η and ϕ , for those features a re-scaled tanh activation is used in the loss computation. In addition, the sum in the MSE loss is modified in order to ignore the zero-padding entries of the input dataset and the corresponding outputs. When training the VAE, the loss is changed to:

$$\mathcal{L} = (1 - \beta)\text{MSE}(\text{Output}, \text{Input}) + \beta D_{\text{KL}}(\vec{\mu}, \vec{\sigma}), \quad (1)$$

where MSE labels the reconstruction loss (also used in the AE training), D_{KL} is the Kullback–Leibler regularization term⁶⁷ usually adopted for VAEs

$$D_{\text{KL}}(\vec{\mu}, \vec{\sigma}) = -\frac{1}{2} \sum_i (\log(\sigma_i^2) - \sigma_i^2 - \mu_i^2 + 1), \quad (2)$$

and β is a hyperparameter defined in the range $[0, 1]$ ⁶⁸.

Both models are trained for 100 epochs with a batch size of 1,024, using early stopping if there is no improvement in the loss observed after ten epochs. All models are trained with floating point precision on an NVIDIA RTX2080 GPU. We refer to these as the baseline floating-point (BF) models.

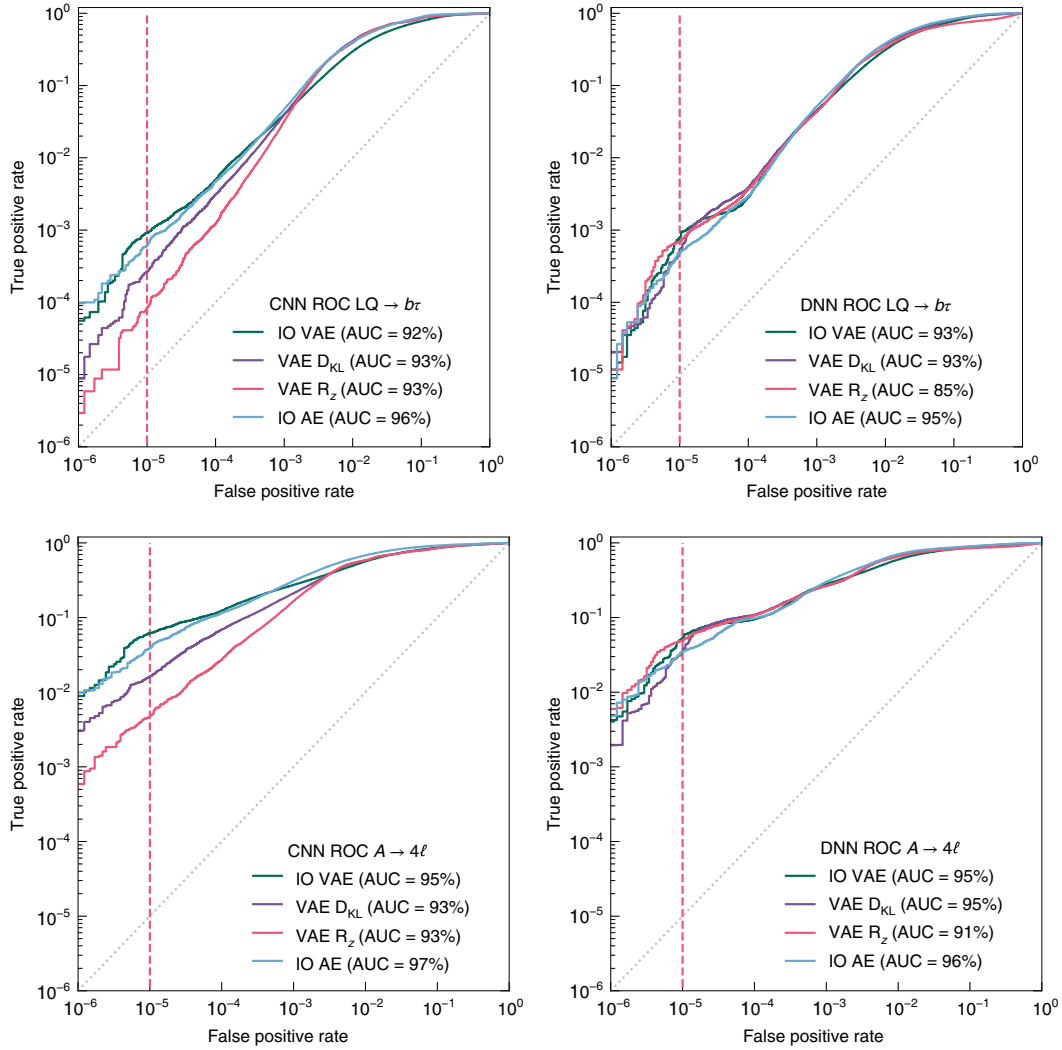


Fig. 2 | Model performance at floating-point precision. ROC curves of four AD scores (IO AD for AE and VAE models, R_z and D_{KL} ADs for the VAE models) for the CNN (left) and DNN (right) models, obtained from two new physics benchmark models: $h^\pm \rightarrow \tau\nu$ (top) and $h^0 \rightarrow \tau\tau$ (bottom).

Anomaly detection scores

An autoencoder is optimized to retain the minimal set of information needed to reconstruct an accurate estimate of the input. During inference, an autoencoder might have problems generalizing to topologies it was not exposed to during training. Selecting events where the autoencoder output is far from the given input is often seen as an effective AD algorithm. The simplest solution is to use the same metric that defines the training loss function. In our case, we use the modified MSE between the input and the output. We refer to this strategy as input–output (IO) AD.

In the case of a VAE deployed in the LIT, one cannot simply exploit an IO AD strategy since this would require sampling random numbers on the FPGA. One could generate pseudo-random numbers exploiting meta information (for example, the event number) or symmetries in data (for example, the ϕ coordinate of one of the objects). This might imply a limitation on the dimensionality of the latent space, which might impact performance. Moreover, one would have to store random numbers on the FPGA, which would consume resources and increase the latency. We did not explore this possibility further. Instead, we consider an alternative strategy by defining an AD score based on the $\vec{\mu}$ and $\vec{\sigma}$ values returned by the encoder (see equation (1)). In particular, we consider two options: the KL divergence term entering the VAE loss (see equation (2)) and the z -score of the origin $\vec{0}$ in the latent space with respect to

a Gaussian distribution centred at $\vec{\mu}$ with standard deviation $\vec{\sigma}$ (ref. ¹⁰):

$$R_z = \sum_i \frac{\mu_i^2}{\sigma_i^2}. \quad (3)$$

These two AD scores have several benefits we take advantage of: Gaussian sampling is avoided; we save significant resources and latency by not evaluating the decoder; and we do not need to buffer the input data for computation of the MSE. During the model optimization, we tune β so that we obtain (on the benchmark signal models) comparable performance for the D_{KL} AD score and the IO AD score of the VAE. In practice, one should train the model using real data, which might contain a very small fraction of signal. Previous studies have verified¹⁶ that small rates of signal contamination have little effect on the training. One would use simulated signals in the same manner as in this paper to tune model parameters. Such a procedure would not bias the architecture choice towards specific signals, given the low dependence of the optimal β value on the nature of the anomaly.

Performance at floating-point precision

The model performance is assessed using the four new physics benchmark signals. The anomaly-detection scores considered in

Table 1 | Performance assessment of the CNN and DNN models, for different AD scores and different new physics benchmark scenarios

Model	AD score	TPR @ FPR 10^{-5} (%)				AUC (%)			
		LQ \rightarrow $b\tau$	A \rightarrow 4ℓ	$h^\pm \rightarrow \nu$	$h^0 \rightarrow \tau\tau$	LQ \rightarrow $b\tau$	A \rightarrow 4ℓ	$h^\pm \rightarrow \nu$	$h^0 \rightarrow \tau\tau$
CNN VAE	IO	0.09	6.19	0.10	0.11	92	95	95	85
	D_{KL}	0.03	1.63	0.08	0.09	93	93	93	82
	R_z	0.01	0.48	0.04	0.04	93	93	93	82
CNN AE	IO	0.06	3.89	0.08	0.09	96	97	96	88
DNN VAE	IO	0.08	5.33	0.08	0.10	93	95	95	85
	D_{KL}	0.05	3.78	0.08	0.10	93	95	94	84
	R_z	0.07	4.90	0.07	0.10	85	91	87	74
DNN AE	IO	0.05	3.47	0.06	0.09	95	96	96	88

The best-performing autoencoder model for each anomaly is highlighted in bold.

this paper are IO AD for the AE models, R_z and D_{KL} ADs for the VAE models. For completeness, results obtained from the IO AD score of the VAE models are also shown. The receiver operating characteristic (ROC) curves in Figs. 1 and 2 show the true positive rate (TPR) as a function of the false positive rate (FPR), computed by changing the lower threshold applied on the different anomaly scores. We further quantify the AD performance quoting the area under the ROC curve (AUC) and the TPR corresponding to an FPR working point of 10^{-5} (see Table 1), which on this dataset corresponds to the reduction of the background rate to approximately 1,000 events per month.

Even if the VAE- D_{KL} TPR is smaller than the corresponding full-precision model for certain benchmark signals, the TPR values are similar after pruning. So, we conclude that D_{KL} can be used as an anomaly metric for the rest of this work. The R_z metric performs worse and is therefore not included in the following studies.

Model compression

We compress the BF model by pruning the dense and convolutional layers by 50% of their connections, following the a previously reported procedure²⁸. Pruning is enforced using the polynomial decay implemented in TENSORFLOW pruning API, a KERAS-based⁶⁹ interface consisting of a simple drop-in replacement of KERAS layers. A sparsity of 50% is targeted, meaning only 50% of the weights are retained in the pruned layers and the remaining ones are set to zero. The pruning is set to start from the fifth epoch of the training to ensure the model is closer to a stable minimum before removing weights deemed unimportant. By pruning the BF model layers to a target sparsity of 50%, the number of floating-point operations required when evaluating the model, can be significantly reduced. We refer to the resulting model as the baseline pruned (BP) model. For the VAE, only the encoder is pruned, since only that will be deployed on FPGA. The BP models are taken as a reference to evaluate the resource saving of the following compression strategies, including QAT and PTQ.

Furthermore, we perform a QAT of each model described in ‘Autoencoder models’, implementing them in the QKERAS library³². The bit precision is scanned between 2 and 16 with a 2-bit step. When quantizing a model, we also impose a pruning of the dense (convolutional) layers by 50%, as done for the DNN (CNN) BP models. The results of QAT are compared to results obtained by applying a fixed-point precision to a BP floating-point model (that is using PTQ), using the same bit precision scan.

Performance of the quantized models, both for QAT and PTQ, is assessed using the TPR obtained for an FPR of 10^{-5} for the given precision. The bottom plots in Fig. 3 and Extended Data Fig. 2 show ratios of QAT performance quantities obtained for each bit width

with respect to the BP model performance of the AE and VAE, respectively. The top plots show ratios of PTQ performance quantities obtained in the same manner as for QAT.

Based on these ratio plots, the precision used for the final model is chosen. The performance of the VAEs is not stable as a function of bit width, since the AD figure of merit used for inference (D_{KL}) is different from those minimized during the QAT training (VAE IO + D_{KL}). Therefore, we use PTQ compression for both DNN and CNN VAEs because they show stable results as a function of the bit width. For autoencoders, both quantization approaches show stable results, and therefore we choose quantization-aware training. For all the models a bit width of 8 is chosen, apart from the CNN VAE for which a bit width of 4 is found to be the best. The performance numbers for the chosen models are summarized in Table 2.

Porting the algorithm to FPGAs

The models described above are translated into firmware using hls4ml, then synthesized with Vivado HLS 2020.1⁷⁰, targeting a Xilinx Virtex UltraScale+ VU9P (xcvu9p-flgb2104-2-e) FPGA with a clock frequency of 200 MHz. In order to have fair resource and latency estimations, obtained from the HLS C simulation we have implemented custom layers in hls4ml, which in the case of AE computes the loss function between the input and network output and for VAE computes the D_{KL} term of the loss.

A summary of the accuracy, resource consumption, and latency for the QAT DNN and CNN BP AE models, and the PTQ DNN and CNN BP VAE models is shown in Table 3. We find the resources are less than about 12% of the available FPGA resources, except for the CNN AE, which uses up to 47% of the look-up tables (LUTs). Moreover, the latency is less than about 365ns for all models except the CNN AE, which has a latency of 1,480 ns. The II for all models is within the required 115ns, again except the CNN AE. Based on these, both types of architectures with both types of autoencoders are suitable for application at the LHC L1T, except for the CNN AE, which consumes too much of the resources.

Since the performance of all the models under study are of a similar level, we choose the ‘best’ model based on the smallest resource consumption, which turns out to be DNN VAE. This model was integrated into the emp-fwk infrastructure firmware for LHC trigger boards⁷¹, targeting a Xilinx VCU118 development kit, with the same VU9P FPGA as previously discussed. Data were loaded into onboard buffers mimicking the manner in which data arrives from optical fibres in the L1T system. The design was operated at 240 MHz, and the model predictions observed at the output were consistent with those captured from the HLS C simulation. For this model we also provide resource and latency estimates for a Xilinx Virtex 7 690 FPGA, which is the FPGA most widely used in the current CMS trigger.

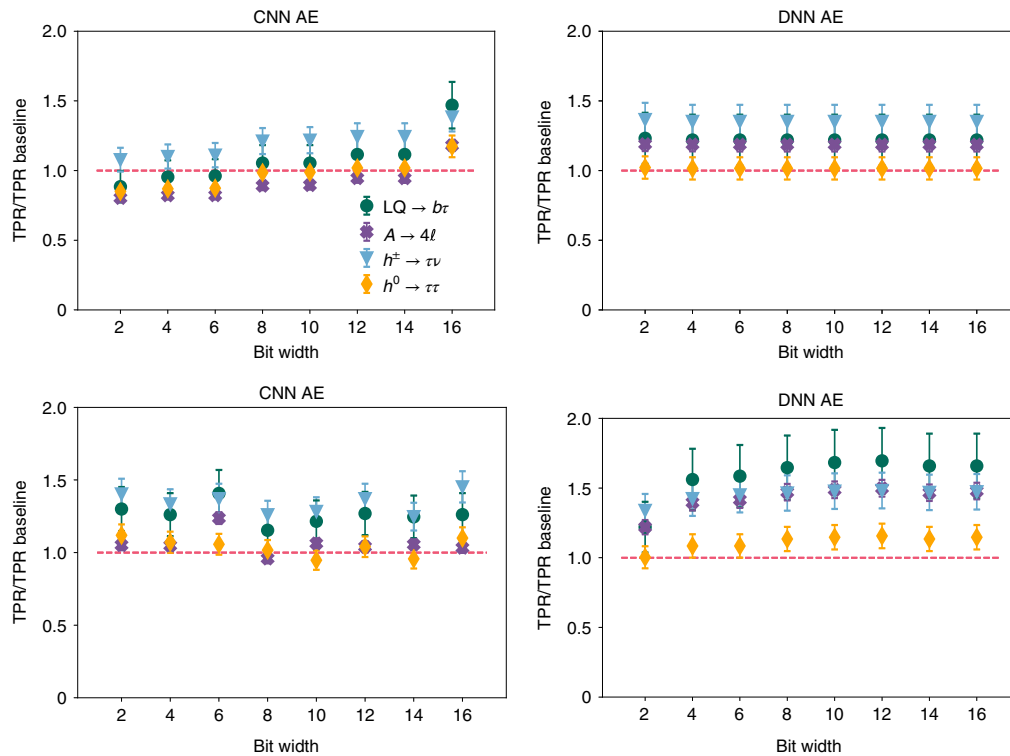


Fig. 3 | Compressed model performance. TPR ratios versus model bit width for the AE CNN (left) and DNN (right) models tested on four new physics benchmark models, using mean squared error as figure of merit for PTQ (top) and QAT (bottom) strategies.

Table 2 | Performance assessment of the quantized and pruned CNN and DNN models, for different AD scores and different new physics benchmark scenarios

Model	AD score	TPR @ FPR 10 ⁻⁵ [%]				AUC[%]			
		LQ → bτ	A → 4ℓ	h [±] → ν	h ⁰ → ττ	LQ → bτ	A → 4ℓ	h [±] → ν	h ⁰ → ττ
CNN AE QAT 4 bits	IO	0.09	5.96	0.10	0.13	94	96	96	88
CNN VAE PTQ 8 bits	D _{KL}	0.05	2.56	0.06	0.12	84	84	85	71
DNN AE QAT 8 bits	IO	0.08	5.48	0.09	0.11	95	96	96	88
DNN VAE PTQ 8 bits	D _{KL}	0.08	3.41	0.09	0.08	92	94	94	81

Conclusions

We discussed how to extend new physics detection strategies at the LHC with autoencoders deployed in the L1T infrastructure of the experiments. In particular, we show how one could deploy a deep neural network or convolutional neural network AE on a FPGA using the hls4ml library, within a $\mathcal{O}(1)\mu\text{s}$ latency and with small resource utilization once the model is quantized and pruned. We show that one can retain accuracy by compressing the model at training time. Moreover, we discuss different strategies to identify potential anomalies. We show that one could perform the AD with a VAE using the projected representation of a given input in the latent space, which has several advantages for an FPGA implementation: (1) no need to sample Gaussian-distributed pseudorandom numbers (preserving the deterministic outcome of the trigger decision) and (2) no need to run the decoder in the trigger, resulting in a significant resource saving.

The DNN (V)AE models use less than 5% of the Xilinx VU9P resources and the corresponding latency is within 130ns, while the CNN VAE uses less than 12% and the corresponding latency is 365ns. All three models have the initiation interval within the strict limit imposed by the frequency of bunch crossing at the LHC. With this work, we have identified and finalized the necessary ingredients

to deploy (V)AEs in the L1T of the LHC experiments for Run 3 to accelerate the search for unexpected signatures of new physics.

The aim is to use these algorithms in the trigger in order to create a catalogue of anomalous events that researchers could explore, for example, with clustering techniques. Furthermore, one could perform traditional data analysis, provided a (non-trivial) understanding of the effect of the trigger selection on the kinematic distribution. In presence of a good description of the loss distribution, the approach used in ref. ⁷² could be adopted.

Data availability

The data used in this study are openly available at Zenodo^{57–60,62}.

Code availability

The QKeras library is available at github.com/google/qkeras, where the work presented here is using QKeras version 0.9.0. The hls4ml library with custom layers used in the paper are under AE_L1_paper branch and available at https://github.com/fastmachinelearning/hls4ml/tree/AE_L1_paper.

Received: 12 August 2021; Accepted: 6 January 2022; Published online: 23 February 2022

Table 3 | Resource utilization and latency for the quantized and pruned DNN and CNN (V)AE models

Model	Hardware	DSP [%]	LUT [%]	FF [%]	BRAM [%]	Latency [ns]	II [ns]
DNN AE QAT 8 bits	Xilinx VU9P	2	5	1	0.5	130	5
CNN AE QAT 4 bits	Xilinx VU9P	8	47	5	6	1,480	895
DNN VAE ^a PTQ 8 bits	Xilinx VU9P	1	3	0.5	0.3	80	5
DNN VAE PTQ 8 bits	Xilinx V7-690	3	9	3	0.4	205	5
CNN VAE PTQ 8 bits	Xilinx VU9P	10	12	4	2	365	115

Resources are based on the Vivado estimates from Vivado HLS 2020.1 for a clock period of 5ns on Xilinx VU9P. ^aFor the DNN VAE model, resources estimation is also provided based on Xilinx V7-690

References

- LHC Machine. JINST 3, S08001 (2008).
- Aad, G. et al. The ATLAS Experiment at the CERN Large Hadron Collider. *J. Instrum.* 3, S08003 (2008).
- Chatrchyan, S. et al. The CMS Experiment at the CERN LHC. *J. Instrum.* 3, S08004 (2008).
- Sirunyan, A. M. et al. Performance of the CMS Level-1 trigger in proton-proton collisions at $\sqrt{s} = 13$ TeV. *J. Instrum.* 15, P10017 (2020).
- The Phase-2 upgrade of the CMS Level-1 trigger. CMS Technical Design Report CERN-LHCC-2020-004 CMS-TDR-021* (2020).
- Aad, G. et al. Operation of the ATLAS trigger system in Run 2. *J. Instrum.* 15, P10004 (2020).
- Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System. ATLAS Technical Design Report CERN-LHCC-2017-020 ATLAS-TDR-029* (2017).
- Aad, G. et al. Observation of a new particle in the search for the standard model Higgs boson with the ATLAS detector at the LHC. *Phys. Lett. B* 716, 1 (2012).
- Chatrchyan, S. et al. Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Phys. Lett. B* 716, 30 (2012).
- Aarrestad, T. et al. The dark machines anomaly score challenge: Benchmark data and model independent event classification for the large hadron collider. *SciPost Phys.* 12, 2542 (2022).
- Kasieczka, G. et al. The LHC olympics 2020: A community challenge for anomaly detection in high energy physics. *Rep. Prog. Phys.* 84, 124201 (2021).
- Kingma, D. P. & Welling, M. Auto-encoding variational Bayes. Preprint at <https://arxiv.org/abs/1312.6114> (2014).
- Rezende, D. J., Mohamed, S. & Wierstra, D. Stochastic backpropagation and approximate inference in deep generative models. Preprint at <https://arxiv.org/abs/1401.4082> (2014).
- Heimel, T., Kasieczka, G., Plehn, T. & Thompson, J. M. QCD or What? *SciPost Phys.* 6, 30 (2019).
- Farina, M., Nakai, Y. & Shih, D. Searching for new physics with deep autoencoders. *Phys. Rev. D* 101, 075021 (2020).
- Cerri, O. et al. Variational autoencoders for new physics mining at the Large Hadron Collider. *J. High Energy Phys.* 2019, 36 (2019).
- Knapp, O. et al. Adversarially Learned Anomaly Detection on CMS Open Data: re-discovering the top quark. *Eur. Phys. J. Plus* 136, 236 (2021).
- Venieris, S. I., Kouris, A. & Bougiani, C.-S. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. Preprint at <https://arxiv.org/abs/1803.05900> (2018).
- Guo, K., Zeng, S., Yu, J., Wang, Y. & Yang, H. A survey of FPGA-based neural network inference accelerators. <https://arxiv.org/abs/1712.08934> (2019).
- 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 (2019).
- 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 (ACM, 2017).
- Blott, M. et al. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. Preprint at <https://arxiv.org/abs/1809.04570> (2018).
- Summers, S. et al. Fast inference of boosted decision trees in FPGAs for particle physics. *J. Instrum.* 15, P05026 (2020).
- Hong, T. M. et al. Nanosecond machine learning event classification with boosted decision trees in FPGA for high energy physics. *J. Instrum.* 16, P08016 (2021).
- Duarte, J. et al. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum.* 13, P07027 (2018).
- Ngadiuba, J. et al. Compressing deep neural networks on FPGAs to binary and ternary precision with HLS4ML. *Mach. Learn. Sci. Technol.* 2, 2632 (2020).
- 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 (2020).
- Aarrestad, T. et al. Fast convolutional neural networks on FPGAs with HLS4ML. *Mach. Learn. Sci. Technol.* 2, 045015 (2021).
- Heintz, A. et al. Accelerated charged particle tracking with graph neural networks on FPGAs. In *34th Conference on Neural Information Processing Systems* (2020).
- Summers, S. et al. Fast inference of boosted decision trees in FPGAs for particle physics. *J. Instrum.* 15, P05026 (2020).
- Coelho, C. Qkeras <https://github.com/google/qkeras> (2019).
- Coelho, C. N. et al. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nat. Mach. Intell.* 3, 675–686 (2021).
- D'Agnolo, R. T. & Wulzer, A. Learning new physics from a machine. *Phys. Rev. D* 99, 015014 (2019).
- Mikuni, V., Nachman, B. & Shih, D. Online-compatible unsupervised non-resonant anomaly detection. Preprint at <https://arxiv.org/abs/2111.06417> (2021).
- LeCun, Y., Denker, J. S. & Solla, S. A. Optimal brain damage. In *Advances in Neural Information Processing Systems* (ed. Touretzky, D. S.) Vol. 2, 598 (Morgan-Kaufmann, 1990).
- Han, S., Mao, H. & Dally, W. J. Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. In *4th Int. Conf. Learning Representations* (ed. Bengio, Y. & LeCun, Y.) (2016).
- Blalock, D., Ortiz, J. J. G., Frankle, J. & Guttat, J. What is the state of neural network pruning? In *Proc. Machine Learning and Systems* Vol. 2, 129 (2020).
- Moons, B., Goetschalckx, K., Berckelaer, N. V. & Verhelst, M. Minimum energy quantized neural networks. In *2017 51st Asilomar Conf. Signals, Systems, and Computers* (ed. Matthews, M. B.) 1921 (2017).
- Courbariaux, M., Bengio, Y. & David, J.-P. BinaryConnect: Training deep neural networks with binary weights during propagations. In *Adv. Neural Information Processing Systems* (eds. Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M. & Garnett, R.) Vol. 28, 3123 (Curran Associates, 2015).
- 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* (eds. Ferrari, V., Hebert, M., Sminchisescu, C. & Weiss, Y.) (2018).
- 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 (2018).
- Rastegari, M., Ordonez, V., Redmon, J. & Farhadi, A. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *14th European Conf. Computer Vision* 525 (Springer, 2016).
- Micikevicius, P. et al. Mixed precision training. In *6th Int. Conf. Learning Representations* (2018).
- Zhuang, B., Shen, C., Tan, M., Liu, L. & Reid, I. Towards effective low-bitwidth convolutional neural networks. In *2018 IEEE/CVF Conf. Computer Vision and Pattern Recognition* 7920 (2018).
- Wang, N., Choi, J., Brand, D., Chen, C.-Y. & Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. In *Adv. Neural Information Processing Systems* (eds. Bengio, S. et al.) Vol. 31, 7675 (Curran Associates, 2018).
- An, J. & Cho, S. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture IE* 2, 1–18 (2015).
- Nagel, M., van Baalen, M., Blankevoort, T. & Welling, M. Data-free quantization through weight equalization and bias correction. In *2019 IEEE/CVF International Conf. Computer Vision* 1325 (2019).
- Meller, E., Finkelstein, A., Almog, U. & Grobman, M. Same, same but different: Recovering neural network quantization error through weight factorization. In *Proc. 36th International Conf. Machine Learning* (eds. Chaudhuri, K. & Salakhutdinov, R.) Vol. 97, 4486 (PMLR, 2019).

49. Zhao, R., Hu, Y., Dotzel, J., Sa, C. D. & Zhang, Z. Improving neural network quantization without retraining using outlier channel splitting. In *Proc. 36th Int. Conference on Machine Learning* (eds. Chaudhuri, K. & Salakhutdinov, R.) Vol. 97, 7543 (PMLR, 2019).
50. Banner, R., Nahshan, Y., Hoffer, E. & Soudry, D. Post-training 4-bit quantization of convolution networks for rapid-deployment. In *Adv. Neural Information Processing Systems* (eds. Wallach, H. et al.) Vol. 32, 7950 (Curran Associates, 2019).
51. Pappalardo, A. brevitas <https://github.com/Xilinx/brevitas> (2020).
52. Shin, S., Boo, Y. & Sung, W. Knowledge distillation for optimization of quantized deep neural networks. In *2020 IEEE Workshop on Signal Processing Systems* (2020).
53. Polino, A., Pascanu, R. & Alistarh, D. Model compression via distillation and quantization. In *Int. Conf. Learning Representations* (2018).
54. Gao, M. et al. An embarrassingly simple approach for knowledge distillation. Preprint at <https://arxiv.org/abs/1812.01819> (2019).
55. Mishra, A. & Marr, D. Apprentice: using knowledge distillation techniques to improve low-precision network accuracy. In *Int. Conf. Learning Representations* (2018).
56. Nguyen, T. Q. et al. Topology classification with deep learning to improve real-time event selection at the LHC. *Comput. Softw. Big Sci.* **3**, 12 (2019).
57. Govorkova, E. et al. Unsupervised new physics detection at 40 mhz: LQ \rightarrow b τ signal benchmark dataset. *Zenodo* <https://doi.org/10.5281/zenodo.5055454> (2021).
58. Govorkova, E. et al. Unsupervised new physics detection at 40 mhz: A \rightarrow 4 leptons signal benchmark dataset. *Zenodo* <https://doi.org/10.5281/zenodo.5046446> (2021).
59. Govorkova, E. et al. Unsupervised new physics detection at 40 mhz: $h^0 \rightarrow \tau\tau$ signal benchmark dataset. *Zenodo* <https://doi.org/10.5281/zenodo.5061633> (2021).
60. Govorkova, E. et al. Unsupervised new physics detection at 40 mhz: $h^+ \rightarrow \tau\nu$ signal benchmark dataset. *Zenodo* <https://doi.org/10.5281/zenodo.5061688> (2021).
61. Govorkova, E. et al. LHC physics dataset for unsupervised new physics detection at 40 MHz. Preprint at <https://arxiv.org/abs/2107.02157> (2021).
62. Govorkova, E. et al. Unsupervised new physics detection at 40 mhz: training dataset. *Zenodo* <https://doi.org/10.5281/zenodo.5046389> (2021).
63. Ioffe, S. & Szegedy, C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proc. 32nd International Conference on Machine Learning* (eds. Bach, F. & Blei, D.) Vol. 37, 448 (PMLR, 2015).
64. Maas, A. L., Hannun, A. Y. & Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing* (2013).
65. Nair, V. & Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *ICML* (eds. Fürnkranz, J. & Joachims, T.) 807 (Omnipress, 2010).
66. Kingma, D. P. & Ba, J. Adam: A Method for Stochastic Optimization. Preprint at <https://arxiv.org/abs/1412.6980> (2014).
67. Joyce, J. M. in *International Encyclopedia of Statistical Science* 720–722 (Springer, 2011); https://doi.org/10.1007/978-3-642-04898-2_327
68. Higgins, I. et al. beta-vae: Learning basic visual concepts with a constrained variational framework (2016).
69. Chollet, F. et al. Keras <https://keras.io> (2015).
70. Xilinx. Vivado design suite user guide: High-level synthesis. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf (2020).
71. EMP Collaboration. emp-fwk homepage. <https://serenity.web.cern.ch/serenity/emp-fwk/> (2019).
72. D'Agnolo, R. T. & Wulzer, A. Learning new physics from a machine. *Phys. Rev. D* **99**, 015014 (2019).

Acknowledgements

This work is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 772369) and the ERC-POC programme (grant no. 996696).

Author contributions

V.L., M.P., A.A.P., N.G., M.G., S.S., J.D. and Z.W. conceived and designed the hls4ml software library. M.P., T.Q.N. and Z.W. designed and prepared the dataset format. E.G., E.P., T.A., T.J., V.L., M.P., J.N., T.Q.N. and Z.W. designed and implemented autoencoders in hls4ml. E.G., E.P., T.A., T.J., M.P. and J.D. wrote the paper.

Competing interests

The authors declare no competing interests.

Additional information

Extended data is available for this paper at <https://doi.org/10.1038/s42256-022-00441-3>.

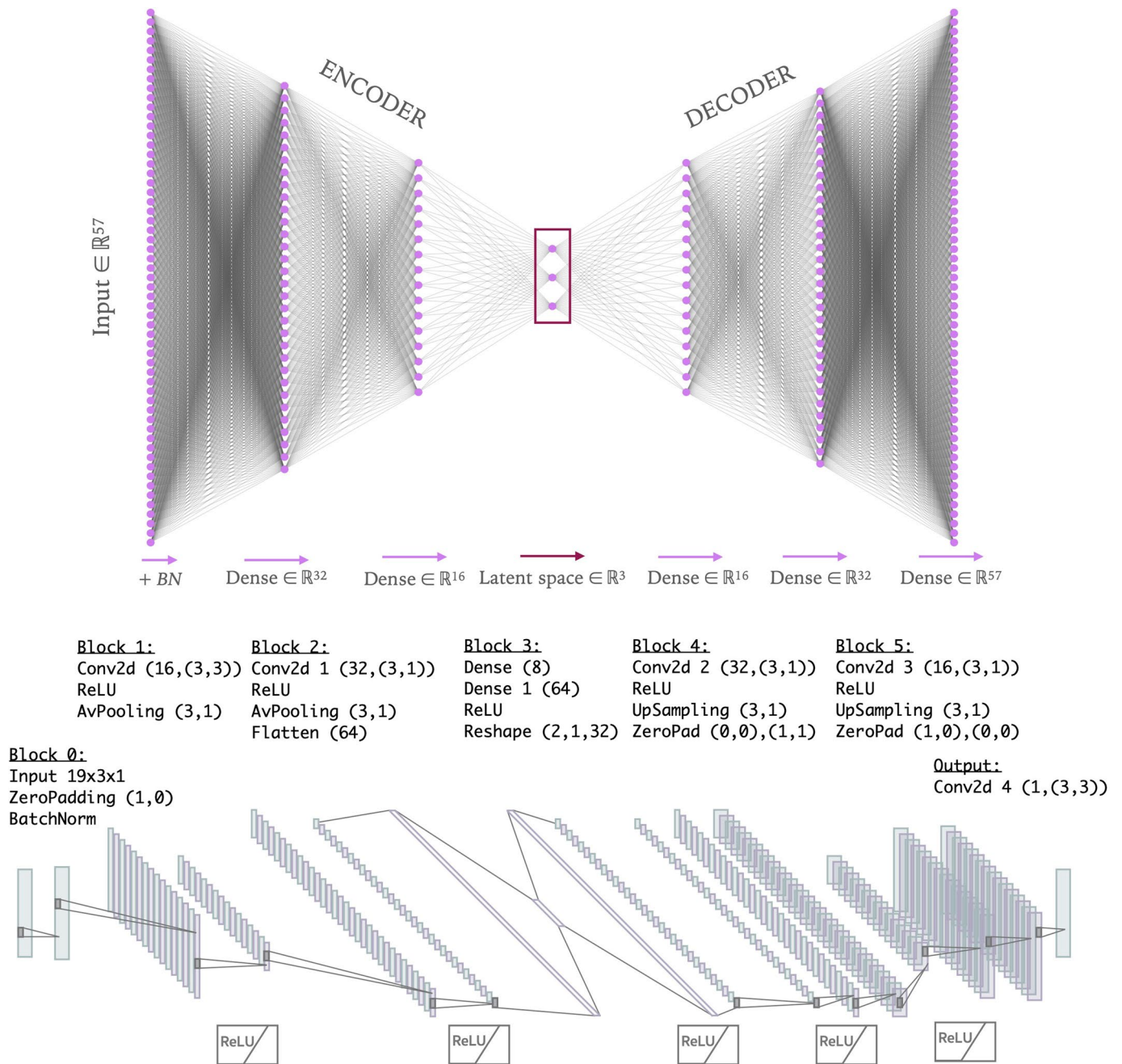
Correspondence and requests for materials should be addressed to Ekaterina Govorkova.

Peer review information *Nature Machine Intelligence* thanks the anonymous reviewers for their contribution to the peer review of this work.

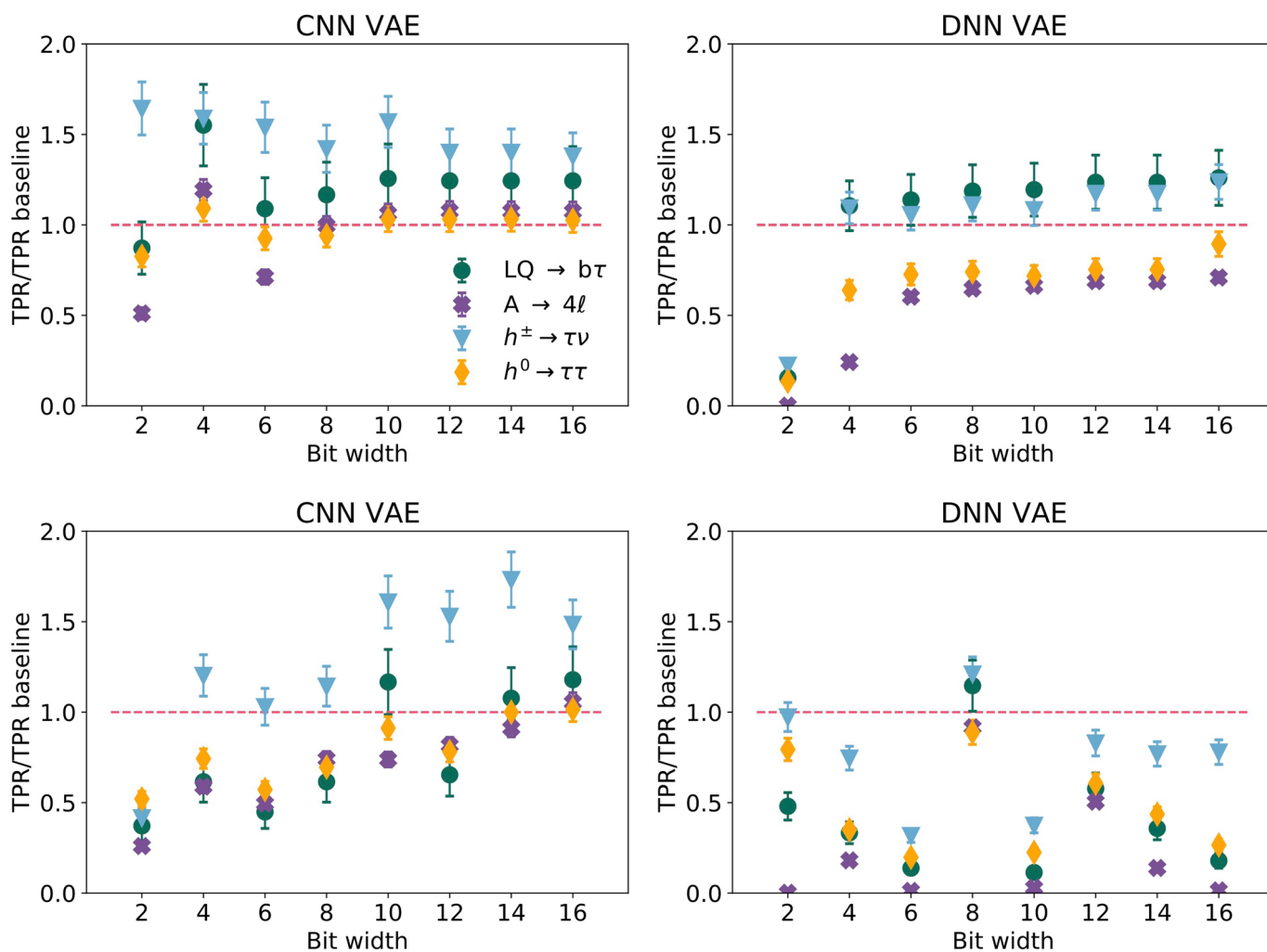
Reprints and permissions information is available at www.nature.com/reprints.

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

© The Author(s), under exclusive licence to Springer Nature Limited 2022






Extended Data Fig. 1 | Network architectures. Network architecture for the DNN AE (top) and CNN AE (bottom) models. The corresponding VAE models are derived introducing the Gaussian sampling in the latent space, for the same encoder and decoder architectures (see text).



Extended Data Fig. 2 | TPR ratios for different bit width. TPR ratios versus model bit width for the VAE CNN (left) and DNN (right) models tested on four new physics benchmark models, using D_{KL} as figure of merit for PTQ (top) and QAT (bottom) strategies.



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

Claudionor N. Coelho Jr¹, Aki Kuusela², Shan Li², Hao Zhuang², Jennifer Ngadiuba³,
Thea Klæboe Aarrestad⁴  , Vladimir Loncar^{4,5}, Maurizio Pierini⁴, Adrian Alan Pol⁴  and
Sioni Summers⁴

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 nanoseconds³. 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 CERN⁴. 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

¹Palo Alto Networks, Palo Alto, CA, USA. ²Google LLC, Mountain View, CA, USA. ³California Institute of Technology (Caltech), Pasadena, CA, USA.

⁴European Organization for Nuclear Research (CERN), Geneva, Switzerland. ⁵Institute 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 AutoQKeras⁶⁰, where simple drop-in replacement of Keras³² 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, hls4ml¹³, 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 $\mathcal{O}(1)\mu\text{s}$ 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 $\mathcal{O}(1)\mu\text{s}$, 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—hls4ml¹³. 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 TensorFlow³³, Keras³², PyTorch³⁴ and ONNX³⁵.

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 $\mathcal{O}(1)\mu\text{s}$ 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 HLS⁴⁰ and Mentor Catapult HLS⁴¹, 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 networks⁴².

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 DNNs⁴³ 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 model³⁰. 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 FINN⁴⁴ and FINN-R⁴⁵ 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, Brevitas⁴⁶, based on PyTorch model formats. The LogicNets design flow⁴⁷, 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-DNN⁴⁸ framework takes TensorFlow³³-described DNNs as input and maps them onto FPGAs. The open-source alternative, DNNWeaver⁴⁹, automatically generates accelerator Verilog code using optimized templates. Other frameworks focusing on the mapping of convolutional architectures onto efficient hardware design include Snowflake⁵⁰, fpgaConvNet^{51–53} and ref. ⁵⁴. For other work on FPGA DNN inference, we refer to refs. ^{36–39,55}. TensorFlow Lite⁵⁶ 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 HAQ²⁸, a hardware-aware automated framework for quantization is introduced. The automatization 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 dataset⁵⁸ 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 interface⁵⁹, 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.

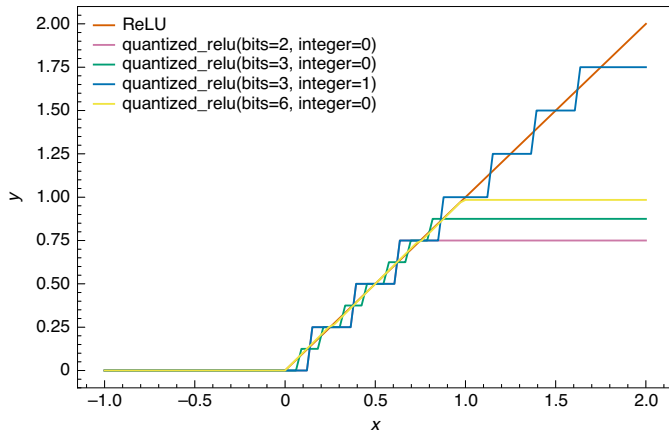


Fig. 1 | Quantized ReLU function in QKeras. The `quantized_relu` function as implemented in QKeras for 2-bit (purple), 3-bit (green and blue) and 6-bit (yellow) precision and for 0 or 1 integer bits. The unquantized ReLU function is shown for comparison (orange).

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.

Keras²² 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 QKeras²⁰: 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 estimator¹⁹, 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 empirical variance σ , while the empirical 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 $\pm \alpha$. 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:

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

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 ReLU⁶¹. 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 repository⁶⁰.

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 HAQ²⁸, 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 analysis⁶², 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, $E_{\text{layer}}^{\text{Real}} = k_1 \times E_{\text{layer}} + k_2$, 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 $E_{L1}^{\text{Real}} > E_{L2}^{\text{Real}}$ if, and only if, the estimated energy $E_{L1} > E_{L2}$.

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								$\frac{E}{E_{Q6}}$	Bits Bits _{Q6}
		Dense	ReLU	Dense	ReLU	Dense	ReLU	Dense	Softmax		
QE	72.3	$\langle 4, 0 \rangle$	$\langle 4, 2 \rangle$	Ternary	$\langle 3, 1 \rangle$	$\langle 2, 1 \rangle$	$\langle 4, 2 \rangle$	w: Stoc. bin. b: $\langle 8, 3 \rangle$	$\langle 16, 6 \rangle$	0.27	0.18
QB	72.8	$\langle 4, 0 \rangle$	$\langle 4, 2 \rangle$	Stoc. bin.	$\langle 4, 2 \rangle$	Ternary	$\langle 3, 1 \rangle$	Stoc. bin.	$\langle 16, 6 \rangle$	0.25	0.17
Q6	74.8	$\langle 6, 0 \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 0 \rangle$	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

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

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 hyperparameter search in Keras Tuner⁶⁴ using random search, hyperband⁶⁵ 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 = $\lceil \log_2 \left(\frac{\sum |w|}{N} \right) \rceil$, where w are the filter

coefficients and N is the number of coefficients, we will end up with the same filter binary $([-0.3, 0.2, 0.5, 0.15]) = \text{binary}([-0.5, 0.4, 0.1, 0.65]) = [-1, 1, 1, 1] \times 0.5$. 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 ResNet⁶⁶, 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.

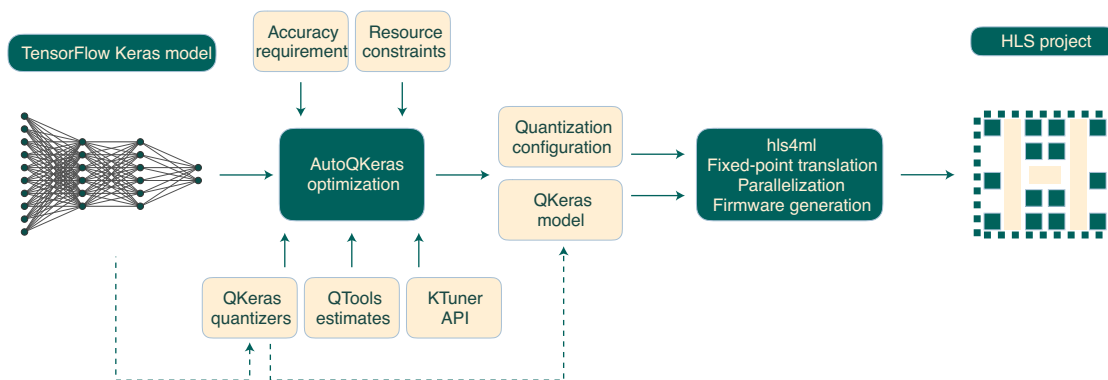


Fig. 2 | The QKeras and hls4ml workflow. The full workflow starting from a baseline TensorFlow Keras Model, which is then converted into an optimally quantized equivalent through QKeras and AutoQKeras. This model is then translated into highly parallel firmware with hls4ml.

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 loss⁵⁷ using the Adam optimizer⁶⁸. 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 $\times 1.5$ 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 (%)	$\frac{E_{QK}}{E_{Q6}}$	$\frac{P_{HLS}}{P_{HLS}(Q6)}$
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)	-	-

^aNot evaluated. ^bUsing a clock frequency of 384 MHz. ^cThe 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 LogicNets⁴⁷, 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 bandwidth-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,

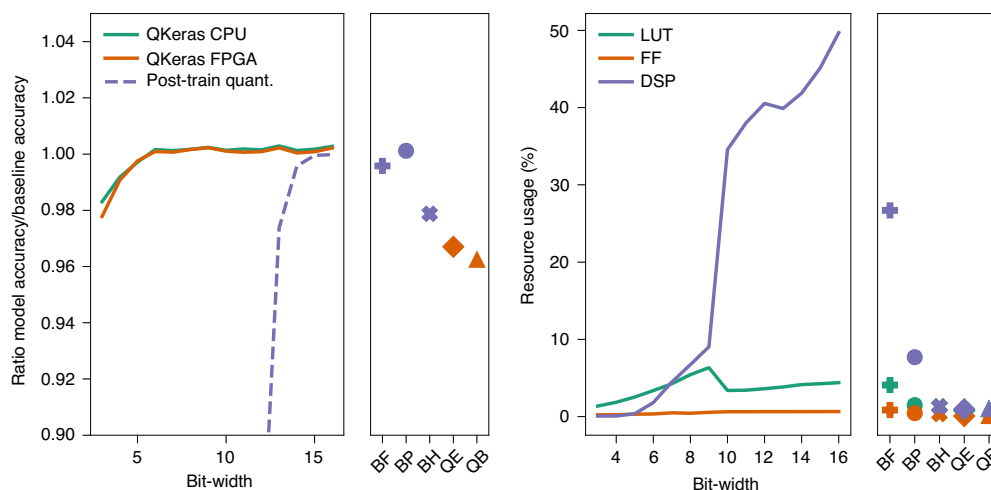


Fig. 3 | Performance on a Xilinx VU9P FPGA. Relative accuracy (left) and resource utilization (right) as a function of bit-width. The right-hand panel shows the metrics for the heterogeneously quantized models. The relative accuracy is evaluated with respect to the floating-point baseline model. Resources 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 dataset⁷⁰, 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 μ s. For larger convolutional architectures like ResNet⁶⁶, 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 $\mathcal{O}(1)$ μ s.

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 $\mathcal{O}(1)$ μ s 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 ReLU⁶¹ 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 MobileNet⁷³. 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 network⁷⁴, 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 weights⁷⁵. 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 Zenodo⁵⁸ 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 $\geq 0.2.1$ 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

- Lin, S.-C. et al. The architectural implications of autonomous driving: constraints and acceleration. *ACM SIGPLAN Notices* **53**, 751–766 (2018).
- 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
- 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).
- The LHC Study Group. *The Large Hadron Collider, Conceptual Design*. Technical Report CERN/AC/95-05 (CERN, 1995).
- 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).
- landola, F. N. et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5-MB model size. Preprint at <https://arxiv.org/pdf/1602.07360.pdf> (2016).
- Howard, A. G. et al. MobileNets: efficient convolutional neural networks for mobile vision applications. Preprint at <https://arxiv.org/pdf/1704.04861.pdf> (2017).
- 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).
- 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).
- Howard, A. et al. Searching for MobileNetV3. In *Proc. IEEE International Conference on Computer Vision* 1314–1324 (IEEE, 2019).
- 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).
- 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).
- Duarte, J. et al. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum.* **13**, P07027 (2018).
- 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).
- 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).
- 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).
- 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).
- Moons, B., Goetschalckx, K., Van Berckelaer, N. & Verhelst, M. Minimum energy quantized neural networks. In *51st Asilomar Conference on Signals, Systems, and Computers* 1921–1922 (ACSSC, 2017).
- 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).
- 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).
- Li, F. & Liu, B. Ternary weight networks. Preprint at <https://arxiv.org/pdf/1605.04711.pdf> (2016).
- 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).
- 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).
- 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).
- Micikevicius, P. et al. Mixed precision training. In *International Conference on Learning Representations (ICLR)*, 2018.
- 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).
- 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>.

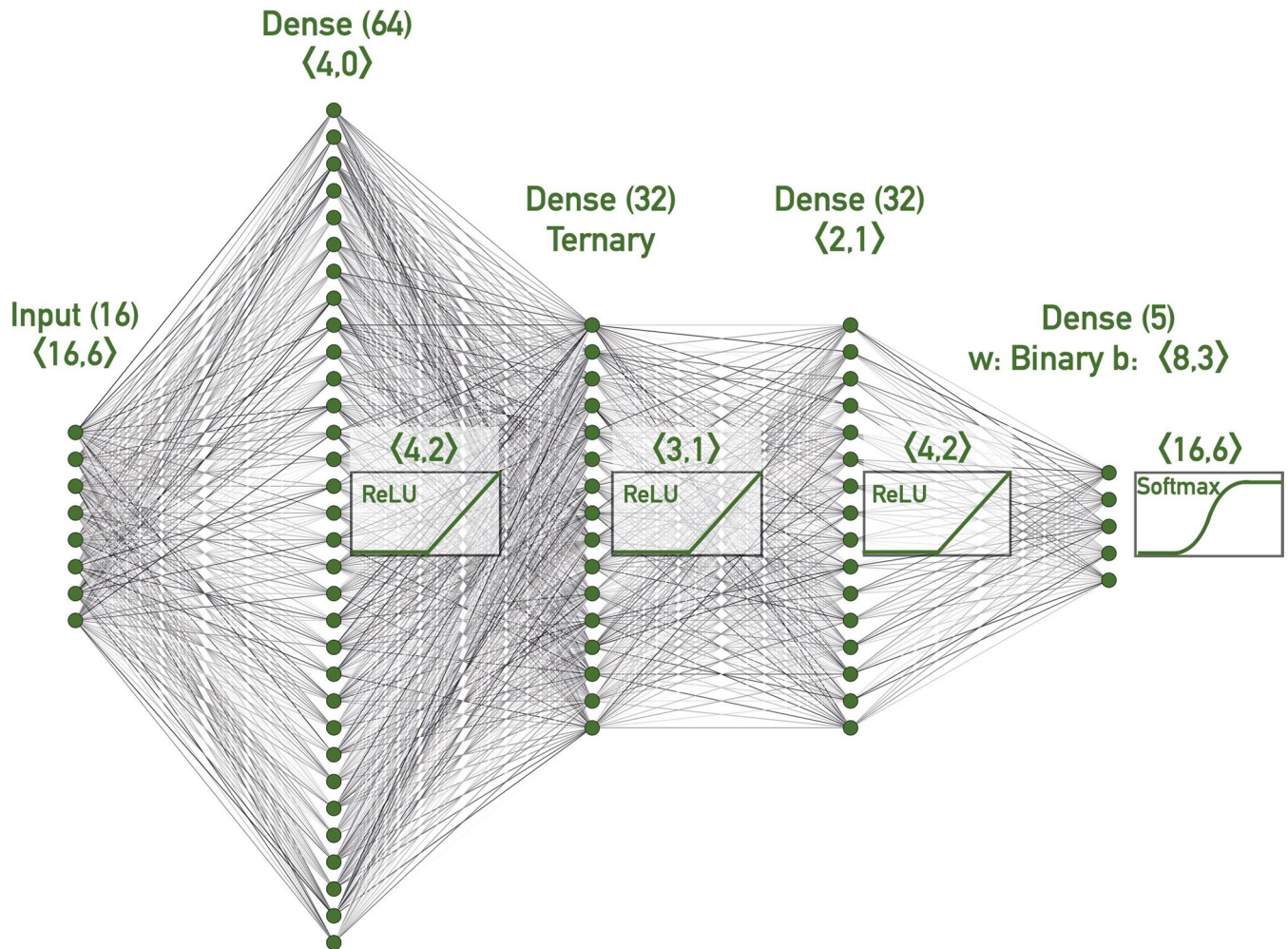
Correspondence and requests for materials should be addressed to T.K.A.

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.

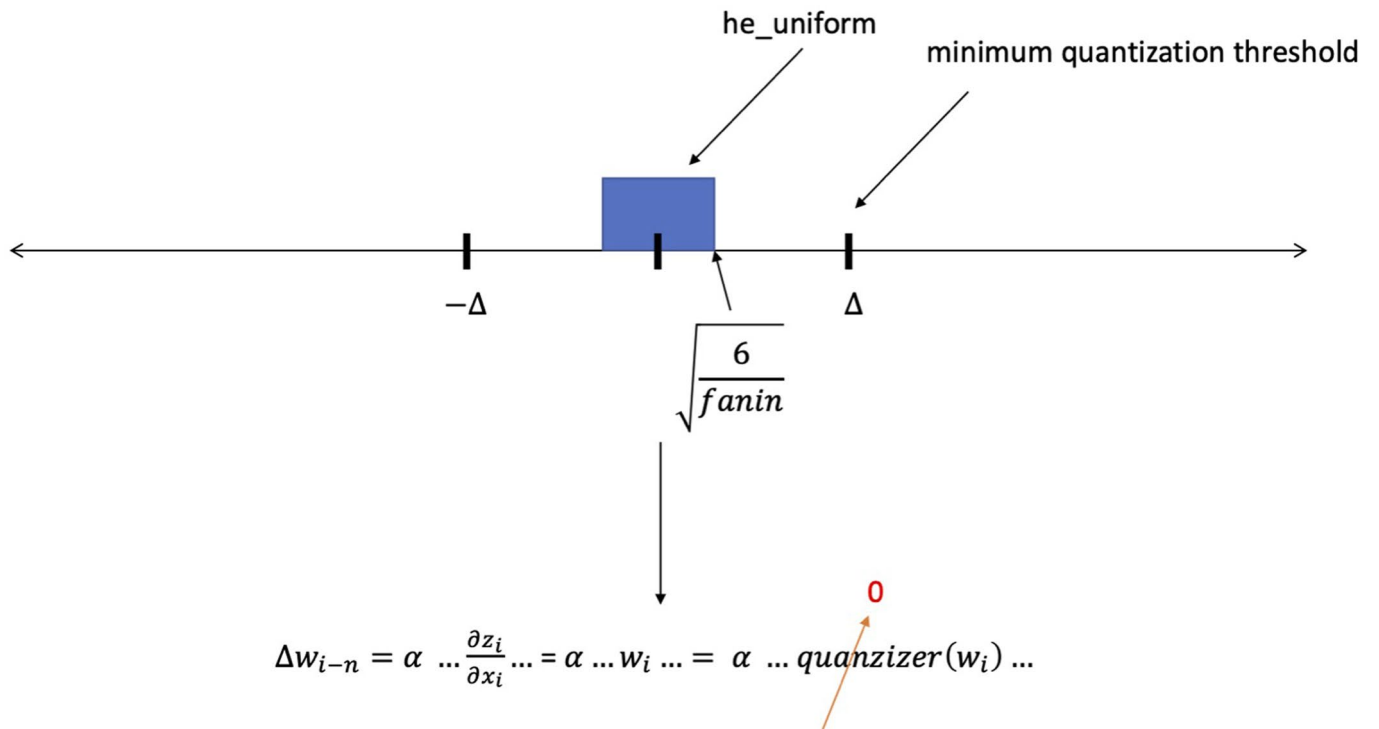
Reprints and permissions information is available at www.nature.com/reprints.

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

© The Author(s), under exclusive licence to Springer Nature Limited 2021



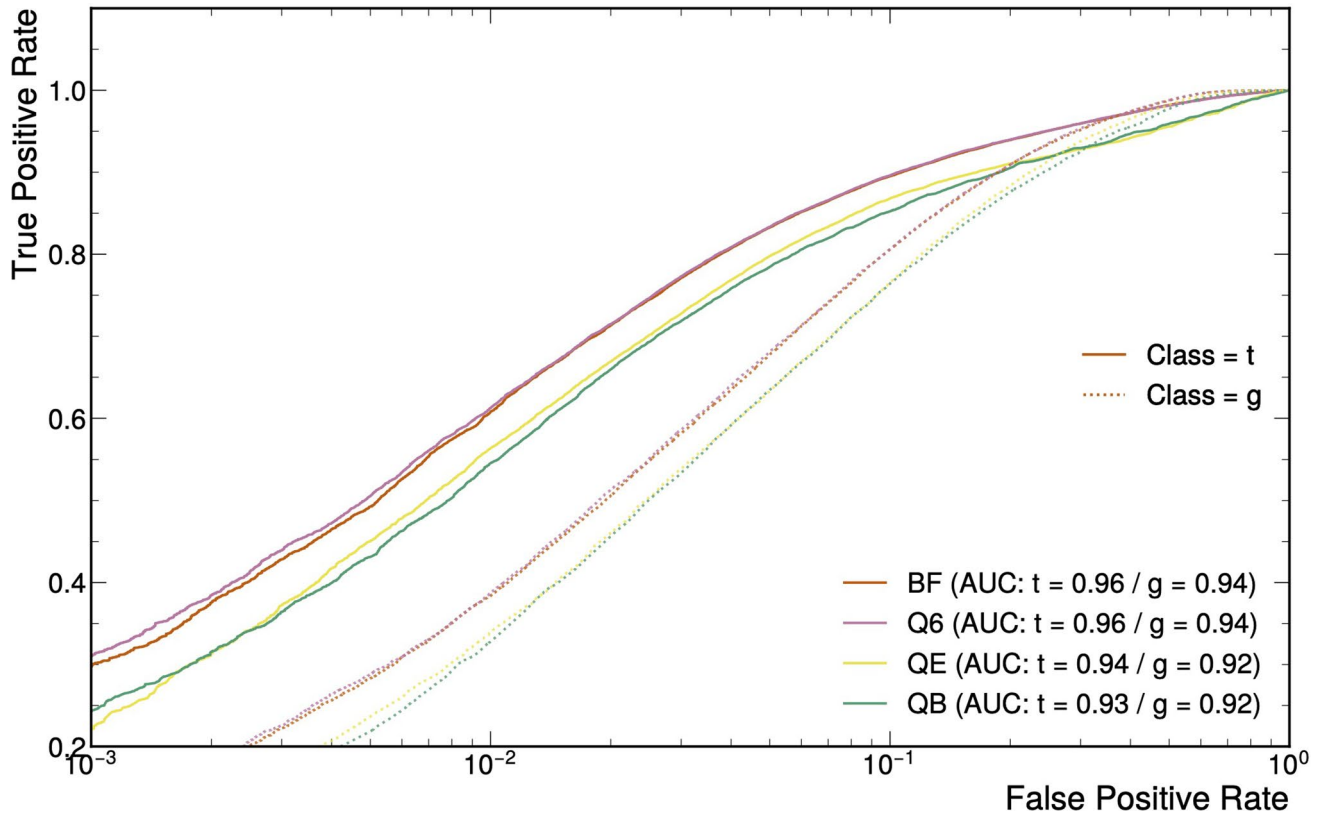
Extended Data Fig. 1 | Model architecture and quantization. Model architecture for the fully-connected NN architecture under study. The numbers in brackets are the precisions used for each layer, quoted as $\langle B, l \rangle$, where B is the precision in bits and l the number of integer bits. When 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.



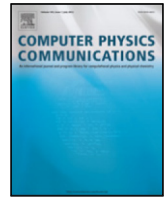
Extended Data Fig. 2 | Variance shift. Variance shift and the effect of initialization in gradient descent.

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), AutoKeras Energy Optimized (QE) and AutoKeras Bits Optimized (QB) models.



C and Fortran OpenMP programs for rotating Bose–Einstein condensates[☆]



Ramavarmaraja Kishor Kumar^a, Vladimir Lončar^b, Paulsamy Muruganandam^{c,d}, Sadhan K. Adhikari^{e,*}, Antun Balaž^b

^a Instituto de Física, Universidade de São Paulo, 05508-090 São Paulo, Brazil

^b Scientific Computing Laboratory, Center for the Study of Complex Systems, Institute of Physics Belgrade, University of Belgrade, Serbia

^c Department of Physics, Bharathidasan University, Palkalaiperur Campus, Tiruchirappalli 620024, Tamilnadu, India

^d Department of Medical Physics, Bharathidasan University, Palkalaiperur Campus, Tiruchirappalli 620024, Tamilnadu, India

^e Instituto de Física Teórica, UNESP – Universidade Estadual Paulista, 01.140-70 São Paulo, São Paulo, Brazil

ARTICLE INFO

Article history:

Received 13 September 2018

Received in revised form 14 February 2019

Accepted 8 March 2019

Available online 18 March 2019

Keywords:

Rotating Bose–Einstein condensate

Gross–Pitaevskii equation

Split-step Crank–Nicolson scheme

C programs

Fortran programs

OpenMP

Partial differential equation

Vortex lattice

ABSTRACT

We present OpenMP versions of C and Fortran programs for solving the Gross–Pitaevskii equation for a rotating trapped Bose–Einstein condensate (BEC) in two (2D) and three (3D) spatial dimensions. The programs can be used to generate vortex lattices and study dynamics of rotating BECs. We use the split-step Crank–Nicolson algorithm for imaginary- and real-time propagation to calculate stationary states and BEC dynamics, respectively. The simulation input parameters for the C programs are provided via input files, while for the Fortran programs they are given at the beginning of each program and therefore their change requires recompilation of the corresponding program. The programs propagate the condensate wave function and calculate several relevant physical quantities, such as the energy, the chemical potential, and the root-mean-square sizes. The imaginary-time propagation starts with an analytic wave function with one vortex at the trap center, modulated by a random phase at different space points. Nevertheless, the converged wave function for a rapidly rotating BEC with a large number of vortices is most efficiently calculated using the pre-calculated converged wave function of a rotating BEC containing a smaller number of vortices as the initial state rather than using an analytic wave function with one vortex as the initial state. These pre-calculated initial states exhibit rapid convergence for fast-rotating condensates to states containing multiple vortices with an appropriate phase structure. This is illustrated here by calculating vortex lattices with up to 61 vortices in 2D and 3D. Outputs of the programs include calculated physical quantities, as well as the wave function and different density profiles (full density, integrated densities in lower dimensions, and density cross-sections). The provided real-time propagation programs can be used to study the dynamics of a rotating BEC using the imaginary-time stationary wave function as the initial state. We also study the efficiency of parallelization of the present OpenMP C and Fortran programs with different compilers.

Program summary

Program title: BEC-GP-ROT-OMP, consisting of: (1) BEC-GP-ROT-OMP-C package, containing programs (i) bec-gp-rot-2d-th and (ii) bec-gp-rot-3d-th; (2) BEC-GP-ROT-OMP-F package, containing programs (i) bec-gp-rot-2d-th and (ii) bec-gp-rot-3d-th.

Program files doi: <http://dx.doi.org/10.17632/cw7tkn22v2.2>

Licensing provisions: Apache License 2.0

Programming language: OpenMP C; OpenMP Fortran. The C programs are tested with the GNU, Intel, PGI, Oracle, and Clang compiler, and the Fortran programs are tested with the GNU, Intel, PGI, and Oracle compiler.

Nature of problem: The present Open Multi-Processing (OpenMP) C and Fortran programs solve the time-dependent nonlinear partial differential Gross–Pitaevskii (GP) equation for a trapped rotating Bose–Einstein condensate in two (2D) and three (3D) spatial dimensions in a fully anisotropic traps.

[☆] This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (<http://www.sciencedirect.com/science/journal/00104655>).

* Corresponding author.

E-mail addresses: kishor.bec@gmail.com (R. Kishor Kumar), vladimir.loncar@ipb.ac.rs (V. Lončar), anand@bdu.ac.in (P. Muruganandam), sk.adhikari@unesp.br (S.K. Adhikari), antun.balaz@ipb.ac.rs (A. Balaž).

Solution method: We employ the split-step Crank–Nicolson algorithm to discretize the time-dependent GP equation in space and time. The discretized equation is then solved by imaginary- or real-time propagation, employing adequately small space and time steps, to yield the solution of stationary and non-stationary problems, respectively.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Previously published Fortran [1] and C [2] programs, and their OpenMP extensions [3,4] are now popular tools for solving the Gross–Pitaevskii (GP) equation and are enjoying widespread use. These programs have been later extended to the more complex scenario of dipolar atoms [5]. The C programs have been adapted to run even faster on modern multi-core computers using general-purpose graphic processing units with Nvidia CUDA and computer clusters using Message Passing Interface (MPI) [6]. In this paper, we present new OpenMP C and Fortran programs to solve the GP equation for a rotating trapped Bose–Einstein condensate (BEC) and to generate a vortex lattice, based on our earlier work [3,4]. This is a problem of general interest for both theoreticians [7,8] and experimentalists [9].

The GP equation for a rotating trapped BEC can be conveniently solved by the imaginary- [10–12] and real-time evolution [13] methods. The solution algorithms rely on transforming the GP equation to the rotating frame, where the rotating BEC with vortices becomes a stationary state [7] and the standard imaginary-time approach can be applied [10]. In the real-time approach [13], a dissipation has to be included in the GP equation to generate the vortices. The imaginary-time approach [10] does not require any dissipation, is simpler to implement and is found to converge faster and lead to accurate results. Here we provide combined imaginary- and real-time programs in two (2D) and three (3D) spatial dimensions without any dissipation [10]. The present imaginary-time program already involves complex variables and is hence combined together with the real-time program. The choice of the type of propagation is made through an input parameter. The imaginary-time approach should be used to solve the GP equation for the rotating BEC and to generate the stationary vortex lattice. A subsequent study of the non-stationary dynamics of the rotating BEC should be done using the real-time propagation. Here we provide C and Fortran programs for the solution of the GP equation for a rotating BEC in a fully anisotropic 3D trap by imaginary- and real-time propagation. We also present C and Fortran programs for the reduced GP equation in 2D, appropriate for a disk-shaped BEC under a tight axial (z -direction) trapping. We use the split-step Crank–Nicolson scheme for solving the GP equation, as in Refs. [1,2].

The imaginary-time algorithm employs a time iteration loop of an initial state until the convergence is reached [1]. The usual initial states are analytic wave functions, generally with one vortex at the center of the trap. However, such an analytic initial function may exhibit slow convergence and often may lead to an inappropriate final vortex lattice structure. We will use an analytic initial function modulated by a random phase at different space points and show that this procedure is essential in addressing the convergence issues, as well as in obtaining the correct vortex lattice structure for a given set of system parameters. Moreover, the GP equation of a rapidly rotating BEC with a very large number of vortices, viz. Figs. 2(c) and (d) with 37 and 61 vortices, faces a convergence difficulty even after random phase modulation. In this latter case, when a pre-calculated converged wave function of the rotating BEC with a smaller number of vortices is used as the initial state, the convergence of the algorithm

is vastly improved, resulting in the reduction of more than 90% in execution time.

In Section 2 we present the GP equation for a rotating BEC in an anisotropic trap. We present the mean-field model and a general scheme for its numerical solution. The reduced 2D GP equation appropriate for a disk-shaped rotating BEC is also presented there. The details about the computer programs, and their input/output files, etc. are given in Section 3. The numerical method and results are given in Section 4, where we illustrate the generation of vortex lattices by employing the imaginary-time propagation in rapidly rotating trapped BECs with different angular frequencies and interaction strengths (nonlinearities). The stability of these vortex lattices is demonstrated in real-time propagation using the corresponding converged solution obtained by the imaginary-time propagation as initial states. The efficiency of parallelization of the present OpenMP programs in multi-core computers using the GNU and Intel compilers is also demonstrated there. Finally, a brief summary is given in Section 5.

2. The Gross–Pitaevskii equation for a rotating condensate

A non-rotating BEC made up of N atoms, each of mass m , can be described by the following mean-field GP equation for a wave function $\phi(\mathbf{r}, t)$ at the space point \mathbf{r} at time t [8]

$$i\hbar \frac{\partial \phi(\mathbf{r}, t)}{\partial t} = \left[-\frac{\hbar^2}{2m} \nabla_{\mathbf{r}}^2 + \frac{1}{2} m \omega^2 (\gamma^2 x^2 + \nu^2 y^2 + \lambda^2 z^2) + \frac{4\pi \hbar^2 a N}{m} |\phi(\mathbf{r}, t)|^2 \right] \phi(\mathbf{r}, t), \quad i = \sqrt{-1}, \quad (1)$$

where $\mathbf{r} \equiv (\boldsymbol{\rho}, z) \equiv (x, y, z)$, a is the atomic s -wave scattering length, and ω is the reference trapping frequency, with γ, ν, λ representing the trap anisotropies along the x, y, z directions, respectively. The normalization condition is $\int d\mathbf{r} |\phi(\mathbf{r}, t)|^2 = 1$. This equation can be derived from the energy functional [8]

$$E[\phi] = \int d\mathbf{r} \left[\frac{\hbar^2}{2m} |\nabla_{\mathbf{r}} \phi|^2 + \frac{1}{2} m \omega^2 (\gamma^2 x^2 + \nu^2 y^2 + \lambda^2 z^2) |\phi|^2 + \frac{2\pi \hbar^2 a N}{m} |\phi|^4 \right]. \quad (2)$$

The formation of a vortex lattice in a rapidly rotating BEC can be conveniently calculated in the rotating frame, where the generated vortex lattice forms a stationary state, which can be obtained by the imaginary-time propagation method. Such a dynamical equation in the rotating frame can be written if we note that the Hamiltonian in the rotating frame is given by $H = H_0 - \Omega L_z$ [14], where H_0 is the laboratory frame Hamiltonian, Ω is the angular frequency of rotation around the z axis, and $L_z = i\hbar(y\partial/\partial x - x\partial/\partial y)$ is the z component of the angular momentum. Consequently, the GP equation in the rotating frame has the explicit form [8,10,11,13,15,16]

$$i\hbar \frac{\partial \phi(\mathbf{r}, t)}{\partial t} = \left[-\frac{\hbar^2}{2m} \nabla_{\mathbf{r}}^2 + \frac{1}{2} m \omega^2 (\gamma^2 x^2 + \nu^2 y^2 + \lambda^2 z^2) + \frac{4\pi \hbar^2 a N}{m} |\phi(\mathbf{r}, t)|^2 - \Omega L_z \right] \phi(\mathbf{r}, t). \quad (3)$$

Using the transformations $\mathbf{r}' = \mathbf{r}/l$, $l = \sqrt{\hbar/(m\omega)}$, $a' = a/l$, $t' = \omega t$, $\phi' = l^{-3/2}\phi$, $\Omega' = \Omega/\omega$, and $L'_z = L_z/\hbar$, we obtain the following convenient dimensionless form of the above equation:

$$i \frac{\partial \phi(\mathbf{r}, t)}{\partial t} = \left[-\frac{1}{2} \nabla_{\mathbf{r}}^2 + \frac{1}{2} (\gamma^2 x^2 + \nu^2 y^2 + \lambda^2 z^2) + g_{3D} |\phi(\mathbf{r}, t)|^2 - \Omega L_z \right] \phi(\mathbf{r}, t), \quad g_{3D} = 4\pi Na, \quad (4)$$

where we have dropped the primes from the transformed dimensionless variables. We note that Eq. (4) can also be derived from the dimensionless energy functional [8]

$$E[\phi] = \int d\mathbf{r} \left[\frac{1}{2} |\nabla_{\mathbf{r}} \phi|^2 + \frac{1}{2} (\gamma^2 x^2 + \nu^2 y^2 + \lambda^2 z^2) |\phi|^2 + \frac{1}{2} g_{3D} |\phi|^4 - \phi^* \Omega L_z \phi \right], \quad (5)$$

obtained using the same transformations and expressing the energy in units of $\hbar\omega$. All derivations and results presented in the following are using these dimensionless variables.

A convenient equation for a quasi-2D disk-shaped BEC under a strong harmonic confinement in the z direction ($\lambda \gg \gamma, \nu$) can be derived using the following ansatz for the wave function [17]:

$$\phi(\mathbf{r}, t) = \psi(\boldsymbol{\rho}, t) \times \frac{1}{(\pi d_z^2)^{1/4}} \exp\left(-\frac{z^2}{2d_z^2}\right), \quad d_z = \sqrt{\frac{1}{\lambda}}, \quad (6)$$

where we assume that because of the strong confinement the dynamics in the z direction will be frozen to a time-independent Gaussian of width d_z , and that the relevant dynamics will evolve only in the x - y plane. If we substitute the ansatz (6) to Eq. (4), we can integrate out the z variable and obtain the corresponding dynamical equation in 2D, valid for a quasi-2D rotating BEC in a disk-shaped trap [1,17]:

$$i \frac{\partial \psi(\boldsymbol{\rho}, t)}{\partial t} = \left[-\frac{1}{2} \nabla_{\boldsymbol{\rho}}^2 + \frac{1}{2} (\gamma^2 x^2 + \nu^2 y^2) + g_{2D} |\psi(\boldsymbol{\rho}, t)|^2 - \Omega L_z \right] \psi(\boldsymbol{\rho}, t), \quad g_{2D} = \frac{4\pi a N \sqrt{\lambda}}{\sqrt{2\pi}}, \quad (7)$$

with the normalization condition $\int d\boldsymbol{\rho} |\psi(\boldsymbol{\rho}, t)|^2 = 1$. The energy functional corresponding to Eq. (7) is

$$E[\psi] = \int d\boldsymbol{\rho} \left[\frac{1}{2} |\nabla_{\boldsymbol{\rho}} \psi|^2 + \frac{1}{2} (\gamma^2 x^2 + \nu^2 y^2) |\psi|^2 + \frac{1}{2} g_{2D} |\psi|^4 - \psi^* \Omega L_z \psi \right]. \quad (8)$$

We use the split-step Crank–Nicolson algorithm for the solution of the GP equations (4) and (7). This approach has been elaborated in detail in Ref. [1]. In the following we describe the necessary modifications for the 2D equation (7). We follow the identical prescription in 3D. Noting that $L_z = i\hbar(y\partial/\partial x - x\partial/\partial y)$, we split the Hamiltonian into three parts:

$$H \equiv H_1 + H_2 + H_3, \quad (9)$$

$$H_1 = \frac{1}{2} (\gamma^2 x^2 + \nu^2 y^2) + g_{2D} |\psi|^2, \quad (10)$$

$$H_2 = -\frac{1}{2} \frac{\partial}{\partial x^2} - i\Omega y \frac{\partial}{\partial x}, \quad (11)$$

$$H_3 = -\frac{1}{2} \frac{\partial}{\partial y^2} + i\Omega x \frac{\partial}{\partial y}. \quad (12)$$

In this approach we perform the time propagation over infinitesimally small time step first over only the part H_1 , and then over the part H_2 , and finally over the part H_3 of the Hamiltonian. Essentially, we split Eq. (7) into

$$i \frac{\partial \psi}{\partial t} = H_1 \psi, \quad i \frac{\partial \psi}{\partial t} = H_2 \psi, \quad i \frac{\partial \psi}{\partial t} = H_3 \psi, \quad (13)$$

and perform the time propagation over these three sub-equations successively and independently of each other, in the given order.

We first solve the first of Eqs. (13) starting from an initial state $\psi(\boldsymbol{\rho}, t_0)$ at $t = t_0$ to obtain the first intermediate solution after an infinitesimal time step Δ . Then this intermediate solution is used as an initial value to solve the second of Eqs. (13), yielding the second intermediate solution at the time $t = t_0 + \Delta$, which is then used to propagate the third of Eqs. (13) over the infinitesimal time Δ to yield the final solution at $t = t_0 + \Delta$, after one full time iteration of Eq. (7). This procedure is repeated n times to get the final solution at time $t_{\text{final}} = t_0 + n\Delta$.

The first equation of (13) with H_1 has the analytic solution [1], which we denote by $\psi^{k+1/3}$ when propagating between the time steps k and $k+1$. Similarly, we denote by $\psi^{k+2/3}$ the wave function after the time propagation with respect to H_2 , and finally by ψ^{k+1} after additional propagation with respect to H_3 , i.e., after one full time iteration. Following Ref. [1] and using notations therein, we discretize the second equation of (13) for H_2 alone as

$$i \frac{\psi_i^{k+2/3} - \psi_i^{k+1/3}}{\Delta} = -\frac{1}{2} \frac{1}{2h_x^2} \left\{ \left(\psi_{i+1}^{k+2/3} - 2\psi_i^{k+2/3} + \psi_{i-1}^{k+2/3} \right) + \left(\psi_{i+1}^{k+1/3} - 2\psi_i^{k+1/3} + \psi_{i-1}^{k+1/3} \right) \right\} - \frac{i\Omega y_j}{4h_x} \left\{ \left(\psi_{i+1}^{k+2/3} - \psi_{i-1}^{k+2/3} \right) + \left(\psi_{i+1}^{k+1/3} - \psi_{i-1}^{k+1/3} \right) \right\}, \quad (14)$$

where $\psi_i^t = \psi(x_i, y_j, t)$ refers to the wave function value at the spatial grid point determined by $x \equiv x_i = -N_x h_x/2 + i h_x$, $y_j = -N_y h_y/2 + j h_y$, $i = 0, 1, 2, \dots, N_x$, and $j = 0, 1, 2, \dots, N_y$. Here h_x, h_y are the space steps along the x and y directions, respectively, and $t = k+1/3$ or $k+2/3$ refers to the time iteration [1], connecting the present ($k+1/3$) to the future ($k+2/3$) in propagation with respect to H_2 .

The above procedure results in a set of tridiagonal equations (14) in $\psi_{i+1}^{k+2/3}$, $\psi_i^{k+2/3}$, and $\psi_{i-1}^{k+2/3}$ at time $t_{k+2/3}$, which are solved using the proper boundary conditions [1]. The tridiagonal equations are written explicitly as $A_i^- \psi_{i-1}^{k+2/3} + A_i^0 \psi_i^{k+2/3} + A_i^+ \psi_{i+1}^{k+2/3} = b_i$, where

$$b_i = \frac{i\Delta}{4h_x^2} \left(\psi_{i+1}^{k+1/3} - 2\psi_i^{k+1/3} + \psi_{i-1}^{k+1/3} \right) - \frac{\Delta\Omega y_j}{4h_x} \left(\psi_{i+1}^{k+1/3} - \psi_{i-1}^{k+1/3} \right) + \psi_i^{k+1/3}, \quad (15)$$

$$A_i^0 = 1 + \frac{i\Delta}{2h_x^2}, \quad A_i^- = -\frac{i\Delta}{4h_x} \left(\frac{1}{h_x} - i\Omega y_j \right),$$

$$A_i^+ = -\frac{i\Delta}{4h_x} \left(\frac{1}{h_x} + i\Omega y_j \right). \quad (16)$$

The discretization for H_3 is performed similarly. The tridiagonal set of equations above is very similar to Eqs. (34) and (35) of Ref. [1], and the real-time propagation routine is programmed and solved in identical fashion after a straightforward modification to include the extra terms due to a non-zero value of Ω in these equations. The imaginary-time propagation routine corresponds to a transformation $t \rightarrow -it$ or $\Delta \rightarrow -i\Delta$ [1] and hence can be obtained by replacing $i\Delta \rightarrow \Delta$ in Eqs. (15) and (16) in the real-time routine, which is performed in our combined real- and imaginary-time programs by the selection parameter `OPTION_RE_IM`.

Instead of evaluating the real energies from Eqs. (5) and (8) in 3D and 2D involving complex algebra over complex wave functions, it is convenient to write a real expression for the energy. To calculate the energy and the chemical potential, we write the two coupled nonlinear equations for the real and imaginary parts

of the wave function ($\psi = \psi_R + i\psi_I$), viz. Eqs. (2.1) of Ref. [15]. The equation satisfied by the real part is

$$i \frac{\partial \psi_R(x, y; t)}{\partial t} = \left[-\frac{1}{2} \nabla^2 + \frac{1}{2} (\gamma^2 x^2 + \nu^2 y^2) + g_{2D} |\psi(x, y; t)|^2 \right] \psi_R(x, y; t) + \Omega \left(y \frac{\partial}{\partial x} - x \frac{\partial}{\partial y} \right) \psi_I(x, y; t). \quad (17)$$

In this equation ψ_R is not normalized to unity. Using Eq. (17), the energy and the chemical potential can be expressed in 2D as

$$\frac{1}{\int dx dy \psi_R^2} \int d\rho \left[-\frac{1}{2} (\nabla_\rho \psi_R)^2 + \frac{1}{2} (\gamma^2 x^2 + \nu^2 y^2) \psi_R^2 + \alpha g_{2D} (\psi_R^2 + \psi_I^2) \psi_R^2 + \Omega \psi_R \left(y \frac{\partial}{\partial x} - x \frac{\partial}{\partial y} \right) \psi_I \right], \quad (18)$$

where the value $\alpha = 1$ corresponds to the chemical potential μ , and the value $\alpha = 1/2$ to the energy E per atom. A similar expression for energy and chemical potential in 3D is

$$\frac{1}{\int d\mathbf{r} \phi_R^2} \int d\mathbf{r} \left[-\frac{1}{2} (\nabla_{\mathbf{r}} \phi_R)^2 + \frac{1}{2} (\gamma^2 x^2 + \nu^2 y^2 + \lambda^2 z^2) \phi_R^2 + \alpha g_{3D} (\phi_R^2 + \phi_I^2) \phi_R^2 + \Omega \phi_R \left(y \frac{\partial}{\partial x} - x \frac{\partial}{\partial y} \right) \phi_I \right]. \quad (19)$$

Eqs.(18) and (19) are equivalent to Eqs. (5) and (8) and involve algebra of real functions. Hence these equations lead to far more accurate numerical results than the previous set of expressions. Specifically, the calculation of the rotational energy and the kinetic energy term involving derivatives and gradients of a complex wave function in Eqs. (5) and (8) can be numerically problematic.

The initial wave function in the imaginary-time programs is taken to be one containing a single vortex at the center, aligned with the z axis. Explicitly, for the 2D and 3D programs, we take, respectively

$$\psi_{\text{initial}}(x, y) = \frac{x + iy}{\sqrt{\pi d_{xy}^2}} \exp \left(-\frac{x^2 + y^2}{2d_{xy}^2} + 2\pi i \mathcal{R}(x, y) \right),$$

$$\phi_{\text{initial}}(x, y, z) = \psi_{\text{initial}}(x, y) \frac{1}{(\pi d_z^2)^{1/4}} \exp \left(-\frac{z^2}{2d_z^2} \right), \quad (20)$$

where d_{xy} and d_z are width parameters in the x - y plane and in the z direction, and $\mathcal{R}(x, y)$ is a random number. In numerical calculation, the random phase ensures that the number of vortices changes by units of one, as parameters, e.g., nonlinearity and angular frequency, are changed. Without the random phase, the number of vortices changes by units of two or multiples of two. In fact, any localized normalizable initial function modulated by a random phase at different space points, e.g., a Gaussian function without any vortices, obtained by setting $(x + iy) = 1$ in Eq. (20), will lead to the same vortex lattice as the initial function (20) with one vortex. Without the random phase these functions usually will lead to different results [16].

3. Details about the programs

All input data (number of atoms, scattering length, harmonic oscillator trap length, trap anisotropy, etc.) are conveniently placed at the beginning of each Fortran program, as before [3]. Hence after changing the input data in a Fortran program a recompilation is required. The C programs use external input files that contain all parameters, and their adjustment does not require a recompilation. The source programs are located in the directory `src` within the corresponding package directory (BEC-GP-ROT-OMP-C for the C programs and BEC-GP-ROT-OMP-F for

the Fortran ones). They can be compiled by the `make` command using the `makefile` in the corresponding package root directory. The examples of produced output files can be found in the directory `output`, although some large density files are omitted, to save space. The programs use an initial state with repeatable random phase. A different random phase can be generated by changing the variable `SEED` in the subroutine `Initialize` for the Fortran programs, or in the corresponding input file for the C programs. The provided Fortran output files are calculated with `SEED = 13` using the one-vortex initial function (20). The change of the variable `SEED` implies a different initial function, thus changing the output files. In the Fortran programs, the random phase is included by the integer parameter `RANDOM`: the value 0 excludes the random phase and 1 includes it. The integer parameter `FUNCTION` permits the selection of a Gaussian or a one-vortex initial function: the value 0 selects a Gaussian function and 1 selects the one-vortex function (20). For the C programs, the input files contain variables providing the same functionality, which is explained there. After running a program and obtaining the results, one can use the file `fig*.gnu` in the directory `output` to visualize the density profiles, relying on a popular software package `gnuplot`. These files are used by invoking the command `gnuplot fig*.gnu` to obtain an eps figure of the generated vortex lattice. Depending on the density file to be plotted, one has to adjust the corresponding line in the `fig*.gnu` file. Currently it is set to use the density file provided as an example and already present in the BEC-GP-ROT-OMP distribution.

The output files are conveniently named such that their contents can be easily identified, following the naming convention introduced in Ref. [3]. For example, a file named `<code>-out.txt`, where `<code>` is a name of the individual program, represents the general output file containing input data, time and space steps, nonlinearity, energy, and chemical potential. A file named `<code>-den2d.txt` is the output file with the reduced (integrated) 2D condensate density. There are output files for reduced (integrated) 1D densities for different programs. Typically, a user first solves the stationary problem using the imaginary-time programs, and then uses the real-time programs to read the pre-calculated stationary wave function and to study the dynamics. To read the pre-calculated wave function the parameter `NSTP` should be set to zero. In this way one can also run the imaginary time program with a pre-calculated wave function. The supplied programs have the pre-defined value `NSTP = 1` and use the analytic wave function (20) as the initial state. In each program the selection for imaginary- or real-time propagation is done by setting the parameter `OPTION_RE_IM` to 1 or 2, respectively. If the imaginary-time propagation is thus selected, the programs run either by using an initial analytic input function (if `NSTP` is not set to zero) or by employing a pre-calculated wave function (if `NSTP` is set to zero). The real-time propagation can successfully work only with a meaningful initial wave function, usually assuming that `NSTP = 0` is set, and that the program will read a pre-calculated wave function by the earlier performed imaginary-time propagation. The reader is advised to consult our previous publication where a complete description of the output files is given [4]. The calculation is essentially done in the NPAS time loop, which are in the Fortran programs conveniently divided into 10 equal intervals (`NPAS/10`). The output files for the reduced 2D densities at the end of each of these intervals are saved as files `<code>*-den-j.txt`, where $j=1, \dots, 10$. If necessary, one can further customize this by changing and recompiling the Fortran programs. In the C programs the selection of output files is done through the input file, when one can set the desired frequency of saving the output densities, as well as the types of density profiles to be saved. A `README.md` file, included in the corresponding root directory for C and Fortran, explains the procedure to compile and run the programs in more detail.

The supplied 2D programs are preset to run the imaginary-time propagation using the space steps $DX=DY=0.05$, numbers of space points $NX=NY=256$, $g_{2D} = 100$, $\Omega = 0.8$, the trap parameters $\gamma = \nu = 1$. The 3D programs use $DX=DY=0.05$, $DZ=0.025$, $NX=NY=256$, $NZ=32$, $\gamma = \nu = 1$, $\lambda = 100$, $g_{2D} = 100$, $g_{3D} = g_{2D}\sqrt{2\pi/\lambda} = 25.0662827$. The large trap parameter λ ensures a disk-shaped BEC, which enables a comparison of the 3D results for the integrated density over the z coordinate with the 2D density profile. This also reduces a transversal instability of the 3D vortex lines. The time steps used are $\Delta = 0.00025$ (imaginary time) and 0.0001 (real time), numbers of time iterations are $NPAS=3,000,000$ and $NSTP=1$ (imaginary time) and $NSTP=0$ (to run real- or imaginary-time propagation with a pre-calculated wave function as an input). To achieve the convergence in some cases (large nonlinearity g_{2D} , g_{3D} and Ω), one may need to increase the values of NX , NY , $NPAS$, and reduce the space and time steps DX , DY , DZ and DT accordingly. Note that the actual spatial grid used contains $(NX+1)\times(NY+1)$ or $(NX+1)\times(NY+1)\times(NZ+1)$ points, since in each dimension the grid index takes the values from 0 to NX , etc. Therefore, the produced output files also contain the data for such grid sizes.

The function (20) always leads to a converged solution after a large number of time iterations in imaginary-time propagation. A Gaussian wave function given as an input in imaginary-time propagation would sometimes face a convergence difficulty and should not be used. Therefore the programs by default use a better initial state, containing one vortex at the center. Once a stationary vortex lattice is obtained for a specific nonlinearity and angular frequency by imaginary-time propagation, the final wave function so obtained should be used as the initial state for the generation of vortex lattices by imaginary-time propagation with larger nonlinearities and/or angular frequencies. For example, to generate closed hexagonal vortex lattices of 19, 37, and 61 vortices in the panels (b), (c) and (d) of Figs. 2 and 5 in the next section, respectively, we have used the previously calculated initial states of 7, 19, and 37 vortices in the corresponding panels (a), (b), and (c), respectively. Such a choice of dynamically generated multi-vortex initial state with a proper phase distribution enhances the convergence of the numerical scheme enormously compared to the propagation starting from a single-vortex initial state. The reduction in the execution time for the calculation done in this fashion could be as much as 99%. The size of the condensate increases as the nonlinearity and/or the angular frequency Ω are increased. To accommodate a larger condensate, the number of space points NX , NY , etc. should be appropriately increased. To read a pre-calculated wave function by setting $NSTP$ to zero, the grid size in the used wave function file should match exactly the number of points used in the current program. The supplied programs assume equal numbers of space step points in both imaginary- and real-time propagation, and in C programs this is configurable through the input files. If the grid sizes in the two calculations are different, the user can customize the programs to accommodate this. For instance, in Fortran programs the `READ` statement in the subroutine `INITIALIZE` should be changed, for instance, from `I=0, NX to I= NX2-NXOLD2, NX2+NXOLD2, 1`, where `NXOLD2` is the `NX2` value of the previous calculation with a smaller number of grid points.

4. Numerical results

To test the programs and to demonstrate their usage, we have generated vortex-lattice structures using the imaginary-time programs and then ran the real-time programs starting from the previously obtained imaginary-time wave functions as inputs. First, we numerically calculate the critical angular frequency Ω_c , for the generation of a single vortex, using the initial

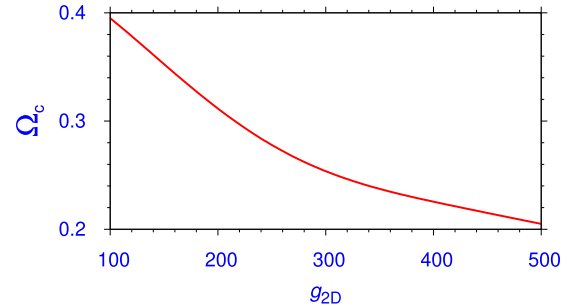


Fig. 1. Critical angular frequency Ω_c for the generation of a single vortex using function (20) with random phase versus nonlinearity g_{2D} for a rotating BEC in 2D. For $\Omega < \Omega_c$ no vortex is generated.

function (20), for a rotating BEC in 2D for different nonlinearities g_{2D} . Without the random phase in the initial wave function this threshold cannot be calculated, as, then, a single vortex continues to exist for $\Omega < \Omega_c$. The result is displayed in Fig. 1. The displayed result is the average over several runs.

We next numerically study the 2D vortex lattice in a rotating BEC using the imaginary-time propagation. The imaginary-time propagation with the supplied 2D program `bec-gp-rot-2d-th` uses the wave function (20) as the initial state and the parameters $g_{2D} = 100$ and $\Omega = 0.8$. The generated vortex lattice with seven vortices arranged in a triangular lattice in the shape of a closed hexagon is exhibited in Fig. 2(a) through the contour density plot. In Fig. 2(b) we illustrate the 2D vortex lattice with 19 vortices arranged in a triangular lattice in the shape of a closed hexagonal form obtained with parameters $g_{2D} = 100$, $\Omega = 0.95$. To illustrate the convergence of the imaginary-time propagation we show in Figs. 3(a)–(d) the 2D density profiles at different times, using the analytic wave function (20) as the initial state and employing the parameters $g_{2D} = 100$, $\Omega = 0.95$, the same as in Fig. 2(b). This scheme shows a slow convergence and the vortex lattice structure practically remains the same from the panel 3(a) for 2×10^5 time steps to the panel 3(c) for 8×10^5 time steps with 19 vortices, before converging to the desired solution in the panel 3(d) after 12×10^5 time steps, containing 19 vortices. The convergence can be highly enhanced if we use the final converged state with a smaller number of vortices as the initial state of a calculation where a larger number of vortices is expected, either because the parameters g_{2D} or Ω or both are larger. In Fig. 3(e)–(h) we demonstrate this and show the vortex lattice evolution of the rotating BEC for the same parameters $g_{2D} = 100$, $\Omega = 0.95$ as in the panels 3(a)–(d), but starting from the initial state with seven vortices, obtained in Fig. 2(a) for $g_{2D} = 100$, $\Omega = 0.8$. In Fig. 3 we see that the convergence in this case is achieved much faster. In practical terms, in panels 3(c) after 20,000 time steps or 3(d) after 30,000 time steps of the imaginary-time propagation the convergence is already reached. The reduction in execution time in the later scheme resulting in Figs. 3(e)–(h) compared to the former resulting in Figs. 3(a)–(d) could be very large, viz. 12×10^5 time iterations and 30,000 time iterations in the two schemes.

In Figs. 2(b)–(d) we illustrate 2D vortex lattices with 19, 37, and 61 vortices, respectively, arranged in triangular lattices in the shape of a closed hexagonal form obtained with parameters $g_{2D} = 100$, $\Omega = 0.95$ in 2(b), $g_{2D} = 500$, $\Omega = 0.92$ in 2(c), and $g_{2D} = 500$, $\Omega = 0.978$ in 2(d). As already suggested above, the vortex lattices of Figs. 2(b), (c), and (d) were obtained using the final wave functions of Fig. 2(a), (b), and (c), respectively, as the initial states, to speed up the convergence. We demonstrate the stability of the obtained vortex lattices using the real-time propagation for 500 time units in Figs. 2(e)–(h).

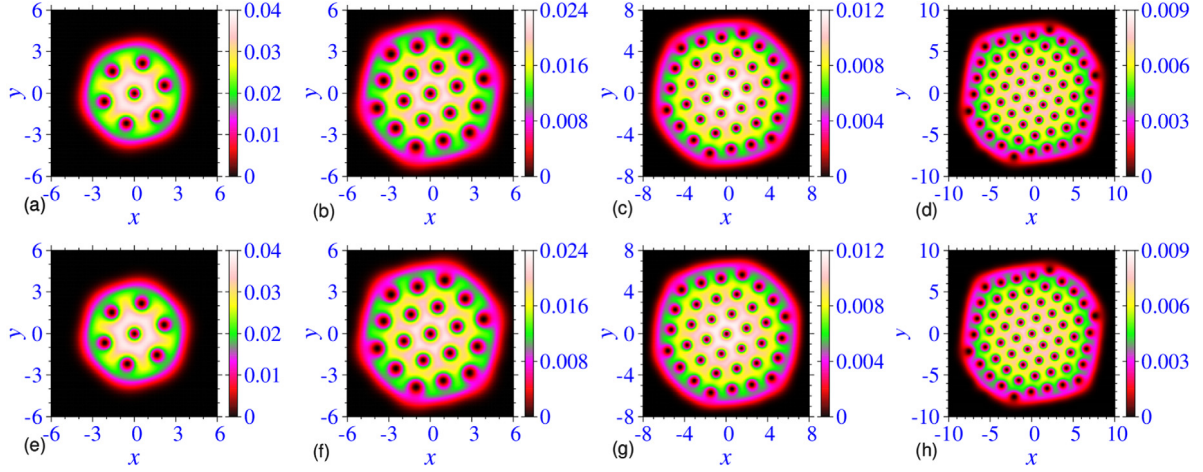


Fig. 2. Contour plots of the density $|\psi(x, y)|^2$ for the generated vortex lattices by the 2D imaginary-time propagation of Eq. (7) for (a) $g_{2D} = 100$, $\Omega = 0.8$, (b) $g_{2D} = 100$, $\Omega = 0.95$, (c) $g_{2D} = 500$, $\Omega = 0.92$, and (d) $g_{2D} = 500$, $\Omega = 0.978$. Panels (e), (f), (g), and (h) display these vortex lattices, respectively, after the additional real-time propagation for 500 units of time using the corresponding imaginary-time wave function as input. The employed trap parameters are $\nu = \gamma = 1$, the space steps are $DX=DY=0.05$, and the time steps are 0.00025 in imaginary time and 0.0001 in real time. The size of the condensate increases as Ω increases from (a) to (b) and from (c) to (d), and as g_{2D} increases from (b) to (c). The space grids used are (a) 257×257 , (b) 321×321 , (c) 401×401 , and (d) 441×441 .

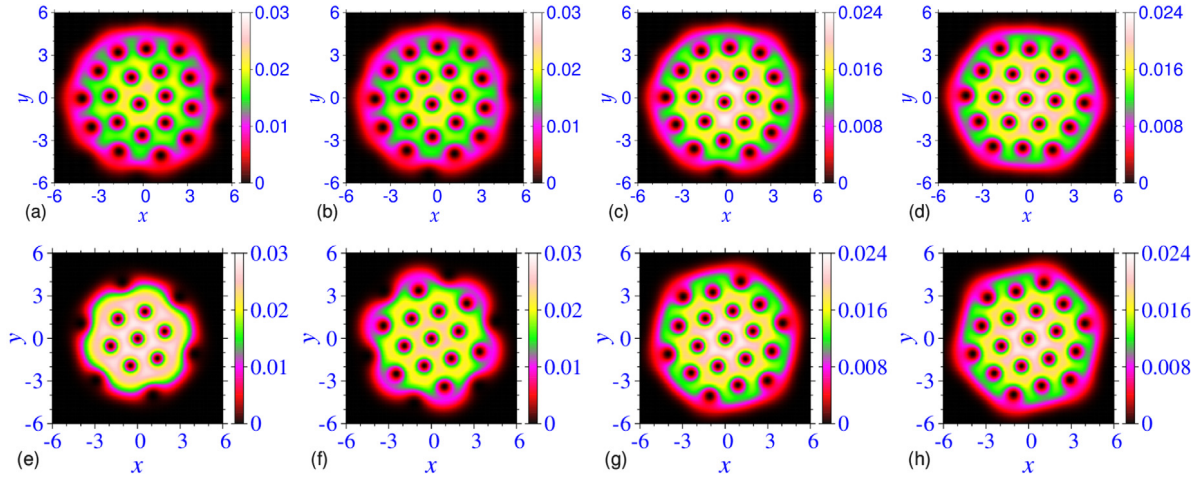


Fig. 3. The convergence of calculation from snapshots at different time steps during the imaginary-time propagation of the 2D equation (7) to generate a vortex lattice for parameters $g_{2D} = 100$, $\Omega = 0.95$. Numerical simulation used the initial state (20), and the panels correspond to (a) 2×10^5 , (b) 4×10^5 , (c) 8×10^5 , and (d) 12×10^5 time steps. For the same parameters, a much faster convergence is obtained in a simulation using as the initial function the converged wave function from Fig. 2(a), obtained for $g_{2D} = 100$, $\Omega = 0.8$. The panels correspond to (e) 5000, (f) 10,000, (g) 20,000, and (h) 30,000 time steps. The employed time step is 0.00025, the space steps $DX=DY=0.05$, and the grid size used is 321×321 in all panels.

In Figs. 4 we show the increase of the number of vortices with the increase of the angular frequency Ω for a fixed $g_{2D} = 100$ as obtained with the one-vortex initial function and the Gaussian initial function, both modulated by a random phase at different space points. The number of vortices and their orientation in space are identical with both functions, although the energy varies a little from one initial function to another. If the random-phase modulation is removed, these two functions lead to different number of vortices, whereas with the random-phase modulation these functions usually lead to the same number of vortices, viz. Fig. 4.

In Figs. 5 we present the z -integrated reduced 2D density $\int dz |\phi(x, y, z)|^2$, calculated from the 3D imaginary-time runs, with 7, 19, 37, and 61 vortices for the parameters: (a) $g_{2D} = 100$, $\Omega = 0.8$, (b) $g_{2D} = 100$, $\Omega = 0.95$, (c) $g_{2D} = 500$, $\Omega = 0.92$, and (d) $g_{2D} = 500$, $\Omega = 0.978$. The vortex lattices of Figs. 5(b)–(d) were generated, as before, by the imaginary-time propagation of Eq. (4) until the convergence using the final wave function of Figs. 5(a)–(c) as the initial states, respectively. Figs. 5(e)–(h)

illustrate the same reduced densities obtained from the 3D real-time runs after 100 time units using as inputs the final converged imaginary-time wave function of Figs. 5(a)–(d), respectively. The agreement between the imaginary- and the real-time densities demonstrates the stability of the vortex-lattice structures and the employed algorithm. The 2D densities of Figs. 5 are quite similar to those in Fig. 2 with the same 2D nonlinearity and the same angular frequency. To the best of our knowledge, such a clean 61-vortex lattice, viz. Fig. 5(d), is obtained for the first time here in the simulation of the 3D GP equation (4).

In Table 1 we show the energy and the chemical potential of the BECs of Figs. 2(a) and 5(a) calculated starting from the analytic function (20) as the initial state. We also give the energy and the chemical potential of the BECs of Figs. 2(b)–(d) and 5(b)–(d), calculated with the converged wave functions of Figs. 2(a)–(c) and 5(a)–(c), respectively, as the initial states. The 2D energy values $E = 3.190$ and 2.209 shown in Table 1 for $g_{2D} = 100$ and $\Omega = 0.8$ and 0.95 , respectively, are in good agreement with the energies $E = 3.1904$ and 2.2106 reported in Fig. 6 of Ref. [15]. The authors

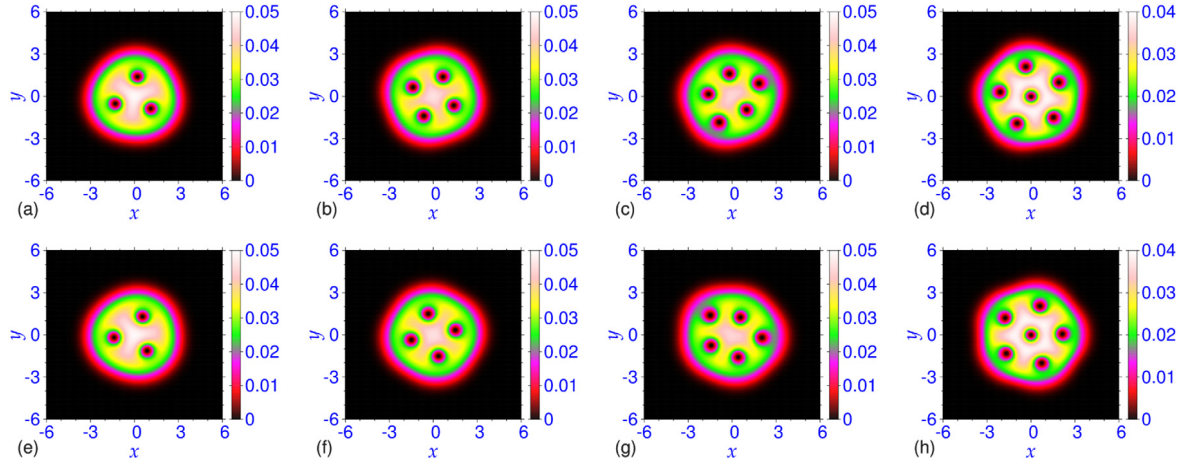


Fig. 4. Contour plots of the density $|\psi(x, y)|^2$ for the generated vortex lattices by the 2D imaginary-time propagation of Eq. (7) for $g_{2D} = 100$, and (a) $\Omega = 0.65$, (b) $\Omega = 0.74$, (c) $\Omega = 0.76$, and (d) $\Omega = 0.78$ obtained with the one-vortex initial state (20). Panels (e), (f), (g), and (h) display these vortex lattices, respectively, obtained with the Gaussian initial state. The employed trap parameters are $\nu = \gamma = 1$, the space steps are $DX=DY=0.05$, the time step is 0.00025 and the space grid is 257×257 .

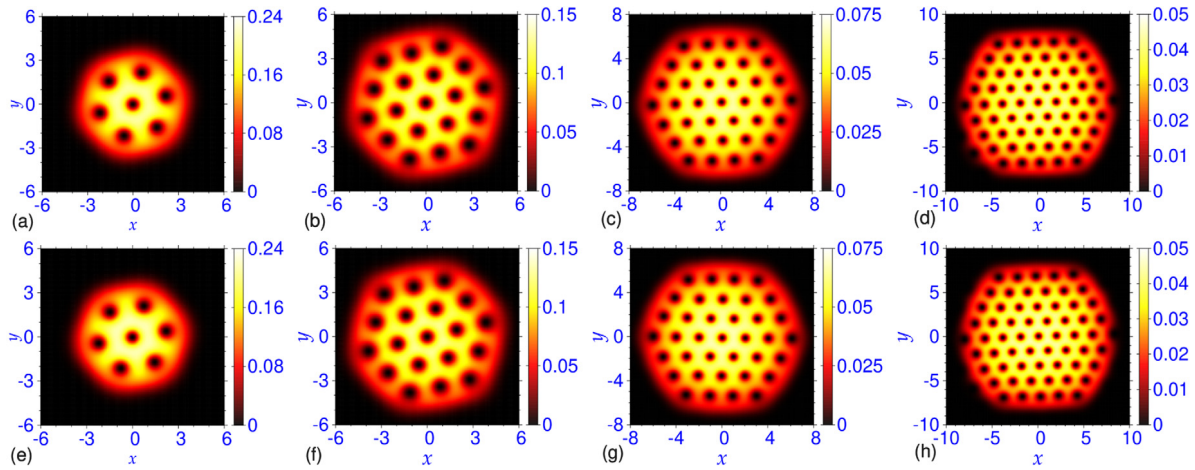


Fig. 5. Contour plots of the density profiles for the generated vortex lattices by the 3D imaginary-time propagation of Eq. (4) for (a) $g_{2D} = 100$, $g_{3D} \equiv g_{2D}\sqrt{2\pi/\lambda} = 25.0662827$, $\Omega = 0.8$, (b) $g_{2D} = 100$, $g_{3D} = 25.0662827$, $\Omega = 0.95$, (c) $g_{2D} = 500$, $g_{3D} = 125.33141$, $\Omega = 0.92$, and (d) $g_{2D} = 500$, $g_{3D} = 125.33141$, $\Omega = 0.978$. Panels (e), (f), (g), and (h) display these vortex lattices, respectively, after the additional real-time propagation for 100 units of time using the corresponding imaginary-time wave function as input. The employed trap parameters are $\nu = \gamma = 1$, $\lambda = 100$, the space steps are $DX=DY=0.05$, $DZ=0.025$, and the time steps are 0.00025 in imaginary time and 0.0001 in real time. The space grids used are (a) $257 \times 257 \times 33$, (b) $321 \times 321 \times 33$, (c) $401 \times 401 \times 33$, and (d) $451 \times 451 \times 33$.

Table 1

Energy E and chemical potential μ for the rotating BECs in 2D and 3D shown in Figs. 2 and 5, respectively. For parameters $g_{2D} = 100$, $\Omega = 0.8$ the calculation is performed with the initial state (20). For the BECs from panels (b), (c), and (d) in Figs. 2 and 5 the calculation is performed with the converged wave functions of the corresponding panels (a), (b), and (c) as the initial states.

	$g_{2D} = 100$ $\Omega = 0.8$	$g_{2D} = 100$ $\Omega = 0.95$	$g_{2D} = 500$ $\Omega = 0.92$	$g_{2D} = 500$ $\Omega = 0.978$
μ (2D)	4.351	2.871	6.257	4.198
E (2D)	3.190	2.209	4.424	2.951
μ (3D)	54.32	52.85	56.20	54.17
E (3D)	53.17	52.19	54.40	52.94

of Ref. [16] also calculated the 2D energy and the chemical potential and we verified using the same parameters that the present energies and chemical potentials are in qualitative agreement with their calculations.

We have tested the performance and scalability of our programs on a modern 8-core Intel Xeon E5-2670 CPUs with 32 GB of RAM. The nodes used for testing contain two CPUs, which allowed us to study the performance of our programs on up to 16 CPU

cores. The testing was done at the PARADOX supercomputing facility of the Institute of Physics Belgrade.

For both the C and the Fortran programs the execution time in the beginning reduces rapidly as the number of threads (used CPU cores) is increased. But eventually the gain in the execution time saturates. This is illustrated in Fig. 6, where we plot the execution time versus the number of threads for both the C and the Fortran programs using GNU 7.2.0 and Intel 17.0.4 compilers, respectively. For both compilers, for a large number of threads the C programs are faster. For a small number of threads (four or less), the Fortran programs compiled with the GNU compiler are faster, whereas for the Intel compiler all programs have similar performance, with the C programs being slightly faster.

For a quantitative estimate of the performance we now study the speedup and the efficiency of the programs using different compilers for a calculation: GNU GCC 7.2.0, Intel C 17.0.4, GNU Fortran 7.2.0, and Intel Fortran 17.0.4. The speedup is defined as the ratio $T(1)/T(n)$ where $T(n)$ is the execution time of a run with n threads. The efficiency is the ratio $T(1)/[nT(n)]$, indicating how many of the threads the computer is effectively utilizing. These are illustrated in Fig. 7 for GNU GCC, Intel GCC, GNU Fortran,

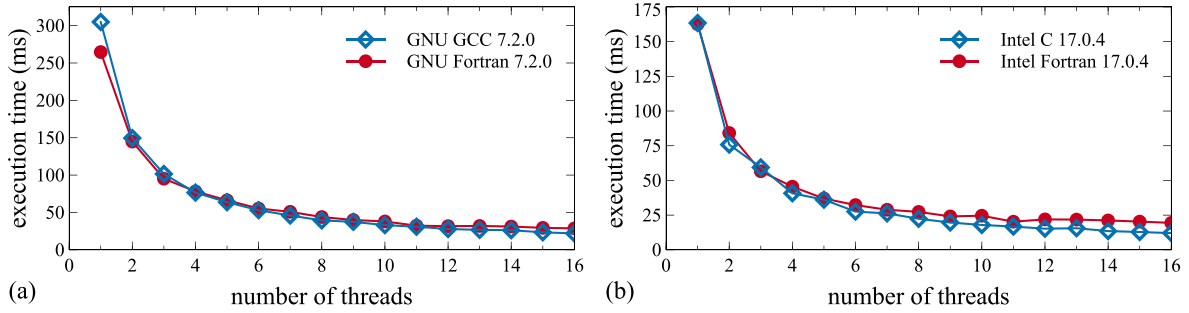


Fig. 6. Wall-clock execution times of BEC-GP-ROT-OMP programs for imaginary-time propagation in 3D (bec-gp-rot-3d-th), compiled with (a) GNU compiler and (b) Intel compiler, as functions of the number of OpenMP threads. The execution times given here are for one iteration, calculated as averages using runs with 1000 iterations (in milliseconds, excluding initialization and input/output operations, as reported by each program) and with the grid size $257 \times 257 \times 33$.

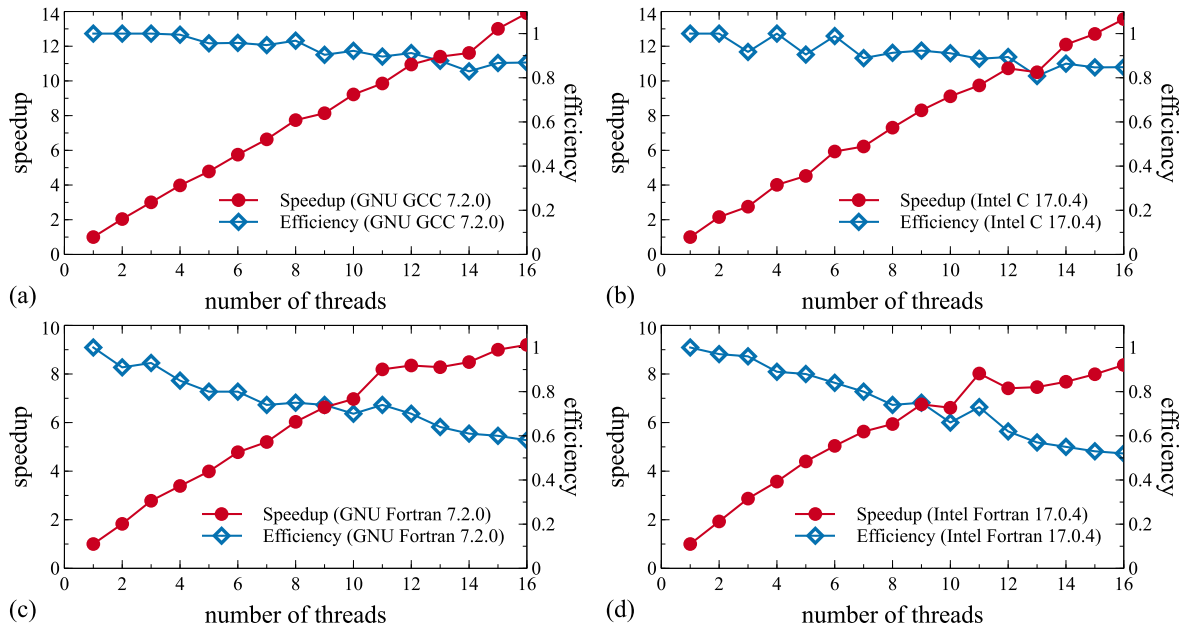


Fig. 7. Speedup in the execution time and scaling efficiency of BEC-GP-ROT-OMP programs for imaginary-time propagation in 3D (bec-gp-rot-3d-th), compared to single-threaded runs for (a) C program compiled with the GNU compiler, (b) C program compiled with the Intel compiler, (c) Fortran program compiled with the GNU compiler, and (d) Fortran program compiled with the Intel compiler. The speedup is calculated as a ratio of the wall-clock execution times $T(1)/T(n)$ for a single-threaded run and a run with n threads, and the scaling efficiency is calculated as a fraction of the obtained speedup compared to a theoretical maximum n . Grid size used for testing is $257 \times 257 \times 33$.

and Intel Fortran compilers, respectively. For a large number of threads, the C programs, viz. plots 7(a)–(b), are more scalable, with large speedup and efficiency compared to the Fortran programs, viz. plots 7(c)–(d). The programs in both programming languages are quite efficient and optimized, but a user should use the specific program and compiler with which he/she has more experience and feels more comfortable.

5. Summary and conclusions

We have presented the efficient OpenMP C and Fortran programs for solving the GP equation for a rotating BEC and use them to calculate the vortex lattices of a rotating BEC by solving the GP equation in the rotating frame. We provide two sets of programs – one for a 3D BEC and the other for a quasi-2D BEC. Each of these programs is capable of executing both the imaginary- and the real-time propagation. We use the split-step Crank–Nicolson algorithm and the programs are based on our earlier OpenMP C and Fortran programs of Ref. [4] for a non-rotating BEC. We solve the GP equation by the imaginary-time propagation with the analytic wave function (20) as the initial state to generate a vortex lattice with a small number of vortices. To solve the GP equation

with a large number of vortices it is much more efficient to use a converged wave function with a smaller number of vortices as the initial state, rather than the analytic function (20). However, the solution can be obtained with any initial state. Nevertheless, the convergence with one initial state could be much faster than with another initial state. For example, to solve the 2D GP equation (7) with parameters $g_{2D} = 100$ and $\Omega = 0.95$ by the imaginary-time propagation using the initial function (20) and obtain the vortex lattice with 19 vortices, one needs 12×10^5 time iterations, viz. Fig. 3. For the same calculation using the pre-calculated vortex lattice with 7 vortices it is sufficient to use only 30,000 time iterations. Although both the C and the Fortran programs produce equivalent results, on a multi-core computer with more than 8 cores, the C programs compiled with both the GCC and the Intel compiler yield a more efficient and faster performance.

The localized normalizable initial function (20) has a random phase at each grid point (x, y) which is necessary to obtain a converged vortex lattice with any number of vortices – even or odd – independent of the initial function. If the random phase is removed from the initial function, the one-vortex initial function (20) leads to a vortex lattice with an odd number of vortices and a Gaussian initial function leads to a vortex lattice with an even

number of vortices. Any localized normalizable initial function with random phase as in Eq. (20), e.g., a Gaussian function or a function with one vortex, usually leads to the same vortex lattice. Without the random phase these functions lead to different vortex lattices.

Acknowledgments

V.L. and A.B. acknowledge support by the Ministry of Education, Science, and Technological Development of the Republic of Serbia under projects ON171017 and III43007. P.M. acknowledges support by the Council of Scientific and Industrial Research (CSIR), Govt. of India under project No. 03(1422)/18/EMR-II. R.K.K. acknowledges support by the FAPESP (Brazil) grant 2014/01668-8. S.K.A. acknowledges support by the CNPq (Brazil) grant 303280/2014-0, by the FAPESP (Brazil) grant 2012/00451-0, and by the ICTP-SAIFR-FAPESP (Brazil) grant 2016/01343-7. Numerical tests were partially carried out on the PARADOX supercomputing facility at the Scientific Computing Laboratory of the Institute of Physics Belgrade.

References

- [1] P. Muruganandam, S.K. Adhikari, *Comput. Phys. Comm.* 180 (2009) 1888.
- [2] D. Vudragović, I. Vidanović, A. Balaž, P. Muruganandam, S.K. Adhikari, *Comput. Phys. Comm.* 183 (2012) 2021.
- [3] L.E. Young-S., D. Vudragović, P. Muruganandam, S.K. Adhikari, A. Balaž, *Comput. Phys. Comm.* 204 (2016) 209.
- [4] L.E. Young-S., P. Muruganandam, S.K. Adhikari, V. Lončar, D. Vudragović, Antun Balaž, *Comput. Phys. Comm.* 220 (2017) 503.
- [5] R. Kishor Kumar, L.E. Young-S., D. Vudragović, A. Balaž, P. Muruganandam, S.K. Adhikari, *Comput. Phys. Comm.* 195 (2015) 117.
- [6] V. Lončar, A. Balaž, A. Bogojević, S. Škrbić, P. Muruganandam, S.K. Adhikari, *Comput. Phys. Comm.* 200 (2016) 406; V. Lončar, L.E. Young-S., S. Škrbić, P. Muruganandam, S.K. Adhikari, A. Balaž, *Comput. Phys. Comm.* 209 (2016) 190; B. Satarić, V. Slavnić, A. Belić, A. Balaž, P. Muruganandam, S.K. Adhikari, *Comput. Phys. Comm.* 200 (2016) 411.
- [7] A.L. Fetter, *Rev. Modern Phys.* 81 (2009) 647.
- [8] A.L. Fetter, *J. Low Temp. Phys.* 161 (2010) 445.
- [9] J.R. Abo-Shaeer, C. Raman, J.M. Vogels, W. Ketterle, *Science* 292 (2001) 476; V. Schweikhard, I. Coddington, P. Engels, S. Tung, E.A. Cornell, *Phys. Rev. Lett.* 93 (2004) 210403; J.R. Abo-Shaeer, C. Raman, W. Ketterle, *Phys. Rev. Lett.* 88 (2002) 070409.
- [10] D.L. Feder, C.W. Clark, B.I. Schneider, *Phys. Rev. Lett.* 82 (1999) 4956; D.L. Feder, C.W. Clark, B.I. Schneider, *Phys. Rev. A* 61 (1999) 011601(R).
- [11] R. Zeng, Y.-Z. Zhang, *Comput. Phys. Comm.* 180 (2009) 854; A. Aftalion, Q. Du, *Phys. Rev. A* 64 (2001) 063603; A. Aftalion, I. Danaila, *Phys. Rev. A* 68 (2003) 023603.
- [12] I. Danaila, F. Hecht, *J. Comput. Phys.* 229 (2010) 6946; G. Vergez, I. Danaila, S. Auliac, F. Hecht, *Comput. Phys. Comm.* 209 (2016) 144; T. Mizushima, Y. Kawaguchi, K. Machida, T. Ohmi, T. Isoshima, M.M. Salomaa, *Phys. Rev. Lett.* 92 (2004) 060407.
- [13] A.A. Penckwitt, R.J. Ballagh, C.W. Gardiner, *Phys. Rev. Lett.* 89 (2002) 260402; M. Tsubota, K. Kasamatsu, M. Ueda, *Phys. Rev. A* 65 (2002) 023603; C. Lobo, A. Sinatra, Y. Castin, *Phys. Rev. Lett.* 92 (2004) 020403.
- [14] L.D. Landau, E.M. Lifshitz, *Mechanics*, Pergamon Press, Oxford, 1960, Section 39.
- [15] B.-W. Jeng, Y.-S. Wang, C.-S. Chien, *Comput. Phys. Comm.* 184 (2013) 493.
- [16] W. Bao, H. Wang, P.A. Markowich, *Commun. Math. Sci.* 3 (2005) 57.
- [17] L. Salasnich, A. Parola, L. Reatto, *Phys. Rev. A* 65 (2002) 043614.

PAPER • OPEN ACCESS

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

To cite this article: Jennifer Ngadiuba *et al* 2021 *Mach. Learn.: Sci. Technol.* **2** 015001

View the [article online](#) for updates and enhancements.

You may also like

- [Lightweight jet reconstruction and identification as an object detection task](#)
Adrian Alan Pol, Thea Aarrestad, Ekaterina Govorkova *et al.*
- [A meshfree moving least squares-Tchebychev shape function approach for free vibration analysis of laminated composite arbitrary quadrilateral plates with hole](#)
Songhun Kwak, Kwanghun Kim, Kwangil An *et al.*
- [Fast convolutional neural networks on FPGAs with hls4ml](#)
Thea Aarrestad, Vladimir Loncar, Nicolò Ghielmetti *et al.*

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

**OPEN ACCESS****RECEIVED**

25 March 2020

REVISED

19 June 2020

ACCEPTED FOR PUBLICATION

26 June 2020

PUBLISHED

1 December 2020

Original Content from this work may be used under the terms of the [Creative Commons Attribution 4.0 licence](#). Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.



Jennifer Ngadiuba¹ , Vladimir Loncar¹, Maurizio Pierini¹, Sioni Summers¹, Giuseppe Di Guglielmo², Javier Duarte³ , Philip Harris⁴, Dylan Rankin⁴, Sergo Jindariani⁵, Mia Liu⁵, Kevin Pedro⁵, Nhan Tran⁵, Edward Kreinar⁶, Sheila Sagear⁷, Zhenbin Wu⁸ and Duc Hoang⁹

¹ European Organization for Nuclear Research (CERN), CH-1211 Geneva 23, Switzerland

² Columbia University, New York, NY 10027, United States of America

³ University of California San Diego, La Jolla, CA 92093, United States of America

⁴ Massachusetts Institute of Technology, Cambridge, MA 02139, United States of America

⁵ Fermi National Accelerator Laboratory, Batavia, IL 60510, United States of America

⁶ HawkEye360, Herndon, VA 20170, United States of America

⁷ Boston University, Boston, MA 02215, United States of America

⁸ University of Illinois at Chicago, Chicago, IL 60607, United States of America

⁹ Rhodes College, Memphis, TN 38112, United States of America

E-mail: jennifer.ngadiuba@cern.ch

Keywords: high-energy physics, fast machine learning inference, FPGAs, quantized neural networks

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 field-programmable gate arrays (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.

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 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 digital 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 $\mathcal{O}(1) \mu\text{s}$ to decide whether to keep or discard a given event. This real-time decision-taking system, referred to as the level-1 (L1) trigger, consists of a set of digital circuits implementing physics-motivated rule-based selection algorithms. Currently, these algorithms are deployed on FPGAs, mounted on custom electronics 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 boosted decision trees (BDTs) 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). Because the complexity of LHC collision events is going to increase after the upcoming high-luminosity upgrade, we expect this approach to become more prevalent.

In order to facilitate the deployment of DNNs in the L1 trigger systems 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 reference [4].

The `hls4ml` design is characterized by two aspects: (i) a reliance on high-level synthesis (HLS) backends, in order to fully automate the workflow from a trained model to FPGA firmware; (ii) a target of 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.

In addition, the `hls4ml` library supports the deployment of BDTs on FPGAs [9]. A BDT trained on high-level features can often reach similar performances than small fully-connected neural networks. On the other hand, neural networks offer the possibility to directly process the raw data, saving time and resources that would be otherwise spent to compute the input features. Depending on the use case, a developer would decide which workflow better fits her needs.

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 using a reduced numerical representation). Comprehensive reviews of these techniques can be found in reference [10, 11]. In a previous publication [2], we showed that pruning [12, 13] and quantization [12, 14] 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 [15–17], following closely the studies presented in references [15, 18, 19]. Network parameters in binary (ternary) networks assume values $+1$ or -1 ($+1, 0$, or -1). They can be represented with one bit (two bits), resulting in a much smaller resource consumption.

In this study, we consider two benchmark problems: MNIST digit classification, which allows a direct comparison with previous literature [18]; the jet tagging problem used as benchmark in our previous study [2] as well as by other groups [4]. The jet tagging problem is particularly relevant for applications at the LHC. Traditional algorithms for jet tagging are too complex to run within L1 latency constraint. Developing resource-friendly ultrafast solutions for jet tagging would drastically increase the L1 selection quality for all-jet collision events. One should keep in mind that our LHC jet data set represents a simplification of more complex realistic conditions. It does not take into account the time and resources one would spend to compute the input features. In the future, the extension of the `hls4ml` library to more complex architectures will allow to consider more realistic use cases, with raw data being directly processed by compressed models.

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. The summary and outlook are 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 [20] and the LHC jet tagging task discussed in reference [2].

The MNIST data set consists of images of hand-written digits. 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

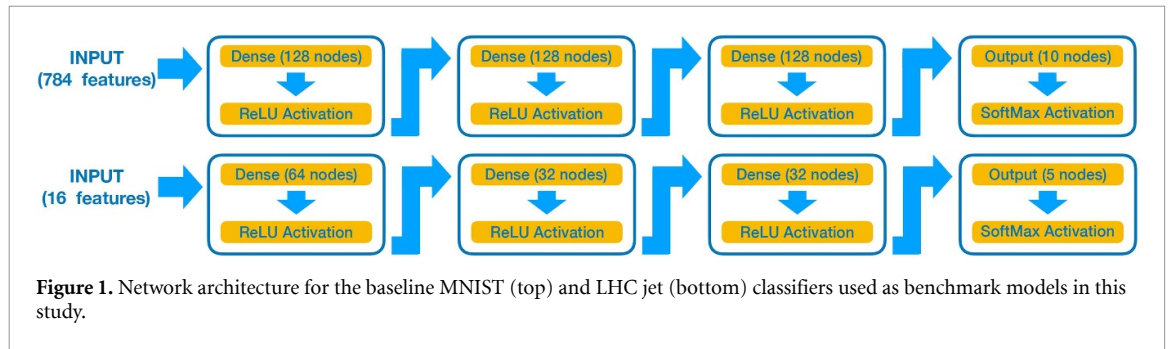


Figure 1. Network architecture for the baseline MNIST (top) and LHC jet (bottom) classifiers used as benchmark models in this study.

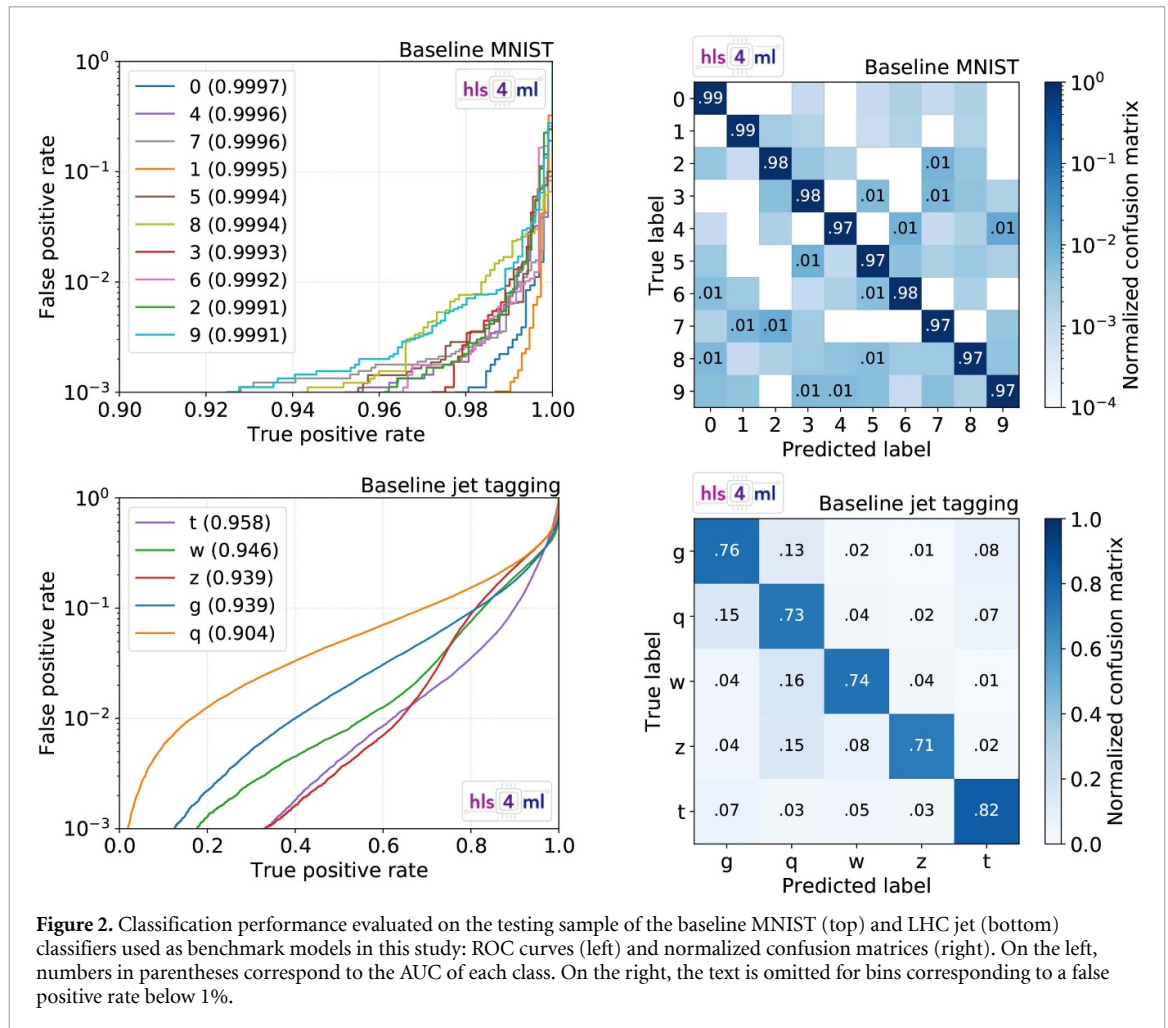


Figure 2. Classification performance evaluated on the testing sample of the baseline MNIST (top) and LHC jet (bottom) classifiers used as benchmark models in this study: ROC curves (left) and normalized confusion matrices (right). On the left, numbers in parentheses correspond to the AUC of each class.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

array is passed as input to a multilayer perceptron (MLP) [21] with an input (output) layer of 784 (10) nodes and three hidden layers with 128 nodes each. Rectified linear unit (ReLU) activation functions [22] are used for the hidden layer nodes, while a softmax activation function is used for the output layer. The MNIST data set comes divided into training-and-validation samples (with 60,000 images) and a testing samples (with 10,000 images). We use 75% of the training-and-validation data set for training, and the remaining 25% for validation.

The other benchmark task consists of classifying jets from a set of 16 physics-motivated high-level features, as described in references [2, 23]. The input data set consists of simulated jets with an energy of order 1 TeV, originating from light quarks (q), gluons (g), W bosons, Z bosons, or top quarks (t) produced in proton-proton collisions at a center-of-mass energy of 13 TeV. Jets are clustered using the anti- k_T algorithm [24], with distance parameter $R = 0.8$. For each jet, the 16 high level features are computed and given as input to a multiclass MLP classifier. The data set is available in the Zenodo repository [25]. More details on the data set can be found in references [2, 23, 26]. The data set consists of approximately 1 million examples and is split in three parts: 20% for test, 60% for training, and 20% for validation. The network

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	<i>g</i>	0.939	89
1	0.9995	99.8			
2	0.9991	99.6	<i>q</i>	0.904	85
3	0.9993	99.6			
4	0.9996	99.6	<i>W</i>	0.946	91
5	0.9994	99.6			
6	0.9992	99.6	<i>Z</i>	0.939	92
7	0.9996	99.6			
8	0.9994	99.4	<i>t</i>	0.958	93
9	0.9991	99.5			

receives as input the 16 high-level features 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 that a given jet belongs to one of the five jet classes.

The architectures of the baseline MNIST and LHC jet classifiers are illustrated in figure 1. Both are implemented and trained with Keras in floating point precision (FPP). Their performance is shown in figure 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 model accuracy as the fraction of correctly labeled examples, also referred to as true positives (TP)

$$\frac{\sum_{i=1}^C TP_i}{N}, \quad (1)$$

where the sum runs over the number of classes C and N is the total number of examples. The accuracy per class is calculated taking into account also the true negatives (TN), i.e. the examples not belonging to that class and that have been predicted in one of the other classes

$$\frac{\sum_{i=1}^C TP_i + TN_i}{N}. \quad (2)$$

In practice, the computation of the model or per-class accuracy is done applying an *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.

These baseline architectures were chosen in order to provide a reasonable performance while keeping the resource utilization within a manageable level. The state-of-the-art performance on MNIST reaches higher accuracy than the models considered here. However, these models are extremely lightweight in terms of their small number of parameters, and low precision. They are therefore optimized for their small footprint of resources and latency in the FPGA inference. Similarly, any jet classifier algorithm with accuracy $\sim 60 - 70\%$, like the one we consider, would be of great benefit for LHC experiments: since the majority of jets produced at the LHC comes from quarks and gluons, our baseline model would allow one to select $> 80\%$ of *W*, *Z*, and *t* jets while reducing the required bandwidth by a factor ~ 10 , saving resources that could be used to extend the physics program of the experiment in other directions.

We consider these models as examples, which are not intended to represent the best reachable performance for a given use case. No architecture optimization was attempted, since the focus of this study is on their implementation on hardware and relative performance drop rather than on absolute performance.

3. Implementing binary and ternary networks in hls4ml

Binary and ternary networks are extreme examples of quantized neural networks [2]. A network is quantized when its parameters (operations) are represented (performed) with reduced numerical precision. This precision could be the same across the full network or specific to each component (e.g. for different layers).

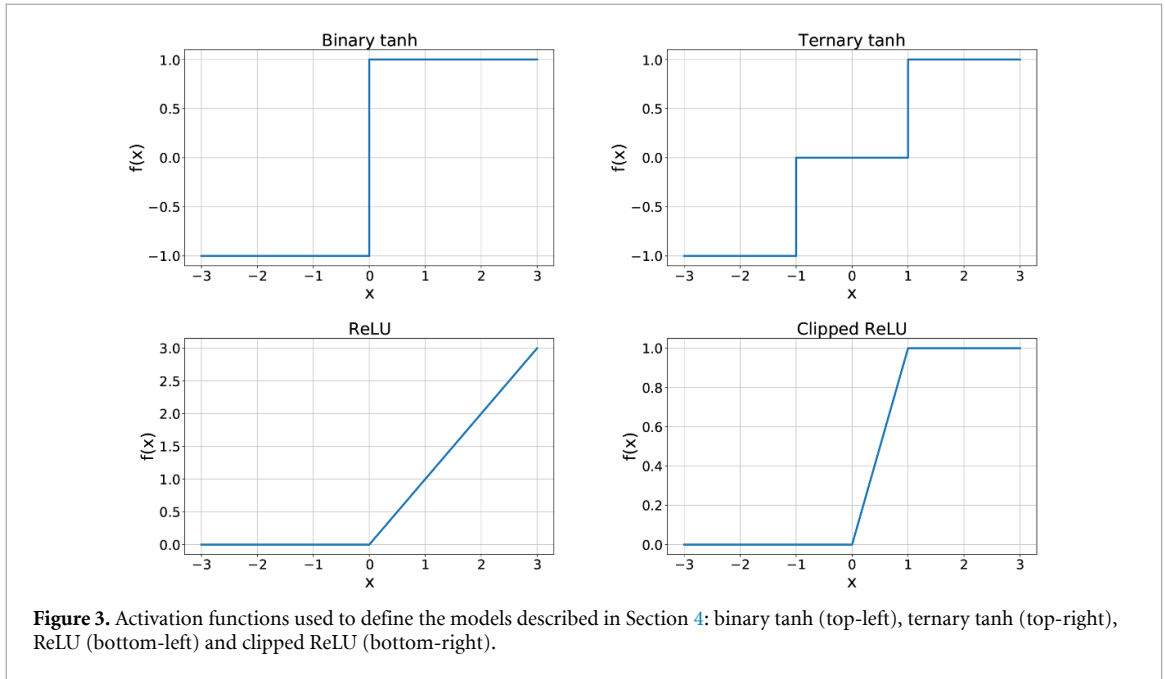


Table 2. Left: All possible products between A and B with values constrained to ± 1 . Right: The corresponding truth-table when the quantities A and B are each encoded with 1 bit, and the XNOR operation is used for the product.

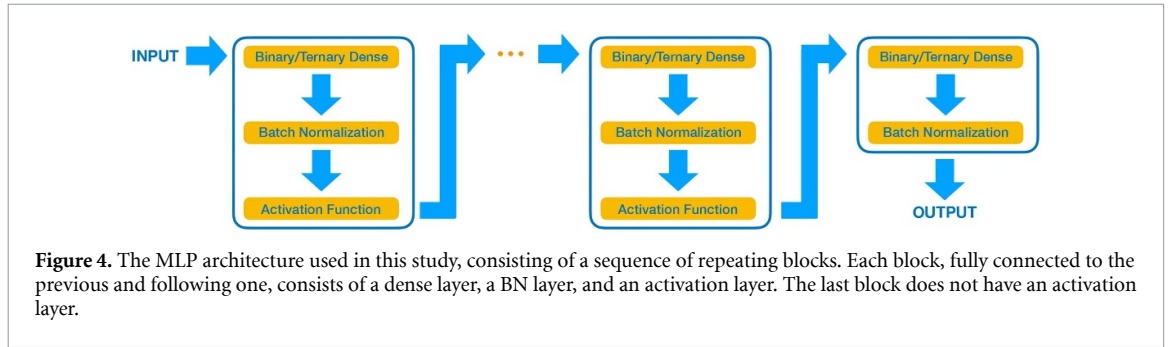
A	B	$A \times B$	A	B	$\overline{A \oplus B}$
-1	-1	1	0	0	1
-1	1	-1	0	1	0
1	-1	-1	1	0	0
1	1	1	1	1	1

Quantization reduces the computing resources of model inference and its level can be tuned to yield little or no loss in model 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 [27], defined as $\min(\text{ReLU}(x), y_{\max})$, with y_{\max} being a positive hyperparameter. In our study, we fix $y_{\max} = 1$. The four functions are shown in figure 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 reference [2]. In addition to that, we exploit a set of custom implementations [18], specific to binary and ternary networks, that allow one to speed up the execution of the building-block architecture shown in figure 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 [28]. With respect to the work presented in reference [2], this version provides a special support for large dense layers containing hundreds 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 optimized as an extremely lightweight operation. By encoding an arithmetical value of -1 as 0 , the product can be expressed as an XNOR operation. As described in table 2, an XNOR filter returns 0 when the two input values are different and 1 otherwise. For models using ternary weights or greater than 1 bit for activations, the much larger FPGA logic is always used rather than digital signal processing (arithmetic) blocks (DSPs), whose number is typically limited.

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 figure 3. A binary or ternary tanh activation layer preceded by a batch normalization (BN) layer [29] can be further optimized. The BN layer 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. The usual BN transformation y for an input x is

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta, \quad (3)$$

given the mean μ , variance σ^2 , scale γ , and shift β learned 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 DSPs, the four BN parameters are used to compute the value of x at which y flips sign. This calculation is performed at compilation 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 compilation time. In the FPGA, each node output is then simply compared against these precomputed thresholds, outputting the corresponding ± 1 , or 0. An additional optimization step sets the type of x in the HLS implementation to integer with a bit width corresponding to the largest integer expected for each binary/ternary layer, found at compilation time. This procedure further saves 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 customized 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 figure 4. Each model consists of a sequence of blocks, each composed of a dense, BN, and activation layer. For binary and ternary tanh, a 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 figure 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 figure 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 performed using GPyOpt [30], 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 figure 3.
- *Hybrid TNN*: same as the TNN model, but with ReLU or clipped ReLU activation functions rather than the ternary tanh of figure 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.

Not all of the operations or intermediate outputs of a binary (ternary) are represented 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 backend using the implementation in [19] for binary and ternary layers, which we also cross-checked with QKeras [31] with similar results. The network training was performed on an NVIDIA Tesla 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 [32]. While the hinge loss has been found to give the best performance for binary/ternary networks [15–17], the same choice for the baseline models is arbitrary. We have verified that the baseline models trained with the hinge loss after replacing the last softmax layer (figure 1) with a dense plus BN layers yield similar results in terms of both accuracy and resource usage.

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. For this configuration we study the FPGA resources used by the models described in section 4. There are four main resource categories: the on-board FPGA memory (BRAM), DSPs, and registers and programmable logic (flip-flops, or FFs, and lookup tables, or LUTs). Unless otherwise specified, the quoted results are derived after the HLS compilation step. The network implementation is further refined by the logic synthesis. This step 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. We verified that this final step does not affect the accuracy while it reduces the resource consumption.

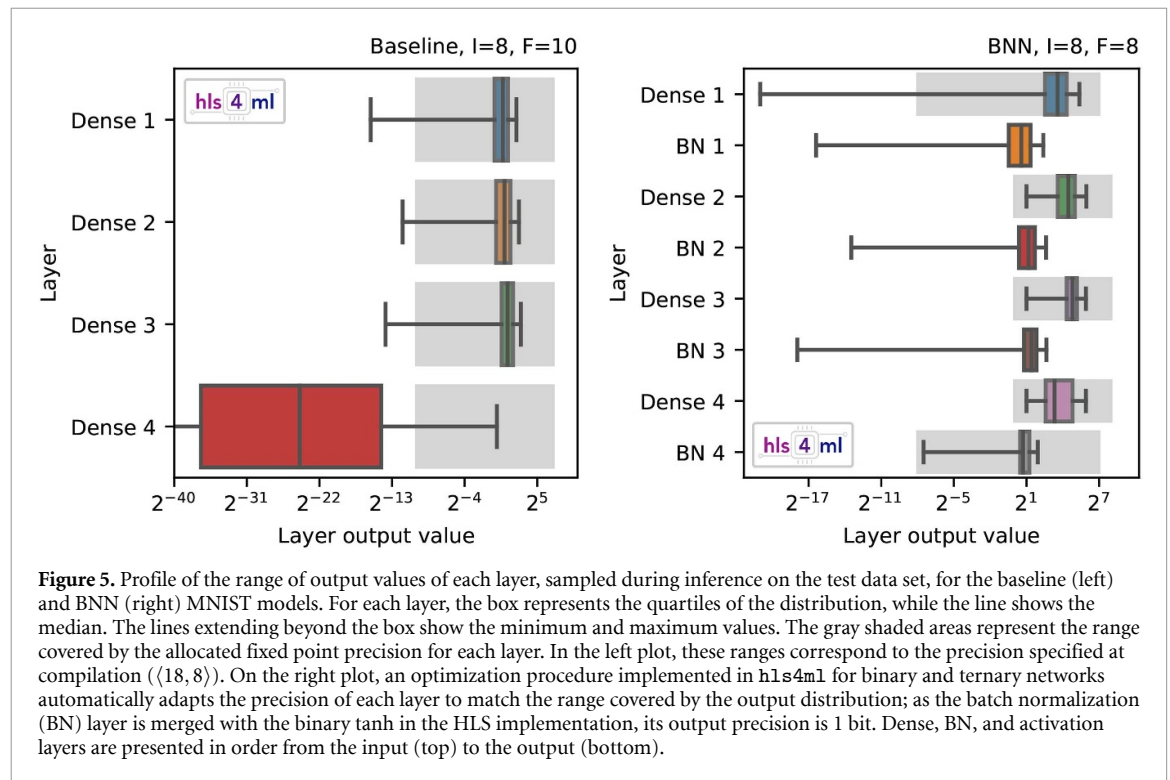
All results quoted in this section are taken from the numerical simulation of the synthesized firmware. This numerical simulation is one of the tools provided by the FPGA vendor and gives bit-identical results to running on a physical device. On the other hand, running on a physical device is a much more consuming operation. Given the large number of tests considered in this study, we omitted this last step, mainly for practical reasons.

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 $\langle T, I \rangle$, where $T = I + F$ is the total number of allocated bits. For each case, the minimum number of bits yielding an accuracy above 90% after

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 (I + F, I). For each case, the AUCs are reported as the range spanned by the classes with lowest and highest identification performance.

Model	Floating point precision		Fixed point precision		
	AUC	Accuracy [%]	Number of bits	AUC	Accuracy [%]
Baseline	0.999 1–0.999 7	98	(18, 8)	0.991 9–0.995 9	95
BNN	0.986 9–0.997 9	93	(16, 8)	0.986 0–0.997 6	93
TNN	0.992 1–0.999 2	95	(16, 6)	0.991 8–0.999 2	95
Hybrid BNN (ReLU)	0.995 3–0.999 0	95	(16, 10)	0.995 6–0.998 9	95
Hybrid TNN (ReLU)	0.997 0–0.999 3	96	(16, 10)	0.997 1–0.999 3	96
Hybrid BNN (clipped ReLU)	0.982 7–0.998 3	95	(16, 10)	0.982 8–0.998 3	95
Hybrid TNN (clipped ReLU)	0.985 7–0.998 9	96	(16, 10)	0.985 9–0.998 8	96



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.

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 figure 5 for the BNN model, where the gray areas cover different numerical ranges for different layers, despite the common precision specified at compilation ((16, 8) 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 figure 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 figure 5) does not cover the bulk of values returned by the layer

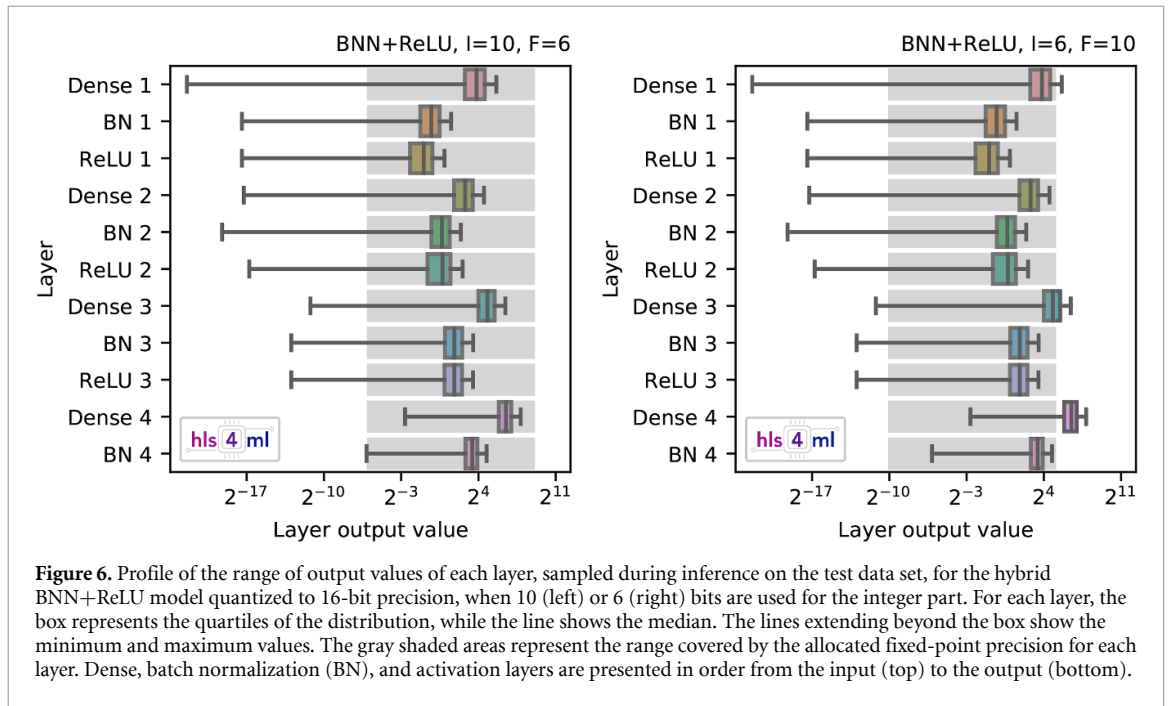


Figure 6. Profile of the range of output values of each layer, sampled during inference on the test data set, for the hybrid BNN+ReLU model quantized to 16-bit precision, when 10 (left) or 6 (right) bits are used for the integer part. For each layer, the box represents the quartiles of the distribution, 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).

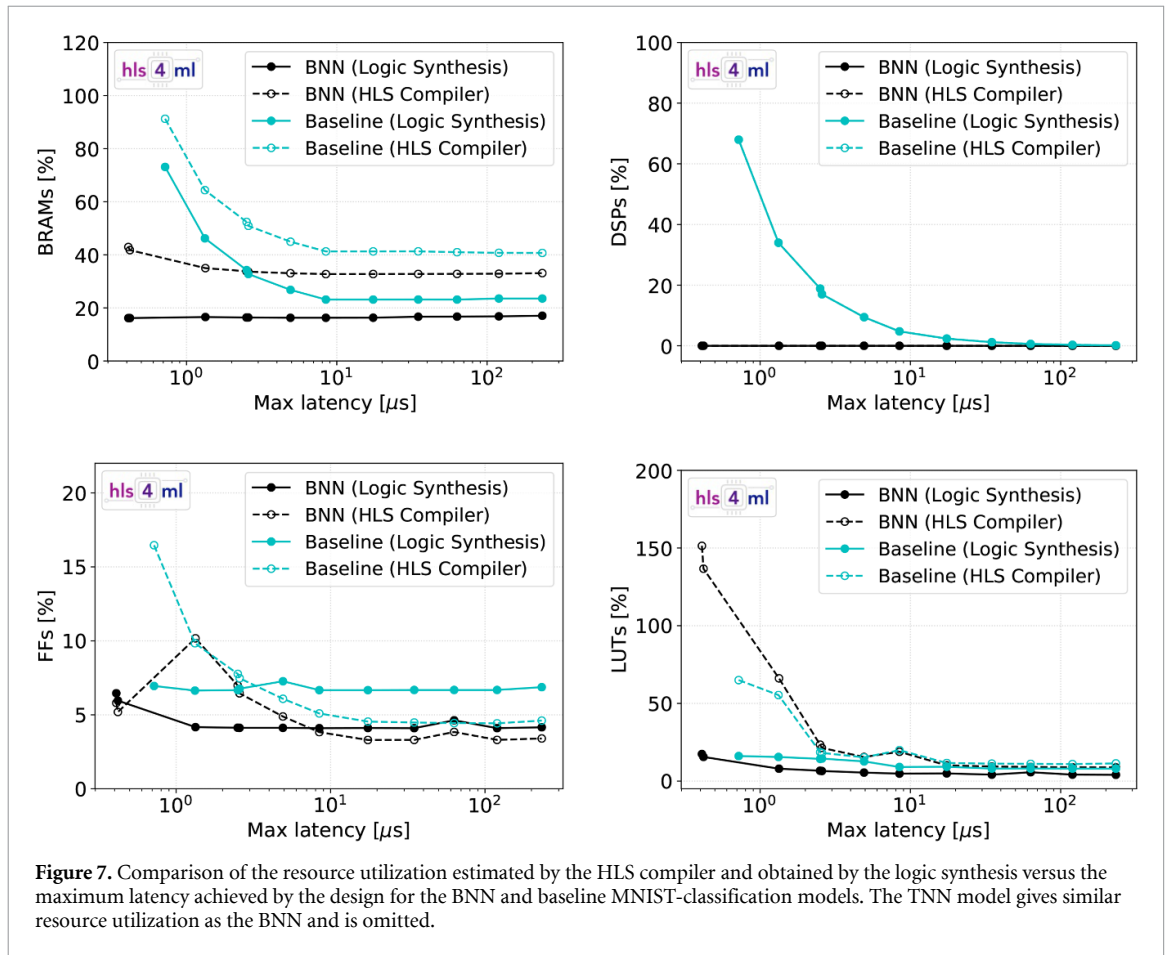
Model	II	Latency [ns]	DSPs [%]		FFs [%]		LUTs [%]		BRAMs [%]	
			C	S	C	S	C	S	C	S
Baseline	28	315	130	100	18	8	69	54	126	61
BNN	14	200	0	0	5	7	155	18	46	16
TNN	14	190	0	0	6	7	174	22	52	16
Hybrid BNN (ReLU)	14	200	1	0.16	7	9	215	31	52	16
Hybrid TNN (ReLU)	14	200	1	1	7	10	217	35	52	16
Hybrid BNN (clipped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

(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. For instance, this would not be a problematic aspect when operating this algorithm through the *Arg Max* function, associating a given example to the class with the largest output.

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 by removing the softmax function at the end of the HLS implementation of the inference, 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 table 3. For each configuration, we quote both the resource utilization estimated by the HLS compiler and



those obtained by the logic synthesis. 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 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 bit width 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, the number of DSPs used is greatly reduced for latency values up to a few μ s. 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. Due to technical aspects of the implementation of very-wide dense layers in hls4ml, it is not possible to configure the model to run with smaller latency values than those shown.

As a final test, we train a larger BNN model consisting of three dense layers with 256 nodes each, as in the study of reference [18], allowing for a direct comparison of our implementation of a binary architecture with what presented 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 0.31μ s with a resource utilization comparable to that in reference [18]. In particular,

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, the AUCs are reported as the range spanned by the classes with lowest and highest identification performance.

Model	Architecture	Floating point precision		Fixed point precision		
		AUC	Accuracy [%]	Number of bits	AUC	Accuracy [%]
Baseline	$16 \times 64 \times 32 \times 32 \times 5$	0.904–0.958	75	$\langle 16, 6 \rangle$	0.900–0.955	75
BNN	$16 \times 64 \times 32 \times 32 \times 5$	0.794–0.891	58	$\langle 16, 6 \rangle$	0.794–0.891	58
TNN	$16 \times 64 \times 32 \times 32 \times 5$	0.854–0.915	67	$\langle 16, 6 \rangle$	0.854–0.915	67
Best BNN	$16 \times 448 \times 224 \times 224 \times 5$	0.886–0.937	72	$\langle 16, 6 \rangle$	0.884–0.938	72
Best TNN	$16 \times 128 \times 64 \times 64 \times 64 \times 5$	0.886–0.931	72	$\langle 16, 6 \rangle$	0.886–0.930	72
Hybrid BNN (ReLU)	$16 \times 64 \times 32 \times 32 \times 5$	0.862–0.920	69	$\langle 16, 6 \rangle$	0.862–0.919	69
Hybrid TNN (ReLU)	$16 \times 64 \times 32 \times 32 \times 5$	0.874–0.934	70	$\langle 16, 6 \rangle$	0.874–0.934	70
Hybrid BNN (clipped ReLU)	$16 \times 64 \times 32 \times 32 \times 5$	0.852–0.916	67	$\langle 16, 6 \rangle$	0.852–0.916	67
Hybrid TNN (clipped ReLU)	$16 \times 64 \times 32 \times 32 \times 5$	0.874–0.921	70	$\langle 16, 6 \rangle$	0.874–0.921	70

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	50	2	2	0	0	6	2	0	0
Hybrid TNN (ReLU)	1	50	2	2	0	0	7	2	0	0
Hybrid BNN (clipped ReLU)	1	50	2	2	0	0	6	2	0	0
Hybrid TNN (clipped ReLU)	1	50	2	2	0	0	7	2	0	0

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. 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 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 $\sim 1 \mu\text{s}$ 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.

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.

Acknowledgments

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 no. 772 369). S J, M L, K P, and N T are supported by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11 359 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 #190 444, #1934 700, #1931 469, #1836 650. Z. W. is supported by the National Science Foundation under Grants No. 1606 321 and 115 164.

Data availability

The data that support the findings of this study are openly available at <https://doi.org/10.5281/zenodo.3602260> (LHC jet dataset) and <https://www.openml.org/d/554> (MNIST handwritten digit database).

ORCID iDs

Jennifer Ngadiuba  <https://orcid.org/0000-0002-0055-2935>

Javier Duarte  <https://orcid.org/0000-0002-5076-7096>

References

- [1] CMS Collaboration 2018 Boosted Decision Trees in the Level-1 Muon Endcap Trigger at CMS *J. Phys. Conf. Ser.* **1085** 042042
- [2] Duarte J et al 2018 Fast inference of deep neural networks in FPGAs for particle physics *JINST* **13** P07027
- [3] Schlag B 2018 Jet Reconstruction in the ATLAS Level-1 Calorimeter Trigger with Deep Artificial Neural Networks Presented <https://cds.cern.ch/record/2670301> 20 Aug
- [4] Wielgosz M and Karwatowski M 2019 Mapping neural networks to FPGA-based IoT devices for ultra-low latency processing *Sensors* **19**
- [5] Abadi M et al 2015 Tensorflow: large-scale machine learning on heterogeneous systems (<https://tensorflow.org/>)
- [6] Chollet F et al 2015 Keras (<https://keras.io>)
- [7] Paszke A et al 2019 PyTorch: An imperative style, high-performance deep learning library *Adv. Neural Information Process. Syst.* **32** 8024
- [8] Bai J et al 2019 Onnx: Open neural network exchange (<https://github.com/onnx/onnx>)
- [9] Summers S et al 2020 Fast inference of boosted decision trees in FPGAs for particle physics *JINST* **15** P05026
- [10] Cheng Y, Wang D, Zhou P and Zhang T 2018 Model compression and acceleration for deep neural networks: the principles, progress and challenges *IEEE Signal Process. Mag.* **35** 126
- [11] Choudhary T, Mishra V, Goswami A and Sarangapani J 2020 A comprehensive survey on model compression and acceleration *Artif. Intell. Rev.*
- [12] Han S, Mao H and Dally W J 2015 Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding *4th Int. Conf. on Learning Representations (ICLR)* arXiv:1510.00149
- [13] Han S, Pool J, Tran J and Dally W J 2015 Learning both weights and connections for efficient neural networks *Adv. Neural Inform. Process. Syst.* **28** 1135
- [14] Lin D D, Talathi S S and Annapureddy V S Fixed point quantization of deep convolutional networks arXiv:1511.06393
- [15] Hubara I et al 2016 Binarized neural networks *Adv. Neural Inform. Process. Syst.* **29** 4107

- [16] Courbariaux M, Bengio Y and David J 2015 BinaryConnect: training deep neural networks with binary weights during propagations *Adv. Neural Inform. Process. Syst.* **28** 3123
- [17] Li F and Liu B 2016 Ternary weight networks arXiv:1605.04711
- [18] Umuroglu Y et al 2017 FINN: A Framework for Fast, Scalable Binarized Neural Network Inference *Proc. of the 2017 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays* 65
- [19] Moons B, Goetschalckx K, Berckelaer N V and Verhelst M 2017 Minimum energy quantized neural networks *51st Conf. on Signals, Systems and Computers* arXiv:1711.00215
- [20] LeCun Y and Cortes C 2010 MNIST handwritten digit database (<http://yann.lecun.com/exdb/mnist/>)
- [21] Rosenblatt F 1958 The perceptron: a probabilistic model for information storage and organization in the brain *Psychol. Rev.* **65**
- [22] Hahnloser R H R et al 2000 Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit *Nature* **405** 947
- [23] Coleman E et al 2018 The importance of calorimetry for highly-boosted jet substructure *JINST* **13** T01003
- [24] Cacciari M, Salam G P and Soyez G 2008 The anti- k_r jet clustering algorithm *JHEP* **04** 063
- [25] Duarte J M et al 2020 HLS4ML LHC Jet dataset (150 particles)
- [26] Moreno E A et al 2020 JEDI-net: a jet identification algorithm based on interaction networks *Eur. Phys. J. C* **80** 58
- [27] Cai Z, He X, Sun J and Vasconcelos N 2017 Deep learning with low precision by half-wave Gaussian quantization *2017 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* 5406–14
- [28] Loncar V et al 2020 hls4ml: v0.1.6 February (<https://github.com/hls-fpga-machine-learning/hls4ml>)
- [29] Ioffe S and Szegedy C Batch normalization: accelerating deep network training by reducing internal covariate shift *ICML'15: Proc. of the 32nd Int. Conf. on Machine Learning* vol. **37** pp 448–56
- [30] The GPyOpt authors 2016 GPyOpt: A Bayesian Optimization framework in python (<http://github.com/SheffieldML/GPyOpt>)
- [31] Coelho C 2019 QKeras (<https://github.com/google/qkeras>)
- [32] Lin Y, Wahba G, Zhang H and Lee Y 2002 Statistical properties and adaptive tuning of support vector machines *Mach. Learn.* **48** 115

PAPER • OPEN ACCESS

Fast convolutional neural networks on FPGAs with hls4ml

To cite this article: Thea Aarrestad *et al* 2021 *Mach. Learn.: Sci. Technol.* **2** 045015

View the [article online](#) for updates and enhancements.

You may also like

- [Graph networks for molecular design](#)
Rocío Mercado, Tobias Rastemo, Edvard Lindelöf et al.
- [InClass nets: independent classifier networks for nonparametric estimation of conditional independence mixture models and unsupervised classification](#)
Konstantin T Matchev and Prasanth Shyamsundar
- [Point cloud transformers applied to collider physics](#)
Vinicius Mikuni and Florencia Canelli



PAPER

Fast convolutional neural networks on FPGAs with hls4ml

OPEN ACCESS

RECEIVED
15 January 2021REVISED
24 April 2021ACCEPTED FOR PUBLICATION
25 June 2021PUBLISHED
16 July 2021

Original Content from
this work may be used
under the terms of the
[Creative Commons
Attribution 4.0 licence](#).

Any further distribution
of this work must
maintain attribution to
the author(s) and the title
of the work, journal
citation and DOI.



Thea Aarrestad^{1,*}, Vladimir Loncar^{1,13}, Nicolò Ghielmetti^{1,14}, Maurizio Pierini¹, Sioni Summers¹, Jennifer Ngadiuba², Christoffer Petersson^{3,15}, Hampus Linander³, Yutaro Iiyama⁴, Giuseppe Di Guglielmo⁵, Javier Duarte⁶, Philip Harris⁷, Dylan Rankin⁷, Sergio Jindariani⁸, Kevin Pedro⁸, Nhan Tran⁸, Mia Liu⁹, Edward Kreinar¹⁰, Zhenbin Wu¹¹ and Duc Hoang¹²

¹ European Organization for Nuclear Research (CERN), CH-1211 Geneva 23, Switzerland

² California Institute of Technology, Pasadena, CA 91125, United States of America

³ Zenseact, Gothenburg 41756, Sweden

⁴ ICEPP, University of Tokyo, Tokyo, Japan

⁵ Columbia University, New York, NY 10027, United States of America

⁶ University of California San Diego, La Jolla, CA 92093, United States of America

⁷ Massachusetts Institute of Technology, Cambridge, MA 02139, United States of America

⁸ Fermi National Accelerator Laboratory, Batavia, IL 60510, United States of America

⁹ Purdue University, West Lafayette, IN 47907, United States of America

¹⁰ HawkEye360, Herndon, VA 20170, United States of America

¹¹ University of Illinois at Chicago, Chicago, IL 60607, United States of America

¹² Rhodes College, Memphis, TN 38112, United States of America

¹³ Also at Institute of Physics Belgrade, Serbia

¹⁴ Also at Politecnico di Milano, Italy

¹⁵ Also at Chalmers University of Technology, Sweden

* Author to whom any correspondence should be addressed.

E-mail: thea.aarrestad@cern.ch

Keywords: deep learning, FPGA, convolutional neural network

Abstract

We introduce an automated tool for deploying ultra low-latency, low-power deep neural networks with convolutional layers on field-programmable gate arrays (FPGAs). By extending the hls4ml library, we demonstrate an inference latency of $5 \mu\text{s}$ using convolutional architectures, targeting microsecond latency applications like those at the CERN Large Hadron Collider. Considering benchmark models trained on the Street View House Numbers Dataset, we demonstrate various methods for model compression in order to fit the computational constraints of a typical FPGA device used in trigger and data acquisition systems of particle detectors. In particular, we discuss pruning and quantization-aware training, and demonstrate how resource utilization can be significantly reduced with little to no loss in model accuracy. We show that the FPGA critical resource consumption can be reduced by 97% with zero loss in model accuracy, and by 99% when tolerating a 6% accuracy degradation.

1. Introduction

The hls4ml library [1, 2] is an open source software designed to facilitate the deployment of machine learning (ML) models on field-programmable gate arrays (FPGAs), targeting low-latency and low-power edge applications. Taking as input a neural network model, hls4ml generates C/C++ code designed to be transpiled into FPGA firmware by processing it with a high-level synthesis (HLS) library. The development of hls4ml was historically driven by the need to integrate ML algorithms in the first stage of the real-time data processing of particle physics experiments operating at the CERN Large Hadron Collider (LHC). The LHC produces high-energy proton collisions (or *events*) every 25 ns, each consisting of about 1 MB of raw data. Since this throughput is overwhelming for the currently available processing and storage resources, the LHC experiments run a real-time event selection system, the so-called level-1 trigger (L1T), to reduce the event rate from 40 MHz to 100 kHz [3–6]. Due to the size of the buffering system, the L1T system operates with a fixed latency of $\mathcal{O}(1 \mu\text{s})$. While hls4ml excels as a tool to automatically generate low-latency ML firmware

for L1T applications, it also offers interesting opportunities for edge-computing applications beyond particle physics whenever efficient, e.g. low power or low latency, on-sensor edge processing is required.

The `hls4ml` software is structured with a set of different back-ends, each supporting a different HLS library and targeting different FPGA vendors. So far, new development has been focused on the Vivado HLS [7] back-end targeting Xilinx FPGAs. We have demonstrated this workflow for fully-connected, or dense, neural networks (DNNs) [1], binary and ternary networks [8], boosted decision trees [9], and graph neural networks [10, 11]. The `hls4ml` library accepts models from TENSORFLOW [12], KERAS [13], PYTORCH [14], and via the ONNX interface [15]. It has recently been interfaced to QKERAS [16], in order to support quantization-aware training (QAT) allowing the user to better balance resource utilization and accuracy.

The `hls4ml` design focuses on fully-on-chip deployment of neural network architectures. This avoids the latency overhead incurred by data transmission between the embedded processing elements and off-chip memory, reducing the overall inference latency. Conversely, this approach constrains the size and complexity of the models that the HLS conversion can easily support. Nevertheless, complex architectures can be supported, as discussed in [10, 11] in the case of graph neural networks.

In this paper, we introduce support for convolutional neural networks (CNNs), through the implementation of streaming-based novel convolutional and pooling layers.

Given the larger number of operations associated to each convolutional layer, a successful deployment on FPGA relies on model compression, through pruning and quantization. The `hls4ml` library supports both these forms of compression through removal of all zero-multiplications during the firmware implementation (a feature of HLS we take advantage of when designing the layer implementation), and through its interface with QKERAS [16].

We demonstrate the QKERAS + `hls4ml` workflow on a digit classifier trained on the Street View House Numbers (SVHN) dataset [17], with a depth and input size appropriate for the latency- and resource-restricted triggering systems at LHC.

This paper is organized as follows: section 2 describes related works. Section 3 introduces the stream-based implementation of CNN layers; section 4 describes the SVHN dataset. The benchmark model is introduced in section 5, while results obtained by pruning and quantization (after and during training) are presented in sections 6 and 7, respectively. Section 8 discusses the model porting to FPGAs. Conclusions are given in section 9.

2. Related work

An early attempt to deploy CNNs on FPGAs for particle physics was shown in [18], and surveys of other existing toolflows for mapping CNNs on FPGAs are given in [19–22]. The FINN [23, 24] framework from Xilinx Research Labs is designed to explore quantized CNN inference on FPGAs, with emphasis on generating dataflow-style architectures customized for each network. It includes tools for training quantized NNs such as BREVITAS [25], the FINN compiler, and the `finn-hlslib` Vivado HLS library of FPGA components for QNNs. The `fpgaConvNet` library [26–29] converts CNNs specified in Caffe [30] or Torch formats into generated Xilinx Vivado HLS code with a streaming architecture. FP-DNN [31] is a framework that takes TENSORFLOW [12]-described CNNs as input, and generates the hardware implementations on FPGA boards with register transfer level (RTL)-HLS hybrid templates. DNNWeaver [32] is an open-source alternative, which also supports CNNs specified in Caffe format and automatically generates the accelerator Verilog code using hand-optimized Verilog templates with a high degree of portability. Caffeine [33] is another CNN accelerator for Caffe-specified models targeting Xilinx devices that support a co-processing environment with a PCIe interface between the FPGA and a host. Snowflake [34] is a scalable and efficient CNN accelerator with models specified in Torch [35] and a single, sequential computation architecture designed to perform at near-peak hardware utilization targeting Xilinx system-on-chips (SoCs). In [36], an FPGA-based accelerator design to execute CNNs is proposed, leveraging TENSORFLOW for model description and exploiting reuse along all dimensions with a 1D systolic array of processing elements. The NullHop [37] accelerator architecture takes advantage of sparse computation in convolutional layers to significantly speed up inference times. A flexible, efficient 3D neuron array architecture for CNNs on FPGAs is presented in [38], describing a technique to optimize its parameters including on-chip buffer sizes for a given set of resource constraint for modern FPGAs. Vitis AI [39] is Xilinx's development platform for AI inference on Xilinx hardware platforms, consisting of optimized IP cores, tools, libraries, models, and example designs for both edge devices and Alveo cards.

Our approach is distinct from many of those above with its emphasis on being a completely open-source and multi-backend tool. In addition, a fully on-chip design is embraced in order to target the microsecond latency imposed in LHC physics experiments.

3. Convolutional layers implementation in hls4ml

A direct implementation of a two-dimensional convolutional layer (Conv2D) requires six nested loops over image height H , width W , number of input channels C , number of output filters N , and filter height J and width K [22]. In particular, calculating one element of the $V \times U \times N$ output tensor Y of a Conv2D layer from the $H \times W \times C$ input tensor X , $J \times K \times C \times N$ weight tensor W , and length- N bias vector β requires three nested loops¹⁶,

$$Y[v, u, n] = \beta[n] + \sum_{c=1}^C \sum_{j=1}^J \sum_{k=1}^K X[v+j, u+k, c] W[j, k, c, n], \quad (1)$$

and repeating the calculation for all output elements $\{u, v, n\} \in \times [1, V] \times [1, U] \times [1, N]$ requires three additional nested loops. For simplicity, we assume $J = K$ (square kernel) in the remainder of this paper.

Without additional optimizations, a plain implementation of these nested loops would result in high latency because, in the RTL implementation, one clock cycle is required to move from an outer loop to an inner loop, and another one to move from an inner loop to an outer loop. This is usually addressed with loop pipelining. However, pipelining an outer loop requires completely parallelizing (or *unrolling*) all nested inner loops, which significantly increases the size of the RTL implementation and the resources used. This approach is then feasible only for very small input sizes or model architectures. Utilizing this direct approach, the total number of unrolled loop iterations (the product $K^2 VUN$) was limited to be less 4096 to avoid the Vivado HLS partitioning limit.

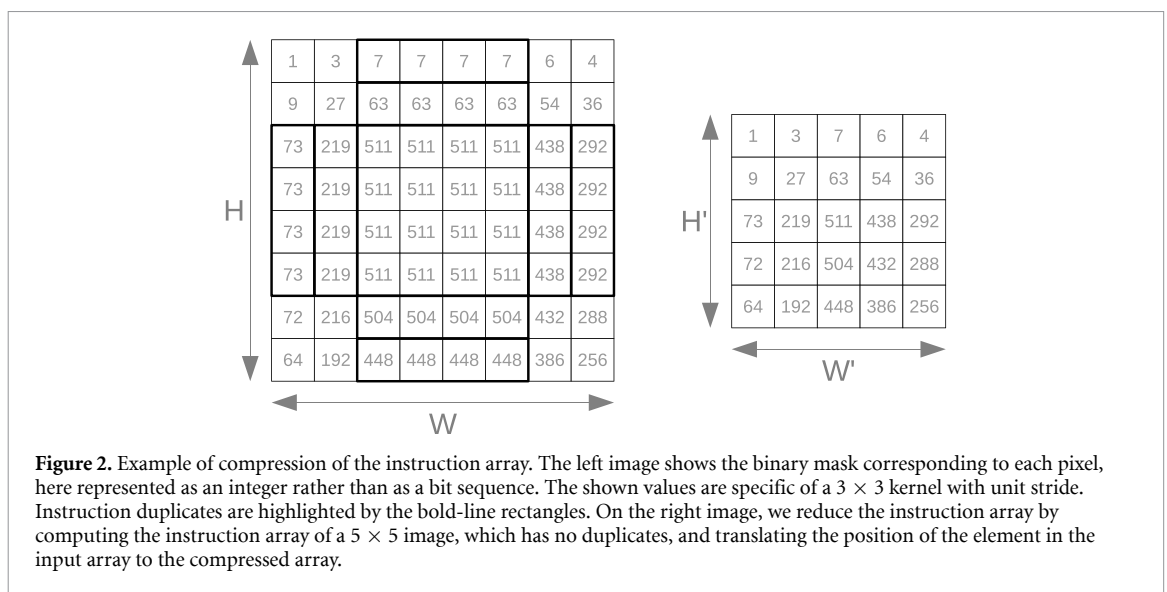
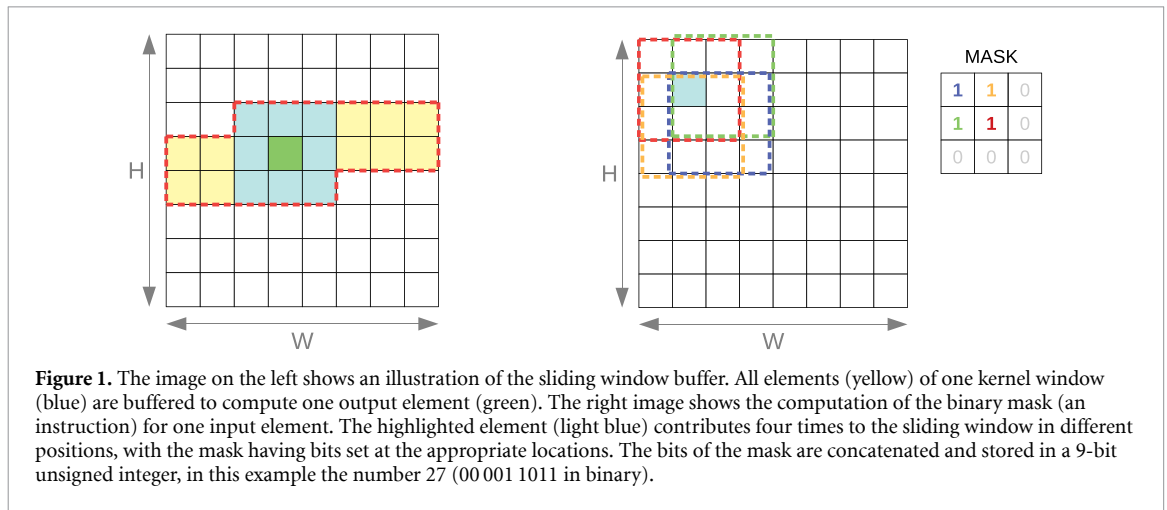
While the direct implementation has the advantage of not requiring extra memory for temporary storage, on most modern computing architectures convolution is implemented using general matrix multiplication, with algorithms like `im2col` and `kn2row` [40]. In `im2col`, each input window is flattened into a column vector and stacked together to form the input matrix, while the kernels are flattened into row vectors and concatenated to form the weight matrix. Matrix multiplication can then be performed using the accelerated library available on the platform, for example using the routines from basic linear algebra subprograms. While this approach can be implemented on FPGAs using HLS, the design choices of `hls4ml`, particularly the decision to store all tensors on the chip itself, mean this approach requires additional $\mathcal{O}(K^2 HWC)$ units of memory, either as block random access memory (BRAM) or registers, to store the input matrix. Additionally, to achieve the lowest latency, `hls4ml` completely partitions the input arrays into individual registers as this allows access to each element within the same clock cycle. This strategy works well for fully connected layers but in case of convolutional layers the input tensor is usually much larger. Following this strategy, one would quickly reach the partitioning limit of Vivado HLS. Relaxing this constraint and using block or cyclic partitioning to create multiple array slices presents another challenge as the access pattern between consecutive layers of the model has to be consistent, otherwise scheduling issues arise and the design may fail to meet timing constraints.

To avoid these limitations, we have implemented convolutional layers using streams. Streams are synthesized in hardware as first in, first out (FIFO) buffers and as they do not require additional address management, they consume less resources than designs based on arrays. Since streams only allow sequential access, and have additional limitations on the reads and writes from different tasks (C++ functions), this requires re-implementing most of the neural network layers in `hls4ml` to support sequential processing.

Our implementation uses an approach similar to the `im2col` algorithm. However, it does not build the entire input matrix, and rather considers one column vector at a time. This allows us to reuse the existing matrix-vector multiplication functions of `hls4ml`. In order to use streams for this implementation, a special C++ class `hls::stream<>` provided in Vivado HLS is used. Given an $H \times W \times C$ input image tensor, we create a stream of HW items where each item is an array containing the C elements. This scheme allows us to efficiently read from the stream and construct column vectors. Because it usually takes one cycle to read or write one element of the stream, the latency of the layer will be at least HW cycles.

Processing input sequentially through streams requires that we buffer all values that we wish to reuse at a later stage as an internal state. For a two-dimensional convolution, we need to buffer all values between the first and last element of the convolutional kernel, or *sliding window*, as shown on the left in figure 1. The buffer can be defined as an array in C++ and implemented by the HLS compiler as a shift register, however this approach requires keeping track of the position in the array, further complicating the implementation. We choose a simpler approach using streams. We create K^2 streams, corresponding to the size of the sliding window, and buffer values at the appropriate position in the window as they stream in. The depth of these

¹⁶ Note that in practice X in equation (1) is shifted by e.g. $(\frac{J+1}{2}, \frac{K+1}{2})$ in order to be symmetric around (v, u) .



streams is determined by the width of the output image and the square kernel size. Once we reach an element that is at the last position of a sliding window, we can compute one output by reading from the buffer. This is highly efficient as we can read the entire column vector in one clock cycle. With the column vector prepared, we can invoke the multiplication with the weight matrix and store the result in the output stream.

While the algorithm described so far allows us to process larger inputs than a plain implementation would, significant resources are allocated for accounting, e.g. the position of the element in the sliding window or handling of the corners of the input image, and this prevents pipelining of loops at the desired latency. We address this issue by eliminating all branching code that handles these special cases, along with their associated state variables, leaving only the internal sliding window buffer. Instead, we pre-compute the positions in the sliding window where a given input element is used, and store this information as a binary mask, represented as a K^2 -bit unsigned integer. In the mask we set bits corresponding to every position in the sliding window where the input element is used, and leave the remaining bits unset (equal to 0), as illustrated on the right in figure 1. This mask can be used as an instruction on how to populate the sliding window buffer, eliminating the need for all branching code. The procedure is applied to every element of the input image, and stored in the instruction array. The instruction array can be significantly compressed by eliminating duplicates and translating the position of the element in the input array to the compressed array. As an example of the compression scheme, figure 2 illustrates how every convolution with a 3×3 kernel and unit stride can be represented with only $H' \times W' = 5 \times 5$ instructions, regardless of the input image size ($H \times W$).

For the pooling layers, a similar technique is used to collect the data in sliding window buffers. As the most common form of pooling assumes a stride equal to the pooling region size (i.e. no overlaps between pooled regions), we create a simpler and more optimal instruction-encoding scheme for this case. Unlike convolution, in both max and average pooling operation, we do not need the position of elements in the

sliding window, only which window they belong to. This allows us to create a simple lookup table (LUT) of $H + W$ elements without the need for translation of the position of the input element.

4. Dataset

To demonstrate the functionality of CNNs in hls4m1, we consider as a benchmark example a digit classifier trained on the SVHN Dataset [17]. The SVHN dataset consists of cropped real-world images of house numbers extracted from Google Street View images, in a format similar to that of the MNIST [41] dataset. However, it presents a much more challenging real-world problem, as illustrated by the examples shown in figure 3. The numbers are part of natural scene images, possibly with other digits appearing as a background on the two sides of the central one, different colors and focus, orientation, etc.

All the images are in RGB format and have been cropped to 32×32 pixels. Unlike MNIST, more than one digit can be present in the same image. In these cases, the center digit is used to assign a label to the image, which is then used as ground truth when training the classifier. Each image can belong to one of ten classes, corresponding to digits '0' through '9'. As a preprocessing step, we divide each pixel by the max RGB value of 255 in order to have numbers in the range between zero and one. We then standardize the input images to have a mean of zero and unit variance by applying a per-pixel scaling factor computed from the full training dataset. The same scaling is applied to the test set.

The SVHN dataset consists of 604 388 images for training (of which 531 131 are considered extra data that are slightly easier to classify) and 26 032 images for testing.

Training is performed using a k -fold cross-validation procedure. The training dataset is split in ten training and validation samples such that 10% of the training set is used for validation and the remaining 90% for training. Training and validation is then repeated k times until each fold of the training set has been used to evaluate the accuracy of the model. Model-performance figures of merit (e.g. accuracy, true and false positive rates (FPRs), etc) are defined considering the mean across the ten folds on the test set. The corresponding uncertainty is quantified through the standard deviation across the ten folds.

5. Baseline model

Keeping in mind that the model is designed for deployment on the resource limited environment of an FPGA, we limit the depth and complexity of the baseline model while preserving reasonable performance. As a target, we aimed at a test error close to 5%, where state-of-the-art test error lies between 1% and 5% [42–47]. To reduce the overall model latency as much as possible, models with fewer large layers (wider) are preferred over models with several smaller layers (deeper). This is due to the parallel nature of the FPGA, making it more resource-efficient to process one large layer in parallel over several small ones sequentially. The dependency of inference latency and resource consumption for increasing depth and width will be further discussed in section 8.

A Bayesian optimization over the model hyperparameters is performed using KERAS TUNER [48]. The first few layers are chosen to be 2D convolutional blocks. Each block consists of a convolutional layer followed by a max pooling layer, a batch normalization [49] layer, and a rectified linear unit (ReLU) [50, 51] activation function. The optimization range is set so that the maximum number of loop iterations per layer is below the unroll limit described in section 3 in order to achieve the lowest possible latency. Pooling layers are used to keep the size of the final dense layers small.

The convolutional blocks are followed by a series of fully-connected layers, the amount of layers and their size again determined through the hyperparameter optimization. A final ten-node dense layer, activated by a softmax function, returns the probability for a given image to be assigned to each of the ten classes. The result of the Bayesian optimization, shown in figure 4, consists of three convolutional blocks and two dense layers. The convolutional layers in the three blocks have 16, 16, and 24 filters, respectively, and each has a kernel size of 3×3 . The pooling layers have a size of 2×2 . The two hidden dense layers consist of 42 and 64 neurons, with batch normalization and ReLU activation. The model is implemented in TENSORFLOW [12], using the KERAS API [13]. To reduce the number of required operations, the bias term is removed from all layers, except for the final output layer, while keeping batch normalization on to prevent internal covariate shift [49].

We refer to this model as the baseline floating-point (BF) model. The number of floating-point operations (FLOPs) and weights for each convolutional or dense layer is listed in table 1. In addition, an estimate of the per-layer energy consumption and the layer size in bits is quoted. These estimates are obtained using QTOOLS [16], a library for estimating model size and energy consumption, assuming a 45 nm process [52]. Despite the first dense layer having the most weights, the number of FLOPs and the energy consumption is significantly higher in the convolutional layers due to the much larger number of multiply-accumulate operations performed. The per-layer summaries does not include results for batch



Figure 3. Examples of digit images extracted from the SVHN train (three leftmost images) and test (three rightmost images) datasets.

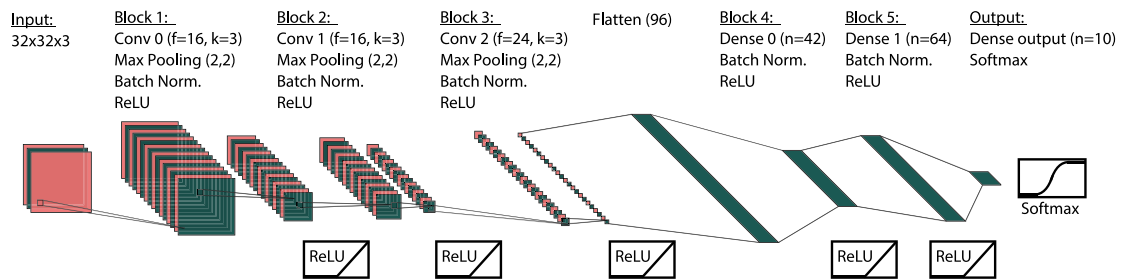


Figure 4. The neural network architecture, chosen through a Bayesian optimization over the hyperparameters, for classifying digits from the SVHN dataset. Each convolutional block consists of a convolutional layer, max pooling, batch normalization, and ReLU activation. The convolutional layers in the three convolutional blocks use 16, 16, and 24 filters, respectively, and each has a kernel size of 3×3 . The pooling layers have a size of 2×2 . The convolutional blocks are followed by two fully-connected layers consisting of 42 and 64 neurons, with batch normalization and ReLU activation. The bias term is removed from all layers except the final output layer.

Table 1. Number of trainable weights, floating-point operations, energy consumption and layer size in bits for each convolutional or dense layer (not including the activation layers). Batch normalization and pooling layers are not included as they are negligible in size and energy consumption in comparison. The energy is estimated assuming a 45 nm process using QTOOLS. The total energy and bit size includes all model layers.

Layer name	Layer type	Input shape	Weights	MFLOPs	Energy (nJ)	Bit size
Conv 0	Conv2D	(32, 32, 3)	432	0.778	1795	3456
Conv 1	Conv2D	(15, 15, 16)	2304	0.779	1802	18 432
Conv 2	Conv2D	(6, 6, 16)	3456	0.110	262	27 648
Dense 0	Dense	(96)	4032	0.008	26	32 256
Dense 1	Dense	(42)	2688	0.005	17	21 504
Output	Dense	(64)	65	0.001	4	5200
Model total			12 858	1.71	3918	170 816

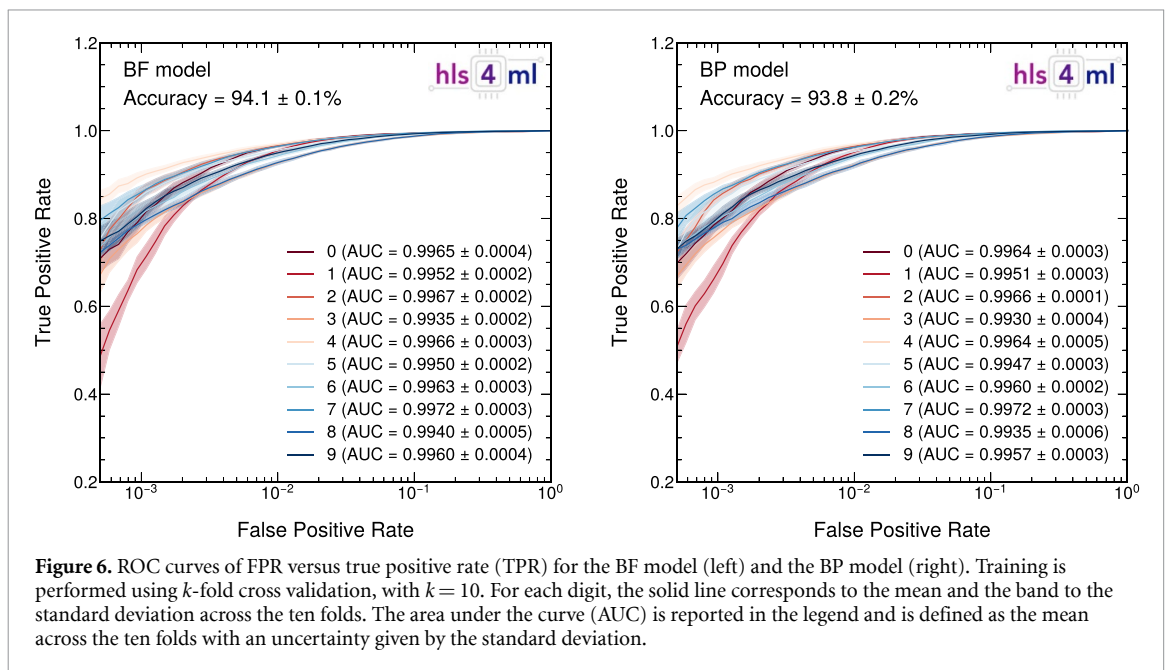
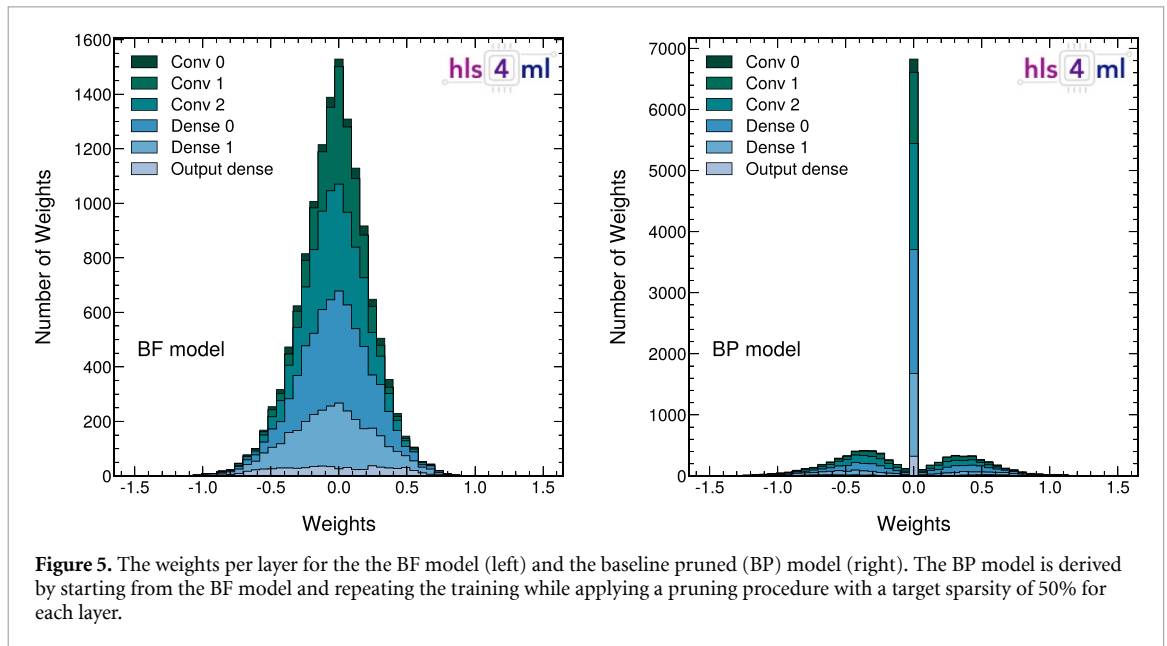
normalization or pooling layers, as the contribution from these are negligible in comparison. The total model energy and bit size includes all layers of the model.

The training is performed minimizing the categorical crossentropy loss [53] using the Adam optimizer [54]. The optimal learning rate is obtained using the hyperparameter optimization described above, found to be 0.003, and is set as the starting learning rate. If there is no improvement in the loss for five epochs, the learning rate is reduced by 90% until a minimum learning rate of 10^{-6} is reached. The batch size is 1024 and the training takes at most 100 epochs. Early stopping is enabled when no improvement in the validation loss is observed across ten epochs.

6. Compression by pruning

Weight pruning is an established strategy to compress a neural network and consequently reducing its resource utilization. One strategy, magnitude-based pruning, consists of eliminating redundant weights in the weight tensors by setting the value of the smallest weights in a tensor to zero [1, 55–59]. All zero-weight multiplications are omitted by the HLS library when translating the network into firmware, consequently saving significant FPGA resources.

Pruning is enforced using the TENSORFLOW pruning API, a KERAS-based interface consisting of a simple drop-in replacement of KERAS layers. A sparsity of 50% is targeted, meaning only 50% of the weights are retained in the pruned layer and the remaining ones are set to zero. Before pruning, the weights of each layer are initialized to the weights of the corresponding model without pruning (i.e. *fine-tuning pruning*), ensuring the model is in a stable minimum before removing weights deemed unimportant. Each model is pruned



starting from the 10th epoch, with the target sparsity gradually increasing to the desired 50% with a polynomial decay of the pruning rate [60].

By pruning the BF model layers as listed in table 1 to a target sparsity of 50%, the number of FLOPs required when evaluating the model, can be significantly reduced. We refer to the resulting model as the BP model.

The distribution of the weight values per layer for the BF and BP models are shown in figure 5. The effect of pruning is seen by comparing the two distributions: the smallest magnitude weights of the BF weight distribution migrate to the spike at zero in the BP weight distribution, while the two tails remain populated, with most of the weights falling in the interval $[-1.5, 1.5]$.

Figure 6 compares the classification performance of the BF and BP models. Specifically, it shows the receiver operating characteristic (ROC) curves and the corresponding AUC for each digit. In addition, we consider the model accuracy, i.e. how often the predictions (after taking the arg max of the output neurons) equals the labels. For each ROC, the solid line corresponds to the mean across the ten folds and the uncertainty to the standard deviation. The mean accuracy and standard deviation across the ten folds is also reported on the plot. Despite removing 50% of the weights for the BP model, the model accuracy is comparable between the BF and BP models.

These models serve as our reference models. The accuracy, latency and resource consumption of these will be discussed in section 8. In general, we observe a significant reduction in FPGA resource consumption for pruned models, as zero-weight multiplications are optimized away by the HLS compiler. Because pruning has little impact on the model accuracy (as demonstrated in figure 6), pruning is always recommended before translation into FPGA firmware with `hls4ml`.

7. Compression by quantization

To further limit the model footprint, we reduce the numerical precision of the model weights before FPGA deployment. During training, one typically relies on single- or double-precision floating-point arithmetic, i.e. 32 or 64 bit precision. However, when deploying a deep neural network on FPGA, reduced precision fixed-point arithmetic (quantization) is often used in order to minimize resource consumption and latency. It has been shown that deep neural networks experience little accuracy loss when QAT is applied, even up to binary quantization of weights [61].

When a quantized model is deployed on an FPGA, all its weights, biases, and activation functions are converted to fixed-point precision before being deployed. This is referred to as post-training quantization (PTQ). The chosen precision is a new tunable hyperparameter. The `hls4ml` library allows users to specify different numerical precisions for different components of the network (known as *heterogeneous quantization*). For instance, it is found that severe PTQ of the activation functions typically results in a greater reduction of accuracy than severe PTQ of the weights [8]. By default, `hls4ml` assumes 16 total bits for every layer, 6 of which are dedicated to the integer part ((16, 6) precision).

In this paper, we consider two approaches to network quantization: PTQ of a floating-point model, and QAT, resulting in a model already optimized for fixed-point precision. Both methods will be described in the following and the result on hardware discussed in detail in section 8. To summarize, we observe a significant reduction in accuracy using PTQ, with no prediction power remaining below a bit width of 14. Using QAT, however, high accuracy is maintained down to extremely narrow bit widths of 3–4. The latency and resource consumption are similar for the two methods (with certain caveats that will be discussed in section 8), and QAT is therefore the preferred solution for model quantization before deployment with `hls4ml`.

7.1. Post-training quantization

The `hls4ml` library converts model weights and biases from floating-point to fixed-point precision, applying the same quantization to the whole network or setting the precision per layer and per parameter type. Bit width and number of integer bits must be tuned carefully to prevent compromising the model accuracy. For each component, an appropriate precision is selected by studying the floating-point weight profiles, i.e. the range of input or output values spanned by the testing data for the trained model, component by component. In order to minimize the impact of quantization on accuracy, the precision can be tuned so that the numerical representation adequately covers the range of values observed in the floating-point activation profile.

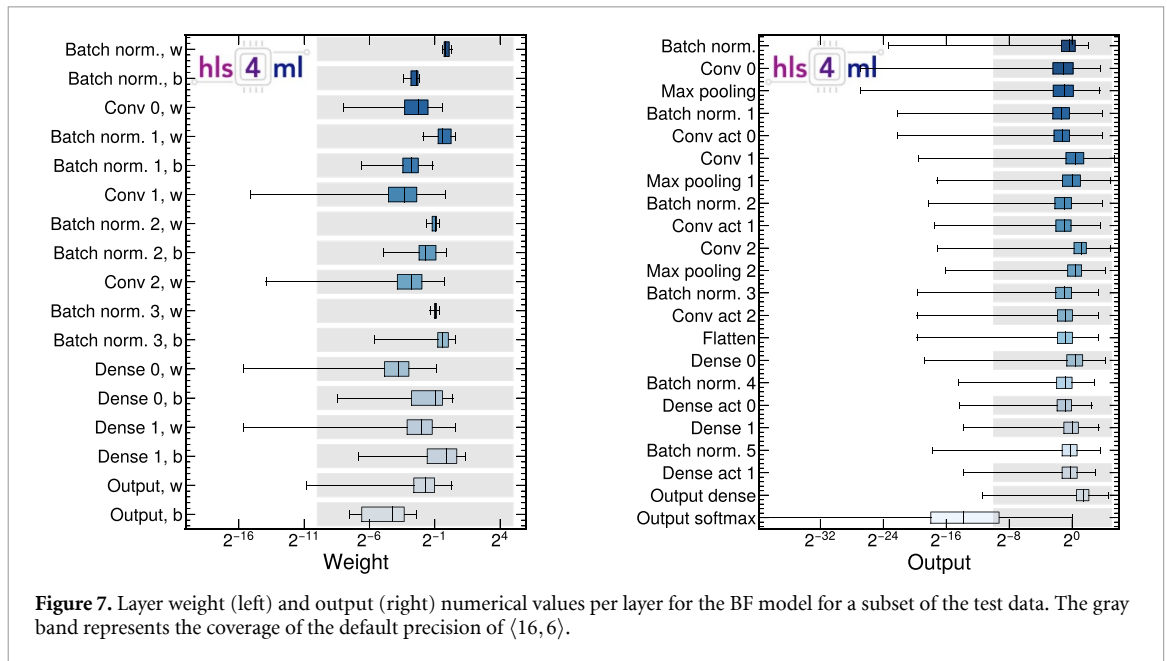
As an example, the by-layer weight profiles of the BF model is shown in figure 7, for both layer weights (left) and outputs (right) using the testing data. The last letter in the label indicates which type of weight is being profiled, where w is for weights and b is for bias. Learnable bias parameters are only included in the final dense layer. The other bias terms are introduced by the fusing of a batch normalization and a dense layer. The gray bands illustrate the numerical range covered by the default (16, 6) precision. No gray band is visible for the Flatten layer as no operations changing the data is performed. Also no gray band is visible for the 4th and 5th batch normalization layer outputs as these are fused with the dense layers.

Typically, extreme PTQ results in a sizeable accuracy loss. The increased spacing between representable numbers enforces a severe weight rounding that leads to a significant reduction in the model accuracy once the resolution becomes too coarse. The amount of compression one can reach by this procedure is balanced by the need to preserve the model accuracy, and how much model accuracy reduction can be tolerated is an application-specific question.

We use PTQ to generate a range of compressed models, further discussed in section 8, scanning bit widths from 16 to 1, with 6 integer bits.

7.2. Quantization-aware training

QAT [62] is an efficient procedure to limit accuracy loss while reducing the numerical precision of the network components. Here, quantized weights and biases are used in the training during the forward pass, while full precision is used in the backward pass in order to facilitate the drift toward the optimal point in the loss minimization (known as the *straight-through estimator*) [63]. The `hls4ml` library supports QAT through its interface to QKERAS [16].



We train a range of quantized QKERAS models using the same architecture as in figure 4, imposing a common bit width across the model. We scan the bit width from 16 to 3, as well as train a ternary and a binary quantized model. We refer to these models as QKeras (Q) models. In addition, we train pruned versions of these models, targeting a sparsity of 50%. These are referred to as QKeras Pruned (QP) models.

Only convolutional layers, dense layers, and activation functions are quantized. The batch normalization layers are not quantized during training, as support for the QKERAS quantized equivalent of the KERAS batch normalization layer is not supported in `hls4ml` at the time of this writing. Support for this is planned for a future version of `hls4ml`. Batch normalization layers in the QAT models are therefore set to the default precision of $\langle 16, 6 \rangle$ by `hls4ml`. The final softmax layer is also kept at the default precision of $\langle 16, 6 \rangle$ in order to not compromise the classification accuracy.

Finally, we define a heterogeneously quantized model using AUTOQKERAS [16], a library for automatic heterogeneous quantization. The AUTOQKERAS library treats the layer precision as a hyperparameter, and finds the quantization which minimizes the model bit size while maximizing the model accuracy. By allowing AUTOQKERAS to explore different quantization settings for different parts of a given network, we obtain an optimal heterogeneously quantized QKERAS model. A Bayesian optimization is performed over a range of quantizers available in QKERAS, targeting a 50% reduction in model bit size. At the same time, the number of filters per convolutional layer and neurons per dense layer is re-optimized as quantization tends to lead to a preference for either (1) more filters as information is lost during quantization or (2) less filters due to some filters effectively being the same after quantization.

The optimization process is shown in figure 8, where the model bit size versus the model validation accuracy is shown for all the models tested in the automatic quantization procedure, showing the different quantization configurations for each of the convolutional layers. The size of the markers correspond to the number of filters used for a given convolutional layer in that trial. The colors correspond to different type of quantizers (binary, ternary or mantissa quantization using different bit widths). The model yielding the best accuracy versus size trade-off is marked by a red arrow. The number of filters per convolutional layer for the selected model is (4, 16, 12), compared to the original (16, 16, 24) for the BF and BP models, and the number of neurons per dense layer is (15, 16) compared to (42, 64) in the original model. Table 2 summarizes the quantization configuration found to be optimal by AUTOQKERAS, and the corresponding model energy consumption estimated using QTOOLS. We note that this model uses almost 90% less energy than the original. We train two versions of this model: an unpruned version (AQ), and a pruned version (AQP). The latter model is the same as AQ, but additionally pruned to a target sparsity of 50%.

Figure 9 shows the ROC curves for the AQ and AQP models. The curves show a slightly lower classification accuracy than those in figure 6, with AUCs differing by approximately 1%.

The numerical values spanned by the AQ model is shown in figure 10 for layer weights (left) and outputs (right). In contrast to those showed in figure 7, different bit widths are now used for the different layers, in correspondence with the bit width used in QKERAS.

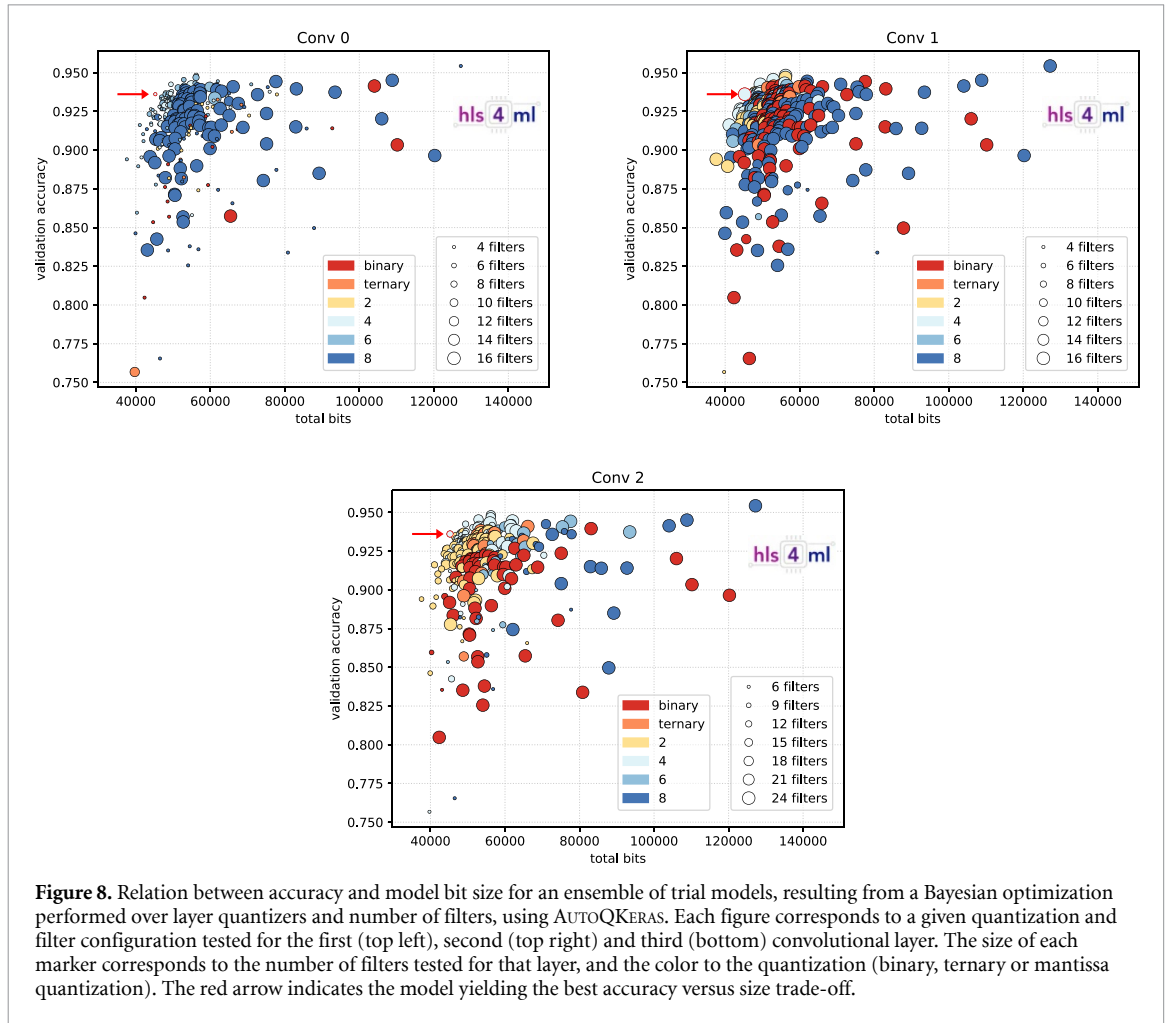


Table 2. Per-layer heterogeneous quantization configuration obtained with AUTOQKERAS, the total estimated energy consumption and model size in bits of the AutoQ (AQ) model. The energy is estimated assuming a 45 nm process using QTOOLS.

Model	Precision per layer											Energy (nJ)	Bit size	
	Conv2D	ReLU	Conv2D	ReLU	Conv2D	ReLU	Dense	ReLU	Dense	ReLU	Dense			Softmax
AQ	<4,0>	<3,1>	<4,0>	<3,1>	<4,0>	<8,4>	<4,0>	<4,2>	<4,0>	<8,2>	<6,0>	<16,6>	465	45 240

Figure 11 summarizes the effects of pruning and quantization. Here, we show the median accuracy and upper and lower quartiles across the ten folds of the unpruned (red) and pruned (green) quantized models, for different choices of bit widths and for the AQ (AQP) models. The unquantized baseline models are shown for reference (BF or BP). For bit widths above four, pruning to 50% sparsity has very little impact on the model accuracy. At very low bit widths, however, pruning negatively impacts the model performance. The accuracy is constant down to four bit precision, with marginal accuracy loss down to three bits. Using ternary quantization, the model accuracy drops to 87%–88% and has a higher statistical uncertainty. When quantizing down to binary precision, the model accuracy is reduced to 72% for the unpruned model and 64% for the pruned model. The significant reduction in accuracy due to pruning for binary networks is due to too little information being available in the network to accurately classify unseen data. A large spread in model accuracy for the binary network across the ten folds is observed, indicating that the model is less robust to fluctuations in the training dataset. As demonstrated in [8], this can be mitigated by increasing the model size (more filters and neurons per layer). The AQ models obtain a slightly lower accuracy than the baselines, but uses, as will be demonstrated in section 8, significantly fewer resources.

Due to the results above, it is recommended that users prune and quantize models using QAT through QKERAS, before proceeding with FPGA deployment with hls4ml.

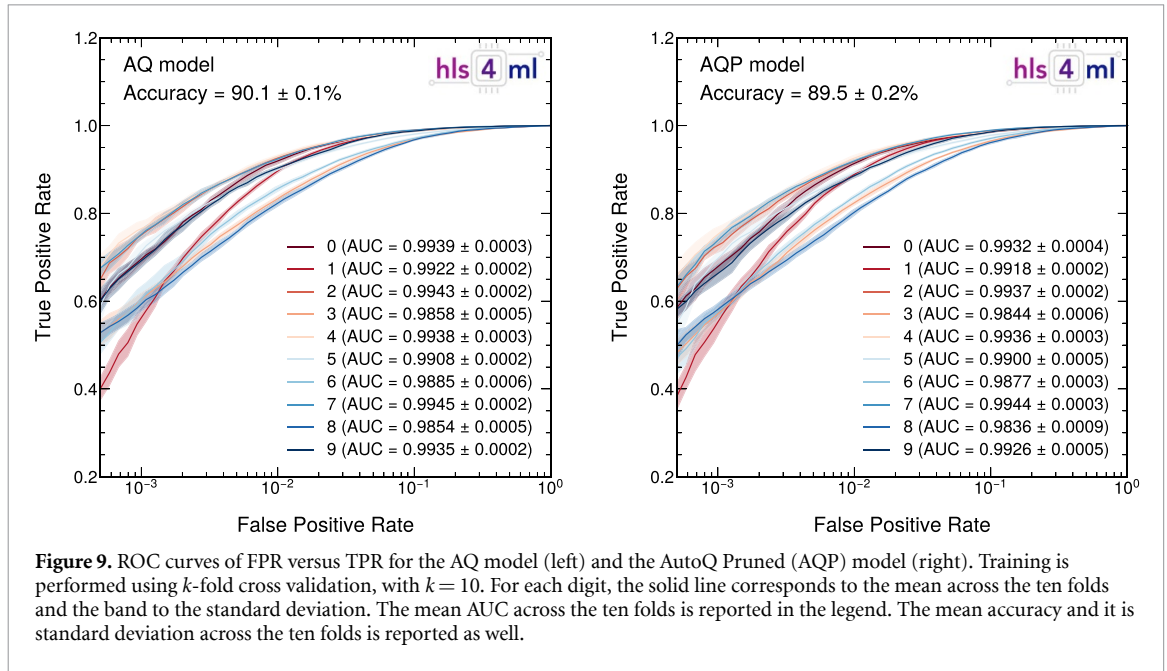


Figure 9. ROC curves of FPR versus TPR for the AQ model (left) and the AutoQ Pruned (AQP) model (right). Training is performed using k -fold cross validation, with $k = 10$. For each digit, the solid line corresponds to the mean across the ten folds and the band to the standard deviation. The mean AUC across the ten folds is reported in the legend. The mean accuracy and its standard deviation across the ten folds is reported as well.

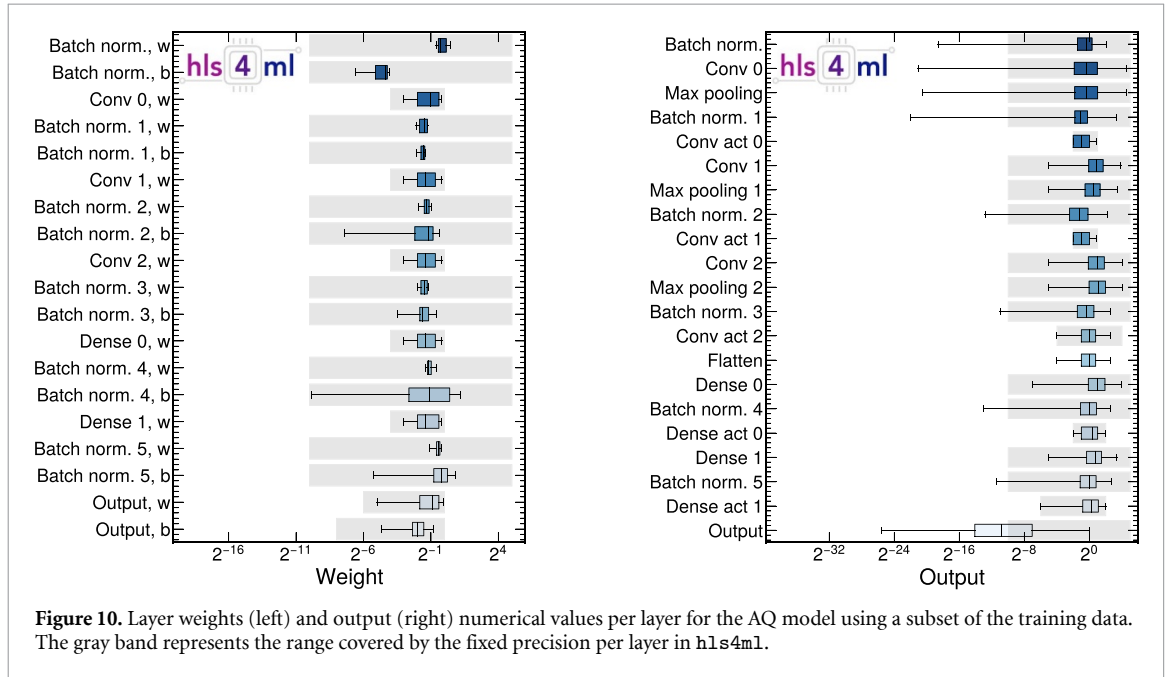


Figure 10. Layer weights (left) and output (right) numerical values per layer for the AQ model using a subset of the training data. The gray band represents the range covered by the fixed precision per layer in hls4ml.

8. FPGA porting

The models described above are translated into firmware using hls4ml version 0.5.0, and then synthesized with Vivado HLS 2020.1, targeting a Xilinx Virtex UltraScale+ VU9P (xcvu9pf1gb2104-2L) FPGA with a clock frequency of 200 MHz. For the QKERAS quantized models, the sign is not accounted for when setting the bit width per layer during QAT, so layers quantized with total bit width b in QKERAS are therefore implemented as fixed-point numbers with total bit width $b + 1$ in hls4ml. We compare the model accuracy, latency, and on-chip resource consumption. The accuracy after translating the model into C/C++ code with hls4ml (solid line) for the different models, is shown in figure 12 and compared to the accuracy evaluated using KERAS. No pre-synthesis results are shown for the BF and BP models, as these are quantized *during* synthesis. Nearly perfect agreement in evaluated accuracy before and after synthesis is observed for the Q and QP models and the translation into fixed-point precision is lossless.

While the accuracy of the Q and QP models trained via QAT remains high down to a bit width of three, the accuracy of the PTQ models fall off sharply with decreasing bit width and have almost no discrimination power for bit widths smaller than 14. PTQ has a higher negative impact on the unpruned models, indicating

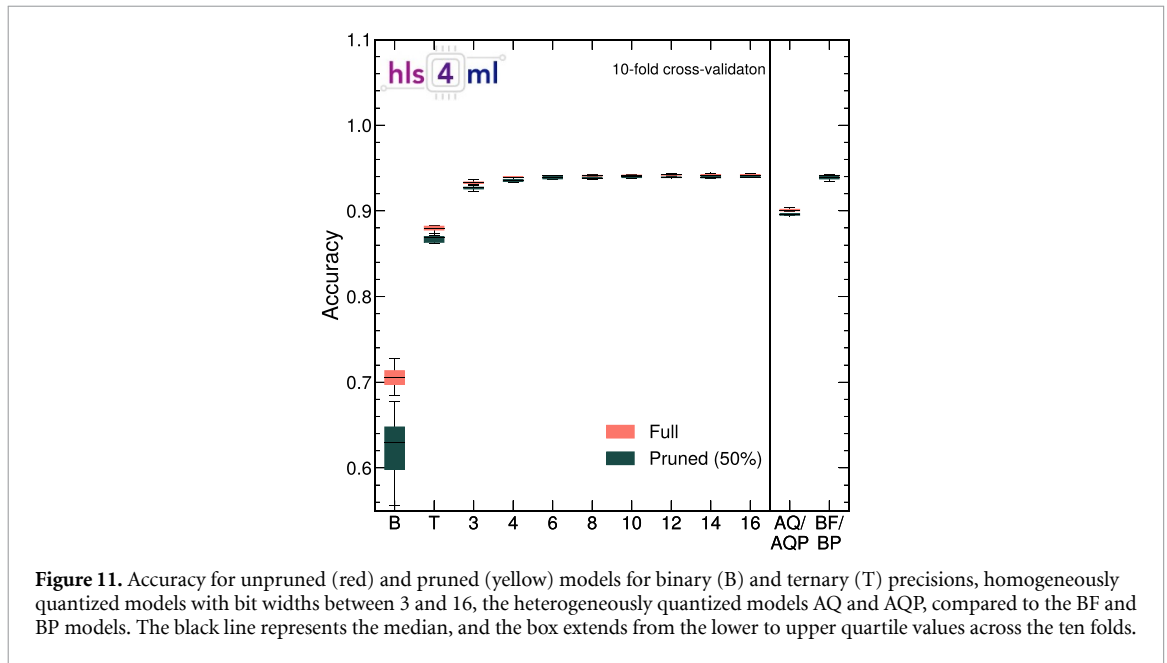


Figure 11. Accuracy for unpruned (red) and pruned (yellow) models for binary (B) and ternary (T) precisions, homogeneously quantized models with bit widths between 3 and 16, the heterogeneously quantized models AQ and AQP, compared to the BF and BP models. The black line represents the median, and the box extends from the lower to upper quartile values across the ten folds.

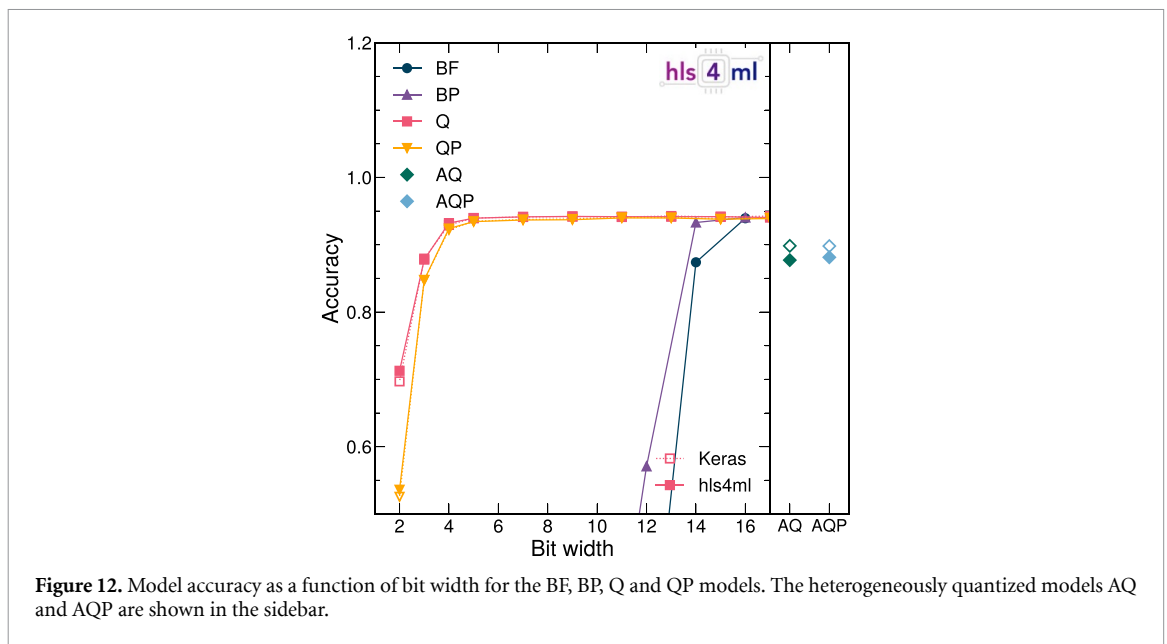


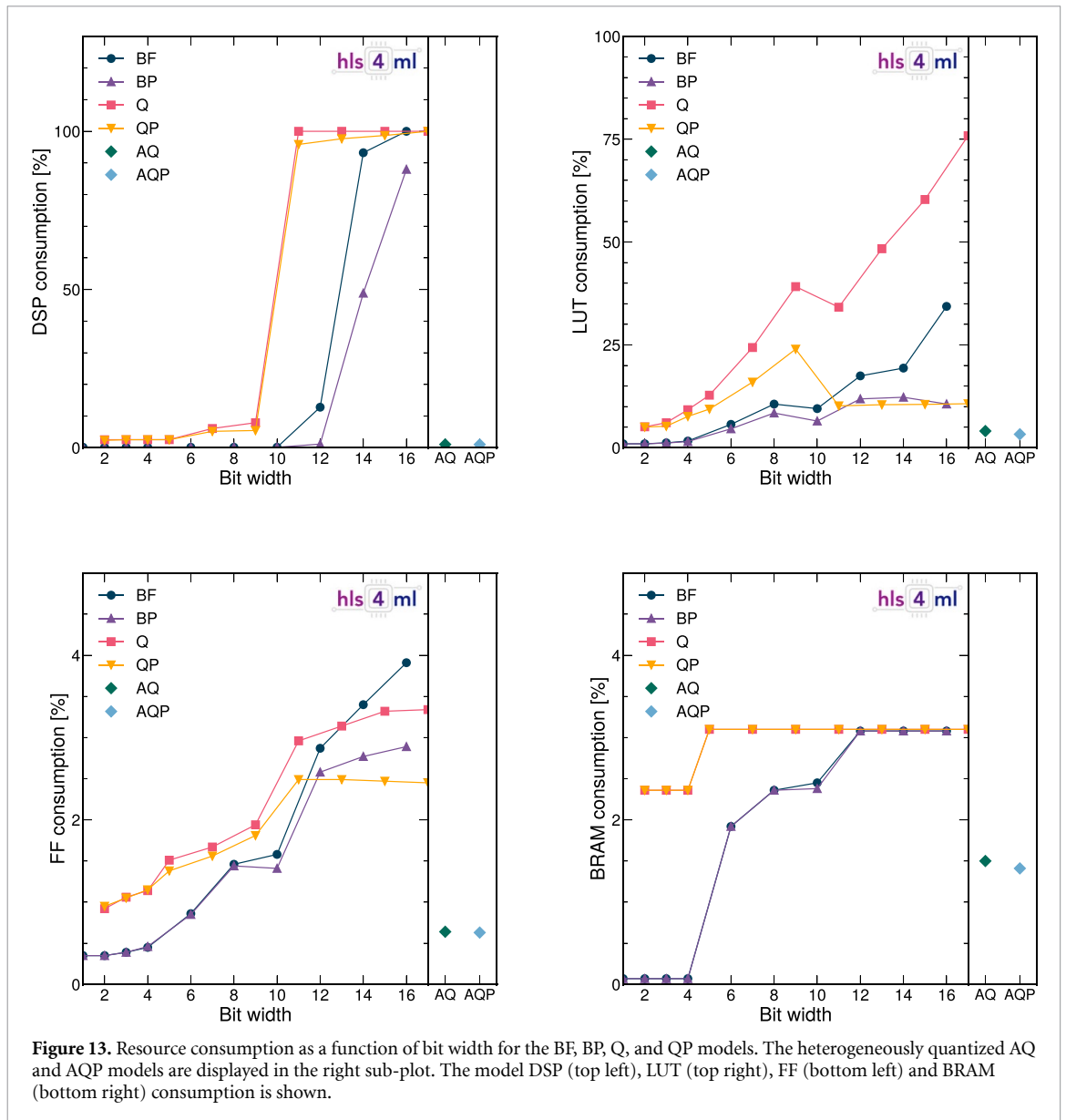
Figure 12. Model accuracy as a function of bit width for the BF, BP, Q and QP models. The heterogeneously quantized models AQ and AQP are shown in the sidebar.

that rounding errors are the biggest cause for accuracy degradation. The heterogeneously quantized models AQ and AQP have slightly lower accuracy than the baseline (16, 6) model.

We then study the resource consumption and latency of the different models after logic-synthesis. The resources available on the FPGA are digital signal processors (DSPs), LUTs, BRAMs, and flip-flops (FFs). In figure 13, the resource consumption relative to the total available resources is shown. Here, a fully parallel implementation is used where each multiplier is used exactly once, which can be achieved by setting the *reuse factor* R [1] to 1 for each layer in hls4ml.

The DSP consumption is slightly higher for the Q and QP models than the BF and BP models due to the batch normalization layers in the QAT models being fixed to (16, 6).

Below a bit width of 10, the DSP consumption is significantly reduced as multiplications are performed using LUTs. DSPs are usually the limiting resource for FPGA inference, and we observe that through QAT, the DSP consumption can be reduced from one hundred percent down to a few percent with no loss in model accuracy (as demonstrated in figure 12). Above a bit width of 10, almost all the DSPs on the device are



in use for the Q and QP models. This routing is a choice of Vivado HLS during optimization of the circuit layout. This is also the reason why pruning appears to have relatively little impact for these models: the DSPs are maximally used and the remaining multiplications are performed with LUTs. The QP models use significantly fewer LUT resources than the unpruned equivalent. The point where most multiplications are moved from DSPs to LUTs is marked by a steep drop in DSP consumption starting at a bit width of 10.

The heterogeneously quantized models, AQ and AQP, consume very little FPGA resources, comparable to that of the Q and QP models quantized to a bit width of three. All models use very few FFs, below 4% of the total budget. The BRAM consumption is also small and below 4% for all models. Some dependence on bit width can be traced back to how operations are mapped to the appropriate resources through internal optimizations in HLS. Depending on the length and the bit width of the FIFO buffers used for the convolutional layer sliding window, HLS will decide whether to place the operation on BRAMs or LUTs and migration between the two is expected. Most of the BRAMs, are spent on *channels*, the output of different layers.

The latency and II for all models is shown in figure 14. A total latency of about $5 \mu\text{s}$ is observed for all models, similar to the II. The latency is independent of bit width when running at a fixed clock period. We leave it for future studies to explore running the board at higher clock frequencies.

A summary of the accuracy, resource consumption and latency for the BF and BP models quantized to a bit width of 14, the Q and QP models quantized to a bit width of 7 and the heterogeneously quantized AQ

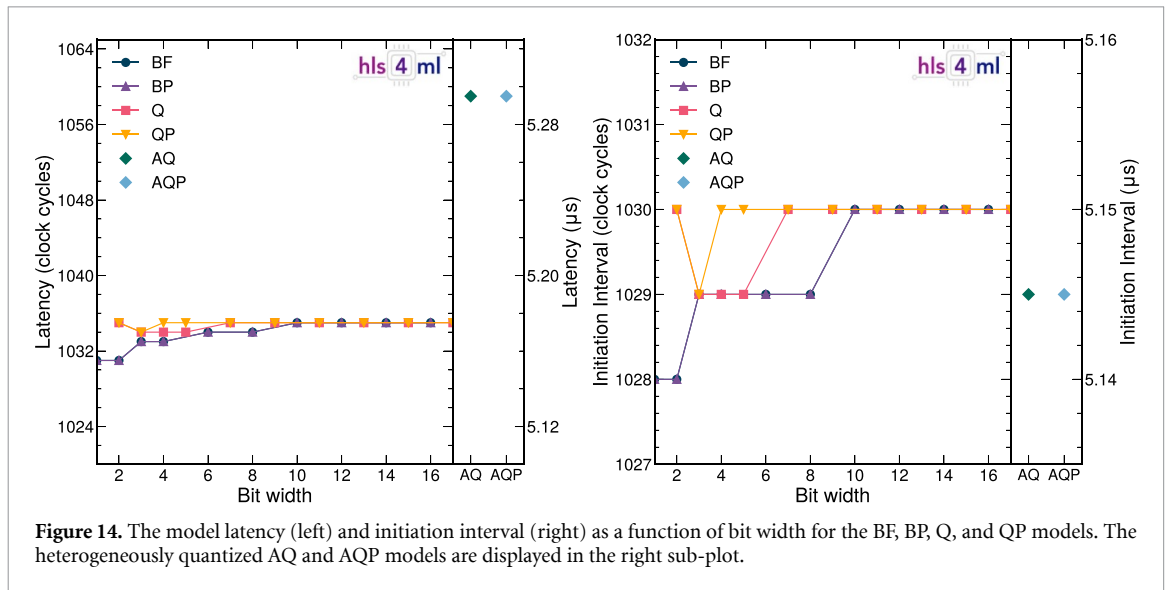


Figure 14. The model latency (left) and initiation interval (right) as a function of bit width for the BF, BP, Q, and QP models. The heterogeneously quantized AQ and AQP models are displayed in the right sub-plot.

Table 3. Accuracy, resource consumption and latency for the BF and BP models quantized to a bit width of 14, the Q and QP models quantized to a bit width of 7 and the heterogeneously quantized AQ and AQP models. The numbers in parentheses correspond to the total amount of resources used.

FPGA: Xilinx Virtex UltraScale+ VU9P							
Model	Accuracy	DSP (%)	LUT (%)	FF (%)	BRAM (%)	Latency (cc)	II (cc)
BF 14-bit	0.87	93.23 (6377)	19.36 (228 823)	3.40 (80 278)	3.08 (66.5)	1035 (5.2 μ s)	1030
BP 14-bit	0.93	48.85 (3341)	12.27 (145 089)	2.77 (65 482)	3.08 (66.5)	1035 (5.2 μ s)	1030
Q 7-bit	0.94	2.56 (175)	12.77 (150 981)	1.51 (35 628)	3.10 (67.0)	1034 (5.2 μ s)	1029
QP 7-bit	0.94	2.54 (174)	9.40 (111 152)	1.38 (32 554)	3.10 (67.0)	1035 (5.2 μ s)	1030
AQ	0.88	1.05 (72)	4.06 (48 027)	0.64 (15 242)	1.5 (32.5)	1059 (5.3 μ s)	1029
AQP	0.88	1.02 (70)	3.28 (38 795)	0.63 (14 802)	1.4 (30.5)	1059 (5.3 μ s)	1029

and AQP models, is shown in table 3. Resource utilization is quoted as a fraction of the total available resources on the FPGA, and the absolute number of resources used is quoted in parenthesis. The accuracy of the post-training quantized BF and BP models drops below 50% for bit widths narrower than 14 and can not be used for inference. The QAT models, Q and QP, quantized to a bit width of 7 maintain a high accuracy despite using only a fraction of the available FPGA resources. The models using the fewest resources are the AQ and AQP heterogeneously quantized models, reducing the DSP consumption by 99% while maintaining a relatively high accuracy. Finding the best trade-off between model size and accuracy in an application-specific way can be done using AUTOQKERAS, as demonstrated in section 7.

To further reduce the resource consumption, the reuse factor R can be increased. This comes at the cost of higher latency. The model latency and resource consumption as a function of bit width and for different reuse factors for the QP models are shown in figure 15. The latency and II increase with R , while the DSP consumption goes down. The LUT consumption is minimally affected by the reuse factor, consistent with the results reported in [1]. The BRAM consumption is the same for all reuse factors, around 3%, and therefore not plotted. The corresponding study for the BF, BP and Q models can be found in appendix.

A summary of the latency and resource consumption for different reuse factors for all the models at a fixed bit width of 16 is shown in figure 16. The latency has a linear dependence on the reuse factor, as expected because each multiplier is used in series one reuse factor at the time. The DSP consumption decreases as $\sim 1/R$ for all models. The first point deviates from this as the maximum number of DSPs are in use, effectively reaching a plateau. The LUT consumption is high for a reuse factor of one, complimenting the ceiling reached in DSP consumption at a reuse of one, since the multiplications that do not fit on DSP are moved to LUTs. The FF consumption is flat as a function of reuse factor. The BRAM consumption does not depend on the reuse factor and is the same for all models, around 3%. We leave it up to hls4ml users to find the optimal trade-off between inference latency and resource consumption for a given application through tuning of the reuse factor.

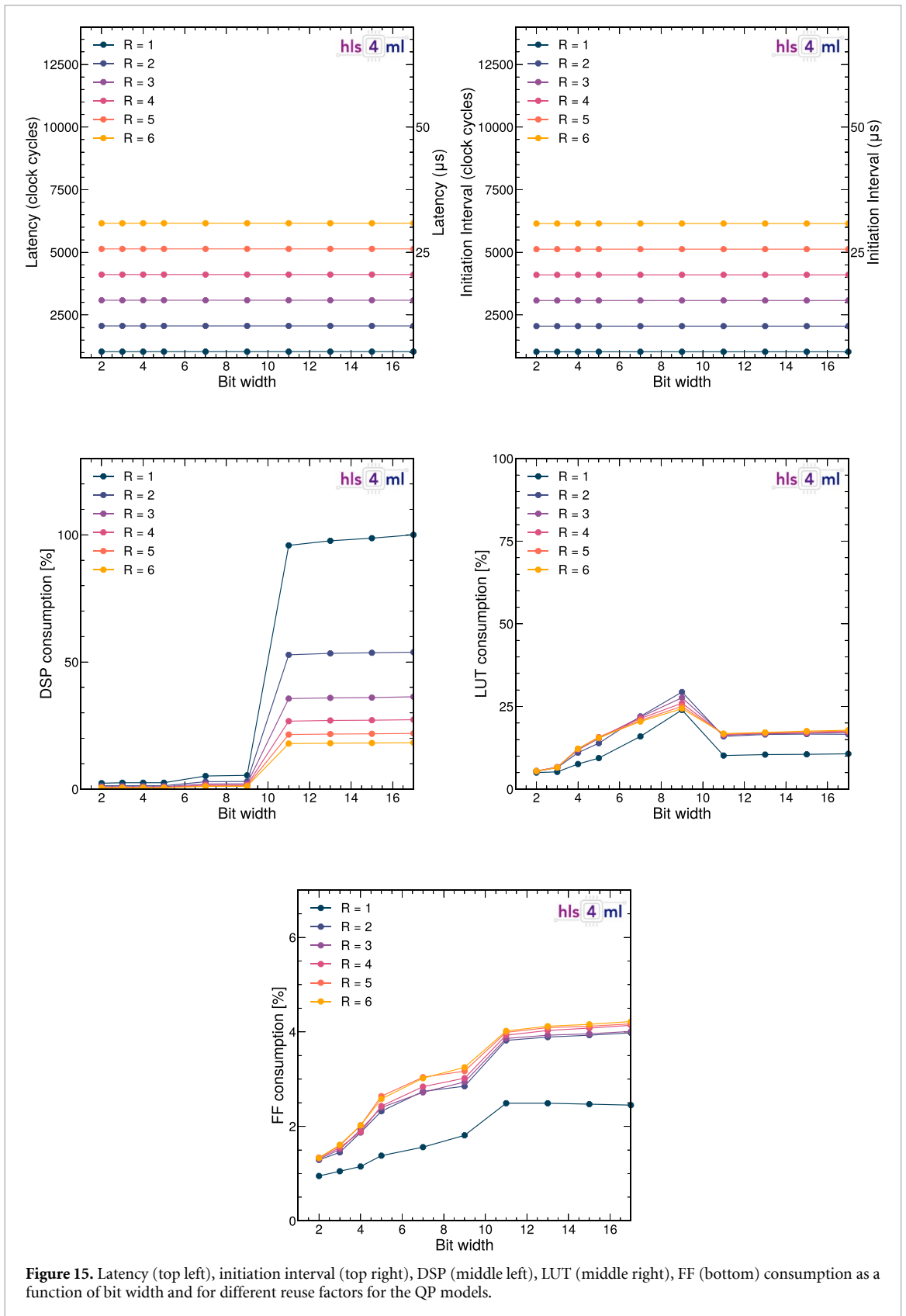


Figure 15. Latency (top left), initiation interval (top right), DSP (middle left), LUT (middle right), FF (bottom) consumption as a function of bit width and for different reuse factors for the QP models.

Although particle physics experiments mostly use large FPGAs, the `hls_4_ml` library can be readily used for smaller FPGAs, like those found on SoC or internet-of-things (IoT) devices, through increasing the reuse factor. To demonstrate this, we synthesize and deploy the smallest model that retains the original model accuracy, QP 7-bit, onto a low-cost TUL PYNQ-Z2 development board, equipped with a Xilinx Zynq

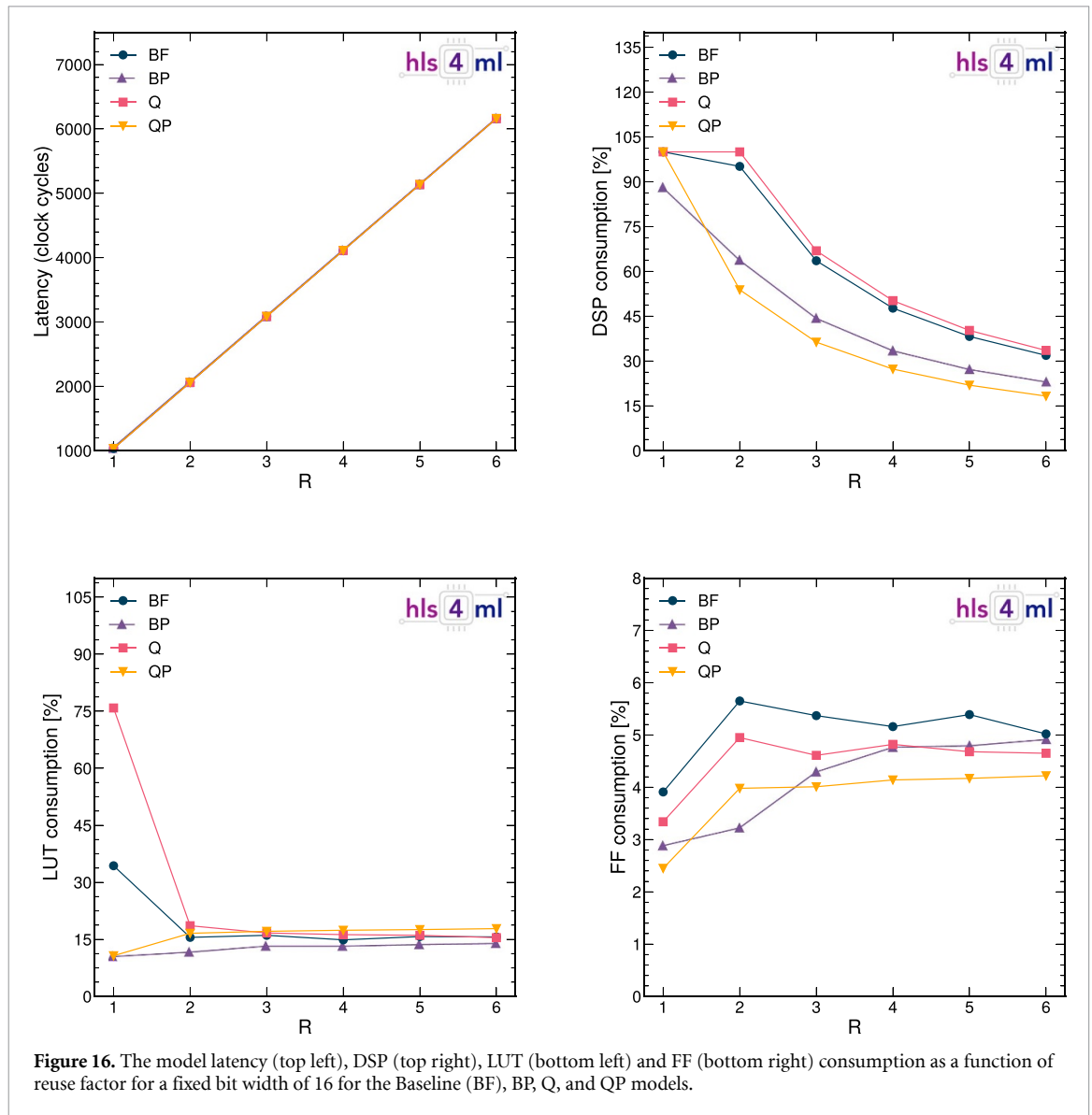


Figure 16. The model latency (top left), DSP (top right), LUT (bottom left) and FF (bottom right) consumption as a function of reuse factor for a fixed bit width of 16 for the Baseline (BF), BP, Q, and QP models.

Table 4. Resource consumption and latency for the QP 7-bit model on a Xilinx Zynq XC7Z020 SoC. A clock frequency of 100 MHz is used.

FPGA: Xilinx Zynq XC7Z020 SoC							
	DSP	LUT	FF	BRAM (18 kb)	Latency (cc)	II (cc)	frame/s
Available	220	53 200	106 400	280	—	—	—
Used	213 (96.82%)	48 259 (90.71%)	35 118 (33.01%)	122 (43.57%)	17 085 (171 μ s)	16 385	2831

XC7Z020 SoC (FPGA part number xc7z020c1g400-1). This FPGA is significantly smaller than the Xilinx Virtex UltraScale+ VU9P, and consists of 13 300 logic slices, each with four 6-input LUTs and 8 FFs, 630 kB of BRAM, and 220 DSP slices. As expected, a large reuse factor is needed in order to fit the QP 7-bit model onto the Zynq XC7Z020. For a clock frequency of 100 MHz, the resulting inference latency is 171 μ s and up to 2831 image classifications per second. This implementation uses a total of 91% of the LUTs, 97% of the DSPs, 33% of the FFs, and 44% of the BRAM. A summary is provided in table 4. This demonstrates the flexibility of hls4ml to accommodate SoC/IoT use cases, which can demand smaller FPGAs and tolerate millisecond latencies.

Finally, in figure 17 we study the resource consumption and latency as a function of the input size for a single convolutional layer with varying number of filters and kernel sizes. Three input channels are always assumed and the input height (H) and width (W) is varied between 10 and 256, such that the input size is

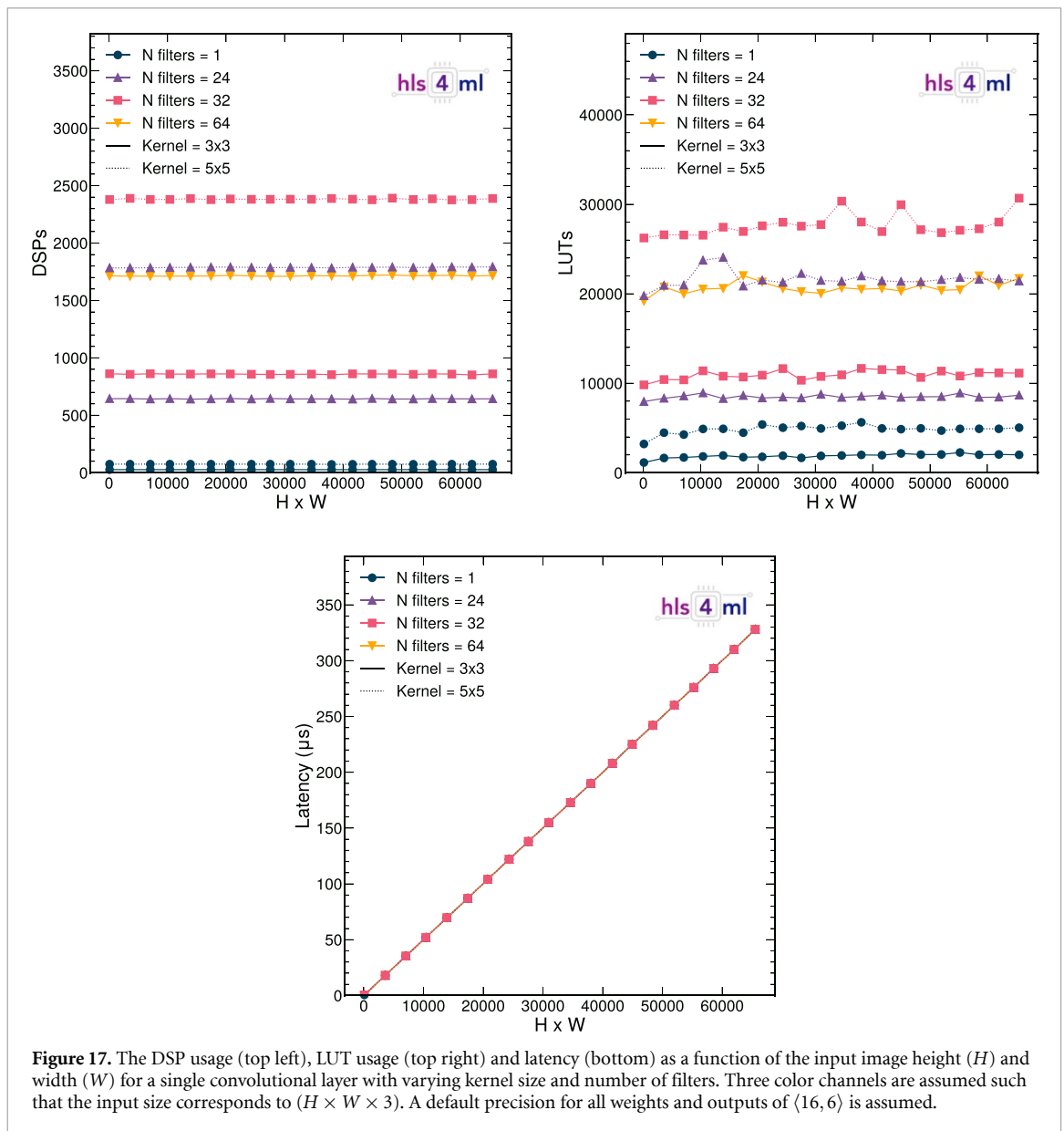
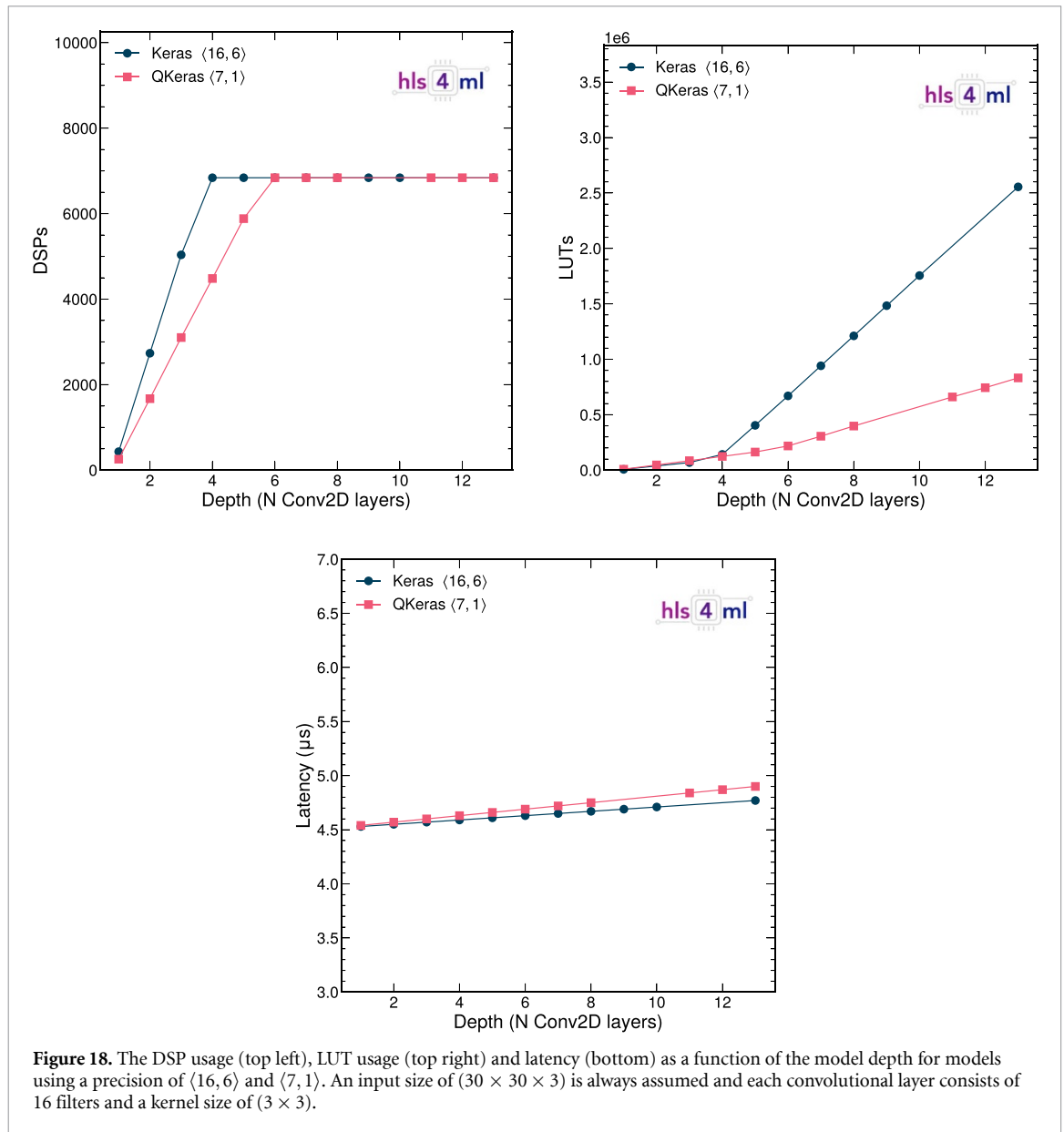


Figure 17. The DSP usage (top left), LUT usage (top right) and latency (bottom) as a function of the input image height (H) and width (W) for a single convolutional layer with varying kernel size and number of filters. Three color channels are assumed such that the input size corresponds to $(H \times W \times 3)$. A default precision for all weights and outputs of $\langle 16, 6 \rangle$ is assumed.

$H \times W \times 3$. A precision of $\langle 16, 6 \rangle$ is assumed for all models to illustrate the dependency of latency/resources on the given layer configurations, although, as we have demonstrated above, resources can be significantly reduced using QAT. The DSP and LUT consumption is constant as a function of the input size, but increases with the number of filters used and the kernel size, due to the higher number of multiplications that need to be performed simultaneously. The latency increases linearly with the input size, but does not depend on the kernel size or the number of filters. We also show the latency as a function of the depth of the model in figure 18. For simplicity, we assume an input size of $30 \times 30 \times 3$, 16 filters and a kernel size of 3×3 for each convolutional layer. The precision is fixed to $\langle 16, 6 \rangle$ or $\langle 7, 1 \rangle$. The DSP consumption scales linearly with the model depth until the maximum number of DSPs are used. When all DSPs are in use, multiplications are moved onto LUTs, seen as a change of slope in the LUT consumption versus model depth. The inference latency increases linearly with the model depth.

Figures 17 and 18 summarize how input size and model architecture affects the inference latency and resource consumption. Through increasing the reuse factor, smaller FPGAs can be targeted through a trade-off between latency and resource consumption. Support for QAT through and pruning further reduce the model footprint. The hls4ml library is therefore capable of providing generic, multi-backend support for a wide range of hardware and latency constraints.



9. Conclusions

We have presented the extension of hls4ml to support CNN architectures for transpilation to FPGA designs, through a stream-based implementation of convolutional and pooling layers. A fully on-chip design is used in order to provide for microsecond latency applications, like those at the CERN LHC. Taking as a benchmark example a CNN classifier trained on the SVHN, we show how compression techniques at training time (pruning and QAT) reduce the resource utilization of the FPGA-converted models, while retaining to a large extent the floating-point precision baseline accuracy. Once converted to FPGA firmware using hls4ml, these models can be executed with 5 μs latency and a comparable initiation interval, while consuming less than 10% of the FPGA resources. We demonstrate the flexibility and scalability of hls4ml to accommodate CNN architectures of varying sizes, and offer solutions both for small SoC FPGAs and for the larger FPGAs used in particle physics experiments. This work enables domain ML specialists to design hardware-optimized ML algorithms for low-latency, low-power, or radiation-hard systems, for instance particle physics trigger systems, autonomous vehicles, or inference accelerators for space applications.

Data availability statement

The data that support the findings of this study are openly available.

Acknowledgment

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. M P, S S and V L are supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant Agreement No. 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 (DOE), Office of Science, Office of High Energy Physics. P H is supported by a Massachusetts Institute of Technology University grant. Z W is supported by the National Science Foundation under Grant Nos. 1606321 and 115164. J D is supported by the DOE, Office of Science, Office of High Energy Physics Early Career Research program under Award No. DE-SC0021187.

Code availability statement

The `hls4ml` library is available at <https://github.com/fastmachinelearning/hls4ml> and archived in the Zenodo platform at [10.5281/zenodo.4161550](https://zenodo.org/record/4161550). The work presented here is based on the Bartsia release, version 0.5.0. For examples on how to use `hls4ml`, the notebooks in <https://github.com/fastmachinelearning/hls4ml-tutorial> serve as a general introduction. The QKERAS library, which also includes AUTOQKERAS and QTOOLS, is available at <https://github.com/google/qkeras>.

The SVHN dataset [17] can be downloaded at <http://ufldl.stanford.edu/housenumbers> or through TENSORFLOW Datasets at www.tensorflow.org/datasets/catalog/svhn_cropped.

Appendix. Performance versus bit width and reuse factor

Figures 19–21 show the model latency, initiation interval, DSP, LUT and FF consumption as a function of bit width and for different reuse factors for the BF, BP and Q models, respectively. A similar behavior is observed for all models, where the latency roughly scales with one unit of reuse factor and the DSP consumption scales as the inverse of the reuse factor. The BRAM consumption does not depend on the reuse factor.

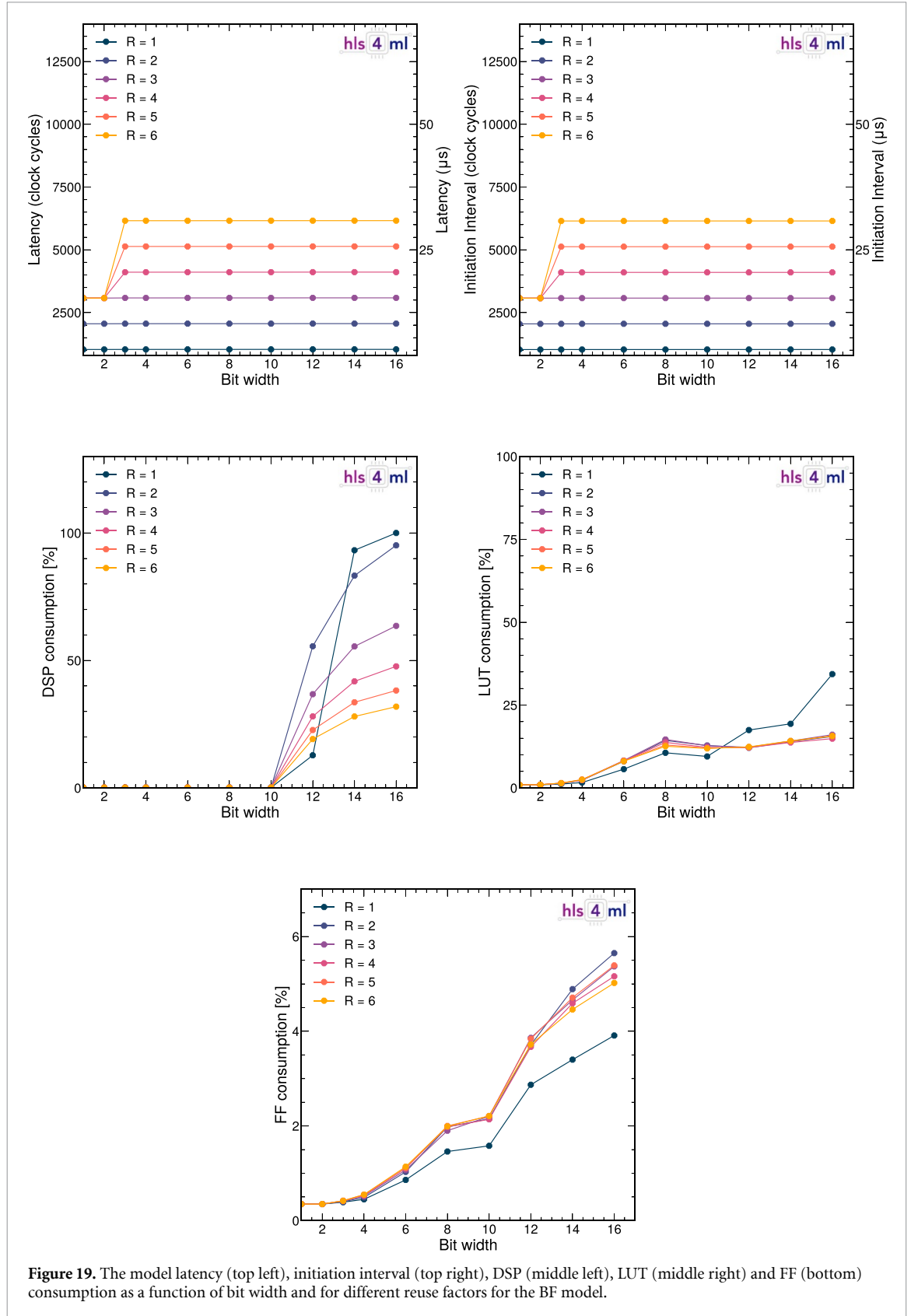


Figure 19. The model latency (top left), initiation interval (top right), DSP (middle left), LUT (middle right) and FF (bottom) consumption as a function of bit width and for different reuse factors for the BF model.

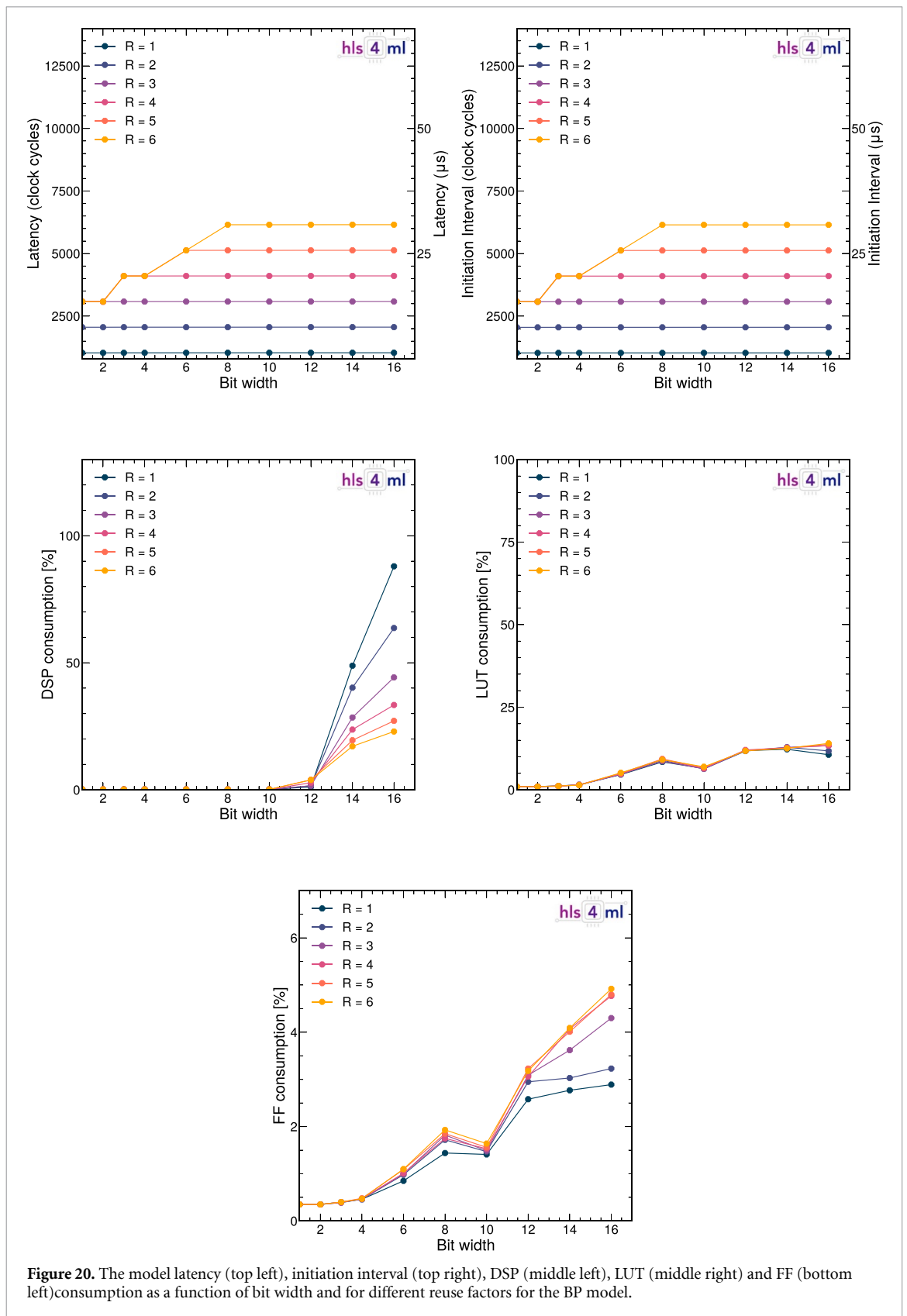


Figure 20. The model latency (top left), initiation interval (top right), DSP (middle left), LUT (middle right) and FF (bottom left) consumption as a function of bit width and for different reuse factors for the BP model.

For the BF models in figure 19, there is an unexpected drop in DSP consumption at a bit width of 12 for models using a reuse factor of one. For this model, only 13% of the DSPs (corresponding to 876 units) are used, whereas the same mode with a reuse factor of six uses 19% of the available DSPs (corresponding to a total of 1311). We would expect models with higher reuse factors to use fewer resources and not vice versa. This unexpected behavior is only observed for one data point. We have investigated things to the extent possible, but can only map the results back to how resources are allocated within HLS.

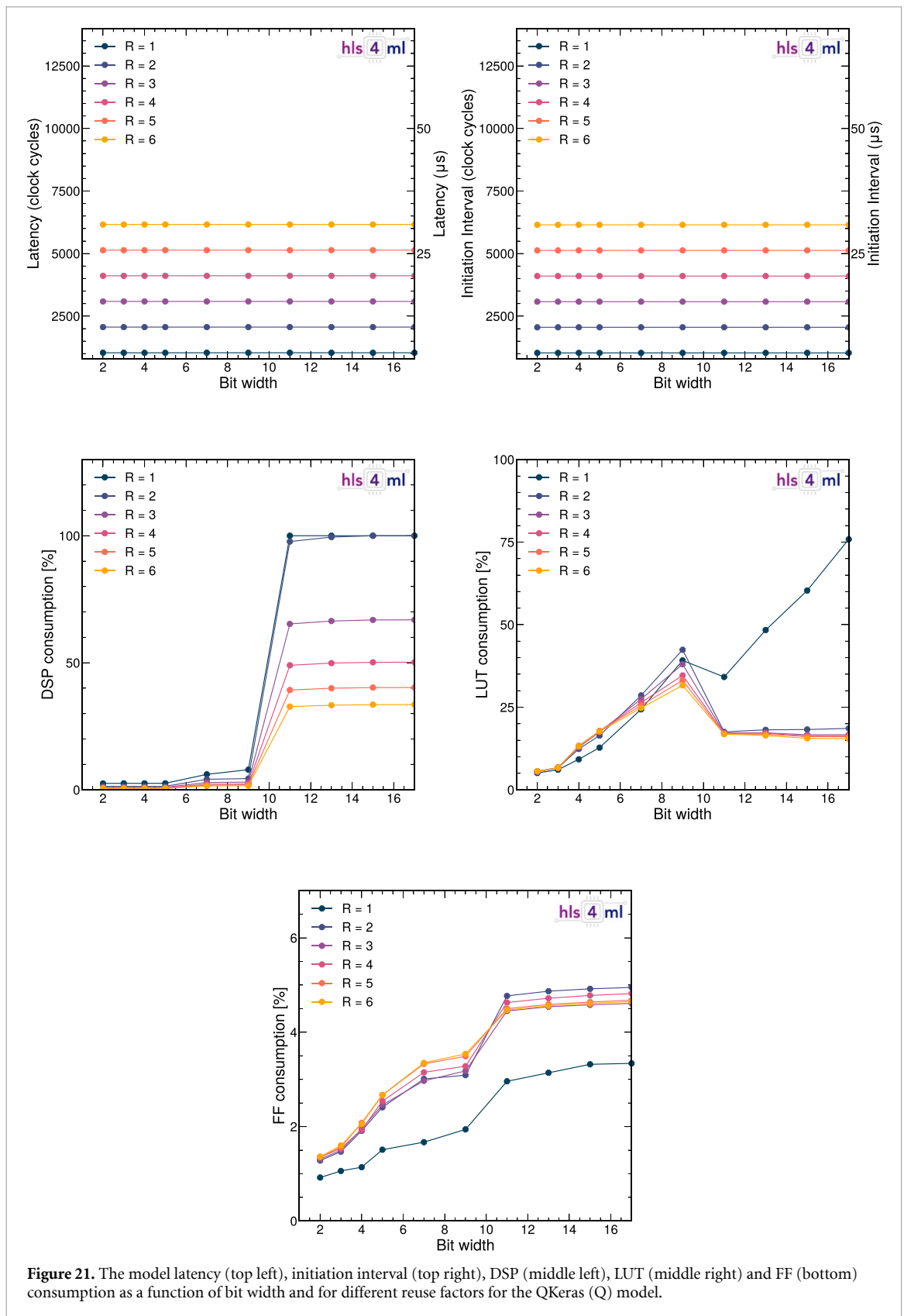


Figure 21. The model latency (top left), initiation interval (top right), DSP (middle left), LUT (middle right) and FF (bottom) consumption as a function of bit width and for different reuse factors for the QKeras (Q) model.

For the Q models in figure 21, the DSP consumption (middle left) of the models using a reuse factor of one and those using a reuse factor of two overlap above a bit width of ten. The reason for this is that the maximum number of DSPs are reached for both model types, and multiplications are therefore forced to other resources. This effect can be seen in the LUT consumption (middle right), where the model using a reuse factor of 1 uses significantly more LUTs than the other models.

ORCID iDs

Thea Aarrestad  <https://orcid.org/0000-0002-7671-243X>

Jennifer Ngadiuba  <https://orcid.org/0000-0002-0055-2935>

Javier Duarte  <https://orcid.org/0000-0002-5076-7096>

Philip Harris  <https://orcid.org/0000-0001-8189-3741>

References

- [1] Duarte J et al 2018 Fast inference of deep neural networks in FPGAs for particle physics *J. Instrum.* **13** 07027
- [2] Loncar V et al 2020 fastmachinelearning/hls4ml: aster (available at: <https://github.com/fastmachinelearning/hls4ml>)
- [3] ATLAS Collaboration 2020 Operation of the ATLAS trigger system in Run 2 *J. Instrum.* **15** 10004
- [4] ATLAS Collaboration 2017 Technical design report for the Phase-II upgrade of the ATLAS TDAQ system *ATLAS Technical Design Report* CERN-LHCC-2017-020 ATLAS-TDR-029
- [5] CMS Collaboration 2020 Performance of the CMS Level-1 trigger in proton-proton collisions at $\sqrt{s} = 13$ TeV *J. Instrum.* **15** 10017
- [6] CMS Collaboration 2020 The phase-2 upgrade of the CMS level-1 trigger *CMS Technical Design Report* CERN-LHCC-2020-004 CMS-TDR-021
- [7] Xilinx 2020 Vivado design suite user guide: high-level synthesis (available at: www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf)
- [8] Ngadiuba J et al Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml (arXiv:2003.06308)
- [9] Summers S et al 2020 Fast inference of boosted decision trees in FPGAs for particle physics *J. Instrum.* **15** 05026
- [10] Iiyama Y et al 2021 Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction in high energy physics *Front. Big Data* **3** 44
- [11] Heintz A et al 2020 Accelerated charged particle tracking with graph neural networks on FPGAs *3rd Machine Learning and the Physical Sciences Workshop at the 34th Conf. on Neural Information Processing Systems* vol 12
- [12] Abadi M et al TensorFlow: large-scale machine learning on heterogeneous systems 2015 (arXiv:1603.04467) (Software available from tensorflow.org)
- [13] Chollet F et al 2015 Keras (available at: <https://keras.io>)
- [14] Paszke A et al 2019 PyTorch: an imperative style, high-performance deep learning library *Advances in Neural Information Processing Systems* 32, ed H Wallach (Curran Associates, Inc.) p 8024
- [15] Open Neural Network Exchange Collaboration 2017 ONNX (available at: <https://onnx.ai/>)
- [16] Coelho Jr C N et al 2021 Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders *Nat. Mach. Intell* accepted (<https://doi.org/10.1038/s42256-021-00356-5>)
- [17] Netzer Y et al 2011 Reading digits in natural images with unsupervised feature learning *NIPS 2011 Workshop on Deep Learning and Unsupervised Feature Learning*
- [18] Boser T, Calafiura P and Johnson I 2017 Convolutional neural networks for track reconstruction on FPGAs *NIPS 2017 Demonstrations*
- [19] Venieris S I, Kouris A and Bouganis C-S 2018 Toolflows for mapping convolutional neural networks on FPGAs: a survey and future directions *ACM Comput. Surv.* **51** 1–39
- [20] Guo K et al 2018 A survey of FPGA-based neural network inference accelerators *ACM Trans. Reconfigurable Technol. Syst.* **12** 1–26
- [21] Shawahna A, Sait S M and El-Maleh A 2019 FPGA-based accelerators of deep learning networks for learning and classification: a review *IEEE Access* **7** 7823–59
- [22] Abdelouahab K, Pelcat M, Serot J and Berry F 2018 Accelerating CNN inference on FPGAs: a survey (arXiv:1806.01683)
- [23] Umuroglu Y et al 2017 FINN: A framework for fast, scalable binarized neural network inference *Proc. 2017 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays* (ACM Press)
- [24] Blott M et al 2018 FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks *ACM Trans. Reconfigurable Technol. Syst.* **11** 1–23
- [25] Alessandro, Franco G, Fraser N, Umuroglu Y and vfdv 2021 Xilinx/brevitas: Release version 0.4.0 (<https://doi.org/10.5281/zenodo.4606672>)
- [26] Venieris S I and Bouganis C S 2017 fpgaConvNet: a toolflow for mapping diverse convolutional neural networks on embedded FPGAs *NIPS 2017 Workshop on Machine Learning on the Phone and Other Consumer Devices*
- [27] Venieris S I and Bouganis C S 2017 fpgaConvNet: automated mapping of convolutional neural networks on FPGAs *Proc. 2017 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays* (ACM) p 291
- [28] Venieris S I and Bouganis C S 2017 Latency-driven design for FPGA-based convolutional neural networks *2017 27th Int. Conf. on Field Programmable Logic and Applications (FPL)* p 1
- [29] Venieris S I and Bouganis C S 2016 fpgaConvNet: a framework for mapping convolutional neural networks on FPGAs *2016 IEEE 24th Annual Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)* (IEEE) p 40
- [30] Jia Y et al 2014 Caffe: convolutional architecture for fast feature embedding *Proc. 22nd ACM Int. Conf. on Multimedia, MM '14* (New York: ACM) p 675
- [31] Guan Y et al 2017 FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates *2017 IEEE 25th Annual Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)* p 152
- [32] Sharma H et al 2016 From high-level deep neural models to FPGAs *2016 49th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO)* (IEEE) p 1
- [33] DiCecco R et al 2016 Caffeinated FPGAs: FPGA framework for convolutional neural networks *2016 Int. Conf. on Field-Programmable Technology (FPT)* pp 265–8
- [34] Gokhale V, Zaidy A, Chang A X M and Culurciello E 2017 Snowflake: a model agnostic accelerator for deep convolutional neural networks (arXiv:1708.02579)
- [35] Collobert R, Kavukcuoglu K and Farabet C 2011 Torch7: a Matlab-like environment for machine learning *BigLearn, NIPS Workshop*
- [36] Majumder K and Bondhugula U 2019 A flexible FPGA accelerator for convolutional neural networks (arXiv:1912.07284)
- [37] Aimar A et al 2019 Nullhop: a flexible convolutional neural network accelerator based on sparse representations of feature maps *IEEE Trans. Neural Networks Learn. Syst.* **30** 644–56

- [38] Rahman A, Lee J and Choi K 2016 Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array *2016 Design, Automation Test in Conf. Exhibition (DATE)* p 1393
- [39] Xilinx Vitis AI 2020 (available at: <https://github.com/Xilinx/Vitis-AI>)
- [40] Vasudevan A, Anderson A and Gregg D 2017 Parallel multi channel convolution using general matrix multiplication *2017 IEEE 28th Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP)* pp 19–24
- [41] LeCun Y and Cortes C 2010 MNIST handwritten digit database (available at: <http://yann.lecun.com/exdb/mnist/>)
- [42] Cubuk E D, Zoph B, Mane D, Vasudevan V and Le Q 2019 Autoaugment: learning augmentation policies from data *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR) (Long Beach, CA, USA)* pp 113–23
- [43] DeVries T and Taylor G W 2017 Improved regularization of convolutional neural networks with cutout (arXiv:1708.04552)
- [44] Liang S, Khoo Y and Yang H 2021 Drop-activation: implicit parameter reduction and harmonic regularization *Commun. Appl. Math. Comput.* **3** 293–311
- [45] Zagoruyko S and Komodakis N 2017 Wide residual networks (arXiv:1605.07146)
- [46] Zhang K et al 2018 Residual networks of residual networks: multilevel residual networks *IEEE Trans. on Circuits and Systems for Video Technology* **28** 1303–14
- [47] Sermanet P, Chintala S and LeCun Y 2012 Convolutional neural networks applied to house numbers digit classification (arXiv:1204.3968)
- [48] O'Malley T et al 2019 Keras Tuner (<https://github.com/keras-team/keras-tuner>)
- [49] Ioffe S and Szegedy C 2015 Batch normalization: accelerating deep network training by reducing internal covariate shift *32nd Int. Conf. on Machine Learning (Lille, France)* vol 37, ed F Bach and D Blei (PMLR) p 448
- [50] Nair V and Hinton G E 2010 Rectified linear units improve restricted Boltzmann machines *27th Int. Conf. on Machine Learning (ICML) (Madison, WI: Omnipress)* p 807
- [51] Glorot X, Bordes A and Bengio Y 2011 Deep sparse rectifier neural networks *14th Int. Conf. on Artificial Intelligence and Statistics (AISTATS) (Fort Lauderdale, FL, USA)* vol 15, ed G Gordon, D Dunson and M Dudík (JMLR) p 315
- [52] Horowitz M 2014 1.1 computing's energy problem (and what we can do about it) *Digest of Technical Papers—IEEE Int. Solid-State Conf.* vol 57 10–14
- [53] Goodfellow I, Bengio Y and Courville A 2016 *Deep Learning* (Cambridge, MA: MIT Press)
- [54] Kingma D P and Ba J 2015 Adam: A method for stochastic optimization *3rd Int. Conf. on Learning Representations (ICLR) 2015, Conf. Track Proc*
- [55] Han S, Mao H and Dally W J 2016 Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding *4th Int. Conf. on Learning Representations (ICLR) 2016, Conf. Track Proc.*
- [56] LeCun Y, Denker J S and Solla S A 1990 Optimal brain damage *Advances in Neural Information Processing Systems 2*, ed D S Touretzky (San Francisco, CA: Morgan–Kaufmann) p 598
- [57] Louizos C, Welling M and Kingma D P 2018 Learning sparse neural networks through l_0 regularization *6th Int. Conf. on Learning Representations* vol 12
- [58] Han S, Pool J, Tran J and Dally W J 2015 Learning both weights and connections for efficient neural networks *Advances in Neural Information Processing Systems 28 (NIPS 2015)*
- [59] Yang T, Chen Y and Sze V 2017 Designing energy-efficient convolutional neural networks using energy-aware pruning *CVPR*
- [60] Zhu M and Gupta S 2017 To prune, or not to prune: exploring the efficacy of pruning for model compression *6th Int. Conf. on Learning Representations (ICLR) 2018, Workshop Track Proc*
- [61] Hubara I et al 2017 Quantized neural networks: training neural networks with low precision weights and activations *J. Mach. Learn. Res.* **18** 6869
- [62] Jacob B et al 2018 Quantization and training of neural networks for efficient integer-arithmetic-only inference (arXiv:1712.05877)
- [63] Courbariaux M, Bengio Y and David J-P et al 2015 BinaryConnect: training deep neural networks with binary weights during propagations *Advances in Neural Information Processing Systems* vol 28, ed C Cortes (Cambridge, MA: MIT Press) p 3123

A Reconfigurable Neural Network ASIC for Detector Front-End Data Compression at the HL-LHC

Giuseppe Di Guglielmo¹, Farah Fahim², *Member, IEEE*, Christian Herwig³, Manuel Blanco Valentin, Javier Duarte⁴, Cristian Gingu, *Member, IEEE*, Philip Harris, James Hirschauer⁵, Martin Kwok, Vladimir Loncar, Yingyi Luo, Llovizna Miranda, Jennifer Ngadiuba, Daniel Noonan, Seda Ogrenic-Memik, Maurizio Pierini, Sioni Summers, and Nhan Tran⁶

Abstract—Despite advances in the programmable logic capabilities of modern trigger systems, a significant bottleneck remains in the amount of data to be transported from the detector to off-detector logic where trigger decisions are made. We demonstrate that a neural network (NN) autoencoder model can be implemented in a radiation-tolerant application-specific integrated circuit (ASIC) to perform lossy data compression alleviating the data transmission problem while preserving critical information of the detector energy profile. For our application, we consider the high-granularity calorimeter from the Compact Muon Solenoid (CMS) experiment at the CERN Large Hadron Collider. The advantage of the machine learning approach is in the flexibility and configurability of the algorithm. By changing

the NN weights, a unique data compression algorithm can be deployed for each sensor in different detector regions and changing detector or collider conditions. To meet area, performance, and power constraints, we perform quantization-aware training to create an optimized NN hardware implementation. The design is achieved through the use of high-level synthesis tools and the `hls4m1` framework and was processed through synthesis and physical layout flows based on a low-power (LP)-CMOS 65-nm technology node. The flow anticipates 200 Mrad of ionizing radiation to select gates and reports a total area of 3.6 mm² and consumes 95 mW of power. The simulated energy consumption per inference is 2.4 nJ. This is the first radiation-tolerant on-detector ASIC implementation of an NN that has been designed for particle physics applications.

Index Terms—Application-specific integrated circuit (ASIC), artificial intelligence (AI), autoencoder, hardware accelerator, high-level synthesis (HLS), Large Hadron Collider (LHC), machine learning (ML), single-event effect (SEE) mitigation.

Manuscript received October 31, 2020; revised February 12, 2021 and April 6, 2021; accepted May 23, 2021. Date of publication June 7, 2021; date of current version August 16, 2021. The work of Farah Fahim, Christian Herwig, Cristian Gingu, James Hirschauer, Llovizna Miranda, and Nhan Tran was supported by the Fermi Research Alliance, LLC through the U.S. Department of Energy (DOE), Office of Science, Office of High Energy Physics under Contract DE-AC02-07CH11359. The work of Javier Duarte was supported by the DOE, Office of Science, Office of High Energy Physics Early Career Research Program under Award DE-SC0021187. The work of Philip Harris was supported by the Massachusetts Institute of Technology University Grant. The work of Vladimir Loncar, Maurizio Pierini, and Sioni Summers was supported by the European Research Council (ERC) through the European Union's Horizon 2020 Research and Innovation Program under Grant 772369.

Giuseppe Di Guglielmo is with the Computer Science Department, Columbia University, New York, NY 10027 USA.

Farah Fahim and Nhan Tran are with the Fermi National Accelerator Laboratory, Batavia, IL 60510 USA, and also with the Electrical and Computer Engineering Department, Northwestern University, Evanston, IL 60208 USA (e-mail: ntran@fnal.gov).

Christian Herwig, Cristian Gingu, James Hirschauer, and Llovizna Miranda are with the Fermi National Accelerator Laboratory, Batavia, IL 60510 USA.

Manuel Blanco Valentin, Yingyi Luo, and Seda Ogrenic-Memik are with the Electrical and Computer Engineering Department, Northwestern University, Evanston, IL 60208 USA.

Javier Duarte is with the Physics Department, UC San Diego, La Jolla, CA 92093 USA.

Philip Harris is with the Massachusetts Institute of Technology, Cambridge, MA 02139 USA.

Martin Kwok is with the Physics Department, Brown University, Providence, RI 02912 USA.

Vladimir Loncar is with CERN, 1211 Geneva, Switzerland, and also with the Institute of Physics Belgrade, 11080 Belgrade, Serbia.

Jennifer Ngadiuba is with the California Institute of Technology, Pasadena, CA 91125 USA.

Daniel Noonan is with the Florida Institute of Technology, Melbourne, FL 32901 USA.

Maurizio Pierini and Sioni Summers are with CERN, 1211 Geneva, Switzerland.

Color versions of one or more figures in this article are available at <https://doi.org/10.1109/TNS.2021.3087100>.

Digital Object Identifier 10.1109/TNS.2021.3087100

0018-9499 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.

See <https://www.ieee.org/publications/rights/index.html> for more information.

Authorized licensed use limited to: CERN. Downloaded on October 25, 2023 at 21:26:19 UTC from IEEE Xplore. Restrictions apply.

I. INTRODUCTION

BREAKTHROUGHS in the precision and speed of sensing instrumentation are impactful on advances in scientific methodologies and theories. Thus, a common paradigm across many scientific disciplines in physics has been to increase the resolution of the sensing equipment in order to increase either the robustness or the sensitivity of the experiment itself. This demand for increasingly higher sensitivity in experiments, along with advances in the design of state-of-the-art sensing systems, has resulted in rapidly growing big data pipelines such that transmission of acquired data to the processing unit via conventional methods is no longer feasible. Data transmission is commonly much less efficient than data processing. Therefore, placing data compression and processing as close as possible to data creation while maintaining physics performance is a crucial task in modern physics experiments.

At the CERN Large Hadron Collider (LHC) and its high luminosity upgrade (HL-LHC), extreme collision rates present extreme challenges for data processing and transmission at multiple stages in detector readout and trigger systems. As the initial stage in the data chain, the on-detector (front-end) electronics that readout detector sensors must operate with low latency and low-power (LP) dissipation in a high-radiation environment, necessitating the use of application-specific integrated circuits (ASICs). In order to mitigate the initial bottleneck of moving data from front-end ASICs to off-detector (back-end) systems based on field-programmable gate arrays (FPGAs), front-end ASICs must provide edge com-

puting resources to efficiently use limited bandwidth through real-time processing and identification of interesting data. Front-end data compression algorithms have historically relied on zero-suppression, threshold-based selection, and sorting or summing of data.

Artificial intelligence (AI), and, more specifically, machine learning (ML), has recently been demonstrated to be a powerful tool for data compression, processing, and analysis in physics [1]–[4] and many other domains. While progress has been made toward generic real-time processing through inference, including boosted decision trees and neural networks (NNs) using FPGAs in off-detector electronics [5], [6], ML methods have not yet been used to address the significant bottleneck in the transport of data from front-end ASICs to back-end FPGAs.

The high-granularity endcap calorimeter (HGCAL) [7] currently under construction by the Compact Muon Solenoid (CMS) experiment [8] for eventual use at HL-LHC provides an excellent example of the big data challenges facing high-energy physics. As an *imaging calorimeter*, the HGCAL includes over six million readout channels, providing an unprecedented level of segmentation for calorimetry at high-energy colliders. In order to provide input to the real-time event filtering (trigger) system of CMS, the HGCAL transmits a stream of trigger data at a frequency of 40 MHz, resulting in massive data rates. At data creation, two ASICs are used to digitize and encode trigger data before transmission to back-end FPGAs for further processing.

In this article, we explore the application of ML algorithms to the task of processing large amounts of data with low latency and LP in a high-radiation environment in order to maximize the efficient use of limited bandwidth. We focus on an ASIC implementation of an autoencoder algorithm that uses a configurable NN to efficiently compress and encode data automatically before transmission. Subsequent stages of data processing can either decode the data or continue analyzing the encoded data. In our ASIC implementation, the NN architecture is fixed, but exceptional flexibility in application is preserved by making the NN *weights* programmable. We apply our methodology to the specific front-end data transmission challenge of the CMS HGCAL, showing that the advantage of our approach lies in the flexibility and configurability of the algorithm, which allows us to generate unique data compression algorithms depending on HGCAL sensor geometry, sensor location on the detector and the corresponding occupancy and signal patterns, changing accelerator conditions, or changing detector conditions.

The remainder of this article is organized as follows. In Section II, we introduce the HGCAL challenge in greater detail and outline our conceptual approach. Then, in Section III, we elaborate on the design and training of the autoencoder NN for the specific case of the CMS HGCAL detector. In Section IV, we present the digital implementation of the trained NN in the ASIC. Finally, we summarize our work and discuss future directions in Section V.

II. SYSTEM CONSTRAINTS AND CONCEPT

The HGCAL is a major upgrade of the CMS endcap calorimeter planned for the HL-LHC and provides a fitting

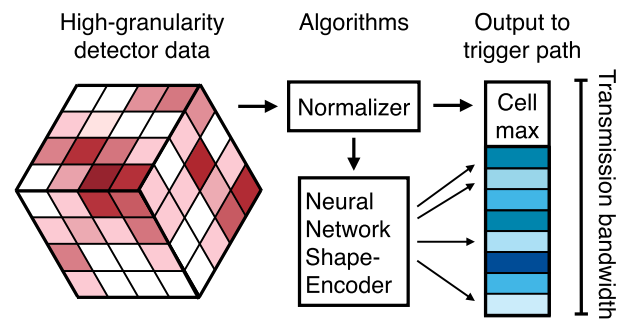


Fig. 1. Simplified version of the internal flow of the autoencoder compression task, which takes the module energy deposits, normalizes them to the sum of the energy in the module, and then performs shape encoding.

demonstrator for the ASIC ML accelerator technology. The HGCAL is described in detail in [7]; relevant implementation details that have changed since the publication of [7] are updated in this article.

This “imaging calorimeter,” which includes over 600 m² of active silicon and over six million readout channels, is composed of 50 layers of active shower-sampling media interleaved with the dense absorber. The active medium of the 28-layer front electromagnetic compartment is silicon, while the 22-layer rear hadronic compartment includes both silicon and plastic scintillators. Silicon layers are tiled with 8” hexagonal sensor modules, with each module including 48 logical trigger cells (TCs) arranged in three 4 × 4 matrices, as shown in Fig. 1. While the NN can be configured for both the silicon and scintillator geometries, silicon geometry is used throughout this article to illustrate the concepts.

To provide input to the CMS trigger system, data must be transmitted from the on-sensor analog-to-digital ASICs to the all-FPGA back-end detector electronics system at the nominal HL-LHC collision rate of 40 MHz. Because bandwidth constraints prohibit transmission of data for all 48 TCs at 40 MHz, a front-end concentrator ASIC (ECON-T) is being developed to compress a single sensor’s information before transmission to the back-end trigger electronics. Each sensor module produces 7 bits of floating-point charge data for each of the 48 TCs at 40 MHz. Thus, the lossy compression task of the ECON-T ASIC is to aggregate the 48 7-bit signals from a sensor and compress the data into a range spanning 48–144 bits while maximally preserving the energy pattern of the sensor. The range of the output bits depends on the location of the sensor module in the detector and the number of links available for a given ECON-T ASIC to transmit the data. The number of links allocated will roughly correspond to the average sensor occupancy, which varies by two orders of magnitude over all sensor locations. The exact task depends on the handling of data framing and TC address information and on whether the ECON-T algorithm operates with fixed or variable latency. The ECON-T design provides the user a choice among three expert algorithms for TC compression, including TC threshold application, sorting and selection of highest energy TC, and aggregation of adjacent TCs. Unused algorithms are clock-gated to conserve power.

The ECON-T ASIC is being developed for the LP-CMOS 65-nm feature size technology and is under active development for CMS. Because it is located on-detector in a high-radiation

environment, its design also requires tolerance to single-event effects. The allocated footprint for the fabrication of this chip is approximately $5 \times 5 \text{ mm}^2$. It is expected that sufficient area will be available for potential inclusion of our NN compression logic with an approximate area of 4 mm^2 . The constraints for the compression algorithms are that they should accept new input data at 40 MHz and complete processing in 50 ns. The power budget of the task is less than 100 mW.

Our contribution is an NN to perform the ECON-T compression task. It is a central tenet of our design that the compression algorithm is *reconfigurable*. Because we are implementing a design for an ASIC, the architecture of the NN will be fixed, but the *weights* of the NN need to be configurable such that the algorithm itself is adaptable. This has several advantages. Through reconfiguration, we will be able to do the following.

- 1) *Enable* more computationally complex compression algorithms, which could improve overall physics performance or allow more flexible algorithms.
- 2) *Customize* the compression algorithm of each sensor based on its location within the detector.
- 3) *Adapt* the compression algorithm for changing detector and collider conditions (for example, if the detector loses a channel or has a noisy channel, it can be accounted for, or if the collider has more pileup than expected, the algorithm can be adjusted to deal with new or unexpected conditions without catastrophic failure).

For our compression algorithm, we choose to utilize an autoencoder architecture. It provides a generic and flexible compression solution, consisting of two NNs: an encoder and a decoder. The encoding network maps inputs to an intermediate *latent* representation with lower dimensionality than the space of inputs, after which the decoding network aims to recover the original signal. In the HGCAL application, the encoding NN would compress HGCAL data on the ASIC before transmission to the calorimeter trigger FPGAs for subsequent decoding. Ultimately, in the final realistic system, we do not anticipate using a full autoencoder architecture because FPGA resources on the back-end FPGA system will not be sufficient to do full decoding for every sensor. However, in the absence of understanding how best to use the latent representation later in the processing chain, we optimize performance for an autoencoder because it is a reasonable proxy for the encoder NN encapsulating the salient sensor features such that the image can be decoded from the latent representation. Finally, in Fig. 1, the compression task is split into two parts: an overall normalization over the entire sensor to preserve the total energy in the sensor and the NN *shape* encoder, which encodes the energy pattern across the sensor.

For the automated design tool flow, it is very important to have a rapid codesign loop between the NN algorithm training and the implementation in hardware in order to understand whether the algorithm is meeting system constraints for power, area, and performance simultaneously. To achieve this, we use `hls4ml` [5] that translates NNs trained in common open-source ML software frameworks into register transfer level (RTL) using high-level synthesis (HLS) tools [9]. The efficacy of this approach will be described in greater detail in Section IV.

III. ALGORITHM DESIGN AND PERFORMANCE

Our task is to design an algorithm that will reproduce the energy pattern in the sensor while simultaneously adhering to hardware constraints, i.e., fitting in the available area within the ECON-T ASIC chip while complying with system latency and power constraints. Because we are training the algorithm based on a single sensor's energy pattern, we will not be able to optimize for multisensor physics performance, such as particle energy resolution. Ultimately, the physics performance may determine the final system optimization; however, it is beyond the scope of this study. Therefore, our target is to design an algorithm that reproduces the original sensor energy pattern as accurately as possible through the autoencoder compression-and-decompression bottleneck.

There are a number of elements needed to design our compression algorithm: a sample of events for training and validation, a preprocessing and normalization block, an optimized NN architecture, and metrics for evaluating the NN performance, both for determining the training loss and the final network evaluation. An essential aspect of the training procedure is quantization-aware training (QAT), i.e., we approximate bit-accurate reduced precision for all of the NN calculations during training. QAT is known to be much more performant than posttraining quantization (PTQ), where the training is done using 32-bit floating-point operations, which are then truncated posttraining to fixed-point or integer representations. In a previous study of the `QKeras` [10] tool, QAT was performed for an LHC trigger task. It was found that, with PTQ, the minimum bit width possible without loss of performance was 14 bits, while, with QAT, the same performance could be achieved with 6-bit weights. Thus, PTQ would lead to a more than fourfold increase in the area of an ASIC design based on the bit operations hardware design metric [11]. Therefore, we use `QKeras` for training the NNs in this study.

Training Sample: Test energy patterns in the sensors are simulated using top-quark-pair events overlaid with 200 simultaneous collisions per bunch crossing (BX) in the CMS software framework [12]. These simulated events create a sample of typical energy patterns in the HGCAL sensors, which we use as a realistic proxy for the sensor data.

Preprocessing: The compression task is factorized into normalization and shape extraction components, as illustrated in Fig. 1. The first stage of ECON-T processing for all algorithms is to expand the 7-bit floating-point TC data to the inherent 22-bit fixed-point TC data. The sum value of all 48 TC is identified and used to normalize the charge distribution across the full sensor (and the sum of TC charge is included in the final data payload to allow subsequent interpretation of normalized TC data). The normalized NN inputs are truncated to 8 bits to allow a more compact NN implementation while ensuring that any omitted cells constitute less than 1% of the total energy recorded within a module.

NN Architecture: The encoding NN architecture consists of successive layers that sequentially process the input data. Convolutional layers are used to extract spatial features from images through the application of filters: matrices of configurable parameters. While convolutional layers

use relatively few parameters, convolution requires many multiply-accumulate operations (MACs). Conversely, a fully connected layer multiplies a vector of input elements by a matrix of configurable weights, generally requiring more configurable parameters and fewer MACs than a convolution. The impact of the choice of precision for all internal parameters (constrained by the available area on the chip) is accounted for by training inherently quantized models with the QKeras package [10]. Because the HGAL sensor data compression task takes as input an image data representation, we consider a convolutional NN layer as a natural approach. Typical convolutions rely on the input being in a Cartesian representation though other shapes can be explored in future work. Here, we map the hexagonal sensor shape to a more typical Cartesian arrangement, as illustrated in Fig. 2, which simplifies the training and hardware implementation.

Training and Performance Metrics: The performance of the autoencoder is based on how well the original image is reproduced after encoding and decoding. We quantify the difference between raw and decoded HGAL data with the energy mover's distance (EMD) [13]. Given a particular normalized energy distribution, one may physically relocate some energy fraction dE by a spatial distance dx , leading to a new distribution with an associated rearrangement cost $dEdx$. This notion can be extended to define an "optimal transport" between two energy distributions \mathcal{A} and \mathcal{B} , as a remapping that minimizes this total rearrangement cost, $\text{EMD}(\mathcal{A}, \mathcal{B})$. While the performance of the autoencoder is assessed with EMD, taking into account the complete hexagonal sensor geometry, this metric is not used directly in the algorithm training, as it involves nondifferentiable and computationally intensive operations. Models are instead trained with a modified χ^2 loss function incorporating cell-to-cell distances, as a fast approximation of EMD. Specifically, individual TCs are resummed into all physical groups of approximately 2×2 and 3×3 "super-cells" based on the full hexagonal cells with corresponding χ^2 values computed for the coarsened images. The total loss sums each such χ^2 together, resulting in a comparatively lenient penalty when misreconstruction occurs only on small spatial scales. Including these additional χ^2 terms in the training procedure is found to yield significant improvements to the autoencoder performance, as measured with EMD. To ensure an unbiased NN optimization, the data are randomly partitioned into separate samples for training (80%) and validation (20%), with training termination set by the loss observed in the validation sample.

A. Baseline Encoder Model

A simple encoding NN with a single convolutional and dense layer architecture is investigated. Normalized inputs from hexagonal sensors are arranged into three arrays of 4×4 to form a regular geometry. The convolution layer consists of eight $3 \times 3 \times 3$ kernel matrices, giving an $8 \times 4 \times 4$ output after convolution. It was found that more than eight kernel matrices brought negligible performance improvement. These 128 values are flattened and fed through a dense layer to yield 16 9-bit output values. Rectified linear unit (ReLU) activations [14], [15] are applied before and after

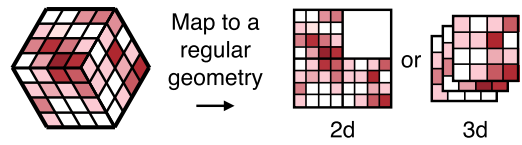


Fig. 2. Mapping the hexagonal sensor geometry to potential Cartesian representations for convolutional layer operations.

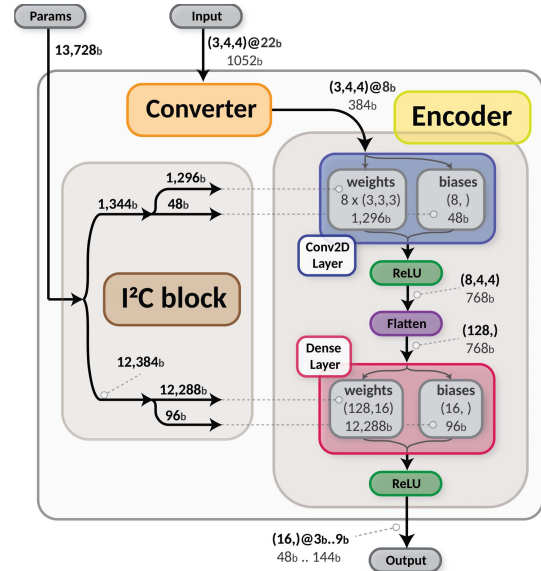


Fig. 3. Autoencoder NN architecture and data flow for the baseline encoder model.

the dense layer. This leads to a total of 2288 weight parameters (dominated by the 2064 parameters used to configure the dense layer), each of which is specified with 6-bit precision. A single inference requires a total of 4448 MACs, with similar requirements from the convolution (2400) and dense layers (2048). The size and complexity of this baseline model are constrained by area, on-chip memory and interfaces, and power, which impose additional optimization considerations. The encoder architecture with the reconfigurable weights is illustrated in Fig. 3.

B. Optimization Considerations and Comparisons

While the presence of a single convolutional layer is critical for good physics performance of the algorithm (approximated by the EMD between input and decoded images), adding more filters or additional convolutional layers only weakly improves physics performance, at the expense of significantly increased area. Changes in the number and size of the dense layers yield more dramatic differences.

Fig. 4 shows a sweep over the number of dense layer outputs, where remaining aspects of the design are fixed based on hardware constraints: the precision of outputs and weights are coherently varied to ensure that both the total number of outputs and the weight bits are fixed. Architectures featuring many outputs with lower relative precision consistently outperform their counterparts. The autoencoder is robust across a variety of conditions and performs well in the high-occupancy regime, which poses the greatest challenge for trigger reconstruction.

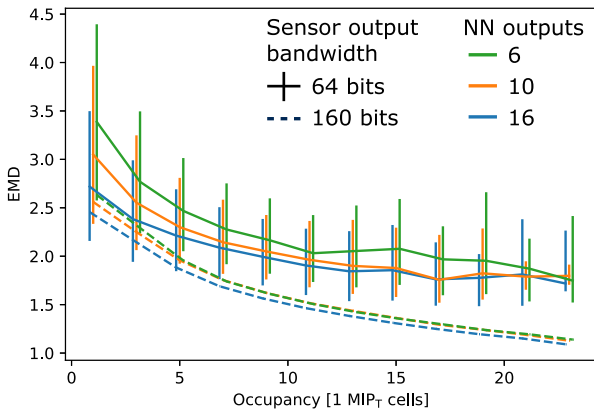


Fig. 4. Median EMD for decoded HGCAL images from the validation dataset, as a function of sensor occupancy for six NN configurations. Vertical lines (suppressed for the 160-bit configurations) denote 68% EMD intervals. Occupancy is defined as the number of TCs with signals exceeding one minimum ionizing particle divided by $\cosh \eta$, where η is the pseudorapidity of the TC. (Results are shown for the version of NN with the maximum of 10 bits for each of 16 outputs rather than 9 bits, as described in the text.)

Reconfigurability: Fig. 4 also demonstrates how the same NN encoder can be reoptimized and configured for new data-taking conditions, by comparing sensors in detector regions requiring low and high throughputs. The maximum data throughput of 144 bits from 16 9-bit outputs can be reduced through fully configurable selective truncation. Expected use cases include transmission of (48, 80, 112, and 144) bits from 16 (3, 4, 7, and 9)-bit outputs though the network can also be configured to transmit fewer than 16 outputs or a mix of precisions.

IV. IMPLEMENTATION METHODOLOGY AND RESULTS

In this section, we detail the implementation of the trained NN described in Section III in the ECON-T ASIC. We discuss the design and verification flow, the architectural and design exploration, steps required for deployment in a radiation environment, design performance metrics, and, finally, the implementation results.

A. Algorithm to Accelerator Development

For our design flow, we adopted the `hls4ml` framework [5] to automate the mapping of ML models onto reconfigurable logic. For this work, we extended `hls4ml` to our ASIC flow. Traditionally, hardware designers utilize hardware description languages (HDLs) and a level of abstraction known as the RTL. In recent years, HLS has become an alternative for generating hardware modules from code written in programming languages, such as C/C++. HLS comes with significant benefits: it raises the level of abstraction and reduces the simulation time; it simplifies the verification phases; and finally, it makes the exploration and evaluation of design alternatives easier. The original flow of `hls4ml` generates state-of-the-art synthesizable C++ code and HLS directives from the ML-model specifications. The generated code is then fed into the Vivado HLS tool to generate an accelerator in HDL RTL code for the deployment on Xilinx FPGAs [16]. We extended `hls4ml` to support Mentor’s Catapult HLS [17]

TABLE I

AREA BREAKDOWN FOR PIPELINED IMPLEMENTATIONS. THE RESULTS ARE FROM CATAPULT HLS ESTIMATIONS, AND AREAS ARE IN μm^2

Initiation Interval	Total Area	Register Area	MUX Area
1	1,138,242	925	0
2	891,195	5,228	12,989
4	765,877	8,503	16,089
8	699,988	8,509	16,252

tool and target our specific 65-nm LP-CMOS technology for ASIC fabrication. We integrated the HLS-generated code with a SystemVerilog RTL IP of the programmable I²C peripheral.¹ We finally created a component database and layout to be incorporated into the ECON-T ASIC top-level assembly using a digital implementation flow. The standard flow was modified to accommodate automatic triple modular redundancy (TMR) implementation for HLS-generated RTL integrated with other SystemVerilog modules.

We complemented our design flow with a robust validation and verification methodology across the various refinement steps. We validated the C++ HLS code against the `QKeras` trained model to guarantee the model’s functional correctness. Earlier in the design flow, we also performed dynamic and static verification of the synthesizable specifications: we checked design rules with static analysis of the C++ HLS code (Mentor CDesignChecker [19]), measured coverage metrics (Mentor CCov [19]), and, finally, ran simulation-based equivalence checking. For the HLS-generated RTL code, we followed a more traditional simulation-based verification to ensure optimized power, area, and speed.

B. Architectural Exploration

`hls4ml` coupled with the industry-standard Catapult HLS (ver. 10.6) tool allowed us to explore the cost and performance tradeoffs of various microarchitectural hardware implementations for our ML model. We decided on a pipelined implementation for our accelerator to increase concurrent execution as an early design decision. A pipelined design can process new inputs every N clock cycles, where N is the initiation interval (II) of the design. Table I shows the area breakdown for different II values (1, 2, 4, and 8). It is noticeable that, although the area is higher for $\text{II} = 1$, the required resources are mostly functional logic to implement a highly parallel datapath, i.e., there are no multiplexers (MUXs). A higher II value implies less design parallelism and more functional-resource reuse. This choice reduces the overall area, but the resource breakdown shows an increase in control logic (MUX) and registers. II of 1 was ultimately selected so that new inputs may be processed in sync with a single clock operating at 40-MHz LHC crossing frequency.

We used a fixed-point representation (`ac_fixed` [20]) for the input, intermediate, and output parameters of our ML model designed with `hls4ml`. This choice provided us with a high degree of flexibility for exploring the area and accuracy tradeoff of the ML-model hardware implementations obtained with HLS. The RTL schematics for the encoder block are

¹The authors use *controller/peripheral* in place of *master/slave* when referring to such I²C devices or processes [18].

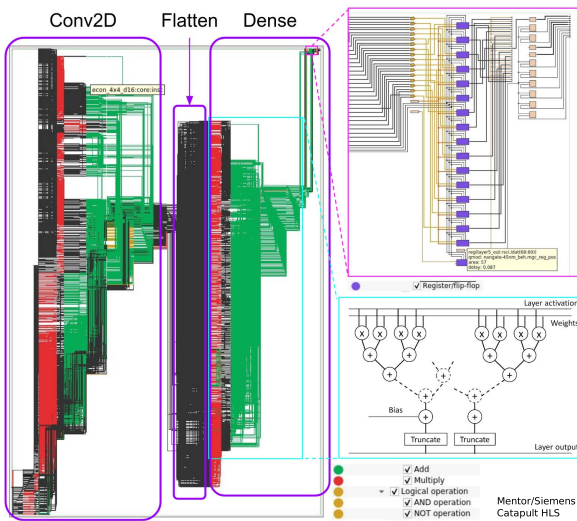


Fig. 5. Encoder RTL schematics and the basic structure of the convolution and dense layers can be seen at the schematic level on the left, and zoomed in images are provided for the output and MAC portions on the right.

shown in Fig. 5. The basic structure of the convolution and dense layers can be seen at the schematic level. The top right and bottom right diagrams are zoomed-in portions of this schematic depicting the output and MACs.

C. Digital Implementation in a Radiation Environment

The digital design consists of three major functional blocks: 1) a converter that is a classical module designed with HLS; 2) an encoder that uses `hls4ml`; and 3) an I²C peripheral that uses a SystemVerilog RTL code. The converter is used for normalizing the 48 (22 b) inputs to 48 (8 b). An encoder is used for data classification and further compression to 16 (9 b) outputs. To have a flexible and reconfigurable algorithm, all the parameters (13 728 b) can be set up via the I²C interface on-chip. The programming of the I²C peripheral takes less than 50 μ s corresponding to a total of 1716 I²C clock cycles, utilizing an 8-b input bus. Once the weights are set up, the algorithm adds a total latency of 2 BX cycles to the trigger path—one cycle to convert and another cycle to encode resulting in total inference latency of 50 ns and a new input accepted every 25 ns.

1) *Integrated Converter, Encoder, and I²C Peripheral*: An integrated approach to the development is needed in order to avoid routing congestion of connecting the weights to the appropriate layers across the encoder. The floor plan of the digital implementation occupying 2.4 mm \times 1.5 mm is shown in Fig. 6. The converter logic is located near the data input at the top of the design, and the majority of the area is occupied by the encoder, interleaved with the distributed I²C network.

2) *Design Considerations for Total Ionizing Dose Performance*: Apart from all requirements considered above, our design must guarantee on-detector circuit reliability in the high-radiation environment of HL-LHC [21], [22]. The circuitry should withstand a total ionizing dose of approximately 200 Mrad over the lifetime of the experiment along with high single-event effect (SEE) rates [23]–[25]. Since previous measurement results have indicated that the average time delay of all cells from the 65-nm LP process library increases after

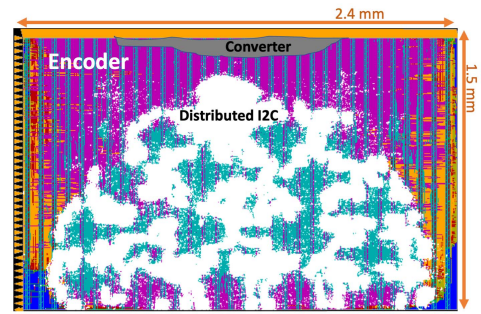


Fig. 6. Design floor plan with an integrated converter, encoder, and I²C peripheral occupying a total area of 3.6 mm². The converter is highlighted in gray, the I²C peripheral is highlighted in white, and the rest of the area is occupied by the encoder.

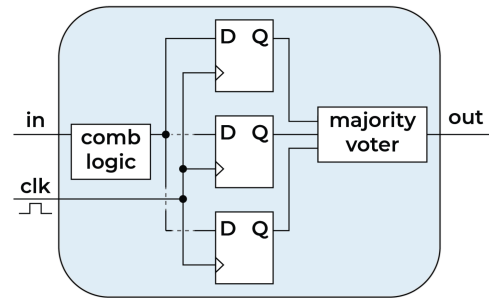


Fig. 7. TMR scheme used for the encoder and converter. Each register is triplicated, and a majority voter determines the output.

200 Mrad irradiation [26], minimum size cells are avoided. Normal Vt standard cell technology library is used. The implementation uses concurrent multimode multicorner static timing analysis for ensuring performance. The foundry worst case libraries are a good stand-in for modeling radiation damage. All weights are stored in registers, and no static random access memories (SRAMs) or dual interlocked storage cell (DICE) [27] are used.

3) *Single-Event Effect Mitigation*: Mitigating SEEs is a critical step in the ASIC implementation for effective performance in the HL-LHC environment. Several techniques have been proposed and used over the years to tackle this specific problem [28]–[30].

TMR is a well-known technique to protect digital circuits against the undesirable effects of SEEs [31], [32]. Depending on the functionality of the block autocorrection features might be required for registers which store data.

We have used two different TMR implementations: simple TMR with triplicated registers and a majority voter for the data path shown in Fig. 7 and fully triplicating the entire module, as shown in Fig. 8, for the I²C peripheral for storing weights.

Since new data arrives at the encoder block every 25 ns, no autocorrection techniques are required. On the other hand, the values of the weights set by the I²C peripheral (parameters of the NN) are vital as they are central to the vector multiplications used in NNs. Once programmed these are not expected to change over long periods of time, hence, autocorrection techniques are used to ensure that register errors due to single-event upsets (SEUs) do not accumulate over time. As shown in Fig. 8, all combinational logic within the module is triplicated, which is used by three majority voters to form the inputs to triplicated registers. The feedback from

TABLE II
DESIGN (D) AND VERIFICATION (V) METRICS FROM MODEL GENERATION TO VERIFIED IP

STEP	TIME	ITER.	SIZE
Model generation (D)	0.98s	50-100	1089 C++ LoC
C Simulation (V)	0.14s		
High-level synthesis (D)	00:30:17	3-100	39,716 Verilog LoC
RTL Simulation (V)	00:00:46		
Logic synthesis (D)	06:04:19	6	769,481 gates
Gate-level simulation (V)	00:25:19		
Place and route (D)	39:33:11		
Post-layout simulation (V)	00:51:41		
Post-layout parasitic simulation (V)	01:51:30		
SEE simulation (V)	04:17:00		
Layout (D)	~ 00:20:00	1	
LVS & DRC (V)	~ 01:00:00		

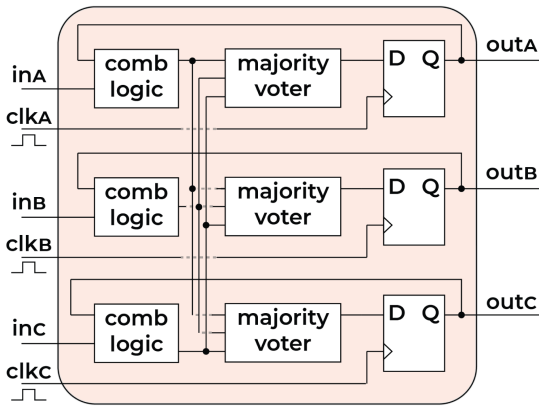


Fig. 8. Full module triplication is used for the I²C peripheral. All combinational logic within the module is triplicated, which is used by three majority voters to form the inputs to triplicated registers. The feedback from the output of the registers enables autocorrection and protects against accumulating errors due to SEUs over time.

the output of the registers enables autocorrection. This method does require the I²C peripheral to be clocked periodically.

Performance Metrics and Implementation Results

Table II lists metrics characterizing our design flow, such as the time spent for the design (D) and validation/verification (V) stages, the number of iterations, and the complexity of design representation at each stage. The table illustrates the main steps required to create a full and validated design. A codesign approach requires being able to have a rapid transition between each step to inform the other steps.

The HLS model description requires approximately 1000 lines of code. This stage is fast (~ 1 s) but requires several hundred iterations to optimize the algorithm performance, driven by the physics goals. The HLS stage determines the level of parallelism in the design, choice of pipelining, resource reuse factor, and clock frequency. This directly impacts the total area, power consumption, and latency of the design. One hot encoding of finite state machines for robust SEU prevention also needs to be introduced at this stage. Clock gating is employed to save system-level power. The digital implementation stage is time-intensive, requiring ~ 65 h of design and verification to meet the speed and area constraints with fewer iterations.

The final implementation results are presented in Table III. While there are challenges due to technology choices in

TABLE III
KEY SIMULATION PERFORMANCE PARAMETERS OF THE DESIGN

Latency	Energy/inference	Power	Area
50 ns	2.38 nJ/inf.	95 mW	3.6 mm ²

making a comparison with our design in an FPGA, we consider a fully unrolled FPGA implementation on a typical Xilinx Kintex Ultrascale FPGA device. Considering algorithm block power only and depending on configuration choices, an equivalent FPGA implementation consumes roughly 2.5–5 W with a latency of ~ 300 ns [5] compared to the ASIC implementation, which consumes 95 mW. The ASIC implementation is expected to provide more than an order of magnitude improvement in power with a reduction in latency.

V. CONCLUSION

A design methodology spanning from ML model generation to ASIC IP block creation has been presented. An LP, low-latency reconfigurable data compression algorithm based on a convolutional NN has been processed through synthesis and physical layout flows based on a 65-nm LP-CMOS process, designed to withstand radiation environments of up to 200 Mrad.

For the ECON-T ASIC, our task is to perform efficient lossy compression of an HGCAL sensor energy pattern to transmit data to off-detector electronics. Compression is accomplished using an NN autoencoder consisting of convolution and dense NN layers. Optimal design and training of the algorithm are performed using QAT techniques to achieve good physics performance while optimizing for LP and area.

The encoder architecture set by the model requires approximately 225 000 MAC to perform vector multiplications every 25 ns. In order to optimize for LP operation while maintaining data throughput of 40 MHz, a highly parallel architecture is chosen at the expense of a larger area. The energy consumption per inference is 2.38 nJ. The final design consists of 800k gates and occupies a total area of 3.6 mm².

The design demonstrates how complex NN architectures can be implemented on the front-end ASICs with realistic area constraints, allowing for minimal loss of information in the trigger data stream. Furthermore, we show that, in spite of a fixed ASIC implementation, ML algorithms can still be designed with sufficient flexibility to enable reconfiguration for new operational conditions. Apart from that, the I²C block allows real-time reconfiguration of weights, thus facilitating

the first steps toward real-time embedded AI. This is the first time that a radiation-tolerant on-detector ASIC implementation of an NN has been designed for particle physics applications.

We look forward to the inclusion of the design IP in the ECON-T ASIC fabrication. This would allow us to test the design in a physical chip. Beyond the ECON-T ASIC, there is vast potential for future work using our design methodology, including other domain applications and adaptations for ultra-LP, longer latency applications; other technology nodes and design considerations; and more types of NN architectures, which could be scaled out to larger and more complex designs.

AUTHORS' CONTRIBUTION

The project was initiated and coordinated by Farah Fahim, James Hirschauer, and Nhan Tran. Christian Herwig, Martin Kwok, and Daniel Noonan led the development of the algorithm training and high-level synthesis (HLS) translation. Giuseppe Di Guglielmo and Christian Herwig performed HLS to register transfer level (RTL) synthesis and validation of the RTL design. Farah Fahim and Yingyi Luo focused on the digital implementation, power analysis, and algorithm interfaces to the rest of the chip including I²C with Cristian Gingu. Farah Fahim, Manuel Blanco Valentin, and Llovizna Miranda developed techniques for making the design radiation-tolerant. James Hirschauer, Seda Ogrenci-Memik, and Nhan Tran provided supervision for the project. Javier Duarte, Philip Harris, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, and Sioni Summers provided support for the h1s4m1 infrastructure and initial algorithm implementations in HLS.

ACKNOWLEDGMENT

The authors would like to acknowledge CAD support from Sandeep Garg and Anoop Saha from Mentor Graphics for Catapult high-level synthesis (HLS) and Bruce Cauble and Brent Carlson from Cadence for Innovus and Incisive. They also like to thank the Fermilab application-specific integrated circuit (ASIC) group for incorporating the autoencoder block into the ECON-T ASIC; CMS high-granularity endcap calorimeter (HGCAL) and Jean-Baptiste Sauvan for providing simulated module images for training; and Andre Davide for extensive input on network optimization. They acknowledge the Fast Machine Learning Collective as an open community of multidomain experts and collaborators. This community was important for the development of this project.

REFERENCES

- [1] K. Albertsson *et al.*, "Machine learning in high energy physics community white paper," *J. Phys., Conf. Ser.*, vol. 1085, Sep. 2018, Art. no. 022008.
- [2] A. Radovic *et al.*, "Machine learning at the energy and intensity frontiers of particle physics," *Nature*, vol. 560, p. 41, Aug. 2018.
- [3] D. Bourilkov, "Machine and deep learning applications in particle physics," *Int. J. Mod. Phys. A*, vol. 34, no. 35, Dec. 2019, Art. no. 1930019.
- [4] G. Carleo *et al.*, "Machine learning and the physical sciences," *Rev. Mod. Phys.*, vol. 91, Dec. 2019, Art. no. 045002.
- [5] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *J. Instrum.*, vol. 13, no. 7, Jul. 2018, Art. no. P07027.
- [6] CMS Collaboration, "The phase-2 upgrade of the CMS level-1 trigger," CERN, Geneva, Switzerland, CMS Tech. Design Rep. CERN-LHCC-2020-004 CMS-TDR-021, 2020. [Online]. Available: <https://cds.cern.ch/record/2714892>
- [7] CMS Collaboration, "The phase-2 upgrade of the CMS endcap calorimeter," CERN, Geneva, Switzerland, CMS Tech. Des. Rep. CERN-LHCC-2017-023 CMS-TDR-019, 2017. [Online]. Available: <https://cds.cern.ch/record/2293646>
- [8] S. Chatrchyan *et al.*, "The CMS experiment at the CERN LHC," in *Proc. JINST*, vol. 3, 2008, Art. no. S08004.
- [9] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.
- [10] C. N. Coelho *et al.*, "Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders," 2020, *arXiv:2006.10159*. [Online]. Available: <https://arxiv.org/abs/2006.10159>
- [11] A. Karbachevsky *et al.*, "Early-stage neural network hardware performance analysis," *Sustainability*, vol. 13, no. 2, p. 717, Jan. 2021.
- [12] CMS Team. *CMSSW on GitHub*. Accessed: Jun. 1, 2020. [Online]. Available: <http://cms-sw.github.io/>
- [13] P. T. Komiske, E. M. Metodiev, and J. Thaler, "Metric space of collider events," *Phys. Rev. Lett.*, vol. 123, no. 4, Jul. 2019, Art. no. 041801.
- [14] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*. Madison, WI, USA: Omnipress, 2010, p. 807.
- [15] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, vol. 15, G. Gordon, D. Dunson, and M. Dudík, Eds., Fort Lauderdale, FL, USA, Apr. 2011, p. 315. [Online]. Available: <http://proceedings.mlr.press/v15/glorot11a.html>
- [16] Xilinx. (2020). *Vivado High-Level Synthesis*. [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>
- [17] Mentor/Siemens. (2020). *Catapult High-Level Synthesis*. [Online]. Available: <https://www.mentor.com/hls-lp/catapult-high-level-synthesis>
- [18] Open Source Hardware Association. (2020). *A Resolution to Redefine SPI Signal Names*. [Online]. Available: <https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names>
- [19] Mentor/Siemens. (2020). *Catapult High-Level Synthesis—Verification*. [Online]. Available: <https://www.mentor.com/hls-lp/catapult-high-level-synthesis/hls-verification>
- [20] Mentor/Siemens. (2020). *HLSLibs: Open-Source High-Level Synthesis IP Libraries*. [Online]. Available: <https://hlslibs.org>
- [21] R. G. Alía *et al.*, "LHC and HL-LHC: Present and future radiation environment in the high-luminosity collision points and RHA implications," *IEEE Trans. Nucl. Sci.*, vol. 65, no. 1, pp. 448–456, Jan. 2018.
- [22] M. Huhtinen, "The radiation environment at the CMS experiment at the LHC," Dept. Phys., Ph.D. dissertation, Helsinki Univ. Technol., Espoo, Finland, 1996.
- [23] R. D. Schrimpf and D. M. Fleetwood, *Radiation Effects and Soft Errors in Integrated Circuits and Electronic Devices*, vol. 12. Singapore: World Scientific, 2004.
- [24] E. Petersen, *Single Event Effects in Aerospace*. Hoboken, NJ, USA: Wiley, 2011.
- [25] P. Dodd, M. Shaneyfelt, J. Schwank, and J. Felix, "Current and future challenges in radiation effects on CMOS electronics," *IEEE Trans. Nucl. Sci.*, vol. 57, no. 4, pp. 1747–1763, Aug. 2010.
- [26] L. M. J. Casas *et al.*, "Characterization of radiation effects in 65 nm digital circuits with the DRAD digital radiation test chip," *J. Instrum.*, vol. 12, no. 2, Feb. 2017, Art. no. C02039. [Online]. Available: <https://cds.cern.ch/record/2275135>
- [27] J. A. Maharrey *et al.*, "Dual-interlocked logic for single-event transient mitigation," *IEEE Trans. Nucl. Sci.*, vol. 65, no. 8, pp. 1872–1878, Aug. 2018.
- [28] D. Fulkerson, "Single-event-effect hardened circuitry," U.S. Patent 11 136 920, Nov. 30, 2006.
- [29] S. Kuboyama, H. Shindou, Y. Iide, and A. Makihara, "Single-event-effect tolerant SOI-based inverter, NAND element, nor element, semiconductor memory device and data latch circuit," U.S. Patent 7 504 850, Mar. 17, 2009.
- [30] R. Gong, W. Chen, F. Liu, K. Dai, and Z. Wang, "A new approach to single event effect tolerance based on asynchronous circuit technique," *J. Electron. Test.*, vol. 24, p. 57, Jan. 2008.
- [31] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Dev.*, vol. 6, p. 200, Apr. 1962.
- [32] S. Habinc, "Functional triple modular redundancy (FTMR). VHDL design methodology for redundancy in combinatorial and sequential logic," Gaisler Res., Des. Assessment, Goteborg, Sweden, Tech. Rep. FPGA-003-01 Version 0.2, 2002.

PAPER • OPEN ACCESS

Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml

To cite this article: Nicolò Ghielmetti *et al* 2022 *Mach. Learn.: Sci. Technol.* **3** 045011

View the [article online](#) for updates and enhancements.

You may also like

- [Gradients should stay on path: better estimators of the reverse- and forward KL divergence for normalizing flows](#)
Lorenz Vaitl, Kim A Nicoli, Shinichi Nakajima et al.
- ['Flux+Mutability': a conditional generative approach to one-class classification and anomaly detection](#)
C Fanelli, J Giroux and Z Papandreou
- [Operationally meaningful representations of physical systems in neural networks](#)
Hendrik Poulsen Nautrup, Tony Metger, Raban Iten et al.



PAPER

OPEN ACCESS

RECEIVED
25 May 2022REVISED
27 August 2022ACCEPTED FOR PUBLICATION
21 October 2022PUBLISHED
4 November 2022

Original Content from
this work may be used
under the terms of the
[Creative Commons
Attribution 4.0 licence](#).

Any further distribution
of this work must
maintain attribution to
the author(s) and the title
of the work, journal
citation and DOI.



Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml

Nicolò Ghielmetti^{1,8} , Vladimir Loncar^{1,9} , Maurizio Pierini¹ , Marcel Roed^{1,10}, Sioni Summers¹, Thea Aarrestad² , Christoffer Petersson^{3,11}, Hampus Linander⁴, Jennifer Ngadiuba⁵ , Kelvin Lin^{6,12} and Philip Harris^{7,*} 

¹ European Organization for Nuclear Research (CERN), CH-1211 Geneva 23, Switzerland

² Institute for Particle Physics and Astrophysics, ETH Zürich, 8093 Zürich, Switzerland

³ Zenseact, Gothenburg 41756, Sweden

⁴ University of Gothenburg, Gothenburg 40530, Sweden

⁵ Fermi National Accelerator Laboratory, Batavia, IL 60510, United States of America

⁶ University of Washington, Seattle, WA 98195, United States of America

⁷ Massachusetts Institute of Technology, Cambridge, MA 02139, United States of America

⁸ Also at Politecnico di Milano, Italy.

⁹ Also at Institute of Physics Belgrade, Serbia.

¹⁰ Also at University of Oxford, United Kingdom.

¹¹ Also at Chalmers University of Technology, Sweden.

¹² Currently at Amazon, United States of America.

* Author to whom any correspondence should be addressed.

E-mail: pcharris@mit.edu

Keywords: FPGA, computer vision, deep learning, hls4ml, machine learning, autonomous vehicles, semantic segmentation

Abstract

In this paper, we investigate how field programmable gate arrays can serve as hardware accelerators for real-time semantic segmentation tasks relevant for autonomous driving. Considering compressed versions of the ENet convolutional neural network architecture, we demonstrate a fully-on-chip deployment with a latency of 4.9 ms per image, using less than 30% of the available resources on a Xilinx ZCU102 evaluation board. The latency is reduced to 3 ms per image when increasing the batch size to ten, corresponding to the use case where the autonomous vehicle receives inputs from multiple cameras simultaneously. We show, through aggressive filter reduction and heterogeneous quantization-aware training, and an optimized implementation of convolutional layers, that the power consumption and resource utilization can be significantly reduced while maintaining accuracy on the Cityscapes dataset.

1. Introduction

Deep Learning has strongly reshaped computer vision in the last decade, bringing the accuracy of image recognition applications to unprecedented levels. Improved pattern recognition capabilities have had a significant impact on the advancement of research in science and technology. Many of the challenges faced by future scientific experiments, such as the CERN High Luminosity Large Hadron Collider [1] or the Square Kilometer Array observatory [2], and technological challenges faced by, for example, the automotive industry, will require the capability of processing large amounts of data in real-time, often through edge computing devices with strict latency and power-consumption constraints. This requirement has generated interest in the development of energy-effective neural networks, resulting in efforts like tinyML [3], which aims to reduce power consumption as much as possible without negatively affecting the model accuracy.

Advances in deep learning for computer vision have had a crucial impact on the development of autonomous vehicles, enabling the vehicles to perceive their environment at ever-increasing levels of accuracy and detail. Deep neural networks are used for finding patterns and extracting relevant information from camera images, such as the precise location of the surrounding vehicles and pedestrians. In order for an autonomous vehicle to drive safely and efficiently, it must be able to react fast and make quick decisions.

This imposes strict latency requirements on the neural networks that are deployed to run inference on resource-limited embedded hardware in the vehicle.

In addition to algorithmic development, computer vision for autonomous vehicles has benefited from technological advances in parallel computing architecture [4]. The possibility of performing network training and inference on graphics processing units (GPUs) has made large and complex networks computationally affordable and testable on real-life problems. Due to their high efficiency, GPUs have become a common hardware choice in the automotive industry for on-vehicle deep learning inference.

Going beyond GPUs, as embedded computer vision-based systems are being developed and deployed in an emerging number of industries, including automotive, healthcare and surveillance, deep learning hardware accelerators are also utilizing field-programmable gate arrays (FPGAs) and application specific integrated circuits (ASICs). Such accelerators are also exploiting the possibility to parallelize the vast number of operations, thereby reducing latency and increasing throughput. In contrast to ASICs, for which the design cannot be modified upon fabrication, FPGAs are flexible and reconfigurable, and commonly used for prototyping and validating ASIC implementations. To fairly compare the performance and efficiency between GPUs and FPGAs is very difficult due to the strong dependence on the details of, for example, the implementation, open software support, data transfers and hardware design. See [5] for a recent survey on efficient semantic segmentation networks using GPUs.

In this paper, we investigate the possibility of exploiting FPGAs as a low-power, inference-optimized alternative to GPUs. By applying aggressive filter-reduction and quantization of the model bit precision at training time, and by introducing a highly optimized firmware implementation of convolutional layers, we achieve the compression required to fit semantic segmentation models on FPGAs. We do so by exploiting and improving the `hls4m1` library, which provides an automatic conversion of a given Deep Neural Network into C++ code, which is given as input to a high level synthesis (HLS) library. The HLS library then translates this into FPGA firmware, to be deployed on hardware. Originally developed for scientific applications in particle physics that require sub-microsecond latency [6–12], `hls4m1` has been successfully applied outside the domain of scientific research [13, 14], specifically in the context of tinyML applications [15].

Applying model compression at training time is crucial in order to minimize resource-consumption and maximize the model accuracy. To do so, we rely on quantization-aware training (QAT) through the QKeras [16] library, which has been interfaced to `hls4m1` in order to guarantee an end-to-end optimal training-to-inference workflow [13].

As a baseline, we start from the ENet [17] architecture, designed specifically to perform pixel-wise semantic segmentation for tasks requiring low latency operations. We modify the architecture, removing resource-consuming asymmetric convolutions, and dilated or strided convolutions. In addition, we apply filter ablation and quantization at training time. Finally, we optimize the implementation of convolutional layers in `hls4m1` in order to significantly reduce the resource consumption. With these steps, we obtain a good balance between resource utilization and accuracy, enabling us to deploy the whole network on a Xilinx ZCU102 evaluation board [18].

This paper is organized as follows: The baseline dataset and model are described in sections 2 and 3, respectively. The model compression and the specific optimization necessary to port the compressed model to the FPGA are described in sections 4 and 5. Conclusions are given in section 6.

2. Dataset

Our experiments are performed using the Cityscapes dataset [19], which involves 5000 traffic scene images collected in 50 different cities with varying road types and seasons. These images have fine-grained semantic segmentation annotations with pixel-level classification labels. We have limited ourselves to the four semantic classes Road, Car, Person and Background. According to the standard Cityscapes split, 2975 images are used for training, 500 for validation and 1525 for testing. We crop and resize the original images to have an input resolution of 240×152 pixels. As a pre-processing step, we normalize all pixel values (integer values in the range $[0, 255]$) to be in the $[0, 1]$ range by dividing each one by 256. In this way all inputs are smaller than one and can be represented by a fixed-point datatype using only 8 bits ($\log_2(256)$) (see section 4). An example image from the dataset is shown in figure 1, together with a visualization of its semantic segmentation mask.

For evaluation metrics we use two typical figures of merit for semantic segmentation:

- The model accuracy (Acc), defined as $\text{Acc} = \frac{TP+TN}{TP+TN+FP+FN}$, where TP , TN , FP , and FN are the fraction of true positives, true negatives, false positives, and false negatives, respectively.
- The mean of the class-wise Intersection over Union (mIoU), i.e. the average across classes of the Intersection-Over-Union (defined as $\text{IOU} = \frac{TP}{TP+FP+FN}$).



Figure 1. An downsampled image from the Cityscapes dataset (left) and the corresponding semantic segmentation target (right), in which the pixels belong to one of the classes {background (blue), road (teal), car (yellow), person (red)}.

Table 1. Model architecture parametrized by the number of filters in the bottlenecks f_i , with $i = 1, \dots, 5$.

Layer	Type	Output resolution
Initial	Downsample	$f_0 \times 120 \times 76$
3 × bottleneck 1	Downsample	$f_1 \times 60 \times 38$
3 × bottleneck 2	Downsample	$f_2 \times 30 \times 19$
3 × bottleneck 3		$f_3 \times 30 \times 19$
3 × bottleneck 4	Upsample	$f_4 \times 60 \times 38$
3 × bottleneck 5	Upsample	$f_5 \times 120 \times 76$
Final	Upsample	$4 \times 240 \times 152$

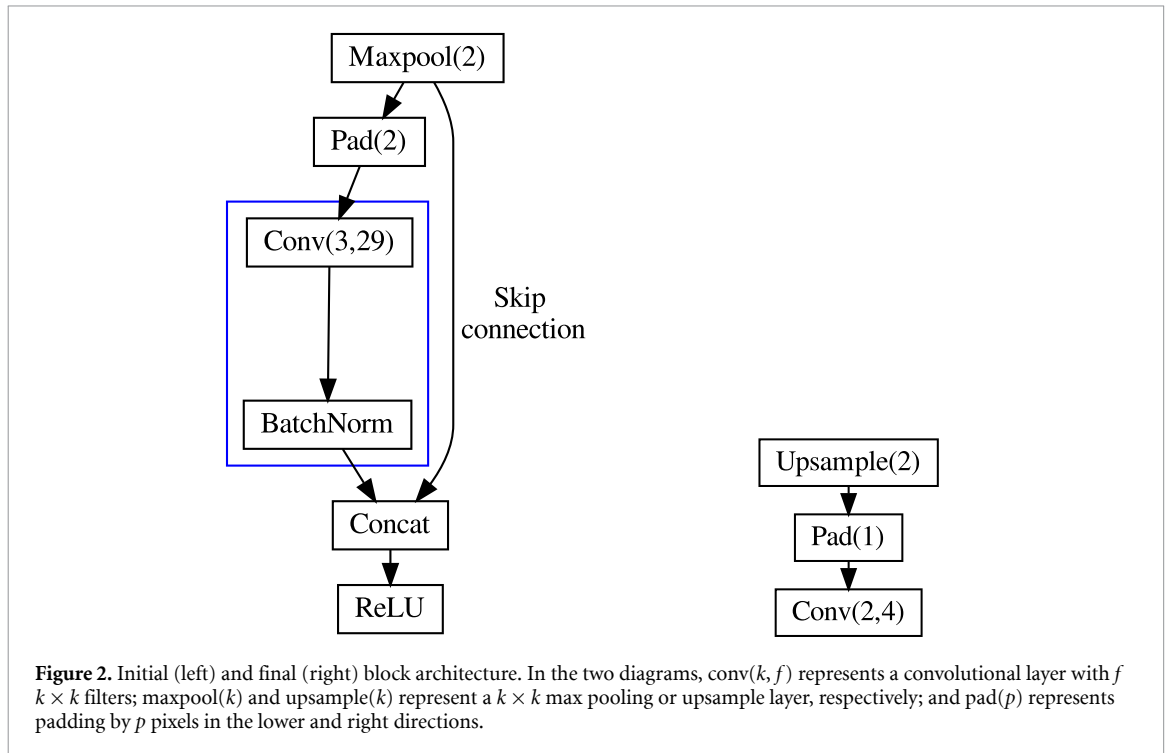
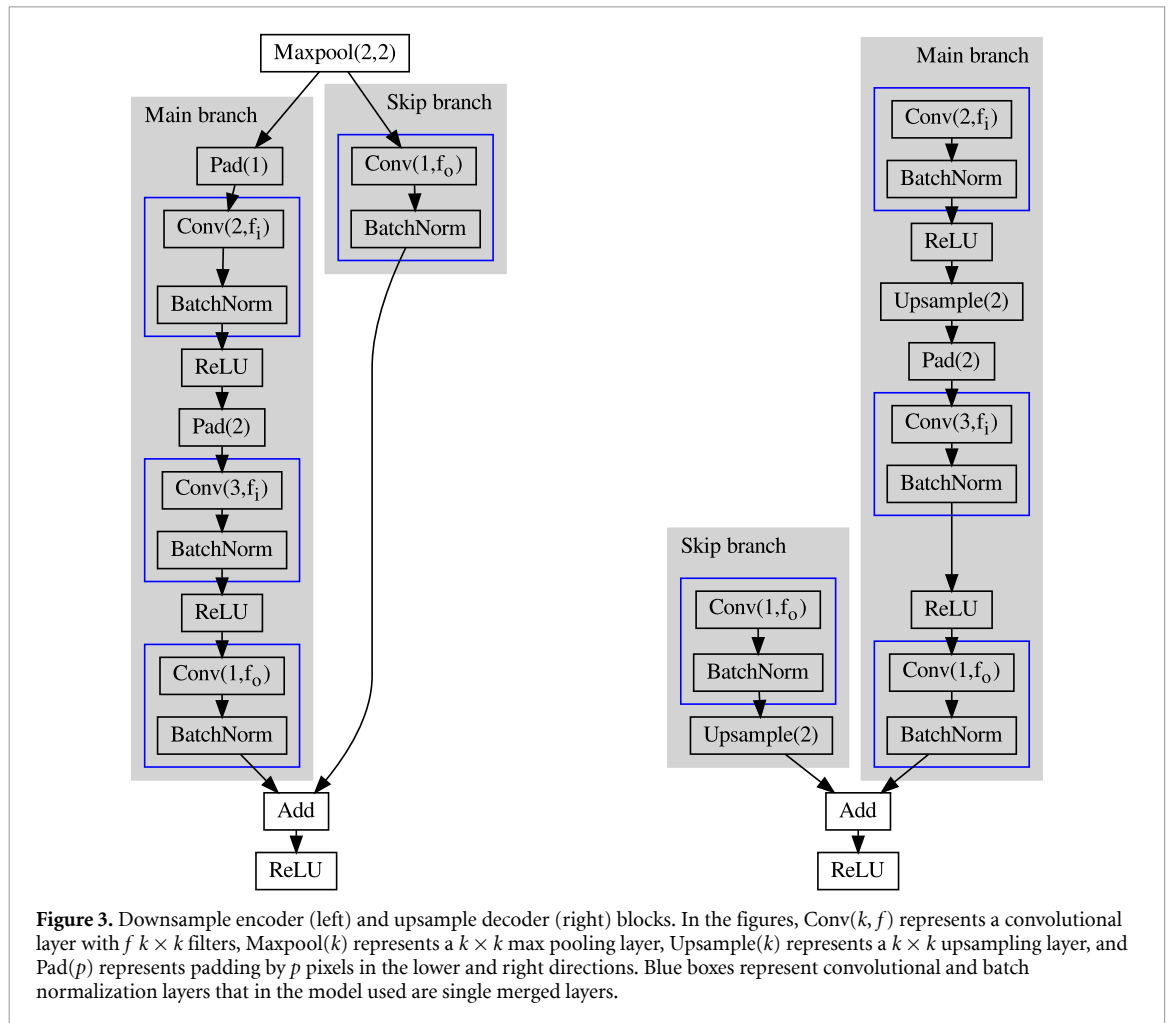


Figure 2. Initial (left) and final (right) block architecture. In the two diagrams, $\text{conv}(k, f)$ represents a convolutional layer with f $k \times k$ filters; $\text{maxpool}(k)$ and $\text{upsample}(k)$ represent a $k \times k$ max pooling or upsample layer, respectively; and $\text{pad}(p)$ represents padding by p pixels in the lower and right directions.

3. Baseline model

The architecture we use is inspired by a fully convolutional residual network called Efficient Neural Network (ENet) [17]. This network was designed for low latency and minimal resource usage. It is designed as a sequence of blocks, summarized in table 1. The initial block, shown in the left figure in figure 2, encodes the input into a $32 \times 120 \times 76$ tensor, which is then processed by a set of sequential blocks of bottlenecks. The first three blocks constitute the *downsampling encoder*, where each block consists of a series of layers as



summarized in the left diagram in figure 3. The final two blocks provide an *upsampling decoder*, as illustrated in the right diagram in figure 3. The final block is shown in the right diagram of figure 2.

Some differences from the original architecture in [17] is that we do not use asymmetric, dilated, or strided convolutions. To further reduce the resource usage, we use three bottlenecks per block instead of five, and we merge convolutional layers with batch normalization layers by rescaling convolutional filter weights with batch normalization parameters (implemented through a QConv2DBatchnorm layer). When we use QAT, this allows us to directly quantize the merged weights during the forward pass, rather than quantizing the batch normalization parameters and the convolutional filters separately. This merging of layers saves resources on the FPGA, since only the merged weights are used. Performing the merging already during training, ensures that the weights used during training and during inference are quantized the same way. The baseline ENet model is obtained fixing the six f hyperparameters of table 1 to (32, 64, 64, 64, 128, 48). This choice results in an architecture with 1.1×10^6 parameters, yielding a mIoU = 63.2% and an accuracy of 91.5%.

Note that, in a real world application, such as autonomous driving, this computer vision task of performing frame-by-frame semantic segmentation constitutes only a sub-task in the full software stack. For example, the outputs of this network will typically need to be transformed into 3d world coordinates and tracked over time before being sent to the planning and decision-making modules. Hence, the single-frame-based metrics, such as accuracy and mIoU, are primarily used to compare different semantic segmentation models, but they are insufficient for gauging the performance of the full autonomous driving stack in real world driving.

4. Model compression

We consider two compression techniques for the model at hand: filter-wise homogeneous pruning, obtained by reducing the number of filters on all the convolutional layers; and quantization, i.e. reducing the number

Table 2. Architecture reduction through internal filter ablation and corresponding performance. As a reference, the baseline architecture is reported on the first row. Highlighted in bold the three models considered further in this work.

Model name	f_i	f_1	f_2	f_3	f_4	f_5	Parameters	mIoU (%)	Accuracy (%)
Enet	32	64	64	64	128	48	1.1×10^6	63.2	91.5
Enet16	32	16	16	16	16	16	5×10^4	54.3	87.9
Enet12	32	12	12	12	12	12	3×10^4	52.0	86.8
Enet8	32	8	8	8	8	8	1.4×10^4	49.4	85.6
Enet6	32	6	6	6	6	6	9×10^3	45.9	84.0
Enet4	32	4	4	4	4	4	5×10^3	36.6	81.5

of bits allocated for the numerical representation of the network components and the output of each layer computation.

In addition, we use the AutoQKeras library [13], distributed with QKeras, to optimize the numerical representation of each component at training time as a hyperparameter. This is done using a mathematical model of the inference power consumption as a constraint in the loss function.

4.1. Filter multiplicity reduction

Normally, network pruning consists of zeroing specific network parameters that have little impact on the model performance. This could be done at training time or after training. In the case of convolutional layers, a generic pruning of the filter kernels would result in sparse kernels. It would then be difficult to take advantage of pruning during inference. To deal with this, filter ablation (i.e. the removal of an entire kernel) was introduced [20]. When filter ablation is applied, one usually applies a restructuring algorithm (e.g. Keras Surgeon [21]) to rebuild the model into the smaller-architecture model that one would use at inference. In this work, we take a simpler (and more drastic) approach: we treat the number of filters in the convolutional layers as a single hyperparameter, fixed across the entire network. We then reduce its value and repeat the training, looking for a good compromise between accuracy and resource requirements.

We repeat the procedure with different target filter multiplicities. The result of this procedure is summarized in table 2, where different pruning configurations are compared to the baseline Enet model.

Out of these models, we select two configurations that would be affordable on the FPGA at hand: a four-filters (Enet4) and an eight-filter (Enet8) configuration. As a reference for comparison, we also consider one version with 16 filters, Enet16, despite it being too large to be deployed on the FPGA in question. We then proceed by quantizing these models through QAT to further reduce the resource consumption.

4.2. Homogeneous quantization-aware training

Homogeneous QAT consists of repeating the model training while forcing the numerical representation of its weight and activation functions to a fixed $\langle T, I \rangle$ precision, where T is the total number of bits and I is the number of integer bits. This is done using the *straight-through estimator*, where quantization functions are applied to weights and activations during the forward pass of the training, but then assuming the quantization is the identity function in the backward pass, as the quantization function is not differentiable. The model training then converges to a minimum that might not be the absolute minimum of the full-precision training, but that would minimize the performance loss once quantization is applied. For a complete overview on quantization methods for neural networks, see [22].

In practice, we perform a homogeneous QAT replacing each layer of the model with its QKeras equivalent and exploiting the existing QKeras-to-hls4m1 interface for FPGA deployment.

We study the impact of QAT for $T \in 2, 4, 8$ with $I = 0$, on the pruned models described above (Enet4, Enet8 and Enet16). The resulting performance is shown in table 3, where we label the three quantization configurations as Q2, Q4 and Q8, respectively.

The resulting resource utilization for Enet4 and Enet8 falls within the regime of algorithms that we could deploy on the target FPGA. We observe similar drops in accuracy when going from full precision to Q8 and from Q4 to Q2, but little differences between the Q4 and Q8 models. In this respect, Q4 would offer a better compromise between accuracy and resources than Q8.

Out of these, the models with the highest accuracy and mIoU that would be feasible to fit on the FPGA, is the 8 filter model quantized to 8 bits (Enet8Q8) and the 8 filter model quantized to 4 bits (Enet8Q4).

The quantization of the model does not have to be homogeneous across layers. In fact, it has been demonstrated that a heterogeneous quantization is the best way to maintain high accuracy at low resource-cost [23]. We therefore define one final model with an optimized combination of quantizers.

Table 3. Homogeneously and heterogeneously quantized models with indicated bitwidth, filter architecture and number of parameters, together with their validation mean IOU trained with quantization aware training using QKeras. The corresponding values before quantization (from table 2) are also reported in the three first rows.

Model name	Quantization	f_i	f_1	f_2	f_3	f_4	f_5	Parameters	mIoU (%)	Accuracy (%)
Enet16	—	32	16	16	16	16	16	5×10^4	54.3	87.9
Enet8	—	32	8	8	8	8	8	1.4×10^4	49.4	85.6
Enet4	—	32	4	4	4	4	4	5×10^3	36.6	81.5
Enet16Q8	8	32	16	16	16	16	16	5×10^4	35.0	79.1
Enet8Q8	8	32	8	8	8	8	8	1.4×10^4	33.4	77.1
Enet4Q8	8	32	4	4	4	4	4	5×10^3	13.6	53.8
Enet16Q4	4	32	16	16	16	16	16	5×10^4	34.1	77.9
Enet8Q4	4	32	8	8	8	8	8	1.4×10^4	33.9	77.6
Enet4Q4	4	32	4	4	4	4	4	5×10^3	13.5	53.6
Enet16Q2	2	32	16	16	16	16	16	5×10^4	27.4	68.6
Enet8Q2	2	32	8	8	8	8	8	1.4×10^4	28.7	71.1
Enet4Q2	2	32	4	4	4	4	4	5×10^3	13.4	53.5
EnetHQ	Heterogeneous	8	2	4	8	4	3	5.3×10^3	36.8	81.1

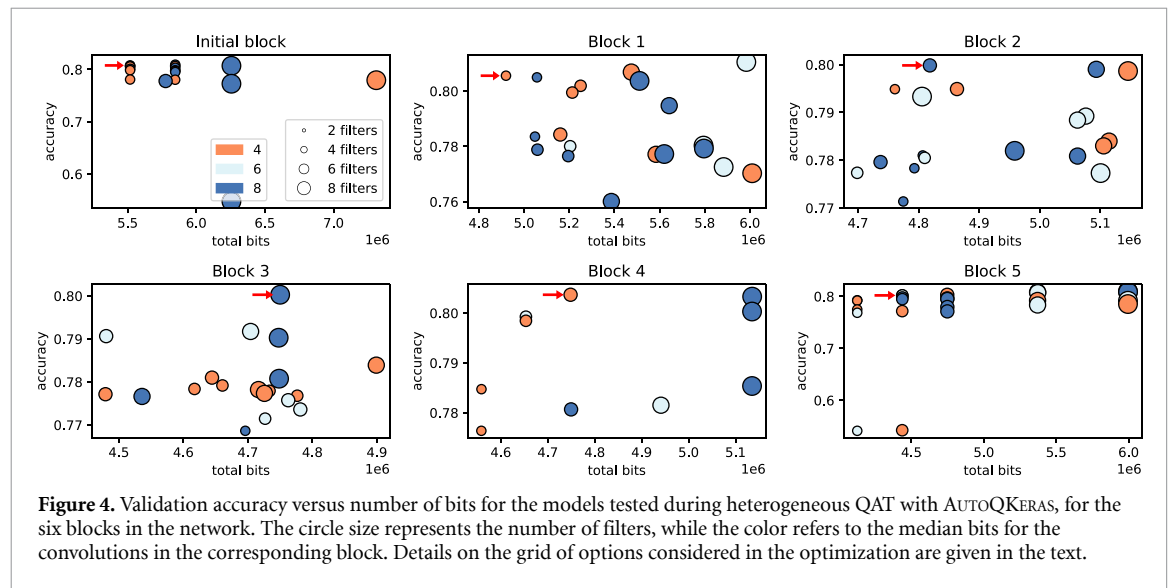


Figure 4. Validation accuracy versus number of bits for the models tested during heterogeneous QAT with AUTOQKERAS, for the six blocks in the network. The circle size represents the number of filters, while the color refers to the median bits for the convolutions in the corresponding block. Details on the grid of options considered in the optimization are given in the text.

4.3. Heterogeneous quantization aware training

Heterogeneous QAT consists in applying different quantization to different network components. For deep networks, one typically deals with the large number of possible configurations by using an optimization library. In our case, we use AUTOQKERAS [13]. In AUTOQKERAS, a hyperparameter search over individual layer quantization conditions and filter counts is performed. Since the model contains skip connections, the scan over number of filters needs to be handled with care. In particular, we use the block features of AUTOQKERAS to ensure that the filter count matches throughout a bottleneck, so that the tensor addition of the skip connection will have valid dimensions.

The search for best hyperparameters, including the choice of individual quantizers for kernels and activations, is carried out using a Bayesian strategy where the balance between accuracy and resource usage is controlled by targeting a metric derived from them both [13]. In our search we permit e.g. a 4% decrease in accuracy if the resource usage also is halved at the same time.

The hyperparameter scan is done sequentially over the blocks, i.e. the Bayesian search over quantization and filter count of the initial layer is performed first and is then frozen for the hyperparameter scan of the first bottleneck and so on. The rest of the model is kept in floating point until everything in the end is quantized.

Figure 4 shows the outcome of the heterogeneous QAT, in terms of validation accuracy and total number of bits for the six blocks in the network. The optimal configuration search is performed taking as a baseline the Enet4 model, scanning the kernel bits in $\{4, 8\}$ and fixing the number of kernels to four times a by-layer multiplicative chosen in $\{0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0\}$. The optimal configuration (EnetHQ) is obtained for $f_i = 8, f_1 = 2, f_2 = 4, f_3 = 8, f_4 = 4,$ and $f_5 = 3$, resulting in 4.7×10^3 parameters, a mIoU = 36.8% and an accuracy of 81.1%. Out of all the quantized models, both homogeneous and heterogeneous, this is the one which performs the best.

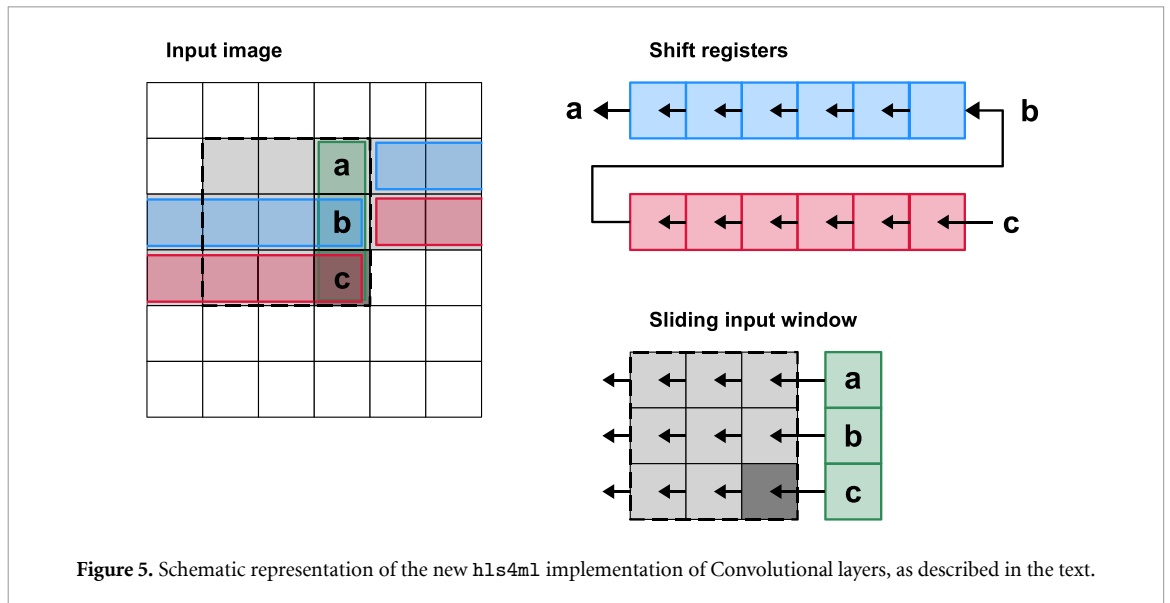


Table 4. Comparison of previous and proposed hls4ml implementation of the convolutional layer, in terms of relative reduction of resource utilization. The estimates are made targeting an xczu9eg-2ffvb1156 MPSoC device on a ZCU102 development kit.

Implementation	BRAM	DSP	FF	LUT
Encoded [14]	4752	5632	195 344	291 919
Line buffer	4064	5632	176 620	305 494
Improvement	−15%	0%	−1%	+5%

5. FPGA implementation, deployment and results

5.1. Resource-efficient convolution algorithm

The hls4ml library has an implementation of convolutional layers that is aimed at low-latency designs [14]. However, this implementation comes at the expense of high resource utilization. This is due to the number of times pixels of the input image are replicated to maintain the state of a sliding input window. For convolutional layers operating on wider images, like in our case, this overhead can be prohibitively large. In order to reduce the resource consumption of the convolutional layers of the model, we introduce a new algorithm that is more resource efficient.

The new implementation, dubbed ‘line buffer’, uses shift registers to keep track of previously seen pixels. The primary advantage of the line buffer implementation over the previous one is the reduction of the size of the buffer needed to store the replicated pixels. For an image of size $H \times W$, with a convolution kernel of size $K \times L$, the line buffer allocates $K - 1$ buffers (chain of shift registers) of depth W for the rows of the image, while the previous implementation allocates K^2 buffers of depth $K \times (W - K + 1)$ for the elements in the sliding input window.

The algorithm is illustrated on figure 5. Initially, each new pixel read from the input image stream is pushed into the shift register chain. If the shift register is full, the first element will be popped and it will be pushed into the next shift register in chain. The process is repeated for all $K - 1$ shift registers in the chain. The popped pixels are stacked with the input pixel into a column vector and are pushed as the rightmost column of the input window. The pixels popped from the leftmost column of the input window are not used further. In our implementation, the propagation of new pixels through the shift register chain and the insertion into the sliding input window are completed in a single clock cycle, making the implementation as efficient as the existing hls4ml implementation.

To compute the output from the populated sliding input window, we rely on the existing routines of hls4ml. We rely on a set of counters to keep track of input window state to know when to produce an output. The algorithm for maintaining the chain of shift registers and populating the sliding input window can be adapted for use in the pooling layers as well.

To compare the two implementations, we consider the resource utilization of an ENet bottleneck block consisting of 8 filters, implemented using either method. The results are summarized in table 4. We observe a substantial reduction in block random access memory (BRAM) usage, at the price of a small increase in look-up table (LUT) utilization.

Table 5. Effect of FIFO depth optimization on FPGA resource usage and model latency. The values in the table are taken from Vivado HLS estimates of resource usage. A comparison using physical resource usage is unfeasible since the model without optimization cannot be synthesized. The estimates are made targeting an xczu9eg-2ffvb1156 MPSoC device on a ZCU102 development kit.

Optimisation	BRAM	LUT	FF	DSP	Latency
No	7270	676 760	230 913	228	3.577 ms
Yes	1398	437 559	146 392	228	3.577 ms
Improvement	−81%	−35%	−37%	0%	0%

Table 6. Accuracy, mIoU, latency and resource utilization for the EnetHQ, Enet8Q4 and Enet8Q8 models. The latency is quoted for a batch size $b = 1$ and $b = 10$. Resources are expressed as a percentage of those available on the xczu9eg-2ffvb1156 MPSoC device on the ZCU102 development kit. The last row is a comparison to work presented in [24].

Model	Acc.	mIoU	Latency (ms)		BRAM	LUT	FF	DSP
			$b = 1$	$b = 10$				
EnetHQ	81.1%	36.8 %	4.9	30.6	224.5 (25%)	76 718 (30%)	87 059 (16%)	450 (18%)
Enet8Q4	77.6%	33.9 %	4.8	30.2	342.0 (37%)	166 741 (61%)	90 536 (16%)	0
Enet8Q8	77.1%	33.4 %	4.8	30.0	508.5 (56%)	126 458 (46%)	134 385 (25%)	1502 (60%)
ENet [24]	—	63.1%	30.38 (720) ^a	—	257	62 599	192 212	689

^a The former is without considering data transfer, pre- and post-processing. The number in parenthesis includes these additional overheads, averaged over 58 images, and is more comparable to the numbers we present.

5.2. FIFO depth optimization

With the dataflow compute architecture of hls4ml, layer compute units are connected with FIFOs, implemented as memories in the FPGA. These FIFOs contribute to the overall resource utilisation of the design. The read and write pattern of these FIFOs depends on the dataflow through the model, which is not predictable before the design has been scheduled by the HLS compiler, and is generally complex. With previous hls4ml releases, these memories have therefore been assigned a depth corresponding to the dimensions of the tensor in the model graph as a safety precaution.

To optimize this depth and thereby reduce resource consumption, we implemented an additional step in the compilation of the model to hardware. By using the clock-cycle accurate RTL simulation of the scheduled design, we can monitor the actual occupancy of each FIFO in the model when running the simulation over example images. This enables us to extract and set the correct size of the FIFOs, reducing memory usage compared to the baseline.

By applying this procedure, we observe a memory efficiency $\frac{\sum_l O_l}{\sum_l F_l} = 19.5\%$, where the index l runs across the layers, O_l is the observed occupancy for the l th layer, and F_l is the corresponding FIFO depth. The corresponding mean occupancy is found to be $\sum_l \frac{O_l}{F_l} = 4.7\%$.

We then resize every FIFO to its observed maximum occupancy and rerun the C-Synthesis, thereby saving FPGA resources and allowing larger models to fit on the FPGA. Table 5 shows the impact of such an optimization on the FPGA resources for one example model, Enet8Q8, demonstrating a significant reduction of resources, which are BRAM, LUT, flip-flop (FF), digital signal processor (DSP).

5.3. Results

The hardware we target is a Zynq UltraScale+ MPSoC device (xczu9eg-2ffvb1156) on a ZCU102 development kit, which targets automotive applications. After reducing the FPGA resource consumption through the methods described above, the highest accuracy models highlighted in table 3 are synthesized. These are the homogeneously quantized Enet8Q8 and Enet8Q4 models, as well as the heterogeneously quantized EnetHQ model. To find the lowest latency implementation, we run several attempts varying the reuse factor (RF) and the clock period. The RF indicates how many times a multiplier can be reused (zero for a fully parallel implementation). Lower RF leads to lower latency, but higher resource usage. We targeted reuse factors of 10, 20, 50, 100, and clock periods of 5, 7, 10 ns. For each model, we then chose the configuration yielding the lowest latency. For Enet8Q8, this is a target clock period of 7 ns and RF = 10. For Enet8Q4 and EnetHQ we use a clock period of 7 ns and RF = 6.

Inference performance of this model was measured on the ZCU102 target device. The final latency and resource utilization report is shown in table 6.

We measured the time taken by the accelerator to produce a prediction on batches of images, with batch sizes of $b = 1$ and $b = 10$. The same predictions have been executed 10^5 times, and the time average is taken as the latency. The single image latency (batch size of 1) is 4.8–4.9 ms for all three models. Exploiting the data

flow architecture, the latency to process images in a batch size of 10 is less than 10 times the latency observed for a batch size of 1. While in a real-world deployment of this model the latency to return the predictions of a single image is the most important metric, a system comprised of multiple cameras may be able to benefit from the speedup of batched processing by batching over the images captured simultaneously from different cameras. The model with the highest accuracy and lowest resource consumption is the heterogeneously quantized EnetHQ model. This model has an mIoU of 36.8% and uses less than 30% of the total resources.

Similar work on performing semantic segmentation on FPGAs include [24] and a comparison is given in table 6. Here, the original ENet model [17] is trained and evaluated on the Cityscapes dataset, and then deployed on a Xilinx Zynq 7035 FPGA using the Xilinx Vitis AI Deep Learning Processor Unit (DPU). There are some crucial differences between the approach taken here and that of [24]. In order to achieve the lowest possible latency, we implement a fully on-chip design with high layer parallelism. We optimize for latency, rather than frame rate, such that in a real-life application the vehicle response time could be minimized. Keeping up with the camera frame rate is a minimal requirement, but a latency lower than the frame interval can be utilized. In our approach, each layer is implemented as a separate module and data is streamed through the architecture layer by layer. Dedicated per-layer buffers ensure that just enough data is buffered in order to feed the next layer. This is highly efficient, but limits the number of layers that can be implemented on the FPGA. Consequently, in order to fit onto the FPGA in question, our model is smaller and achieves a lower mIoU. Jia *et al* [24] does not quote a latency, but a frame rate. A best-case latency is then computed as the inverse of this frame rate, which corresponds to 30.38 ms. However, this does not include any overhead latency like data transfer, pre- and post-processing. Including these, the average time per image increases to 720 ms.

6. Conclusions

In this paper, we demonstrate that we can perform semantic segmentation on a single FPGA on a Zynq MPSoC device using a compressed version of ENet. The network is compressed using automatic heterogeneous quantization at training time and a filter ablation procedure, and is then evaluated on the Cityscapes dataset. Inference is executed on hardware with a latency of 4.9 ms per image, utilizing 18% of the DSPs, 30% of the LUTs, 16% of the FFs and 25 % of the BRAMs. Processing the images in batches of ten results in a latency of 30 ms per batch, which is significantly faster than ten times the single-image batch inference latency. This is relevant when batching over images captured from different cameras simultaneously. By introducing an improved implementation of convolutional layers in `hls4ml`, we significantly reduce resource consumption, allowing for a fully-on-chip deployment of larger convolutional neural networks. This avoids latency overhead caused by data transfers between off-chip memory and FPGA processing elements, or between multiple devices. Also taking into account the favorable power-efficiency of FPGAs, we conclude that FPGAs offer highly interesting, low-power alternatives to GPUs for on-vehicle deep learning inference and other computer vision tasks requiring low-power and low-latency.

Data availability statement

All data that support the findings of this study are included within the article (and any supplementary files).

Acknowledgments

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. M P, M R, S S and V L are supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant Agreement No. 772369). M P and N G are supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant Agreement No. 966696).

ORCID iDs

Nicolò Ghielmetti  <https://orcid.org/0000-0002-4660-9757>

Vladimir Loncar  <https://orcid.org/0000-0003-3651-0232>

Maurizio Pierini  <https://orcid.org/0000-0003-1939-4268>

Thea Aarrestad  <https://orcid.org/0000-0002-7671-243X>

Jennifer Ngadiuba  <https://orcid.org/0000-0002-0055-2935>

Philip Harris  <https://orcid.org/0000-0001-8189-3741>

References

- [1] Apollinari G et al 2017 *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1* CERN yellow reports: monographs (Geneva: CERN) (<http://dx.doi.org/10.23731/CYRM-2017-004>)
- [2] Garrett M A et al 2010 (arXiv:1008.2871)
- [3] Banbury C R et al 2021 Benchmarking tinyml systems: challenges and direction (arXiv:2003.04821)
- [4] Raina R, Madhavan A and Ng A Y 2009 Large-scale deep unsupervised learning using graphics processors *Proc. 26th Annual Int. Conf. on Machine Learning* (ACM) pp 873–80
- [5] Holder C J and Shafique M 2022 On efficient real-time semantic segmentation: a survey (arXiv:2206.08605)
- [6] Duarte J et al 2018 Fast inference of deep neural networks in FPGAs for particle physics *J. Instrum.* **13** 07027
- [7] Summers S et al 2020 Fast inference of boosted decision trees in FPGAs for particle physics *J. Instrum.* **15** 05026
- [8] Loncar V et al 2021 Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml *Mach. Learn.: Sci. Technol.* **2** 015001
- [9] Iiyama Y et al 2021 Distance-weighted graph neural networks on fpgas for real-time particle reconstruction in high energy physics *Front. Big Data* **3** 598927
- [10] Heintz A et al 2020 Accelerated charged particle tracking with graph neural networks on FPGAs *3rd Machine Learning and the Physical Sciences Workshop at the 34th Annual Conf. on Neural Information Processing Systems* vol 12
- [11] Francescato S Giagu S, Riti F, Russo G, Sabetta L and Tortonesi F 2021 Model compression and simplification pipelines for fast deep neural network inference in FPGAs in HEP *Eur. Phys. J. C* **81** 969
Francescato S Giagu S, Riti F, Russo G, Sabetta L and Tortonesi F 2021 *Eur. Phys. J. C* **81** 1064 (erratum)
- [12] Sun C, Nakajima T, Mitsumori Y, Horii Y and Tomoto M 2022 Fast muon tracking with machine learning implemented in FPGA (arXiv:2202.04976)
- [13] Coelho C N Kuusela A, Li S, Zhuang H, Ngadiuba J, Aarrestad T K, Loncar V, Pierini M, Pol A A and Summers S 2021 Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors *Nat. Mach. Intell.* **3** 675–86
- [14] Aarrestad T et al 2021 Fast convolutional neural networks on FPGAs with hls4ml *Mach. Learn.: Sci. Technol.* **2** 045015
- [15] Fahim F et al 2021 hls4ml: an open-source codesign workflow to empower scientific low-power machine learning devices *TinyML Research Symp. 2021* vol 3
- [16] Coelho C et al 2019 Qkeras (available at: <https://github.com/google/qkeras>)
- [17] Paszke A, Chaurasia A, Kim S and Culurciello E 2016 Enet: a deep neural network architecture for real-time semantic segmentation (arXiv:1606.02147)
- [18] Xilinx ZCU102 evaluation board (available at: www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html#information)
- [19] Cordts M et al 2016 The cityscapes dataset for semantic urban scene understanding (arXiv:1604.01685)
- [20] Girshick R, Donahue J, Darrell T and Malik J 2014 Rich feature hierarchies for accurate object detection and semantic segmentation (arXiv:1311.2524)
- [21] Whetton B 2016 Keras surgeon (available at: <https://github.com/BenWhetton/keras-surgeon>)
- [22] Gholami A et al 2021 A survey of quantization methods for efficient neural network inference (arXiv:2103.13630)
- [23] Coelho C N, Kuusela A, Li S, Zhuang H, Ngadiuba J, Aarrestad T K, Loncar V, Pierini M, Pol A A and Summers S 2021 Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors *Nat. Mach. Intell.* **3** 675–86
- [24] Jia W, Cui J, Zheng X and Wu Q 2021 Design and implementation of real-time semantic segmentation network based on FPGA *2021 7th Int. Conf. on Computing and Artificial Intelligence, ICCAI* (New York: Association for Computing Machinery) pp 321–5

PAPER • OPEN ACCESS

Lightweight jet reconstruction and identification as an object detection task

To cite this article: Adrian Alan Pol *et al* 2022 *Mach. Learn.: Sci. Technol.* **3** 025016

View the [article online](#) for updates and enhancements.

You may also like

- [Atom cloud detection and segmentation using a deep neural network](#)
Lucas R Hofer, Milan Krstajić, Péter Juhász et al.
- [DIM: long-tailed object detection and instance segmentation via dynamic instance memory](#)
Zhao-Min Chen, Xin Jin, Xiaoqin Zhang et al.
- [Deep learning in electron microscopy](#)
Jeffrey M Ede



PAPER

OPEN ACCESS

RECEIVED

15 February 2022

REVISED

14 June 2022

ACCEPTED FOR PUBLICATION

17 June 2022

PUBLISHED







4 July 2022

Original Content from this work may be used under the terms of the [Creative Commons Attribution 4.0 licence](#).

Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.



Lightweight jet reconstruction and identification as an object detection task

Adrian Alan Pol^{1,2,*} , Thea Aarrestad¹ , Ekaterina Govorkova¹ , Roi Halily³, Anat Klempner³, Tal Kopetz³, Vladimir Loncar^{1,4} , Jennifer Ngadiuba⁵ , Maurizio Pierini¹ , Olya Sirkin³ and Sioni Summers¹

¹ European Organization for Nuclear Research (CERN), Geneva, Switzerland

² Princeton University, Princeton, NJ, United States of America

³ CEVA Inc., Herzliya, Israel

⁴ Institute of Physics Belgrade, Belgrade, Serbia

⁵ Fermi National Accelerator Laboratory, Winfield Township, DuPage County, IL, United States of America

* Author to whom any correspondence should be addressed.

E-mail: adrianalan.pol@cern.ch

Keywords: high energy physics, jet tagging, jet reconstruction, jet images, object detection, attention mechanism, quantization aware training

Abstract

We apply object detection techniques based on deep convolutional blocks to end-to-end jet identification and reconstruction tasks encountered at the CERN large hadron collider (LHC). Collision events produced at the LHC and represented as an image composed of calorimeter and tracker cells are given as an input to a Single Shot Detection network. The algorithm, named PFJet-SSD performs simultaneous localization, classification and regression tasks to cluster jets and reconstruct their features. This all-in-one single feed-forward pass gives advantages in terms of execution time and an improved accuracy w.r.t. traditional rule-based methods. A further gain is obtained from network slimming, homogeneous quantization, and optimized runtime for meeting memory and latency constraints of a typical real-time processing environment. We experiment with 8-bit and ternary quantization, benchmarking their accuracy and inference latency against a single-precision floating-point. We show that the ternary network closely matches the performance of its full-precision equivalent and outperforms the state-of-the-art rule-based algorithm. Finally, we report the inference latency on different hardware platforms and discuss future applications.

1. Introduction

The world's largest and most powerful particle accelerator, the CERN large hadron collider (LHC) [1], operates at a nominal proton-proton collision rate of 40 MHz. Due to storage constraints and technological limitations (e.g. fast enough read-out electronics), the volume of recorded data must be significantly reduced by the experiments operating around the accelerator ring. To this purpose, a set of algorithms collectively referred to as the *trigger system* are typically used to filter the incoming data stream. Trigger algorithms are designed to reduce the rate of recorded collision *events* (e.g. the collection of sensor readouts at each bunch crossing) while preserving the physics reach of the experiments. For example, at the Compact Muon Solenoid (CMS) experiment, the trigger system [2, 3] is structured in two stages using increasingly complex information and more refined algorithms:

- the Level 1 (L1) Trigger, implemented on custom-designed electronics; reduces the 40 MHz input to a 100 kHz rate in $<10 \mu\text{s}$.
- the high level trigger (HLT), a collision reconstruction software running on a computer farm; reduces the 100 kHz rate output of the L1 trigger to 1 kHz in $<150 \text{ ms}$.

With the planned LHC high-luminosity upgrade [4], the number of proton-proton collisions per second will surge approximately four-fold. The latency of legacy reconstruction algorithms will increase by more

than the factor of three as they may suffer from execution time scaling worse than linearly [5]. Along with the computing infrastructure upgrades, it is worth investigating solutions that could execute many tasks at once, while retaining accuracy and benefiting from the additional speedup offered by parallel computing architectures. Deep neural networks, such as those used for computer vision tasks, are an obvious candidate in this endeavour.

The majority of particles produced in LHC events are unstable and immediately decay to lighter particles. The new particles can decay themselves to others in a so-called decay chain. Such a process terminates when the decay products are stable particles, e.g. charged pions. This collimated shower of particles with adjacent trajectories is called a *jet*. Jets are central to many physics studies at the LHC experiments [6–9]. In particular, a successful physics program requires aggregating particles into jets (*jet clustering*), an accurate determination of the jet momentum (*momentum measurement*) and the identification of which particle kind started the shower (*jet tagging*) [10–13].

In this work, we show how jet clustering, momentum measurement, and tagging could all be handled simultaneously on parallel computing architectures. Besides the practical advantages of our approach, one could benefit from multitask learning when accomplishing more tasks at once [14]. For instance, a classifier and a regression running at once can learn that calibration constants depend on the nature of the jet, an issue which is now handled with ad-hoc post-processing [15], i.e. when factorizing the reconstruction problem to energy regression and tagging the overall performance may drop for both. Our main contributions are as follows:

- We introduce the **PFJet-SSD** algorithm to perform localization, classification and additional regression tasks on jets in a single feed-forward pass (concurrently, or *single-shot*). We combine ideas from different fields of deep learning, i.e. object detection, attention mechanisms, network slimming and quantization.
- We report acceleration on different computing architectures.
- We generate and publicly share a dataset of simulated LHC collisions, pre-processed to be suited for computer vision applications similar to those discussed in this work, as well as for point-cloud end-to-end reconstruction. The dataset is available on Zenodo [16] and it is accompanied by annotated jet labels, to be used as ground truth during training.

We use the CMS detector and trigger system as an illustrative example. One could apply the same approach to other detectors, adapting the architecture to the detector granularity and latency constraints. The dataset, instructions, and code to fully reproduce our results are available at <https://github.com/AdrianAlan/PF-Jet-SSD>.

The remainder of this paper is structured as follows. In section 2 we review the key building blocks for this work, i.e. jet images, single-shot detection, attention mechanisms, and efficient model design. In section 3 we introduce the PFJet-SSD model and its quantized variants. In section 4 we describe the dataset and the training procedure. Finally, in sections 5 and 6 we discuss the results and future directions, respectively.

2. Techniques

In this section, we review the background techniques for this work, i.e. jet images, single-shot detection, attention mechanism and designing efficient inference networks with pruning and quantization. We examined architecture suggestions from [17], which lists methods for designing efficient networks for computer vision tasks achieving state-of-the-art results. Some of these methods, e.g. GELU activation layer [18], are currently unsupported by SensPro, our target hardware (see section 5.2). Thus we excluded them from this work but we suggest they are examined in further optimization studies.

2.1. Jet images

Traditional approaches to jet tagging rely on features, such as jet substructure, designed by experts that detect characteristic energy deposit patterns [19–27]. In recent years, several studies applied computer vision for event reconstruction at particle colliders, e.g. [28–43]. This was obtained by projecting the lower level detector measurements of the emanating particles onto a cylindrical detector and then unwrapping the inner surface of the calorimeter on a rectangle. Such information was further interpreted as an image with calorimeter cells as pixels, where pixel intensity maps the energy deposit of the cell, i.e. *jet images*. This approach was also applied to end-to-end reconstruction, considering not just the individual jet but the whole event [44, 45]. Building on these works, we extend the end-to-end reconstruction to include a localization task, merging the jet clustering and classification tasks in a single operation. Centralized computing

environments are the only viable options for this: end-to-end approaches require as input a raw data representation, which is not available with reduced analysis data formats. For this reason, we also consider how the model could be compressed to reduce computing footprint, having in mind an approach optimized for a trigger application.

2.2. Single shot detection

Object detection is a fundamental task in computer vision. It is defined as the classification of objects from predefined categories in the image along with their precise spatial locations. The spatial location and extent of an object can be defined coarsely using a bounding box, which is an axis-aligned rectangle tightly bounding the object. Modern object detection focuses on using primarily convolutional neural networks (CNNs) as the building block. Deep learning object detection achieved state-of-the-art results in tasks such as face [46] or pedestrian detection [47]. For a general survey on this subject, see [48, 49].

Deep-learning-based object detection models are typically divided into one- [50–54] or two-stage [55–59] detectors. Two-stage detectors generate a sparse set of regions with a high probability of an object being present first (region proposals), followed by a simple classification step. This two-step process is inefficient for real-time applications, due to task serialization. Single-step approaches classify and regress object locations concurrently (in a single feed-forward pass) and as such tend to achieve lower accuracy than two-stage detectors but are simpler and significantly more latency and memory efficient, hence having greater applicability to online problems.

The single-shot multibox detector (SSD) [60], is a simple one-stage, anchor-based detector. First, a set of default regions in an image with a fixed shape and size is predefined to discretize the output space of bounding boxes, called anchors. These anchors have a diverse set of shapes to detect objects with different dimensions, i.e. multiple scales and aspect ratios. Based on the ground truth, the object locations are matched with the most appropriate anchors to obtain the supervision signal for the anchor estimation. At inference, each anchor is refined by four box coordinates (width, height, x and y offsets) and predicts the categorical probabilities. To avoid a huge number of negative proposals dominating training gradients, hard negative mining is used to train the network, which fixes the foreground and background ratio [61]⁶. Alternatively, a focal loss [52] could be used. In this case, the price to pay would be more hyperparameters to tune. The SSD architecture is fully convolutional, with initial layers based on a pre-trained backbone architecture, such as VGG-16 [62], followed by extra convolutional and pooling layers which progressively decrease image size and thus increase the receptive field. The information in the last layer may be too coarse spatially to allow precise localization and at the same time, detecting large objects in shallow layers is non-optimal without large enough receptive fields. As a countermeasure for this issue, the SSD performs detection over multiple scales by operating on multiple feature maps, i.e. at different depths of the network. Each of these feature maps is responsible for detecting objects according to their receptive field. To detect large objects and increase receptive fields extra convolutional feature maps were added to the backbone architecture. The final prediction is made by merging all detection results from different feature maps followed by a non-maximum suppression (NMS) [60] step and producing the final detection information. NMS removes duplicate predictions originating from multiple anchors.

2.3. Attention mechanisms

Visual attention gates (AGs), e.g. [63–65], learn to suppress feature activations in irrelevant regions in an input image without additional supervision. At inference, the gates generate soft region proposals to highlight salient features useful for a specific task. Recently, the performance of deep CNNs on visual tasks was improved with scale-aware [66, 67], spatial-aware [68, 69] and channel-wise [70, 71] attention. On the contrary, most of the attention modules inevitably increase model complexity. Efficient channel attention (ECA) gate [72] is a soft attention mechanism that addresses this issue. It avoids dimensionality reduction and captures cross-channel interaction efficiently. ECA gate ω is given by $\omega = \sigma(\mathbf{W} \odot g(y))$, where $y \in \mathbb{R}^C$ is the feature map activation with channels C , g is channel-wise global average pooling, σ is the Sigmoid function and \mathbf{W} is a weight tensor of a 1D convolution of filter size k .

2.4. Quantization

Optimizing deep neural networks for efficient inference is an essential task in modern machine learning pipelines due to limitations presented by edge devices. Models should provide high accuracy with a

⁶ By background we refer to the areas without target objects, i.e. jets.

minimum of computing time and resources. Apart from accelerating inference online, e.g. through parallelization or hardware optimizations, models can be optimized offline, through compression [73].

Network compression [74] is a common technique to reduce the number of operations and model size, energy consumption, and over-training of deep neural networks. As neural network synapses and neurons can be redundant, compression techniques attempt to reduce the total number of them, effectively reducing multipliers. Several approaches have been successfully deployed without much loss in accuracy, including selective removal of parameters based on a particular ranking and regularization, i.e. parameter pruning [75–77], compact network architectures [78–80], and reducing the precision of operations and operands, i.e. quantization [81–88].

It has been observed that reducing the precision of the calculations, i.e. weights and biases, has little impact on performance compared to speedup and resource usage gains. This includes moving away from 32-bit floating-point calculations (or *full-precision*, FP) to fixed points, reducing bit-width and weight sharing. An example of a very aggressive strategy is reducing weight precision to ternary values restricted to $\{-1, 0, 1\}$ only, called ternary weight network (TWN) [89]. The quantization is performed during training, using a straight-through estimator [81], where ternary weights are used during the forward and backward propagation but not during the parameter update. To quantize the full precision weights \mathbf{W} to ternary ones \mathbf{W}^* , TWN uses a threshold value Δ :

$$\mathbf{W}^* = \begin{cases} +1 & \text{if } \mathbf{W} > \Delta \\ 0 & \text{if } |\mathbf{W}| \leq \Delta, \\ -1 & \text{if } \mathbf{W} < -\Delta \end{cases}$$

with approximated solution $\Delta^* \approx 0.7 \cdot E(|\mathbf{W}|)$, where E is the expectation value. To make the network perform well, TWN minimizes the Euclidian distance between \mathbf{W} and \mathbf{W}^* along a non-negative scaling factor α that can be implemented with per-network, per-layer or per-channel granularity, transforming the weights to $\alpha\mathbf{W}^*$. For any Δ the optimal α is computed as: $\alpha_{\Delta}^* = \frac{1}{|\mathbf{I}_{\Delta}|} \sum_{i \in \mathbf{I}_{\Delta}} |\mathbf{W}_i|$, where $\mathbf{I}_{\Delta} = \{i | |\mathbf{W}_i| > \Delta\}$ and $|\mathbf{I}_{\Delta}|$ denotes number of elements in \mathbf{I}_{Δ} .

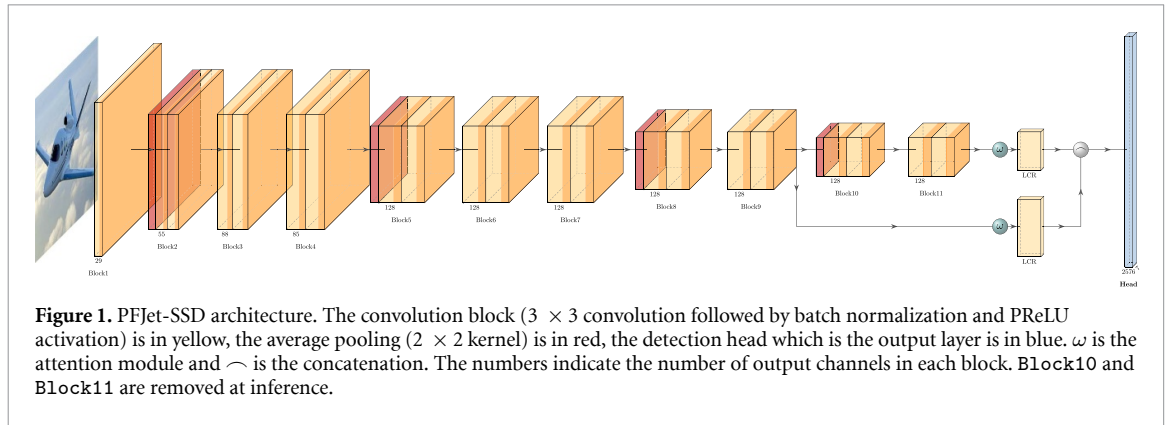
3. Methodology

The PFJet-SSD architecture is shown in figure 1. We modify the original SSD architecture [60] and Jet-SSD architecture proposed in [90]. Having in mind an HLT application with a typical latency of ≈ 150 ms, we extend the event image representation to include the information from the charged-particle reconstruction. We do so by adding a *tracker* channel to the image, in front of the calorimeter channels already introduced in [90]. We use a lightweight MobileNet architecture [78] as a backbone for our detector which replaces the convolution operation with a combination of depthwise and pointwise versions. Each convolution is followed by a batch normalization [91, 92] and parametric rectified linear unit (PReLU) [93] activation layers. We use the AveragePool layer to decrease the size of the feature map. The extra convolutional layers proposed by the original SSD do not contribute to accurate detection (recall the remark about the increasing receptive field from section 2.2). This is due to the size of the jets. As done in [90] we remove these layers already at the training time. Retaining the deeper layers of the backbone, i.e. Block10 and Block11, does not show improvements at inference but is necessary during training due to additional signals during back-propagation. Hence, these deeper layers are only purged after training, i.e. the concatenation layer ignores them only at inference. This alone reduces the number of parameters in the final model by approximately 30%.

We add two new modules to the network. First, the initial convolutional layer is now followed by spatial dropout [94] (with $p = 0.1$). Second, we attach the ECA gate [72] (with $k = 3$) before the localization classification regression (LCR) layer.

The detection head, which is the concatenation of LCR layers, outputs correspond to jet class, localization (η and ϕ offsets) and p_T value (see the definitions in section 4.1). One might easily extend this output to include jet mass regression as well (we left this out for simplicity). Each row in the detection head corresponds to an anchor box, i.e. fixed position in the image. For localization, we regress only the centre of the jet, as we can determine its size from its class. For wide jets we assume $\Delta R = 0.8\text{--}46$ px, for narrow jets $\Delta R = 0.4\text{--}23$ px. This allows us to set only one scale and one aspect ratio for anchors in each feature map which reduces the complexity of the network. The detection head is an input to the NMS layer.

We use magnitude pruning [95] during training to find the optimal allocation of resources between layers. Unstructured pruning generality leads to a higher compression rate and/or higher accuracy when compared to the structured version, but it requires special software or hardware accelerators to fully benefit



from it. Since the outcome of an unstructured pruning is a sparse tensor, one needs a dedicated way to handle sparse memory access on hardware to turn pruning compression into a computational advantage at inference time. We use an alternative, a version of structured pruning that removes whole channels in a convolutional block slimming the network without increasing sparsity. We target the hardware implementation that benefits from fusing batch normalization and convolution parameters at runtime. Doing so, the target filter weights \mathbf{W} of block l are $\mathbf{W}_l = \gamma_l \mathbf{W}_l^{\text{conv}}$, where the \mathbf{W}^{conv} are the weights of the convolution and γ is the scale parameter of the affine transformation of the subsequent batch normalization layer. We thus add a regularizer that pushes the influence of filters down through batch normalization γ L1 penalty, similarly to [77, 96]. We scale this penalty based on the number of operations \mathcal{O} in each layer. The sparsifying regularizer $G(\gamma)$ is calculated as $G(\gamma) = \sum_l |\gamma_l| \mathcal{O}_l$. We mark channels to prune based on the γ distribution in each layer, using the rule: $|\gamma_l| < \mu_{|\gamma_l|} - \sigma_{|\gamma_l|}^2$, where $\mu_{|\gamma_l|} = \frac{1}{N} \sum_{i=1}^N |\gamma_l^i|$, $\sigma_{|\gamma_l|}^2 = \sqrt{\frac{1}{N} \sum_{i=1}^N (|\gamma_l^i| - \mu_{|\gamma_l|})^2}$ and N is the number of channels for each layer. When this rule is not sufficient to remove the specified number of channels we simply select the remaining ones based on ascending magnitudes of γ .

Also during training, we quantize the network to homogeneous 8-bit fixed point precision for both weights and activations and 2-bit TWN with layer- and channel-dependent scaling factors. For the latter, we experimented with a grace period of frozen quantization for which the Δ and α parameters remain unchanged. Training TWN in this manner may offer greater stability, i.e. weights have time to adjust to new parameters, but in our case, the final results did not improve.

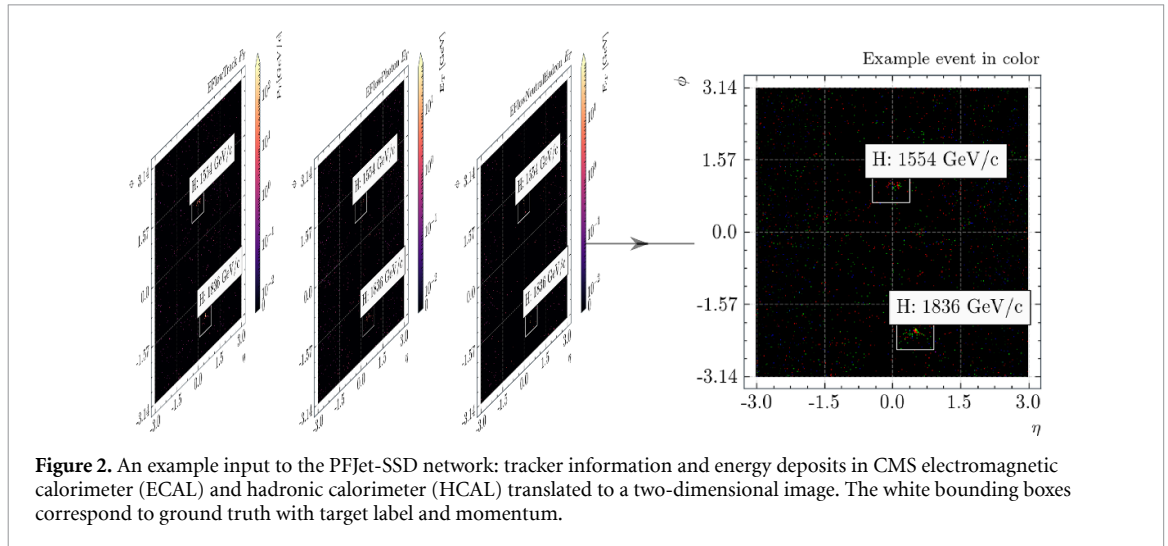
4. Experiments

In this section, we review the experimental dataset and training procedure used for the experiments:

4.1. Dataset

The input dataset consists of 13 TeV proton-proton collision events, in which Randall-Sundrum (RS) gravitons with 3.5 TeV mass are produced. This is a proxy of a sample that would give us jets of various kinds and populating a large spectrum of p_T ranges, i.e. RS gravitons decay to $b\bar{b}$, $g\bar{g}$, $q\bar{q}$, HH , WW , ZZ , or $t\bar{t}$ final states. The choice of this particular process is motivated by the possibility of creating well-defined jet pairs belonging to specific jet classes and with the same kinematic properties across classes. In addition to the hard collision, parasitic *pileup* collisions are also simulated, overlapping minimum bias events. The number of pileup collisions is sampled from a Poisson distribution. We note that this process populates a large spectrum of p_T ranges.

The detector effects and hadronization have an important effect on a jet substructure. Events are generated with Pythia [97]. We use the CMS Delphes [98] description to mimic the effect of detector reconstruction. To apply this algorithm to another detector (e.g. ATLAS), one would have to modify the geometry of the input layer to match the detector geometry. In addition, one would have to repeat the training. Other effects (e.g. theoretical uncertainties related to hadronization models) would be detector independent. These kinds of uncertainties also affect rule-based algorithms and are usually neglected at the trigger stage, where they are subdominant. These uncertainties are measured with data control samples at the analysis stage and, usually, they are mitigated by applying a selection on the offline object so that the trigger behaviour is stable. The same set of state-of-the-art procedures could be applied to the algorithm we present.



Being all this part of a standard data analysis workflow (and beyond the scope of this paper), we do not comment on this further.

The core of the CMS detector is a multi-layer silicon tracking device, operating in a 4 T magnetic field. Two calorimeter layers surround the tracker: the lead tungstate crystal ECAL is designed to stop particles whose main interaction is electromagnetic (photons and electrons); the brass and scintillator HCAL is designed to stop hadrons. They give a measurement of the energy of particles (charged and neutrals). Each of them is composed of a barrel and two endcap sections. Forward calorimeters extend the pseudorapidity (η) coverage provided by the barrel and endcap detectors. The calorimeter cells (towers) in the barrel region together with tracker cells are arranged in a fixed discrete space with fine segmentation in η and ϕ , where ϕ is the translated azimuthal angle. A more detailed description of the CMS detector, together with a definition of the coordinate system used and the relevant kinematic variables, can be found in [99].

Before the LHC, jets were usually reconstructed from their calorimeter deposits (known as CaloJet). With the start of the LHC, the CMS particle flow (PF) algorithm [100] demonstrated that the additional information from track reconstruction could increase the accuracy of jet reconstruction. In CMS, this was crucial to compensate for the poor energy resolution of the HCAL. In the long term, this strategy was found to be effective beyond jet momentum measurement, since the angular resolution of the tracking algorithm provided valuable information for jet tagging and substructure algorithms.

The PF algorithm for jet reconstruction was eventually adopted also by the ATLAS experiment [101]. Taking this as our starting point, we build our event image starting from the PF jet constituents (as returned by the Delphes PF algorithm), arranging the particles in three groups: charged particles, used to create the tracker channel; photons and electrons, used for the ECAL channel; neutral hadrons, used for the HCAL channel. In a real-life application, one could use the same approach or build the channels from the raw detector hits in the tracker, ECAL, and HCAL. The best approach to follow depends on the context of the application (e.g. online vs offline).

We unwrap the cylindrical detector to compose the final image which is formed by translating the calorimeter energy deposits and tracker momentum into pixels using ECAL granularity, which results in $340 \times 360 \times 3$ pixel samples. An example is shown in figure 2. Some previous studies on jet images implemented data pre-processing steps such as translation, rotation, re-pixelation, or inversion. However, in our study, we only limit the input to $\eta \in (-3, 3)$ and standardize pixel intensities.

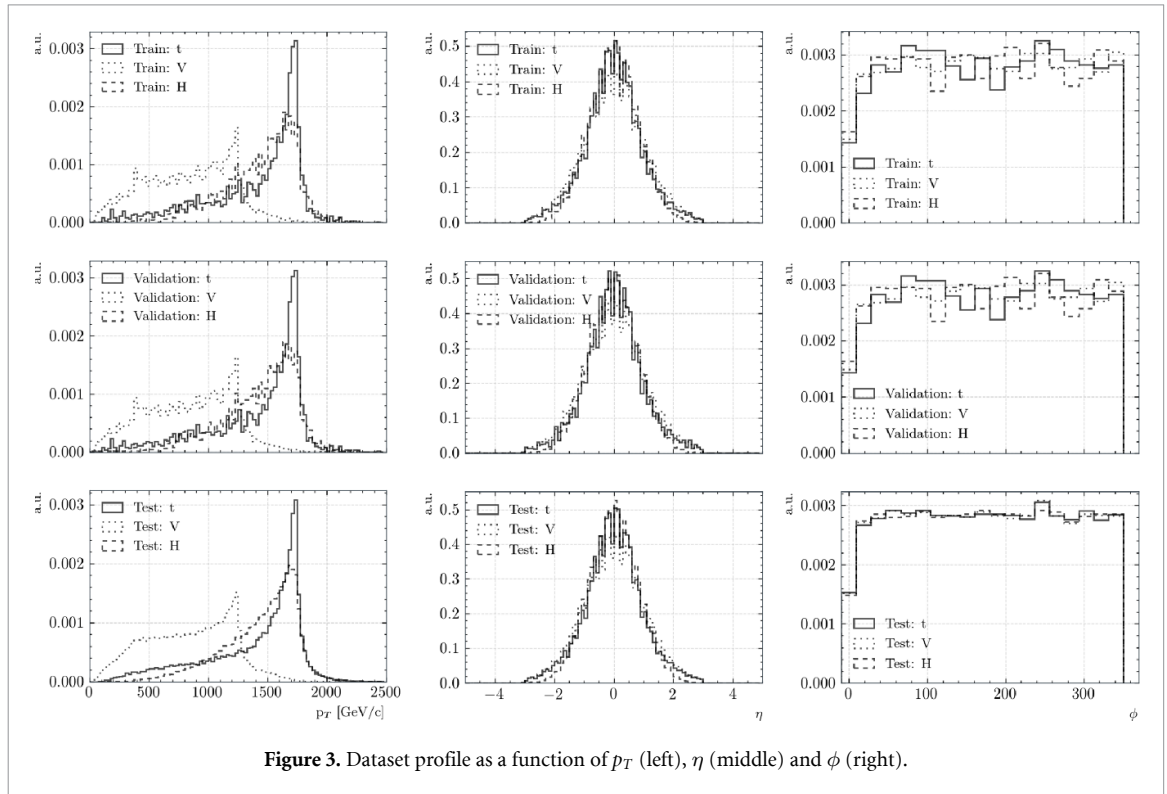
Jet labels are obtained using generator-level information. We assign the jet η (pseudo-rapidity and not rapidity as it is normally done in L1 trigger reconstruction), ϕ and p_T (transverse momentum) measurements to the properties of the same particle. The minimum jet p_T in the dataset is 7 GeV. Details on the dataset profile are given in table 1 which describes the jet statistics across datasets. Figure 3 shows the p_T , η and ϕ distributions.

4.2. Training procedure

The PFJet-SSD network is implemented on Nvidia Tesla GPUs using PyTorch [102]. For training, we use stochastic gradient descent with an initial learning rate of 10^{-3} with momentum set to 0.9 and weight decay to 0.0005. We train the network for 100 epochs with a batch size of 25, decreasing the learning rate by a factor of 2 after every 10 epochs after the 20th epoch. We use 90k and 36k samples for training and validation,

Table 1. Number of samples in the datasets.

	Train	Validation	Test
t jets	59 388	23 802	59 392
W/Z jets	118 701	47 493	118 832
H jets	59 967	23 997	59 978
Σ	238 056 (41.6%)	95 192 (16.7%)	238 202(41.6%)

**Figure 3.** Dataset profile as a function of p_T (left), η (middle) and ϕ (right).

respectively. The training is performed in mixed-precision to speed up computation and distributed across 3 GPUs. Thus, we replace the standard batch normalization layer with the SyncBatchNorm layer provided by PyTorch to synchronize statistics across the machines while training.

We minimize the following cost function:

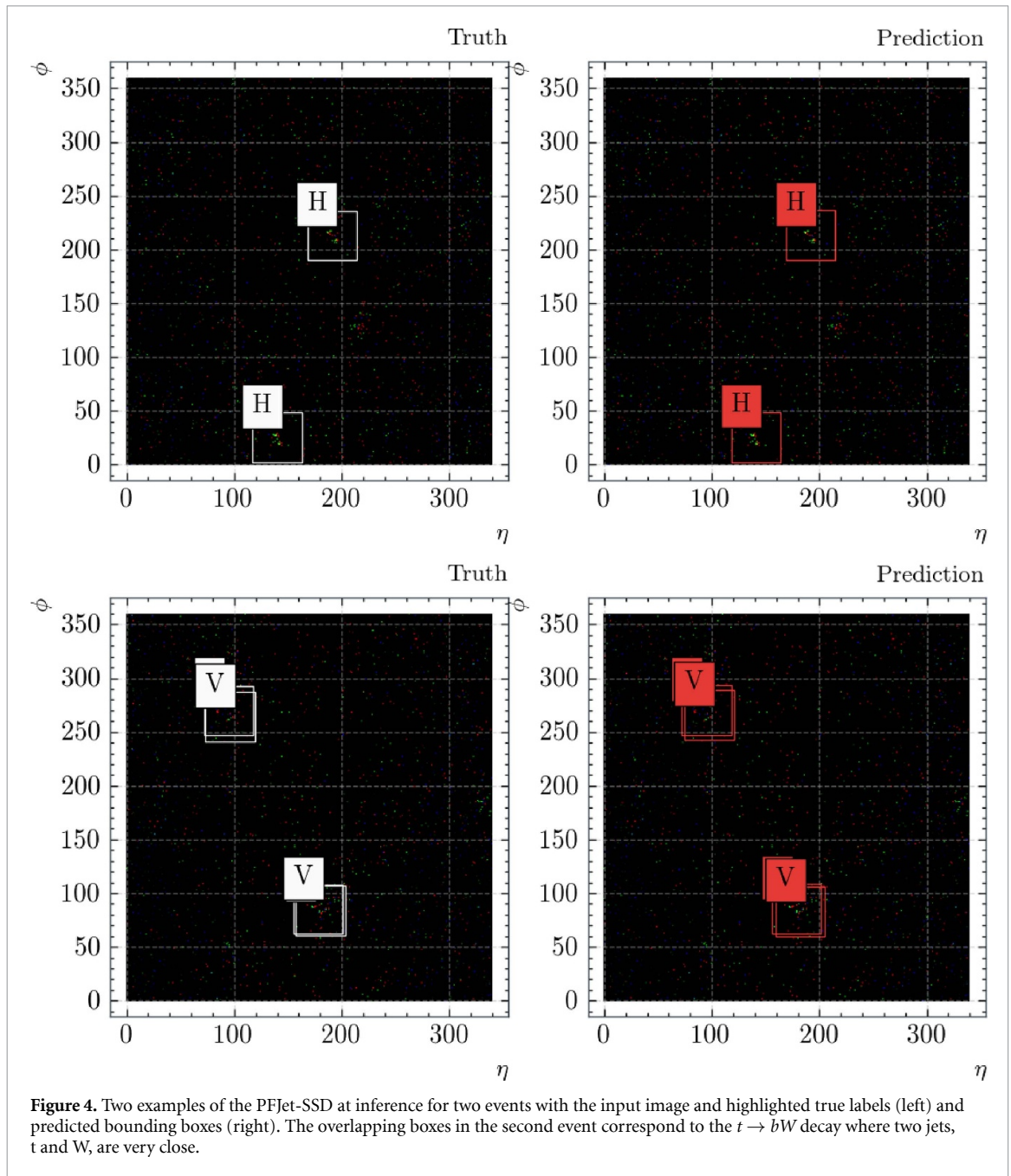
$$\mathcal{L}_{\text{SSD}} = \mathcal{L}_c + \mathcal{L}_l + \mathcal{L}_r,$$

where the \mathcal{L}_c is the classification loss, the \mathcal{L}_l is the localization loss, the \mathcal{L}_r is the regression loss. We use cross-entropy with smooth labels ($\alpha = 0.1$) for classification [103], and Huber loss [57] ($\delta = 1$) for localization and regression.

A common challenge when training object detection models from scratch is the insufficient amount of training data which may lead to overfitting⁷. Thus it is common to see practitioners pre-loading weights from pre-trained classification models on the real-world ImageNet [104] dataset. We found that such a procedure slows down our learning as the real-world images have little relation to our calorimeter images. The full precision network (FPN) can learn faster by using Xavier uniform initialization [105] (which helps with the sparsity of the input). We also augment the training dataset by random flips along η and ϕ dimensions, which we find to greatly stabilize the training. We did not experiment with other augmentation techniques such as changing brightness, contrast, saturation and hue as jets are not invariant to such transitions. The experiments with other commonly used techniques such as Mix-Up [106] or Mosaic [107] yield subpar results, again. This is likely because of the different nature of our input.

We perform five steps of iterative pruning, each with 20 epochs of retraining a gradually decreasing number of channels in each block. We then retrain the network for the last time for 100 epochs. We found out that pre-loading FPN weights when training the quantized versions, i.e. TWN and 8-bit fixed-precision (INT8) network, greatly speeds up convergence.

⁷ That is not a problem in our case as we can generate more events with low cost.



5. Results

In this section, we present the detection and latency performance of PFJet-SSD.

5.1. Detection performance

As a proof of concept, we investigate the tagging of the top-quark (t), W and Z bosons (V) and Higgs boson (H) jet. An example of the PFJet-SSD output is shown in figure 4. PFJet-SSD outputs predicted categorical label, prediction confidence and the centre coordinates of the object. In object detection true positive is defined as prediction with predicted category equal to the ground truth label and intersection over union (IoU) above the predefined threshold, usually 0.5. Successful prediction meets both criteria, otherwise, it is considered as a missed detection. In our case we substitute the IoU requirement with the distance metric $d = \sqrt{\Delta\phi^2 + \Delta\eta^2} < 33$ pixels as we regress only the centre of the box and box dimensions are universal across target classes.

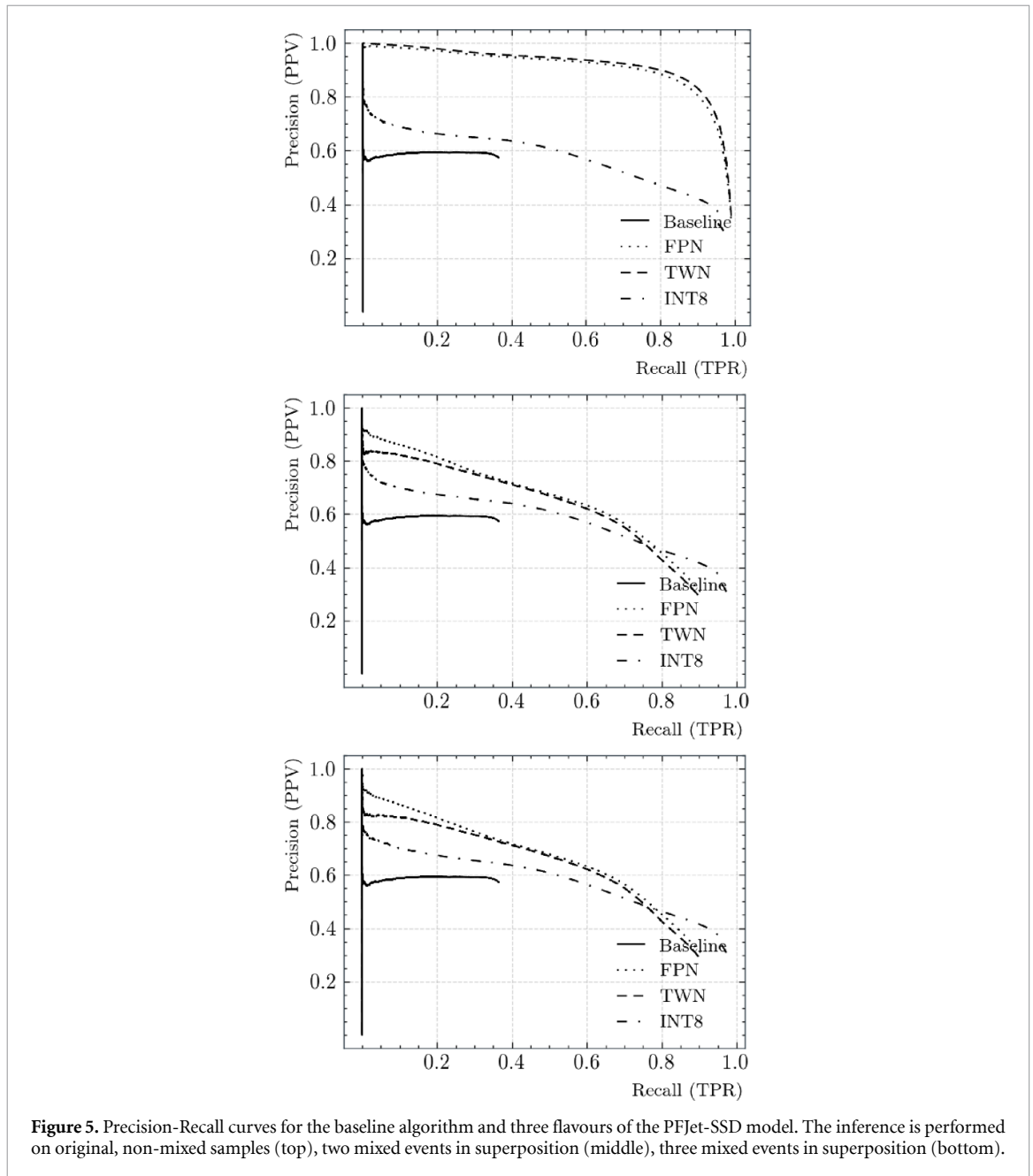


Figure 5. Precision-Recall curves for the baseline algorithm and three flavours of the PFJet-SSD model. The inference is performed on original, non-mixed samples (top), two mixed events in superposition (middle), three mixed events in superposition (bottom).

Our investigation into inference does not find any systematic issues. Occlusion, such as the one in $t \rightarrow bW$ decay, where jets are near, is not an obstacle against correct detection. Also, the jets close to the image edges are, generally, correctly classified.

To evaluate the model we use precision (or positive predictive value, PPV, $\frac{TP}{TP+FP}$) and recall (true positive rate, TPR) curve, and an average precision metric (AP), see figure 5. Intuitively, precision measures how accurate the predictions are while recall measures the quality of the positive predictions. Collectively, they determine how well the found set of jets corresponds to the set we expect to find. To draw a precision-recall (PR) curve, the predictions are first sorted in order of confidence followed by calculation of PPVs and TPRs for each confidence threshold. We held out 90k samples as our test dataset. The TWN network results are closely matching the results of the FPN. TWN benefits from the long retraining period, as it yields marginally better AP. For performance details across target jet classes see table 2.

For non-mixed samples in figure 5 TWN and FPN remarkably agree and yield an appealing precision for any given recall. Fix-point INT8 network drops in detection precision hinting that the network is sensitive to activation but not weight quantization. Hence, in the future, a mixed-precision should be explored as it is likely that not all layers contribute equally to this reduced performance. All flavours of the PFJet-SSD

Table 2. Detection performance for the baseline and PFJet-SSD algorithms, reporting the number of parameters (NoP), the number of operations (NoOps), the precision of weights/activations (W/A), average precision (AP) and precision at 0.3 ($P@R = .3$) and 0.5 ($P@R = .5$) recall. The table does not report parameters and bit precision for the baseline as it is a non-parametric method: not applicable (N/A). The baseline is also unable to reach 0.3 and 0.5 in several cases: no statistics (N/S).

		PFJet-SSD			
		Physics baseline	FPN	TWN	INT8
NoP		N/A		111 228	
NoOps		N/A		1.095G	
W/A		N/A	32/32	2/32	8/8
AP		.161	.848	.857	.566
t jet	AP	.420	.865	.872	.473
	$P@R = .3$.736	.985	.988	.531
	$P@R = .5$.627	.975	.980	.453
W/Z jet	AP	.245	.847	.859	.629
	$P@R = .3$.584	.944	.955	.673
	$P@R = .5$	N/S	.929	.943	.653
H jet	AP	.107	.860	.872	.335
	$P@R = .3$	N/S	.992	.996	.453
	$P@R = .5$	N/S	.978	.986	.400

outperform the physics baseline. We also experimented with two and three events overlaid as the input to the network. This creates much noisier input and results in visibly reduced performance of PFJet-SSD. However, the network was not trained on such samples and such a drop is expected. Besides, the difference between two and three mixed events is minor. In the future, we suggest training the network with Mix-Up [106] or Mosaic [107] augmentations (techniques for mixing multiple samples) which could improve performance on noisier inputs.

Throughout, we compare PFJet-SSD to the *baseline* which is a physics-based algorithm combining a jet soft-drop mass [108], m , selection (under a specific mass hypothesis) and threshold requirement on the appropriate ratio of N-subjettiness [109] variables, τ . In particular, we require $105 < m < 210$ GeV for t jets and use the τ_3/τ_2 N-subjettiness ratio as a score defined in [0, 1]. With this score we make a performance assessment that we can directly compare to that obtained with the PFJet-SSD algorithm. Similarly, we require $65 < m < 105$ GeV for V jets and $105 < m < 140$ GeV for H jets, using the τ_2/τ_1 N-subjettiness ratio as a score for these baseline taggers. This physics-motivated baseline has performance that is typical of a rule-based state-of-the-art substructure jet tagger, with the typical recall of 0.3 for the precision of 0.6.

Figure 6 shows the dependence of the precision at fixed recall across different jet classes. The precision is rather flat in all cases. The TWN results match closely the FPN ones, while an overall drop in performance (approximately constant across η , ϕ , and p_T) is observed for the INT8 network. A drop is observed at the boundaries of the η region, as a consequence of jets leaking out of acceptance at the edge of the endcaps (missing information of a part of the shower). Such a drop is not observed in the ϕ dimension suggesting that the network can handle the periodicity of the image. The precision across p_T stays relatively flat, however, the sudden drop in the high p_T region of V jets is due to the low number of samples in that region, see the details in section 4. Notice that the drop in TRP at low jet p_T for top, W/Z, and H tagging is induced by transition from a boosted-jet to a resolved jets regime. Despite the fact that there so far little use of boosted jets from heavy particles in this low p_T regime, it is interesting to notice that the drop in efficiency of the PFJet-SSD in the low- p_T regime is less pronounced than for the baseline algorithm, which could be interesting to increase the reconstruction efficiency in transition between boosted and resolved topologies.

Figure 7 shows the residual in the determination of η and ϕ and the ratio of the reconstructed-to-true jet p_T , as a function of the jet p_T for the different classes.

Finally, we visualize the most repeating filters of the TWN in figure 8. Remarkably, the network optimizes to use a set very similar to the commonly used ones, e.g. smoothing, corner detection or edge detection filters.

5.2. Latency and power measurements

We investigate the latency and throughput of the proposed algorithm on architectures where parallel computing is more adequate. We compare the baseline, running native PyTorch inference on the Intel Xeon Silver 4114 CPU with ONNX accelerated version and TensorRT optimized version on Nvidia Tesla V100. Results are given in figure 9, separately for CPUs and GPUs. Having in mind an offline application, one could

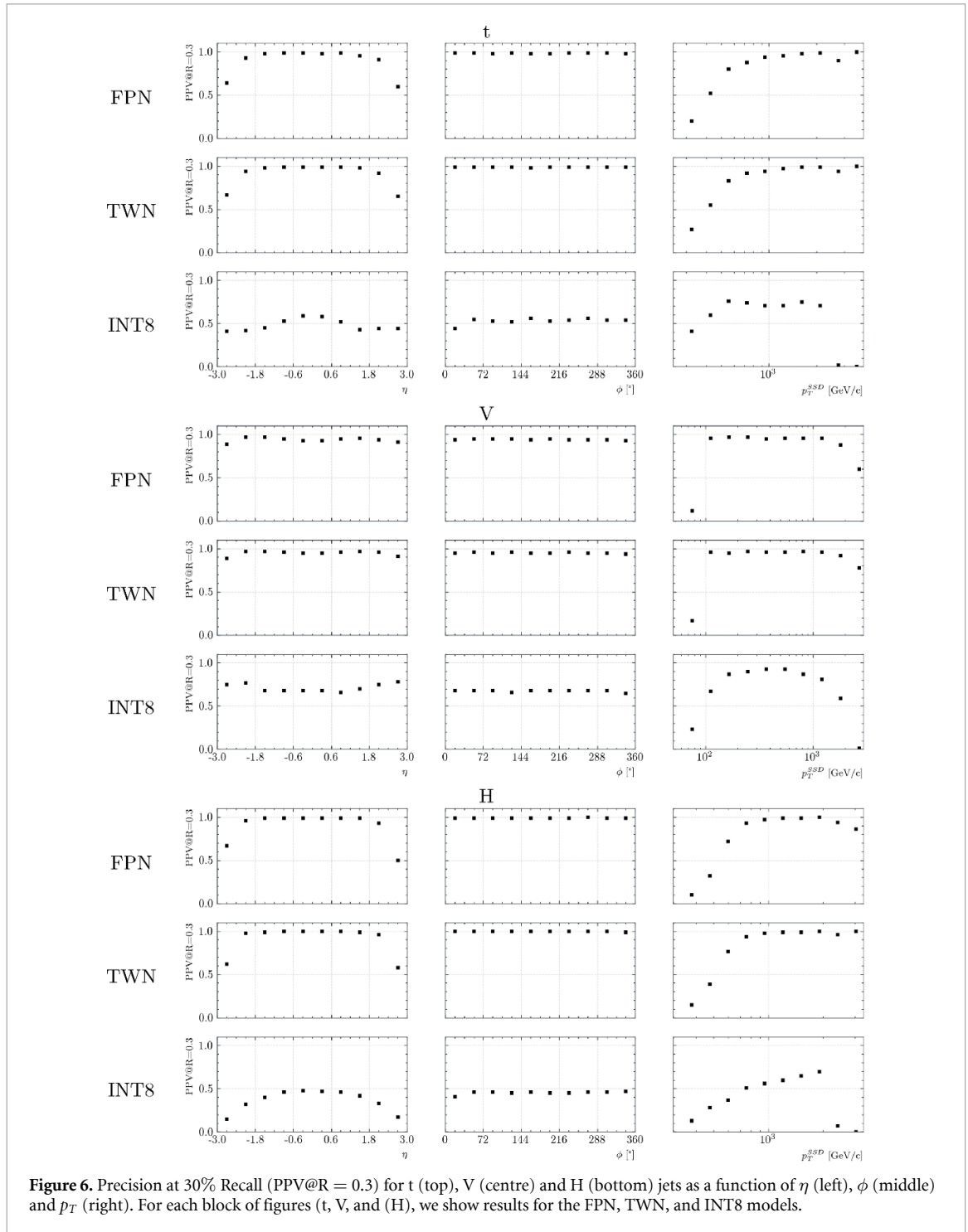


Figure 6. Precision at 30% Recall (PPV@R = 0.3) for t (top), V (centre) and H (bottom) jets as a function of η (left), ϕ (middle) and p_T (right). For each block of figures (t, V, and H), we show results for the FPN, TWN, and INT8 models.

maximize the throughput by running the network at once across batches of events, e.g. implementing the inference-as-a-service concept discussed in [110].

While the inference-as-a-service paradigm could also be implemented online, the current design of HLT farms foresees that processing parallelization is achieved by sending different events to different computing units. In this context, the batch size is constrained to one, since the inference of the proposed SSD model happens per event. In this case, execution on CPU would be borderline, within the average event processing latency but consuming most of it. On the other hand, moving the execution to a GPU would reduce the execution time to negligible levels. This could be particularly interesting under the assumption that GPUs would be used to run the local reconstruction [111–113] and the creation of PF candidates [114].

Deep learning inference at scale requires high power consumption, especially with the use of GPUs and CPUs. It is possible to keep the power and die area at more manageable levels by deploying an AI-specific

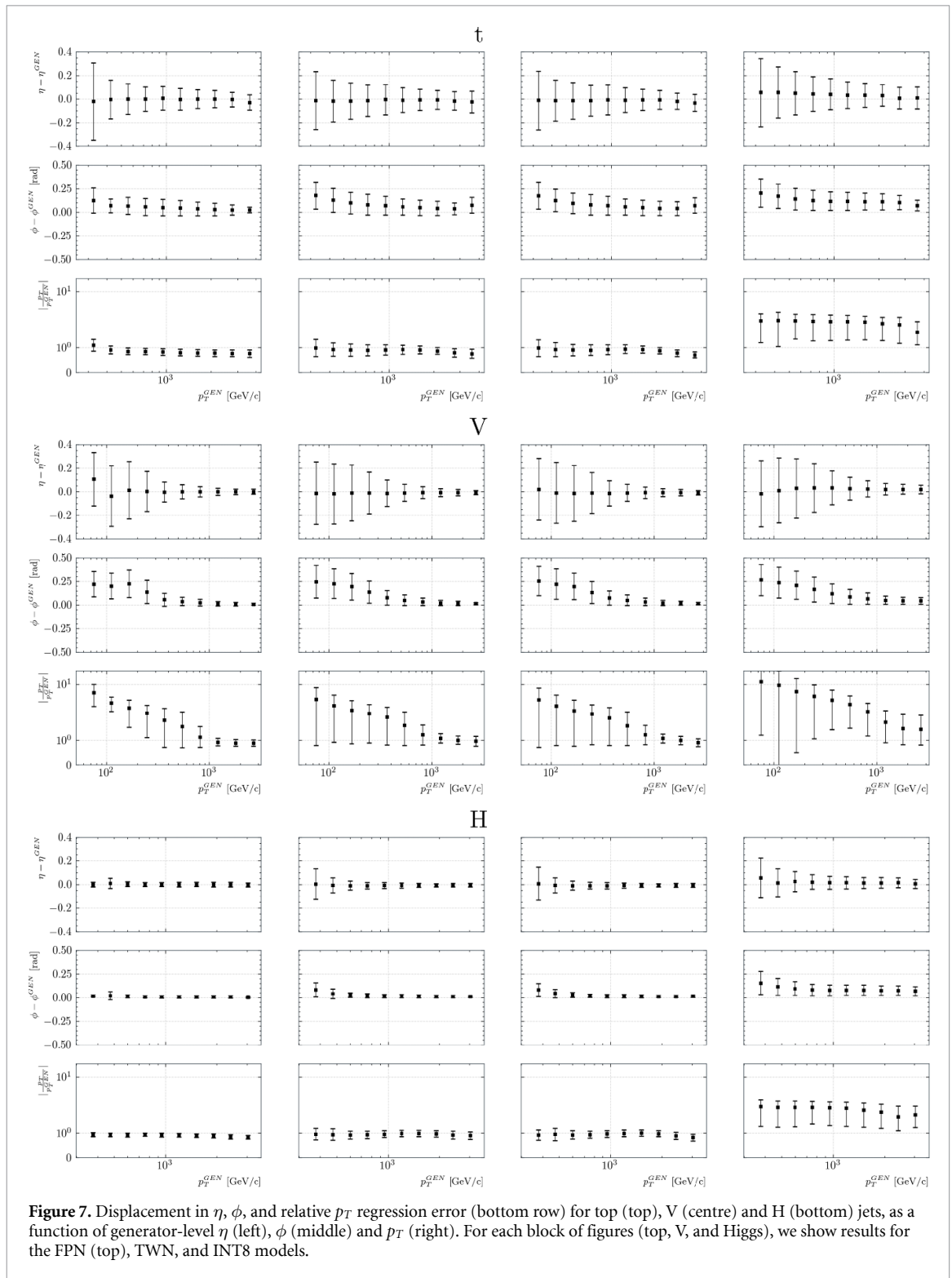


Figure 7. Displacement in η , ϕ , and relative p_T regression error (bottom row) for top (top), V (centre) and H (bottom) jets, as a function of generator-level η (left), ϕ (middle) and p_T (right). For each block of figures (top, V, and Higgs), we show results for the FPN (top), TWN, and INT8 models.

hardware platform as used in edge devices. Since edge devices usually operate on batteries where power is a limited resource, AI-specific hardware platforms for edge devices are highly power efficient. With smaller die areas, manufacturing costs and power consumption can be reduced.

SensPro is a family of ultra-light AI DSPs that can perform efficient inference while consuming only a fraction of the power and area used by GPUs and CPUs. CEVA's hardware platform for jet detection consists of a stack of ten SensPro (SP) DSP cores. Each core delivers 2 TOPS. An additional SP core is added to serve as a controller. This solution delivers 20 TOPS and can run TWN natively, reaching latency comparable to a GPU running an 8-bit network. This proposed layout has orders of magnitude lower area and power consumption than GPU and CPU, see table 3. The SP ultra-light solution can also be synthesized to an FPGA and used in collision detection.



Figure 8. Most common filters of the PFJet-SSD TWN.

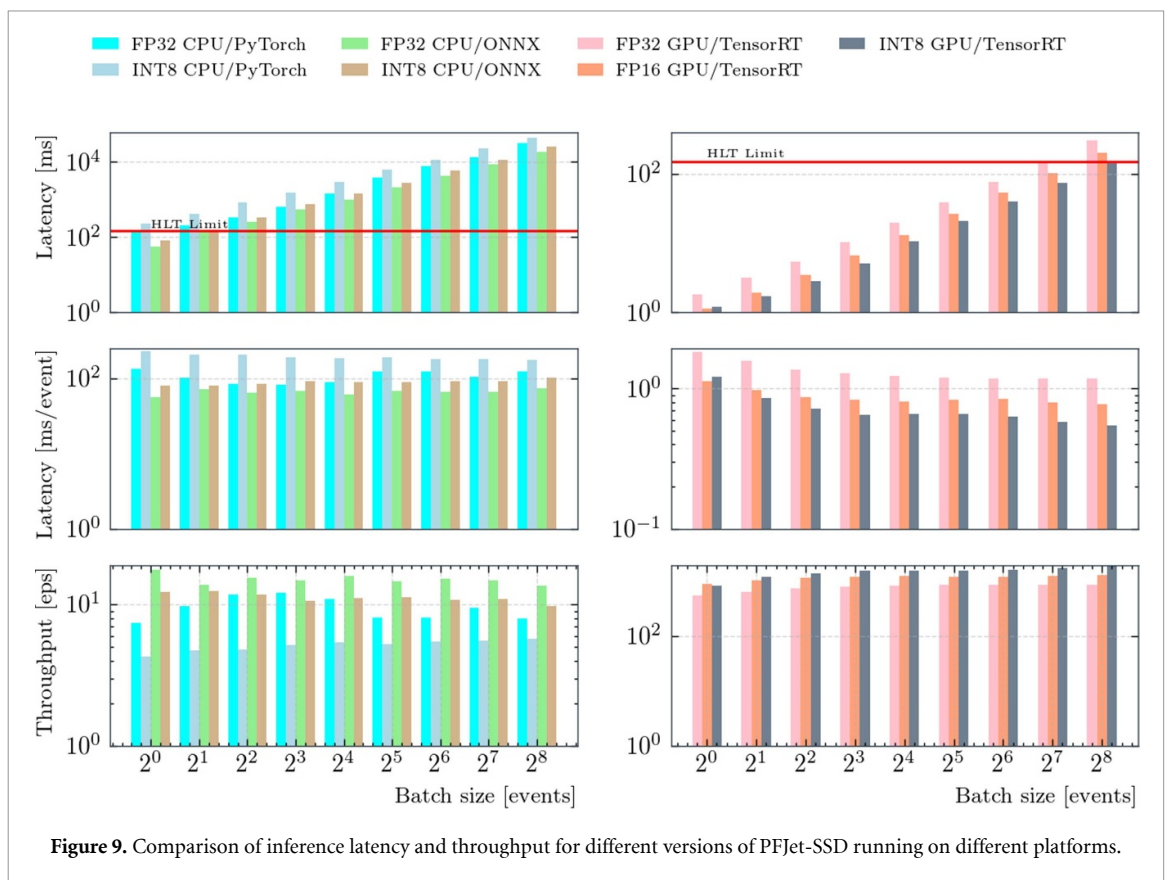


Figure 9. Comparison of inference latency and throughput for different versions of PFJet-SSD running on different platforms.

Table 3. Die area, power and latency measurements for different hardware architectures. The latency is measured for inference on a single input. The reason for this is that collision detection is done sequentially in real-time. The input data is fed to the network directly from the sensor without storing it.

	Die area (mm ²)	Power (W)	Latency (ms)
DSP CEVA SP1000 2x8	0.77	0.75	8.5
DSP CEVA 10xSP1000 + Controller 2x8	8.47	8.25	0.9
GPU Nvidia Tesla V100 8x8	815	250	1.1
CPU Intel Xeon Silver 4114. 32float	4294	85	134

6. Conclusions

We propose a fast and lightweight detection algorithm for jet tagging and reconstruction based on computer vision techniques. Naturally high precision and generalization are required, but nuisance factors of variations can break the algorithm. That makes this problem hard. Intra-class variations, such as perspective distortion,

e.g. rotation; densely arranged jets (occlusion); or blurred signatures (the detector response may not be clear) are common challenges. Besides, as jets are small objects, a reappearing issue with object detectors and background pileup may further disturb their visual appearance. Thus, robustness to detector effects, its imperfections and failures is required.

Even after a successful proof-of-concept deployment to production will still produce challenges as many of the problems lay outside of the simulation. More importantly, the real-time detection requirements force further investigations into more optimizations on algorithm and hardware runtime.

The PFJet-SSD paves the way for solving these issues. The algorithm did not experience accuracy drops during pruning, suggesting that the depth of the network is more important than the width. The number of channels can likely be reduced further and thus speed up computations. We observed a gap between TWN and INT8 performance which suggests to us that the optimal quantization level could be achieved through mixed-precision, a possible direction for future studies.

From the physics point of view, the algorithm manifests an interesting behaviour in low momentum regions out of reach for the baseline model, see high precision results in figure 6, which could help increasing the reconstruction efficiency for all-hadronic decays of heavy particles in the transition regime between boosted and resolved topologies.

Data availability statement

The data that support the findings of this study are openly available at <https://doi.org/10.5281/zenodo.4883651>.

Acknowledgments

We thank Loukas Gouskos and Huilin Qu for useful discussions and suggestions. A A P, M P, S S and V L are supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant Agreement No. 772369). A A P is supported by CEVA under the CERN Knowledge Transfer Group.

ORCID iDs

Adrian Alan Pol  <https://orcid.org/0000-0002-9034-0230>
Thea Aarrestad  <https://orcid.org/0000-0002-7671-243X>
Ekaterina Govorkova  <https://orcid.org/0000-0003-1920-6618>
Vladimir Loncar  <https://orcid.org/0000-0003-3651-0232>
Jennifer Ngadiuba  <https://orcid.org/0000-0002-0055-2935>
Maurizio Pierini  <https://orcid.org/0000-0003-1939-4268>

References

- [1] The LHC Study Group 1995 The large Hadron Collider, conceptual design *Technical Report* CERN/AC/95-05 (LHC) (Geneva)
- [2] Khachatryan V et al 2017 *J. Instrum.* **12** 01020
- [3] CMS Collaboration 2016 arXiv:1609.02366
- [4] Apollinari G, Brüning O, Nakamoto T and Rossi L 2017 arXiv:1705.08830
- [5] Albrecht J et al 2019 *Comput. Softw. Big Sci.* **3** 1–49
- [6] Butterworth J M, Davison A R, Rubin M and Salam G P 2008 *Phys. Rev. Lett.* **100** 242001
- [7] Skiba W and Tucker-Smith D 2007 *Phys. Rev. D* **75** 115010
- [8] Baumgart M, Leibovich A K, Mehen T and Rothstein I Z 2014 *J. High Energy Phys.* **2014** 173
- [9] Aad G et al 2015 *J. High Energy Phys.* **2015** 1–39
- [10] Adams D et al 2015 *Eur. Phys. J. C* **75** 409
- [11] Abdesselam A et al 2011 *Eur. Phys. J. C* **71** 1661
- [12] Altheimer A et al 2012 *J. Phys. G: Nucl. Part. Phys.* **39** 063001
- [13] Altheimer A et al 2014 *Eur. Phys. J. C* **74** 2792
- [14] Caruana R 1997 *Mach. Learn.* **28** 41–75
- [15] Sirunyan A M et al CMS 2020 *Comput. Softw. Big Sci.* **4** 10
- [16] Pol A A and Pierini M 2020 Jet single shot detection (available at: <https://zenodo.org/record/4883651>)
- [17] Liu Z, Mao H, Wu C Y, Feichtenhofer C, Darrell T and Xie S 2022 arXiv:2201.03545
- [18] Hendrycks D and Gimpel K 2016 arXiv:1606.08415
- [19] Plehn T, Spannowsky M, Takeuchi M and Zerwas D 2010 *J. High Energy Phys.* **2010** 1–20
- [20] Larkoski A J, Marzani S, Soyez G and Thaler J 2014 *J. High Energy Phys.* **2014** 146
- [21] Thaler J and Van Tilburg K 2011 *J. High Energy Phys.* **2011** 15
- [22] Larkoski A J, Salam G P and Thaler J 2013 *J. High Energy Phys.* **2013** 108
- [23] Krohn D, Thaler J and Wang L-T 2010 *J. High Energy Phys.* **2010** 84

- [24] Ellis S D, Vermilion C K and Walsh J R 2010 *Phys. Rev. D* **81** 094023
- [25] Dasgupta M, Fregoso A, Marzani S and Salam G P 2013 *J. High Energy Phys.* **2013** 29
- [26] Dasgupta M, Fregoso A, Marzani S and Powling A 2013 *Eur. Phys. J. C* **73** 1–32
- [27] Dasgupta M, Powling A and Siodmok A 2015 *J. High Energy Phys.* **2015** 1–54
- [28] Cogan J, Kagan M, Strauss E and Schwartzman A 2015 *J. High Energy Phys.* **2015** 118
- [29] Almeida L G, Backović M, Cliche M, Lee S J and Perelstein M 2015 *J. High Energy Phys.* **2015** 1–21
- [30] Baldi P, Bauer K, Eng C, Sadowski P and Whiteson D 2016 *Phys. Rev. D* **93** 094034
- [31] de Oliveira L, Kagan M, Mackey L, Nachman B and Schwartzman A 2016 *J. High Energy Phys.* **2016** 1–32
- [32] Guest D, Collado J, Baldi P, Hsu S-C, Urban G and Whiteson D 2016 *Phys. Rev. D* **94** 112002
- [33] de Oliveira L, Paganini M and Nachman B 2017 *Comput. Softw. Big Sci.* **1** 1–24
- [34] Pearkes J, Fedorko W, Lister A and Gay C 2017 arXiv:1704.02124
- [35] Kasieczka G, Plehn T, Russell M and Schell T 2017 *J. High Energy Phys.* **2017** 6
- [36] Komiske P T, Metodiev E M and Schwartz M D 2017 *J. High Energy Phys.* **2017** 110
- [37] Barnard J, Dawe E N, Dolan M J and Rajcic N 2017 *Phys. Rev. D* **95** 014018
- [38] Macaluso S and Shih D 2018 *J. High Energy Phys.* **2018** 121
- [39] Butter A, Kasieczka G, Plehn T and Russell M 2018 *SciPost Phys.* **5** 028
- [40] Lan S-Q 2018 *Adv. High Energy Phys.* **2018** 4350287
- [41] Kasieczka G et al 2019 *SciPost Phys.* **7** 014
- [42] Bhimji W et al 2018 *J. Phys.: Conf. Ser.* **1085** 042034
- [43] Nguyen T Q, Weitekamp D, Anderson D, Castello R, Cerri O, Pierini M, Spiropulu M and Vlimant J-R 2019 *Comput. Softw. Big Sci.* **3** 1–14
- [44] Andrews M, Paulini M, Gleyzer S and Poczos B 2020 *Comput. Softw. Big Sci.* **4** 6
- [45] Andrews M, Alison J, An S, Burkle B, Gleyzer S, Narain M, Paulini M, Poczos B and Usai E 2020 *Nucl. Instrum. Methods Phys. Res. A* **977** 164304
- [46] Zhang K, Zhang Z, Li Z and Qiao Y 2016 *IEEE Signal Process. Lett.* **23** 1499–503
- [47] Zhang L, Lin L, Liang X and He K 2016 Is faster R-CNN doing well for pedestrian detection? *European Conf. on Computer Vision* (Springer) pp 443–57
- [48] Zou Z, Shi Z, Guo Y and Ye J 2019 arXiv:1905.05055
- [49] Liu L, Ouyang W, Wang X, Fieguth P, Chen J, Liu X and Pietikäinen M 2020 *Int. J. Comput. Vis.* **128** 261–318
- [50] Redmon J, Divvala S, Girshick R and Farhadi A 2016 You only look once: unified, real-time object detection 2016 *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* (IEEE) pp 779–88
- [51] Redmon J and Farhadi A 2017 YOLO9000: better, faster, stronger 2017 *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* (IEEE) pp 6517–25
- [52] Lin T Y, Goyal P, Girshick R, He K and Dollar P 2017 Focal loss for dense object detection 2017 *IEEE Int. Conf. on Computer Vision (ICCV)* (IEEE) pp 2999–3007
- [53] Fu C Y, Liu W, Ranga A, Tyagi A and Berg A C 2017 arXiv:1701.06659
- [54] Zhou X, Wang D and Krähenbühl P 2019 arXiv:1904.07850
- [55] Girshick R, Donahue J, Darrell T and Malik J 2014 Rich feature hierarchies for accurate object detection and semantic segmentation 2014 *IEEE Conf. on Computer Vision and Pattern Recognition* (IEEE) pp 580–7
- [56] Ren S, He K, Girshick R B and Sun J 2015 Faster R-CNN: towards real-time object detection with region proposal networks *Annual Conf. on Neural Information Processing Systems 2015 (Montreal, Quebec, Canada, 7–12 December 2015) (Advances in Neural Information Processing Systems)* eds C Cortes, N D Lawrence, D D Lee, M Sugiyama and R Garnett vol 28 pp 91–99 (available at: <https://proceedings.neurips.cc/paper/2015/hash/14bfa6bb14875e45bba028a21ed38046-Abstract.html>)
- [57] Girshick R 2015 Fast R-CNN 2015 *IEEE Int. Conf. on Computer Vision (ICCV)* (IEEE) pp 1440–8
- [58] Dai J, Li Y, He K and Sun J 2016 R-FCN: object detection via region-based fully convolutional networks *Annual Conf. on Neural Information Processing Systems 2016 (Barcelona, Spain, 5–10 December 2016) (Advances in Neural Information Processing Systems)* eds D D Lee, M Sugiyama, U von Luxburg, I Guyon and R Garnett vol 29 pp 379–87 (available at: <https://proceedings.neurips.cc/paper/2016/hash/577ef1154f3240ad5b9b413aa7346a1e-Abstract.html>)
- [59] Xu H, Lv X, Wang X, Ren Z, Bodla N and Chellappa R 2018 Deep regionlets for object detection *Proc. European Conf. on Computer Vision (ECCV)* pp 798–814
- [60] Liu W, Anguelov D, Erhan D, Szegedy C, Reed S, Fu C Y and Berg A C 2016 SSD: single shot multibox detector *European Conf. on Computer Vision* (Springer) pp 21–37
- [61] Felzenszwalb P F, Girshick R B, McAllester D and Ramanan D 2010 *IEEE Trans. Pattern Anal. Mach. Intell.* **32** 1627–45
- [62] Simonyan K and Zisserman A 2015 Very deep convolutional networks for large-scale image recognition (arXiv:1409.1556)
- [63] Jetley S, Lord N A, Lee N and Torr P H S 2018 Learn to pay attention 6th *Int. Conf. on Learning Representations, ICLR 2018 (Vancouver, BC, Canada, 30 April–3 May 2018)* (OpenReview.net) (available at: <https://openreview.net/forum?id=HyzbhfWRW>)
- [64] Anderson P, He X, Buehler C, Teney D, Johnson M, Gould S and Zhang L 2018 Bottom-up and top-down attention for image captioning and visual question answering 2018 *IEEE/CVF Conf. on Computer Vision and Pattern Recognition* (IEEE) pp 6077–86
- [65] Oktay O et al 2018 arXiv:1804.03999
- [66] Li Y, Chen Y, Wang N and Zhang Z X 2019 Scale-aware trident networks for object detection 2019 *IEEE/CVF Int. Conf. on Computer Vision (ICCV)* (IEEE) pp 6053–62
- [67] Yi J, Wu P and Metaxas D N 2019 *Comput. Vis. Image Underst.* **189** 102827
- [68] Woo S, Park J, Lee J Y and Kweon I S 2018 Cbam: Convolutional block attention module *Proc. European Conf. on Computer Vision (ECCV)* pp 3–19
- [69] Fu J, Liu J, Tian H, Li Y, Bao Y, Fang Z and Lu H 2019 Dual attention network for scene segmentation 2019 *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)* (IEEE) pp 3146–54
- [70] Hu J, Shen L and Sun G 2018 Squeeze-and-excitation networks 2018 *IEEE/CVF Conf. on Computer Vision and Pattern Recognition* (IEEE) pp 7132–41
- [71] Chen Y, Kalantidis Y, Li J, Yan S and Feng J 2018 A2-nets: double attention networks *Annual Conf. on Neural Information Processing Systems 2018, NeurIPS 2018 (Montréal, Canada, 3–8 December 2018) (Advances in Neural Information Processing Systems)* ed S Bengio, H M Wallach, H Larochelle, K Grauman, N Cesa-Bianchi and R Garnett vol 31 pp 350–9 (available at: <https://proceedings.neurips.cc/paper/2018/hash/e165421110ba03099a1c0393373c5b43-Abstract.html>)
- [72] Zhou D and Bie Ju L 2021 Deep convolutional neural networks (arXiv:1910.03151)

- [73] Han S, Mao H and Dally W J 2015 arXiv:1510.00149
- [74] Cheng Y, Wang D, Zhou P and Zhang T 2018 *IEEE Signal Process. Mag.* **35** 126–36
- [75] LeCun Y, Denker J and Solla S 1989 Optimal brain damage *Advances in Neural Information Processing Systems (Denver, Colorado, USA, 27–30 November 1989)* vol 2 pp 598–605
- [76] Louizos C, Welling M and Kingma D P 2018 Learning sparse neural networks through L_0 regularization *6th Int. Conf. on Learning Representations, ICLR 2018 (Vancouver, BC, Canada, 30 April–3 May 2018)* (OpenReview.net) (available at: <https://openreview.net/forum?id=H1Y8hhg0b>)
- [77] Gordon A, Eban E, Nachum O, Chen B, Wu H, Yang T J and Choi E 2018 MorphNet: fast and simple resource-constrained structure learning of deep networks *2018 IEEE/CVF Conf. on Computer Vision and Pattern Recognition (IEEE)* pp 1586–95
- [78] Howard A G, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, Andreetto M and Adam H 2017 arXiv:1704.04861
- [79] Iandola F N, Han S, Moskewicz M W, Ashraf K, Dally W J and Keutzer K 2016 arXiv:1602.07360
- [80] Cohen T and Welling M 2016 Group equivariant convolutional networks *Proc. 33rd Int. Conf. on Machine Learning, ICML 2016 (New York City, NY, USA, 19–24 June 2016) (JMLR Workshop and Conf. Proc.)* vol 48, ed M Balcan and K Q Weinberger (JMLR.org) pp 2990–9 (available at: <http://proceedings.mlr.press/v48/cohen16.html>)
- [81] Courbariaux M, Bengio Y and David J 2015 BinaryConnect: training deep neural networks with binary weights during propagations *Annual Conf. on Neural Information Processing Systems 2015 (Montreal, Quebec, Canada, 7–12 December 2015) (Advances in Neural Information Processing Systems)* ed C Cortes, N D Lawrence, D D Lee, M Sugiyama and R Garnett vol 28 pp 3123–31 (available at: <https://proceedings.neurips.cc/paper/2015/hash/3e15cc11f979ed25912df5b0669f2cd-Abstract.html>)
- [82] Courbariaux M, Hubara I, Soudry D, El-Yaniv R and Bengio Y 2016 arXiv:1602.02830
- [83] Zhou S, Wu Y, Ni Z, Zhou X, Wen H and Zou Y 2016 arXiv:1606.06160
- [84] Rastegari M, Ordonez V, Redmon J and Farhadi A 2016 XNOR-Net: imagenet classification using binary convolutional neural networks *European Conf. on Computer Vision (Berlin: Springer)* pp 525–42
- [85] Hubara I, Courbariaux M, Soudry D, El-Yaniv R and Bengio Y 2017 *J. Mach. Learn. Res.* **18** 6869–98
- [86] Zhu C, Han S, Mao H and Dally W J 2017 Trained ternary quantization *5th Int. Conf. on Learning Representations, ICLR 2017 (Toulon, France, 24–26 April 2017)* (OpenReview.net) (available at: https://openreview.net/forum?id=S1_pAu9xl)
- [87] Lee E H, Miyashita D, Chai E, Murmann B and Wong S S 2017 LogNet: energy-efficient neural networks using logarithmic computation *2017 IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP) (IEEE)* pp 5900–4
- [88] Cai Z, He X, Sun J and Vasconcelos N 2017 Deep learning with low precision by half-wave Gaussian quantization *2017 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) (IEEE)* pp 5406–14
- [89] Li F, Zhang B and Liu B 2016 arXiv:1605.04711
- [90] Pol A A et al 2021 *EPJ Web Conf.* **251** 04027
- [91] Ioffe S and Szegedy C 2015 Batch normalization: accelerating deep network training by reducing internal covariate shift *Proc. 32nd Int. Conf. on Machine Learning, ICML 2015 (Lille, France, 6–11 July 2015) (JMLR Workshop and Conf. Proc.)* eds F R Bach and D M Blei (JMLR.org) vol 37 pp 448–56 (available at: <http://proceedings.mlr.press/v37/ioffe15.html>)
- [92] Sari E, Belbahri M and Nia V P 2020 How does batch normalization help binary training? (arXiv:1909.09139)
- [93] He K, Zhang X, Ren S and Sun J 2015 Delving deep into rectifiers: surpassing human-level performance on ImageNet classification *2015 IEEE Int. Conf. on Computer Vision (ICCV) (IEEE)* pp 1026–34
- [94] Ghiasi G, Lin T and Le Q V 2018 DropBlock: a regularization method for convolutional networks *Annual Conf. on Neural Information Processing Systems 2018, NeurIPS 2018 (3–8 December, 2018)* eds S Bengio, H M Wallach, H Larochelle, K Grauman, N Cesa-Bianchi and R Garnett pp 10750–60 (available at: <https://proceedings.neurips.cc/paper/2018/hash/7edcfb2d8f6a659ef4cd1e6c9b6d7079-Abstract.html>)
- [95] Han S, Pool J, Tran J and Dally W J 2015 arXiv:1506.02626
- [96] Liu Z, Li J, Shen Z, Huang G, Yan S and Zhang C 2017 Learning efficient convolutional networks through network slimming *2017 IEEE Int. Conf. on Computer Vision (ICCV) (IEEE)* pp 2755–63
- [97] Sjöstrand T, Mrenna S and Skands P 2008 *Comput. Phys. Commun.* **178** 852–67
- [98] De Favereau J, Delaere C, Demin P, Giammanco A, Lemaître V, Mertens A and Selvaggi M 2014 *J. High Energy Phys.* **2014** 57
- [99] CMS Collaboration 2008 *J. Instrum.* **3** S08004
- [100] Sirunyan A M et al 2017 *J. Instrum.* **12** 10003
- [101] Aaboud M et al ATLAS 2017 *Eur. Phys. J. C* **77** 466
- [102] Paszke A et al 2019 PyTorch: an imperative style, high-performance deep learning library *Annual Conf. on Neural Information Processing Systems 2019, NeurIPS 2019 (Vancouver, BC, Canada, 8–14 December 2019) (Advances in Neural Information Processing Systems)* ed H M Wallach, H Larochelle, A Beygelzimer, F d'Alché-Buc, E B Fox and R Garnett vol 32 pp 8024–35 (available at: <https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f701272740-Abstract.html>)
- [103] Szegedy C, Vanhoucke V, Ioffe S, Shlens J and Wojna Z 2016 Rethinking the inception architecture for computer vision *Proc. IEEE Conf. on Computer Vision and Pattern Recognition* pp 2818–26
- [104] Deng J, Dong W, Socher R, Li L, Li K and Li F 2009 ImageNet: a large-scale hierarchical image database *2009 IEEE Conf. on Computer Vision and Pattern Recognition (IEEE)* pp 248–55
- [105] Glorot X and Bengio Y 2010 Understanding the difficulty of training deep feedforward neural networks *Proc. Thirteenth Int. Conf. on Artificial Intelligence and Statistics (Proc. of Machine Learning Research) (Chia Laguna Resort, Sardinia, Italy)* vol 9, ed Y W Teh and M Titterton (JMLR Workshop and Conf. Proc.) pp 249–56
- [106] Zhang H, Cissé M, Dauphin Y N and Lopez-Paz D 2018 Mixup: beyond empirical risk minimization *6th Int. Conf. on Learning Representations, ICLR 2018 (Vancouver, BC, Canada, 30 April–3 May 2018)* (OpenReview.net) (available at: <https://openreview.net/forum?id=r1Ddp1-Rb>)
- [107] Bochkovskiy A, Wang C Y and Liao H Y M 2020 arXiv:abs/2004.10934
- [108] Larkoski A J, Marzani S, Soyez G and Thaler J 2014 *J. High Energy Phys.* **2014** 146
- [109] Thaler J and Van Tilburg K 2011 *J. High Energy Phys.* **2011** 015
- [110] Krupa J et al 2021 *Mach. Learn.: Sci. Technol.* **2** 035005
- [111] Bocci A, Innocente V, Kortelainen M, Pantaleo F and Rovere M 2020 *Front. Big Data* **3** 601728
- [112] Rovere M, Chen Z, Di Pilato A, Pantaleo F and Seez C 2020 *Front. Big Data* **3** 591315
- [113] Qasim S R, Long K, Kieseler J, Pierini M, Nawaz R, Pierini M, Nawaz R and CMS 2021 *EPJ Web Conf.* **251** 03072
- [114] Pata J, Duarte J, Vlimant J-R, Pierini M and Spiropulu M 2021 *Eur. Phys. J. C* **81** 381

PAPER • OPEN ACCESS

Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml

To cite this article: Elham E Khoda *et al* 2023 *Mach. Learn.: Sci. Technol.* **4** 025004

View the [article online](#) for updates and enhancements.

You may also like

- [Novel heuristic-based hybrid ResNeXt with recurrent neural network to handle multi class classification of sentiment analysis](#)
Lakshmi Revathi Krosuri and Rama Satish Aravapalli
- [The reusability prior: comparing deep learning models without training](#)
Aydin Göze Polat and Ferda Nur Alpaslan
- [DeepAstroUDA: semi-supervised universal domain adaptation for cross-survey galaxy morphology classification and anomaly detection](#)
A iprijanovi, A Lewis, K Pedro et al.



PAPER

OPEN ACCESS

RECEIVED

26 September 2022

REVISED

15 February 2023

ACCEPTED FOR PUBLICATION

1 March 2023

PUBLISHED











10 April 2023

Original Content from this work may be used under the terms of the [Creative Commons Attribution 4.0 licence](#).

Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.



Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml

Elham E Khoda^{1,*} , Dylan Rankin^{2,*} , Rafael Teixeira de Lima^{3,*} , Philip Harris² , Scott Hauck¹ , Shih-Chieh Hsu¹ , Michael Kagan³ , Vladimir Loncar² , Chaitanya Paikara¹, Richa Rao¹, Sioni Summers⁴ , Caterina Vernieri³  and Aaron Wang¹

¹ University of Washington, Seattle, WA 98195, United States of America

² Massachusetts Institute of Technology, Cambridge, MA, 02139, United States of America

³ SLAC National Accelerator Laboratory, Menlo Park, CA 94025, United States of America

⁴ European Organization for Nuclear Research (CERN), Geneva, Switzerland

* Authors to whom any correspondence should be addressed.

E-mail: ekhoda@uw.edu, drankin@mit.edu and rafaelt@slac.stanford.edu

Keywords: deep learning, recurrent neural network, LSTM, GRU, hls4ml, FPGA

Abstract

Recurrent neural networks have been shown to be effective architectures for many tasks in high energy physics, and thus have been widely adopted. Their use in low-latency environments has, however, been limited as a result of the difficulties of implementing recurrent architectures on field-programmable gate arrays (FPGAs). In this paper we present an implementation of two types of recurrent neural network layers—long short-term memory and gated recurrent unit—within the hls4ml framework. We demonstrate that our implementation is capable of producing effective designs for both small and large models, and can be customized to meet specific design requirements for inference latencies and FPGA resources. We show the performance and synthesized designs for multiple neural networks, many of which are trained specifically for jet identification tasks at the CERN Large Hadron Collider.

1. Introduction

Machine learning (ML) has seen a huge expansion in its range of uses over the last decade. It is difficult to find a field of industry or science that has not at least explored ML in some capacity. One particular field where ML usage has seen widespread interest is in high energy physics, which benefits from complex multidimensional problems, large datasets of accurate simulation, and substantial existing computing infrastructure. These all contribute to a field which has adopted ML algorithms for many aspects of research. While most ML algorithms in high energy physics are run using central processing units (CPUs) and graphics processing units (GPUs) which provide inference latencies in the milliseconds, field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs) have begun to be used for those applications that demand low latencies [1–4]. Up to now recurrent neural networks (RNNs) have received relatively little attention for these low latency applications, despite their success in many physics tasks and prevalence in the field at large.

RNNs are neural network architectures that treat their inputs as a sequence with a well-defined order. RNNs act successively on each entry of the input sequence, utilizing the same set of weights in each step, allowing for RNNs to act on sequences of variable length. Most modern RNN applications utilize one of two recurrent layer implementations: long-short term memory layer (LSTM) [5] or gated recurrent unit layers (GRU) [6]. Both LSTMs and GRUs contain *forget gates*, which avoid vanishing gradient problems [7] and allows for long-distance correlations to be learned by the algorithm. These RNN models, often employed to treat time-series signal processing problems, have been successfully employed by physicists to handle different types of data. For example, RNN architectures have been utilized to represent hadronic showers

(jets) in collider events, aiding on tasks of identifying different jet types by using their constituents to form sequences [8–10]; have been applied to finding interaction vertices in lepton colliders [11]; and have been explored as monitoring tools for the CERN Large Hadron Collider (LHC) superconducting magnets [12]. In general, RNNs have critical applications also outside the realm of collider physics, having been applied, among others, to gravitational waves detection experiments [13], to neutrino detectors from nuclear reactors [14], and to the reconstruction of quantum dynamics of superconducting qubits [15].

In this paper, we focus on particle physics datasets produced by the LHC at CERN [16]. Collisions within the LHC occur at 40 MHz, making it impossible to readout and store the entire collisions record. To handle this rate, LHC experiments filter out only interesting events through an online selection system called the trigger. These systems are usually set in two stages, where the first stage (Level-1 trigger or L1T) needs to operate at 40 MHz with a latency of $O(1 \mu s)$. The selections performed at these stages need to ensure that interesting events are kept, while discarding common, non-interesting events, which occur several orders of magnitude more frequently than the former. Therefore, utilizing complex algorithms such as RNNs is of utmost importance.

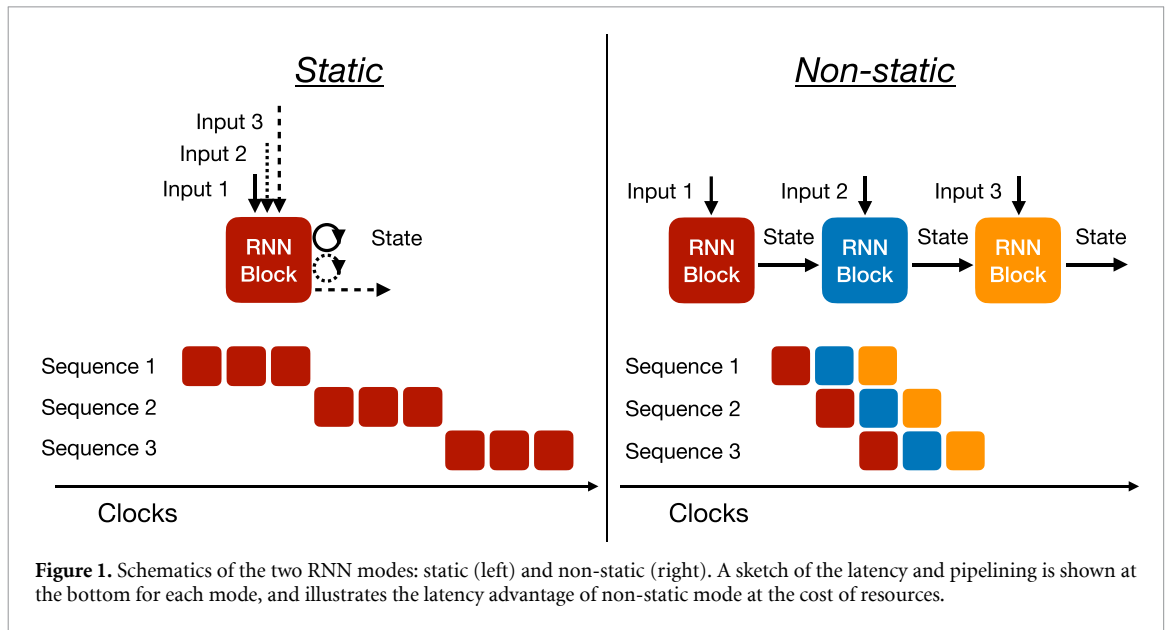
The severe constraints of the trigger prevent the usage of CPUs and GPUs. Instead, custom low-latency hardware such as FPGAs and ASICs must be deployed to meet the latency requirements and offer the flexibility to adapt to changing conditions. These devices are also able to take advantage of high parallelism making their designs both efficient and fast. ML inference in this regime has not seen much support due to its specialized nature, but some tools specifically designed for ultra-low latency inference have emerged [17]. Support in this area has focused primarily on dense and convolutional layers, owing to their versatility and popularity. In this work we present a generalized and flexible implementation of RNNs written in high level synthesis (HLS) for the hls4ml package [18]. The implementation supports a wide range of RNN sizes, design requirements, and is capable of translating both GRUs and LSTMs trained in the Keras framework [19]. Using three different benchmark neural network models of varying size, we show that ultra-low latency inference can be achieved within the resource limitations of modern FPGAs. This integration into hls4ml opens the door for much wider usage of RNNs for low latency applications. While the focus in this work is on applications in physics, we note that there is also a demand for low-power efficient RNNs in industry as well.

2. Related work

Previous work in the realm of fast RNN inference on FPGAs has focused largely on millisecond-latency inferences [20–23]. However, for the applications discussed above we are interested primarily in latencies in the microsecond range, or faster. Some previous work has explored this design space in the context of low-power sparse LSTMs [24], small LSTMs for real-time energy reconstruction [25], RNNs for gravitation-wave experiments [26], and highly quantized RNNs [27]. In contrast, the work in this paper is focused on general support for both large and small LSTMs and GRUs for problems with a range of latency and device constraints. The examples we use are largely chosen from the high energy physics domain, but the applicability is by no means limited to this field. The work is built on top of HLS and the hls4ml framework.

HLS tools are designed to simplify the use of FPGAs by automatically transforming algorithms written in C into the register-transfer level (RTL) [28]. There are multiple advantages of these tools. One substantial advantage is that they allow users without a knowledge of highly technical Verilog/VHDL languages to generate effective RTL [29]. Additionally, they can greatly simplify the effort required in prototyping designs, especially those that are complex. FPGA manufacturers like Xilinx and Altera have their own HLS compilers for their devices. There also exist open-source compilers, such as Catapult HLS [30]. In this work we use the Vivado/Vitis compiler from Xilinx [31, 32].

The hls4ml framework is built on top of HLS compilers, and is capable of converting neural network models into fully-ready HLS projects. The details of the HLS design, in particular the resources and latency, can be controlled through multiple tunable parameters in hls4ml. These are important to allow a flexible design flow that performs well for a wide range of network sizes, architectures, and FPGAs. They also allow the design to be optimized for the target use case. hls4ml already has support for multilayer perceptrons, convolutional neural networks (CNNs), graph neural networks, and several other architectures. Building on hls4ml allows for models with these architectures to be interfaced with the models in this paper. Furthermore, extensive work has been done to hls4ml to ensure that matrix multiplications and other core ML components are optimized. Our work uses the hls4ml design flow in order to leverage these existing framework capabilities.



3. Implementation details

RNNs at their core are comprised of many standard operations in ML. Our implementation relies on this fact to avoid re-implementing any unnecessary operations in the hls4ml framework. Taking the LSTM, we see that each state update requires 4 distinct matrix multiplications, given by

$$\begin{aligned}
 i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
 f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
 o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
 c_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c)
 \end{aligned} \tag{1}$$

where W and U are the weight matrices (denoted as the kernel and recurrent kernel, respectively), b are the biases for the input gate, forget gate, output gate, and cell state, respectively, x_t is the input at time-step t , and h_{t-1} is the computed recurrence value from the previous state. Each of the operations involving W and U in equation (1) are standard matrix-vector multiplications and the activation functions can be taken directly from the hls4ml library and integrated into the LSTM-specific layer implementations. The GRU is composed of two gates (update and reset) and a hidden state, but the weights of the kernel and recurrent kernel for the gates are again packaged together and can thus be handled together with one dense layer call each. The remaining operations to complete a state update for both an LSTM and a GRU are Hadamard products, which is not part of the existing hls4ml library of operations. In this paper, we implemented an HLS-optimized Hadamard product. Because many of the operations in our implementation are taken from the existing hls4ml framework, we are able to trivially support the standard tuning knobs for reuse and precision.

In addition to the pre-existing hls4ml methods for adjusting resources and latency, RNNs introduce an additional possibility which we refer to as ‘static’ and ‘non-static’, shown in figure 1. In static mode, a single RNN block is created, which processes every input for every sequence. This block stores the necessary state vectors internally, and outputs the final result at the end of the sequence. Since there is only one RNN block in static mode, the resources are kept to a minimum. However, the initiation interval (II) of the design increases linearly with the length of the sequence since a new RNN inference cannot begin until the previous inference is complete; in other words, the II is equal to the latency. In contrast to static mode, non-static mode creates RNN blocks for each input in the sequence, and the necessary state vectors are passed from one RNN block to another. This results in a resource utilization that is a factor of the sequence length larger than the resource utilization in static mode. For large RNNs, or inference with long sequences, this means that static mode can be the only viable option. However, for those RNNs for which it is possible to use non-static mode, the II can be dramatically reduced with respect to static mode since non-static mode allows a new inference to begin once the first RNN block has finished processing the first input from the previous inference. This reduces the

II by a factor of the length of the sequence, and thus increases the overall throughput of the RNN layer by the same factor. Further increases in throughput are possible when the II of a single RNN block can be made small since in this case each individual block may be used simultaneously by distinct inferences. While not implemented in this paper, we note that multiple inferences can be cached during static mode when the initiation interval of a single RNN block is less than its latency, thus allowing for higher throughput.

4. Benchmark studies

In order to measure the performance of the hls4ml translation of the RNN-based architectures, we use three different tasks as benchmarks. The sizes of the networks are chosen such that they span a range of use cases, input dimensions, and numbers of weights. The first benchmark is a binary classifier with approximately 4000 parameters, trained to classify jets of particles coming from top-quark decays. The second benchmark is a multi-class classifier, with approximately 50 000 parameters, trained for heavy-flavor jet identification using the reconstructed trajectories of charged particles (tracks) within a jet. The final benchmark is also a multi-class classifier with approximately 130 000 parameters, trained to classify sequences of strokes into five different image classes. Each benchmark is studied with two models using LSTM and GRU recurrent layers, respectively. All the models are trained using Keras and TensorFlow. A summary of each of these models is given in table 1, and details are given in the following sections.

4.1. Top quark tagging

The top quark tagging algorithm is trained to classify top quarks from light-flavor quarks using simulated events generated at $\sqrt{s} = 13$ Te V for comparison to LHC performance. Algorithms designed for this task could be utilized in the Level-1 trigger systems of LHC experiments to help increase the acceptance for these types of interesting decays. Their use would require algorithm latencies of less than approximately a few microseconds in order to fit within system constraints.

The data [33] used for training and testing consists of parton-level scattering processes with top quark-antiquark pair ($t\bar{t}$) and light-flavor quark-antiquark pair ($q\bar{q}$) final states that are generated at leading-order using MadGraph [34] with the NNPDF23LO1 parton distribution functions [35]. The transverse momenta (p_T) of the partons are generated in a window with energy spread given by $\delta p_T/p_T = 0.01$, centered at 1 Te V. These parton-level events are then decayed and showered using Pythia8 [36] (version 8.212) with the Monash 2013 tune [37], including the contribution from the underlying event. A custom detector simulation is used which reproduces the main resolution effects relevant for jet substructure reconstruction through particle level smearing and granularization. We used a configuration for a ‘CMS-like’ detector as described in [38]. Jets are clustered using the anti- k_T algorithm, with a distance parameter of 0.8. Only low level features are used in this study. Particles inside a jet are ordered according to their p_T and up to 20 particles are used in this study. For each particle six features are used: p_T , pseudorapidity (η), azimuthal angle (ϕ), energy, relative angular distance from the jet axis, particle ID given by the generator. The generated events are split into training (95%) and testing (5%), and during training 20% of the training data is used for validation.

Two networks are trained for identifying the jets coming from top-quark decay. The padded sequence of particles, with maximum length of 20, is fed into a recurrent layer with an output size of 20. The output from the final recurrent layer is passed through a dense layer of size 64 before sending it to the output layer. The recurrent layer used sigmoid and hyperbolic tangent activation functions. The activation function for the hidden layers is rectified linear unit (ReLU) [39] while the output layer activation function is a sigmoid function. The binary cross-entropy loss function is minimized with L1 (10^{-5}) and L2 (10^{-4}) regularization of the weights using the Adam algorithm [40] with a learning rate of 2×10^{-4} and a batch size of 246. The two models use different recurrent layers; one uses GRU and the other uses LSTM. There are total 3089 and 3569 trainable parameters for the LSTM and GRU models, respectively.

4.2. Jet flavor tagging

The jet flavor tagging algorithm was trained on Compact Muon Solenoid (CMS) experiment open data samples containing top quark pairs decaying hadronically with center-of-mass energy of 7 Te V [41]. These events are rich in bottom quark jets (b jets), charm quark jets (c jets) and jets from light quarks and gluons (light jets), and so are optimal for training this class of algorithms. Jets are labeled b jets if they contain bottom quarks, c jets if they do not contain bottom quarks but contain charm quarks, and light jets if they do not contain bottom or charm quarks. The main feature that separates b jets (and c jets) from light jets is the presence of the displaced vertex corresponding to the decay of the hadron containing the b (or c) quark.

Table 1. Network hyperparameters and total number of trainable parameters for different benchmark models.

Benchmark	Sequence length	Input vector size	Hidden vector size	Dense layer sizes	output vector size	Trainable parameters		
						Non-RNN layers	LSTM	GRU
Top quark tagging	20	6	20	64	1	1409	2160	1680
Jet flavor tagging	15	6	120	50/10	3	6593	60,960	46,080
QuickDraw	100	3	128	256/128	5	66,565	67,584	51,072

These hadrons can travel significant distances before decaying due to their mass, depending on their momenta. The algorithm proposed here aims to identify the presence of tracks that are consistent with these displaced vertices with the usage of an RNN architecture. This strategy is inspired by the RNNIP algorithm used by the ATLAS experiment [8].

In this study, we consider jets reconstructed with the anti-kt algorithm with a distance parameter of 0.5, with transverse momenta larger than 30 Ge V and absolute pseudorapidities less than 2.0. Tracks with transverse momenta larger than 1 Ge V are associated to the nearest jet according to the angular distance ΔR ; a maximum ΔR of 0.5 is required for association. Tracks within a jet are ordered by the significance of their transverse impact parameter ($\mathcal{S}(d_0)$), and only the first 15 tracks are used by the algorithm. Each track is represented by a vector of the following features: relative transverse momentum ($p_T(\text{track})/p_T(\text{jet})$); $\Delta R(\text{track}, \text{jet})$; transverse and longitudinal impact parameters (d_0, d_z) and their significances ($\mathcal{S}(d_0), \mathcal{S}(d_z)$).

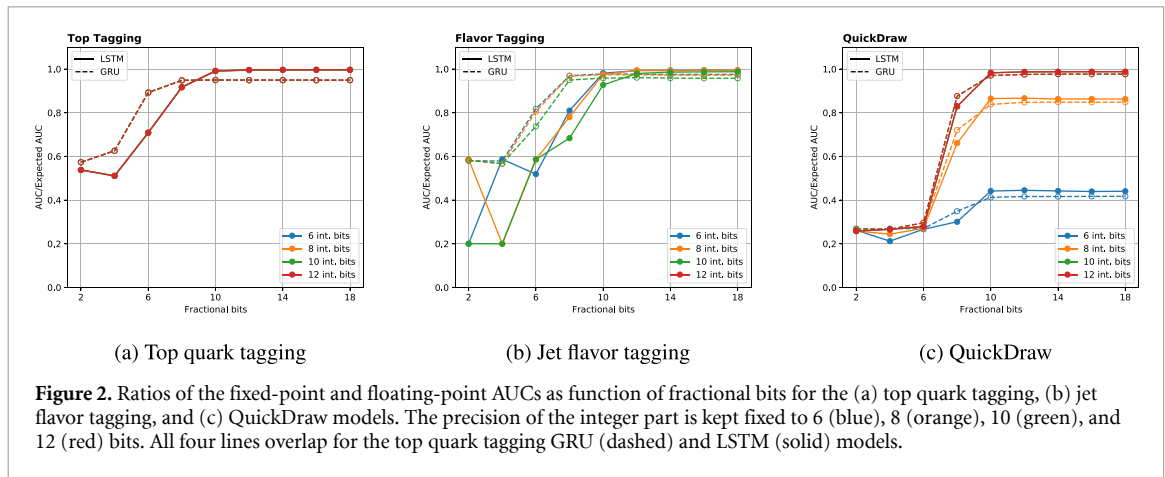
Flavor tagging models were constructed using Keras/TensorFlow, using either GRU or LSTM layers. The padded sequence of tracks, with maximum length of 15, is fed into either one recurrent layer (GRU or LSTM) with 120 hidden units. The recurrent layer outputs a representation of the padded sequence directly into two dense layers with ReLU activation function, with 50 and 10 hidden layers. The following layer outputs the probabilities of a jet to be classified as either a b jet, c jet or light jet; it contains three output nodes with softmax activation function. The training is performed with a categorical cross-entropy loss, with 30% of the training data retained as the validation dataset and used for early stopping based on the accuracy metric. The GRU (LSTM) architecture contains 52 673 (67 553) trainable parameters, of which 46 080 (60 960) are in the recurrent layer.

4.3. Quickdraw dataset

The QuickDraw dataset [42] is a collection of 50 million drawings in 345 categories created by Google and contributed by players of the game Quick! Draw. In this game, users are asked to draw a specified drawing in under 15 s. The drawings are recorded as a time-stamped sequence of the strokes from which the drawing is created. For each 15 s stroke the x and y coordinates of the pen are recorded 100 times. The coordinates along with the timestamp make up the network inputs. While 345 different drawing categories exist, we use only 5 for our tests; these are *ants*, *butterflies*, *bees*, *mosquitos* and *snails*. Contrary to other popular representations of images, the QuickDraw dataset is completely stroke-based. We train two RNNs to classify these sequences into each respective category. We use these networks as proxies for any networks acting on large sequences of low-dimensional inputs. For example, these networks could be used to identify and classify tracks based on the sequence of their hits, or incident particles based on their showers in finely-segmented calorimeters. For example, networks developed by the ATLAS experiment to identify showers originating from pions take as input a low dimensional set of up to 100 clusters [43]. Although these particular applications are not appropriate for the Level-1 trigger environment, running these algorithms at later stages in the trigger for a whole event could still require low latencies under a millisecond depending on the exact application.

The two networks process the sequence of 100 stroke inputs from the QuickDraw dataset with a recurrent layer whose output size is 128. The final recurrent layer output is passed through two dense layers of sizes 256 and 128, respectively, before being sent to the final output layer. Dropout layers are placed before the two dense layers to regularize during training. The recurrent layer uses a hyperbolic tangent activation function, the dense layers use ReLU activations, and the output is a five-class softmax layer. The only difference in the two networks is that in one the recurrent layer is a GRU and in the other it is an LSTM. These networks have total sizes of 117 637 and 134 149 trainable parameters, respectively. The two networks perform well and show top-1 area under the curve (AUC)⁵ that are nearly identical, approximately 99% for each of the five classes.

⁵ AUC measures the area under the receiver operating characteristic (ROC) curve.



5. Performance, resource and latency estimation

The models described in section 4 are translated into HLS using the hls4ml framework. Vivado HLS 2019.2 is used for HLS synthesis with the synthesis clock frequency set to 200 MHz. For the top quark tagging and jet flavor tagging models we use a Xilinx Kintex UltraScale FPGA (part number xcku115-f1vb2104-2-i) as the target device and for the Quickdraw models we use a Xilinx Alveo U250 (part number xcu250-figd2104-2-e). For each of the three benchmark models a range of different settings are considered simultaneously to accurately profile the full design space. These settings modify the quantization of the model by adjusting the fixed-point data type and modify the degree of parallelism of the design by using the hls4ml reuse parameter. Finally, we also consider the static and non-static implementations discussed in section 3.

5.1. Quantization

The weights and biases in trained models are typically stored with 32-bit floating-point precision. However, 32-bit floating-point calculations are often not required for optimal network inference, and are costly to implement on FPGAs. Other quantization techniques can offer more efficient ways of compressing neural networks by reducing the number of bits used to represent the weights and biases, ideally with no or minimal loss in performance. In hls4ml, all the inputs, weights, biases, sums, and outputs of each layer are represented as fixed-point numbers. In this scheme the amount of bits used to store the integer and decimal components of the number are configured, such that, for example, an unsigned fixed point number with 4 integer bits and 3 fractional points is capable of storing values between 0 and 15.875 with a granularity of 0.125. The total numbers of bits is also referred to as the precision of the fixed-point number. Hls4ml allows a different precision to be chosen for the computations and internal values of each individual layer; for the sake of consistency we fix the precision to be the same for all layers in the scans below. We do find that it is necessary to increase the precision and size of the lookup table (LUT) used for the softmax computation at the end of the flavor-tagging and QuickDraw models, but this has a minimal impact on the overall resource usage.

The optimal precision for each model depends on the training details, the specific task, and the inputs. All the models are quantized only after training, a method referred to as ‘post-training quantization’ (PTQ). For each model we profile the performance of the synthesized design from hls4ml as a function of the bit precision of the weights and activation functions. Figure 2 shows the ratio of the AUC from the quantized model to the AUC from the floating-point model as a function of fractional bits while keeping the precision of the integer part fixed to 6, 8, 10, or 12 bits.

The best performance for each model, measured by the AUC ratio, is generally achieved with at least 10 fractional bits irrespective of the values of the integer bit. For the top quark and flavor tagging models, 6 integer bits are sufficient, while the QuickDraw models require at least 10 integer bits. For further results in this paper, we fix the integer bits for each model to these values. We note that there is a small performance degradation in the GRU models after quantization for all three benchmark cases. The difference is particularly visible for the top quark tagging model, but it is less than 5%. It is possible that quantization-aware training or sequence masking could potentially enable models with lower precision to perform as well as or better than the ones we present in this paper.

Table 2. Minimum and maximum latencies for the top quark tagging model.

Model	Latency (μs)	$R = (6, 5)$ (μs)	$R = (12, 10)$ (μs)	$R = (30, 20)$ (μs)	$R = (60, 60 [40])$ (μs)
GRU	1.7–1.7	2.4–6.5	3.2–7.3	5.0–9.1	8.0–12.1
LSTM	1.7–1.7	2.7–6.8	3.5–7.6	5.3–9.4	8.3–12.4

Table 3. Minimum and maximum latencies for the jet flavor tagging model.

Model	$R = (48, 40)$ (μs)	$R = (90, 60)$ (μs)	$R = (120, 120)$ (μs)	$R = (240, 240)$ (μs)
GRU	6.7–24.8	9.8–27.9	11.5–29.6	20.5–38.6
LSTM	6.9–25.0	10.1–28.2	11.7–29.8	20.7–38.8

Table 4. Minimum and maximum latencies for the QuickDraw model.

Model	$R = (48, 32)$ (μs)	$R = (96, 64)$ (μs)	$R = (192, 128)$ (μs)	$R = (384, 384 [256])$ (μs)
GRU	35.4–164.0	59.4–188.0	107.0–235.0	203.0–331.0
LSTM	35.9–164.0	59.9–188.0	107.0–236.0	203.0–332.0

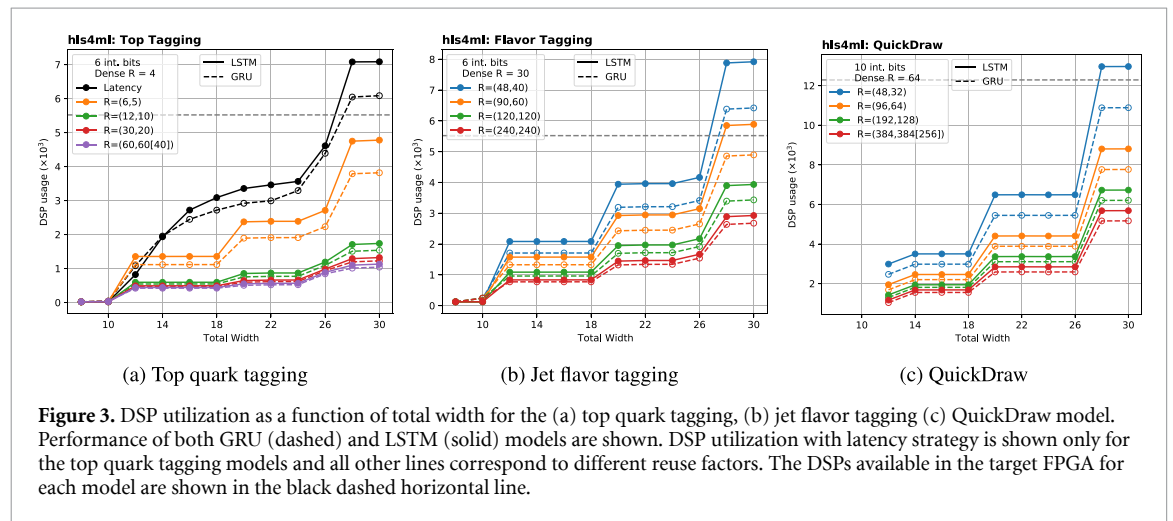
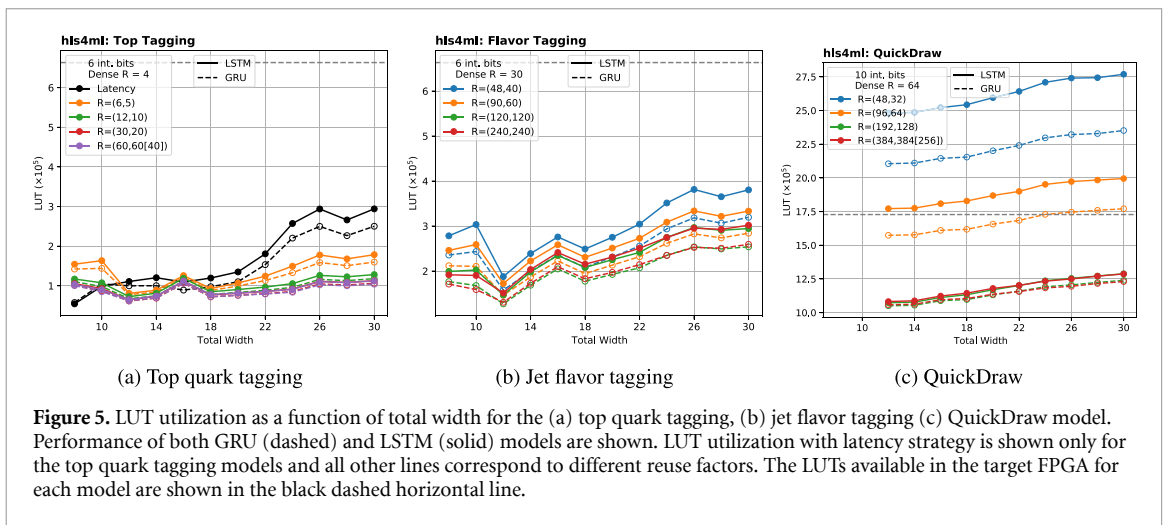
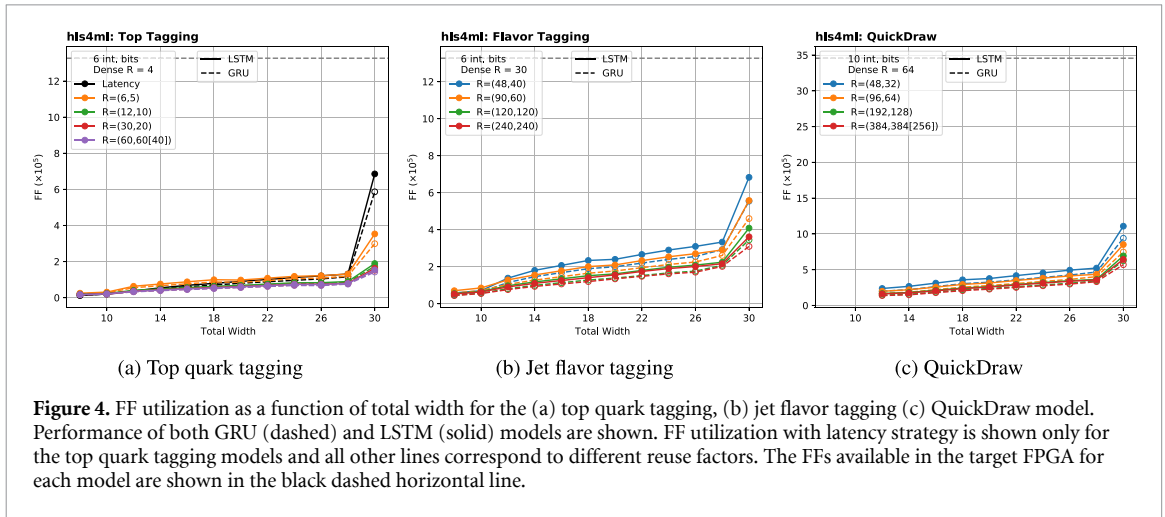


Figure 3. DSP utilization as a function of total width for the (a) top quark tagging, (b) jet flavor tagging (c) QuickDraw model. Performance of both GRU (dashed) and LSTM (solid) models are shown. DSP utilization with latency strategy is shown only for the top quark tagging models and all other lines correspond to different reuse factors. The DSPs available in the target FPGA for each model are shown in the black dashed horizontal line.

5.2. Parallelization

The other main tuning knob besides the precision is the amount of parallelism employed during weight matrix multiplication. This is controlled in hls4ml through a parameter called ‘reuse’. Specifically, reuse is the number of multiplication operations each digital signal processing (DSP) block must do for a given matrix multiply. Setting reuse to 1, i.e. the fully parallel case, means that each multiplication is done by its own DSP and can happen simultaneously. Increasing the reuse factor reduces the number of DSPs that are required, but increases the latency and initiation interval of the layer computation in proportion to the reuse. All three benchmark models are synthesized with different values of the reuse factor (R) and fractional bit precision. The results are expressed for different FPGA resource categories: onboard FPGA memory (BRAM), DSPs, and registers and programmable logic like flip-flops (FFs) and LUTs. In hls4ml, a model can either be synthesized to minimize the latency (*latency strategy*) or the resource utilization (*resource strategy*). For large models with 40 k or more trainable parameters it becomes difficult to synthesize the models with the latency strategy, and so resource strategy must be used. With resource strategy the design is optimized for low resource utilization by reusing existing hardware to complete operations in multiple stages. Out of the three benchmark models only the top quark tagging model is small enough to be synthesized with both latency and resource strategies, whereas only resource strategy is used for the other two models. The minimum and maximum latencies for each model are shown in tables 2–4. The amounts of DSPs, FFs, and LUTs for each model are shown for different reuse factor values in figures 3–5, respectively. The resource utilization is shown as a function of total width, which is the sum of the integer and fractional bits used to represent the weights and biases of each layer of the neural network. In these results the reuse factor values are written in the form $R = (X, Y)$, where X and Y correspond to the reuse factors for the kernel and recurrent kernel matrix multiplications discussed in equation (1). The numbers shown in the square brackets correspond to



reuse factor for the LSTM layer in the case when the reuse factor differs between the LSTM and GRU implementations.

We observe that all resources generally increase with smaller values of R and increased precision. In the case of FFs and LUTs, this increase is roughly linear, while for DSPs the utilization remains flat until the precision exceeds the DSP input width. The latency, on the other hand, follows a scaling inverse to that of the FFs and LUTs with respect to the reuse. Thus, as with other architectures supported under hls4ml, reuse can be used to reduce FFs and LUTs at the expense of latency. This simple scaling is critical for allowing users to tune the resource usage and latency such that the synthesized designs to meet desired requirements. The latency strategy adds another finer option to this tuning space for latency-limited tasks, but comes at the cost of larger resource usage. As expected we find that the GRU models use approximately 1/4 less resources when compared to the LSTM models. This is a result of the 3:4 ratio between the number of matrix multiplications in GRU and LSTM models. Finally, it is important to note that the results shown in this paper are from HLS synthesis. When running Vivado synthesis we observe a reduction in LUT usage between 20% and 65% and in FF usage between 10% and 20%. This is particularly important to note in the case of the larger flavor tagging and QuickDraw models where the estimated LUT usages from HLS synthesis are quite large.

For the top quark tagging models we observe that designs with maximal quantized performance can be implemented on one SLR of a Xilinx Virtex Ultrascale+ VU9P board, the planned future device for an upgrade to the CMS Level-1 trigger [44]. We observe slightly larger resource usage for the flavor tagging models, as expected, but still within the resource constraints of a single SLR of a VU9P board. In both cases the latencies for the designs are also within the task requirements. For the QuickDraw models, the estimates from Vivado synthesis suggest that maximal quantized performance could be implemented on a Xilinx Alveo U250, a popular device for the types of coprocessor applications we envision for these models. Extrapolating from the initiation interval (II), we find the average throughput of the QuickDraw LSTM model is between 4300 to 9700 events/second. Tests of the batch 1 inference for the same model using an Nvidia Tesla V100 GPU yield a throughput of 660 events/second. Increasing the batch size to 10 increases the throughput to

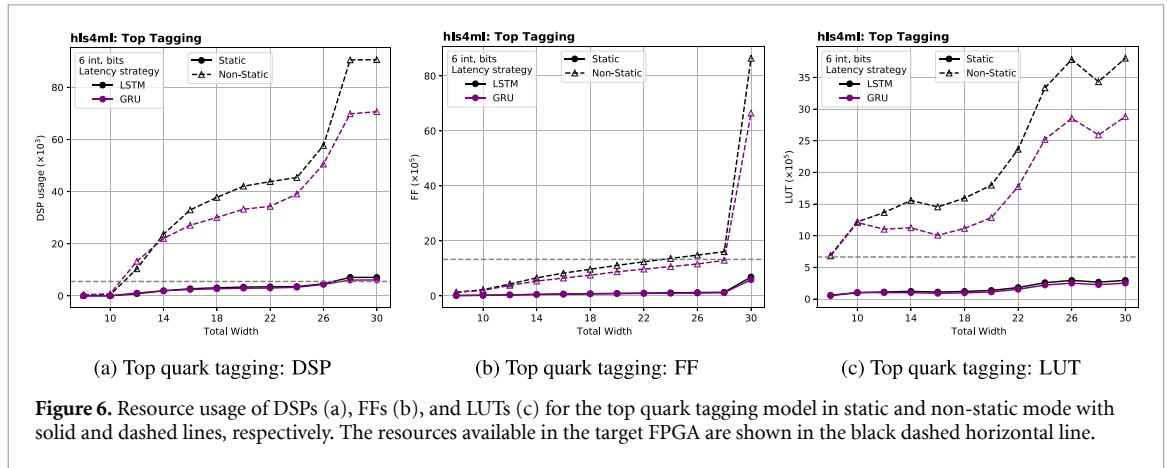


Figure 6. Resource usage of DSPs (a), FFs (b), and LUTs (c) for the top quark tagging model in static and non-static mode with solid and dashed lines, respectively. The resources available in the target FPGA are shown in the black dashed horizontal line.

Table 5. Minimum and maximum latencies and initiation intervals for the top quark tagging model in both static and non-static mode.

Model	Static latency (μs)	Non-static latency (μs)	Static II	Non-static II
GRU	1.7–1.7	1.6–1.6	315	1
LSTM	1.6–1.6	1.5–1.5	314	1

7700 events/second, comparable with the FPGA throughput. The throughput increases further by a factor of five to approximately 30 000 if the batch size is increased to 100. While it is unsurprising that GPU inference at large batch sizes is able to outperform an FPGA, many physics tasks are inherently low-batch problems. This is because each event must be processed separately and latency is extremely important, therefore the maximal batch size is dictated by the amount of inferences necessary only for a single event. For example, algorithms to classify particle-induced showers in a detector need only to be run once every event if the algorithm can use the full detector information in one pass. Thus, a factor of ten improvement in the FPGA inference for batch-1 inference is highly relevant for future trigger applications.

5.3. Static and non-static comparison

In order to study the impacts of the static and non-static modes discussed in section 3 we limit our consideration to the top quark tagging models. As shown in figure 6, resource usage for non-static mode increases dramatically compared to static mode. For even moderate-sized models, non-static mode requires too many resources to be feasible. In the case of the top quark tagging model we see that non-static mode is able to fit within the available resources of the chip only for very small bitwidths. However, table 5 confirms that although non-static mode offers similar overall latency to static mode, the initiation interval (II) in non-static mode is reduced from 315 (314) to 1 for the GRU (LSTM) models. This results in a increased throughput for non-static mode by a factor of more than 300. The increased throughput of non-static mode would be vital for Level-1 trigger applications that run inferences at rates of up to 40 MHz. While this particular top quark tagging model suffers in performance using a total bitwidth of 10 (6 integer and 4 fractional bits in this case), there are multiple options, such as per-layer quantization or quantization-aware training, that were not considered for this study but could potentially allow a performant version of this model to be synthesized in non-static mode.

6. Summary and outlook

RNNs have shown substantial success for many tasks in particle physics. They are particularly well-suited to those problems involving sequences of particle or detector signals, outperforming densely connected deep neural networks (DNNs) [45] and convolutional neural networks (CNNs) [46] on certain jet classification tasks. In spite of this success, RNNs have not seen the widespread adoption in ultra-low latency environments in physics when compared to DNNs and CNNs. This difference is owed in part to tools such as hls4ml that simplify the adaptation of the latter models from Keras to HLS. The support for GRUs and LSTMs in hls4ml that we present in this work represents the removal of a major barrier to the use of RNNs in ultra-low latency environments. This has ramifications not only for high energy physics but also other research areas where RNNs have become popular. While we have focused on the usage of hls4ml with FPGAs, it is important to note that hls4ml can also be used to create ASIC designs [47], and thus this work also allows for the possibility of RNN usage on ASICs as well.

The implementation we present in hls4ml in this work maintains the main tuning features of hls4ml, namely the reuse factor and per-layer bit precision. This is necessary to allow the customization of the synthesized design to meet the needs of a given task. The benchmark models chosen cover a range of sizes, latencies, and problems, and showcase the quality of the hls4ml support for a variety of realistic scenarios. We also add an RNN-specific tuning parameter to hls4ml called the RNN mode, with static and non-static settings capable of further adjusting the behavior of the synthesized design. While we show that this work is capable of producing results with high accuracy, there are multiple possibilities for future development. In particular, we observe that even small RNN models can require a substantial amount of resources to implement. While the PTQ scheme we have used here is able to minimize resources to a certain extent, other methods, such as quantization-aware training, have shown that even more resource reduction can be possible with little to no cost to performance. This is perhaps even more true for RNNs than dense neural networks due to the repeated use of the recurrent layer weights. Other techniques such as masking are also a possible method for reducing both resource usage and dependence on small weight values (high bit precision).

The recurrent or repeating nature of many modern algorithms, such as RNNs, transformers and graph neural networks, make them very difficult to be run, particularly at low latency, on FPGAs. In this work, we present the successful deployment of RNNs in models with number of trainable parameters ranging from $\mathcal{O}(1\text{ k})$ to $\mathcal{O}(100\text{ k})$ achieving latencies of $\mathcal{O}(1\ \mu\text{s})$ to $\mathcal{O}(100\ \mu\text{s})$. This represents an important step in enabling support in hls4ml for more complex architectures with recursive computations.

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: <http://opendata.cern.ch/record/204>, <https://zenodo.org/record/3601436>, <https://github.com/googlecreativelab/quickdraw-dataset#sketch-rnn-quickdraw-dataset>.











Acknowledgments

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. We thank Javier Duarte, Maurizio Pierini, Zhiqiang Que and Nhan Tran for their valuable comments and suggestions. M Kagan and R Teixeira de Lima are supported by the US Department of Energy (DOE) under Grant DE-AC02-76SF00515. P Harris and D Rankin acknowledge DOE Grant DE-SC0021943 and National Science Foundation (NSF) Grants Nos. 1934700 and 1931469. S-C Hsu and E Khoda are supported by NSF Grants Nos. 2117997 and 1934360.

Code availability statement

The hls4ml library is available at <https://github.com/fastmachinelearning/hls4ml> and RNN support is available as of commit <https://github.com/fastmachinelearning/hls4ml/commit/59ed8249f4bbdb4b23ff0c6f0bfc976b44d3ac7e>. For examples on how to use hls4ml, the notebooks in <https://github.com/fastmachinelearning/hls4ml-tutorial> serve as a general introduction.

ORCID iDs

Elham E Khoda  <https://orcid.org/0000-0001-8720-6615>
Dylan Rankin  <https://orcid.org/0000-0001-8411-9620>
Rafael Teixeira de Lima  <https://orcid.org/0000-0001-5545-6513>
Philip Harris  <https://orcid.org/0000-0001-8189-3741>
Scott Hauck  <https://orcid.org/0000-0001-9516-0311>
Shih-Chieh Hsu  <https://orcid.org/0000-0001-6214-8500>
Michael Kagan  <https://orcid.org/0000-0002-3386-6869>
Vladimir Loncar  <https://orcid.org/0000-0003-3651-0232>
Sioni Summers  <https://orcid.org/0000-0003-4244-2061>
Caterina Vernieri  <https://orcid.org/0000-0002-0235-1053>

References

- [1] Duarte J et al 2018 Fast inference of deep neural networks in FPGAs for particle physics *J. Instrum.* **13** 07027
- [2] Ngadiuba J et al 2020 Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml *Mach. Learn.: Sci. Technol.* **2** 015001

- [3] Årrestad T et al 2021 Fast convolutional neural networks on FPGAs with hls4ml *Mach. Learn.: Sci. Technol.* **2** 045015
- [4] Rankin D S et al 2020 FPGAs-as-a-service toolkit (FaaS) 2020 *IEEE/ACM Int. Workshop on Heterogeneous High-Performance Reconfigurable Computing (H2RC)*
- [5] Hochreiter S and Schmidhuber J 1997 Long short-term memory *Neural Comput.* **9** 1735–80
- [6] Cho K et al 2014 Learning phrase representations using RNN encoder-decoder for statistical machine translation *CoRR* (arXiv:1406.1078)
- [7] Pascanu R, Mikolov T, and Bengio Y 2012 On the difficulty of training recurrent neural networks (arXiv:1211.5063)
- [8] ATLAS Collaboration 2017 Identification of jets containing b -hadrons with recurrent neural networks at the atlas experiment (available at: <https://inspirehep.net/literature/1795312>)
- [9] ATLAS Collaboration 2019 Identification of hadronic tau lepton decays using neural networks in the atlas experiment (available at: <https://inspirehep.net/literature/1795210>)
- [10] de Lima R T 2021 Sequence-based machine learning models in jet physics (arXiv:2102.06128)
- [11] Goto K et al 2021 Development of a vertex finding algorithm using recurrent neural network (arXiv:2101.11906)
- [12] Wielgosz M, Skoczeń A and Mertik M 2017 Using lstm recurrent neural networks for monitoring the lhc superconducting magnets *Nucl. Instrum. Methods Phys. Res. A* **867** 40–50
- [13] Schmitt A, Fu K, Fan S and Luo Y 2019 Investigating deep neural networks for gravitational wave detection in advanced ligo data *Proc. 2nd Int. Conf. on Computer Science and Software Engineering* (New York: Association for Computing Machinery) pp 73–78
- [14] Li A et al 2022 KamNet: an integrated spatiotemporal deep neural network for rare event search in KamLAND-Zen (arXiv:2203.01870)
- [15] Flurin E, Martin L S, Hacoen-Gourgy S and Siddiqi I 2020 Using a recurrent neural network to reconstruct quantum dynamics of a superconducting qubit from physical observations *Phys. Rev. X* **10** 011006
- [16] CERN Accelerating science (n.d.) 2016 (available at: <https://home.cern/science/accelerators/large-hadron-collider>)
- [17] Umuroglu Y et al 2017 FINN *Proc. 2017 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (ACM)*
- [18] FastML Team fastmachinelearning/hls4ml 2022 (available at: <https://github.com/fastmachinelearning/hls4ml/tree/main>)
- [19] Chollet F et al 2015 Keras (available at: <https://keras.io>)
- [20] Heelan C, Nurmikko A V and Truccolo W A 2018 Fpga implementation of deep-learning recurrent neural networks with sub-millisecond real-time latency for bci-decoding of large-scale neural sensors (104 nodes) *2018 40th Annual Int. Conf. IEEE Engineering in Medicine and Biology Society (EMBC)* pp 1070–3
- [21] Chang A X M, Martini B and Culurciello E 2015 Recurrent neural networks hardware implementation on FPGA *CoRR* (arXiv:1511.05552)
- [22] Lee M et al 2016 Fpga-based low-power speech recognition with recurrent neural networks *CoRR* (arXiv:1610.00552)
- [23] Fowers J et al 2018 A configurable cloud-scale dnn processor for real-time AI *Proc. 45th Annual Int. Symp. on Computer Architecture (ISCA)* (IEEE Press) pp 1–14
- [24] Han S et al 2016 ESE: efficient speech recognition engine with compressed LSTM on FPGA *CoRR* (arXiv:161200694)
- [25] Aad G et al 2021 Artificial neural networks on FPGAs for real-time energy reconstruction of the atlas LAr calorimeters *Comput. Softw. Big Sci.* **5** 19
- [26] Que Z et al 2021 Accelerating recurrent neural networks for gravitational wave experiments *32nd IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors* p 6
- [27] Rybalkin V et al 2018 Finn-l: library extensions and design trade-off analysis for variable precision lstm networks on FPGAs (arXiv:1807.04093)
- [28] Coussy P and Morawiec A 2008 *High-Level Synthesis* 1st edn (Dordrecht: Springer)
- [29] Nane R et al 2016 A survey and evaluation of FPGA high-level synthesis tools *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **35** 1591
- [30] Mentor/Siemens 2020 Catapult high-level synthesis (available at: www.mentor.com/hls-lp/catapult-high-level-synthesis)
- [31] AMD/Xilinx 2022 Vivado (available at: www.xilinx.com/support/university/vivado.html)
- [32] Feist T 2012 Vivado design suite *White Paper* 5 30
- [33] Pierini M, Duarte J, Tran N and Freytsis M 2020 HLS4ML LHC Jet dataset (30 particles) (available at: <https://zenodo.org/record/3601436#Y95r7uxByeB>)
- [34] Alwall J et al 2014 The automated computation of tree-level and next-to-leading order differential cross sections and their matching to parton shower simulations *J. High Energy Phys.* **JHEP07(2014)079**
- [35] Ball R D et al 2013 Parton distributions with LHC data *Nucl. Phys. B* **867** 244–89
- [36] Sjöstrand T, Ask S, Christiansen J R, Corke R, Desai N, Ilten P, Mrenna S, Prestel S, Rasmussen C O and Skands P Z 2015 An introduction to pythia 8.2 *Comput. Phys. Commun.* **191** 159–77
- [37] Skands P, Carrazza S and Rojo J 2014 Tuning pythia 8.1: the monash 2013 tune *Eur. Phys. J. C* **74** 3024
- [38] Coleman E, Freytsis M, Hinzmann A, Narain M, Thaler J, Tran N and Vernieri C 2018 The importance of calorimetry for highly-boosted jet substructure *J. Instrum.* **13** T01003
- [39] Nair V and Hinton G E 2010 Rectified linear units improve restricted boltzmann machines *ICML 2010* pp 807–14
- [40] Kingma D P and Ba J 2015 Adam: a method for stochastic optimization *3rd Int. Conf. on Learning Representations, ICLR 2015 (San Diego, CA, USA, 7–9 May 2015, Conf. Track Proc.)*, ed Y Bengio and Y LeCun
- [41] Sander C and Schmidt A MC: TTbar sample from the CMS HEP Tutorial (available at: <http://opendata.cern.ch/record/204>)
- [42] Google The Quick, Draw! Dataset (available at: <https://github.com/googlecreativelab/quickdraw-dataset#sketch-rnn-quickdraw-dataset>)
- [43] ATLAS Collaboration 2022 *Point Cloud Deep Learning Methods for Pion Reconstruction in the ATLAS Experiment* ATL-PHYS-PUB-2022-040 CERN (available at: <https://cds.cern.ch/record/282537>)
- [44] CMS Collaboration 2020 The Phase-2 upgrade of the CMS Level-1 trigger *CMS Technical Design Report* CERN-LHCC-2020-004, CMS-TDR-021 (CERN)
- [45] Egan S et al 2017 Long Short-Term Memory (LSTM) networks with jet constituents for boosted top tagging at the LHC (arXiv:1711.09059)
- [46] Fraser K and Schwartz M D 2018 Jet charge and machine learning *J. High Energy Phys.* **JHE10(2018)093**
- [47] Di Guglielmo G et al 2021 A reconfigurable neural network ASIC for detector front-end data compression at the HL-LHC *IEEE Trans. Nucl. Sci.* **68** 2179–86



Algean: An Open Framework for Deploying Machine Learning on Heterogeneous Clusters

NAIF TARAFDAR, University of Toronto, Canada

GIUSEPPE DI GUGLIELMO, Columbia University, USA

PHILIP C. HARRIS, Massachusetts Institute of Technology, Institute for Artificial Intelligence and Fundamental Interactions, USA

JEFFREY D. KRUPA, Massachusetts Institute of Technology, USA

VLADIMIR LONCAR, CERN, Switzerland

DYLAN S. RANKIN, Massachusetts Institute of Technology, USA

NHAN TRAN, Fermilab, USA

ZHENBIN WU, University of Illinois, USA

QIANFENG SHEN and PAUL CHOW, University of Toronto, Canada

Algean, pronounced like the sea, is an open framework to build and deploy machine learning (ML) algorithms on a heterogeneous cluster of devices (CPUs and FPGAs). We leverage two open source projects: *Galapagos*, for multi-FPGA deployment, and *hls4ml*, for generating ML kernels synthesizable using Vivado HLS. *Algean* provides a full end-to-end multi-FPGA/CPU implementation of a neural network. The user supplies a high-level neural network description, and our tool flow is responsible for the synthesizing of the individual layers, partitioning layers across different nodes, as well as the bridging and routing required for these layers to communicate. If the user is an expert in a particular domain and would like to tinker with the implementation details of the neural network, we define a flexible implementation stack for ML that includes the layers of Algorithms, Cluster Deployment & Communication, and Hardware. This allows the user to modify specific layers of abstraction without having to worry about components outside of their area of expertise, highlighting the modularity of *Algean*. We demonstrate the effectiveness of *Algean* with two use cases: an autoencoder, and ResNet-50 running across 10 and 12 FPGAs. *Algean* leverages the FPGA's strength in low-latency computing, as our implementations target batch-1 implementations.

This work was supported by Xilinx, NSERC, and CMC Microsystems. P. C. Harris, D. S. Rankin, and J. D. Krupa were partially supported by NSF grants 1934700 and 1931469, and the IRIS-HEP grant NSF 1836650. Cloud computing by NSF 1904444, grant. Additionally, support was received from the NSF Institute for AI and Fundamental Interactions (Cooperative Agreement PHY-2019786).

Authors' addresses: N. Tarafdar, Q. Shen, and P. Chow, University of Toronto, 10 King's College Road, Toronto, Ontario M1L 0B4, Canada; emails: {naif.tarafdar, qianfeng.shen}@mail.utoronto.ca, pc@eecg.toronto.edu; G. Di Guglielmo, Columbia University, Computer Science Building, 1214 Amsterdam Avenue, Mail Code 0401, New York, NY 10027, USA; email: giuseppe.diguglielmo@columbia.edu; P. C. Harris, Massachusetts Institute of Technology, 77 Massachusetts Ave, Cambridge, MA 02139, USA, Institute for Artificial Intelligence and Fundamental Interactions, Massachusetts Institute of Technology, 77 Massachusetts Ave, Cambridge, MA 02139, USA; email: pcharris@mit.edu; J. D. Krupa and D. S. Rankin, Massachusetts Institute of Technology, Massachusetts Institute of Technology, 77 Massachusetts Ave, Cambridge, MA 02139, USA; emails: {krupa, drankin}@mit.edu; V. Loncar, 1211 Geneva, 23, Switzerland; email: vladimer.loncar@cern.ch; N. Tran, Fermilab, PO Box 500, Batavia, IL 60510-5011, USA; email: ntran@fnal.gov; Z. Wu, University of Illinois, 845 W. Taylor St., Chicago, IL 60607, USA.; email: zhenbin.wu@cern.ch.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

1936-7406/2021/12-ART23 \$15.00

<https://doi.org/10.1145/3482854>

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Hardware**; • **Computing methodologies** → **Machine learning**; **Parallel computing methodologies**;

Additional Key Words and Phrases: FPGAs, data center, hardware/software co-design

ACM Reference format:

Naif Tarafdar, Giuseppe Di Guglielmo, Philip C. Harris, Jeffrey D. Krupa, Vladimir Loncar, Dylan S. Rankin, Nhan Tran, Zhenbin Wu, Qianfeng Shen, and Paul Chow. 2021. *Algean*: An Open Framework for Deploying Machine Learning on Heterogeneous Clusters. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 23 (December 2021), 32 pages.

<https://doi.org/10.1145/3482854>

1 INTRODUCTION

The interest in using FPGAs for computing at scale has become desirable because of the need for increased performance and reducing power. The flagship example of this is the Microsoft Catapult project that has led to an FPGA being deployed in every Microsoft server [1]. FPGAs at Microsoft are used for search engine acceleration, in a **machine learning (ML)** framework for applications within the data center as well as for many network and packet-processing tasks.

A distinguishing feature of the Catapult architecture is that the FPGAs can directly communicate with other FPGAs and CPUs as *peers* on the network versus the more common *accelerator* model for FPGAs where the FPGAs are attached to a CPU and only accessible through the CPU. The peer model is more efficient for applications that are large enough to span multiple FPGAs requiring low-latency communication between the FPGAs. Although Microsoft has shown significant success in scaling up and using multiple FPGAs in a single application, such as Project Brainwave [2, 3] used for real-time AI, there is no public description of how the applications are built and deployed to the FPGAs, and the platform and tools are not available for others to build their own applications. There is also no known equivalent open source platform available where someone can build their own version of Brainwave.

Brainwave has shown how useful multi-FPGA implementations can be as they leverage having all their weights in on-chip memory as opposed to accessing memory in off-chip DRAM. Having a framework to be able to build custom circuits, like the one in Brainwave, will allow users to create their own networks, which at the moment is quite difficult due to the lack of abstractions within FPGA systems. On top of being able to access on-chip memory, within an infinitely large fabric available through a multi-FPGA framework we could unroll all our computations completely, or to any desired level of unrolling. This will enable the construction of very low latency, high throughput networks that can run at batch 1. In this work, it is our hope to provide the abstraction of an infinitely large FPGA fabric by abstracting the difficulties of network-connected FPGAs. This article leads to a broad range of possible applications where low-latency, large AI inference is needed to process information in real time. Examples include systems controls, web search, real-time physics applications, and medical image processing.

For this work, we define a cluster of network-connected FPGAs, (i.e., all FPGAs have direct connections to the network) as a *multi-FPGA cluster*. By this definition, Brainwave is a multi-FPGA application.

The focus of this article is to describe how we created *Algean*, which is an open source platform that can be used to build multi-FPGA ML applications on multi-FPGA clusters. *Algean* provides the user with multiple layers of abstraction. The user can use *Algean* as a black box that takes neural net descriptions as inputs and get an output of programmed FPGAs. Our black box is responsible for the creation of IP cores, communication protocols, partitioning the neural net across multiple devices, and finally generating the final bitstreams of all FPGAs. Our focus with *Algean* is ease

of use as well as modularity. However, the parts of the black box are implemented as a stack of abstraction layers and can be further customized by users who are experts in the various layers. This stack is built in modular pieces, which also allows for alternative implementations at each layer. In particular, a user can modify the architecture of a particular convolution layer, and implement the layer in hardware on an FPGA or in software on a processor. The communications layer can be modified to use different protocols, such as UDP, TCP/IP, layer 2 Ethernet, PCIe, parallel buses between devices, or any custom protocol. A change at any of the layers of *Algean* does not affect any of the other layers, especially the application layer at the top of the stack. This provides portability between platforms, particularly across different types of FPGAs.

We started with two open source projects: *hls4ml* [4] and Galapagos [5, 6]. By using *hls4ml*, we can convert ML descriptions into C++ code synthesizable with **high-level synthesis (HLS)**. Galapagos is a framework for deploying streaming computation kernels onto a cluster of heterogeneous devices [5, 6], especially FPGAs, which are particularly suited to streaming computation. We define streaming as communication via streams of data moving from one kernel to another where the processing is effectively done *on-the-fly* versus a mechanism like a source kernel writing to memory and the destination kernel reading from that memory.

Although conceptually *Algean* is a combination of two existing platforms, a significant effort was required to integrate the two platforms. Initially, the idea seemed straightforward, but when considering the details, much more is required. *Hls4ml* was not initially designed to build the layers of the network as individual cores and required significant enhancements to enable the output of separate cores. The interfaces between layers needed to comply with the streaming interfaces required by Galapagos, and the output cores had to be put into a directory structure suitable for processing by Galapagos. Galapagos had not been tested with a large application, and the deployment of a large neural network was the first attempt at doing so. We then realized that for very large applications, an automated partitioner is required and Galapagos was enhanced to have a new layer that can do the partitioning. The first partitioner is only enough to build *Algean*, but significant future work can enhance it in many ways. These contributions would not have come to light without *Algean* and are important considerations for developing future application frameworks that leverage Galapagos.

An important contribution of this work is to describe that effort and more generally show the challenges of building multi-FPGA application frameworks that can be customizable and portable across multiple kinds of FPGA hardware. The main outcome is an ML platform that enables ML practitioners to use familiar tools and map them to a multi-FPGA cluster without needing to do any hardware design. We contrast *Algean* with the current vendor ML flows [7, 8] that only target a few FPGAs hosted in a single server and lack the ability to scale easily.

Our contributions in this work are as follows:

- (1) A fully push-button flow to take an ML network input from popular ML tools and deploy the network to a multi-FPGA/CPU back-end. Abstracted away from the user is the creation of the hardware IP cores for the given ML network, the partitioning of these IP cores, and the connecting and routing between them. Some of the core functionality was already handled by *hls4ml* and Galapagos, but large modifications and additions were required for scaling out to using multiple FPGAs.
- (2) Modifications to *hls4ml* to generate separate IP cores for each layer and the automatic inclusion of a bridge to combine the many parallel streams between the *hls4ml* cores into the single stream supported by the Galapagos framework. The bridges are created at compile time as the width of the bridges are dependent on the number of dimensions in the layer the user wants to deploy.

- (3) Galapagos modifications to add the partitioning layer of the stack. This layer decides how many FPGAs are required and where to place the IP cores generated by our modified `hls4ml`. Our first partitioner is a simple greedy partitioner leaving a lot of room for future research into partitioners that can produce more efficient results. A partitioner is required within our *Algean* stack to enable the seamless push-button flow from front-end to multi-node back-end. Given that the partitioner is a separate abstraction layer, changing the partitioner can be done without requiring any changes in the other layers.
- (4) A framework that allows for incremental development and deployment of an ML application because we can seamlessly integrate hardware and software IP cores. For example, the first step to deploying an ML network is to do it entirely in software targeting a multi-CPU back-end. By simply changing a configuration file, layers of the network can be incrementally switched from running in software to running on FPGAs. Eventually, all layers can be targeted for FPGAs, or the user may choose to run with a heterogeneous implementation where some layers are in software and some are in hardware.
- (5) A large use case of ResNet-50 deployed with two configurations, one with 10 FPGAs and the other with 12 FPGAs. Changing between these implementations is done by changing only a few lines of `hls4ml` code and re-running the flow. This is also a case study that demonstrates the effort required to create a multi-FPGA application on the Galapagos platform.
- (6) A fully integrated hardware and application layer stack that starts with FPGA shells, the layer in the FPGA that abstracts the application logic from the specifics of each FPGA board, the hardware middleware layer that deals with the connectivity between IP cores, a communications layer that implements the desired networking protocol between IP cores instantiated on different CPUs or FPGAs, and an application layer that takes ML networks as input and generates the required IP cores. These carefully defined abstraction layers provide an excellent research platform for experts at each layer to tinker and make each layer better. *Algean* is available as open source to enable further research at all the layers of its stack and can be downloaded at <https://github.com/UofT-HPRC/Algean>.

In Section 2, we describe related work, followed by Section 3, where we provide an overview of `hls4ml` and Galapagos. We describe the implementation and tool flow of *Algean* in Section 4 and present some results in Section 5. Future work is described in Section 6, and, finally, we present conclusions in Section 7.

2 RELATED WORK

We describe *Algean* as a platform that can be used to build heterogeneous ML implementations with a particular focus on using FPGAs and CPUs. As a platform, *Algean* spans the full computing stack from the hardware to the tools used to create the inputs to *Algean*. We have built *Algean* with the goal of making it flexible and modifiable at all levels of the stack to enable research and continued improvement. With this view, we present the related work according to our model of the full ML computing stack. We first describe the model and then present the related work as it fits within our model.

2.1 The ML Computing Stack

The ML computing stack is shown in Figure 1. At the top of the stack, we have a wide range of Applications & Algorithms, many of these applications having strict performance constraints. At the Cluster Deployment & Communication layer of the stack, we have petabytes of data being transferred, and at the Hardware layer, we have many mathematical operations (typically matrix/vector multiplications) implemented on a computing substrate ranging from programmable processors to custom hardware. Each layer of this stack provides its challenges. For example, at the Applications

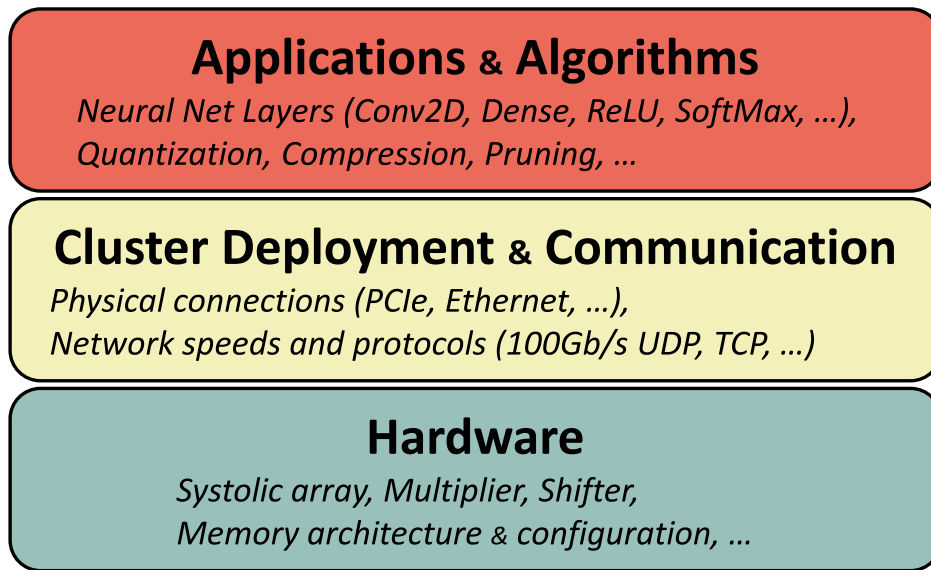


Fig. 1. Abstraction stack for common ML frameworks.

layer, the user has to decide which error rates are acceptable for their given application. At the Communication layer, the user has to decide how they will connect their devices (consisting of computing devices as well as sensors gathering data). Finally, at the low-level Hardware layer, the user may want to make optimizations on bit-level operations for their given application or define different levels of parallelism. There are opportunities for research at all levels of this stack.

2.2 Software ML Frameworks

We define software frameworks as those that mainly target CPUs and GPUs that are programmed via software. Leading software ML frameworks include TensorFlow [9], Torch [10], and Caffe [11]. They provide the users with libraries in various programming languages (e.g., Python, C++) to describe their ML applications. These frameworks then compile the applications into a series of instructions to be executed. Furthermore, they offer an interface to create custom layers that can be compiled into instructions to run on different back-end devices. Finally, they also support connectivity across multiple devices. For example, TensorFlow provides an API [12] to run on distributed clusters, where the communication between different devices (CPUs, GPUs, and TPUs [13]) is either through the CPU network link, through NVLink [14] (i.e., a proprietary link between certain NVIDIA GPUs), or via a direct network link. NVIDIA also provides the NVIDIA Collective Communication Library [15], which implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and networking. This enables scaling GPU computations across large numbers of GPUs available on a network and is supported by several popular deep learning frameworks. Note that in the current implementation of the NVIDIA Collective Communication Library, the GPUs do not have direct connections to the network, unlike what we are able to do with FPGAs. The GPUs connect to the server's network interface through PCIe. We expect that with the acquisition of Mellanox by NVIDIA [16], GPUs will soon also be able to access the network directly and bypass the need for a PCIe transfer.

These frameworks have a high level of customization at the application level. They also allow the users to input custom instructions, but the underlying hardware circuitry is limited to CPU, GPU, and TPU computation and cannot be modified.

2.3 FPGA Overlay Frameworks

Frameworks such as the Xilinx ML Suite and the Intel Deep Learning Accelerator (DLA) provide overlays implemented on FPGAs [17, 18]. An overlay is essentially a programmable engine implemented on the FPGA, which has a limited level of customization. These suites are integrated in existing OpenCL development environments (IDEs), Xilinx SDAccel [19], and Intel OpenCL [20].

The OpenCL IDEs use HLS to improve the accessibility of the design of accelerators for FPGAs over traditional approaches based on VHDL or Verilog, which are time consuming and unfamiliar to most ML experts. In addition, these suites both provide libraries for Direct Memory Access (DMA), buffers, and communication channels, and abstract the underlying hardware, such as device drivers, PCIe link, interconnect, and accelerator placement [21]. These frameworks are similar to the Software ML Frameworks except they add the capability to customize the processor by tuning the overlay architecture on the FPGA.

Nurvitadhi et al. [22] describe a platform that supports multiple PCIe-connected FPGAs in a single server. They build a software stack on top of the Intel OPAE [23] and tightly couple operations on the CPU with operations on the FPGA. Their goal is to implement low-latency neural machine translation and do this by keeping the model in on-chip memories to avoid slower off-chip memory accesses. The ability to leverage multiple FPGAs makes this feasible. This work shows how to leverage the communication layer implemented with PCIe to target multiple FPGA overlays, but their scalability is limited by the number of boards available in one node.

These frameworks allow application developers to seamlessly deploy ML applications on FPGAs thanks to mature software and hardware development environments. However, on the one hand, the high level of abstraction through overlays minimizes the FPGA design time, and on the other hand, it reduces the user control of the generated hardware. With respect to the MS stack we define in Figure 1, these overlay frameworks allow some flexibility in the algorithms and limited flexibility in the hardware. Depending on the hooks available, a user can implement different supported layers to make their own customizations. The hardware flexibility is quite limited as an IP core is already generated. Some frameworks allow the user to modify the IP core through parameters, but this is generally limited in flexibility.

2.4 FPGA ML Core Generators

In this category of work, the focus is at the hardware level of our ML stack where the goal is to make it easier to generate cores for ML computations. These cores must then be integrated into a system that provides the full ML computing stack. Here, we present open source tools¹ that generate ML accelerators as third-party IPs to be integrated into FPGA projects.

CHaiDNN [24] is an ML library for the acceleration of deep neural networks on Xilinx UltraScale MPSoCs. The library provides a subset of ML operators to be synthesized with Vivado HLS and uses 6/8-bit integer arithmetic. Pynq DL [25] provides only a configurable IP for the convolution on Xilinx Zynq SoCs. FINN [26] is a framework for the implementation of binary neural networks that use a dataflow architecture. PipeCNN [27] is an OpenCL-based FPGA accelerator for large-scale CNNs and uses pipelined functional kernels to achieve improved throughput in inference computation. The design is scalable both in performance and hardware resources, and thus can be deployed on a variety of FPGA platforms. HLSLibs [28] is a set of libraries implemented in standard C++ for bit-accurate HLS design. Many of the library operators (e.g., MatMult, SoftMax, Sigmoid) can be easily integrated into the design of ML accelerators. Recently, CNN implementations similar to the design in this article have been produced for low bit precision CNNs [29] and for sparse

¹We report only tools publicly available on GitHub and with a high user rating (Star Metric).

CNNs [30]. These solutions provide more flexibility as the developer, in some cases, can modify the generated cores, as well as integrate additional circuitry around the provided IP cores. However, this design flow is only accessible to those with hardware design knowledge.

With respect to the ML stack, ML core generators provide full flexibility of the hardware and supported algorithms. However, they provide very little when it comes to support for integrating systems at a much larger scale, as it is the user's responsibility to integrate the generated cores into their larger design.

2.5 ML Computing on Multi-FPGA Clusters

In a multi-FPGA cluster, all FPGAs are network connected, and Brainwave [2, 3] built on top of Microsoft's Catapult network-connected FPGA framework [1] is the most successful and well known. Each FPGA contains a customizable overlay. The focus of Brainwave is to minimize latency. Thus, the entire processing only uses on-chip memory and resources, and the neural network is partitioned across multiple FPGAs accordingly. The links between the network-connected FPGAs use Catapult's 40-Gb/s custom Lightweight-Transport-Layer, a lightweight reliability layer on top of a communication protocol similar to that of UDP. When characterizing Brainwave using the stack defined in Figure 1, it can be observed that Brainwave also provides a flexible Application layer as multiple types of neural networks are supported. Brainwave is limited to the Lightweight-Transport-Layer for cluster communication between FPGAs, but this is still an improvement over frameworks that force all accelerator communication through a CPU. Finally, Brainwave provides some flexibility at synthesis time to customize precision, vector size, number of data lanes, and the size of the matrix-vector tile engine. These works allow users to scale a large ML framework across multiple nodes, providing the cluster deployment layer in the ML stack. These works also support a number of layers allowing for the user to customize their algorithm. However due to the scale, there is little hardware flexibility, as the parameterization happens at the node level as opposed to the level of the IP core.

2.6 Where Algean Fits

Although *Algean* can be used with a single FPGA, it best fits the category of Section 2.5, or *ML Computing on Multi-FPGA Clusters*, and has a similar goal as Brainwave. Both platforms use network-connected FPGAs in a peer-to-peer configuration. Brainwave uses a programmable overlay that has some parameterization that can be invoked at the time the overlay is synthesized and can implement many different ML networks depending on the program that is loaded. *Algean* synthesizes custom hardware cores and implements each ML network directly in hardware, so changing an ML network will take much longer than recompiling the program for an overlay. With *Algean*, there is the ability for researchers to experiment at the hardware implementation layer with `hls4ml`, the possibility to experiment with the communication protocols used, and to specify how the computation kernels are deployed. All of the related works provide some layer of the ML stack defined in Figure 1. *Algean* is the only work that can provide support at all of these layers, allowing users to parameterize at the IP core level and at the cluster level, and support many algorithms. This is all made possible by the layered approach used by *Algean* and because everything is available as open source.

3 BACKGROUND

The goal of *Algean* is to provide a scalable platform for implementing ML applications using multiple FPGAs. We use `hls4ml` to build the ML cores and Galapagos as the substrate for deploying an application across multiple FPGAs. In this section, we present the background required to understand `hls4ml` and Galapagos before describing how they are integrated into the platform we call *Algean*.

3.1 Hls4ml

We need a way to implement hardware ML-inference cores that can take specifications from common ML frameworks. We choose `hls4ml` [4] because it can translate the specification of ML models from common frameworks such as Keras [31], PyTorch [32], ONNX-formatted models [33], and the quantized version of KERAS, QKERAS [34], into **Register-Transfer Level (RTL)** implementations for FPGAs using HLS tools [35]. In our experiments, we use Vivado HLS [36] as the `hls4ml` back-end though the flow can be extended to other HLS tools. `Hls4ml` currently has support for Vivado HLS, Quartus HLS, and Mentor Catapult HLS [37].

At the start of the *AIgean* development, `hls4ml` was a tool that was only targeted to implement ML applications that fit on a single FPGA. In this section, we describe the baseline capabilities of `hls4ml`, and in Section 4.2, we describe the changes we made to integrate `hls4ml` into *AIgean*. A more detailed description of the changes to `hls4ml` is found in Appendix A.1.

An ML designer prepares a neural network for a specific task, such as image classification, in Keras or PyTorch. After an iterative training phase that ends when the target accuracy/error goals are met, the ML designer releases a final model to be deployed for inference. The model is usually described as two files in standard formats: a JSON file for the model architecture, and an HDF5 file for the model weights and biases. These are the inputs for `hls4ml`. At this point, a hardware designer can fine-tune the `hls4ml` project and *push-button* translate it into a complete Vivado HLS specification (C++ and TCL files) to be synthesized and implemented for a target FPGA.

The hardware designer faces the challenge of creating an optimal FPGA implementation from the given ML model. The `hls4ml` framework exposes a crafted set of configuration parameters (HLS knobs) to balance the FPGA resource usage and the latency and throughput goals. The design of ML-inference accelerators using HLS is simplified with `hls4ml` by hiding the large variety of HLS knobs and providing carefully optimized layer implementations for HLS.

The conversion from deep learning model to HLS-based software is done by constructing a custom intermediate network representation that is amenable to low-latency design. From this intermediate representation, HLS code is generated with design guidelines specified in a configuration file. Optimized HLS implementations of neural network layers are generated, with the optimization dependent on specified configuration parameters. The code is thoroughly modular, and most optimizations can be tuned after the HLS code generation. `Hls4ml` has been used to construct MLP networks, CNNs, Graph neural networks, RNNs, and BDTs [38–41].

The `hls4ml` design flow explicitly focuses on batch-1 processing. Larger batch processing is not considered. The design flow is similar to the FINN architecture [26, 42] in that model-specific layers are implemented. A critical element of the design of `hls4ml` is to allow for very low latency implementation of ML algorithms with a low initiation interval.² As a consequence, `hls4ml` generates an HLS firmware implementation of the neural network on a layer-by-layer basis. Each layer corresponds to a different firmware block, and therefore individual layers can be run concurrently. This design paradigm differs from most other FPGA deep learning implementations, such as Xilinx ML Suite, where the same firmware blocks are repeatedly used to perform the inference computations over many layers of a neural network. Separation of the layers into separate firmware blocks is particularly tractable for multiprocessor use since layers can easily be split into separate IP blocks without any modifications in the algorithm design or changes in resource usage. We leveraged this capability for *AIgean*.

The trade-off among latency, initiation interval, and resource usage determines the parallelization of the accelerator logic (and vice versa). In `hls4ml`, this trade-off is configured with a single

²In HLS, the initiation interval specifies the number of clock cycles between the introduction of new inputs in a pipeline.

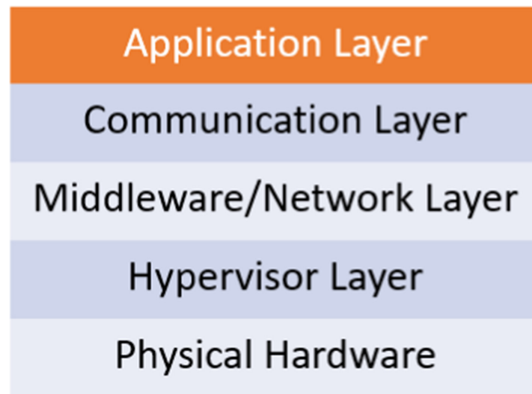


Fig. 2. The *Algean* stack. It includes an Application layer on top of the previously developed Galapagos stack [6].

configuration parameter—the *reuse factor*. The choice of a value for the reuse factor affects the initiation interval of the RTL pipelines and the number of critical resources (e.g., DSPs) in each layer of the neural network.

Within the *hls4ml* implementation, the reuse factor dictates the number of times a single DSP is reused within a single matrix multiplication. This factor translates directly to the number of resources that each layer uses. In particular, both the DSP usage and the number of BRAM partitions will scale with the reuse. By scaling the reuse factor value, the designer can explore various implementations. A reuse factor of 1 generates a completely parallel implementation (lowest latency); a reuse factor of R , generates an implementation with $1/R$ fewer DSPs (lower resource usage) and BRAM partitions. Designers may choose a larger value for reuse factor in the case of limited resources and a smaller value when they can afford higher parallelism.

For the development of *Algean*, a number of improvements were made within *hls4ml*. These developments include:

- Streaming dataflow between the layers (with Galapagos)
- Optimized large layers for the Dense/Linear Layer, CNN Layer, Pooling Layer, Split Layer, and Merge Layer
- Modified Reuse Factor for CNN throughput
- Weight reconfiguration through the use of external block RAM ports

Finally, the generated ML accelerators have interfaces that are system agnostic. In Section 4, we illustrate our extension to the Galapagos flow that enables a designer to rapidly prototype ML accelerators and deploy them in a Galapagos system with minimal effort.

3.2 *Algean* Stack

Algean is a development stack for deploying ML applications across multi-FPGA and CPU clusters. This logically can be seen as a superset of the Galapagos development stack with a specific application layer. Galapagos is a hardware stack that provides customization at different levels of abstraction [6]. The main goal of Galapagos is to abstract the low-level hardware plumbing required to deploy an application across multiple FPGAs while also providing the ability to port applications across multiple FPGA platforms (i.e., platforms built using different FPGA cards with different networking infrastructures). We know of no other platform that can take as input just the computation kernels and a logical description of the connections between the kernels and then generate all of the FPGA bitstreams with all of the network connectivity included. Without

Galapagos, an application developer with a multi-FPGA cluster would need to be an expert in hardware design. In addition to building the computation kernels, the developer would need to incorporate into their design the interfaces to the on-board memory, the network interfaces, the network protocol hardware (most likely hardware UDP or TCP/IP cores), and configure Ethernet MAC addresses, IP addresses, the routing information for moving data between kernels, as well as build all of the packet formatting and protocol translation between the computation kernels. The FPGA vendor platforms for OpenCL [19, 20] are usable by non-hardware application developers because they abstract away these details. Galapagos does the equivalent abstraction, but for a multi-FPGA cluster environment. By building on Galapagos for *AIgean*, we can leverage the multi-FPGA abstraction that is provided by Galapagos, and can focus on the integration of `hls4ml` and not worry about the low-level hardware plumbing required.

The structure of Galapagos is analogous to a traditional software or networking stack, with each layer of the stack providing an API for the layer above. The lower the layer in the stack, the closer it is to the physical hardware. Figure 2 shows the *AIgean* stack.

Physical hardware and connectivity. This layer represents the physical hardware that runs applications, and for this work we focus on the FPGAs. Aside from implementing the computations in FPGA logic, we can also implement different forms of connectivity. In Galapagos, we can use PCIe, 10G SFP+ Ethernet, 100G QSFP28 Ethernet, and L1 circuit switching. For Ethernet, we can select TCP/IP, UDP, and raw L2 Ethernet. Once configured in this lower level, typical software and ML practitioners can work at a higher level of abstraction.

Hypervisor. The hypervisor³ abstracts away the I/O interfaces of a single FPGA so that the hardware applications only needs to connect to a standardized interface, and they can then be implemented on any FPGA that has the same hypervisor. This is the key requirement that enables applications to be portable across multiple hardware platforms enabled with Galapagos. In the same way, the hypervisor in the software world provides an abstraction of the hardware and some level of services, typically I/O and memory.

Middleware. This layer connects the different devices within the Galapagos cluster and sets the off-chip network communication protocols between them. Within Galapagos, computation kernels can address each other and are agnostic of their placement.

Communication layer. The communication layer provides the APIs with the ability to send packets using the connections laid out by the middleware. Galapagos transmits packets using the AXI-Stream protocol [43]. All of the kernels within the cluster can reach any other kernel via AXI-Stream. Since the middleware layer provides the network address translation functionalities to convert AXI-Stream into off-chip network packets using the desired network communication, the network details and locations of kernels are abstracted away from the user. In software, this is the role of network-socket libraries or other network and communication protocols such as the Message Passing Interface (MPI) [44].

Application layer. For *AIgean*, the application layer is the ML layer provided by `hls4ml` and the tools used to generate the inputs to `hls4ml`.

4 IMPLEMENTATION AND TOOL FLOW

The implementation of *AIgean* requires a significant effort to create a seamless integration of `hls4ml`, which builds hardware cores for ML, and Galapagos, which builds multi-FPGA applications. We had to make substantial modifications to `hls4ml` so that it generated streaming cores, and we needed to build the adaptation layer that can take output from `hls4ml` and convert it into the format for input to Galapagos. Furthermore, as part of this work, we implemented a number of

³In Microsoft terminology, this layer is called the *shell* [1].

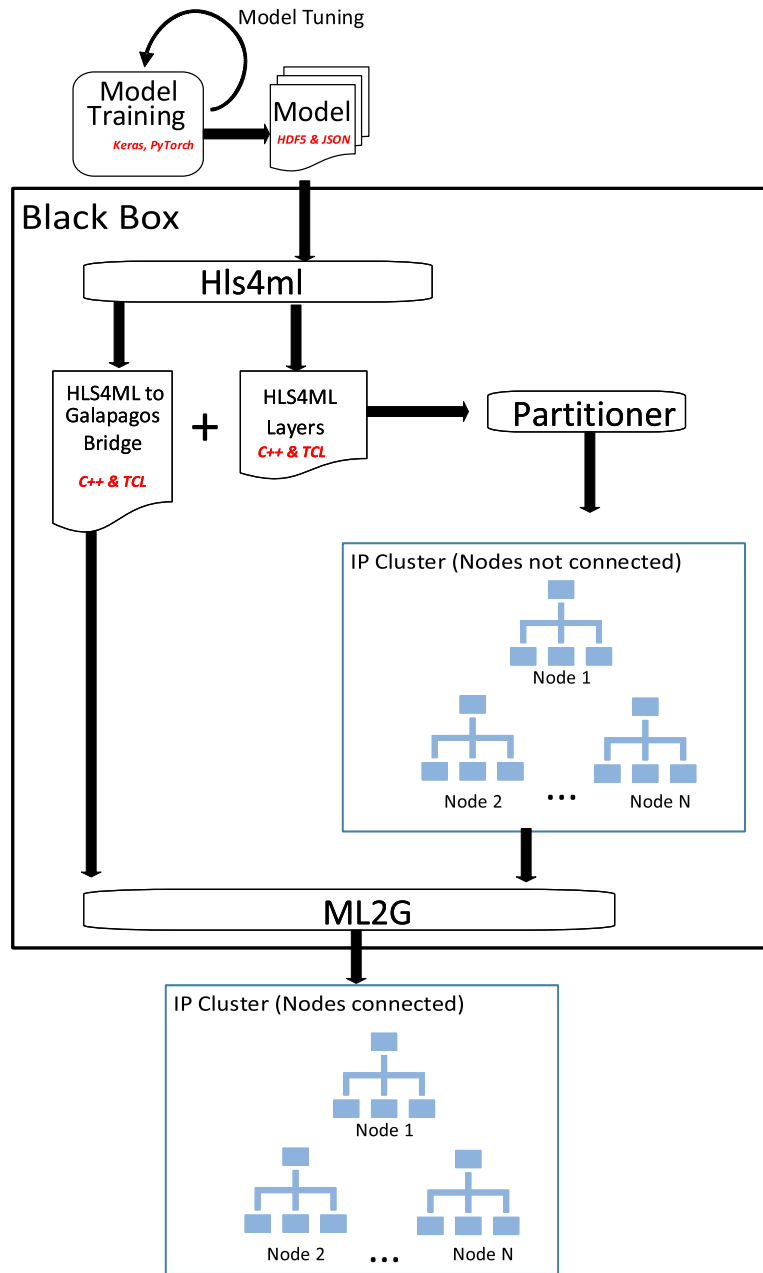


Fig. 3. The *Algean* flow. The components in the black box are abstracted away for the average users.

improvements to both `hls4ml` and Galapagos to further optimize the functionality of both systems. In this section, we highlight the details of our *Algean* tool flow along with the specific changes to `hls4ml` and Galapagos required to integrate them into *Algean*.

4.1 Tool Flow

The stages of the *Algean* tool flow are visually presented in Figure 3. The *Algean* automated flow provides a black box that takes an ML model to a CPU/FPGA cluster. In the following sections, we highlight the inner components of the black box because they can be modified by domain experts working within each part of the tool flow to explore a large design space relevant to their interests.

Each of these stages corresponds to layers of the abstraction stack for common ML frameworks we described in Figure 1. The stages are described as follows.

4.1.1 Implementation-Agnostic Model Tooling. This stage of the flow corresponds to the top layer of the ML stack, *Applications & Algorithms*. This layer of the stack is for the data scientist and ML experts, where they can tune their network for a given accuracy independent of the implementation and performance. For a given application, the users will decide on the algorithms they wish to use for their ML implementation. Using their application-specific test data, they can determine a suitable accuracy for a given neural network, independent of the implementation being done in hardware or software. Once a model is trained with the appropriate precision and performance requirements, *AIgean* will take this model and perform a full conversion to a distributed deep learning inference engine.

4.1.2 HLS Layer Implementation. At this stage, the input from the previous stage is transformed by `hls4ml` into RTL synthesizable C++ code that can be executed as software to verify functional behavior. It is at the discretion of the user to select the granularity of the IP blocks generated, where the finest granularity the user can select is at the boundary of the neural net layers. A coarse granularity can limit flexibility in partitioning networks across multiple devices and may create IP blocks that are too large, but it is simpler. The user will also select the reuse factor, where a lower reuse factor unrolls the implementation of the IP blocks to use more DSP and BRAMs. Once the user tunes these cores for their resources, the functional correctness of the individual IP blocks can be verified by running the code in software. Users who wish to tinker with implementations of individual layers can work at this level.

In Section 5, we explore two different IP core implementations of ResNet-50 as an example. This part of the flow first generates a directory structure with many sub-directories, and each sub-directory is for an individual IP core (one IP core per layer), containing the HLS source code and build files. The top-level directory also has a build file, and then the user can then do a parallel build across all sub-directories of all the IP cores. For our ResNet-50 case study in Section 5.5, the generation of the directory structure and HLS source files can take on the order of minutes, whereas the HLS can take on the order of a few hours. The HLS also does an out-of-context place and route so that we can get a more accurate resource utilization that is then used in the partitioner.

4.1.3 Layer Partitioning. At this stage, the user begins with IP cores described in C++ that were generated from `hls4ml`. Each IP core is input to the HLS tool to generate RTL, which is then placed and routed out of context—that is, as a stand-alone circuit, from which an estimate of resource usage is generated for each IP core. Using these estimates, the user can allocate one or more IP cores to FPGAs. We have implemented a simple partitioning tool within Galapagos that can automate the placement of IP cores on FPGAs. Galapagos can take IP cores labeled as “floating” IPs within the cluster and place them on any available FPGA. Our implementation of this is a simple greedy algorithm by using the resource estimation of the IP cores and available resources on the respective FPGAs. Once the IP cores are partitioned, our tool analyzes the graph to investigate the edges between FPGA node boundaries. Based on the boundaries, our tool places an `hls4ml`-to-Galapagos bridge that is custom made to fit the dimensions on the output and input FPGAs. This is needed as an `hls4ml` kernel has a parallel stream for each dimension tensor, whereas a Galapagos kernel has a single stream.

This is our first implementation of this partitioner and leaves much more room for future work focused on the partitioning. Given that the partitioner is a separate abstraction layer, changing the partitioner can be done without requiring any changes in the other layers. The partitioner is also implemented within Galapagos as this is independent of the ML use case and can be applied to other domain spaces.

Once we get a partitioning from our Galapagos partitioner, our *AIgean*-specific bridging cores then provide bridging based on the kernels that occur at the edges of each FPGA. This is done

separately by our ML to Galapagos layer (ML2G) as the bridges required at each edge is specific to the partitioning as we have a different bridge depending on the width of the ML kernel on the edge. The `hls4ml` bridge is explained in detail in Section 4.2. Once the bridges have been appended to the partitioned cluster, we can generate bitstreams or model the partitioned cluster in software. The output of this layer is the Galapagos configuration files, describing all the kernels and their connectivity. In our case study, in ResNet-50 this can take on the order of a few seconds.

4.1.4 Software Cluster Implementation. This stage is optional but highly recommended for heterogeneous development. The underlying Galapagos framework can wrap HLS synthesizable C++ code with software libraries to enable network socket communication. The underlying Galapagos software library [45] translates Galapagos stream packets into network packets in a user-specified off-chip network protocol (e.g., UDP, TCP). We describe the underlying Galapagos framework in Appendix A.3. Galapagos can be seen as using the standard AXI-streaming protocol, typically used for streaming kernels within a single Xilinx FPGA. There is also basic routing with a destination field within AXI-stream. Galapagos can take AXI-stream and encapsulate packets with higher-level protocols to get the convenience of a single device AXI-stream but over multiple nodes. The user at this stage can create a homogeneous cluster partitioned across multiple software nodes (with each software node taking the place of a hardware node), recreating the network topology the user wishes to have for their heterogeneous deployment. All the network connections, binary generation, and deployment are automated with the underlying Galapagos framework.

4.1.5 Heterogeneous Cluster Implementation. Once the neural network is partitioned across multiple software nodes and shown to be working correctly, the user can then migrate parts of their software deployment into hardware nodes. This is done by simply changing a parameter in one of the Galapagos files to indicate that an IP core should be implemented in hardware rather than run in software. Since Galapagos ensures that both software and hardware nodes use the same protocol, the migration is seamless. The migration of cores from software to hardware can be done in an iterative process as the generation of hardware bitstreams can be a time-consuming process. The outputs of this stage are the final bitstreams. For each FPGA, the IP cores are put together and synthesized to a bitstream. In our ResNet-50 case study, this took on the order of a couple of hours.

4.2 Hls4ml Modifications

The full details of the modifications implemented in `hls4ml` are specified in Appendix A.1. In particular, we modified `hls4ml` to produce HLS cores with streaming interfaces so that they can fit with the streaming model of Galapagos. As part of these modifications, an auxiliary channel is added between layers to allow for network inference to reset in the middle of an inference. This additional option is useful for multi-FPGA implementations where data streams are vulnerable and can be interrupted. Additionally, `hls4ml` was extended to have the option of large CNN layers with millions of weights. The previous CNN implementation was intended for low-latency use and could not support as many weights. The core of `hls4ml`, including the fully connected layers and activation functions, remain the same and are embedded in the streaming implementation. As a consequence, the full functionality of `hls4ml` is preserved in this streaming implementation, allowing for a broad range of models to be implemented.

Further optimizations are applied for ResNet-50, including the fusing of batchnorm layers with the convolutional layers and the compression of 8-bit weights into single 16-bit weights so that DSP multiplier units can be used and the total number of needed multiplications is halved. Finally, an additional configuration parameter is added to the autogeneration that allows for approximate tuning of the reuse factor to obtain the desired CNN throughput. With this new option, the tuning

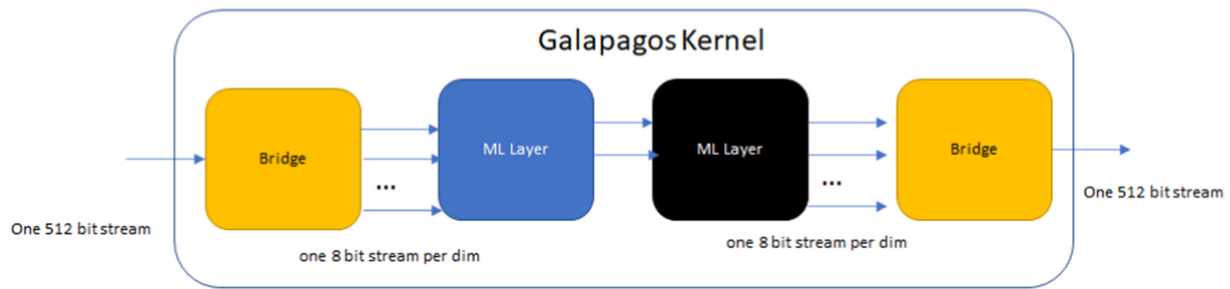


Fig. 4. An hls4ml kernel can be surrounded by custom bridges to make it possible for off-chip communication.

factor for the network is defined by the desired throughput in operational clocks and the reuse factor is adjusted so that every layer achieves the desired throughput. As a result of the throughput tuning, the reuse factor will be adjusted to ensure the inter-layer latency is roughly the same. A balanced throughput avoids significant bottlenecks between the layers. The full details of the throughput tuning is described in Appendix A.1.

The hls4ml streams have no side channels. There are only the data payload (e.g., fixed-point data) and ready/valid signals between kernels. This kind of stream suffices for point-to-point connections within one FPGA. However, off-chip communication between cores requires additional routing information. Galapagos IP cores use HLS streams with side channels to provide routing information (i.e., destination). The destination field that is used in Galapagos by default is 16 bits, but this is configurable depending on the number of IP cores we have in our cluster. We designed bridges to transform hls4ml streams to Galapagos streams by adding the additional routing information and packing the data in larger-bit-width Galapagos streams. These bridges convert a single tensor consisting of many parallel AXI streams, with one stream per dimension, into a single large bit width stream for off-chip communication. Since the bridge's size is dependent on the hls4ml IP core (the bridges input size depends on the number of streams), this also needs to be auto-generated. A visual representation of this can be seen in Figure 4.

Furthermore, with the processing of streams, it allows us to send flits of data corresponding to different images within the same packet, allowing for a more efficient use of bandwidth. These streams are also configurable by allowing the user to configure how many AXIS stream flits⁴ to pack within one network packet. This solution can be significant for FPGA-CPU links where it is crucial to amortize the cost of network communication on the CPU, due to the overhead added by the Linux network stack.

4.3 Galapagos Modifications

To explore the design space of large ML networks (like ResNet-50) across multiple FPGAs, we developed an automated partitioner to work with the rest of the Galapagos framework. When we turned to ResNet-50 to implement a very large network, it quickly became clear that we needed an automated means for partitioning a large application to make the best use of the resources. Our first partitioner is described in Section 4.1.3 and is not specific for just hls4ml kernels but can be used for any streaming kernels that require placement.

The original Galapagos framework supported 10G TCP and L2 Ethernet for off-chip communication. We designed a bridge to provide the option for 100G UDP cores to increase the performance

⁴A flit is the amount of data transferred in one clock cycle in an AXIS stream.

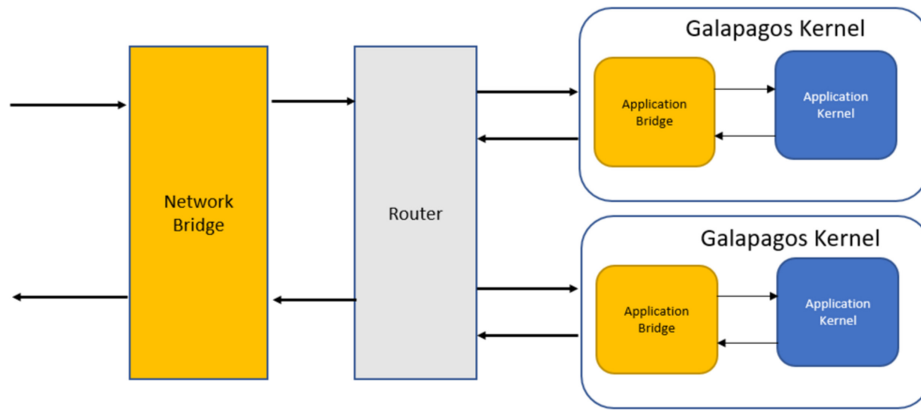


Fig. 5. The IP cores providing the bridging in Galapagos.

of our network links and to reduce the probability of the network communication being the bottleneck. This enhances the capability of any application using Galapagos, not just *Algean*.

To support 100G, we also improved the portability within Galapagos to support multiple bit-widths of data. The prior bridging within Galapagos assumes all kernels communicate over AXI-stream with a destination side channel. However, due to the additional required bridge needed to allow hls4ml kernels to communicate over AXI-stream with a destination side channel, we needed to modify Galapagos to include support for the insertion of application-specific bridging. This is shown in Figure 5. For more details on the bridging provided within Galapagos, please refer to Appendix A.3. In this case, one of such application-layer bridges is the hls4ml bridge described in Section 4.2. An application-layer bridge transforms an application-layer protocol into Galapagos packets. There is a configuration control path for the user to adjust the properties of the bridge. For the network bridge, it allows users to adjust the routing between kernels by allowing the user to adjust the mapping of destinations to IP addresses.

A major goal of *Algean* and Galapagos is the ease of development. Galapagos offers functional portability of cores between hardware and software by implementing software libraries to model the hardware routers and bridges shown in Figure 5. The software library (i.e., libGalapagos) is described in the work of Tarafdar and Chow [45]. Furthermore, Galapagos allows fast simulations of the cluster thanks to the combination of the HLS IP cores (in C++) and libGalapagos for the connections between cores. When we designed the additional cores and bridges for *Algean* (e.g., 100G UDP core), we also implemented the libGalapagos equivalent of these cores to maintain functional portability. On top of prototyping the entire cluster in software, we have also added the ability to simulate the entire cluster in RTL. We designed an RTL model of a network switch that is configurable and can simulate the latency between network links. This capability has been invaluable during the development of *Algean* as a platform but would not be required during the normal use of *Algean*. The combined efforts of both these frameworks result in a fully configurable design space exploration tool of multi-node heterogeneous ML clusters.

5 RESULTS

This section presents the outcomes of our efforts to build *Algean*. It is important to emphasize that the initial goal of this work is to build a platform to enable the development of multi-FPGA ML applications. The performance results that we report here demonstrate that *Algean* is working, and even with our first example applications, the results are reasonable. For this work, we claim success if we are able to easily build ML applications and map them to multiple network-connected

FPGAs. There is much room to tune for application performance given a working *AIgean*, and we will now be using *AIgean* to explore opportunities for tuning and to build other kinds of networks.

We first describe the hardware testbed used for our experiments and discuss the ease of use of *AIgean* in its current state with a case study we did for our own experiments. Then we present more quantitative results by addressing the physical limits of the communication links, and finally we present the current performance results of our first applications starting with a small network to illustrate the latency benefits of using network-connected FPGAs and then for ResNet-50 as a test to see whether we can implement a very large network.

5.1 Hardware Testbed and Tools

Our hardware testbed comprises Supermicro servers with Intel Xeon E5-2650V4 CPUs and 64 GB of memory. The FPGA boards we have available are Alpha Data 8K5s with a Xilinx KU115-2 FPGA, Fidus Sidewinders with Xilinx ZU19EG FPGAs, and Xilinx Alveo U200 and U250 cards with XCU200 and XCU250 FPGAs, respectively. We have 16 Sidewinders mounted in a 16-slot PCIe chassis, and the other boards are mounted in PCIe slots of our servers. For the network interconnect, we have two Dell S4048-ON 10G and two Z9100-ON 100G switches. The servers are connected to a 10G switch and the FPGA boards are mostly connected to 100G switches. For the *AIgean* tests reported here, we used Vivado 2019.1 and Sidewinder FPGA boards connected to 100G switches.

The SDAccel platform we used was on an Amazon f1.2xlarge instance using SDAccel v2018.2. For those tests, `h1s4m1` was used to generate the cores, and they were invoked as OpenCL kernels using SDAccel. The GPU tests used an Nvidia 1080Ti.

5.2 Ease of Use Case Study

While working on this article, we have gone through several iterations of different layers of the stack. One iteration involved optimizing our HLS library to use DSPs more efficiently, with the same functionality. We describe the results in Section 5.5, but we would like to discuss the steps required to change our cluster implementation between the two IP core implementations. This change was done in the `h1s4m1` level, particularly the domain of a hardware expert looking to optimize the hardware implementation of a particular IP core. Following the change, we ran `h1s4m1` generating a directory structure and a makefile, with a subdirectory per IP core. At this point, we can build at the top-level makefile by typing “make,” and this will rebuild all IP cores that have changed their implementation. Then we point our IP core directory to the rest of the *AIgean* flow and type “make.” This will then partition the IP cores, add the bridges, and generate the bitstreams. This case study shows that an expert in the IP core generation only has to focus on their layer of abstraction and then rebuild the entire cluster by typing “make” twice.

5.3 Communication Protocol

In this section, we present the latency and throughput measurements for different link configurations. This is to provide some understanding of the penalties for communication over network links. For communication between nodes, we can use a 100G UDP core [46] or a 10G UDP core [47] on the FPGA. These are interchangeable within our framework by the user. The 100G core uses a 512-bit interface as compared to the 64-bit interface for the 10G core. The CPU NIC we use is a 10G SFP NIC [48], even when communicating to a 100G FPGA. The specific FPGA board we are using is the Fidus Sidewinder with an MPSoC FPGA [49]. For latency measurements, we send a single flit of data (8 bytes) using the four different types of links listed in Table 1. The results in Table 1 involving software are shown with the 100G UDP core on the FPGA, but similar results are observed when using the 10G UDP core. From hardware to software, we observe the FPGA outputting at 100 Gb/s, but we experience packet drop in the software when doing the throughput

Table 1. Round-Trip Latencies and Throughputs of Three Different Links

Link	Latency	Throughput
Software to Hardware	0.029 ms	0.244 Gb/s
Hardware to Hardware QSFP	0.00017 ms	100 Gb/s
Hardware to Hardware SFP	0.0003 ms	10 Gb/s
Hardware to Software	0.0203 ms	N/A

Table 2. Autoencoder Resources Compared to Target FPGA Resources

	Initiation Interval	DSPs	BRAMs	LUTs	Flip-Flops
Autoencoder Resources	552	768	35.9 MB	1.02M	335K
F1 Resources Per FPGA	–	9.2K	72.6 MB	1.29M	2.59M
Sidewinder Resources Per FPGA	–	1.9K	34.6 MB	522K	1.04M

measurement. Observe that the links involving software are limited by the CPU network stack and library implementation, whereas the FPGA-to-FPGA links can transfer at the full network bandwidth. Note that the results in Table 1 are for the raw throughput, including the protocol headers, which is why it is possible to achieve the full link bandwidth when using the FPGAs.

5.4 Autoencoder

Here we describe our first small multi-FPGA network implemented with *Algean*. We consider an example network with applications for high-energy physics. Specifically, our network is an *autoencoder* designed to detect anomalous events, potentially from new physics. An autoencoder is an unsupervised learning technique that leverages a neural network where a bottleneck in the shape of the network forces a compressed representation of the original input. Details about the model and use cases can be found in Appendix A.2.

This network is a very interesting size for our studies, as it can be implemented on a single FPGA, but this requires a high degree of resource reuse that necessarily increases the inference latency. When splitting the network across multiple FPGAs, we can adjust the throughput and latency of the network by changing the reuse factor and compiling the network across multiple FPGAs. The network split across multiple FPGAs will have a higher throughput but incurs some latency from the transfer of the intermediate results.

The resources for the autoencoder network are shown in Table 2 along with the resources available on the FPGAs we used. To test this autoencoder, we considered two separate implementations of the network: an implementation using an AWS F1-instance (VU9P FPGA) using SDAccel, and a second implementation using *Algean* on three Sidewinder (ZU19EG FPGA) boards. What is notable is that the single FPGA implementation would not be able to fit on a single Sidewinder board, and it would have to be spread over multiple FPGAs for the chosen reuse factor. The single FPGA implementation also requires more than one super logic region, and as a consequence has difficulty meeting timing when compiled on the F1 instance with SDAccel.

Table 3 highlights the results from implementing the autoencoder on various devices as well as on a single FPGA using SDAccel and three FPGAs using *Algean*.

Our 1-FPGA autoencoder is clocked at 125 MHz at a low reuse factor when using SDAccel. Limitations in our version of SDAccel, as well as the resources required for the FPGA, prevented us from using a higher clock speed. For the 3-FPGA version, we used *Algean* and were able to achieve 200 MHz for two of the FPGAs and 190 MHz for the third one. We did not try to improve it, so we

Table 3. Round-Trip Latency of a Single Batch Inference

Device	Latency (ms)
Xeon E5-2650V4	3.3
Nvidia 1080Ti	2.5
1 FPGA Implemented in SDAccel (125 MHz)	0.24
3 FPGAs Implemented in <i>Algean</i> (190 MHz)	0.08
3 FPGAs Implemented in <i>Algean</i> (125 MHz)	0.12

will use 190 MHz since that is the limitation. To make a fair comparison to the 1-FPGA implementation, we scale the *Algean* latency by the ratio of clock speeds and get $0.08 \times 190/125 = 0.12\text{ms}$, which is still $0.24/0.12 = 2$ times better than the latency using SDAccel. This shows that there is still a significant architectural advantage to using multiple FPGAs and is not unexpected because more resources are available. The performance increase with three FPGAs can be attributed to (a) the use of networking to directly communicate with the FPGA, yielding low latency, and (b) less demanding resources per FPGA since only one-third of the model is implemented on each FPGA.

The implementations of this model on both a single FPGA and the full three FPGAs have an initiation interval of 552 clocks and require roughly the same resources (the reuse factor is the same). In other words, the three FPGAs are capable of processing a new image every $2.76 \mu\text{s}$ (362 KHz). Such a throughput approaches the demands needed for real-time processing of anomalies at the LHC. Although the single FPGA implementation with SDAccel has a potential throughput that is half that of the 3-FPGA implementation, achieving this throughput would require efficiently buffering the inputs and outputs by sending larger batches of calls on and off the FPGA through the DDR and PCIe transfers. As a consequence, the individual (batch-1) latency would be significantly degraded for the final throughput to approach half that of the 3-FPGA implementation.

5.5 ResNet-50

To test *Algean* on a much larger network, we have developed a multi-FPGA implementation of ResNet-50 [50]. The flexibility provided by *Algean* allows us to target a high throughput implementation whereby we unroll the multiplication in each CNN layer at a rate corresponding to the number of pixels that are being used in each respective CNN layer. This allows for the design of ResNet-50 that can be balanced across the different CNN layers to have a uniform throughput.

Most of ResNet-50's architecture can be broken down into many sub-blocks consisting of a Split, two to three convolutions followed by a Relu, and an addition operator as shown in Figure 6. The dashed boxes represent the IP block granularity that we have used within our implementations.

We have two implementations of ResNet-50: the first requires 12 Sidewinder boards using int-8 precision (ranging from 80% to about 90% of the resources used on each FPGA); the second is more DSP efficient and requires 10 Sidewinder boards using int-8 precision as well. We have one FPGA available to use as a 100G data generator that can feed inputs at line rate to the FPGAs. For the 12-FPGA configuration, we tested in a piece-wise fashion.⁵ We have tested the traffic generator and the first 10 of 12 FPGAs followed by testing the traffic generator and the remaining 2 FPGAs. We have verified that the full 10-FPGA configuration and the piece-wise 12-FPGA configuration can run at 660 images per second.

Table 4 summarizes the throughput and latency results of our full 12-FPGA implementation of ResNet-50. When the source data is coming from the CPU, we observe that the maximum

⁵We could not get access to enough boards.

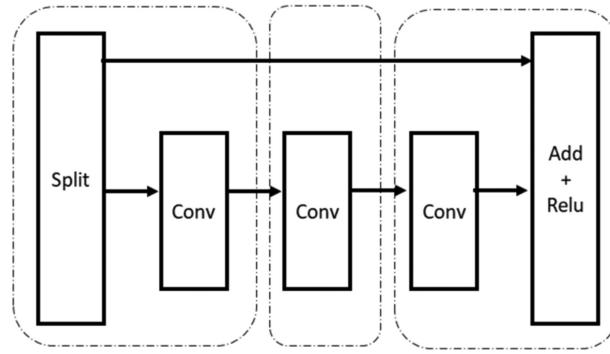


Fig. 6. Sub-blocks found throughout ResNet-50 and our IP cores.

Table 4. Performance of Different Layers and Implementations at Batch 1

Implementation	Throughput	Latency
Algean Using CPU/FPGA network	400 images/s	7 ms
Algean Using FPGA Data Generator	660 images/s	1.9 ms
Microsoft Brainwave Batch 1	559 images/s [51]	10 ms
Nvidia V100 GPU Mixed Precision Batch 1	250 images/s [51]	5.9 ms

throughput is only 400 images per second with a latency of 7 ms due to the bandwidth limitation between the CPU and the FPGA (5-ms latency between the CPU and the FPGA). To demonstrate the full performance achievable with the FPGAs, we use the FPGA data generator and observe a throughput of 660 images per second with a latency of about 1.9 ms. The latency is determined through a simulation of the full ResNet-50 network where each layer is separately run in parallel. The network delay between each FPGA is estimated from Table 1 using the QSFP. For 10 hops, the total network delay would be 0.0017 ms, which is insignificant compared to the computation latency. The next row gives the values for Microsoft’s Brainwave [51]. For the latency of Brainwave, we quote the end-to-end latency determined from sending an image to a Brainwave server and then receiving the result for a CPU within the same computing cluster. The final row shows the performance for an Nvidia V100 GPU using the mixed precision implementation of ResNet-50 applied for batch 1. The latency and throughput quoted is obtained through the use of the Triton inference server with a client on the same machine. As a consequence, the latency numbers include the PCIe transfer time in addition to the network inference. Equivalent numbers quoted by Nvidia yield a batch-2 latency of 1 ms with a throughput of 2,000 images per second for the same model [52]; batch 1 latency is not quoted.

Table 5 summarizes the resources used for our 12-FPGA implementation. Note that this was partitioned with our greedy partitioning scheme that uses a heuristic of 80% utilization before allocating the next FPGA. The 10-FPGA configuration is very similar in terms of resources but with half the DSPs. Some other noteworthy details are that a number of the layers early in the network are smaller, and we can see that the FPGAs are DSP limited as compared to the larger layers later in the network being logic limited. The highest resource utilized for each FPGA is shown in bold, representing the limiting factor of each FPGA (with exception of the last FPGA that

Table 5. Resource Utilization Percentage of Each FPGA and Total Resources Available Per FPGA

FPGA Number	FLip-Flops (%)	LUTs (%)	DSPs (%)	BRAM (%)
0	31.8	40.1	71.5	1.73
1	26.7	35.1	74.8	11.9
2	11.12	12.06	74.8	0.68
3	49.3	66.6	65.0	6.90
4	38.3	50.9	71.5	2.00
5	20.5	23.2	78.5	14.0
6	54.0	72.6	65.0	7.08
7	57.3	75.9	65.0	10.1
8	60.1	78.2	68.3	13.8
9	58.9	76.5	52.0	7.26
10	44.5	57.4	58.5	5.12
11	30.9	39.9	38.7	8.72
Total Absolute Resources Used Across All FPGAs	5.05 M	3.28 M	15.4 K	31.5 MB
Total Resources Available Per FPGA	1.04 M	522 K	1.9 K	34.6 MB

is not fully used.). For perspective, the total resources available on an individual FPGA are shown at the bottom of Table 5. This FPGA is approximately equivalent to a single SLR of the VU9P FPGA in the Amazon F1 instance (each VU9P having three SLRs) [53]. For further perspective, we can also compare this to the Xilinx Alveo U250 [54]. Our current utilization is DSP limited, and we could fit our entire ResNet-50 implementation on two Alveo U250 boards, where the U250 board has 12.2K DSPs.

Last, we would like to contrast this implementation with previous implementations of ResNet-50. The design flow of *Algean* differs from previous 8-bit implementations of ResNet-50 in that no overlay is used, and each layer is implemented separately. In this scenario, it is possible to continuously stream images through the implementation without having to wait for an image to be complete. With the overlay architecture, the images are streamed through each layer to a buffer and then subsequent layers are loaded and the next layer is streamed. As a consequence, a scheme is needed for buffering of each input. Additionally, some time is needed to switch between layers. With the *Algean* design flow, the whole network exists on the FPGA fabric, and so images can be continuously pumped through. This leads to a more efficient use of multiplier resources, at a cost of additional resources to route individual layers together. Since images are continuously pumped through, we achieve batch-1 streaming. Additionally, since we are continuously pumping images through, the amount of buffering between the layers is limited to just the partial image that is needed for matrix multiplications of the CNN applied to nearby pixels.

To understand the efficient use of resources, we compute the total number of multiplication operations needed for a perfectly efficient FPGA clocked at 200 MHz. With our implementation of ResNet-50, we find a total of 4B multiplications, which if we divide by 3×10^5 clocks to achieve a 1.5-ms latency at 200 MHz yields a total of 13,500 multiplications per cycle. Our current implementation uses 15,419 DSPs, which is slightly more due to the fact that many of the individual layers are tuned to a latency that is actually below 1.5 ms. The number of DSPs can be reduced through two

means: first, through the sharing of DSPs, which is only partially implemented here, and second, through the use of a faster clock frequency. The sharing of DSPs would lead to roughly a factor of 2 reduction in DSPs. A faster clock frequency would yield a lower latency for the same number of DSPs. Since each multiplier unit is mapped directly to a specific multiplication within the network, the only way to inefficiently use the DSP resources results from the case where an allowed reuse parameter for a specific latency is not near the desired throughput and, as a consequence, the individual layer has a significantly lower latency than its neighboring layers.

Adjustment of the reuse parameter effectively modifies the initiation interval of each layer. A reuse factor of 5,000, corresponds to a layer that has an initiation interval of 5,000. To efficiently adjust the reuse parameters with `hls4ml`, the reuse needs to split the dense matrix multiply embedded within the layer across DSPs so as to maintain a regular systolic-array architecture. As a consequence, optimal implementations of the reuse can only be certain numbers, which is determined by the number of input and output features of each layer. Our current implementation is near ideal since 1.5 ms allows for a consistent set of reuse values that are near the 1.5-ms ideal latency point. To achieve a higher throughput, we need to adjust the reuse factor to the desired throughput and re-implement the whole design. Although this procedure requires a lot of computing, the whole procedure is automated through the *Algean* design flow.

When adjusting the reuse factor, we observe a direct correlation with the number of DSPs. Halving the reuse factor will halve the initiation interval of the matrix multiply within a layer, and it will also double the number of DSPs. Flip-Flops, and LUTs will not change as significantly since they largely exist to store partial images. BlockRAMs are used primarily to store weights of the neural network on the FPGA. Their second use is to act as a buffer between layers. As a consequence, the BlockRAM resources will not change significantly with reuse factor. In this current implementation, since DSP sharing of the multiplications is only partially used, the resulting resources are more consistent with a ResNet-50 implementation having a latency of roughly half the observed latency (0.75 ms).

Faster implementations of ResNet-50 are possible by adjusting the reuse factor. However, for CNNs, a lower bound is present in the current, pixel-by-pixel implementation of the algorithm. The lower bound results from the fact that for each pixel that streams through the algorithm, there is a one clock latency. Furthermore, there is an additional latency of three clocks to prepare the inputs to run the matrix multiplication. For layers within the network, where there are many pixels, such as the first layer, the ultimate latency is limited by these operations. Applying this limit to the first layer of ResNet-50, we find that the single layer throughput is bounded to be greater than roughly 0.4 ms. Lower single-inference latencies can still be achieved by splitting the image into sub-images and simultaneously streaming these sub-image streams into separate, cloned implementations of the chosen layer. Although the use of multiple streams effectively reduces the single inference throughput by the $(\text{number of streams})^{-1}$, it has the added cost of increasing the resources by the number of streams.

6 FUTURE WORK

The work demonstrated within this article is the first prototype of what is possible with an open multi-FPGA ML platform. This leaves much room to improve in all areas of the ML stack in Figure 1. Within the hardware section, there leaves much room for optimization of the IP cores themselves. Furthermore, the potential for splitting images into multiple streams can effectively block any throughput limitation at the cost of large resource usage, as well as larger throughput.

Once the IP cores are further optimized, it is our hope that the communication once again is the bottleneck. When this is the case, then we should explore more intelligent partitioning schemes that limit communication across the FPGA boundary. At the moment, this is a greedy solution

looking solely at resource utilization without taking into consideration the communication patterns between IP cores within the cluster.

Finally, `hls4ml` has the flexibility to run a broad range of other network architectures including transformer networks [55] and binary/ternary networks [56]. This work and new developments with `hls4ml` can be directly integrated into the *Algean* flow. We can now explore a broad range of deep learning architectures with many different sizes across multiple FPGAs.

7 CONCLUSION

Algean is a platform for mapping ML applications onto a cluster of network-connected FPGAs. This is much more scalable and has higher performance for computing than using the FPGA vendor tools, which are principally targeted at a single server with a handful of PCIe-connected FPGAs. Results from our initial implementations of actual networks show the benefits of using FPGAs for low-latency applications. We have also built two implementations of ResNet-50 to show that *Algean* can implement very large networks.

The structure of *Algean* is a number of abstraction layers spanning the entire computing stack from the ML development layer at the top to the physical hardware layer at the computing and communication layer implemented in the FPGA. This gives multiple opportunities to optimize the computing stack and for research depending on the area of interest and the design expertise available—that is, from ML algorithms down to low-level hardware design.

The layered approach makes it easier to implement *Algean* because it is possible to leverage the Galapagos multi-FPGA platform and only add an additional application bridge to the Galapagos library. It also makes it possible to quickly add automation in the translation of the `hls4ml` protocols and interfaces to the Galapagos protocol.

By leveraging Galapagos, *Algean* is also portable to other FPGA platforms as long as the low-level hypervisor layer in the FPGA is created. *Algean* also leverages the ability of Galapagos to deploy computing kernels to either CPUs or FPGAs such that an application can be first debugged and characterized entirely in software before committing all or parts of it to FPGA hardware.

The experience of developing *Algean* has demonstrated the challenges of building a multi-FPGA application development platform that is portable across many FPGA boards, but it proves that it is feasible in a reasonable amount of time.

Algean is available as an open source project and can be downloaded at <https://github.com/UofT-HPRC/Algean>.

A APPENDIX

This is an appendix covering details on both `hls4ml` and *Galapagos* as well as the details about the models we used in Section 5.

A.1 HLS4ML

`Hls4ml` was initially designed to address the necessity for ultra low latency inference. In this context, it was developed to allow for the possibility of deep neural network inference at timescales below $1\mu\text{s}$ that are pipelined with initiation intervals that are a few tens of nanoseconds. Such low-latency, high throughput networks are required to process information at the Large Hadron Collider in an all FPGA system with an approximate throughput of 1 Petabit/s. In this context, algorithms were explicitly designed to achieve the fastest possible latency at the cost of utilizing more hardware. In place of reusing network layers, deep neural networks are unrolled on the processor to allow for sequential, batch-1 processing of network inferences with initiation intervals smaller than the total network latency. As a result of this design flow, the focus on `hls4ml` was towards

the development of small networks that used a large amount of resources but that could also run at very low latencies.

Extending this paradigm to larger networks, such as ResNet-50, was not considered part of the scope of hls4ml since such large networks would require resources greater than a single FPGA. As a consequence, hls4ml did not contain large layer implementations. However, with *Algean*, through the distribution of a single network across many FPGAs, the possibility of extensive, low-latency networks under the hls4ml design flow is achievable. Consequently, hls4ml was extensively adapted to handle these large networks to create *Algean*. In particular, we created or heavily adapted implementations of many new layers of hls4ml, including:

- Dense/Linear Layer
- CNN Layer
- Pooling Layer
- Split Layer
- Merge Layer

These layers make up the core of most deep neural networks used. Furthermore, the large layer design flow established through the development of *Algean* will enable the fast implementation of other layers following the *Algean* implementations. In this appendix, we will outline the various adaptations needed in hls4ml to run a wide variety of algorithms at large scales. Furthermore, we will comment on new features added to hls4ml that enhance the core software framework and enables large model, high throughput implementations.

A.1.1 Design Flow. To enable large CNN layers within hls4ml, the flow of hls4ml was re-factorized to work at longer latencies. In the original hls4ml implementation, to run NNs at ultra fast latencies, layers were connected through large arrays written simultaneously. For *Algean*, we added the option of streaming arrays that allow for the outputs to be arrays of streams, which are then interfaced with Galapagos. Arrays of streams allow for the ability to stream out partial results between layers. This element is crucial for processing data under a CNN where the application of the same CNN kernel is repeated many times on a different part of a larger image. Arrays of streams differ from other network architectures, particularly FINN [26], in that there is still the potential for substantial throughput between layers since the array size can be adjusted to the latency requirements.

A.1.2 Neural Network Weights. Within hls4ml, the NN weights are compiled and embedded within the HLS project. While this feature can allow for optimized place and route of neural networks that account for unstructured pruning, this adds a complexity to the HLS compilation that can substantially slow down the compilation time. To avoid this and to have the added flexibility of weight updates, we have added the functionality to treat the weights as external Block RAM or UltraRAM ports. The use of external ports keep the total HLS compilation time to under one hour per layer.

A.1.3 Dense/Linear Layer. A single dense (tf notation) or linear (PyTorch notation) layer is the core of most NN implementations where matrix multiplication is performed. In hls4ml, this consists of a systolic array with a dedicated size compiled directly for that designed layer. In particular, the parameter reuse is built into the dense layer construction, and it dictates the size of the systolic array through the use of DSPs. To adapt the dense layer for *Algean* we added the possibility of streaming between layers. To account for partial images of streams, we embedded NN flattening within the dense layer. The dense layer multiplications were modified to allow for the option of merging two 8-bit multiplies into one single DSP operation [57].

A.1.4 CNN Layer. To avoid large outputs and to improve the overall throughput, a new CNN layer was developed, which operates on an image pixel by pixel. In this scenario, images are streamed through an array of streams between layers, with each depth element in the stream corresponding to a single pixel of an image. Pixels are then streamed one at a time into a layer, and the resulting output pixel is streamed to the next set of layers. The partial image is stored within a layer using a line buffer implementation. The line buffer is implemented as an array of shift registers, and we rely on the specialized HLS shift register objects to ensure the final implementation uses explicit shift register logic elements (SRLs). The line buffer is further optimized to store the minimal number of pixels required by the kernel size (Convolution kernel height \times row). An intermediate buffer is also used to store the kernel window before the matrix multiply needed for the convolution kernel. The matrix multiplication within the CNN kernel uses the default dense layer within hls4ml. The total latency before the kernel matrix multiplication takes three clocks (one for reading the inputs, one for the appreciation of the shift register, and one for filling the kernel window).

The reuse factor for the CNN kernel, thus, defines the reuse per output pixel (i.e., the reuse is tied to the matrix multiplication for the Kernel). For the base implementation, the overall latency of the convolution kernel is five clocks per output pixel + the additional reuse factor for the dense layer. This five-clock overhead can be reduced for small networks by utilizing a fully partitioned line buffer at the cost of more resources. Zero Padding for the individual layers is built into the layer implementations.

A.1.5 Pooling Layer. In addition to a CNN layer implementation, a pooling layer is added following the same data flow as in the CNNs (array of streams). The pooling layer implementation is similar to the CNN layer, except that it performs pooling instead of the convolution kernel matrix multiplication present in the CNN.

A.1.6 Split/Merge Layers. To allow for the possibility of ResNet-50, split and merge layers are added to hls4ml using the array of streams data flow. Since the splitting of arrays of streams can use a large number of resources, the split and merge layers are generated with the ability to time-multiplex and de-multiplex the streams. This leads to a longer latency to perform the layer operation. However, the throughput for these layers is typically heavily subdominant to other layers in a network.

A.1.7 Reuse Factor. Hls4ml has one main tuneable parameter, the reuse factor. The reuse factor defines the usage parameter for how often a DSP is reused within a matrix multiply. For example, a reuse factor of 25 implies that a single DSP will be used 25 times to perform single matrix multiplication. As a consequence, the initiation interval of the layer with reuse 25 would be 25 clocks. Furthermore, the number of DSPs used would be equal to the total number of multiplications in that layer divided by the reuse factor. To ensure a regular architecture, the reuse factor needs to be a multiple of the number of total multiplications used in the layer. As an example, consider the last convolutional layer in ResNet-50. This layer consists of matrix multiplication of a (1×1 pixel kernel) with 512 input features and 2048 output features or 1.04M multiplications. For a 0.25ms implementation of ResNet-50, a reuse factor of 1024 is used, leading to an initiation interval of 1024 clocks, and 1024 DSPs without DSP merging, and 512 with DSP merging. Furthermore, this convolution kernel is run on a 7×7 input image or 49 separate times on ResNet-50 leading to a total latency of exactly 0.25ms.

The reuse factor for the CNN points directly to the reuse factor of the Dense layer implementation. This reuse factor of the dense layer has two internal implementations depending on the size of the reuse factor. For instances where the reuse factor is smaller than the number of input

features, the systolic array is split across the input regions so that neighbouring multiplications are accumulated into the same or adjacent output feature. For instances where the reuse factor is larger than the number of inputs, the systolic array is split across the output feature. The input features are multiplexed and multiplied before being accumulated across the output features. These optimizations were chosen to ensure optimal resource usage in the matrix multiply to allow for large matrix multiplications with millions of weights. As a consequence of these choices, certain reuse factors that are multiples of the inputs and outputs are particularly resource optimized. Within hls4ml code generation, an automatic adjustment of reuse factor is performed to allow for the nearest optimized reuse factor for that layer.

To allow for a balanced throughput between the layers, we have modified the reuse factor for *Algean* kernels to account for the per layer throughput. Instead of defining the DSP reuse factor for the network, the reuse factor per layer is dynamically computed based on a desired per-layer throughput for the total network. To compute the optimized reuse factor, we rely on an analytic formula of the total layer throughput based on the reuse factor. This analytic formula yielding the total throughput per layer, R , in clocks, is defined as

$$R = N_{pixel} \left(6 + \frac{N_{in}N_{out}}{r} \right) \quad (1)$$

Where in this case, r is the reuse factor per layer, and N_{pixel} is the number of pixels in a CNN layer, N_{in} is the number of input features and N_{out} is the number of output features. The additional 6 approximates the number of clocks per pixel that a single layer requires to perform a shift and fill and reuse one matrix multiply; this presents a lower bound on the latency for a single CNN layer of $6N_{pixel}$. Ultimately, this limitation is soft since multiple CNN layers can be used on separate parts of an image. Within the hls4ml configuration, the throughput latency R is defined in clocks before the project generation, and then the reuse factors are automatically computed in the project generation. As a consequence, to change the throughput, one need only adjust to the desired latency R , regenerate and recompile the project.

A.2 Models

In this appendix, we present a detailed description of each NN model used in this paper. The choice of these architectures is partly motivated by use in high energy physics, where low-latency deep neural network inference is an essential tool for operation. As a consequence, we comment on the model architecture and its application to problems within physics.

A.2.1 Autoencoder. Autoencoders are unsupervised networks often used to identify anomalous features. By creating an information bottleneck within the network, autoencoders can compress and classify detector level information. The autoencoder considered in this example is capable of identifying LHC collisions that occur anomalously at the LHC. In particular, this network can identify top quark pair productions, Higgs boson pair production, and other more exotic final states.

At the Large Hadron Collider, this network has a direct application through the integration of data scouting/trigger level analyses [58]. Data scouting is a process whereby partially reconstructed collisions are read out and processed to investigate collisions that are normally thrown out in the LHC data flow chain. This technique is particularly powerful in the search for Dark Matter [59]. In this case, events can be analyzed at a rate as large as 40 MHz, and the autoencoder can be used to create an “anomaly stream” at a reduced rate. With a maximum data rate of the order of 50 Terabits per second within the LHC trigger “scouting” stream, throughput and low latency are critical, since any delay in even a single inference would require significantly larger buffers.

Distribution of this system onto multiple FPGAs brings a significant advantage since it would allow for very low latency while preserving the ability to pipeline events with small initiation interval.

The autoencoder network is trained using events that involve known and understood physics processes. Thus, any event that cannot be encoded and decoded accurately is a potential candidate for new physics searches. The inputs and outputs to the network are 276 expert event features. The number of hidden features in the first layer is 276. The second layer goes down by 1/3 to 184, the third by 1/2 to 92, and remains having 92 features for the next 6 layers before the hidden features expand back symmetrically to 276 output features. The compression factor in the bottleneck is thus 3, and in total, the network consists of 12 fully connected layers and over 300,000 weights. Between the fully connected layers Relu activation is used.

A.2.2 ResNet-50. Lastly, we consider the well-known ResNet-50 benchmark. ResNet-50 is a deep neural network used for image processing [60]. The ResNet-50 architecture has been shown to have a lot of versatility. In particular, quantized ResNet-50 has been retrained for the process of top quark identification within high energy physics, leading to results that are comparable to world leading algorithms [61]. More recently, ResNet-50 has become the standard algorithm for benchmarking algorithm performance. With the development of a quantized neural network, the 8-bit implementation of ResNet-50 has taken over the floating point implementation as the standard benchmarking algorithm for neural network inference. The 8-bit ResNet-50 has been shown to yield almost identical performance to the full ResNet-50 implementation with the added advantage of 8-bit operations.

For the implementation of ResNet-50 considered in this paper, we use the 8-bit implementation. The network consists of 50 convolutional layers, 2 pooling layers, a dense layer, 16 merge layers, 16 split layers, and 50 batch normalization layers. To minimize the total amount of computation, the batch normalization layers are fused with the convolutional layers. ResNet-50 takes an input image that is 224×224 pixels and will iteratively reduce the size of the image from 224×224 to a final image that is 7×7 pixels. The total number of multiplications present in ResNet-50 is 4.8 Billion. For a target throughput of 650 Hz, this translates to 3.1 trillion multiplications per second or 15000 continuous multiplications at 200 MHz. In the *AIgean* implementation, we perform two 8-bit multiplications per DSP. As a consequence, we require 7.5k DSPs for the full implementation if we have 100% efficiency. Our actual usage is 9.9k DSPs, corresponding to an aggregate 77% efficiency. The reason for the 77% efficient computation, and not higher, is a result of the fact that the ResNet-50 target design was actually synthesized for a throughput faster than 1.5 ms per image to ensure the target latency is met. The latency range of each layer ranges between 1.0 ms to 1.4 ms, with most layers having a throughput of 1.3 ms.

A.3 Galapagos

Galapagos is a heterogeneous deployment stack that allows users to deploy streaming IP cores on a cluster of FPGAs and CPUs. First we will describe the high-level abstraction model of Galapagos and then delve into each layer of abstraction that we implemented.

A.3.1 High-Level Abstraction. The end goal of Galapagos is to be able to overlay a data flow graph of streaming IP cores to a cluster of devices, without the user having to worry about how to physically connect these IP cores amongst each other on one device as well as across devices. These IP cores should be able to address their destinations with respect to their target IP core, independent to how and where the IP core is implemented and placed. The IP cores themselves use AXI-stream to communicate with each other, with a destination side-channel. This is typically used within a single FPGA for routing amongst AXI-stream kernels. Our goal is to be able to provide this seamlessly across many devices to provide the user an abstraction similar to a single

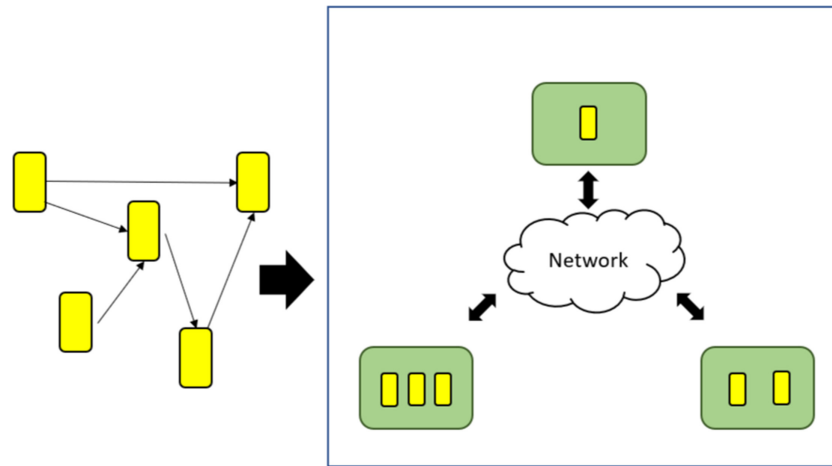


Fig. 7. An overview of Galapagos. The user provides the implementation and network agnostic data flow graph of streaming IP cores, and our tool flow implements the right hand side, with the appropriate bridging to connect the devices together.

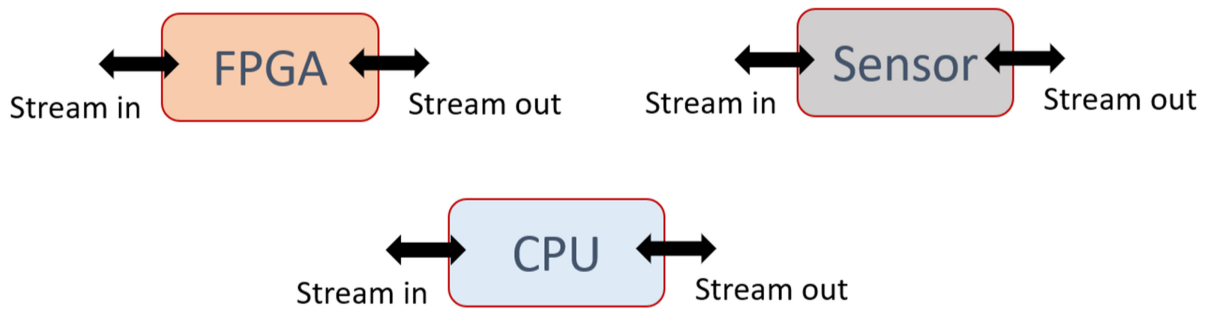
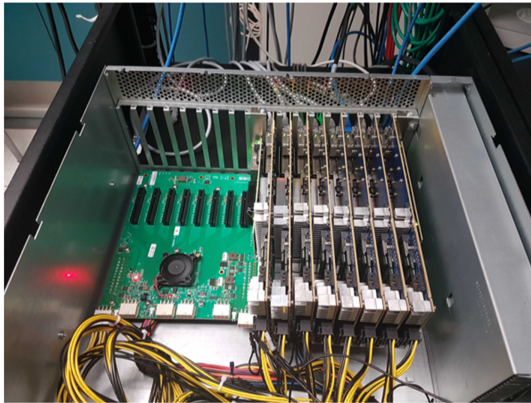


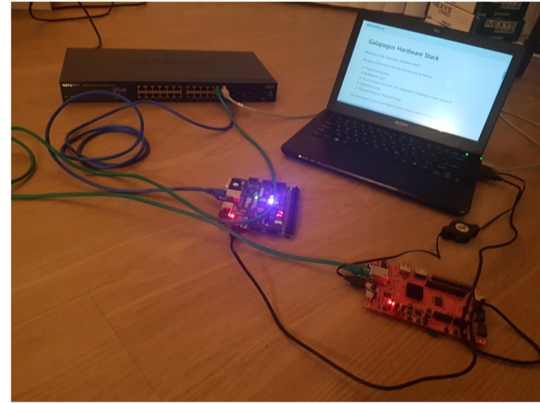
Fig. 8. An example of the lowest level of abstraction. With the abstraction provided, all of these streaming devices have a consistent interface and can communicate with one another

device. This can be seen as AXI-stream over the data center. We accomplish this by automating the encapsulation and decapsulation of AXI-stream packets with higher level network protocols. Figure 7 shows the high-level overview of Galapagos. On the left-hand side is a placement and implementation agnostic data-flow graph of streaming IP cores, and the right-hand side is how they are placed and implemented. Prior to this work the user had to provide configuration parameters to define the mapping of kernels to devices, but even that is now abstracted away with a partitioner. While the vendor tool kits provide the ability to add networking to a design [62, 63] the application must be explicitly aware of the networking.

Galapagos is built from the bottom up through several layers of the stack. From the bottom, individual devices are abstracted to appear to be streaming devices. This is shown in Figure 8. We have implemented this on different FPGAs and CPUs but this could be extended to other devices such as IoT sensors. Once each device is abstracted we can connect them together seamlessly at the protocol level. Furthermore we can look at a finer granularity of IP cores that can run on these devices, and even migrate implementations of these nodes as long as they have a consistent interface. Galapagos provides higher levels of abstraction to place streaming IP cores on these devices and connect and route amongst these IP cores on one device as well as target multiple devices. This is done through the implementation of the following layers of the stack: Physical Hardware and Connectivity, Hypervisor, Middleware Layer, and the Communication Layer.



MPSoC Boards Connected via 100 GB/s Cables
~\$100 000



Nexys, Pynq, Laptop, 1G switch
~\$2000

Fig. 9. Two examples of “data centers” where we deployed Galapagos.

A.3.2 Physical Hardware and Connectivity. This layer of the stack refers to the physical devices you would have in your cluster and how they are connected. Currently we have created clusters of FPGAs (The Fidus Sidwinder, Alhadata 7v3, Pynq ZC702), x86 CPUs, and connectivity using 1G Ethernet, 10G SFP, and 100G QSFP. Currently the requirement for these devices is to have some connection to the network that we can connect to a network switch, but even this requirement can be abstracted in some way by the Hypervisor above. We have tested the same abstraction layers on these different devices to show the consistency of our higher levels of abstraction. Two examples of Galapagos setups can be seen in Figure 9.

A.3.3 Hypervisor. This layer of the stack refers to the abstraction of the physical device to standardize their interfaces to appear like Figure 8. Our standard model assumes a control path and a data path. The control path is used for configuration, programming, and monitoring the devices. Typically in our FPGAs we either use PCIe for FPGAs that do not have a tightly coupled ARM (Alhadata 7v3), or AXI for FPGAs with a tightly coupled ARM (Fidus Sidwinder, Pynq ZC702). The data path is used by the application IP cores to communicate off-chip to other nodes within the cluster. For the hypervisor to comply with the rest of the layers of the stack, it has to provide an AXI-stream interface. This standardization makes it simple for a user to add their own board within the stack. All they would need to do is provide an AXI-stream interface that can connect to a network switch. An example FPGA hypervisor is shown in Figure 10.

A.3.4 Middleware and Communication Layers. The middleware is responsible for partitioning the kernels onto different FPGAs. This was previously done by a user-specified configuration, that provides a hint to the our middleware layer to place kernels on different devices. However we now automate this partitioning. This is described in Section 4.1.3. Once we have the placement of kernels, the middleware then places bridges to allow for kernels to communicate off-chip. The hypervisor guarantees an AXI-stream without a side channel available for the destination field. However with a Galapagos router and bridge we can take AXI-stream packets destined for off-chip and append the destination as a header. The Galapagos router has a routing table that specifies the location of all kernels by destination within the cluster. Furthermore, the off-chip communication can be done over various network communication protocols, handled by the communication layer. Depending on the destination FPGA, our network bridge encapsulates the packet with the correct network header. The network bridge is specific for each off-chip communication protocol the user wishes to support. If the user wishes to implement Galapagos on top of their own network protocol,

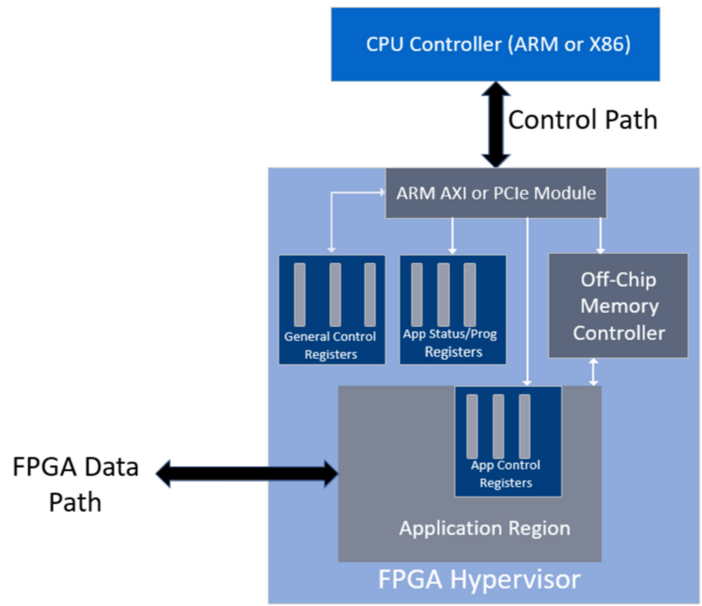


Fig. 10. An example Galapagos FPGA Hypervisor.

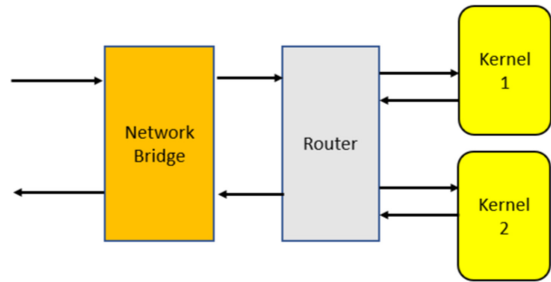


Fig. 11. Automated Middleware IP cores in Galapagos.

they would need to supply a bridge that can translate their network packets into AXI-stream packets with a Galapagos header. The formation of the Galapagos router, routing table, and network bridges is all automated. The IP cores generated by the Middleware are shown in Figure 11.

ACKNOWLEDGMENTS

We sincerely thank the reviewers for their helpful comments that significantly improved the quality of this article.

REFERENCES

[1] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, et al. 2016. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE, Los Alamitos, CA, Article 7, 13 pages.

[2] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, et al. 2018. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (March 2018), 8–20. <https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/>.

[3] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 1–14. DOI : <http://dx.doi.org/10.1109/ISCA.2018.00012>

- [4] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 7 (July 2018), 305.
- [5] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2017. Enabling flexible network FPGA clusters in a heterogeneous cloud data center. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, 237–246.
- [6] Naif Tarafdar, Nariman Eskandari, Varun Sharma, Charles Lo, and Paul Chow. 2018. Galapagos: A full stack approach to FPGA integration in the cloud. *IEEE Micro* 38, 6 (2018), 18–24.
- [7] Xilinx. n.d. Xilinx Vitis AI. Retrieved April 10, 2021 from <https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html>.
- [8] Intel. n.d. OpenVINO. Retrieved April 10, 2021 from <https://www.intel.com/content/www/us/en/artificial-intelligence/programmable/solutions.html>.
- [9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).
- [10] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: A Modular Machine Learning Software Library*. Technical Report. Idiap.
- [11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*. ACM, New York, NY, 675–678.
- [12] TensorFlow. n.d. Distributed Training with TensorFlow. Retrieved April 8, 2021 from https://www.tensorflow.org/guide/distributed_training/.
- [13] Google. n.d. Cloud Tensor Processing Units (TPUs). Retrieved April 8, 2021 from <https://cloud.google.com/tpu/docs/tpus/>.
- [14] NVIDIA. 2020. NVIDIA NVLink Fabric: Advanced Multi-GPU Processing. Retrieved November 3, 2021 from <https://www.nvidia.com/en-us/data-center/nvlink>
- [15] NVIDIA. n.d. NVIDIA NCCL. Retrieved April 8, 2021 from <https://developer.nvidia.com/nccl/>.
- [16] NVIDIA. n.d. NVIDIA Completes Acquisition of Mellanox, Creating Major Force Driving Next-Gen Data Centers. Retrieved April 10, 2021 from <https://nvidianews.nvidia.com/news/nvidia-completes-acquisition-of-mellanox-creating-major-force-driving-next-gen-data-centers>.
- [17] Xilinx. 2019. Machine Learning (ML) Suite. Retrieved November 3, 2021 from <https://github.com/Xilinx/ml-suite>.
- [18] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O’Connell, Nitika Shanker, Joseph Chu, et al. 2018. DLA: Compiler and FPGA overlay for neural network inference acceleration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL ’18)*. IEEE, Los Alamitos, CA, 411–4117.
- [19] Xilinx. 2020. SDAccel Development Environment. Retrieved November 3, 2021 from <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>.
- [20] Intel. 2020. Intel FPGA SDK for OpenCL Software Technology. Retrieved November 3, 2021 from <https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html>.
- [21] Lester Kalms and Diana Göhringer. 2017. Exploration of OpenCL for FPGAs using SDAccel and comparison to GPUs and multicore CPUs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL ’17)*. IEEE, Los Alamitos, CA, 1–4.
- [22] E. Nurvitadhi, A. Boutros, P. Budhkar, A. Jafari, D. Kwon, D. Sheffield, A. Prabhakaran, K. Gururaj, P. Appana, and M. Naik. 2019. Scalable low-latency persistent neural machine translation on CPU server with multiple FPGAs. In *Proceedings of the 2019 International Conference on Field-Programmable Technology (ICFPT’19)*. 307–310.
- [23] GitHub. n.d. Open Programmable Acceleration Engine. Retrieved November 3, 2021 from <https://opae.github.io/>.
- [24] Xilinx. 2018. CHaiDNN. Retrieved November 3, 2021 from <https://github.com/Xilinx/CHaiDNN>.
- [25] Xilinx. 2019. PYNQ DL. Retrieved November 3, 2021 from <https://github.com/Xilinx/PYNQ-DL>.
- [26] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, 65–74.
- [27] Dong Wang. 2019. An OpenCL-based FPGA Accelerator for Convolutional Neural Networks. Retrieved November 3, 2021 from <https://github.com/doonny/PipeCNN>.
- [28] HLSLibs. 2019. Open-Source High-Level Synthesis IP Libraries. Retrieved November 3, 2021 from <https://hlslibs.org>.
- [29] Lucian Petrica, Tobias Alonso, Mairin Kroes, Nicholas J. Fraser, Sorin Cotofana, and Michaela Blott. 2020. Memory-efficient dataflow inference for deep CNNs on FPGA. *CoRR* abs/2011.07317 (2020). arXiv:2011.07317 <https://arxiv.org/abs/2011.07317>.

- [30] Mathew Hall and Vaughn Betz. 2020. From TensorFlow graphs to LUTs and wires: Automated sparse and physically aware CNN hardware generation. In *Proceedings of the 2020 International Conference on Field-Programmable Technology (ICFPT'20)*. 56–65. DOI: <http://dx.doi.org/10.1109/ICFPT51103.2020.00017>
- [31] François Chollet. 2020. Keras: The Python Deep Learning Library. Retrieved November 3, 2021 from <https://keras.io>
- [32] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS'17)*.
- [33] Junjie Bai, Fang Lu, and Ke Zhang. 2019. ONNX: Open Neural Network Exchange. Retrieved November 3, 2021 from <https://github.com/onnx/onnx>.
- [34] Claudionor N. Coelho, Aki Kuusela, Shan Li, Hao Zhuang, Thea Aarrestad, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Adrian Alan Pol, and Sioni Summers. 2020. Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders. *arXiv:2006.10159* (2020). [arXiv:physics.ins-det/2006.10159](https://arxiv.org/abs/2006.10159)
- [35] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, et al. 2015. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2015), 1591–1604.
- [36] Xilinx. 2020. Vivado Design Suite. Retrieved November 3, 2021 from <https://www.xilinx.com/products/design-tools/vivado.html>.
- [37] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, M. B. Valentin, et al. 2021. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. In *Proceedings of the tinyML Research Symposium 2021*. [arXiv:cs.LG/2103.05579](https://arxiv.org/abs/2103.05579)
- [38] T. Aarrestad, V. Loncar, N. Ghielmetti, M. Pierini, S. Summers, J. Ngadiuba, C. Petersson, et al. 2021. Fast convolutional neural networks on FPGAs with hls4ml. *arXiv:2101.05108* (1 2021). [arXiv:cs.LG/2101.05108](https://arxiv.org/abs/2101.05108)
- [39] A. Heintz, V. Razavimaleki, J. Duarte, G. DeZoort, I. Ojalvo, S. Thais, M. Atkinson, et al. 2020. Accelerated charged particle tracking with graph neural networks on FPGAs. In *Proceedings of the 34th Conference on Neural Information Processing Systems*. [arXiv:physics.ins-det/2012.01563](https://arxiv.org/abs/2012.01563)
- [40] S. Summers, G. Di Guglielmo, J. M. Duarte, P. Harris, D. Hoang, S. Jindariani, E. Kreinar, et al. 2020. Fast inference of boosted decision trees in FPGAs for particle physics. *Journal of Instrumentation* 15, 05 (2020), P05026. DOI: <http://dx.doi.org/10.1088/1748-0221/15/05/P05026> [arXiv:physics.comp-ph/2002.02534](https://arxiv.org/abs/physics.comp-ph/2002.02534)
- [41] Y. Iiyama, G. Cerminara, A. Gupta, J. Kieseler, V. Loncar, M. Pierini, S. R. Qasim, et al. 2020. Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction in high energy physics. *Frontiers in Big Data* 3 (2020), 598927. DOI: <http://dx.doi.org/10.3389/fdata.2020.598927> [arXiv:physics.ins-det/2008.03601](https://arxiv.org/abs/physics.ins-det/2008.03601)
- [42] Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. 2018. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems* 11, 3 (2018), Article 16, 23 pages. [arXiv:cs.AR/1809.04570](https://arxiv.org/abs/cs.AR/1809.04570)
- [43] ARM. 2010. AMBA 4 AXI4-Stream Protocol. Retrieved April 19, 2020 from https://static.docs.arm.com/ih0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf.
- [44] Marc Snir. 1998. *MPI—The Complete Reference: The MPI Core*. Vol. 1. MIT Press, Cambridge, MA.
- [45] Naif Tarafdar and Paul Chow. 2019. libGalapagos: A software environment for prototyping and creating heterogeneous FPGA and CPU applications. In *Proceedings of the 6th International Workshop on FPGAs for Software Programmers*. 1–7.
- [46] Qianfeng Shen. 2019. GULF-Stream. Retrieved January 13, 2020 from <https://github.com/QianfengClarkShen/GULF-Stream>.
- [47] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. 2015. Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'15)*. IEEE, Los Alamitos, CA, 36–43.
- [48] Intel. 2019. *Intel 82599 10 GbE Controller Datasheet*. Intel.
- [49] Fidus. 2019. Fidus Sidewinder. Retrieved January 13, 2020 from <https://fidus.com/products/sidewinder/>
- [50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
- [51] Javier Duarte, Philip Harris, Scott Hauck, Burt Holzman, Shih-Chieh Hsu, Sergio Jindariani, Suffian Kha, et al. 2019. FPGA-accelerated machine learning inference as a service for particle physics computing. *arXiv preprint arXiv:1904.08986* (2019).
- [52] NVIDIA. 2021. NVIDIA Data Center Deep Learning Product Performance. Retrieved June 17, 2021 from <https://developer.nvidia.com/deep-learning-performance-training-inference>.
- [53] Amazon. 2021. Amazon EC2 F1 Instances. Retrieved January 19, 2021 from <https://aws.amazon.com/ec2/instance-types/f1/>.

- [54] Xilinx. 2020. Alveo U200 and u250 Data Center Accelerator Cards Data Sheet. Retrieved January 19, 2021 from https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf.
- [55] Yutaro Iiyama, Gianluca Cerminara, Abhijay Gupta, Jan Kieseler, Vladimir Loncar, Maurizio Pierini, Shah Rukh Qasim, et al. 2021. Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction in high energy physics. *Frontiers in Big Data* 3 (2021), 598927. arXiv:[hep-ex/2008.03601](https://arxiv.org/abs/2008.03601)
- [56] Jennifer Ngadiuba, Vladimir Loncar, Maurizio Pierini, Sioni Summers, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, et al. 2020. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. *Machine Learning: Science and Technology* 2, 1 (Dec. 2020), 015001. DOI : <http://dx.doi.org/10.1088/2632-2153/aba042>
- [57] Xilinx. 2017. Deep Learning with INT8 on Xilinx Devices. Retrieved November 3, 2021 from https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf.
- [58] Javier Duarte. 2018. Fast reconstruction and data scouting. In *Proceedings of the 4th International Workshop Connecting the Dots 2018*. arXiv:[hep-ex/1808.00902](https://arxiv.org/abs/1808.00902)
- [59] A. M. Sirunyan, A. Tumasyan, W. Adam, F. Ambrogi, T. Bergauer, M. Dragicevic, J. Ero, et al. 2020. Search for a narrow resonance lighter than 200 GeV decaying to a pair of muons in proton-proton collisions at $\sqrt{s} = \text{TeV}$. *Physical Review Letters* 124, 13 (2020), 131802. DOI : <http://dx.doi.org/10.1103/PhysRevLett.124.131802> arXiv:[hep-ex/1912.04776](https://arxiv.org/abs/1912.04776)
- [60] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *CoRR* abs/1512.03385 (2015). arXiv:[1512.03385](https://arxiv.org/abs/1512.03385) <http://arxiv.org/abs/1512.03385>
- [61] J. Duarte, P. Harris, S. Hauck, B. Holzman, S.-C. Hsu, S. Jindariani, S. Khan, et al. 2019. FPGA-accelerated machine learning inference as a service for particle physics computing. *Computing and Software for Big Science* 3, 1 (2019), 13. DOI : <http://dx.doi.org/10.1007/s41781-019-0027-2> arXiv:[physics.data-an/1904.08986](https://arxiv.org/abs/physics.data-an/1904.08986)
- [62] GitHub. n.d. Xilinx Vitis Network Example. Retrieved November 4, 2021 from https://github.com/Xilinx/xup_vitis_network_example.
- [63] GitHub. n.d. Xilinx Vitis with 100G TCP/IP. Retrieved November 4, 2021 from https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP.

Received January 2021; revised June 2021; accepted July 2021



TAILOR: Altering Skip Connections for Resource-Efficient Inference

OLIVIA WENG and GABRIEL MARCANO, University of California San Diego, USA

VLADIMIR LONCAR, Massachusetts Institute of Technology, USA

ALIREZA KHODAMORADI, AMD, USA

ABARAJITHAN G and NOJAN SHEYBANI, University of California San Diego, USA

ANDRES MEZA and FARINAZ KOUSHANFAR, University of California San Diego, USA

KRISTOF DENOLF, AMD, USA

JAVIER MAURICIO DUARTE and RYAN KASTNER, University of California San Diego, USA

Deep neural networks use skip connections to improve training convergence. However, these skip connections are costly in hardware, requiring extra buffers and increasing on- and off-chip memory utilization and bandwidth requirements. In this paper, we show that skip connections can be optimized for hardware when tackled with a hardware-software codesign approach. We argue that while a network's skip connections are needed for the network to learn, they can later be removed or shortened to provide a more hardware efficient implementation with minimal to no accuracy loss. We introduce TAILOR, a codesign tool whose hardware-aware training algorithm gradually removes or shortens a fully trained network's skip connections to lower their hardware cost. TAILOR improves resource utilization by up to 34% for BRAMs, 13% for FFs, and 16% for LUTs for on-chip, dataflow-style architectures. TAILOR increases performance by 30% and reduces memory bandwidth by 45% for a 2D processing element array architecture.

CCS Concepts: • **Hardware** → **Hardware-software codesign**; • **Computer systems organization** → **Neural networks**.

Additional Key Words and Phrases: Hardware-software co-design, neural networks

1 INTRODUCTION

Convolutional neural networks (NNs) often rely on skip connections—identity functions that combine the outputs of different layers—to improve training convergence [17, 45]. Skip connections help mitigate the vanishing gradient problem [4, 15] that occurs when training large CNNs, which helps increase the network's accuracy. Skip connections allow NNs to have fewer filters/weights than architectures that lack skip connections [17], such as VGG [43].

However, skip connections are generally detrimental to hardware efficiency. They have an irregular design that is ill-suited for hardware acceleration. This is due to their long lifetimes, which span several NN layers, increasing memory utilization and bandwidth requirements. This is particularly true in ResNets [17], which introduced skip connections that spanned across five layers: two convolutions, two batch normalizations (BNs), and a ReLU activation [16, 34] (see Fig. 1a). The skip connection involves minimal computation—it is either the identity or a 1×1 convolutional layer for scaling—but it extends the necessary lifespan of the input data. Thus, we must store skip connection data for the duration of time needed to compute the five NN layers. In total, a

Authors' addresses: Olivia Weng, oweng@ucsd.edu; Gabriel Marcano, University of California San Diego, USA; Vladimir Loncar, Massachusetts Institute of Technology, USA; Alireza Khodamoradi, AMD, USA; Abarajithan G; Nojan Sheybani, University of California San Diego, USA; Andres Meza; Farinaz Koushanfar, University of California San Diego, USA; Kristof Denolf, AMD, USA; Javier Mauricio Duarte; Ryan Kastner, University of California San Diego, USA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2023 Copyright held by the owner/author(s).

1936-7406/2023/9-ART

<https://doi.org/10.1145/3624990>

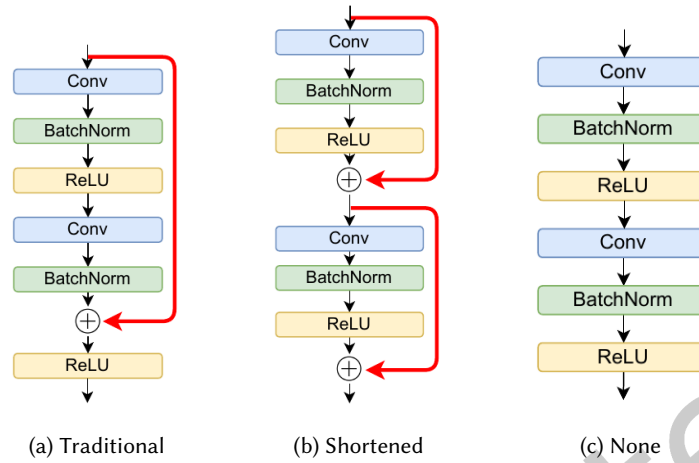


Fig. 1. Neural networks with traditional skip connections, like ResNet (a), have inefficient hardware implementations because the skip connection data must be preserved in memory during five layers of computation. This irregular topology increases memory resources and bandwidth. A more regular topology with reduced skip connection lifetimes would use fewer resources. TAILOR achieves this by shortening skip connections (b) or by eliminating them completely (c). Skip connections are in red.

model’s skip connection data accounts for $\sim 10\%$ of its memory bandwidth either on or off chip. Buffering skip connections on chip increases on-chip memory utilization, whereas moving them off chip not only increases off-chip memory bandwidth but also requires extra control logic for scheduling [29, 30].

Optimizing skip connections requires careful *hardware-software codesign*. Skip connections are crucial for model convergence; naively removing them to reduce hardware resources leads to low accuracy [32, 50]. Instead, we must codesign how the model is (1) trained and (2) implemented in hardware to achieve a model that is both accurate and resource-efficient.

We develop TAILOR, a codesign method that gradually alters a NN’s skip connections during training to produce a highly accurate and resource-efficient NN. Our results in Sec. 4 show that TAILOR can remove or shorten skip connections to achieve topologically regular NNs (Fig. 1b and 1c) that substantially reduce hardware resources, reduce memory bandwidth, and increase performance with minimal to no accuracy loss.

TAILOR takes an *existing* pre-trained model and reduces the hardware complexity of its skip connections with minimal to no accuracy loss. Moreover, TAILOR exploits the flexibility of the FPGA architecture to customize the skip connection memories, which is not possible on a GPU or CPU. TAILOR accomplishes this dynamically during retraining in one of two ways: (1) SKIPREMOVER removes the skip connections altogether (Fig. 1c) to eliminate all associated hardware complications or (2) SKIPSHORTENER shortens each skip connection by splitting it into multiple shorter ones (Fig. 1b).

We evaluate TAILOR’s applicability and benefit on ResNets [17, 18] and QuartzNets [23]—two important classes of NNs that contain skip connections of varying lengths. We also study implementing skip connections with an on-chip, dataflow-style FPGA architecture using hls4ml [2, 12] and a 2D array of multiply-accumulate processing elements. TAILOR reduces resource utilization of hls4ml architectures by up to 34% for BRAMs, 13% for FFs, and 16% for LUTs. TAILOR increases the performance of 2D array architecture by 30% and reduces memory bandwidth by 45%.

TAILOR’s hardware-software codesign approach reduces hardware complexity and resources by altering skip connections dynamically during retraining. Our contributions are:

- the TAILOR software methodology of removing or shortening skip connections from existing NNs with minimal to no loss in accuracy,
- the TAILOR hardware designs that exploit FPGA-specific architecture optimizations, which are not possible on GPU/CPU, to produce less resource-intensive skip connection implementations,
- experiments demonstrating that SKIPSHORTENER and SKIPREMOVER models are implemented more efficiently with better performance and resource utilization than their traditional skip connection counterparts,
- and public release of the Tailor hardware-software codesign framework [1].

In Sec. 2, we review related work. In Sec. 3, we explain how TAILOR’s NN alterations optimize the hardware architecture. We then describe TAILOR’s two training methods, SKIPREMOVER and SKIPSHORTENER, that alter skip connections with little to no loss in accuracy. Sec. 4 provides training, quantization, and hardware results for SKIPREMOVER and SKIPSHORTENER. Sec. 5 discusses the tradeoffs TAILOR presents between accuracy and hardware resource reductions. Sec. 6 concludes the paper.

2 BACKGROUND

2.1 Removing Skip Connections

While skip connection removal has been studied before [8, 25, 32, 50, 51], prior work is lacking in several ways: (1) preliminary work [32, 50, 51] only studies shallow models (up to 34 layers); (2) Li et al. [25] do not remove all of the skip connections in the models they evaluate; (3) Ding et al. [8] and Li et al. [25] both have limited architectural evaluations (e.g., GPU & mobile) that do not consider the highly customized skip connections memories enabled by FPGAs; and (4) Ding et al. [8] require starting with an entirely new NN topology whose skip connections are removable.

Monti et al. [32] start with a standard ResNet and introduce a new training method. This method uses an objective function that penalizes the skip connections and phases them out by the end of the training. This technique has only been applied to smaller ResNets (18 to 34 layers) with a small decrease in accuracy between 0.5 and 3%.

Zagoruyko and Komodakis [50] also develop a method for removing skip connections in a NN. They replace skip connections with Dirac parameterization, creating a new NN called DiracNet. The Dirac parameterization is shown in Eq. 1,

$$\text{DiracNet [50]: } y = \sigma(x + Wx) \quad (1)$$

$$\text{ResNet [17]: } y = x + \sigma(Wx), \quad (2)$$

where $\sigma(\cdot)$ is the nonlinear activation function, W is the layer weight matrix, x is the layer input, and y is the layer output. For ease of comparison with ResNets, Eq. 2 is simplified to show only one convolutional layer. In fact, skip connections in ResNets hop over more than one convolutional layer, while in DiracNets, the identity mapping is over one single convolutional layer. Therefore, the weights and the identity mapping of the input can be folded because $x + Wx = (I + W)x$. This change requires DiracNets to widen the NN layers in the ResNets that they started with. The authors showed that their technique could be used to create models with up to 34 layers. Although it works for shallower models, DiracNets show a decrease in accuracy between 0.5% and 1.5% compared to ResNets. In contrast, SKIPREMOVER eliminates skip connections without widening the layers in the NN and does not need to make this accuracy tradeoff.

Li et al. [25] develop residual distillation (RD), which is a modified knowledge distillation framework. RD starts with a ResNet as the teacher and a plain CNN without skip connections as the student. Unlike standard knowledge distillation, RD passes the teacher’s gradients to the student during training. This differs from TAILOR because RD starts with a student model without skip connections, whereas TAILOR *gradually* modifies a model’s skip connections every few epochs during training without sharing gradients. Moreover, while RD removes all

skip connections from models evaluated on simpler datasets like CIFAR-10 and CIFAR-100 [24], it fails to remove all skip connections in its ImageNet evaluation, leaving 18% of them in the network, which is a costly choice. In our ImageNet evaluation (see Sec. 4.1), our SKIPREMOVER method removes all skip connections with minimal accuracy loss.

Ding et al. [8] introduce a new model architecture RepVGG, which trains using 3×3 convolutional layers that are each skipped over by both a 1×1 convolution and an identity connection. At inference time, these connections can be re-parameterized into the parameters of the 3×3 convolutional layers. While RepVGG is more accurate than our SKIPREMOVER model, it requires starting from their specialized training model architecture. This is costly to developers who have already trained a model with skip connections on their dataset. Similarly, transferring a pre-trained RepVGG model to a new dataset via transfer learning can be time-consuming given the many different methods [36, 47, 52] to evaluate. As such, TAILOR is ideal for these developers because it modifies the skip connections of an *existing* pre-trained model to be more resource-efficient with minimal to no accuracy loss. Developers can leverage the training they have already done and need not start from scratch with a brand new RepVGG architecture.

2.2 Simplifying Skip Connection Hardware

ShuffleNet [28], DiracDeltaNet [48], and DenseNet [20] simplify skip connections by making them *concatenative*, i.e., they concatenate, rather than add, the skip connection data to the output of a layer. Concatenative skip connections take advantage of the fact that spatially consecutive memory accesses are typically faster than random accesses. This concatenation and off-chip data movement is possible using a simple controller (e.g., DMA engine).

TAILOR uses two techniques to simplify the skip connection hardware. SKIPREMOVER eliminates all logic and memory needed for a skip connection, making them less expensive than concatenative skip connections. Careful retraining allows skip connection removal in smaller networks with no degradation in accuracy. For larger networks, SKIPSHORTENER shortens the additive skip connections. By reducing their lifespans, the hardware implementation requires fewer resources. SKIPSHORTENER is not necessarily simpler than ShuffleNet [28] or DiracDeltaNet [48]. However, these concatenative skip connections have only been evaluated on image classification and object detection tasks. In our work, we demonstrate our SKIPREMOVER and SKIPSHORTENER methods on multifarious NNs and classification tasks, namely image-classifying ResNets of varying depths, DNA-bascalling QuartzNet- 5×5 , and automatic-speech-recognition QuartzNet- 10×5 . With respect to DenseNet [20], SKIPSHORTENER ResNets use much less memory and bandwidth because DenseNet relies on significantly more skip connections throughout its NN. Given a NN with L layers, DenseNet needs the memory and bandwidth to execute $L(L+1)/2$ concatenative skip connections, compared with SKIPSHORTENER ResNets' mere L skip connections. With so many more skip connections, DenseNet is more expensive for hardware than SKIPSHORTENER ResNets.

Finally, all these techniques simplify skip connection hardware from the outset, building their models with modified skip connections and then training them from scratch. TAILOR differs because its hardware-aware training method *dynamically* alters the skip connections every few epochs during training, taking advantage of what the NN has learned with skip connections. Thus TAILOR allows the NN to gradually adapt to shortened skip connections (SKIPSHORTENER) or none at all (SKIPREMOVER).

3 TAILOR

Skip connections are important for training (to provide good accuracy), yet complicate implementation (requiring additional hardware resources and reducing performance). TAILOR modifies skip connections to make their

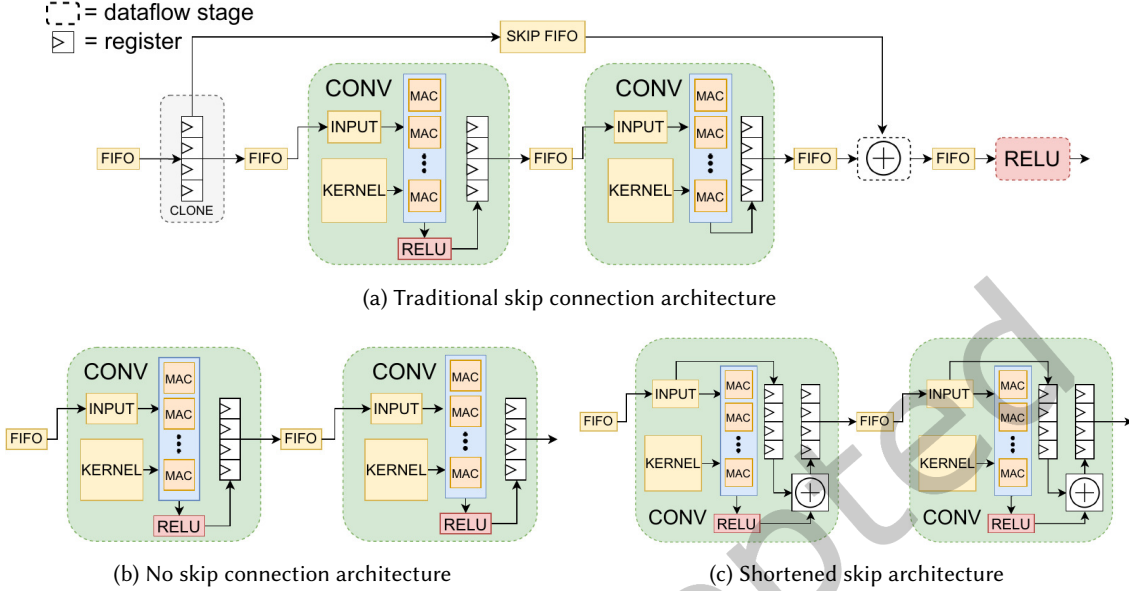


Fig. 2. The hls4ml hardware architectures for traditional, shortened, and no skip connections. hls4ml pipelines each layer as is common for latency-critical tasks in resource-constrained environments [2, 12]. The three architectures correspond to a ResNet implemented with a traditional skip connection (a), shortened skip connections (b), and no skip connections (c). Note that we combine the batch normalization parameters with the kernel, as is commonly done [21].

hardware implementation more efficient. TAILOR uses a retraining method that gradually alters the network, resulting in little to no loss in accuracy.

3.1 Hardware Design

Fig. 2 shows three hardware implementations for NNs with traditional, shortened, and no-skip connections. The implementations correspond to accelerators produced by hls4ml—a tool that translates Python models into high-level synthesis code [11]. hls4ml creates a separate datapath for each layer and performs task-level pipelining across the layers. The layers communicate using FIFOs (AXI streams). Everything encapsulated by a dashed line resides in one pipeline stage. The inputs are fed into the architecture using a stream, and the results are given as an output stream. The weights are all stored on-chip, and all the internal results are stored on-chip. We evaluate each of these designs on FPGA later in Sec. 4.2 along with another style of architecture using a 2D array of processing elements. TAILOR allows us to trade off between accuracy, performance, and resource usage through co-design of the neural network using hardware-aware training.

Fig. 2a shows the hardware needed to implement a single ResNet’s skip connection. Note that in all of the designs shown in Fig. 2, we fuse the batch normalization parameters with the kernel, as is commonly done [21]. To be low latency and high throughput, the design uses task-level pipelining (i.e., the HLS dataflow pragma) for each NN layer, or a small grouping of layers, and streams the data between each dataflow stage using first-in first-out buffers (FIFOs). Since FIFOs can only be read from once, skip connections complicate the design. We must spend a dataflow stage on cloning the skip connection data from its input FIFO into two other FIFOs so that it can be read twice for its two datapaths. The first path goes through a collection of convolutional and ReLU

layers, and the second stores the data in a FIFO exclusive to skip connections (skip FIFO). Once the data has gone through the first path, we read from the skip FIFO to perform the addition to complete the skip connection's identity function. As such, implementing a skip connection on chip requires several extra FIFOs for handling the skip connection data, and this in turn increases on-chip memory resource utilization.

Ideally, we would eliminate the skip connections. As seen in Fig. 2b, without skip connections, we cut the number of dataflow stages in half (no more Clone, Add, or ReLU stages) and use less than half of the requisite FIFOs compared with Fig. 2a. All we need to do is pass the data through the convolutional and ReLU layers. This reduces resource utilization by up to 16% (see Sec. 4.2).

It may not be possible to remove the skip connections because they are essential for training convergence. In these cases, shortening the skip connections can simplify their hardware implementation. Fig. 2c shows a modified network with shortened skip connections such that *each skip connection's lifespan resides within a single dataflow stage*. We do not need additional dataflow stages to clone skip connection data. The shorter lifespans allow the shortened skip connections to be stored in *shift registers*, which can be implemented using the more abundant FFs as opposed to BRAMs, which is used in the traditional skip connection's hardware design. In this way, we exploit the short skip connections' lifetimes and use simpler, more efficient hardware memories to implement them (see Sec. 4.2). As such, we achieve a similar architecture to the version without skip connections (Fig. 2b), and similarly reduce resources spent on additional dataflow stages and FIFOs in Fig. 2a. SKIPSHORTENER is thus more resource-efficient than the traditional skip connection design. In fact, SKIPSHORTENER provides a tradeoff between the SKIPREMOVER and traditional designs because it uses more resources than SKIPREMOVER but less than the traditional one (see Sec. 4.2). But as we later show in Sec. 4.1, SKIPSHORTENER maintains accuracy in cases where SKIPREMOVER accuracy drops off. Thus, SKIPSHORTENER allows for design space exploration to balance accuracy and resource usage.

When used with hls4ml, TAILOR reduces resource consumption without changing the performance. This is a consequence of hls4ml's dataflow design; the resources we remove are not on the critical path—they are operating in parallel to the critical path. A dataflow design uses task-level pipelining, so reducing the resources spent on stages not on the critical path does not help or hurt overall throughput. Based on our Vivado co-simulation results, the clone stage executes in microseconds while the convolutional layer executes in milliseconds, an order of magnitude difference. Therefore, removing the clone buffer (Fig. 2b) or implementing it more efficiently (Fig. 2c) will not affect the overall dataflow latency because its latency is an order of magnitude less than the convolution's latency. This means TAILOR's resource reductions do not increase or decrease latency or throughput for this architecture style, as later shown in Tab. 7.

Another prevalent style of FPGA CNN architectures instantiates a 2D processing element (PE) array and iteratively programs the convolutions and other operations onto that PE array. We call this style of computation a Reconfigurable DNN Architecture. Fig. 3 provides an example architecture used in our experiments. We build this architecture using DeepSoCFlow¹. Following the taxonomy described in [22], the reconfigurable DNN architecture is a 2D array of processing elements that optimally perform standard convolution and matrix multiplication with high data reuse. The dataflow is primarily output stationary while prioritizing maximal weight reuse and also reusing inputs to an extent. The engine performs fixed-point computations, where the input, weight, and output bit widths are adjustable as synthesis parameters, along with the number of rows and columns of processing elements. The weights rotator prefetches the weights of the next iteration into one block of on-chip memory while the other bank delivers weights, rotating them hundreds of times for maximal data reuse. The Pixel Shifter shifts perform vertical convolution. Partial sums are shifted to the PE on the right to compute horizontal convolution. The results are streamed out through the output DMA to the off-chip memory. The runtime controller would perform the residual addition, quantization, and activation on the processing system

¹<https://github.com/abarajithan11/deepsocflow>

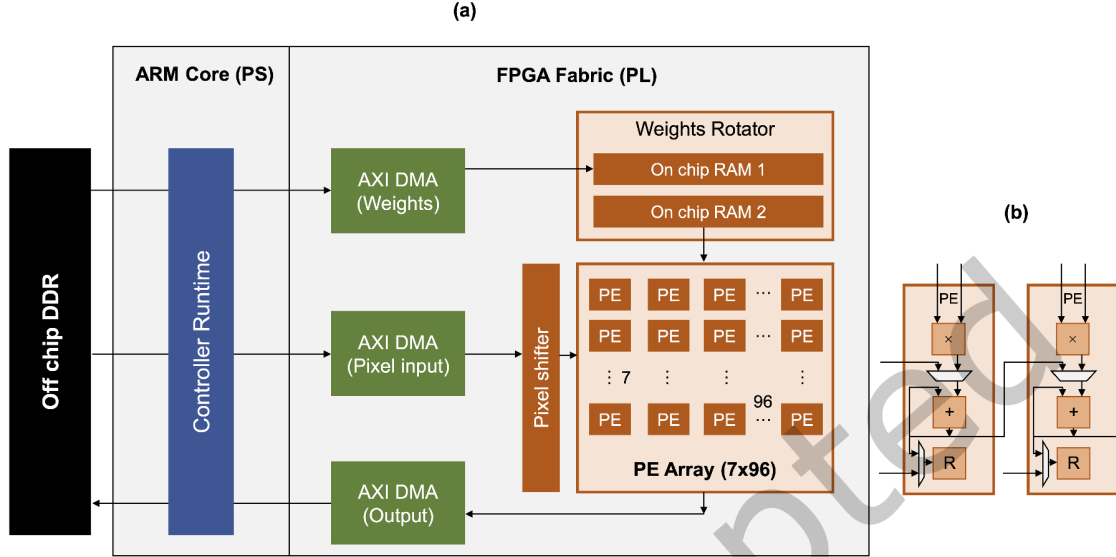


Fig. 3. (a) A Reconfigurable DNN Architecture synthesized on a ZCU102 FPGA development board. The architecture has a 2D array of processing elements that are iteratively programmed to compute layer operations. The controller runtime programs the DMA engines to load off-chip inputs and weights and store the intermediate and final results off-chip. (b) The processing elements (PE) are a multiply-accumulate datapath.

side while the engine computes the next iteration. Our implementation uses the ARM processor available in the Zynq chip. FPGAs without processors could instantiate a softcore processor to perform the controller runtime operations.

The TAILOR optimizations have different effects on the Reconfigurable DNN architecture as compared to hls4ml architecture. The Reconfigurable DNN architecture computes skip connections by loading input data from off-chip memory and performing the required operations upon it (addition, convolution) Thus, unlike in hls4ml, removing a skip connection does not change the architecture; instead it changes how computations are mapped to that architecture. Skip connection removal eliminates the need to fetch the skip connection data and perform the associated convolution and addition operations. This increases the overall performance as we describe in Sec. 4.

3.2 Hardware-aware Training

It is difficult to modify a NN's skip connections without reducing accuracy. Naively removing all skip connections before or after training a NN is detrimental to its accuracy. Instead, TAILOR consists of two training algorithms, *SKIPREMOVER* and *SKIPSHORTENER*, that gradually alter a NN's skip connections on the fly—removing or shortening them every few epochs—in order to make them resource-efficient. Gradually altering the model during training tempers the performance drop of removing or shortening the skip connections, yielding minimal to no loss in accuracy as well as significant advantages in the hardware implementation, as described above.

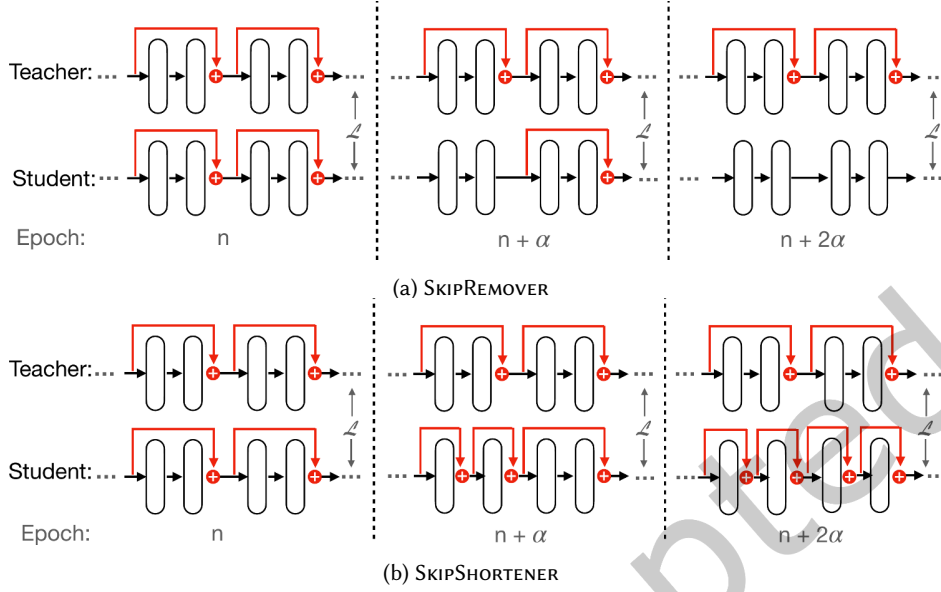


Fig. 4. Three iterations in the SKIPREMOVER and SKIPSHORTENER algorithms as applied to a ResNet. In this example, skip connections are altered every α epochs and $\alpha|n$. Each pill block represents a set of convolutional, BN, and ReLU layers, and the skip connections are in red. \mathcal{L} is the KD loss function defined in Eq. 3. Only the student model is used for inference.

TAILOR’s iterative learning approach finetunes the altered NNs using a compression method known as *knowledge distillation (KD)* [19]. KD distills the knowledge of a larger, more complex NN (the teacher) into a smaller, simpler NN (the student). While the student model is training, it compares its output to the teacher model’s output and thus learns from the teacher to perform better. KD provides impressive results for compressing NNs for various applications [31, 41, 44]. In traditional KD, the teacher model is already trained, and the student model is trained to match the teacher’s behavior by replicating its output. The student achieves this by training with a loss function

$$\mathcal{L} = (1 - \beta)\mathcal{G}(\ell, s) + \beta\mathcal{H}(t, s) \quad (3)$$

where \mathcal{G} and \mathcal{H} are distance functions, s and t are student and teacher output vectors respectively, ℓ is the correct label vector, and β is a tunable parameter [19].

With this idea in mind, both SKIPREMOVER and SKIPSHORTENER start with two identical pre-trained NNs with traditional skip connections, where one serves as the teacher and the other serves as the student. During the retraining stage, SKIPREMOVER removes a given skip connection every few epochs. SKIPSHORTENER takes a similar iterative approach and, every few epochs, splits a given skip connection into multiple shorter ones. The skip connections are removed or shortened starting from the first skip connection encountered in the NN (from the input) to the last.

Fig. 4 visualizes both SKIPREMOVER’s (Fig. 4a) and SKIPSHORTENER’s (Fig. 4b) training algorithms for a ResNet-style NN. During training, we remove (SKIPREMOVER) or shorten (SKIPSHORTENER) one of the student’s skip connections every α epochs. If n is divisible by α (as in Fig. 4), then at epoch n , the student has had n/α skip connections altered, and we are viewing the next two skip connections to be modified in the student model: the $(n/\alpha) + 1$ st and $(n/\alpha) + 2$ nd. At epoch $n + \alpha$, the $(n/\alpha) + 1$ st skip connection is altered (removed under

SKIPREMOVER or split into two shorter skip connections under SKIPSHORTENER). The NN then trains for α epochs so that the student model can improve its weights given the latest model topology. Afterwards, at epoch $n + 2\alpha$, the $(n/\alpha) + 2$ nd skip connection is similarly altered. During the entire skip modification retraining process, the student uses the KD loss function \mathcal{L} defined in Eq. 3 to learn from the teacher and the true labels. The teacher’s model topology and weights remain fixed during training. Once all skip connections have been altered, the student model continues training under KD for the remaining number of training epochs as defined by the user. Only the student model is used for inference.

TAILOR is novel because it *dynamically* transforms skip connections every few epochs during training. This is an instance of *hardware-aware training* because the skip connection are slowly altered specifically to reduce hardware resources, as previously discussed in Sec. 3.1. The gradual skip connection alterations allow the NN to take advantage of what it has learned with skip connections, so that it can dynamically adapt to shortened skip connections (SKIPSHORTENER) or none at all (SKIPREMOVER). Alg. 1 describes TAILOR’s hardware-aware training process.

Algorithm 1: HARDWARE-AWARE TRAINING

```

1 set alter // REMOVE or SHORTEN
2 let  $\alpha$  = how often to modify a skip connection
3 teacher = pre-trained model
4 student = pre-trained model
5 let current-skip = student's first skip connection from the input side
6 let current-layers = all layers skipped by current-skip
7 Function SkipRemover(current-skip):
   | // see Fig. 4a
   | remove current-skip
   | return student model's next skip connection from the input side
8 Function SkipShortener(current-skip, current-layers):
   | // see Fig. 4b
   | Split current-skip into  $\text{len}(\text{current-layers})$  skip connections
   | current-skip = student model's next skip connection from the input side
   | current-layers = student's next layers skipped by the new current-skip
   | return current-skip, current-layers
9 for i in epochs do
10 | if  $i \neq 0$  and  $i \bmod \alpha = 0$  then
11 | | if alter == REMOVE then
12 | | | current-skip = SkipRemover(current-skip)
13 | | else if alter == SHORTEN then
14 | | | current-skip, current-layers = SkipShortener(current-skip, current-layers)
15 | | end
16 | end
17 | train student using Eq. 3
18 end
19 save the student model

```

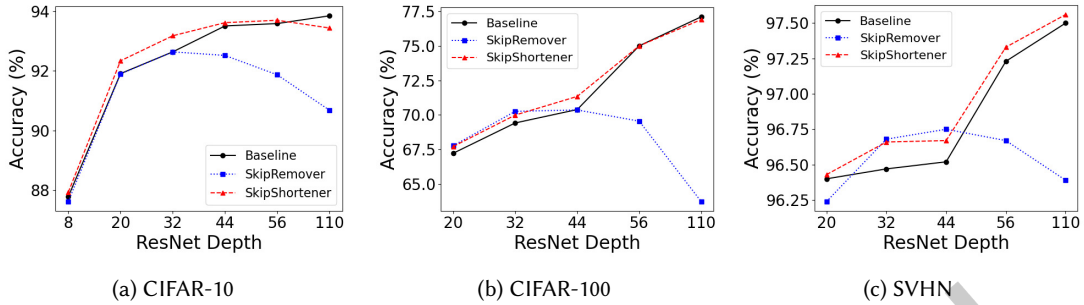


Fig. 5. Top-1 accuracy of SKIPREMOVER and SKIPSHORTENER ResNets of increasing depth on various datasets. “Baseline” refers to an unmodified ResNet with conventional skip connections.

4 RESULTS

We evaluate TAILOR on two popular kinds of NNs that rely on skip connections: ResNets [17] and QuartzNets [23]. We study the effects of TAILOR on model accuracy, quantization, and hardware resource utilization.

4.1 Training results

To evaluate how TAILOR affects a NN’s accuracy, we train ResNets and QuartzNets of varying depths using our SKIPREMOVER and SKIPSHORTENER algorithms in PyTorch [39]. The ResNets range from 20 to 110 layers and are trained on the CIFAR-10 [24], CIFAR-100 [24], and SVHN [35] datasets. We also evaluate ResNet50, which has a different skip connection topology than standard ResNets, on the ImageNet dataset [7]. The QuartzNets span between 29 and 54 layers. Their structure is determined by the number and lifetimes of their skip connections. For instance, a QuartzNet-10×5 has 10 skip connections that each have a lifetime of 5 sets of layers. We train a QuartzNet-5×5 on the Oxford Nanopore Reads dataset [42], a DNA basecalling task. We also train a QuartzNet-10×5 on the LibriSpeech dataset [37], an automatic speech recognition (ASR) task, which converts speech audio to text. ASR tasks are assessed using word error rate (WER), which measures the percent of words that the model predicted incorrectly. In all of our ResNet and QuartzNet-10×5 training experiments, we set $\alpha = 3$ in Alg. 1, so skip connections are removed or shortened every three epochs. For QuartzNet-5×5, we set $\alpha = 1$ instead because it trains better this way. For the ResNets, we set \mathcal{G} and \mathcal{H} in Eq. 3 to *categorical cross entropy* and *mean-squared error*, respectively, and set $\beta = 0.35$. For the QuartzNets, we set Eq. 3’s parameters similarly, except for \mathcal{G} , which we set to *connectionist temporal classification loss*, which is used to train difficult tasks involving sequence alignment (like DNA basecalling and ASR). Note that in our training results, “Baseline” refers to the unmodified NN counterpart with conventional skip connections.

Fig. 5 shows that SKIPREMOVER works well for ResNet-44 and smaller, at times even outperforming its baseline (traditional skip connection model). However, its accuracy drops as the number of layers increases. This indicates that shallower NNs do not need skip connections for these classification tasks, but they become more necessary for deeper networks. SKIPSHORTENER mostly outperforms the baseline on all three datasets, even on deep models.

4.1.1 Ablation Studies. We also perform *ablation studies* in which we remove key parts of TAILOR to understand why they are critical to minimizing accuracy loss. One key part of SKIPREMOVER/SKIPSHORTENER is the dynamic skip connection removal/shortening that occurs every few epochs during training under KD. We thus take away this dynamic model alteration by first altering the NNs to have either no skip connections or shortened skip connections. These pre-modified NNs are then trained under KD only. Another key part of SKIPREMOVER and

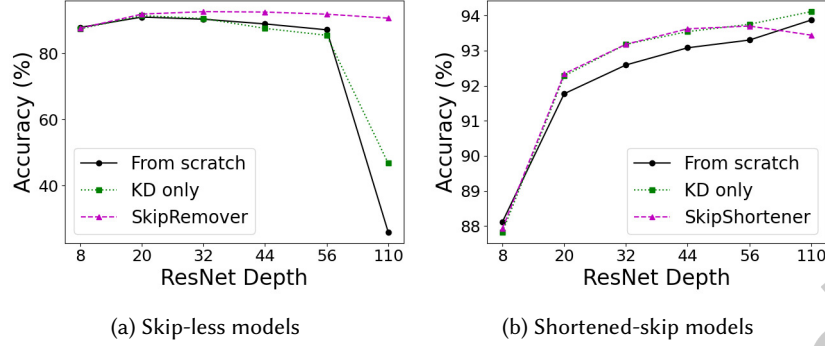


Fig. 6. Accuracy results for ResNets whose skip connections are all altered before training (apart from SKIPREMOVER and SKIPSHORTENER) on CIFAR-10. “From scratch” means training with randomly initialized weights without KD. “KD only” means training without dynamic skip alterations.

Table 1. Top-1 accuracy of ResNet-50 on the ImageNet dataset. *RD [25] only removes 82% of the skip connections.

Model	Accuracy (%)
ResNet-50	75.85
No skips (from scratch)	58.36
No skips (KD only)	69.40
Residual distillation (RD)* [25]	76.08
RepVGG-A2 [8]	76.48
SKIPREMOVER	75.36

SKIPSHORTENER is KD. We evaluate how skip-less and shortened-skip NNs perform without KD, training from randomly initialized weights (i.e., from scratch).

For ResNets trained on CIFAR-10, SKIPREMOVER and SKIPSHORTENER usually yield better results than either normal training or using KD-only on a statically pre-modified network on CIFAR-10 per Fig. 6a and Fig. 6b. The difference between all of the approaches in the figures is minimal for smaller models, but it becomes more apparent as NN depth increases. For instance, skip-less ResNet-110 under regular training yields an accuracy of 26.02% versus SKIPREMOVER, which achieves an accuracy of 90.68%, a 64.66% difference. SKIPREMOVER marginally outperforms regular training and KD-only on smaller skip-less models, but performs much better in comparison as the networks deepen. SKIPSHORTENER also generally performs better than the other two approaches for shortened skip models. Regular training mostly lags behind both KD and SKIPSHORTENER for shortened skip models.

For ResNet-50 on ImageNet, we only apply SKIPREMOVER because it uses an irregular skip connection architecture known as a “bottleneck block” to reduce the number of parameters [17]. This block has a skip connection spanning three layers: a 1×1 convolution, then a 3×3 convolution, then another 1×1 convolution (Fig. 7a). This irregular topology is not optimal for SKIPSHORTENER because it requires the majority of the shortened skip connections to pass through extra downsampling 1×1 convolutions to match the activation tensor shapes, significantly increasing the number of model parameters. As such, for ResNets with bottleneck blocks, like ResNet-50, we recommend SKIPREMOVER. As seen in Tab. 1, SKIPREMOVER incurs a 0.49% accuracy loss compared to the

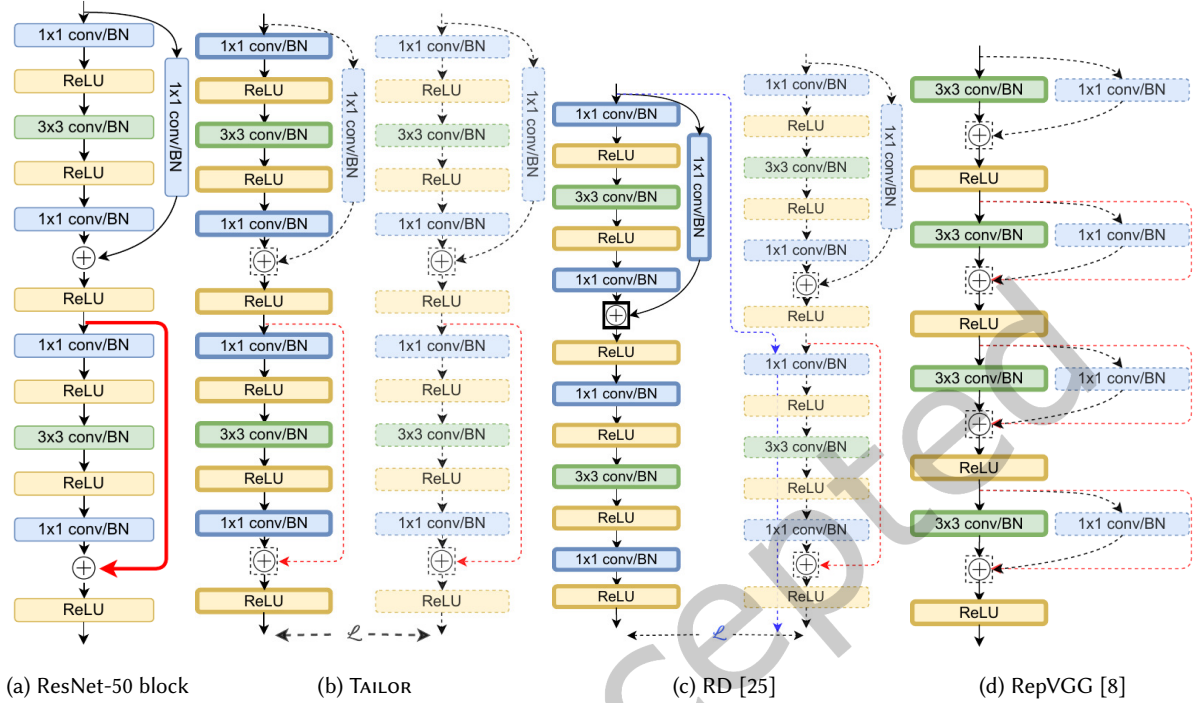


Fig. 7. Comparing TAILOR’s ResNet-50 skip removal method with residual distillation (RD) [25] and RepVGG [8]. The dashed portions are only used during training and are later removed, leaving the final inference NNs, indicated by bolder lines. Note that TAILOR (b) removes skip connections from a pretrained ResNet-50 (a). RD does the same but uses a modified KD method that does not remove the 1×1 convolution addition (c). RepVGG starts training from a different NN topology altogether (d).

traditional ResNet-50. Compared to prior work such as RD [25] and RepVGG [8], SKIPREMOVER has slightly lower accuracy (at most 1.12% accuracy difference)².

Nevertheless, SKIPREMOVER has two advantages compared with these methods. First, SKIPREMOVER removes all skip connections from ResNet-50, whereas RD only removes 82% of them. RD does not remove the 1×1 convolution addition used for downsampling (see Fig. 7c), which is particularly detrimental. In our experiments on hls4ml architectures, Vivado HLS estimates that ResNet-50’s large 1×1 convolution skip connection consumes as many resources as the layers it skips over, effectively doubling resource consumption for that skip connection block. Although Vivado HLS has a tendency to overestimate the actual place-and-route (P&R) resource utilization, these estimates demonstrate that performing the 1×1 convolution is a nontrivial task that significantly affects resource consumption. Second, SKIPREMOVER removes the skip connections from an *existing* pre-trained model, whereas RepVGG requires developers to adopt a new model topology (see Fig. 7d). If developers do not already have a model on hand, RepVGG is a better option. However, if developers already have a ResNet trained for their specific dataset, it is advantageous to use SKIPREMOVER if they can afford a small accuracy loss. This prevents starting from scratch with RepVGG, which could require extensive hyperparameter tuning. Even finetuning a pre-trained RepVGG model to a new dataset using transfer learning is time consuming, as it is unclear which of

²Ding et al [8] introduce RepVGG models of varying depths. We compare against RepVGG-A2 because it is about the same size as ResNet-50.

Table 2. Top-1 accuracy of QuartzNet-5×5 on the Oxford Nanopore Reads dataset.

Model	Accuracy (%)
QuartzNet-5×5	95.107
No skips (from scratch)	94.475
No skips (KD only)	94.863
SKIPREMOVER	95.086
Shortened skips (from scratch)	95.019
Shortened skips (KD only)	95.016
SKIPSHORTENER	94.902

Table 3. Word error rate (WER) of QuartzNet-10×5 on LibriSpeech dataset. This includes clear (“dev-clean”) and noisy (“dev-other”) audio samples. “—” indicates the model failed to converge.

Model	dev-clean WER (%)	dev-other WER (%)
QuartzNet-10×5	5.56	16.63
No skips (from scratch)	—	—
No skips (KD only)	—	—
SKIPREMOVER	—	—
Shortened skips (from scratch)	6.40	17.68
Shortened skips (KD only)	7.14	19.95
SKIPSHORTENER	7.86	21.16

the many methods [36, 47, 52] would work best. Instead, SKIPREMOVER allows developers to take advantage of their existing work and achieve a more resource-efficient model.

For QuartzNet-5×5, the SKIPREMOVER model performs the best—only 0.021% from the baseline (Tab. 2). These results all have high accuracy likely because DNA basecalling is an easier sequence alignment task (only four classes) and the model is more than sufficient. For a harder ASR task like LibriSpeech, QuartzNet-10×5 fails to converge without skip connections. Since the model must translate audio samples to text, the audio samples can be noisy, making ASR harder. LibriSpeech, in fact, divides its test samples into “dev-clean” for clearly spoken samples and “dev-other” for noisy samples. With such a challenging task, it is not possible to remove the skip connections (like with DNA basecalling). Nonetheless, QuartzNet-10×5 performs well under SKIPSHORTENER, as it is within 2% of the baseline WER (Tab. 3). For both QuartzNet-5×5 and -10×5, the best performing shortened skip connection model was one whose skip connections were shortened first and then trained from scratch. While SKIPSHORTENER has minimal accuracy loss for both QuartzNets, we recommend training a model with shortened skip connections from scratch for this task.

Overall, SKIPREMOVER and SKIPSHORTENER perform better than either training on randomly initialized weights or training with KD only. For harder tasks like ASR though, training a shortened-skip model from scratch is a better choice. Nevertheless, the success of SKIPREMOVER and SKIPSHORTENER lies in augmenting KD with dynamic skip alterations.

4.2 Hardware Results

We first quantize ResNets ranging from 20 to 56 layers deep to see how TAILOR’s accuracy fares under reduced precision. We then evaluate TAILOR’s effects on hardware resources and latency by performing a case study on ResNet-20-style skip connections implemented using the hls4ml architecture, i.e., the designs illustrated in Fig. 2.

We select this style of skip connection because it is the fundamental building block of ResNets that range from 20 to 110 layers. In our case study, we vary the bit precision and number of filters to see how TAILOR scales up. Based on how TAILOR’s resource reductions scale, designers can understand how TAILOR extrapolates to their own hardware designs. We report latency as well as P&R resource results on the Alveo U200 FPGA accelerator card (part no. xcu200-fsgd2104-2-e). For end-to-end application results, we evaluate the benefits of TAILOR on two different styles of CNN architectures. The first uses the hls4ml tool to generate architectures. The second is the Reconfigurable DNN Engine—a 2D array of processing elements. Both styles of architectures are described in Sec. 3.1.

4.2.1 Quantization. The parameters of a hardware-accelerated NN are typically quantized from floating-point to fixed-point precision [6, 33, 48].

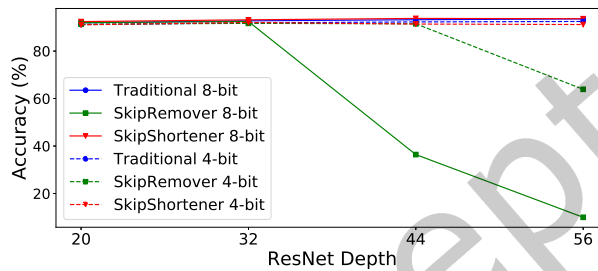


Fig. 8. Quantized accuracy results for 8-bit and 4-bit fixed point using Brevitas.

Quantizing deep NNs with minimal accuracy loss is a largely manual and time-consuming task [14]. We use Brevitas [38] to quantize our SKIPREMOVER and SKIPSHORTENER ResNets with depths of 20 to 56 from 32-bit floating-point (float32) to 8-bit and 4-bit fixed-point precision on the CIFAR-10 dataset. We modified TAILOR’s hardware-aware training algorithm where the teacher continues to use floating-point representation whereas the student is quantized. This results in the student undergoing quantization-aware training. In Fig. 8, we find that SKIPSHORTENER ResNets consistently outperform traditional ResNets under Brevitas quantization-aware training by 0.5%. SKIPREMOVER ResNets start to suffer from the lack of bits as they get deeper, with accuracy dropping to random classification for ResNet-56. But, Brevitas is only one of dozens of ways to quantize neural networks [9, 10, 14, 33, 46], so it may be the case that a SKIPREMOVER ResNet-56 requires a different method of quantization to achieve a quantized accuracy similar to its float32 counterpart.

4.2.2 FPGA Evaluation. Our first study looks solely at one ResNet block. The second study performs an end-to-end implementation of ResNet8 and ResNet50.

For our case study on a ResNet skip connection blocks (see designs in Fig. 2), we evaluate TAILOR at `ap_fixed<8, 3>` and `ap_fixed<16, 6>` precisions using the hls4ml architecture. Under both bitwidths, we increase the number of filters for all designs from 16 to 32 to 64. This way, we can understand how TAILOR scales with the number of filters. We use hls4ml [12] to translate these hardware designs into Vivado HLS, targeting the Alveo U200 FPGA accelerator card. hls4ml uses task-level pipelining (i.e., HLS dataflow) for each NN layer, or small group of layers and streams data between dataflow stages using FIFOs. hls4ml also exposes a knob known as *reuse factor*, which determines how often multipliers are reused in a design. To fairly compare our designs as the number of filters increases, we fix the reuse factor to 576. We then synthesize our designs to report P&R resource utilization as well as co-simulation latency results. Lastly, we run the designs on the U200 to verify correctness.

Table 4. Place-and-route resource utilization of a skip connection block as the number of filters increases for $\langle 8, 3 \rangle$ precision on an Alveo U200. SKIPREMOVER reduces LUT and FF usage, whereas SKIPSHORTENER trades an increase in FFs for a decrease in LUTs. T = Traditional, R = SKIPREMOVER, S = SKIPSHORTENER.

# filters	LUT			T	FF		DSP T/R/S	BRAM T/R/S
	T	R	S		R	S		
16	9,984	8,482	9,764	8,654	7,841	8,916	0	18.5
32	19,566	16,512	18,993	16,183	14,506	16,489	0	36.5
64	42,688	36,882	42,121	31,124	27,815	31,850	0	82

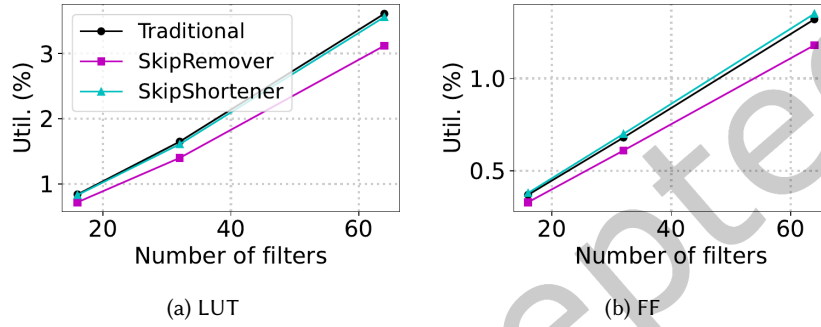


Fig. 9. Percent resource utilization of a $\langle 8, 3 \rangle$ skip connection block at various filter sizes on an Alveo U200. DSPs and BRAMs remain the same across the three designs, so they are not shown. SKIPREMOVER and SKIPSHORTENER LUT and FF reductions scale linearly, as expected.

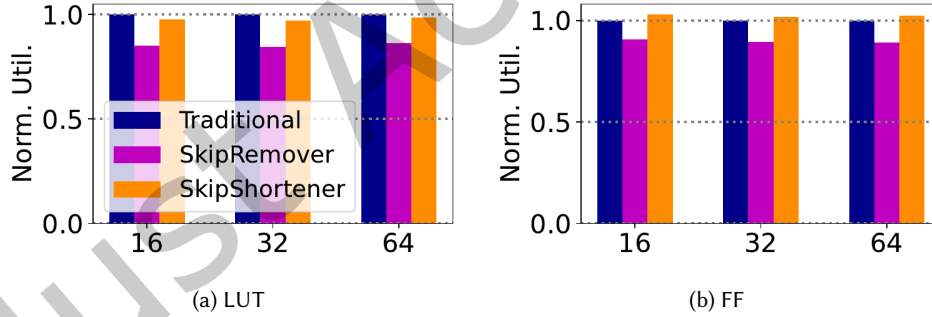


Fig. 10. Resource utilization normalized to the traditional design of a $\langle 8, 3 \rangle$ skip connection block at various filter sizes. DSPs and BRAMs remain the same across the three designs, so they are not shown. SKIPREMOVER and SKIPSHORTENER LUT and FF reductions scale proportionally, as expected.

Under 8-bit precision, we find that both SKIPREMOVER and SKIPSHORTENER reduce resources. Tab. 4 summarizes our P&R results. Since our model uses 8-bit precision, we see that all of our models exhibit low DSP usage and higher LUT and FF utilization. This is because Vivado HLS maps multiplications on datatypes that are less than 10 bits to LUTs instead of DSPs, as noted by [2, 48]. It is possible to pack two 8-bit weights into a DSP [13], but this is out of scope and orthogonal to the effects TAILOR has on hardware. Furthermore, all of the traditional and TAILOR designs use the same amount of BRAMs with respect to the number of filters because here the BRAMs are

Table 5. Place-and-route resource utilization of a skip connection block as the number of filters increases for (16, 6) precision on an Alveo U200. SKIPREMOVER reduces resources across the board, whereas SKIPSHORTENER trades an increase in LUTs for a decrease in FFs and BRAMs. T = Traditional, R = SKIPREMOVER, S = SKIPSHORTENER.

# filters	LUT			FF			DSP T/R/S	BRAM		
	T	R	S	T	R	S		T	R	S
16	14,733	13,320	14,933	17,044	14,935	16,438	12	60.5	52.5	42.5
32	28,498	25,330	28,184	32,923	28,747	31,764	48	124	108	84.5
64	55,699	50,074	55,720	64,564	56,263	62,252	192	267.5	235.5	203.5

used solely for on-chip weight storage, which does not differ across design. Nonetheless, SKIPREMOVER decreases LUT usage by up to 16% and FF usage by up to 11% compared with the traditional design (Fig. 10). These resource savings represent the extra hardware needed to implement a skip connection and subsequently the resources saved. As previously mentioned in Sec. 3.1, the extra dataflow stages that carry out a skip connection are no longer necessary. More importantly, SKIPREMOVER’s savings scale linearly as the number of filters increases from 16 to 64 (Fig. 9). SKIPSHORTENER’s resource reductions present a tradeoff, increasing FFs by 2% in exchange for decreasing LUTs by 3% (Fig. 10). SKIPSHORTENER lowers LUT utilization because the lifespan of each skip connection lasts only one dataflow stage instead of the traditional two. This means we need not spend extra logic on the dataflow stages needed to copy the skip connections to buffers that last longer than one stage. However, since the shortened skip connection now fully resides in a single dataflow stage (previously described in Fig. 2c), this requires some extra FFs. This represents the tradeoff SKIPSHORTENER provides at 8-bit precision: some extra FFs for fewer LUTs. These resource tradeoffs also scale linearly as the number of filters scales up, as seen in Fig. 9.

We find more dramatic resource reductions when we look at our 16-bit designs. Tab. 5 summarizes our P&R results. In contrast with our 8-bit designs, at higher precision, our designs rely more on DSPs and BRAMs. This time the BRAMs are used not only to store weights on chip but also to implement the FIFOs that connect the dataflow stages. Therefore, as we tailor the dataflow stages according to each design (e.g., SKIPREMOVER or SKIPSHORTENER), the BRAMs now also reflect these changes. At its best, SKIPREMOVER lowers LUTs by 11%, FFs by 13%, and BRAMs by 13%. Without a skip connection to implement, SKIPREMOVER uses fewer resources than the traditional design. The DSPs remains unchanged because they are used solely for the convolutional layers’ multiplications and not the skip connection, which is also the case for SKIPSHORTENER.

Similar to the 8-bit designs, SKIPSHORTENER presents a resource tradeoff—this time trading a small increase in LUTs (at most 1%) for decreases in FFs and BRAMs. In the best case, SKIPSHORTENER reduces LUTs by 1%, FFs by 4%, and BRAMs by 34%. While SKIPSHORTENER uses fewer LUTs than the traditional case for 32 filters, SKIPSHORTENER pays about a 1% increase in LUTs for 16 and 64 filters in exchange for decreases in FFs and BRAMs. This small disparity is likely an artifact of the heuristics Vivado P&R uses to allocate resources. Again, these resource tradeoffs and savings are possible because the shortened skip connections can be implemented within a single dataflow stage due to its reduced lifetime. Tab. 6 shows that the lifetime of each shortened skip connection is a little less than half the lifetime of the traditional one. With shorter lifetimes, we find that the SKIPSHORTENER’s skip connections’ FIFOs can now be implemented using shift registers instead of BRAMs, which is what the traditional design still uses (Tab. 6). Shift registers are much more efficient memories compared to BRAMs. As such, it is advantageous to hardware designers to consider how SKIPSHORTENER provides opportunity to implement skip connections with a more efficient memory architecture like shift registers. This leads to 30–34% fewer BRAMs than the traditional design, even as the number of filters scales up. While in this case SKIPSHORTENER uses fewer BRAMs than SKIPREMOVER does, SKIPSHORTENER offsets this difference by using more FFs than SKIPREMOVER does. For both SKIPREMOVER and SKIPSHORTENER, resource utilization (and the associated reductions) scale linearly, as seen in Fig. 11.

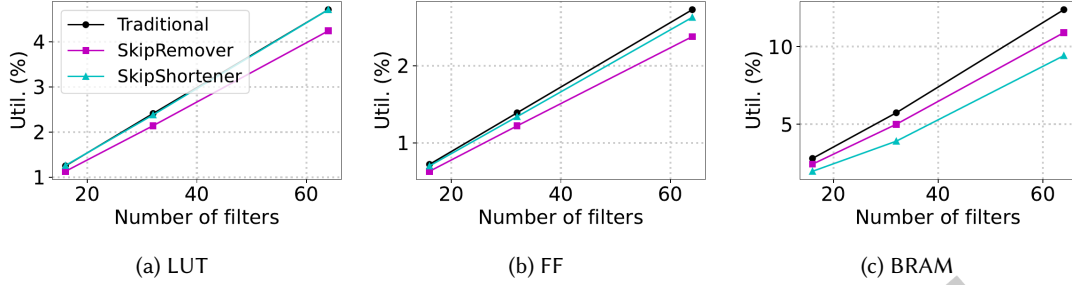


Fig. 11. Percent resource utilization of a $\langle 16, 6 \rangle$ skip connection block at various filter sizes on an Alveo U200. SKIPREMOVER and SKIPSHORTENER resource reductions scale linearly, as expected.

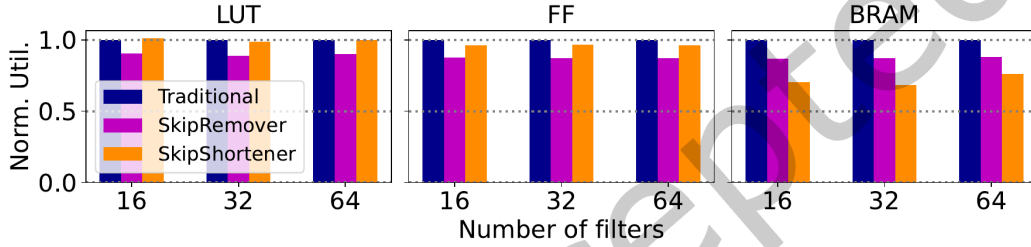


Fig. 12. Resource utilization normalized to the traditional design of a $\langle 16, 6 \rangle$ skip connection block at various filter sizes. The SKIPREMOVER and SKIPSHORTENER resource savings scale proportionally as the number of filters scales up.

Table 6. FIFO depths of a single skip connection hardware design at 16-bit precision. SKIPREMOVER has no skip connections, so it has no skip connection FIFOs.

Hardware Design	FIFO Depth	FIFO Implementation
Traditional	69	BRAM
SKIPREMOVER	0	—
SKIPSHORTENER 1st skip	33	Shift Register
SKIPSHORTENER 2nd skip	34	Shift Register

Table 7. Latency co-simulation results of a skip connection block at $\langle 8, 3 \rangle$ and $\langle 16, 6 \rangle$ precision. The latency for the Traditional, SKIPREMOVER, and SKIPSHORTENER designs are the same for each number of filters because they all rely on task-level pipelining that reuses multipliers at the same rate (576 \times).

# filters	Latency (ms)	
	Traditional/SKIPREMOVER/SKIPSHORTENER	
16	23.38	
32	23.05	
64	22.39	

TAILOR does not affect latency for hls4ml architectures. As seen in Tab. 7, for each number of filters, all designs exhibit the same latency, according to co-simulation on an Alveo U200. The slight decrease in latency

as the number of filters scales is due to an increase in DSPs and a higher degree of parallelism. As discussed in Sec. 3.1, hls4ml designs pipeline their tasks. The convolutions’ multiplication tasks dominate the overall dataflow latency. The tasks that SKIPREMOVER eliminates and SKIPSHORTENER implements more efficiently, namely the skip connection cloning and addition stages, have significantly lower latency than the convolutions and are thus not on the critical path. The throughput thus remains the same.

By shortening skip connections, we reduce their lifespans, which provides an opportunity for simplifying their hardware implementation specifically for hls4ml architectures. However, shortening skip connections is not beneficial for all architectures. As seen in Tab. 8, shortening skip connections is worse for both GPU and CPU because doing so increases off-chip memory accesses. These extra accesses lower throughput by 5% on GPU and 2% on CPU. On FPGAs with hls4ml architectures, however, we can modify the architecture to take advantage of shortened skip connections, reducing resource consumption without negatively affecting throughput (Tab. 8).

Table 8. Normalized throughput of a ResNet20. The GPU and CPU both were run with batch size = 64, whereas FPGA was run with batch size = 1. Throughput is normalized column-wise to the top entry. GPU = 1080Ti. CPU = AMD Ryzen 9 5900X. FPGA = Alveo U200. SKIPREMOVER increases GPU and CPU throughput because it decreases off-chip memory accesses. SKIPSHORTENER, however, decreases GPU and CPU throughput because it increases off-chip memory accesses. For a fully on-chip, dataflowed FPGA architecture, neither SKIPREMOVER nor SKIPSHORTENER have any effect on throughput.

Model	GPU	CPU	FPGA
Traditional skip connections	1×	1×	1×
SKIPREMOVER	1.11×	1.03×	1×
SKIPSHORTENER	0.95×	0.98×	1×

We performed two studies to understand how TAILOR performs for end-to-end implementations of ResNet models. The first is ResNet8 from MLPerf Tiny that was designed in hls4ml [3, 5]. The second is ResNet50 implemented on the Reconfigurable DNN architecture.

The ResNet8 model targets the Alveo U200. It uses 16-bit fixed-point representation with six integer bits. The reuse factor for the layers was hand-tuned to 72, which directly affects the resource usage and latency of the layers. The reuse factor is one of the more important knobs for design space exploration in hls4ml and is often hand-tuned to maximize resource usage of the platform while optimizing the overall network performance.

Table 9. MLPerf Tiny ResNet8 model implemented using hls4ml with skip connection, with shortened skip connections, and without skip connections.

	With Skip Connections	Shortened Skip Connections	Without Skip Connections
Accuracy (%)	87.39	87.93	87.62
LUTs	158609	165699	144206
FFs	196012	204914	181768
DSP48s	1083	1083	1043
BRAMs	173	158.5	156

Tab. 9 shows the resource usage results for the ResNet8 model with skip connections, with shortened skip connections, and without skip connections. Removing the skip connections has clear benefits across all the resources. Shortening the skip connections reduces BRAMs while increasing LUTs and FFs. Both the shortened skip connections and the removed skip connections models show improved accuracy over traditional skip connections. In all cases, the latency remains the same, requiring 304,697 cycles running at 100 MHz (approximately 3ms/inference).

Our second full model case study implemented a Reconfigurable DNN architecture on the ZCU102 development board which contains a Zynq UltraScale+ MPSoC. The Reconfigurable DNN array is configured to have 7 rows \times 96 columns for a total of 672 of processing elements (PEs) that support 8-bit inputs and 8-bit weights. Each PE contains a multiplier and an accumulator implemented using DSPs on FPGA fabric. Input pixels and weights are streamed into the engine as AXI-Stream packets. Images are processed in batches of 7, to increase the reuse and reduce memory accesses. The Reconfigurable DNN architecture was synthesized, placed & routed at a clock frequency of 250 MHz on a ZCU102. The architecture with $7 \times 96 = 672$ PEs used 49057 LUTs (18%), 81446 flip flops (15%), 114 BRAMs (13%), and 1344 DSPs (53%) on the FPGA fabric.

We implemented a ResNet50 model with and without skip connections on a 672-element Reconfigurable DNN architecture running on the ZCU102. Tab. 10 shows the performance of ResNet50. Removing the skip connections largely benefits the performance due to the removal of the 1×1 convolution blocks. Removing the skip connections also removes those layers, which no longer need to be scheduled on the PE array. The results are much better performance in terms of all metrics: approximately 30% increases in FPS and latency and approximately 45% decrease in memory accesses.

Table 10. ResNet50 performance with and without skip connections on the Reconfigurable DNN architecture. The architecture has 672 processing elements and runs on the ZCU102 development board at 250 MHz.

	With skip connections	Without skip connections
Accuracy (%)	75.85	75.36
Frames per second (FPS)	28.69	37.47
Time per image (s)	0.035	0.027
Latency (s)	0.244	0.187
Memory access per image (Mb)	140.95	92.71

5 DISCUSSION

With these results in hand, designers can now consider which accuracy versus resource tradeoffs they are willing to make during the hardware-software codesign process.

SKIPREMOVER provides minimal accuracy loss while reducing resource consumption and increasing performance—a win-win scenario. As seen in Sec. 4.1, SKIPREMOVER ResNet-50 is only 0.49% less accurate than the baseline on ImageNet. But, SKIPREMOVER is less effective on deeper NNs (such as QuartzNet-10 \times 5 and ResNet-110). In fact, QuartzNet-10 \times 5 fails to converge when trained under SKIPREMOVER. For such deep NNs trained on difficult tasks like ASR, skip connections are instrumental in training convergence [17]. By removing skip connections, we expect and see a degradation in accuracy for deeper NNs. This degradation is not as drastic for other tasks. For instance, ResNet-110 still converges when trained using SKIPREMOVER, but it is 3.72% less accurate on CIFAR-10 and 9.61% less accurate on CIFAR-100, compared to the original baseline model. We propose this tradeoff between NN size and SKIPREMOVER performance as an additional consideration during design space exploration. In response, SKIPSHORTENER is more suitable for deeper NNs when SKIPREMOVER is less effective. SKIPSHORTENER maintains accuracy comparable to its original skip connection models and reduces resource requirements by up to 34% compared to the traditional skip connection model.

Based on our hls4ml evaluation, designers can extrapolate to their own designs because, as we have shown in Fig. 9 and Fig. 11, the resource usage and savings scale linearly as the number of filters grows. We have also shown that at the higher 16-bit precision, TAILOR provides significant resource reductions, so if designers need more precision, TAILOR’s savings will follow. If they need lower 8-bit precision, SKIPREMOVER still manages to lower the 8-bit designs’ LUTs by 16% and FFs by 11%. Even SKIPSHORTENER decreases LUTs by 3% despite a

2% increase in FFs, though these smaller resource savings are offset by its overall higher accuracy performance compared with SKIPREMOVER. As a result, it is up to the designer to consider how to best apply TAILOR’s codesign methods given their accuracy and resource requirements.

5.1 Theoretical Understanding

Prior work investigated why skip connections are so helpful to ResNets. Veit et al [45] argue that ResNets behave like ensembles of smaller subnetworks that vary in depth and allow the NN to train and converge more easily. Li et al [26] and Yao et al [49] show that introducing skip connections make the NN loss landscape to be much smoother and have less nonconvexity. They show that naively removing these skip connections causes an explosion of nonconvexity in the loss landscape, which makes training significantly more difficult. We confirm these results in our ablation studies (Sec. 4.1), as accuracy indeed drops when skip connections are removed naively. With both KD and SkipRemover, we see an improvement in accuracy. Since the student is trying to mimic the teacher’s outputs, it is possible that the teacher’s outputs guide the student in such a way that prevents the loss landscape from becoming less smooth. Theoretical work from Lin et al [27] has proven that a ResNet with one-neuron hidden layers is a universal approximator. This work suggests that adding more neurons to the hidden layers creates an over-parameterized ResNet. Since stochastic gradient descent performs better in the presence of over-parameterization, having more neurons per hidden layer increases training efficiency, making it easier to converge. This work also argues that a ResNet is essentially a sparse version of a fully connected NN because the identity skip connections create simpler paths within the ResNet, which was similarly posited by Lin et al [27]. Given that CNNs and ResNets have both been proven to be universal approximators [27, 40], this implies that there exists a set of parameters for a CNN that can mimic a ResNet such that they equal the same function. It is mainly easier to find a well performing ResNet because Lin et al [27] showed that one-neuron hidden layers is sufficient for a ResNet to be a universal approximator.

5.2 Future Work

In our work, TAILOR has taken removing and shortening skip connections to their extremes: it either fully removes or fully shortens all the skip connections in a NN. It would be worthwhile to understand the accuracy versus resource utilization tradeoff under less extreme cases, e.g., removing only half of the skip connections. It would also be interesting to mix SKIPREMOVER and SKIPSHORTENER together to try and recover accuracy in the instances when SKIPREMOVER fails. These approaches may help address SKIPREMOVER’s scalability issues and strike a balance between SKIPSHORTENER’s high accuracy and SKIPREMOVER’s resource savings and performance improvements.

6 CONCLUSION

TAILOR introduces two new methods, SKIPREMOVER and SKIPSHORTENER, that alters NNs with skip connections dynamically during retraining to fit better on hardware, achieving resource-efficient inference with minimal to no loss in accuracy. With SKIPREMOVER, NNs no longer need to rely on skip connections for high accuracy during inference. With SKIPSHORTENER, we retrain NNs to use shorter skip connections with minimal to no loss in accuracy. Shortening skip connections is beneficial for hardware architectures generated by the hls4ml tool as it reduces the skip connection lifetime. We demonstrate FPGA resource consumption reductions of up to 34% for BRAMs, 13% for FFs, and 16% for LUTs. We show that TAILOR is also valuable for optimizing 2D PE array architectures. SKIPREMOVER increases performance by 30% and decreases memory bandwidth by 45%. Designers can decide which accuracy versus resource tradeoffs offered by SKIPREMOVER and SKIPSHORTENER are suitable to their design requirements. As a result, TAILOR is another tool in the hardware-software codesign toolbox for designers to use when building customized accelerators.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments and helpful suggestions. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-2038238. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also partially supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research under the “Real-time Data Reduction Codesign at the Extreme Edge for Science” Project (DE-FOA-0002501). JD was also supported by the DOE Office of Science, Office of High Energy Physics Early Career Research Program under award number DE-SC0021187 and the National Science Foundation (NSF) under award number 2117997 (<https://a3d3.ai>).

REFERENCES

- [1] 2023. Tailor. <https://github.com/oliviaweng/tailor>.
- [2] Thea Aarrestad et al. 2021. Fast convolutional neural networks on FPGAs with hls4ml. *Mach. Learn.: Sci. Technol.* 2, 4 (2021), 045015. <https://doi.org/10.1088/2632-2153/ac0ea1> arXiv:2101.05108 [cs.LG]
- [3] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. 2021. Mlperf tiny benchmark. *arXiv preprint arXiv:2106.07597* (2021).
- [4] Y. Bengio, P. Simard, and P. Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Netw.* 5, 2 (March 1994), 157. <https://doi.org/10.1109/72.279181>
- [5] Hendrik Borras et al. 2022. Open-source FPGA-ML codesign for the MLPerf Tiny Benchmark. In *Workshop on Benchmarking Machine Learning Workloads on Emerging Hardware (MLBench)*. arXiv:2206.11791 [cs.LG]
- [6] Sung-En Chang, Yanyu Li, Mengshu Sun, Runbin Shi, Hayden K-H So, Xuehai Qian, Yanzhi Wang, and Xue Lin. 2021. Mix and Match: A novel FPGA-centric deep neural network quantization framework. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 208. <https://doi.org/10.1109/HPCA51647.2021.00027> arXiv:2012.04240
- [7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. 248. <https://doi.org/10.1109/CVPR.2009.5206848>
- [8] Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. 2021. RepVGG: Making VGG-style convnets great again. In *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 13733. <https://doi.org/10.1109/CVPR46437.2021.01352> arXiv:2101.03697
- [9] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks. 33 (2020), 18518. arXiv:1911.03852 <https://proceedings.neurips.cc/paper/2020/file/d77c703536718b95308130ff2e5cf9ee-Paper.pdf>
- [10] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2019. HAWQ: Hessian aware quantization of neural networks with mixed-precision. In *Proc. IEEE/CVF International Conference on Computer Vision*. 293. arXiv:1905.03696
- [11] Javier Duarte et al. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum.* 13, 07 (2018), P07027. arXiv:1804.06913
- [12] Farah Fahim et al. 2021. hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices. In *1st TinyML Research Symposium*. arXiv:2103.05579 [cs.LG]
- [13] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. 2017. *Deep learning with INT8 optimization on Xilinx devices*. White Paper WP486. <https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8>
- [14] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. (2021). arXiv:2103.13630
- [15] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proc. 13th International Conference on Artificial Intelligence and Statistics*, Yee Whye Teh and Mike Titterton (Eds.), Vol. 9. 249. <https://proceedings.mlr.press/v9/glorot10a.html>
- [16] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep Sparse Rectifier Neural Networks. In *Proc. 14th International Conference on Artificial Intelligence and Statistics*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.), Vol. 15. 315. <http://proceedings.mlr.press/v15/glorot11a.html>
- [17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770. <https://doi.org/10.1109/CVPR.2016.90> arXiv:1512.03385
- [18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). 630. https://doi.org/10.1007/978-3-319-46493-0_38 arXiv:1603.05027

- [19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. (2015). arXiv:1503.02531
- [20] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proc. IEEE conference on computer vision and pattern recognition*. 4700. <https://doi.org/10.1109/CVPR.2017.243> arXiv:1608.06993
- [21] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proc. IEEE conference on computer vision and pattern recognition*. 2704. <https://doi.org/10.1109/CVPR.2018.00286>
- [22] Leonardo Rezende Juracy, Rafael Garibotti, Fernando Gehm Moraes, et al. 2023. From CNN to DNN Hardware Accelerators: A Survey on Design, Exploration, Simulation, and Frameworks. *Foundations and Trends® in Electronic Design Automation* 13, 4 (2023), 270–344.
- [23] Samuel Kriman, Stanislav Beliaev, Boris Ginsburg, Jocelyn Huang, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, and Yang Zhang. 2020. QuartzNet: Deep automatic speech recognition with 1D time-channel separable convolutions. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 6124. <https://doi.org/10.1109/ICASSP40776.2020.9053889>
- [24] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *Tech Report* (2009).
- [25] Guilin Li, Junlei Zhang, Yunhe Wang, Chuanjian Liu, Matthias Tan, Yunfeng Lin, Wei Zhang, Jiashi Feng, and Tong Zhang. 2020. Residual distillation: Towards portable deep neural networks without shortcuts. *Advances in Neural Information Processing Systems* 33 (2020), 8935. <https://proceedings.neurips.cc/paper/2020/file/657b96f0592803e25a4f07166fff289a-Paper.pdf>
- [26] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems* 31 (2018).
- [27] Hongzhou Lin and Stefanie Jegelka. 2018. Resnet with one-neuron hidden layers is a universal approximator. *Advances in neural information processing systems* 31 (2018).
- [28] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet v2: Practical guidelines for efficient CNN architecture design. In *Proc. European conference on computer vision (ECCV)*. 116. <https://doi.org/10.1109/ASPCON49795.2020.9276669> arXiv:1807.11164
- [29] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2018. Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA. *IEEE Trans Very Large Scale Integr. VLSI Syst.* 26, 7 (2018), 1354. <https://doi.org/10.1109/TVLSI.2018.2815603>
- [30] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J. Seo. 2017. End-to-end scalable FPGA accelerator for deep residual networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1. <https://doi.org/10.1109/ISCAS.2017.8050344>
- [31] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. 2020. Improved Knowledge Distillation via Teacher Assistant. In *AAAI*, Vol. 34. 5191. <https://doi.org/10.1609/aaai.v34i04.5963> arXiv:1902.03393
- [32] Ricardo Pio Monti, Sina Tootoonian, and Robin Cao. 2018. Avoiding Degradation in Deep Feed-Forward Networks by Phasing Out Skip-Connections. *Artificial Neural Networks and Machine Learning (ICANN)* 11141 (2018). https://doi.org/10.1007/978-3-030-01424-7_44
- [33] Bert Moons, Koen Goetschalckx, Nick Van Berckelaer, and Marian Verhelst. 2017. Minimum Energy Quantized Neural Networks. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 1921. <https://doi.org/10.1109/ACSSC.2017.8335699> arXiv:1711.00215 [cs.NE]
- [34] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. 27th International Conference on Machine Learning*. 807. <https://icml.cc/Conferences/2010/papers/432.pdf>
- [35] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. 2011. Reading Digits in Natural Images with Unsupervised Feature Learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning* (2011).
- [36] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng.* 22, 10 (2009), 1345. <https://doi.org/10.1109/TKDE.2009.191>
- [37] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. LibriSpeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5206. <https://doi.org/10.1109/ICASSP.2015.7178964>
- [38] Alessandro Pappalardo. 2022. *Xilinx/brevitas*. <https://doi.org/10.5281/zenodo.3333552>
- [39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Vol. 32. 8024. arXiv:1912.01703 <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [40] Philipp Petersen and Felix Voigtlaender. 2020. Equivalence of approximation by convolutional neural networks and fully-connected networks. *Proc. Amer. Math. Soc.* 148, 4 (2020), 1567–1581.
- [41] Bharat Bhusan Sau and Vineeth N. Balasubramanian. 2016. Deep Model Compression: Distilling Knowledge from Noisy Teachers. (2016). arXiv:1610.09650
- [42] Jordi Silvestre-Ryan and Ian Holmes. 2021. Pair consensus decoding improves accuracy of neural network basecallers for nanopore sequencing. *Genome Biol.* 22 (2021), 38. <https://doi.org/10.1186/s13059-020-02255-1>
- [43] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and

- Yann LeCun (Eds.). <http://arxiv.org/abs/1409.1556>
- [44] Antti Tarvainen and Harri Valpola. 2017. Mean Teachers Are Better Role Models: Weight-Averaged Consistency Targets Improve Semi-Supervised Deep Learning Results. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. 1195. arXiv:1703.01780 <https://proceedings.neurips.cc/paper/2017/file/68053af2923e00204c3ca7c6a3150cf7-Paper.pdf>
 - [45] Andreas Veit, Michael J Wilber, and Serge Belongie. 2016. Residual networks behave like ensembles of relatively shallow networks. *Advances in Neural Information Processing Systems* 29 (2016). arXiv:1605.06431 <https://proceedings.neurips.cc/paper/2016/file/37bc2f75bf1bcfe8450a1a41c200364c-Paper.pdf>
 - [46] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-aware automated quantization with mixed precision. In *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8612. <https://doi.org/10.1109/CVPR.2019.00881> arXiv:1811.08886
 - [47] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *J. Big Data* 3, 1 (2016), 1. <https://doi.org/10.1186/s40537-016-0043-6>
 - [48] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzyniek, et al. 2019. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs. In *Proc. 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 23. <https://doi.org/10.1145/3289602.3293902>
 - [49] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W Mahoney. 2020. Pyhessian: Neural networks through the lens of the hessian. In *2020 IEEE international conference on big data (Big data)*. IEEE, 581–590.
 - [50] Sergey Zagoruyko and Nikos Komodakis. 2017. DiracNets: Training Very Deep Neural Networks Without Skip-Connections. (2017). arXiv:1706.00388
 - [51] Sergey Zagoruyko and Nikos Komodakis. 2018. DiracNets: Training Very Deep Neural Networks Without Skip-Connections. (2018). arXiv:1706.00388
 - [52] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43. <https://doi.org/10.1109/JPROC.2020.3004555>

OPEN ACCESS

Fast inference of Boosted Decision Trees in FPGAs for particle physics

To cite this article: S. Summers *et al* 2020 *JINST* **15** P05026

View the [article online](#) for updates and enhancements.

You may also like

- [Nanosecond machine learning regression with deep boosted decision trees in FPGA for high energy physics](#)
B.T. Carlson, Q. Bayer, T.M. Hong et al.
- [CLASSIFICATION AND RANKING OF FERMI/LAT GAMMA-RAY SOURCES FROM THE 3FGL CATALOG USING MACHINE LEARNING TECHNIQUES](#)
P. M. Saz Parkinson, H. Xu, P. L. H. Yu et al.
- [A morphological study of cluster dynamics between critical points](#)
Thibault Blanchard, Leticia F Cugliandolo and Marco Picco

Fast inference of Boosted Decision Trees in FPGAs for particle physics

S. Summers,^{a,1} G. Di Guglielmo,^b J. Duarte,^c P. Harris,^d D. Hoang,^e S. Jindariani,^f
E. Kreinar,^g V. Loncar,^{a,h} J. Ngadiuba,^a M. Pierini,^a D. Rankin,^d N. Tran^f and Z. Wuⁱ

^aCERN, Esplanade des Particules 1, Geneva 23 1211, Switzerland

^bDepartment of Computer Science, Columbia University,
500 West 120 Street, New York, NY 10027, U.S.A.

^cDepartment of Physics, University of California San Diego,
9500 Gilman Dr., La Jolla, CA 92093, U.S.A.

^dDepartment of Physics, Massachusetts Institute of Technology,
77 Massachusetts Avenue, Cambridge, MA 02139, U.S.A.

^eDepartment of Physics, Rhodes College,
2000 North Parkway, Memphis, TN 38112, U.S.A

^fFermi National Accelerator Laboratory,
Batavia, IL 60510, U.S.A.

^gHawkEye360, Herndon, VA 20170, U.S.A.

^hInstitute of Physics Belgrade, Pregrevica 118, Belgrade, Serbia

ⁱDepartment of Physics, University of Illinois at Chicago,
W. Taylor St., Chicago, IL 60607, U.S.A.

E-mail: sioni.summers@cern.ch

ABSTRACT: We describe the implementation of Boosted Decision Trees in the `hls4ml` library, which allows the translation of a trained model into FPGA firmware through an automated conversion process. Thanks to its fully on-chip implementation, `hls4ml` performs inference of Boosted Decision Tree models with extremely low latency. With a typical latency less than 100 ns, this solution is suitable for FPGA-based real-time processing, such as in the Level-1 Trigger system of a collider experiment. These developments open up prospects for physicists to deploy BDTs in FPGAs for identifying the origin of jets, better reconstructing the energies of muons, and enabling better selection of rare signal processes.

KEYWORDS: Analysis and statistical methods; Data processing methods; Digital electronic circuits; Trigger algorithms

ARXIV EPRINT: [2002.02534](https://arxiv.org/abs/2002.02534)

Corresponding author.

Contents

1	Introduction	1
2	Building Boosted Decision Trees with hls4ml	3
3	Implementation and performance	4
3.1	FPGA implementation	4
3.2	Varying the precision	5
3.3	Performance and cost	6
3.4	Resource model	8
3.5	Varying clock frequency and precision	9
4	Summary and outlook	11

1 Introduction

Starting with the work of the MiniBooNE collaboration [1, 2], Boosted Decision Trees (BDTs) have been extremely prevalent within the field of High Energy Physics (HEP) [3], used mainly for regression and classification tasks, both in event reconstruction and subsequent data analysis. In the high-profile discovery of the Higgs boson, BDTs were used to increase the sensitivity of the CMS analysis in the decay channel of the Higgs to two photons [4], and have been used significantly in further analyses of Higgs properties.

At the Large Hadron Collider (LHC) experiments, proton collisions occur at such a frequency that the full rate of data cannot be stored. With the LHC delivering collisions every 25 ns, the experiments CMS and ATLAS have to deal with tens of terabytes of data produced each second. Each experiment operates an online data reduction system, called the trigger, to filter out only a fraction of events for further analysis. Due to the extreme data rates, this processing must necessarily be extremely fast, and since the rejected events can never be recovered, the selection must be highly robust.

The CMS and ATLAS experiments deploy a two-stage trigger system, starting with the Level-1 Trigger (L1T) performing a first selection, with a second High Level Trigger (HLT) performing a more refined selection. The L1T must process each LHC event, at the full 40 MHz collision rate, and return its decision within approximately 10 μ s, the latency for which the event data can be buffered. Due to these constraints, the L1T is implemented using high speed electronics, consisting of ASICs and FPGAs on custom cards, with high-speed optical interconnects.

Recently, Deep Neural Networks (DNNs) have been investigated as an alternative to BDTs for HEP applications, due to their superior performance and the increasing availability of parallel

For an extensive discussion of use cases, see ref. [5] and references therein.

processors capable of high throughput training and inference. Despite the large amount of studies showing interesting use cases for DNN applications, the number of DNN models deployed in the central data processing of the LHC experiments during previous LHC running was very limited. This was mainly due to the lack of optimal deployment solutions that would meet the strong constraints of central processing systems (e.g., real-time event selection in the trigger systems), both in terms of latency and computing resource footprint.

Previously, we introduced the `hls4ml` library to facilitate the deployment of DNN models on L1T systems [6]. The aim of that work was to establish an automatic workflow to convert a given DNN model into an electronic circuit, evaluated on an FPGA through a fully-on-chip firmware implementation. The workflow consists of converting a given NN model into an expertly written C++ code, which is then converted to an FPGA firmware by a High Level Synthesis (HLS) tool (e.g., Xilinx Vivado HLS). In ref. [6], we demonstrated how a DNN model for jet identification at the LHC could be compressed and quantized, to run on an FPGA with 75 ns latency.

In this work, we present an extension of the `hls4ml` library to also support BDTs. As shall be seen in the following sections 2 and 3, the BDT implementation in FPGAs is capable of achieving similar performance to a DNN, with a relatively lightweight usage of device resources. The critical FPGA resource for BDTs is Look Up Tables (LUTs), whereas the availability of DSPs for multiplication is the limiting factor for DNNs. Given this, the BDT can be seen as a lightweight solution which is complementary to a DNN.

Another motivation for the introduction of BDTs is the need to support the legacy of the LHC Run II: as of today, BDTs are still the most commonly used ML algorithm for LHC experiments. For instance, the LHCb collaboration makes extensive use of BDTs (as well as neural networks) in their trigger, which runs in software only. To accelerate the computation, a binned BDT method, Bonsai BDT, is used [7].

BDTs remain a particularly appealing solution for use in the earliest processing stages at LHC experiments, thanks to their good performance with relatively low computational cost. The first use case of an ML technique in the L1T of an LHC experiment was a BDT used to perform a regression of muon p_T for the CMS L1T endcap muon trigger [8]. The technique gave a three-times reduction in rate for the trigger threshold compared to the previous approach, removing unwanted low p_T muons. An external DRAM of 1.2 GB was used as a look-up-table (LUT) to store the pre-computed BDT output for every variation in the input variables. The LUT was filled offline and queried with low latency online. The solution proposed in this paper would allow an on-chip implementation going beyond a full-LUT approach.

Other works have implemented ensembles of Decision Trees (BDTs and Random Forests) for FPGAs [9–13]. These generally target applications of FPGA accelerated inference in a combined CPU-FPGA system, where the relevant performance goals are throughput and energy consumption. Further, the use of external memories and traversal over trees by fetching nodes from memory gives these approaches flexibility and scalability. The work of [9] and [10], in particular, is designed to be scalable to very large ensembles in a way that the implementation in this paper is not. In the context of targeting LHC triggers, however, the main performance goal is of extremely low latency, and secondly to maintain a modest resource usage.

The project code can be accessed at: <https://github.com/hls-fpga-machine-learning/hls4ml/tree/bdt>.

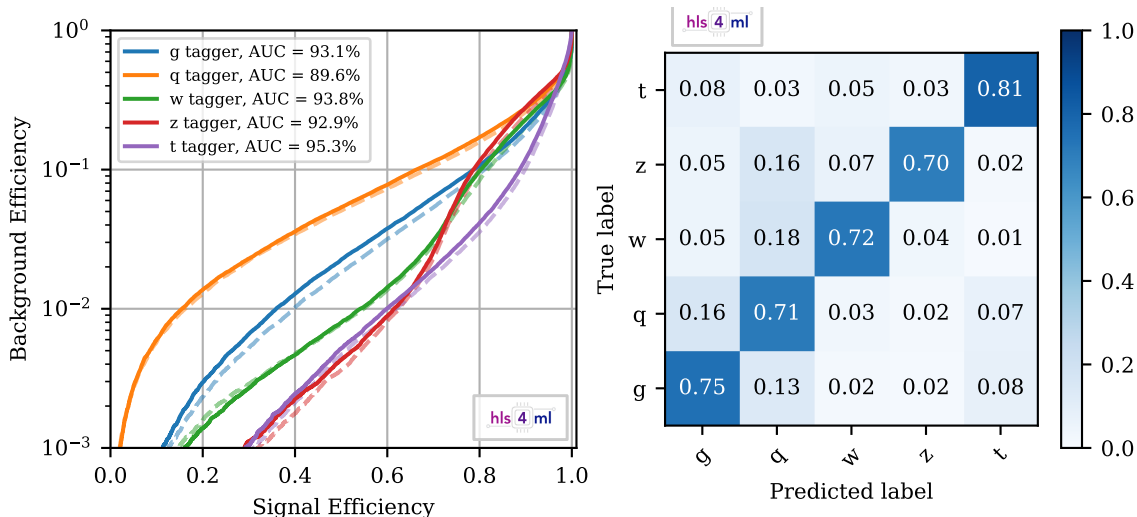


Figure 1. Left: the solid curves show signal efficiency vs. misidentification rate using a BDT with 100 trees of depth 4 for the five jet classes: gluon, quark, W boson, Z boson, and top quark. The dashed curves show the performance of the 3 layer MLP from [6]. Right: confusion matrix for the BDT.

2 Building Boosted Decision Trees with hls4ml

In the previous work on translation of neural networks to FPGA firmware with `hls4ml`, we presented a demonstration data set for discrimination of quarks (q), gluons (g), W and Z bosons, and top (t) jets [14]. The data consist of a set of 16 physics-motivated high-level features, representing information of the event jet substructure. With this information at hand, one can distinguish traditional single-prong q and g jets from two- (W and Z) and three-prong jets

This problem is typical of searches for physics beyond the standard model at ATLAS and CMS. To our knowledge there is no algorithm currently employed in the L1T systems of these two experiments that exploits this kind of *substructure* information to select events with multi-prong jets. This data set provides a benchmark on which to evaluate the classifier performance and its realisation in FPGA implementation as an example application for the L1T. We use the same data set in this work to prepare a classifier, this time a BDT.

We performed the BDT training using the `scikit-learn` package [15], randomly splitting the data set into training (80%) and testing (20%) partitions. A BDT with 100 estimators and a maximum depth of 4 was found to give similar performance to the DNN model trained on the same data set, providing a useful point of comparison. The cross-entropy loss function was used.

The resulting receiver operating characteristic (ROC) curve is shown in figure 1, displaying the background misidentification efficiency (false-positive rate) as a function of the signal efficiency (true-positive rate) for five jet selectors, defined using the five scores returned by the BDT for the five jet categories. Overall, the trained BDT reaches state-of-the-art discrimination performance, with a small performance loss with respect to the DNN model of ref. [6].

The operations used in inference of a BDT are very different from those used for a neural network. While a (fully connected) neural network comprises a series of matrix-vector products and evaluations of non-linear activation functions, the BDT inference involves evaluating decision paths

over many decision trees. This tree traversal requires comparisons against thresholds, effectively partitioning the feature space. In terms of the number of parameters, the trained BDT with 100 estimators of depth 4, and 5 classes, is summarised by 7,500 threshold values, and 8,000 scores. The fully connected neural network presented in [6], with the same 16 inputs, 5 outputs, and three hidden layers of 64, 32, and 32 neurons, has 4,389 trainable parameters. A BDT is only able to make cuts orthogonal to the feature axes, while the activation functions of a neural network add non-linearity to the classification.

We use this model as a benchmark example to show the use of `hls4ml` to derive an FPGA firmware implementation.

3 Implementation and performance

3.1 FPGA implementation

Decision Trees in `hls4ml` are implemented as an unrolled tree of decisions, as illustrated in figure 2. Each node in the tree performs a comparison of one of the input features against a constant threshold, learned in training. These thresholds are statically fixed in the logic of the FPGA firmware, rather than being fetched from an external memory. Nodes pass the results of the comparison (true or false) to their children. The decision path is then encoded by the series of Boolean values propagated along the nodes. By construction, only a single leaf node can be activated, and the index of the active leaf is used to address a small look-up-table containing the tree scores for each path. These scores map to the probability that the given input features correspond to a certain class. For multiclass classification, each ‘estimator’ uses as many decision trees as the number of classes, while only one tree per estimator would be used for a binary classification problem.

The score of the BDT ensemble is the sum of scores of all of the decision trees. Since each decision tree is independent, a high degree of parallelisation is possible in the FPGA. The sum is performed with a balanced adder tree, reducing the scores to their sum in a pair-wise tree structure. The implementation of BDTs in `hls4ml` targets low latency applications, such as LHC hardware triggers, by executing all trees, and all decisions within each tree, in parallel.

We developed two code implementations, both targeting the architecture described. The first uses Xilinx’s Vivado HLS, written in C++, and the second is developed at the Register-Transfer Level (RTL), using VHDL. Generally, an RTL implementation does not benefit from some of the features of Vivado HLS, such as automatic pipelining depending on the target clock frequency, and easy loop rolling/un-rolling. However, the RTL implementation synthesises to more reliable results for ‘large’ BDTs, as will be seen in the section 3.3. Both implementations are fully pipelined, capable of an ‘initiation interval’ of 1 clock cycle.

A trained BDT, with specific features, thresholds and scores for each tree, can be evaluated with the FPGA implementation described above using `hls4ml`. Models trained and exported from the `scikit-learn`, `xgboost` [17], and `TMVA` [18] packages are supported. From the FPGA code produced, which is either using Vivado HLS or VHDL, the user is then able to run the usual FPGA vendor workflow to integrate the BDT into a specific project and compile to a bitfile.

For an introduction to FPGA concepts and terminology, see ref. [16].

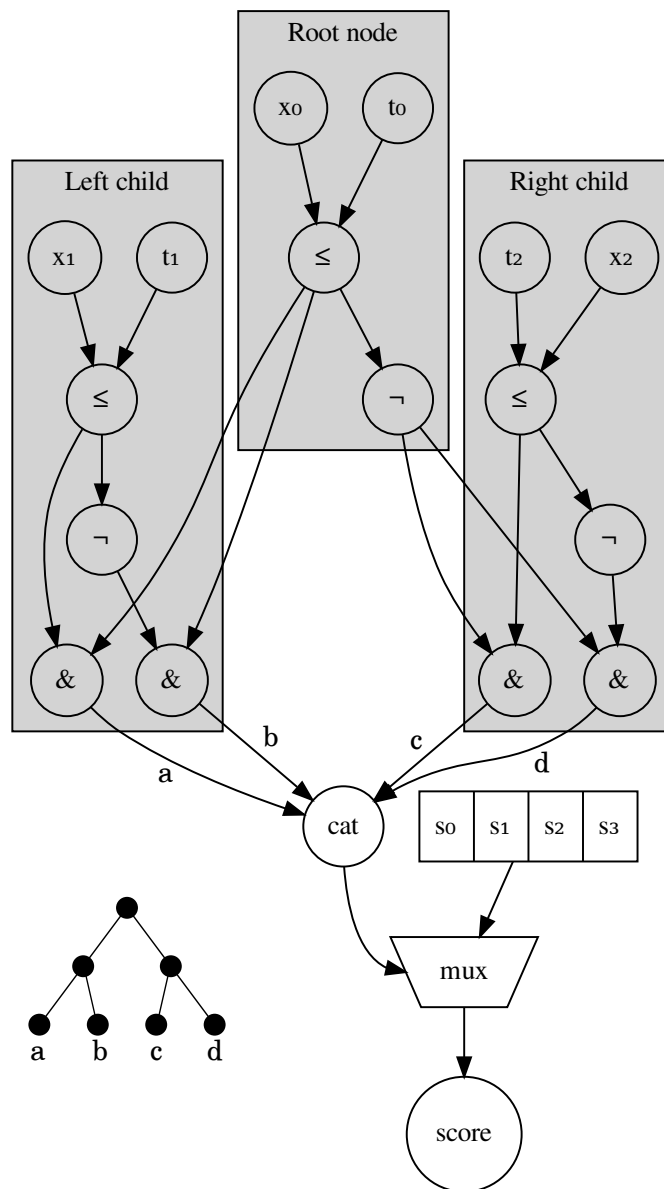


Figure 2. Schematic of the implementation of decision trees in hls4ml, showing a single tree with depth of 2. The x are the node features, and t the thresholds. The \neg is the unary ‘not’ operator, and $\&$ the binary ‘and’. The Boolean leaf activations are concatenated and used to address a look-up-table of output scores. The labels ‘a’, ‘b’, ‘c’, and ‘d’ on the schematic correspond to the respective labelled leaf nodes of the tree represented at the bottom left.

3.2 Varying the precision

The generic, programmable-logic cells in FPGAs support completely customised data representations. Floating point types are supported, but generally require more resources, latency, and achieve lower clock frequencies than integer types. The fixed point representation uses integer operations, but with a radix point in the number to represent fractional values. In the FPGA, any bitwidth and radix position may be used. A narrower bitwidth will enable smaller resource usage.

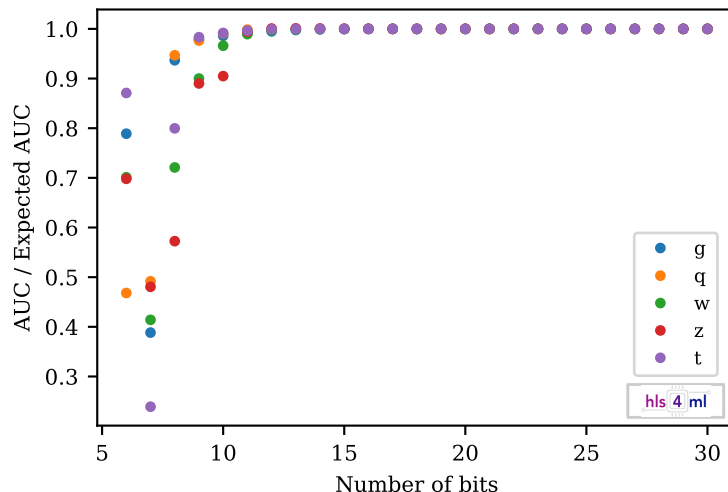


Figure 3. The ratio of Area Under the Curve (AUC) obtained from the fixed point implementation to the AUC expected from the floating point software, as a function of the fixed point bitwidth used, for each of the five tag categories. The ratio saturates at around 15 bits.

The trade-off for using narrower bitwidths is a loss of precision. The loss of discriminating power is investigated by measuring the ratio of the AUC obtained testing with fixed point representation to the area under the ROC curve (AUC) from the original floating point, and shown in figure 3 as a function of the bitwidth, for the benchmark jet-classification BDT introduced in section 2. The number of integer bits was kept at 4 for all bitwidths, as required by the range of the features and scores in the data to avoid overflow. A significant reduction in AUC is seen for bitwidths of 10 and below. The AUC with fixed point variables reaches 99% of the AUC with floating point for all taggers with 11 bits. Below 10 bits the behaviour becomes unstable, as numerical rounding effects cause unpredictable misclassification. The consequences of the bitwidth on resource usage are discussed in the next section.

3.3 Performance and cost

We studied the FPGA resource utilisation and inference latency using BDTs trained on the jet classification task described in section 2. These metrics are expected to vary with the number of trees and their depth. Other hyperparameters, while having an impact on the classification performance, do not affect these FPGA performance metrics. All HLS evaluations of BDTs were built for a Xilinx `vu9p-f1gb2104-2L-e` FPGA at 200 MHz target clock frequency. FPGAs of this size or similar could be used in future LHC upgrades, and would generally be used to execute several algorithms (including feature pre-processing) as well as any ML inference. All features, thresholds, and scores were encoded with 18 bits, which is sufficient to achieve identical classification results to the `scikit-learn` original, as was shown in section 3.2.

The resource utilisation of LUTs, FFs, DSPs, and BRAMs for the benchmark BDT with 100 estimators and a depth of 4 is shown in table 1. This utilisation is reported after running the logic synthesis step with the VHDL implementation. The inference latency for this ensemble is 12 clock cycles, corresponding to 60 ns execution time at the chosen target clock frequency. This is compatible with the requirements for use in the LIT system.

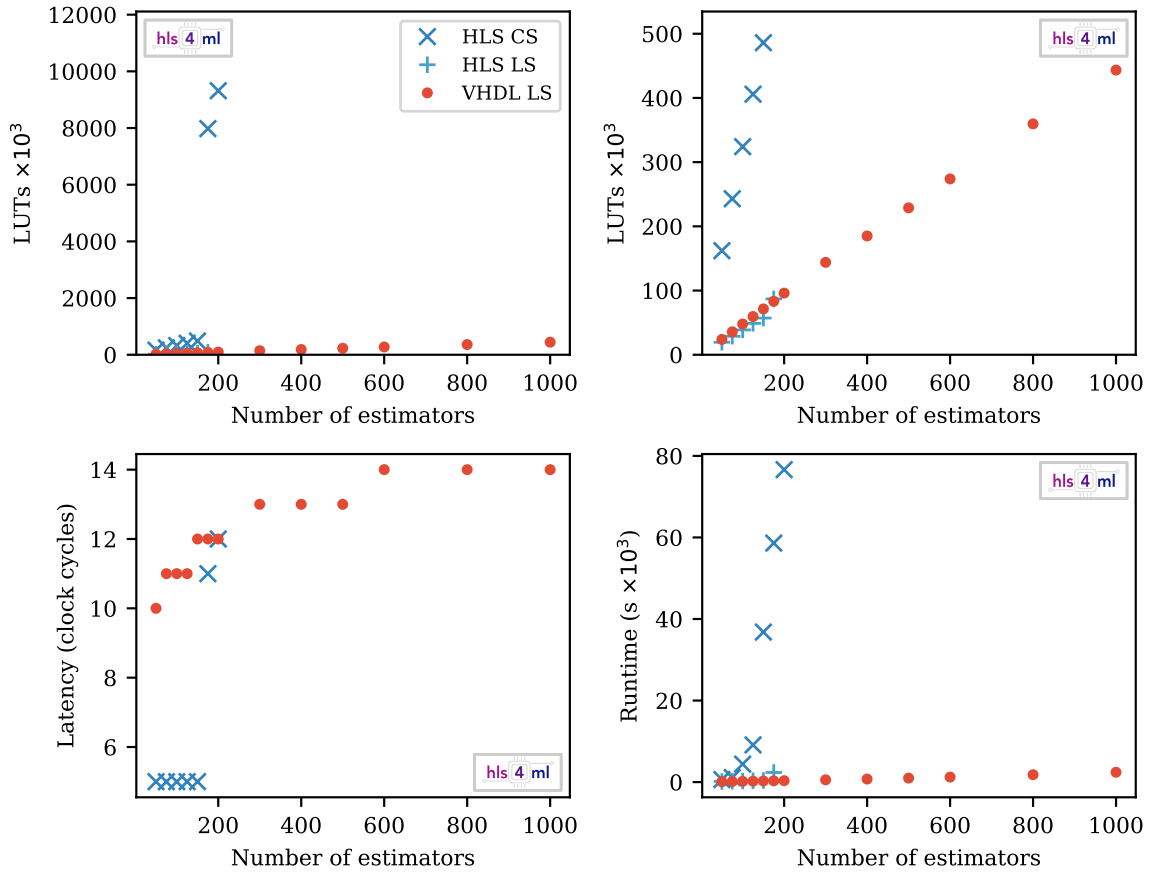


Figure 4. Dependence of LUT usage (top row), inference latency (bottom left), and synthesis time (bottom right) on the number of estimators of the BDT, with depth fixed at 3. The top right plot is a view of the same data as the top left, with reduced range. Different stages of synthesis are shown: C-Synthesis estimate of HLS (HLS CS), utilisation report after Logic Synthesis step of RTL produced by HLS (HLS LS), and utilisation report after Logic Synthesis of the VHDL implementation (VHDL LS).

Table 1. Resource usage of the BDT with 100 estimators of depth 4.

Resource	LUTs	FFs	DSPs	BRAMs
Number Used	96148	42802	0	0
Percentage of VU9P	8.1	1.8	0	0

Figure 4 shows the variation in resource usage with the number of estimators, n_e , of the BDT, with the depth fixed at 3. Each estimator uses as many trees as the number of classes, in this case of the jet classification dataset, five. Only one tree per estimator would be used for a binary classification problem, reducing the resource cost by a factor five in such cases. For the VHDL implementation, the utilisation is reported after logic synthesis with Vivado. For the HLS implementation, the Vivado HLS resource estimate after C-synthesis is reported, as well as the result after executing logic synthesis on the produced RTL with Vivado. The HLS estimate of LUT and FF usage tend to be larger than the eventual usage after the full synthesis and implementation workflow. Pipelining of the HLS implementation is determined by the Vivado HLS compiler during

the C-synthesis step, placing registers optimally to achieve timing closure. The latency of the HLS implementation is set during this step, and not affected by later execution of logic synthesis.

Up to $n_e = 150$, the LUT utilisation for both implementations increases linearly, with the HLS implementation using slightly fewer than the VHDL version (referring to the utilisation reported after Vivado synthesis). With $n_e > 150$, the LUT usage of the HLS implementation increases dramatically, and the Vivado synthesis of the produced RTL also yields poor results. In this regime, the LUT usage of the VHDL implementation continues to increase linearly with n_e .

The inference latency of the VHDL implementation increases logarithmically with n_e , as the depth of the balanced adder tree used to sum tree score increases. The HLS implementation inference latency is more constant, as HLS packs the adder tree into a single cycle for most ensemble sizes. For $n_e > 150$ the latency of the HLS result increases significantly. The VHDL implementation latency is typically longer than the latency achieved by the HLS. The VHDL is pipelined to achieve timing closure at higher clock frequencies than the 200 MHz target used for the HLS.

The time taken to synthesise the BDT increases linearly with n_e for the VHDL implementation, taking 40 minutes for the 1000 estimators ensemble. The HLS C synthesis time increases exponentially with the number of estimators, with synthesis for 200 estimators taking 21 hours. Vivado synthesis times for the HLS RTL output are significantly faster than the HLS C Synthesis which must run before, and increase linearly with the number of estimators.

Figure 5 shows the dependence of the same FPGA performance metrics on the maximum depth of the BDT, with n_e fixed at 10. The LUT usage increases exponentially with depth, with each additional layer in the trees adding as many nodes as there are above it. As before, the HLS estimate of the LUTs is high compared with the report after synthesising the produced RTL with Vivado. The LUT usage of the VHDL and Vivado-synthesized HLS are very similar, until at maximum depth of 6, the HLS implementation resource usage suddenly increases. At the same point, the latency and synthesis time drastically increase. The latency of the VHDL implementation increases linearly, with one extra clock cycle per depth. Synthesis time increases exponentially with depth, with the synthesis for a depth of 10 taking 27 hours.

3.4 Resource model

Given the expected scaling of LUTs with model hyperparameters — linear with n_e and exponentially (base two) with depth — we analytically describe the resource usage using the following relation:

$$r = k_0 \cdot n_e + k_1 \cdot n_e \cdot 2^d,$$

where r is the resource usage (LUTs), n_e the number of estimators, d the tree depth, and k_0, k_1 are unknown constants. The term linear in n_e represents the resource of the adder-tree which grows with the number of trees. The term linear in n_e and exponential with d represents the logic used for the trees, of which there are n_e , while the number of decision nodes doubles at each layer in depth. Other hyperparameters — such as the loss function, learning rate, and number of features — may impact the classification performance of the model, but would not affect the resource usage. A fit to the measurements of trained and synthesised BDTs using the VHDL implementation was performed, yielding:

$$r = 22 \cdot n_e + 53 \cdot n_e \cdot 2^d.$$

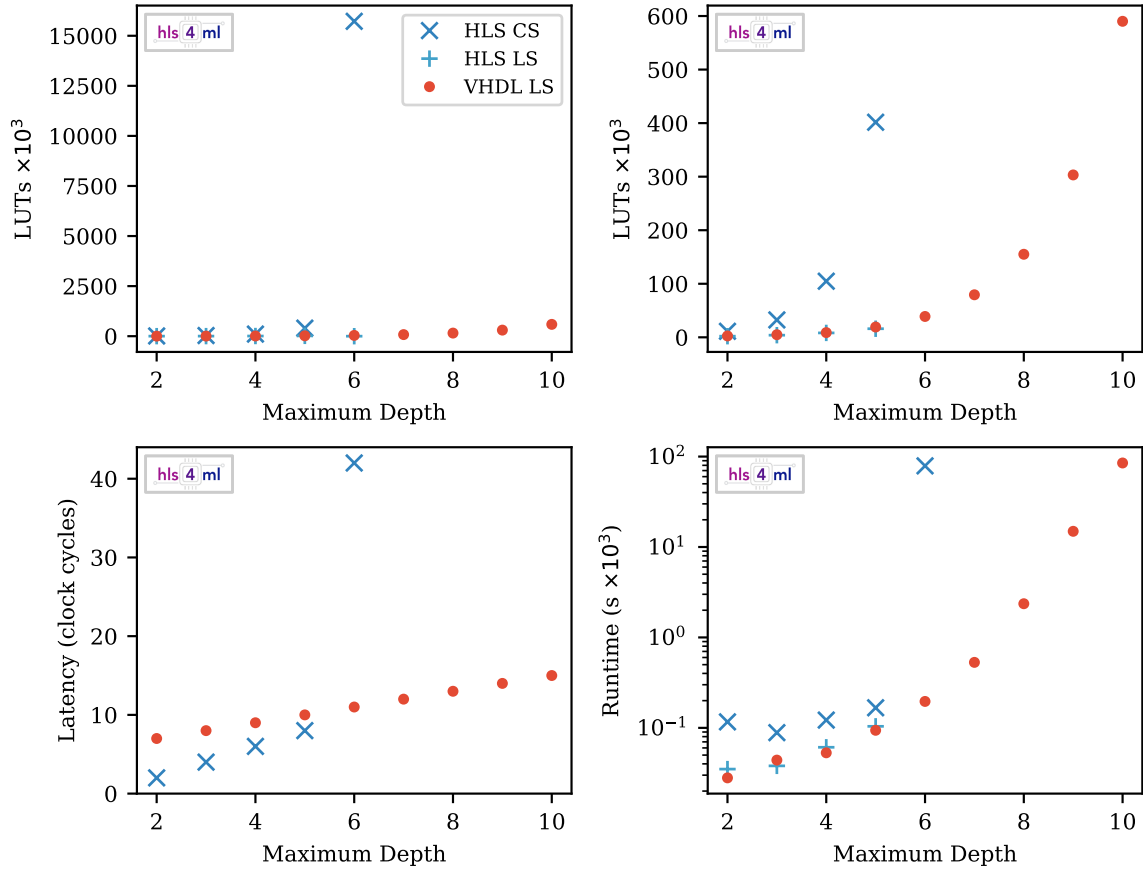


Figure 5. Dependence of LUT usage (top row), inference latency (bottom left), and synthesis time (bottom right) on the maximum depth of the BDT, with 10 estimators. The top right plot is a view of the same data as the top left, with reduced range. Different stages of synthesis are shown: C-Synthesis estimate of HLS (HLS CS), utilisation report after Logic Synthesis step of RTL produced by HLS (HLS LS), and utilisation report after Logic Synthesis of the VHDL implementation (VHDL LS).

All features, thresholds and scores were encoded with 18 bits. Figure 6 shows this scaling model over the measured BDT results used for the single-parameter scans in figures 4 and 5, showing good agreement.

3.5 Varying clock frequency and precision

Vivado HLS automatically pipelines FPGA designs, according to the target clock period specified by the developer. When using the HLS workflow, the `hls4ml` library allows the user to choose a target clock period for the BDT model. Generally, a faster target clock frequency requires more pipeline stages, so more clock cycles will be needed to perform the inference. The left plot of figure 7 shows the pipeline depth increasing with target clock frequency from 6 clock cycles at 100 MHz to 29 cycles at 500 MHz. The single inference latency in nanoseconds (the product of the latency in clock cycles and the clock period) is relatively constant with the target clock frequency. The lowest single inference latency is 52 ns at 250 MHz while the highest is 62.2 ns at 450 MHz. Using a higher clock frequency will achieve overall faster inference when classifying several input feature vectors, since the initiation interval is 1 in all cases.

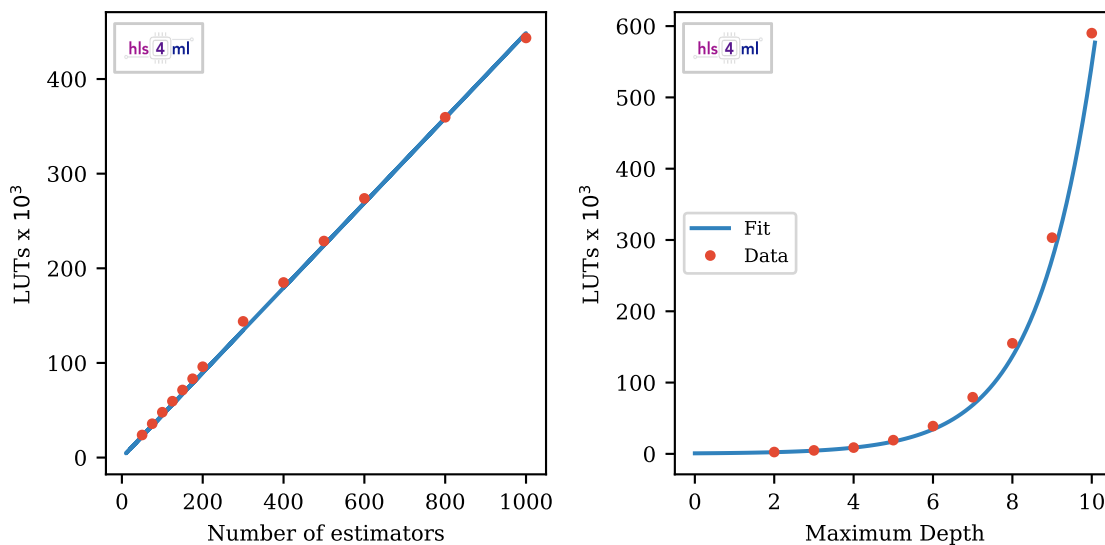


Figure 6. Comparison of LUT usage between measured results (points) and resource usage model (curve). Left: as a function of number of estimators with depth fixed at 4. Right: as a function of the depth, with number of estimators fixed at 10.

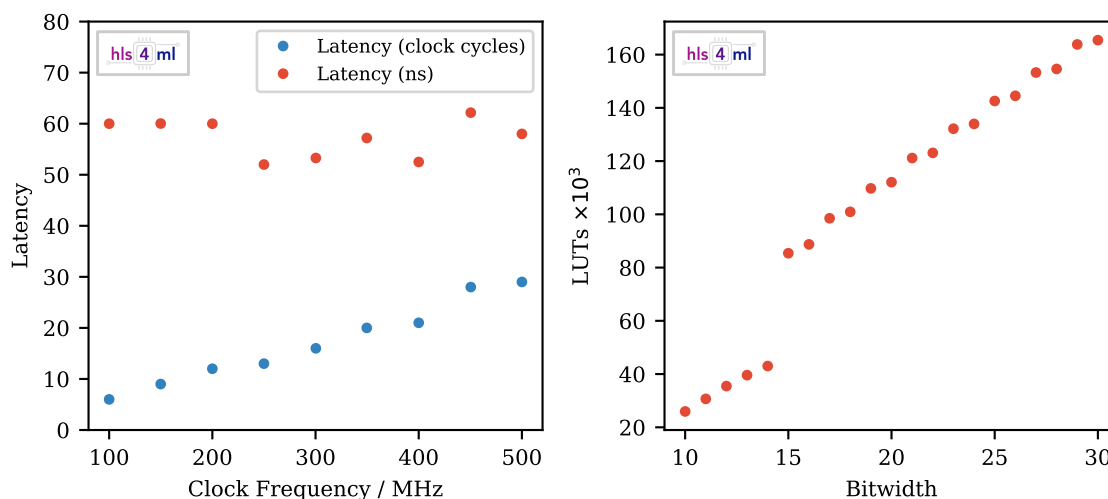


Figure 7. Left: latency — in clock cycles and nanoseconds — of the benchmark BDT model with 100 estimators of depth 4, as a function of the target clock frequency. Right: resource usage as a function of the bitwidth used for all features, thresholds and scores.

The variation of resource usage with the bitwidth is shown in the right plot of figure 7 for LUTs, the dominant resource used for BDTs. Four integer bits were used in all cases, as in figure 3. The increase in resource usage with bitwidth is approximately linear, but with a significant step change transitioning from 14 to 15 bits.

The benchmark model, as well as scans over the number of estimators and maximum depth, were evaluated using 18 bits. From figure 3 this can be seen to be comfortably sufficient to give numerically equivalent results to the CPU evaluation of the model. Using 14 bits, the ratio between Area Under the ROC Curve (AUC) achieved with the FPGA versus CPU inference is above 99.9%

for all classes. For many applications, this performance will be adequate, and the resource saving seen in figure 7 from using 14 bits or below can be taken advantage of. In other cases, for example selecting rare signals with a large background, the extra precision from using more bits may be desirable to maintain reliable, stable performance not susceptible to numeric effects.

4 Summary and outlook

We presented the implementation of BDT conversion to FPGA firmware in the `hls4ml` library. Taking as an example a multiclass classification problem from high energy physics (the identification of boosted jets based on substructure information), we show how a state-of-the-art algorithm could be deployed on an FPGA with a typical inference time of 12 clock cycles (i.e., 60 ns at a clock frequency of 200 MHz). We discussed the dependence of the FPGA resource usage and inference latency upon the model hyperparameters, presenting a model which predicts the resource usage well. We compared an HLS-based implementation to a VHDL one, as a function of the model size. Both the workflows are supported in `hls4ml`. The presented workflow provides a resource effective alternative to Neural Network deployment, which we discussed in a previous publication [6]. Compared to a Neural Network applied to the same problem, a BDT is able to achieve very similar performance, with a comparable inference latency. The implementation of BDTs in the FPGA utilises LUTs most heavily, while the Neural Network predominantly uses DSPs. This functionality of the `hls4ml` library could support an efficient deployment of algorithms analogous to that described in ref. [8], which took data at the CMS experiment during the LHC Run II.

Acknowledgments

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. 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., 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. Z.W. is supported by the National Science Foundation under Grants No. 1606321 and 115164.

References

- [1] B.P. Roe, H.-J. Yang, J. Zhu, Y. Liu, I. Stancu and G. McGregor, *Boosted decision trees, an alternative to artificial neural networks*, *Nucl. Instrum. Meth. A* **543** (2005) 577 [[physics/0408124](#)].
- [2] H.-J. Yang, B.P. Roe and J. Zhu, *Studies of boosted decision trees for MiniBooNE particle identification*, *Nucl. Instrum. Meth. A* **555** (2005) 370 [[physics/0508045](#)].
- [3] A. Radovic et al., *Machine learning at the energy and intensity frontiers of particle physics*, *Nature* **560** (2018) 41.
- [4] CMS collaboration, *Observation of a New Boson at a Mass of 125 GeV with the CMS Experiment at the LHC*, *Phys. Lett. B* **716** (2012) 30 [[arXiv:1207.7235](#)].

- [5] D. Guest, K. Cranmer and D. Whiteson, *Deep Learning and its Application to LHC Physics*, *Ann. Rev. Nucl. Part. Sci.* **68** (2018) 161 [[arXiv:1806.11484](#)].
- [6] J. Duarte et al., *Fast inference of deep neural networks in FPGAs for particle physics*, *2018 JINST* **13** P07027 [[arXiv:1804.06913](#)].
- [7] V.V. Gligorov and M. Williams, *Efficient, reliable and fast high-level triggering using a bonsai boosted decision tree*, *2013 JINST* **8** P02013 [[arXiv:1210.6861](#)].
- [8] CMS collaboration, *Boosted Decision Trees in the Level-1 Muon Endcap Trigger at CMS*, *J. Phys. Conf. Ser.* **1085** (2018) 042042.
- [9] M. Owaida, H. Zhang, C. Zhang and G. Alonso, *Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms*, in proceedings of the *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Ghent, Belgium, 4–8 September 2017, pp. 1–8.
- [10] M. Owaida, A. Kulkarni and G. Alonso, *Distributed inference over decision tree ensembles on clusters of FPGAs*, *ACM Trans. Reconfigurable Technol. Syst.* **12** (2019) 17.
- [11] M. Barbareschi, S. Del Prete, F. Gargiulo, A. Mazzeo and C. Sansone, *Decision Tree-Based Multiple Classifier Systems: An FPGA Perspective*, *Lect. Notes Comput. Sci.* **9132** (2015) 194.
- [12] S. Buschjäger and K. Morik, *Decision tree and random forest implementations for fast filtering of sensor data*, *IEEE Trans. Circuits Syst. I Regul. Pap.* **65** (2018) 209.
- [13] R. Kułaga and M. Gorgon, *FPGA implementation of decision trees and tree ensembles for character recognition in Vivado HLS*, *Image Process. Commun.* **19** (2014) 71.
- [14] M. Pierini, J.M. Duarte, N. Tran and M. Freytsis, *HLS4ML LHC Jet dataset (150 particles)*, (2020), <https://doi.org/10.5281/zenodo.3602260>.
- [15] F. Pedregosa et al., *Scikit-learn: Machine learning in Python*, *J. Mach. Learn. Res.* **12** (2011) 2825 [[arXiv:1201.0490](#)].
- [16] National Instruments, *FPGA Fundamentals*, (2019) <https://www.ni.com/it-it/innovations/white-papers/08/fpga-fundamentals.html>.
- [17] T. Chen and C. Guestrin, *XGBoost: A scalable tree boosting system*, in proceedings of the *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*, San Francisco, CA, U.S.A., 13–17 August 2016, pp. 785–794.
- [18] A. Hocker et al., *TMVA — Toolkit for Multivariate Data Analysis*, [physics/0703039](#).

Large-Scale Distributed Training Applied to Generative Adversarial Networks for Calorimeter Simulation

*Jean-Roch Vlimant*¹, *Felice Pantaleo*², *Maurizio Pierini*², *Vladimir Loncar*², *Sofia Vallecorsa*^{2,3}, *Dustin Anderson*¹, *Thong Nguyen*¹, and *Alexander Zlokapa*¹

¹California Institute of Technology, Pasadena, California, U.S.

²CERN, Meyrin, Switzerland

³Gangneung-Wonju National University, Korea

Abstract. In recent years, several studies have demonstrated the benefit of using deep learning to solve typical tasks related to high energy physics data taking and analysis. In particular, generative adversarial networks are a good candidate to supplement the simulation of the detector response in a collider environment. Training of neural network models has been made tractable with the improvement of optimization methods and the advent of GP-GPU well adapted to tackle the highly-parallelizable task of training neural nets. Despite these advancements, training of large models over large data sets can take days to weeks. Even more so, finding the best model architecture and settings can take many expensive trials. To get the best out of this new technology, it is important to scale up the available network-training resources and, consequently, to provide tools for optimal large-scale distributed training. In this context, our development of a new training workflow, which scales on multi-node/multi-GPU architectures with an eye to deployment on high performance computing machines is described. We describe the integration of hyper parameter optimization with a distributed training framework using Message Passing Interface, for models defined in keras [12] or pytorch [13]. We present results on the speedup of training generative adversarial networks trained on a data set composed of the energy deposition from electron, photons, charged and neutral hadrons in a fine grained digital calorimeter.

1 Introduction

Deep neural networks (DNN) are machine learning models with many parameters that are effectively trained using stochastic gradient descent methods. The power of DNN at executing challenging tasks learned from data is very attractive to the most diverse fields of science, business and society at large, and notably in High Energy Physics (HEP). In recent years, there have been a significant number of articles reporting promising results with applying deep learning to HEP challenges. DNN can be used in supervised learning, an approach with which one wants to learn, from a training data set, a mapping from a set of input features to a set of target features, in order to predict future target values on previously not seen data.

Within the context of unsupervised learning and generative models, multiple neural networks can be trained concurrently in the generative adversarial (GAN) scheme [1]. In this scheme a neural network is generating featured data starting from random numbers, and a

second neural network is set to distinguish between generated samples and the data from an initial data set. Upon convergence the generative model is able to generate new data that looks statistically identical to the presented training data. Training GAN with large input data sets and a large input space turns out to be very intensive, taking several hours per epoch. Such generative model have had tremendous publicity recently in the field of data science thanks to their great success in generating complex data (images mostly) and application of such models in the field of high energy physics are showing great promises [3].

Deep models require quite large data set so as to be trained (because of the large number of parameters to be adjusted) : even with the large boost given by general purpose graphical units (GP-GPU), training remains quite a computing intensive task that may last from days to weeks to converge, if not worse. Therefore we explore several parallelism approaches to the stochastic gradient descent method in order to speed up further the training of neural networks.

In training GAN, as in training other neural network models, the choice of some parameters of the models that cannot be learned with stochastic gradient (such as batch size, learning rate, number of hidden layers, ...) are left for optimization. Trial and errors optimization on such hyper parameters is time consuming and requires a rather high level of educated expertise. In this work, we explore the use of Bayesian optimization with a Gaussian process assumption for the prior, as well as an evolutionary algorithm on the hyper parameters.

This document is structured as follows, we provide the relevant details of artificial neural network and generative adversarial networks in section 2 as well as details on stochastic gradient descent in section 3. The problem of hyper parameter optimization is described in section 5. The directions of distributed training are explored in section 4 and results obtained at various facilities are provided in section 6. We conclude with some outlook on distributed training and optimization in section 7.

2 Neural Networks and Generative Adversarial Network

Among the many mathematical models in the field of machine learning, Artificial neural network (ANN) are a type of model initially inspired from the biological functioning of the brain. Such a model is composed of an input layer with as many nodes (neurons) as desired input features, an output layer with a number of neurons in adequacy with the problem that one wishes to solve and one or multiple internal layers composed of internal nodes arranged in a variety of topology [2]. The only restriction to the topology and computation will be clear from section 5 with the necessity of having a differentiable computation graph.

ANN can be used for supervised learning, a task for which one possesses a target feature (boolean for classification, continuous for regression) which one wishes to learn, from a corpus of training data, and then predict on unseen similar data. On the other hand, with unsupervised learning, one may wish to train a model that learns the structure of a data set so as to be able to generate new samples, that statistically resemble the original data. Such models are particularly attractive because, even though they might be hard to train, as a one time process, generating new data can be significantly faster, by several orders of magnitude, than existing simulation software. In the context of the resource constraint computing model for the High Luminosity Large Hadron Collider, these models, even with reduced fidelity, could enable the production of the large data set of simulated collisions required during analysis.

One class of generative model is the so called Generative Adversarial Network [1] composed of two ANN. The first one is the generative model (generator) which produces new data from numerical vectors drawn at random from pre-determined distributions and aiming at mimicing sample drawn from the original data set. It is however hard to train such model

with traditional methods because of the absence of a tractable metric to estimate the goodness of identity of the generated data and the original data. The second ANN, the so called discriminator, is set to classify properly the generated data and the original data. It therefore takes input from the output space of the generator and produces a label on the origin of the input (fake or original). The two ANN are trained in an alternating fashion, the discriminator is exposed to the original data and generated samples while the generator is trained with the loss of the discriminator with generated samples labelled as original. During this procedure the generator is trained to fool the discriminator. Further in depth details on GAN can be found in [1].

3 Model Training with Stochastic Gradient Descent

ANN parameters are learned from data through an optimization procedure aiming at maximizing likelihood or minimizing errors of the model, cast as a minimization of a loss function. A standard method in convex optimization is gradient descent, during which the parameter space is navigated by taking infinitesimal steps on the opposite direction of the gradient of the loss with respect to the parameters of the model. By having ANN models and loss function fully differentiable, one can compute the gradient analytically and is left with only the evaluation of these gradients. Mini-batch Stochastic gradient descent (SGD) is an algorithm that computes the direction of stepping not from the gradient of a single sample but from a collection of samples ("batch"). The optimization of the loss function of an ANN is not a convex optimization problem, however stochastic gradient descent has shown great success in finding good parameters for ANN [4]. An epoch indicates a cycle of the SGD where all batches of the training data set have been taken into account. Several epochs are usually needed to achieve convergence of the model.

Further manipulations can be performed on the batch gradients in the optimizer algorithm in order to reach faster convergence: these algorithms are driven by various parameters, such as the learning rate (measuring the step size in the parameter space). Further information on optimizers are available in [5].

4 Distributed Training

In this section we present several ways for parallel computation of the gradients needed for SGD. One can leverage these levels of parallelism on high performance computing (HPC) centers composed of many nodes with high bandwidth connectivity and obtain a shorter time to solution. The computation and communication is orchestrated using the MPI [6] framework, abstracting the communication protocols from the computation. An MPI program is executed over multiple processes, running on multiple physical hosts on the cluster. We call each process a worker, and it does not matter a priori if they get executed on the same physical node. Depending on the topology of the HPC, there can be more than one GP-GPU per physical host, and we enforce that we do not get more than one process associated with one GP-GPU. De-facto, in the following, each worker is referring to a process with at least one dedicated GP-GPU attached. Results of applying the following techniques are reported in section 6.

4.1 Batch Parallelism

In the scheme of the SGD, with a small batch size, the effective gradient is very noisy due to statistical fluctuations over the small number of gradients averaged. It is therefore necessary to have a sufficiently large batch size. Even with ever growing memory in GP-GPU, the

amount of input data and network information might become larger than the available memory, preventing from doing efficient computation on GP-GPU. When faced with this situation, the batch can be divided in sub-batches and distributed to multiple workers. The gradients then are efficiently gathered for the averaging over the batch. We use the Horovod [7] library, developed by Uber, efficiently implementing this process, with modifications that we brought to the library interface. With these modifications, sub-groups of ranks can be individually initialized and work in concert for batch parallelism. This therefore allowed each sub-group to work on the computation of the gradient of one batch. HPC topology with many GP-GPU per node are well suited for this method since the communication can be implemented very efficiently with GPU-2-GPU fast communication such as Nvlink [8]. In summary, batch parallelism enables running SGD with significant batch size when the GP-GPU memory is a limiting factor.

4.2 Data Parallelism

The SGD algorithm is sequential in the successive batches used to compute updates to the model parameters. The computation of the gradient for multiple batches can however be distributed from a master process to multiple processes so as to calculate them in parallel. One of the immediate points, to be noted about doing so, is that all workers are calculating the gradient evaluated for the set of model parameters, over different batches of data. After successive update of the parameter of the master model with the gradients computed in this manner, one updates a set of hyper parameters using gradients calculated on outdated hyper parameter values. This generates, if not mitigated algorithmically, the staleness of the gradients and loss of convergence at fixed number of epochs. Observation of such effect is available in [6] and can be mitigated with dedicated algorithm like the one in [10]. There is a trade-off between the speedup obtained with computing the gradient from multiple batches and the degradation of the convergence rate of the model. In summary, data parallelism is having the gradient of many batches done in parallel and integrated to a master model.

4.3 Model Parallelism

Deep neural networks can turn up very large, with billions of parameters. It might be grown at a point at which the amount of memory required is too large (assuming a fixed batch size that itself fits on the device, see section 4.1 otherwise). By virtue of the chain rule in calculating the gradient of the loss, the calculation can be factorized by layers of the ANN. It is therefore effectively possible to distribute part of the computation graphs corresponding to successive layers of the model to several devices, with the need to only communicate the activation of the layer at the boundaries. We use the native functionality of tensorflow [11] to put part of the computation graph on different device, in the case of multiple GP-GPU per node. In summary, model parallelism allows to spread the forward and backward passes of SGD over multiple devices.

5 Model Parameter Optimization

There are however parameters of the models that cannot be adjusted using SGD and that have significant impact on model performance. Such parameters are for example learning rate of the optimizer or the number of nodes in internal layers of the ANN. These parameters are often called hyper-parameters to differentiate them from the other parameters. The hyper parameters are often scanned by a developer while looking for a set with good performance

after training, and such scanning is a lengthy, painful and sometimes random process. Such search can be replaced with a full grid search but is computationally prohibitive in large hyper-parameter space dimension. Gradient descent is often not a possible algorithm because of the discreteness of hyper parameters. Below, we give details of two methods commonly used for hyper parameter optimization. It should be noted that it is not necessary to use the loss function previously defined as the performance metric of the model. Any other metric may be used to quantify the performance of the model, and be used as figure of merit (FOM) during hyper parameter optimization. Specifically, this figure of merit does not need to be differentiable. We eventually provide details of the cross validation procedure, as a must for model comparison.

5.1 Bayesian Optimization using Gaussian Process Prior

A method commonly used for hyper parameter optimization is using Bayesian optimization with modeling the FOM with Gaussian processes. Details of the algorithms can be found in [14]. We use the scikit-optimize implementation [15] of the optimizer, which one queries for values of the hyper parameters to evaluate the performance for. With successive sampling of the hyper parameter space, the optimizer gets better at providing suggestions close to the optimal hyper parameters. The complexity of this algorithm grows as the cube of the number of sampling points and can become prohibitive for large hyper parameter space in which the convergence is taking multiple sampling iterations. While the process of trial and error is essentially sequential, multiple set of hyper parameters can be evaluated simultaneously.

5.2 Evolutionary Algorithms

Another class of algorithms used for hyper parameters optimization is based on genetic evolution [16]. We implement a simple version where the hyper parameters are trivially encoded as the chromosome vector and the FOM is trivially mapped to the fitness of a chromosome. The algorithm starts with an initial population of chromosomes taken at random within the allowed space, a fraction of chromosome with best fitness are kept to carry on producing the next generation. The population of the next generation is created from random linear interpolation from fittest chromosomes and additional mutation are obtained by moving the chromosome at random within an infinitesimal volume. The fitness of the next generation is evaluated and the process is repeated for a certain number of iterations. This algorithm is expensive in the number of evaluation calls (training of a model to convergence) but can have an advantage over the previous method when the hyper optimization requires many iterations. The algorithm is sequential in the successive generation, but fully parallel in evaluating the fitness of a given generation.

5.3 Cross Validation

The training of ANN is subject to some level of fluctuation due to the initial random weights and numerical rounding happening during SGD. There is another stochastic component in batch parallelism (see section 4.2) where gradients are considered in an order driven by computation time on each node. The performance of a model is usually evaluated on the validation set, distinct from the training one: its finite size and choice introduces a bias in the measured performance. One can estimate both performance uncertainties using the K-split cross validation algorithm in which the initial data set is split into multiple parts that are used to concurrently evaluate the performance of a choice of model hyper parameters. The initial dataset is divided in K parts (a.k.a. splits): K-1 splits are used for training, the remaining one

is left out for validation. One can quote the average performance over the K FOM values obtained with the K-splits cross validation. The training and evaluation of the different models are trivially parallel, and one can leverage large number of nodes at an HPC to run cross validation in the same time it would take to train one model, with the advantage of having a better estimate of the performance.

6 Results

We report here on scaling performance obtained in training a GAN on 3D calorimetry simulated data on several supercomputers and using libraries implementing different schemes of distributed training. Further details on the model and data set are available in [17].

We report scaling performance using the mpi-learn [19] package adapted to train GAN models, and with the extension for performing hyper-parameter optimization in mpi-opt [22]. The original paper reported quasi-linear scaling up to 8-15 workers [19]. The GAN training was performed on PiZ Daint [20] and Titan [21] supercomputers equipped with NVIDIA®V100 GP-GPU and K20 respectively. As can be seen in figure 1 the preliminary results on scaling for GAN is not great with a factor of about 1:2 speedup per worker up to 20 workers, and 20x speedup with 100 workers. We have observed no degradation of the fidelity of the trained model up to using 15 workers. In the current setup this is not the most efficient use of the resource as the speed up is not linear, but still provide a significant improvement over using a single node. We hypothesize that the deviation from a linear speed-up is due to the fact that the workload for the workers is too small and that most time is spend with the master handling the weights update and communication to the workers. Better understanding of the scaling would require further in depth analysis, which the authors are planning as future work.

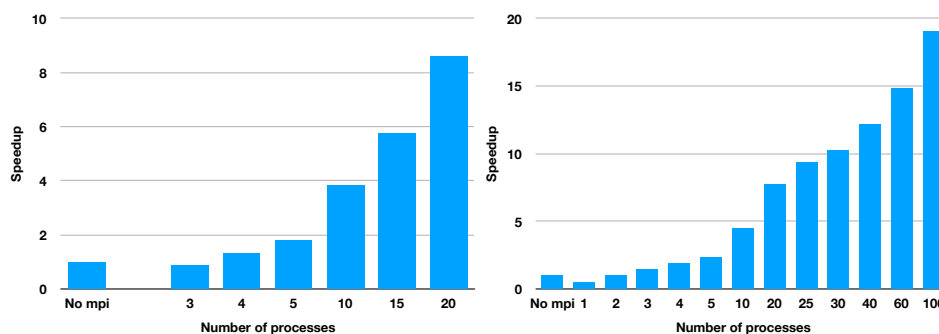


Figure 1. Speed up of training a 3D calorimeter energy deposition generative adversarial network, as a function of the number of training processes and with respect to training with one process only. Results obtained when running on CSCS Piz Daint (left) and ORNL Titan (right).

7 Discussion

In this article, we review the technicalities of training neural networks on distributed systems. We present several ways to parallelise training of neural networks, including Generative Adversarial Networks. We describe methods to perform hyper parameter optimization, implemented in a python library for deployment on HPC resource. We report results on scaling of

distributed training on two supercomputers, obtaining promising speedup without noticeable degradation in model fidelity.

The authors continue to work on detailed validation and optimization of these preliminary results, subject to resource availability (large scale training benchmarking are expensive on resource allocation), toward releasing a turn-key software for distributed training and optimization of neural networks using keras, tensorflow and pytorch.

Acknowledgement

Part of this work was conducted on Piz Daint at CSCS under the allocations d59 (2016) and cn01 (2018). Part of this work was conducted on Titan at OLCF under the allocation csc291 (2018). Part of this work was conducted at "*iBanks*", the AI GPU cluster at Caltech. We acknowledge NVIDIA, SuperMicro and the Kavli Foundation for their support of "*iBanks*". Part of the team is funded by ERC H2020 grant number 772369.

References

- [1] Generative Adversarial Networks, Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, <https://arxiv.org/abs/1406.2661>
- [2] The Neural Network Zoo, Fjodor van Veen, <http://www.asimovinstitute.org/neural-network-zoo/>
- [3] Machine learning at the energy and intensity frontiers of particle physics, Alexander Radovic, Mike Williams, David Rousseau, Michael Kagan, Daniele Bonacorsi, Alexander Himmel, Adam Aurisano, Kazuhiro Terao, Taritree Wongjirad, <https://www.nature.com/articles/s41586-018-0361-2>
- [4] A stochastic approximation method, Robbins, H. and S. Monro (1951), The annals of mathematical statistics, 400–407.
- [5] An overview of gradient descent optimization algorithms, Sebastian Ruder, <https://arxiv.org/abs/1609.04747>
- [6] MPI Forum. MPI: a message passing interface standard. (1994). Technical Report (1994).
- [7] Horovod: fast and easy distributed deep learning in TensorFlow, Alexander Sergeev and Mike Del Balso, <https://arxiv.org/abs/1802.05799>
- [8] <https://www.nvidia.com/en-us/data-center/nvlink/>
- [9] NVIDIA Collective Communications Library (NCCL), <https://developer.nvidia.com/nccl>
- [10] Gradient Energy Matching for Distributed Asynchronous Gradient Descent, Joeri Hermans, Gilles Louppe, <https://arxiv.org/abs/1805.08469>
- [11] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [12] Keras, François Chollet and others, <https://keras.io>
- [13] Automatic differentiation in PyTorch, Adam Paszke et al., <https://pytorch.org>
- [14] Practical Bayesian Optimization of Machine Learning Algorithms Jasper Snoek, Hugo Larochelle and Ryan P. Adams Advances in Neural Information Processing Systems, 2012
- [15] scikit-optimize python library, <https://scikit-optimize.github.io/>
- [16] Genetic Algorithms, Numerical Optimization, and Constraints, Zbigniew Michalewicz, In: Morgan Kaufmann, 1995, pp. 151–158.
- [17] Calorimetry with Deep Learning: Particle Classification, Energy Regression, and Simulation for High-Energy Physics, Benjamin Hooberman et al., https://dl4physicalsciences.github.io/files/nips_dlps_2017_15.pdf

- [18] Intel® Math Kernel Library for Deep Neural Networks, <https://github.com/intel/mkl-dnn>
- [19] An MPI-Based Python Framework for Distributed Training with Keras, Dustin Anderson, Jean-Roch Vlimant, Maria Spiropulu, <https://arxiv.org/abs/1712.05878>https://github.com/vlimant/mpi_learn
- [20] <https://www.cscs.ch/computers/piz-daint/>
- [21] <https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/>
- [22] https://github.com/vlimant/mpi_opt

PAPER • OPEN ACCESS

Generative Adversarial Networks for fast simulation

To cite this article: Federico Carminati *et al* 2020 *J. Phys.: Conf. Ser.* **1525** 012064

View the [article online](#) for updates and enhancements.

You may also like

- [The 2022 Plasma Roadmap: low temperature plasma science and technology](#)
I Adamovich, S Agarwal, E Ahedo et al.
- [Deep learning-based methods in structural reliability analysis: a review](#)
Sajad Saraygord Afshari, Chuan Zhao, Xinchun Zhuang et al.
- [Atomic Layer Deposition of Sidewall Spacers: Process, Equipment and Integration Challenges in State-of-the-Art Logic Technologies](#)
Michael P. Belyansky, Richard Conti, Shahrukh Khan et al.

Generative Adversarial Networks for fast simulation

Federico Carminati¹, Gulrukh Khattak¹, Vladimir Loncar², Thong Q Nguyen³, Maurizio Pierini¹, Ricardo Brito Da Rocha¹, Konstantinos Samaras-Tsakiris¹, Sofia Vallecorsa¹ and Jean-Roch Vlimant³

¹ CERN, 1 Esplanade des Particules, Geneva, Switzerland

³ California Institute of Technology, 1200 E. California Blvd., Pasadena, CA 91125, USA

² Institute of Physics Belgrade, University of Belgrade, Pregrevica 118, 11080, Belgrade, Serbia

Abstract. Deep Learning techniques are being studied for different applications by the HEP community: in this talk, we discuss the case of detector simulation. The need for simulated events, expected in the future for LHC experiments and their High Luminosity upgrades, is increasing dramatically and requires new fast simulation solutions. Here we present updated results on the development of 3DGAN, one of the first examples using three-dimensional convolutional Generative Adversarial Networks to simulate high granularity electromagnetic calorimeters. In particular, we report on two main aspects: results on the simulation of a more general, realistic physics use case and on data parallel strategies to distribute the training process across multiple nodes on public cloud resources.

1. Introduction and Related Work

High Energy Physics (HEP) relies heavily on Monte Carlo simulation in order to model complex processes. However the detailed Monte Carlo approach is both time and resource intensive: currently more than 50% of the Worldwide LHC Grid [1] resources are devoted to simulation [2]. The need in terms of simulated data is expected to increase by a factor $100\times$ for the High Luminosity LHC [3] and fast simulation alternatives are being investigated. Existing techniques, based, for example, on parametrization [4, 5, 6] can reach different speed-ups while retaining various levels of accuracy.

Detailed simulation of high granularity calorimeters is particularly time-consuming, however, their output, a pattern of energy depositions in the different cells, can be interpreted as pixel intensities in a three-dimensional image. Particle characteristics, such as its type, energy and incident angle are input to the simulation process and can be used to condition the training of Deep Neural Networks and obtain models capable of generating the desired output according to a specific set of input parameters. Image generation is an important aspect of machine learning: a number of approaches exists including Generative Adversarial Networks (GAN). GANs implement the idea of adversarial training for generating sharp and realistic images [7]; their application to HEP simulation was introduced by the LAGAN [8] and the CaloGAN [9] models. In this case particle showers for simplified calorimeters were simulated as sets of two-dimensional images. Since then, several studies have tested the application of GANs to HEP simulations (for example [10, 11]).

Our previous work [12, 13] introduced the simulation of an example of high granularity calorimeter using true 3D convolutions to further exploit the correlations in the volumetric space. We demonstrated the benefits of such an approach by training a network to generate



calorimeter energy showers according to the primary particle energy entering the calorimeter at a fixed (orthogonal angle). Here we generalise these results to a more realistic use case considering a particle entering the detector with a variable incident angle. Thus, our network learns a joint distribution of both the primary energy and the incident angle, it reproduces a detector volume that is $4\times$ larger and it introduces domain knowledge in order to reach a high level of accuracy. We also present updated results on the data parallel approach we used to train our network on distributed systems. Our final goal is to prove that, by using meta-optimization and hyper-parameters scans, it is possible to tune the network architectures to simulate different detectors. In this perspective, an efficient training process becomes essential and the accent should therefore be on optimizing the computing resources needed to train the networks, studying parallelization and cross-platform development. This report is organised as follows: firstly we introduce our 3D convolutional GAN model, the data set and the training strategy. Section 3 summarises some example results obtained from the validation of physics performance. Section 4 discusses the implementation of a data parallel training approach together with preliminary scaling benchmarks on public cloud resources. We then conclude with a brief outlook on our plans for future development.

2. The 3D convolutional GAN

The 3DGAN model, described in [12], represents a first proof of concept of the possibility to use 3D convolutional GANs to simulate high granularity calorimeters. It addresses, however, an extremely simplified use case: the 3D image is limited in size and the simulated particles enter the detector with a fixed 90° angle. The primary particle energy is used to condition the training process, loosely following the ACGAN [14] approach. The Hadamard product between the energy and the latent space vector is calculated and used as an input to the generator network. At the same time, the loss function includes a constraint on the total deposited energy [15, 16].

This work extends the scope of [12] and simulates more realistic particles entering the detector with a variable incident angle and generating images that are $4\times$ larger in size. Here, the GAN learns an angle-energy multivariate distribution and therefore the training is conditioned using both the incident angle and energy. As explained below, the 3DGAN architecture and the corresponding loss functions are modified to take into account physics based constraints.

2.1. Data set

The training data are generated in an effort to provide a common realistic data set that can be used to foster development of different Deep Learning and Machine Learning applications, from classification and regression networks to improve physics analysis to generative models for simulation. This data simulate a high granularity electromagnetic calorimeter (ECAL), designed in the context of the detector studies for the CLIC accelerator project [17], and they consist of a regular grid of 5.1 mm^3 cells and an inner calorimeter radius of 1.5 m . Further details can be found in [18, 16].

Here, we show results obtained using 140,000 showers produced by single electrons with a primary energy (Ep) range of $100 - 200\text{ GeV}$ and incident angle (θ) uniformly distributed between 60° and 120° . The final result is a three dimensional $51 \times 51 \times 25$ pixelized image centered around the barycenter of the shower: Figure 1 (a) presents 2D projections on the xy , xz and yz planes for Geant4 events.¹ Typically only a small fraction of cells (below 20%) receives some energy depositions and the size of the deposit can vary over a very large range (spanning more than 10 orders of magnitude). This high sparsity and large dynamic range

¹ The z axis lies along the detector depth and x , y are the transverse axes.

represent huge challenges compared to the typical RGB image generation problem where the pixel dynamic range is limited.

2.2. Architecture

A general GAN is made of two components, a discriminator and a generator: as explained in [12], the 3DGAN generator and discriminator networks consist of four 3D convolution layers. The discriminator has 16 filters with $5 \times 6 \times 6$ kernels and leaky ReLU activation functions in each layer. A batch normalization layer is added after the activation in all except the first layer. The output of the final convolution layer is flattened and connected to a sigmoid neuron corresponding to the GAN real/fake output as well as a linear unit performing energy regression. The generator has a latent vector of size 256. The first convolution layer has 64 filters with $6 \times 6 \times 8$ kernels. The next two layers have 6 filters of $5 \times 8 \times 8$ and $3 \times 5 \times 8$ kernels respectively. The last layer has a single filter of $2 \times 2 \times 2$ kernel. Leaky ReLU activation functions are used in all but the last layer that uses a ReLU. Batch normalization layers were added after the first and the second layer. The RMSprop [19] optimiser is used to train the network. The model is implemented in Keras [20] and Tensorflow v1.2 [21].

2.3. The loss function

As shown by the equation below, the loss function is build as a sum of several terms pertaining to the discriminator real/fake probability (L_G), the primary particle energy regression task (L_P) and a constraints on the total deposited energy (L_E), the pixel intensity spectrum (L_B) and the incident angles measurement (L_A). W are the corresponding weights, balancing the individual contributions.

$$L_{Tot} = W_G L_G + W_P L_P + W_A L_A + W_E L_E + W_B L_B$$

Extending the approach in [12], the last two quantities introduce domain-related terms in the loss function and they are essential in order to achieve the required level of accuracy over a very large pixel intensity dynamic range. L_G is implemented as a binary cross entropy, while a mean absolute percentage error is used for the primary energy, the deposited energy and the pixel intensity spectrum loss terms. The mean absolute error is used for the incident angle loss term.

2.4. The data preparation step and training process

Given the large pixel dynamic range, shown in figure 2, we do not normalize the training data, instead we slightly reduce the dynamic range by applying a power function using an exponent smaller than one. The exponent is treated as a hyper-parameter and adjusted, to a value of 0.85, through a trial-and-error procedure. This procedure is essential to improve the description of the lower end of the pixel intensity spectrum.

The adversarial approach designs the training as a competition between the discriminator and the generator [7]. We adopt a balanced approach, by training the discriminator and the generator alternatively using the same number of steps. Initially, the discriminator is trained on a batch of real images and a batch of generated images (and label switching is applied [22]). Then, the generator is trained twice while keeping the discriminator weights fixed. Training on a single NVIDIA GeForce GTX 1080 card for one epoch requires about two hours and the training runs for 60 epochs.

3. Validation and Optimization

As a preliminary check we visually verify that 3DGAN can reproduce typical energy deposition patterns for different energy and incident angle values. Figure 1 shows some examples: it

compares 2D energy shower projections on the xy , xz and yz planes, as predicted by Geant4 and 3DGAN.

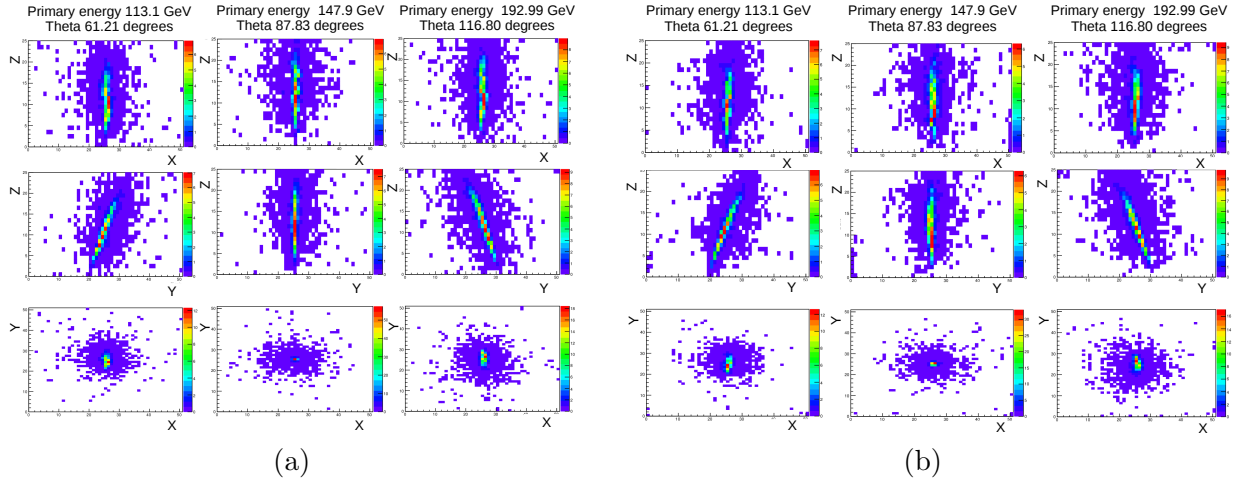


Figure 1. Geant4 vs. GAN generated events with similar primary energies and angles; a) Geant4 events ; b) generated events.

Figure 2 shows the Geant4/GAN ratio of the total energy deposited in the calorimeter as a function of the primary particle energy (a) and the single cell energy spectrum in linear (b) and logarithmic scale (c). It can be seen that, overall, GAN reproduces correctly the total energy deposited in the calorimeter and that single cell energies agree down to MeV values. As expected, the lower end of the spectrum is harder to simulate. Figure 3 (left) shows the Geant4 - GAN correlation difference calculated for different quantities (such as shower shapes distributions, deposited energy, incident angle, primary particle energy, etc..) as predicted by GAN and Geant4. It can be seen that the 3DGAN can correctly reproduce most internal correlations.

4. Distributed Training on public cloud

As Deep Learning models increase in complexity, model size and training time, the role played by distributed training becomes of primary importance: 3DGAN, for example, sums up to slightly more than a million parameters and it reaches convergence in about 5 days of training.

Several different algorithms for distributed training have been developed in recent years. Generally these algorithms work by splitting the training load across multiple concurrent

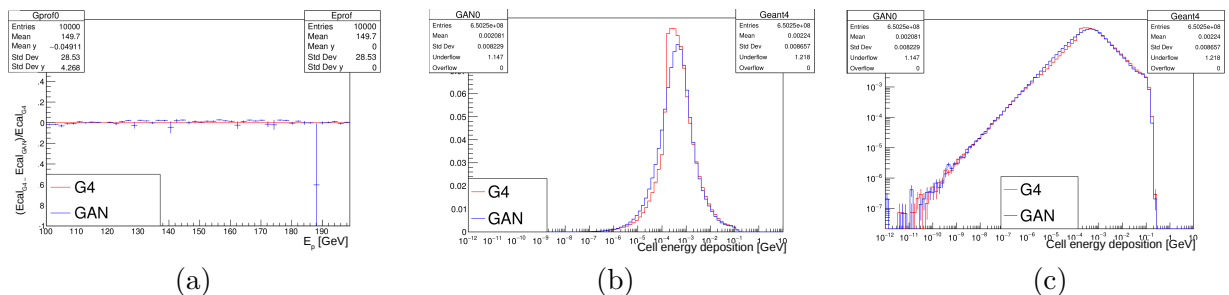


Figure 2. Geant4 vs. GAN comparison for 100-200 GeV primary particle energies; a) total energy deposited in calorimeter; b) single cell energy; c) single cell energy in log scale.

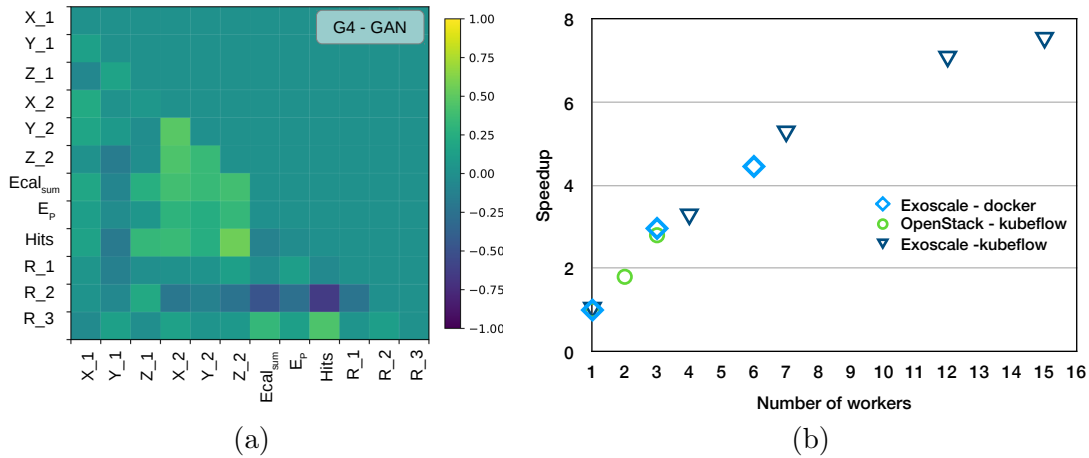


Figure 3. a) Geant4 - GAN difference between internal correlations calculated for several shower features; b) Speed up of training 3DGAN on the orthogonal incident angle sample, as a function of the number of MPI workers and with respect to training with one MPI worker.

processes, either threads on a single machine or jobs spread across separate nodes [23]. In order to reduce the training time we have interfaced 3DGAN to several frameworks, including Horovod [24] and mpi-learn [25], and benchmarked the parallel training process on different HPC systems [26, 23, 12].

Here we report on updated results on 3DGAN scaling performance on public clouds. Several initiatives exist that aim at understanding how the scientific community can integrate public clouds in their computing models. The European Commission funded project Helix Nebula Science Cloud (HNSciCloud) [27], for example, explored an hybrid cloud model linking together commercial cloud service providers and research organisations' in-house resources in order to provide an innovative vision for supporting the growing computing needs of the research community.

We have created a mpi-learn based docker [28] image and integrated it to kubernetes [29] and kubeflow [30] in order to smoothly deploy our workload on commercial cloud providers (for example Exoscale², equipped with NVIDIA P100 GPUs), via the HNSciCloud project. Results are shown in figure 3 (right): we have tested different deployment configurations and no overhead, due to the docker, kubernetes or kubeflow additional layer, has been observed. We have also compared the speedup performance obtained running on external cloud (blue) and on a small local set of GPUs, available in CERN Openstack (green) and we observed no difference in timing. Training time is significantly reduced, but the current speed-up is not linear. A possible explanation is that the workload for the workers is too small with respect to communication time and weights updates processing by the master. Analysis and optimisation of resource usage is part of our on-going work.

5. Summary and Plans

We presented updated results on the development and optimisation of our 3DGAN model: a three-dimensional convolutional Generative Adversarial Network for electromagnetic shower simulations in high granularity calorimeters. In particular, we have obtained results within 10% of Geant4 by introducing specific physics-related terms in the loss function and proved that our network can learn complex joint distributions. We have developed a fast deployment framework, based on commercially available platforms, such as kubernetes and kubeflow, to parallelise the

² <https://www.exoscale.com>

3DGAN training process across distributed resources available on public cloud. We obtained promising results that suggest that alternatives to in-house resources could be used to efficiently perform this kind of tasks.

In order to further optimise 3DGAN performance and extend it to the simulation of different calorimeter geometries we are currently developing a hyper-parameter optimisation framework based on genetic algorithms.

References

- [1] Bird I 2011 *Annual Review of Nuclear and Particle Science* **61** 99–118 (Preprint <https://doi.org/10.1146/annurev-nucl-102010-130059>) URL <https://doi.org/10.1146/annurev-nucl-102010-130059>
- [2] The Hep Software Foundation 2019 *Computing and Software for Big Science* **3** 7 ISSN 2510-2044 URL <https://doi.org/10.1007/s41781-018-0018-8>
- [3] Apollinari G, Bjar Alonso I, Brning O, Fessia P, Lamont M, Rossi L and Tavian L 2017 *CERN Yellow Rep. Monogr.* **4** 1–516
- [4] Lukas W 2012 *International Conference on Computing in High Energy and Nuclear Physics* vol 396
- [5] Orbaker D 2010 *International Conference on Computing in High Energy and Nuclear Physics* vol 219
- [6] Autiero D *et al.* (NOMAD Collaboration) 1998 *Nucl. Instrum. Methods Phys. Res., A* **425** 188. 28 p URL <https://cds.cern.ch/record/364720>
- [7] Goodfellow I J, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A and Bengio Y 2014 *ArXiv e-prints (Preprint 1406.2661)*
- [8] de Oliveira L, Paganini M and Nachman B 2017 *Comput. Softw. Big Sci.* **1** 4 (Preprint 1701.05927)
- [9] Paganini M, de Oliveira L and Nachman B 2018 *Phys. Rev. Lett.* **120** 042003 (Preprint 1705.02355)
- [10] Erdmann M *et al.* *ArXiv High Energy Physics - Experiment e-prints (Preprint https://arxiv.org/abs/1807.01954)*
- [11] Chekalina V *et al.* *ArXiv High Energy Physics - Experiment e-prints (Preprint https://arxiv.org/abs/1812.01319)*
- [12] Khattak G, Vallecorsa S and Carminati F 2018 *2018 25th IEEE International Conference on Image Processing (ICIP)* pp 3913–3917 ISSN 2381-8549
- [13] Carminati F, Khattak G and Vallecorsa S 2018 *2018 Computing in High Energy and Nuclear Physics(CHEP)*
- [14] Odena A, Olah C and Shlens J 2016 *ArXiv e-prints (Preprint 1610.09585)*
- [15] Carminati F, Gheata A, Khattak G, Lorenzo P M and Vallecorsa S 2017 *ACAT (Seattle)* URL <https://indico.cern.ch/event/567550/contributions/2627179/>
- [16] Carminati F, Khattak G, Pierini M, Vallecorsafa S, Farbin A, Hooberman B, Wei W, Z M, P Vitoria B, Spiropulu M and Vlimant J 2017 *NIPS* URL https://dl4physicalsciences.github.io/files/nips.dlps.2017_15.pdf
- [17] Boland M J *et al.* (CLIC, CLICdp) 2016 Updated baseline for a staged Compact Linear Collider (Preprint 1608.07537)
- [18] Pierini M *et al.* 2016 The lcd dataset URL <https://indico.hep.caltech.edu/event/102/contributions/41/>
- [19] Hinton G, Srivastava N and Swersky K 2012 Lecture 6a overview of minibatch gradi-ent descent.
- [20] Chollet F *et al.* 2015 Keras <https://github.com/fchollet/keras>
- [21] Abadi M *et al.* 2015 TensorFlow: Large-scale machine learning on heterogeneous systems software available from tensorflow.org URL <https://www.tensorflow.org/>
- [22] Chintala S, Denton E, Arjovsky M and Mathieu M 2016 How to train a gan? tips and tricks to make gans work URL <https://github.com/soumith/ganhacks>
- [23] Vlimant J R *et al.* 2018 *CHEP 2018 conference, in publication*
- [24] Sergeev A and Balso M D 2018 *CoRR* abs/1802.05799 (Preprint 1802.05799) URL <http://arxiv.org/abs/1802.05799>
- [25] Anderson D, Vlimant J and Spiropulu M 2017 *CoRR* abs/1712.05878 (Preprint 1712.05878) URL <http://arxiv.org/abs/1712.05878>
- [26] Vallecorsa S *et al.* 2018 *High Performance Computing* vol 11203 URL https://doi.org/10.1007/978-3-030-02465-9_35
- [27] Fernandes J *et al.* 2018 Activity report HNSciCloud pilot phase
- [28] Merkel D 2014 *Linux J.* **2014** ISSN 1075-3583 URL <http://dl.acm.org/citation.cfm?id=2600239.2600241>
- [29] The Kubernetes authors *Kubernetes Manual* [Online; accessed 31-May-2019] URL <https://kubernetes.io/>
- [30] The Kubeflow authors *Kubeflow* [Online; accessed 31-May-2019] URL <https://www.kubeflow.org/>

Accelerating Recurrent Neural Networks for Gravitational Wave Experiments

Zhiqiang Que*, Erwei Wang*, Umar Marikar*, Eric Moreno†, Jennifer Ngadiuba†, Hamza Javed‡, Bartłomiej Borzyszkowski‡, Thea Aarrestad‡, Vladimir Loncar‡, Sioni Summers‡, Maurizio Pierini‡, Peter Y Cheung*, Wayne Luk*

† California Institute of Technology, Pasadena, CA, USA

‡ European Organization for Nuclear Research (CERN), Geneva, Switzerland,
{jennifer.ngadiuba, sioni.paris.summers, Maurizio.Pierini}@cern.ch

*Imperial College London, UK, {z.que, w.luk}@imperial.ac.uk

Abstract—This paper presents novel reconfigurable architectures for reducing the latency of recurrent neural networks (RNNs) that are used for detecting gravitational waves. Gravitational interferometers such as the LIGO detectors capture cosmic events such as black hole mergers which happen at unknown times and of varying durations, producing time-series data. We have developed a new architecture capable of accelerating RNN inference for analyzing time-series data from LIGO detectors. This architecture is based on optimizing the initiation intervals (II) in a multi-layer LSTM (Long Short-Term Memory) network, by identifying appropriate reuse factors for each layer. A customizable template for this architecture has been designed, which enables the generation of low-latency FPGA designs with efficient resource utilization using high-level synthesis tools. The proposed approach has been evaluated based on two LSTM models, targeting a ZYNQ 7045 FPGA and a U250 FPGA. Experimental results show that with balanced II, the number of DSPs can be reduced up to 42% while achieving the same IIs. When compared to other FPGA-based LSTM designs, our design can achieve about 4.92 to 12.4 times lower latency.

I. INTRODUCTION

Recurrent Neural Networks (RNNs) are a type of architecture specialized for processing ordered data, for example time-series data. These networks have applications in speech recognition [1], DNA sequence analysis, and physics experiments [2, 3]. An exciting physics experiment concerns the detection of gravitational waves, predicted by Albert Einstein a hundred years ago. The first detected wave came from a collision between two black holes, reaching the earth after 1.3 billion years. The detectors at the Laser Interferometer Gravitational-Wave Observatory (LIGO) produce time-series data, as they capture cosmic events such as black hole mergers which happen at unknown times and of varying durations. Accelerating RNN inference using reconfigurable accelerators such as FPGAs would enable sophisticated processing, such as anomaly detection, to run in real time on the data stream from the detector and generate a fast response. Among the many RNN variants, the most popular one is Long Short-Term Memory (LSTM). FPGAs have been used to speed up the inference of RNNs/LSTMs [1, 4, 5, 6, 7], which offer benefits of low latency and low power consumption compared to CPUs or GPUs.

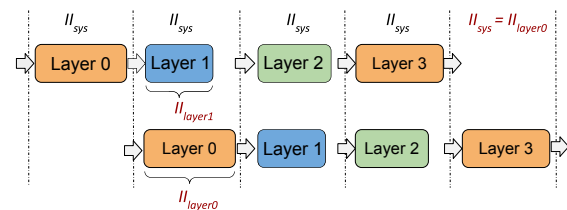


Fig. 1: Unbalanced layer IIs among various cascaded layers in an RNN model

However, existing LSTM accelerators cannot support low-latency and effective multi-layer execution, especially when targeting small LSTM models with requirements of ultra low latency and ultra high throughput for scientific applications. Many existing FPGA-based LSTM accelerators are designed with the same idea as their GPU counterparts, which utilize a single computational engine architecture where the engine is designed to run one block or layer at one time, and the whole network is processed by running the engine repeatedly [5, 6]. Their design consists of arranging computing resources to form a single core with many processing elements, leveraging data level parallelism. For example, Brainwave [5] is a single-threaded neural processing unit (NPU) which has 96,000 processing elements (PEs). However, when the size of the targeted LSTM layer is small, these hardware resources will not be fully utilized, e.g., when targeting a small LSTM layer, the Brainwave hardware utilization is lower than 1% [5], while the utilization of the NPU can be lower than 15% [6]. Moreover, since a single engine is used, the various layers must have the same amount of parallelism which is not flexible to take full advantage of the customizability of FPGAs. Thus, this work applies a layer-wise architecture to map all the LSTM layers on-chip and perform the computation for different layers on their own unit with independent optimization to achieve low latency and high system throughput.

Unlike CNN inference designs [8, 9] which only have forward datapaths and can be fully pipelined, there are feedback datapaths in RNN inference and data dependencies exist between the current timestep and the next timestep. Unrolling the timesteps fully may help, however the sequence length

(timestep) of an LSTM model is usually larger than the number of layers [10], e.g., 1500 timesteps in an LSTM layer in DeepSpeech [11], which makes the full unrolling of timesteps impractical on FPGAs because of the limited hardware resources.

To accelerate an RNN model with multiple LSTM layers, this work proposes coarse grained pipelining with balanced II (initiation interval) to improve system throughput and reduce latency. This is achieved by identifying appropriate reuse factors for each layer, resulting in fast response and enhanced resolution for processing sensor data. It can achieve the best (smallest) system level II for a neural network with multiple LSTM layers on a given FPGA. The II is the number of clock cycles before a unit can accept new inputs and is generally the most critical performance metric in systems [12]. A perfect pipeline has $II = 1$ cycle, as this is required to keep all pipeline stages busy. However, the II of an LSTM layer is generally larger than one because of the data dependencies. For a model with multiple layers in sequence, the initiation interval of this model is decided by the largest II among all the layers [13], as shown in Fig. 1. The unbalanced IIs in various layers result in hardware inefficiency and low throughput. Accelerating a deep LSTM model is challenging since the computation load varies greatly among layers and data dependency exists both time-wise and layer-wise.

Our approach is to ensure all the layer IIs are balanced to eliminate system stall, so that the system becomes a coarse grained seamless pipeline. It increases pipeline parallelism by performing more computations without increasing latency, and without introducing additional memory traffic or storage. Unbalanced IIs in a pipeline is a common issue, but few studies address balancing IIs in the context of accelerating multi-layer DNNs, especially for RNNs/LSTMs. The proposed coarse-grained pipelining is similar to layer parallelism but the granularity in our approach does not need to cover an entire layer. An LSTM layer can still be divided into multiple blocks with pipeline parallelism. In addition, a customizable template for this architecture has been designed, which enables the generation of low-latency FPGA designs with efficient resource utilization using high-level synthesis (HLS) tools. Moreover, We develop an optimization algorithm such that, given the dimensions of the LSTM layers and a resource budget, computes a partitioning of the FPGA resources for an efficient

To the best of our knowledge, this is the first work to propose balancing IIs for a coarse-grained pipelined architecture to enable fast multi-layer LSTM data analysis in gravitational wave experiments. This work could help improve performance of next generation Gravitational Wave detectors.

We make the following contributions in this paper:

- A novel technique for balancing IIs of multi-layer LSTM inference to increase hardware efficiency and system throughput for data analysis in gravitational wave experiments.
- A scalable and low latency LSTM template which enables the generation of low-latency FPGA designs with efficient

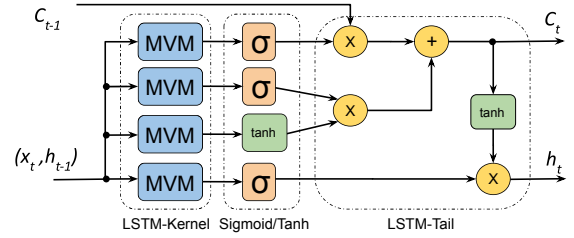


Fig. 2: Structure of an LSTM cell

resource utilization by HLS tools. We open source the templates with some examples¹.

- A comprehensive evaluation of the proposed method and hardware architecture.

The specific RNN layered structure and coefficients are LIGO specific, but the need for low latency would benefit many other applications, especially those requiring real-time response, e.g., low latency would benefit the Large Hadron Collider (LHC) physics [14], adaptive radiotherapy [15] and electronic trading [16]. The proposed techniques can be adapted to address these other applications.

II. BACKGROUND AND PRELIMINARIES

RNNs/LSTMs have been shown to have useful properties with many significant applications. This study follows the standard LSTM cell [4, 5, 6]. 2 shows an LSTM cell. It consists of three main parts. At the front, there are four LSTM gates which perform matrix-vector multiplication (MVM), followed by activation functions. While in the tail, there are a few element-wise operations. The hidden state h_t , which will be fed back from the tail to the front, is produced by the following equations:

$$\begin{aligned} i_t &= \sigma(W_i[x_t, h_{t-1}] + b_i), & f_t &= \sigma(W_f[x_t, h_{t-1}] + b_f) \\ g_t &= \tanh(W_g[x_t, h_{t-1}] + b_u), & o_t &= \sigma(W_o[x_t, h_{t-1}] + b_o) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t, & h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

Here, σ , \tanh and \odot stand for the sigmoid function, the hyperbolic tangent function and element-wise multiplication respectively. i, f, g and o represent the input, forget, input modulation and output gate respectively. The input modulation gate is often considered as a sub-part of the input gate. The input vector and hidden vector are combined so that W represents the weight matrix for both vectors. Bias term is represented as b . The output c_t is the internal memory cell state and h_t is the output of the cell, also called the hidden vector, which is passed to the next timestep or next layer.

III. DESIGN AND OPTIMIZATION METHODOLOGY

This section analyzes unbalanced II issues and introduces several optimizations for multi-layer RNN designs. We define a few parameters, as shown in Table I for later calculations.

¹https://github.com/walkieq/RNN_HLS

TABLE I: System Parameters

II_{sys}	System initiation interval
TS	Timestep number
ii_N	Timestep loop initiation interval in the LSTM layer N
II_N	Initiation interval for layer N
LT_N	Latency of a single timestep loop for layer N
LT_α	Latency of the unit α ; α could be mult / mvm / tail / σ
x_t	The input vector x at timestep t
h_t	The hidden vector h at timestep t
W_x	LSTM gates weight matrix for input vector.
W_h	LSTM gates weight matrix for hidden vector.
L_x	Number of elements in the input vector x
L_h	Number of elements in the hidden vector h
R_x	Reuse factor for MVM involving LSTM input vector x_t
R_h	Reuse factor for MVM involving LSTM hidden vector h_t
R_t	Reuse factor for LSTM tail unit

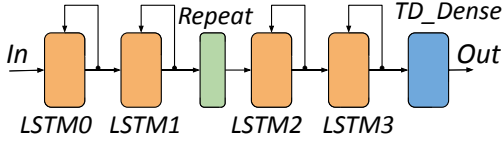


Fig. 3: Overview of the LSTM-based autoencoder

A. LSTM-based autoencoder for gravitational wave detection

Fig. 3 shows an overview of the LSTM-based autoencoder used for gravitational wave detection. The models and the dataset are available on GitHub [17, 18]. The autoencoder consists of two components, an encoder and decoder. The encoder learns to transform data from the input layer into a latent-space representation, which acts as a data "bottleneck". The decoder then reconstructs the output of the reduced latent representation as close as possible to its original input. When the error between input and reconstructed values is high, the input is flagged as anomalous. In this work, an LSTM-based autoencoder is used as an unsupervised prediction model to detect the anomalies for gravitational waves. This works by only training the LSTM-autoencoder to encode and decode normal background conditions at the LIGO interferometers. When an event containing a gravitational wave passes through the autoencoder, the model cannot encode and decode the additional strain provided by the gravitational wave. Both the encoder and decoder have two LSTM layers. A TimeDistributed dense layer is applied before the data output.

B. System II for multi-layer LSTM networks

Accelerating a deep LSTM model which has multiple layers is challenging since the computation varies greatly among layers and data dependencies exist both time-wise and layer-wise. An efficient technique to improve throughput and reuse computational resources is to pipeline hardware units. If each input can overlap with itself, we can achieve simultaneously inference parallelism within a run by coarse grained pipelining as shown in Fig. 1.

However, a naive implementation can result in a large number of idle cycles due to inter-layer dependencies since the

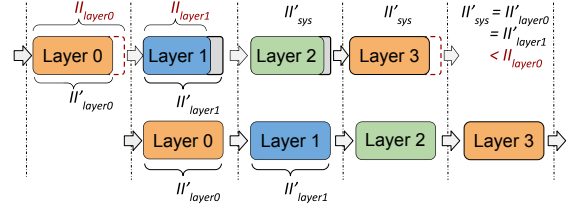


Fig. 4: Overview of the method used to balance IIs

pipeline is not seamless; a particular layer might stall until the previous layer finishes. The unbalanced IIs in various layers results in hardware inefficiency and low system throughput. Typically, the particular layer with the largest II should be optimized since it dominates the system II. Generally, the II cycles can be reduced if more hardware resources are allocated to that particular layer by adding more parallelisms. So the targeted layer should be allocated as many hardware resources as possible. However, the hardware resources on a given FPGA is limited, which means that the other layers may occupy less hardware resources. When the resources for a layer decrease, the II of that layer will increase. Then this layer may become the one that has the largest II and dominates the design. Thus, the optimal case is that all the layers have the same II, in which scenario the design utilizes the hardware resources efficiently and achieves the highest system throughput as shown in Fig. 4.

Besides, we find that we do not need to unroll every unit in order to achieve the lowest II. Some hardware resources can be saved from the units which do not require full unrolling. And then these saved hardware resources can be reallocated to the other units which dominate the system to achieve low initiation intervals. As shown in Fig. 4, the hardware resources for layer 1 can be reduced so that the saved resources can be reallocated for layer 0. The II_{layer1} is increased to II'_{layer1} while the II_{layer0} which is the largest can be reduced to II'_{layer0} so that the final system II_{sys} can be reduced.

Partitioning FPGA resources to enhance throughput has been studied for CNNs [8, 9, 19, 20] but they do not touch the RNNs and the recurrent nature as well as the data dependencies in RNN computations, which are absent from CNNs. We develop an optimization algorithm such that, given the dimensions of the LSTM layers and a resource budget, computes a partitioning of the FPGA resources for an efficient and balanced high-performance design. Our algorithm runs in seconds and produces a set of reuse factors [14]. We then use these factors to parameterize an LSTM template design specified using HLS to form a complete multi-layer LSTM implementation. Since all the layers have the same II, we only need to focus on the optimization for a single LSTM layer. The layer II and system II are

$$II_N = ii_N \times TS \quad (1)$$

$$II_{sys} = \max(II_0, II_1, \dots, II_N) \quad (2)$$

The original II_N should be $II_N = ii_N \times TS + (LT_N - ii_N)$. However, the extra $(LT_N - ii_N)$ cycles can be eliminated after using the rewind for Vivado_HLS `#pragma pipeline`.

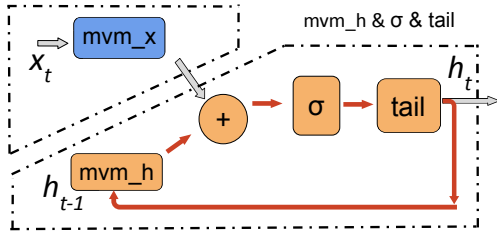


Fig. 5: An LSTM layer after performing the transformation

The `rewind` is an optional keyword that enables rewinding, or continuous loop pipelining with no pause between the end of one loop iteration and the start of the next iteration. So the proposed balancing method has two benefits. First, it improves throughput due to pipelining. Second, it reduces system latency since if the LSTM loop initiation interval, ii_N , can be reduced by 1 cycle, then the system latency can be reduced by TS cycles in total according to Equation (1).

C. The II of a single LSTM layer

This work splits one LSTM layer into two sub-layers. The first one is the `mvm_x` which has no data dependencies and performs MVM operations for the LSTM gates involving the input vectors while the second one includes all the others which form a loop with data dependencies, as shown in Fig. 5. For accelerating LSTM layers used for gravitational wave detection, the system is designed to achieve the average latency (system II) as small as possible. To achieve the lowest system II, fully unrolling the neural network model is an effective method which utilizes a multiplier only once in the computation of a layer. E.g., a fully connected (FC) layer with input size num_in and output size num_out can achieve the lowest latency if there are $num_in \times num_out$ multipliers. This is the most parallel and fast way a layer can be computed. It has been demonstrated in the HLS4ML based DNN designs for particle physics [14]. However, unlike forward computation in the FC layers used in the design of [14], there are data dependencies in LSTM computations.

After we have split the LSTM layer into two sub-layers, the two can be pipelined as shown in Fig. 6. According to the discussion in Section III-B, the optimal case is when the two sub-layers have the same II . Since the second sub-layer is complex and its II is usually larger than the one of the first sub-layer, the parallelism for the first sub-layer does not need to be as large as possible, resulting in a reduction of the number of multipliers needed to process the `mvm_x` unit. The saved multipliers can then be reallocated for other layers to achieve a lower system II. Reducing the parallelism of `mvm_x` does not hurt the system latency. Normally, each input vector can finish the calculation in the shadow region of processing the h_t because of the pipelining. Besides, the cycles for processing the first `mvm_x` can be eliminated when calculating the layer II because of the keyword of `rewind` in Vivado HLS.

While the second sub-layer may seem complex, if the design is split into more sub-layers, these sub-layers cannot be coarse grained pipelined. The reason is that the start of the next

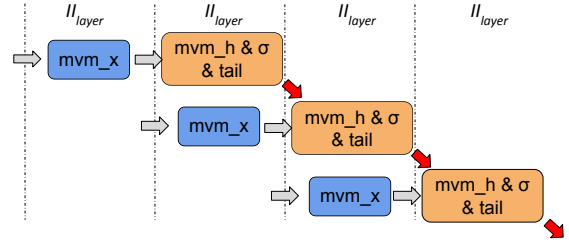


Fig. 6: Coarse grained pipelining in an LSTM layer

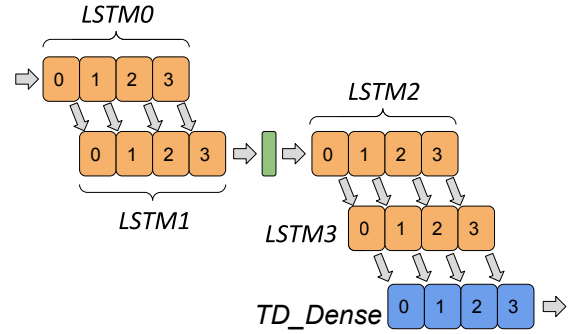


Fig. 7: Timestep overlapping

iteration needs the result from the current iteration, as shown by the red arrows in Fig. 6

D. Overlapping the computations in cascaded LSTM layers

In the proposed coarse grained pipelining, the processing of the cascaded LSTM layers can be overlapped. The second layer does not need to wait for the whole sequence of hidden vectors to be ready. Just one hidden vector from the former LSTM layer is sufficient to start the calculation of the next LSTM layer as shown in Fig. 7. It helps to reduce the overall system latency. It has to be noted that the LSTM2 can only start after the LSTM1 calculation is completed, since only the last timestep hidden vector is returned in LSTM1, which is decided by the structure of the autoencoder.

IV. IMPLEMENTATION

A. HLS implementation

This work maps all the layers on-chip and different layers run in a fashion of coarse grained pipelining to increase the system throughput. Besides, this work always seeks to achieve extremely low latency by utilizing as many hardware resources as possible. However, because of the data dependencies between different timesteps in LSTM calculation, the initiation interval is typically larger than 1. In this case, HLS will automatically increase the initiation interval until it can find a feasible schedule. For complex codes it is common to partition functionality into multiple modules, streaming data between them through explicit interfaces. Smaller components are more modular, making them easier to reuse, debug and verify. The effort required by the HLS tool to schedule code sections increases dramatically with a large number of operations that

need to be considered for the dependency and pipelining analysis. Scheduling logic in smaller chunks is thus beneficial for compilation time and sometimes also for system latency. Our experiments show that inlining every function, especially the mvm_x and mvm_h in the LSTM gates, brings large II when the involved matrices are large.

The trade-off between latency, throughput and FPGA resource usage is determined by the parallelization of the inference calculation. This work adopts the reuse factor used in [14] to fine tune the parallelism, which is configured to set the number of times a multiplier is used in the computation of a module. In one extreme, all multiplications can be performed simultaneously using a maximal number of multipliers, while alternatively in the other extreme, one can use only one multiplier and perform the multiplications sequentially; between these extremes the user can fine tune algorithm throughput versus resource usage. With a reuse factor of one, the computation is fully parallel. With a reuse factor of R , $\frac{1}{R}$ of the computation is done at a time with a factor of $\frac{1}{R}$ fewer multipliers.

The total number of multiplications required to infer a given LSTM layer using 16-bit is:

$$DSP_{layer} = \frac{4 \times Lx \times Lh}{R_x} + \frac{4 \times Lh^2}{R_h} + 4 \times Lh \quad (3)$$

$$DSP_{model} = \sum_{layer=1}^N DSP_{layer} \leq DSP_{total} \quad (4)$$

Compared with the number of multipliers used in LSTM gates, the one required in the LSTM tail unit is small so the R_t is set to 1. Otherwise, $\frac{4 \times Lh}{R_t}$ should be used in Equation (3). Besides, since the LSTM cell status, c_{t-1} , is represented in 32-bit, the $f_t \times c_{t-1}$ in the LSTM tail needs two Xilinx DSPs to implement one multiplier. Thus, the LSTM tail unit consumes $4 \times Lh$ DSPs. The activation function sigmoid is implemented using BRAM-based lookup tables with a range of precomputed input values. The hyperbolic tangent function is implemented as piecewise linear function [21, 22] to reduce the latency. In the next subsection, we introduce our method for determining R_x and R_h with a given FPGAs.

B. Design space exploration

FPGA multipliers are pipelined; therefore, the latency of one MVM computation, LT_{mvm} , is approximately

$$LT_{mvm} = LT_{mult} + (R - 1) \times II_{mult} \quad (5)$$

where LT_{mult} is the latency of the multiplier, II_{mult} is the initiation interval of the multiplier, which is one cycle in this work. Equation (5) is approximate because, in some cases, additional cycles could be introduced for signal routing. Besides, the Vivado HLS tool will replace a multiplier by an adder when the corresponding weight is simple.

As we discussed in Section III, the optimal case is that the two sub-layers in an LSTM layer have the same II, which results in Equation (6).

$$II_{sublayer} = LT_{mvm_x} = LT_{mvm_h} + LT_{\sigma} + LT_{tail} \quad (6)$$

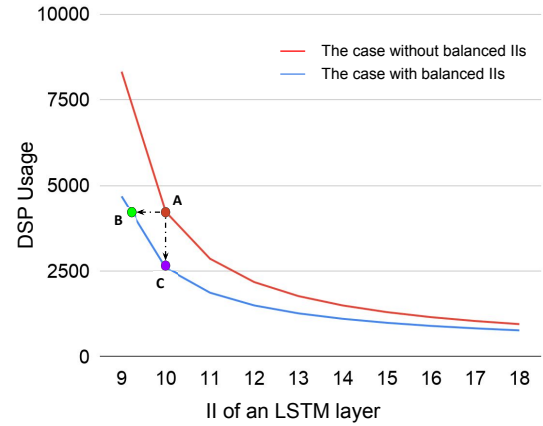


Fig. 8: Pareto frontier

where LT_{mvm_x} and LT_{mvm_h} are the latencies of the MVM units involving input vectors x and hidden vectors h respectively. LT_{σ} is the latency of the sigmoid function and LT_{tail} is the latency of the LSTM tail unit. These units are shown in Fig. 5. If we substitute the Equation (5) into Equation (6) and then we get

$$R_x = R_h + LT_{\sigma} + LT_{tail}. \quad (7)$$

The architecture designed in this section serves as a baseline to deploy our methodology, whose goal is to find Pareto-optimal sets of reuse factors of the proposed accelerator to achieve a good trade-off between our design objectives, which are hardware resources, energy, and performance. To achieve low latency, the reuse factors should be as small as possible since when they decrease the parallelism increases, leading to high throughput. However, when reuse factors decrease, the required hardware resources increase and may easily exceed the number of total hardware resources on an FPGA. If we substitute the Equation (7) and Equation (3) into Equation (4), we can get a quadratic inequality of R_h , which gives the minimum R_h for a given number of DSPs.

Fig. 8 illustrates the exploration results of an LSTM layer with $(Lx, Lh) = (32, 32)$ and different values of reuse factors, which are from 1 to 10. The red line represents the cases with the same R_x and R_h . The blue line shows the cases with balanced IIs, where R_x and R_h meet the constraint in Equation (7). For simplicity, LT_{σ} is set to 3 and the LT_{tail} is 5. Please note that LT_{σ} and LT_{tail} are both system dependent and can vary depending on clock frequency and FPGA devices. After balancing IIs, the Pareto frontier moves from red line to blue line. With the proposed technique, we can achieve a same II with less DSP usage (from point A to point C) or we can achieve a better II (from point A to point B) as shown in Fig. 8.

V. EVALUATION AND ANALYSIS

This section presents the performance of the RNN models developed for gravitational wave detection on two generations of Xilinx FPGAs demonstrating the scalability of the proposed optimization.

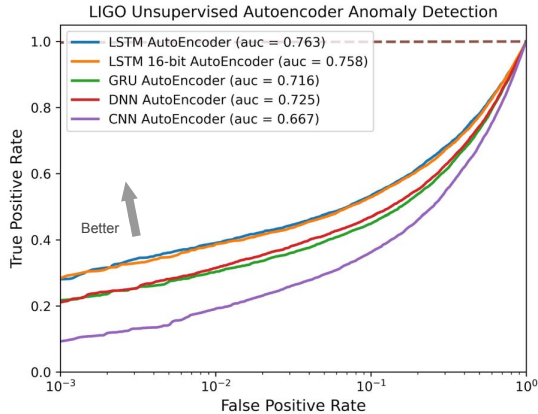


Fig. 9: AUCs and ROC curves for various autoencoders

A. Experimental setup

Simulated gravitational waves are generated using the GGWD library [23]. Noise is generated at a specified power spectral density (PSD) to mimic normal detector background conditions using PyCBC [24]. This approach to simulated data generation ignores glitches, blips, and other transient sources of detector noise, though this algorithm can be re-purposed for identifying these detector glitches with unsupervised methods. Signal events are generated simulating GW production from compact binary coalescences using PyCBC [24], which itself uses algorithms from LIGO’s LAL Suite [25]. Signal events containing GWs were created overlaying simulated GWs, with the SEOBNRv4 Approximant, on top of detector noise. This provides an analogous situation to a real GW, in which the strain from the incoming wave is recorded in combination with the normal detector noise. Data are then whitened and band-passed, then normalized. The training set has 240K gravitational wave events. The validation set and test set have 60k and 50k events respectively. To study the performance and limitations of the proposed optimizations and hardware architecture, the designs are implemented using Vivado HLS 19.2. Two generations of Xilinx FPGAs, the ZYNQ 7045 and U250, are evaluated and compared with previous work.

B. Model accuracy

To quantify the performance of the autoencoders for anomaly detection implemented by various neural networks, we use the AUC metric, or area under the Receiver Operating Characteristic (ROC) curve, as shown in Fig. 9, with higher AUC corresponding to better performance. The default timestep [17] of 100 is used. AUC is a common metric for evaluating models as it is classification-threshold-invariant. The threshold for flagging an anomaly by its loss spike can be calculated by setting a false positive rate (FPR) on noise events. The higher the threshold for detecting an anomaly, the lower the FPR will be. This threshold can be used to calculate the corresponding true positive rate (TPR) on signal events. We observe that the LSTM-based autoencoder has the highest AUC, and hence the best performance, among the

TABLE II: Performance comparison of the FPGA designs

	Z1	Z2	Z3	U1	U2	U3
FPGA	Zynq 7045			U250		
DSP total	900			12,288		
R_h	1	2	1	1	1	4
R_x	1	2	9	1	9	12
LUT used	45k (21%)	45k (21%)	43k (20%)	449k (26%)	463k (27%)	516k (30%)
DSP used	1,058 (118%)	578 (64%)	744 (83%)	11,123 (91%)	9,021 (73%)	2,713 (22%)
ii_{layer} cycles	9	10	9	12	12	13
II_{layer} cycles	72	80	72	96	96	104

unsupervised designs [17] with various NN layers, including GRU, CNN and DNN. Additionally, Qkeras [26] is used to quantize the LSTM-based autoencoder to 16-bit. We find this precision to have a negligible effect on the NN performance.

C. Performance and efficiency comparison

To illustrate the benefits of our proposed approach, two LSTM-based autoencoders are evaluated. The first one is a small autoencoder which has the same architecture as the one used in gravitational wave detection described in Section III-A but only has two LSTM layers, each having 9 hidden units. The results are shown in Table II. It is running at 100MHz with 8 timesteps. The weights and input are 16 bits. The bias and LSTM cell status are both 32 bits to keep the accuracy. To achieve the lowest latency, the reuse factors should be set to one so that all the operations are unrolled, e.g., the design Z1 in Table II. However the required number of DSPs exceed the one of the total DSPs on this FPGA. One may increase the re-use factor from one to two to fit the design into this FPGA device. However the cost is that now the timestep loop initiation interval, ii_{layer} , increases by one cycle which results in T/S cycles increase for the layer II, e.g., the design Z2 in Table II. However, it is not necessary to fully unroll all units in order to achieve the lowest latency. Some hardware resources can be saved from the units which do not require full unrolling and can be allocated to the other units which are dominating to achieve low latency.

With the proposed balancing of IIs, some of the DSPs resources can be rearranged from implementing mvm_x to mvm_h to achieve lower latency, e.g., the design Z3. So this design can still achieve the lowest II like the case with full unrolling, and it is still able to fit in this FPGA device as shown in Table II, showing the benefits of balanced IIs. Besides, with heterogeneous reuse factors, the parallelism of the design can be fine-tuned to make the trade-off between latency, throughput and FPGA hardware resources as shown in Fig. 10. With the balanced II, the number of DSPs can be reduced up to 42% while achieving the same IIs.

Besides, to show the adaptability of our technique, the nominal autoencoder [17] developed for gravitational wave detection is implemented using a larger FPGA, U250, running at 300MHz with 8 timesteps. It has four LSTM layers which have

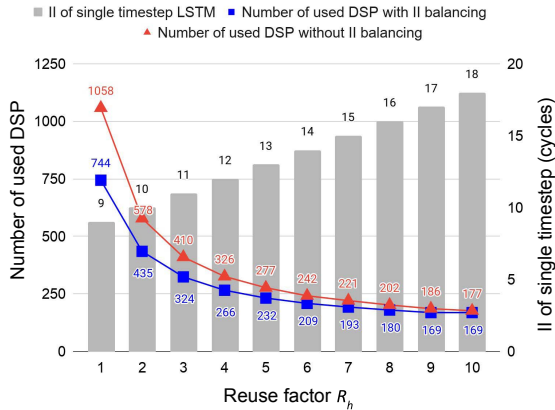


Fig. 10: Initiation intervals and DSP numbers using various reuse factor R_h on Zynq 7045

a number of hidden units equal to 32, 8, 8, 32 respectively and one TimeDistributed dense layer before the output. Since the U250 has 12,288 DSPs, the whole fully unrolled autoencoder can be fit into this FPGA with both R_x and R_h set to one, shown as the design U1 in Table II. With our technique of balancing IIs, the DSPs of the design U2 can be reduced by 2102 while achieving the same design IIs and same design throughput. After HLS synthesis, the II is slightly larger than the one estimated by the performance model since the DSP usage is very high and some additional cycles are incurred for signal routing. The design U3 is an interesting version with reuse factors (R_h , R_x) as (4, 12). It achieves a slightly worse II, as shown in Table II, however it consumes 3.3 and 4.1 times less DSPs than design U2 and design U1 respectively. Sometimes, the user may only care about the latency of the LSTM running on the FPGAs, then they can just take the point that gives them the lowest latency with most resources. However, if the user can bear with a slightly reduced latency then they can choose a smaller and cheaper FPGA as shown in Table II. One can choose between using less resources but increasing latency and vice versa. Please note because of the data dependence, the iv_{layer} could be hard to optimize to 1. However, it could be further optimized to a smaller value using fast multipliers or fast activation functions. We leave that for future work since it has a limited impact on the conclusions we draw from our study in this paper.

To compare the performance of the proposed design on FPGA with other platforms, we implement the same LSTM-based autoencoder on Intel CPU and NVIDIA GPU. The AVX2 vector instructions are enabled for the CPU while the CuDNN libraries are enabled for the GPU. Compared with the designs running on CPU and GPU, our FPGA design runs much faster, as shown in Table III. We are processing each inference sequentially (batch 1) since requests need to be processed as soon as they arrive. The GPUs provide large throughput by running many parallel inferences but may not perform well when the batch is small, especially there are data dependencies in LSTMs. However, FPGAs work fast on a single inference with a fully unrolled tailor-made design.

TABLE III: Latency comparison of the FPGA design versus CPU and GPU

	CPU	GPU	This work
Platform	Intel E2620	TITAN X	U250
Precision	F32	F32	16 Fixed
Latency	39.7 ms	32.1 ms	0.40 us

TABLE IV: Comparison with previous FPGA-based LSTM designs for anomaly detection and physics

	[28], 2018	[27], 2020	This work	This work
FPGA	Kintex7 K410T	KU115	U250	U250
Model	Single Layer	Single Layer	Single Layers	Four Layers
Application Domain	Anomaly Detection	Physics	-	Anomaly Detection
LSTM hidden units L_h	32	16	32	32,8,8,32
DSPs	1091	2374	2221	9021
Preci. (bits)	16 fixed	16 fixed	16 fixed	16 fixed
Freq. (MHz)	155	200	300	300
Latency (us)	4.27	1.35	0.343	0.867

Some other HLS-based RNN/LSTM accelerators on FPGAs are compared with ours in Table IV. In this table, we focus on latency since the throughput, power or power efficiency of the other designs are not reported. Our design achieves 4.92 to 12.4 times lower latency compared to the state-of-the-art FPGA designs targeting anomaly detection. Our single-layer design, with a similar amount of DSP resources to another design [27], is 3.9 times faster as shown in Table IV. Note that because of the structure of an autoencoder, the processing of the encoder and the decoder cannot be overlapped, which increases the end-to-end latency of the design. Nevertheless, we still achieve better latency than the others which contain only one LSTM layer. Moreover, while the other designs report Vivado HLS synthesis latency, we report the RTL co-simulation latency which is likely to be more accurate.

VI. RELATED WORK

A latency-optimized LSTM-based anomaly detection is proposed in [28] on FPGAs and we achieve 4.9 times faster than it. [14] proposes the HLS4ML tool and introduces a deep FC-layer model for substructure-based jet tagging in LHC physics. [27] introduces HLS LSTMs for the same physics problem.

Partitioning FPGA resources to improve throughput has been studied for CNNs [8, 9, 19, 20], but they do not touch the RNNs and the recurrent nature and data dependency in RNN computations which are absent in CNNs. The FiC-RNN [29] proposes to accelerate multi-layer RNNs using an FPGA cluster, in which each RNN layer occupies a single FPGAs. The authors in [10] put each LSTM layer on each multi-core to achieve coarse grained pipelining. In [30, 31, 32, 33], the batching technique is used to improve the hardware throughput and utilization for LSTM inferences. However, latency can suffer since different inputs may not come at the same time, meaning that a newly arrived request has to wait until the batch is formed, which imposes a significant latency penalty.

Some of the previous studies [1, 34, 35, 36, 37, 38] are focusing on weight pruning and model compression to achieve good performance and efficiency. Some researchers use low bitwidth, even binarized, datapaths [30, 39, 40] and investigate the trade-off between precision and performance. These studies are orthogonal to our proposed approach and hardware architecture. These techniques can be complementary to our approach to achieve even lower latency of RNN inferences on FPGAs.

VII. CONCLUSIONS AND FUTURE WORK

This paper aims to pioneer new data analysis architectures to support next-generation low-latency anomaly detection on time series data, relevant to many fundamental physics experiments including gravitational wave detection. We present a novel approach for minimizing the initiation intervals for the execution of a multi-layer LSTM network by optimizing the reuse factors for each layer. Results show latency reduction of up to 12.4 times over the existing FPGA-based LSTM design. Current and future work includes exploring the use of new FPGA resources such as the AI Engines [41] and the AI Tensor Blocks [42], and incorporating the proposed approach into the design of the data analysis architecture for next-generation gravitational wave detectors.

ACKNOWLEDGEMENT

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1, and EP/S030069/1), CERN and Xilinx is gratefully acknowledged. We thank Prof. Zhiru Zhang and Yixiao Du for their help and advice.

REFERENCES

- [1] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 75–84.
- [2] A. Schmitt *et al.*, "Investigating Deep Neural Networks for Gravitational Wave Detection in Advanced LIGO Data," in *Proceedings of the 2nd International Conference on Computer Science and Software Engineering*, 2019.
- [3] Y.-C. Lin and J.-H. P. Wu, "Detection of gravitational waves using Bayesian neural networks," *Physical Review D*, vol. 103, no. 6, p. 063034, 2021.
- [4] Y. Guan *et al.*, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017.
- [5] J. Fowers, K. Ovcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 1–14.
- [6] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar *et al.*, "Why compete when you can work together: Fpga-asic integration for persistent rns," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 199–207.
- [7] Z. Que, H. Nakahara, E. Nurvitadhi, H. Fan, C. Zeng, J. Meng, X. Niu, and W. Luk, "Optimizing Reconfigurable Recurrent Neural Networks," in *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 10–18.
- [8] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.
- [9] H. Nakahara *et al.*, "High-Throughput Convolutional Neural Network on an FPGA by Customized JPEG Compression," in *28th FCCM*. IEEE, 2020.
- [10] L. Peng *et al.*, "Exploiting Model-Level Parallelism in Recurrent Neural Network Accelerators," in *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*. IEEE, 2019.
- [11] A. Hannun *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.
- [12] Xilinx, "SDSoC Profiling and Optimization Guide."
- [13] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of High-Level Synthesis Codes for High-Performance Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2020.
- [14] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.
- [15] D. Thorwarth and D. A. Low, "Technical Challenges of Real-Time Adaptive MR-Guided Radiotherapy," *Frontiers in Oncology*, vol. 11, p. 332, 2021.
- [16] S. Denholm *et al.*, "Low latency FPGA acceleration of market data feed arbitration," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 2014.
- [17] E. Moreno. [Online]. Available: <https://github.com/eric-moreno/Anomaly-Detection-Autoencoder>
- [18] E. Moreno *et al.* in preparation.
- [19] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.
- [20] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.
- [21] B. Moons *et al.*, "Minimum energy quantized neural networks," in *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2017.
- [22] E. Azari and S. Vrudhula, "An energy-efficient reconfigurable lstm accelerator for natural language processing," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 4450–4459.
- [23] "Generate Gravitational-Wave Data (GGWD)," <https://github.com/timothygebhard/ggwd>, 2019.
- [24] A. Nitz *et al.*, "gwastro/pycbc: Pycbc release v1.16.9," Aug. 2020. [Online]. Available: <https://doi.org/10.5281/zenodo.3993665>
- [25] LIGO Scientific Collaboration, "LIGO Algorithm Library - LALSuite," free software (GPL), 2018.
- [26] C. Coelho, A. Kuusela, S. Li, H. Zhuang, T. Aarrestad, V. Loncar, J. Ngadiuba, M. Pierini, A. Pol, and S. Summers, "Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders," *arXiv preprint arXiv:2006.10159*.
- [27] R. Rao, "Implementation of Long Short-Term Memory Neural Networks in High-Level Synthesis Targeting FPGAs," Ph.D. dissertation, 2020.
- [28] Y. H. Lee, D. J. Moss, J. Faraone, P. Blackmore, D. Salmund, D. Boland, P. H. Leong *et al.*, "Long Short-Term Memory for Radio Frequency Spectral Prediction and its Real-Time FPGA Implementation," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 1–9.
- [29] Y. Sun and H. Amano, "Fic-rnn: A multi-fpga acceleration framework for deep recurrent neural networks," *IEICE Transactions on Information and Systems*, vol. 103, no. 12, pp. 2457–2462, 2020.
- [30] V. Rybalkin *et al.*, "FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs," in *28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018.
- [31] Z. Que *et al.*, "Efficient Weight Reuse for Large LSTMs," in *30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2019.
- [32] A. Boutros *et al.*, "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs," in *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2020.
- [33] Z. Que *et al.*, "Mapping Large LSTMs to FPGAs with Weight Reuse," *Journal of Signal Processing Systems*, 2020.
- [34] Z. Chen, A. Howe, H. T. Blair, and J. Cong, "Clink: Compact lstm inference kernel for energy efficient neurofeedback devices," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2018, pp. 1–6.
- [35] S. Cao *et al.*, "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019.
- [36] R. Shi *et al.*, "E-LSTM: Efficient inference of sparse LSTM on embedded heterogeneous system," in *56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019.
- [37] G. Nan *et al.*, "DC-LSTM: Deep Compressed LSTM with Low Bit-Width and Structured Matrices," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020.
- [38] Z. Chen *et al.*, "BLINK: bit-sparse LSTM inference kernel enabling efficient calcium trace extraction for neurofeedback devices," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020.
- [39] E. Nurvitadhi *et al.*, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," in *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016.
- [40] V. Rybalkin and N. Wehn, "When Massive GPU Parallelism Ain't Enough: A Novel Hardware Architecture of 2D-LSTM Neural Network," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 111–121.
- [41] "Xilinx AI Engines and Their Applications," in *WP506(v1.1)*, July 10, 2020.
- [42] M. Langhammer *et al.*, "Stratix 10 NX Architecture and Applications," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

Jet Single Shot Detection

Adrian Alan Pol^{1*}, Thea Aarrestad¹, Katya Govorkova¹, Roi Halily⁴, Tal Kopetz⁴, Anat Klemptner⁴, Vladimir Loncar^{1,3}, Jennifer Ngadiuba², Maurizio Pierini¹, Olya Sirkin⁴, and Sioni Summers¹

¹European Organization for Nuclear Research (CERN), Geneva, Switzerland

²California Institute of Technology, Pasadena, USA

³Institute of Physics Belgrade, Belgrade, Serbia

⁴CEVA, Herzliya, Israel

Abstract. We apply object detection techniques based on Convolutional Neural Networks to jet reconstruction and identification at the CERN Large Hadron Collider. In particular, we focus on CaloJet reconstruction, representing each event as an image composed of calorimeter cells and using a Single Shot Detection network, called Jet-SSD. The model performs simultaneous localization and classification and additional regression tasks to measure jet features. We investigate Ternary Weight Networks with weights constrained to $\{-1, 0, 1\}$ times a layer- and channel-dependent scaling factors. We show that the quantized version of the network closely matches the performance of its full-precision equivalent.

1 Introduction

The majority of particles produced at the CERN Large Hadron Collider (LHC) are unstable and immediately decay in different particles. When quarks and gluons are produced, QCD confinement prevents them from travelling across the detector. Instead, they shower other quarks and gluons, eventually hadronizing into particles. The result of this process is a *jet*, a collimated showers of particles with adjacent trajectories. Jets are key in many physics analyses done on the data collected by the LHC experiments, e.g. [1–4]. The procedure of classifying the origin of these jets, i.e. the nature of the particle that initiated the shower, known as *jet tagging* [5–8] is a fundamental task for collision reconstruction at the LHC. Similarly, it is important to determine the jet energy, momentum, and mass.

Traditional approaches to jet tagging rely on features designed by experts that detect characteristic energy deposition patterns [9–17]. In recent years, several studies projected the lower level detector measurements of the emanating particles into an image, known as *jet images*. This opened the path to applying computer vision and machine learning techniques [18–30], with particular attention to Convolutional Neural Networks (CNNs) [31].

The goal of this paper is to extend this approach to the problem of jet clustering, e.g., to replace FastJet [32] on computing architectures where parallel computing is more adequate. At the same time, we aim at demonstrating that jet clustering, mass measurement, and tagging could all be handled simultaneously. Besides the practical advantages of a single-shot

*e-mail: adrianalan.pol@cern.ch

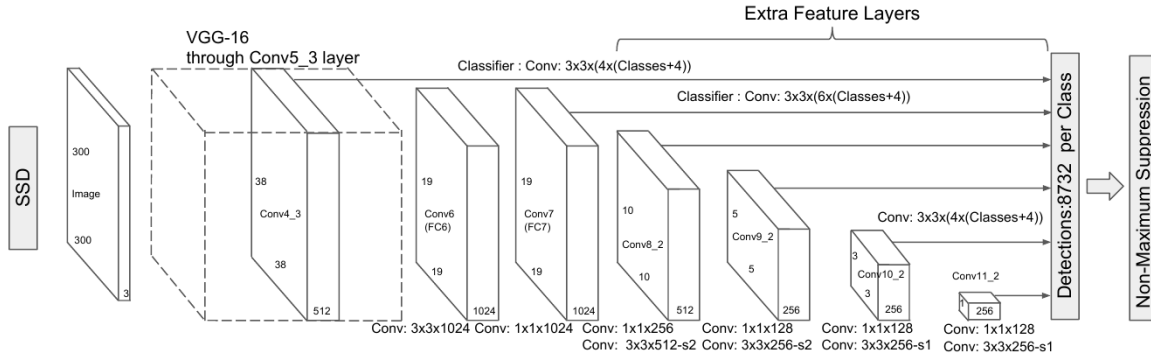


Figure 1. The architecture of the SSD network, proposed by [49].

approach to jet reconstruction, one would benefit from mutual learning when accomplishing more tasks at once. For instance, a classifier and a regression running at once can learn that calibration constants depend on the nature of the jet, an issue that is not handled with ad-hoc post-processing (see [33] as an example).

With the luminosity increase expected in the future, traditional reconstruction algorithms might suffer from execution time scaling worse than linearly with the number of collisions happening in one bunch crossing. For this reason, it is worth investigating solutions that could execute many tasks at once, while retaining accuracy and benefiting from the additional speed up offered by parallel computing architectures. Deep neural networks, such as those used for computing vision tasks, are an obvious candidate.

On the other hand, memory consumption is also an important aspect to keep under control. To this purpose, we investigate the use of extreme quantization, up to ternary precision, which is applied already at training time to retain accuracy.

The remainder of this paper is structured as follows. In Sections 2 and 3 we briefly review single-shot detection and efficient model design techniques. In Section 4 we introduce the dataset and in Section 5 model architecture, implementation details and training procedure. Finally, in Section 6 we present the evaluation metric and results.

2 Single-shot object detection

Object detection is a fundamental task in computer vision. It is defined as the classification of objects from predefined categories in the image along with their precise spatial locations. The spatial location and extent of an object can be defined coarsely using a bounding box, which is an axis-aligned rectangle tightly bounding the object. Instead, a precise pixel-wise segmentation mask corresponds to the segmentation task.

Starting from Overfeat Network [34], the field of object detection focused on using primarily CNNs as a building block, achieving state-of-the-art results in tasks such as face [35] or pedestrian detection [36]. For a general survey on this subject, see [37, 38].

The deep learning-based object detection models are divided into two groups: one [39–43] or two [44–48] stage detectors. Two-stage detectors tend to achieve better accuracy, while one-stage detectors are simpler and faster, hence more suitable to online tasks.

The Single Shot Multibox Detector (SSD) [49], shown in Figure 1, is a simple one-stage, anchor-based detector. First, a set of default regions in an image with a fixed shape and size is predefined to discretize the output space of bounding boxes, called anchors. These anchors have a diverse set of shapes to detect objects with different dimensions, i.e multiple scales and aspect ratios. At each location, the same amount of anchors is defined. Based on the

ground truth, the object locations are matched with the most appropriate anchors to obtain the supervision signal for the anchor estimation.

During training, each anchor is refined by four box coordinates offsets (width, height, x and y) optimized by localization loss (a smooth L1 loss) and predict the categorical probabilities (including background), optimized by classification loss (categorical cross-entropy). To avoid a huge number of negative proposals dominating training gradients, hard negative mining is used to train the network, which fixes the foreground and background ratio.

The SSD architecture is fully convolutional, with initial layers based on a pre-trained backbone architecture, such as VGG-16 [50], followed by extra convolutional layers, progressively decreasing in size. The information in the last layer may be too coarse spatially to allow precise localization and at the same time, detecting large objects in shallow layers is non-optimal without large enough receptive fields. SSD performs detection over multiple scales by operating on multiple feature maps, i.e. at different depths of the network. Each of these feature maps is responsible for detecting objects according to their receptive field.

The final prediction is made by merging all detection results from different feature maps followed by a non-maximum suppression (NMS) step to produce the final detection. NMS removes duplicate predictions originating from multiple anchors.

3 Efficient inference

Network compression [51] is a common technique to reduce the number of operations, model size, energy consumption, and over-training of deep neural networks. As neural network synapses and neurons can be redundant, compression techniques attempt to reduce the total number of them, effectively reducing multipliers. Several approaches have been successfully deployed without much loss in accuracy, including parameter pruning [52–54] (selective removal of parameters based on a particular ranking and regularization), low-rank factorisation [55–57] (using matrix decomposition to estimate informative parameters), compact network architectures [58–61], and knowledge distillation [62] (training a compact network with distilled knowledge of a large network).

A particularly successful compression technique is weight quantization [63–71], which is reducing the precision of operations and operands. It has been observed that 32-bit floating-point calculations or full-precision (FP) are not needed at inference to achieve optimal performance. Thus, reducing the precision of the calculations, i.e. weights and biases, has little impact on performance compared to speed up and resource usage. This includes moving away from floating point to fixed point, reducing bit-width and weight sharing. An example of a very aggressive strategy is reducing weight precision to ternary values restricted to $\{-1, 0, 1\}$ only, called Ternary Weight Network (TWN) [68]. The quantization is performed during training, using a straight-through estimator [63], where ternary weights are used during the forward and backward propagation but not during the parameters update. To make the network perform well, TWNs minimize the Euclidian distance between full precision weights and the ternary ones with the use of a non-negative layer- and channel-dependent scaling factor α .

4 Dataset

The CERN LHC experiments implement a real-time selection process, called trigger [72], to store a fraction of the events for further analysis. Jets are useful for many measurements and physics searches. A truly minimal approach to perform identification and tagging is with jet images. Generally, jets need a component of tracks as well to be properly reconstructed.

However, one could reconstruct the calorimeter part alone (known as CaloJet). The energy measurements of the emanating particles can be projected onto a cylindrical detector and represented as images by unfolding the inner surface of the calorimeter on a rectangle, and using the crystals as pixels, as in [73].

The detector effects and hadronization have an important effect on the jet substructure. In this work, we use an emulation of the Compact Muon Solenoid (CMS) apparatus as a reference. There are two calorimeters within the solenoid volume of the CMS detector. A lead tungstate crystal Electromagnetic Calorimeter (ECAL) is designed to stop particles whose main interaction is electromagnetic (photons, electrons). A brass and scintillator Hadronic Calorimeter (HCAL) is designed to stop hadrons. They give a measurement of the energy of particles (charged and neutrals). Each of them is composed of a barrel and two endcap sections. Forward calorimeters extend the pseudorapidity range (η) coverage provided by the barrel ($\eta \leq 1.4$) and endcap detectors ($1.4 < |\eta| \leq 3.0$). A more detailed description of the CMS detector, together with a definition of the coordinate system used and the relevant kinematic variables, can be found in [74].

This study aims at identifying different kinds of jets. To this purpose, we consider 13 TeV proton-proton collision events, in which RS gravitons decay to $b\bar{b}$, HH, WW, ZZ, or $t\bar{t}$ final states. Events are generated with Pythia [75] and the CMS detector effects are emulated using the Delphes [76] library. In addition to the hard collision, parasitic *pileup* collisions are also simulated, overlapping minimum bias events. The number of pileup collisions is sampled from a Poisson distribution. The calorimeter cells (towers) in the barrel region are arranged in a fixed discrete space with fine segmentation in η , ϕ , where ϕ is the translated azimuthal angle. The final image is formed by translating the calorimeter energy deposits into pixels, which results in a 340×360 pixel image. The intensity of each pixel is proportional to the sum of the energy of the corresponding cell. The previous studies on jet images implemented data pre-processing steps such as translation, rotation, re-pixelation, or inversion. However, in our study we only limit the input to barrel and endcap section, $\eta \in (-3, 3)$, and normalize pixel intensities to a fixed range $<0, 1>$, using maximum scaling. The ground truth labels for jets above threshold momentum (30 GeV/c for b and 200 GeV/c for the jets from boosted heavy particles) are obtained using a simple cone algorithm, i.e. associating together particles whose trajectories lie within a circle of radius $R = 0.4$ from the jet centre.

As a proof of concept, we investigate the tagging of the bottom (b) W boson (W), Higgs boson (H), or top quark (t) jet. An example input, energy deposits translated to two-dimensional images with two channels (corresponding to ECAL and HCAL) together with marked ground truth bounding boxes is shown in Figure 2.

5 Model, implementation and training procedure

The Jet-SSD architecture is shown in Figure 3. Several modifications are applied to the original architecture [49]. Due to target hardware constraints, all filters in convolution layers are of size 3×3 with no dilatation and all pooling layers have 2×2 filters. Each convolution block is followed by batch normalization [77, 78] and parametric rectified linear unit (PReLU) layers. To compress the model we use half of the channels of the VGG-16 in each layer. We also remove bias from all convolution layers. The extra layers proposed by the original paper do not contribute to accurate detection due to the size of jets and thus they are removed at the training. Retaining the deeper layers in the base network does not show improvements in the final detection results either, but they are critical during training due to additional signal during back-propagation. Hence, we only purge them at inference.

The Jet-SSD network is implemented on an NVidia Tesla GPU using PyTorch [79]. For training, we use stochastic gradient descent with an initial learning rate of 10^{-3} with momen-

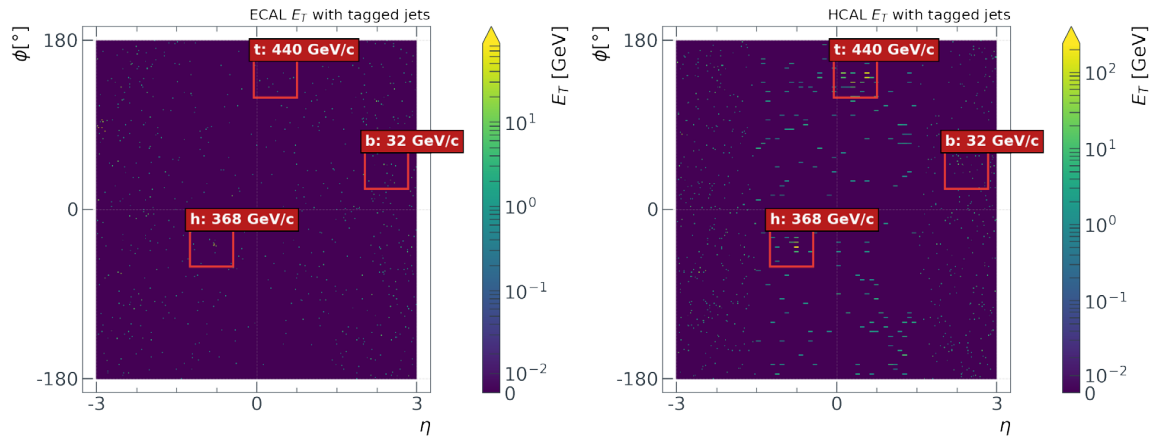


Figure 2. Energy deposits in CMS ECAL and HCAL translated to a two-dimensional image, an example input to the SSD network. The red bounding boxes correspond to ground truth with target label and momentum.

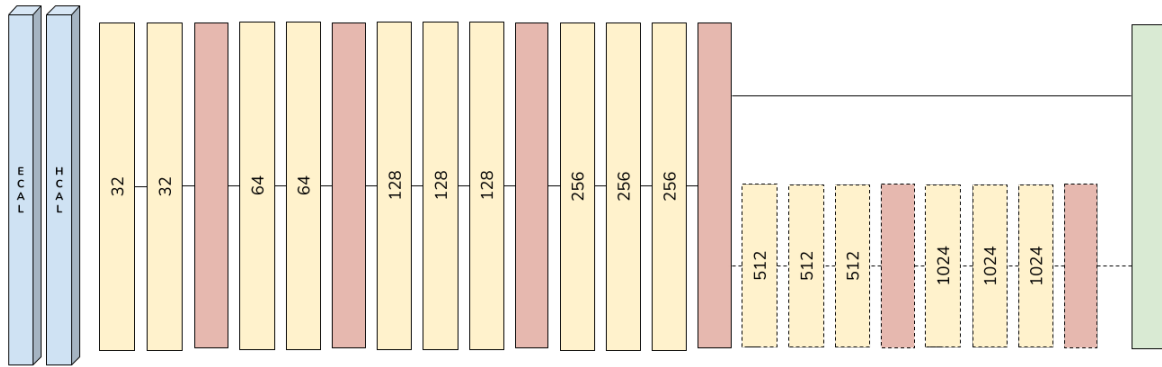


Figure 3. Jet-SSD architecture: input (in blue), convolution block, i.e. convolution layer followed by batch normalization and PReLU activation (in yellow), average pooling (in red) and output layer (in green). The numbers indicate the number of output channels in each block. The part of the network highlighted by dashed lines is used only during the training step.

tum set to 0.9 and weight regularization to 0.0005. We train the network for 100 epochs with a batch size of 25, decreasing the learning rate by a factor of 2 after 20, 30, 50, 60, 70, 80 and 90 epochs. We use 90k and 30k samples for training and validation, respectively. The training is performed in mixed-precision to speed up computation and distributed across 3 GPUs.

The full precision network (FPN) is trained from scratch using Xavier uniform initialization [80] (which helps with the sparsity of the input) as the pre-trained classification models on the real-world ImageNet [81] dataset have little relation to our calorimeter images. A common challenge when training models from scratch is the insufficient amount of training data which may lead to overfitting. However, it is not a problem in our case: the training dataset is large enough and, if overfitting occurred, we can go back and generate an even larger one. For TWN training we find out that pre-loading trained FPN weights greatly speeds up the process. And per-layer and per-channel scaling factor α improves the results.

The final detection layer returns a classification label (background, b, W/H or t jet) and three regression values. Two of them correspond to the centre of the jet, i.e. offset in η and ϕ plane from the anchor. The last one is jet mass regression which is an example of an auxiliary function that Jet-SSD can be tasked with.

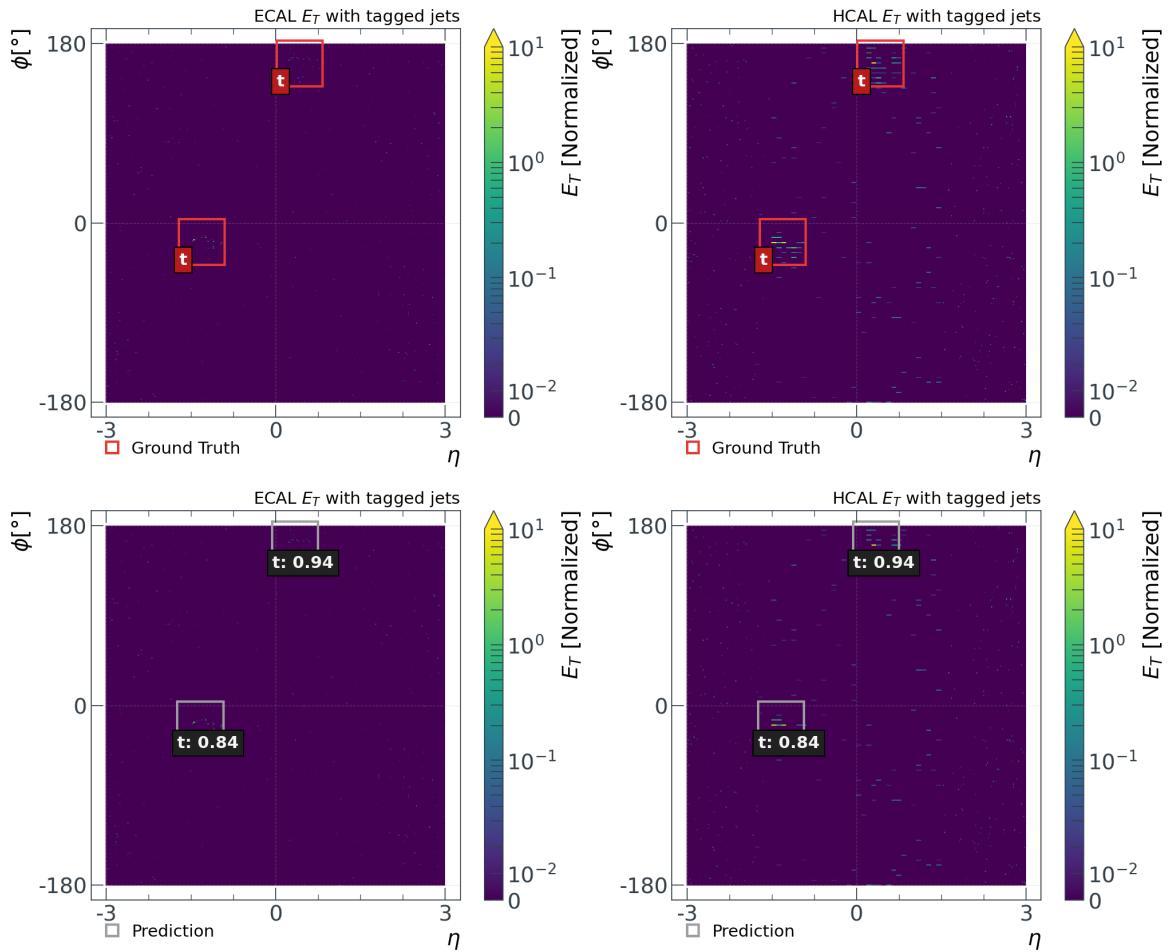


Figure 4. An example of the Jet-SSD at inference for one event with the input calorimeter image and highlighted true labels (top) and prediction bounding boxes (bottom). The fractional number next to the categorical label corresponds to the network confidence score.

6 Results

An example of the Jet-SSD in action is shown in Figure 4. Jet-SSD outputs predicted categorical label of the object, confidence and bounding boxes. In object detection true positive is defined as prediction with category equal to the ground truth label and Intersection over Union (IoU) above the predefined threshold, in our case 0.5. Successful prediction meets both criteria, otherwise, it is considered as a false negative.

To evaluate the model we use precision and recall (true positive rate), and average precision (AP) metric, which is computed for each category separately. Classification tasks usually report on the receiver operator characteristic (ROC) curve, which is a function of the false positive rate (fall-out or the background efficiency) as a function of the true positive rate (sensitivity or signal efficiency). In the case of object detection, the false positive rate is not very informative as there is a big imbalance between positive and negative class (there are no objects in most locations). Thus, the false positive rate is replaced by precision or positive predictive value (PPV). Intuitively, precision measures how accurate the predictions while recall measures the quality of the positive predictions. To draw a precision-recall (PR) curve, the predictions are first sorted in order of confidence followed by calculation of posi-

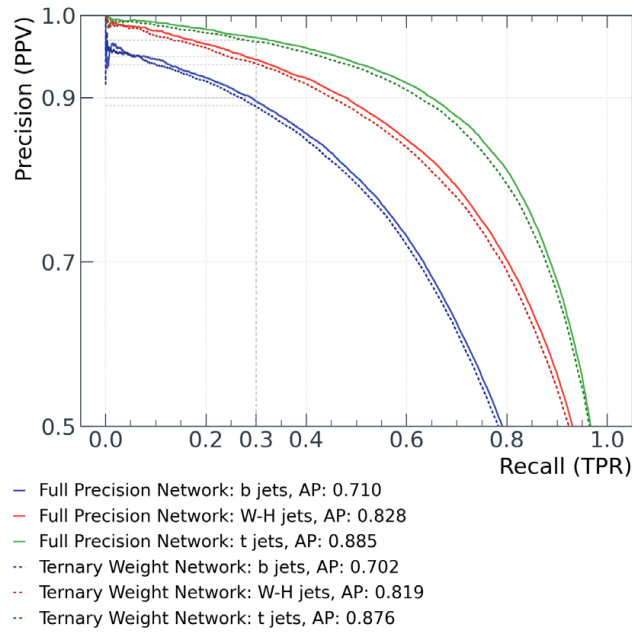


Figure 5. Jet-SSD PR curve and AP values for three target categories (b, W-H and t) and two weights precision (FPN and TWN).

tive predictive value and true positive rate for each confidence threshold. For the relationship between ROC and PR curve, see [82].

The PR curve of Jet-SSD, evaluated on a held-out test dataset consisting of 90k samples, is shown in Figure 5. The TWN results are closely matching the results of the FPN, which is reflected in an AP score. To calculate the value of AP, the maximum precision is calculated for the recall values that range from 0 to 1 with a step size of 0.1 and finally averaging over the results. From the PR curve, we can conclude that t jets are the easiest to identify while b jets detection is lacking. The result is not surprising for two reasons. Firstly, b jets have a lower momentum threshold, making the energy deposits more challenging to detect. Secondly, CNN based object detection is more challenging as the scale of the target object decreases; and b jets have a smaller radius than t, W and H jets. The latter issue can be further mitigated as small scale object detection is an active research field in machine learning (for example [83]).

Finally, we report the mean and median localization error in ϕ and η and the relative error in mass regression. These results are shown in Figure 6. The ϕ localization error is smaller than η due to input information loss. Remind that we limit input in η dimension. In the case when the jet centre is close to the edge, i.e. $|\eta| \approx 3.0$, part of the information is lost beyond image boundaries. Due to the cylindrical structure of the detector, this is not happening in ϕ dimension. Furthermore, we notice that the error does not decrease with p_T for η for which we don not find a reason. Finally, the mass regression relative error can be further decreased with re-balancing of the SSD training loss, i.e. increasing regression error contribution to back-propagation by introducing a new scaling hyper-parameter β : $loss = classification + localization + \beta \times auxiliary$, where $\beta > 1$.

7 Conclusions

In this paper, we introduced Jet-SSD, a deep learning network able to simultaneously localize, tag and estimate the mass of jets, a collimated spray of particles produced in high energy

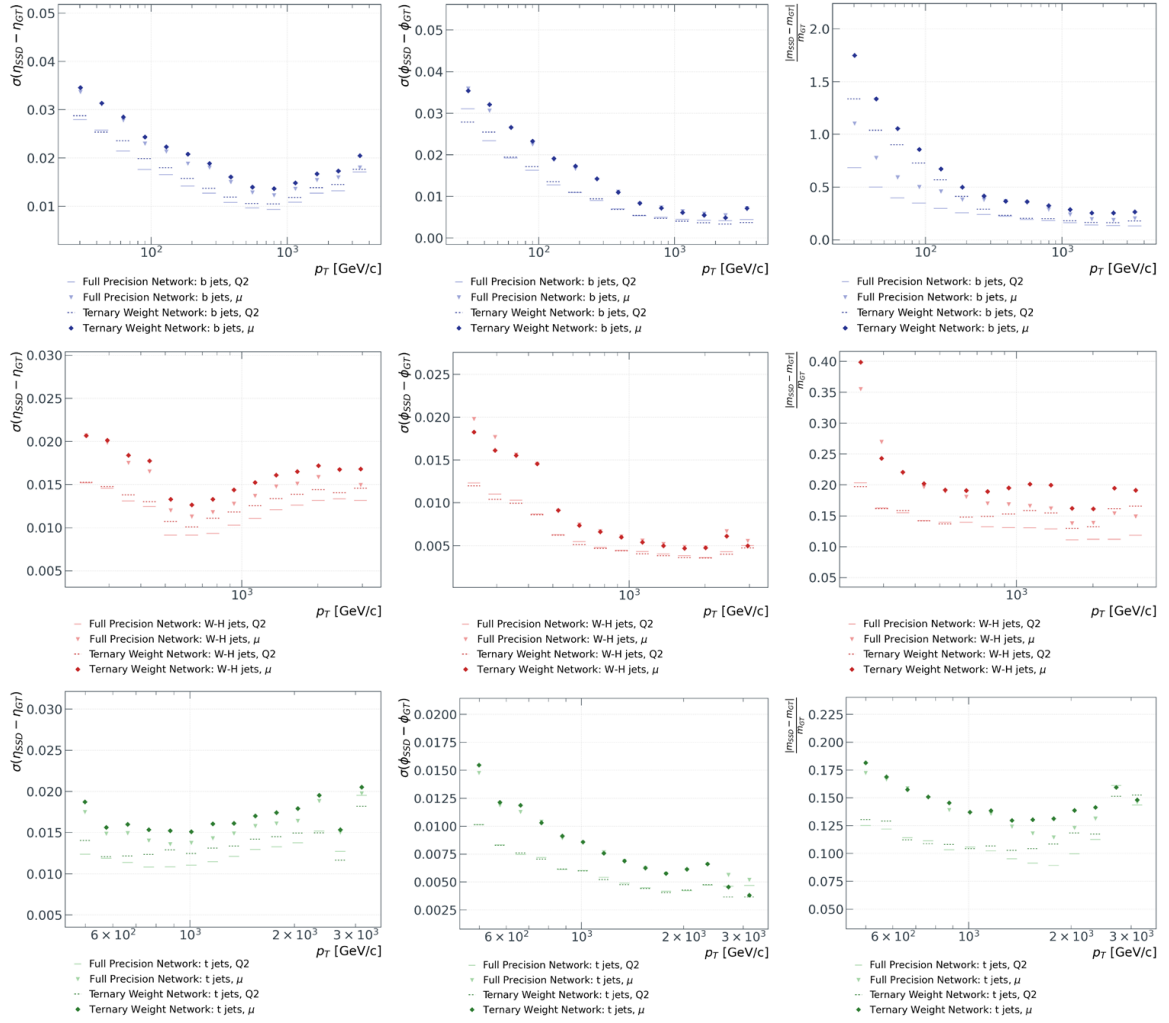


Figure 6. The mean (μ) and median (Q2) localization error in η (left column), ϕ (centre column) and relative error in mass regression between ground truth (GT) and Jet-SSD output (SSD) for FPN and TWN versions. The results are reported for each class independently: b jets (top row), W-H jets (centre row) and t jest (bottom row). All results are calculated as a function of p_T .

physics experiments. We showed that the compressed model via quantized weights to ternary values with layer- and channel-dependent scaling factor closely matches the performance of the full precision model. We seek to examine the performance of the network on dedicated hardware.

Acknowledgments

A. A. P., M. P., S. S. and V. L. are supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement n^o 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 thank Simons Foundation, Flatiron Institute and Ian Fisk for granting access to computing resources used for this project.

References

- [1] J.M. Butterworth, A.R. Davison, M. Rubin, G.P. Salam, *Physical review letters* **100**, 242001 (2008)
- [2] W. Skiba, D. Tucker-Smith, *Physical Review D* **75**, 115010 (2007)
- [3] V. Khachatryan, A.M. Sirunyan, A. Tumasyan, W. Adam, T. Bergauer, M. Dragicovic, J. Erö, C. Fabjan, M. Friedl, R. Fruehwirth et al., *Journal of High Energy Physics* **2014**, 173 (2014)
- [4] G. Aad, B. Abbott, J. Abdallah, R. Aben, M. Abolins, O. AbouZeid, H. Abramowicz, H. Abreu, R. Abreu, Y. Abulaiti et al., *Journal of High Energy Physics* **2015**, 1 (2015)
- [5] D. Adams, A. Arce, L. Asquith, M. Backovic, T. Barillari, P. Berta, D. Bertolini, A. Buckley, J. Butterworth, R.C. Toro et al., *The European Physical Journal C* **75**, 1 (2015)
- [6] A. Abdesselam, A. Belyaev, E.B. Kuutmann, U. Bitenc, G. Brooijmans, J. Butterworth, P.B. de Renstrom, D.B. Franzosi, R. Buckingham, B. Chapleau et al., *The European Physical Journal C* **71**, 1 (2011)
- [7] A. Altheimer, S. Arora, L. Asquith, G. Brooijmans, J. Butterworth, M. Campanelli, B. Chapleau, A. Cholakian, J. Chou, M. Dasgupta et al., *Journal of Physics G: Nuclear and Particle Physics* **39**, 063001 (2012)
- [8] A. Altheimer, A. Arce, L. Asquith, J.B. Mayes, E.B. Kuutmann, J. Berger, D. Bjergaard, L. Bryngemark, A. Buckley, J. Butterworth et al., *The European Physical Journal C* **74**, 1 (2014)
- [9] T. Plehn, M. Spannowsky, M. Takeuchi, D. Zerwas, *Journal of High Energy Physics* **2010**, 1 (2010)
- [10] A.J. Larkoski, S. Marzani, G. Soyez, J. Thaler, *Journal of High Energy Physics* **2014**, 146 (2014)
- [11] J. Thaler, K. Van Tilburg, *Journal of High Energy Physics* **2011**, 15 (2011)
- [12] A.J. Larkoski, G.P. Salam, J. Thaler, *Journal of High Energy Physics* **2013**, 108 (2013)
- [13] D. Krohn, J. Thaler, L.T. Wang, *Journal of High Energy Physics* **2010**, 84 (2010)
- [14] S.D. Ellis, C.K. Vermilion, J.R. Walsh, *Physical Review D* **81**, 094023 (2010)
- [15] M. Dasgupta, A. Fregoso, S. Marzani, G.P. Salam, *Journal of High Energy Physics* **2013**, 29 (2013)
- [16] M. Dasgupta, A. Fregoso, S. Marzani, A. Powling, *The European Physical Journal C* **73**, 1 (2013)
- [17] M. Dasgupta, A. Powling, A. Siodmok, *Journal of High Energy Physics* **2015**, 1 (2015)
- [18] J. Cogan, M. Kagan, E. Strauss, A. Schwartzman, *Journal of High Energy Physics* **2015**, 118 (2015)
- [19] J. Pearkes, W. Fedorko, A. Lister, C. Gay, arXiv preprint arXiv:1704.02124 (2017)
- [20] P. Baldi, K. Bauer, C. Eng, P. Sadowski, D. Whiteson, *Physical Review D* **93**, 094034 (2016)
- [21] S. Macaluso, D. Shih, *Journal of High Energy Physics* **2018**, 1 (2018)
- [22] L.G. Almeida, M. Backović, M. Cliche, S.J. Lee, M. Perelstein, *Journal of High Energy Physics* **2015**, 1 (2015)
- [23] L. de Oliveira, M. Kagan, L. Mackey, B. Nachman, A. Schwartzman, *Journal of High Energy Physics* **2016**, 1 (2016)
- [24] D. Guest, J. Collado, P. Baldi, S.C. Hsu, G. Urban, D. Whiteson, *Physical Review D* **94**, 112002 (2016)
- [25] J. Barnard, E.N. Dawe, M.J. Dolan, N. Rajcic, *Physical Review D* **95**, 014018 (2017)

- [26] A. Butter, G. Kasieczka, T. Plehn, M. Russell, *SciPost Phys* **5**, 028 (2018)
- [27] P.T. Komiske, E.M. Metodiev, M.D. Schwartz, *Journal of High Energy Physics* **2017**, 110 (2017)
- [28] J. Lin, M. Freytsis, I. Moutl, B. Nachman, *Journal of High Energy Physics* **2018**, 1 (2018)
- [29] G. Kasieczka, T. Plehn, A. Butter, K. Cranmer, D. Debnath, B.M. Dillon, M. Fairbairn, D.A. Faroughy, W. Fedorko, C. Gay et al., arXiv preprint arXiv:1902.09914 (2019)
- [30] G. Kasieczka, T. Plehn, M. Russell, T. Schell, *Journal of High Energy Physics* **2017**, 6 (2017)
- [31] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *Proceedings of the IEEE* **86**, 2278 (1998)
- [32] M. Cacciari, G.P. Salam, G. Soyez, *The European Physical Journal C* **72**, 1 (2012)
- [33] A.M. Sirunyan et al. (CMS), *Comput. Softw. Big Sci.* **4**, 10 (2020), 1912.06046
- [34] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, Y. LeCun, arXiv preprint arXiv:1312.6229 (2013)
- [35] K. Zhang, Z. Zhang, Z. Li, Y. Qiao, *IEEE Signal Processing Letters* **23**, 1499 (2016)
- [36] L. Zhang, L. Lin, X. Liang, K. He, *Is faster R-CNN doing well for pedestrian detection?*, in *European conference on computer vision* (Springer, 2016), pp. 443–457
- [37] Z. Zou, Z. Shi, Y. Guo, J. Ye, arXiv preprint arXiv:1905.05055 (2019)
- [38] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, M. Pietikäinen, *International journal of computer vision* **128**, 261 (2020)
- [39] J. Redmon, A. Farhadi, *YOLO9000: better, faster, stronger*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 7263–7271
- [40] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, *You only look once: Unified, real-time object detection*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 779–788
- [41] C.Y. Fu, W. Liu, A. Ranga, A. Tyagi, A.C. Berg, arXiv preprint arXiv:1701.06659 (2017)
- [42] X. Zhou, D. Wang, P. Krähenbühl, arXiv preprint arXiv:1904.07850 (2019)
- [43] T.Y. Lin, P. Goyal, R. Girshick, K. He, P. Dollár, *Focal loss for dense object detection*, in *Proceedings of the IEEE international conference on computer vision* (2017), pp. 2980–2988
- [44] R. Girshick, J. Donahue, T. Darrell, J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2014), pp. 580–587
- [45] S. Ren, K. He, R. Girshick, J. Sun, arXiv preprint arXiv:1506.01497 (2015)
- [46] R. Girshick, *Fast r-cnn*, in *Proceedings of the IEEE international conference on computer vision* (2015), pp. 1440–1448
- [47] J. Dai, Y. Li, K. He, J. Sun, arXiv preprint arXiv:1605.06409 (2016)
- [48] H. Xu, X. Lv, X. Wang, Z. Ren, N. Bodla, R. Chellappa, *Deep regionlets for object detection*, in *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 798–814
- [49] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.Y. Fu, A.C. Berg, *Ssd: Single shot multibox detector*, in *European conference on computer vision* (Springer, 2016), pp. 21–37
- [50] K. Simonyan, A. Zisserman, arXiv preprint arXiv:1409.1556 (2014)
- [51] Y. Cheng, D. Wang, P. Zhou, T. Zhang, arXiv preprint arXiv:1710.09282 (2017)

- [52] Y. LeCun, J.S. Denker, S.A. Solla, R.E. Howard, L.D. Jackel, *Optimal brain damage.*, in *NIPs* (Citeseer, 1989), Vol. 2, pp. 598–605
- [53] S. Han, H. Mao, W.J. Dally, arXiv preprint arXiv:1510.00149 (2015)
- [54] C. Louizos, M. Welling, D.P. Kingma, arXiv preprint arXiv:1712.01312 (2017)
- [55] A. Sironi, B. Tekin, R. Rigamonti, V. Lepetit, P. Fua, *IEEE transactions on pattern analysis and machine intelligence* **37**, 94 (2014)
- [56] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, R. Fergus, arXiv preprint arXiv:1404.0736 (2014)
- [57] M. Jaderberg, A. Vedaldi, A. Zisserman, arXiv preprint arXiv:1405.3866 (2014)
- [58] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, *Going deeper with convolutions*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9
- [59] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, arXiv preprint arXiv:1704.04861 (2017)
- [60] F.N. Iandola, S. Han, M.W. Moskewicz, K. Ashraf, W.J. Dally, K. Keutzer, arXiv preprint arXiv:1602.07360 (2016)
- [61] T. Cohen, M. Welling, *Group equivariant convolutional networks*, in *International conference on machine learning* (PMLR, 2016), pp. 2990–2999
- [62] C. Buciluă, R. Caruana, A. Niculescu-Mizil, *Model compression*, in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), pp. 535–541
- [63] M. Courbariaux, Y. Bengio, J.P. David, arXiv preprint arXiv:1511.00363 (2015)
- [64] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. Bengio, arXiv preprint arXiv:1602.02830 (2016)
- [65] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, Y. Zou, arXiv preprint arXiv:1606.06160 (2016)
- [66] M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi, *Xnor-net: Imagenet classification using binary convolutional neural networks*, in *European conference on computer vision* (Springer, 2016), pp. 525–542
- [67] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, *The Journal of Machine Learning Research* **18**, 6869 (2017)
- [68] F. Li, B. Zhang, B. Liu, arXiv preprint arXiv:1605.04711 (2016)
- [69] C. Zhu, S. Han, H. Mao, W.J. Dally, arXiv preprint arXiv:1612.01064 (2016)
- [70] E.H. Lee, D. Miyashita, E. Chai, B. Murmann, S.S. Wong, *Lognet: Energy-efficient neural networks using logarithmic computation*, in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (IEEE, 2017), pp. 5900–5904
- [71] Z. Cai, X. He, J. Sun, N. Vasconcelos, *Deep learning with low precision by half-wave gaussian quantization*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 5918–5926
- [72] CMS Collaboration, arXiv preprint arXiv:1609.02366 (2016)
- [73] W. Bhimji, S.A. Farrell, T. Kurth, M. Paganini, E. Racah et al., **1085**, 042034 (2018)
- [74] CMS Collaboration, *JInst* **3**, S08004 (2008)
- [75] T. Sjöstrand, S. Mrenna, P. Skands, *Computer Physics Communications* **178**, 852 (2008)
- [76] J. De Favereau, C. Delaere, P. Demin, A. Giammanco, V. Lemaitre, A. Mertens, M. Selvaggi, D. Collaboration et al., *Journal of High Energy Physics* **2014**, 57 (2014)
- [77] S. Ioffe, C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift* (2015), 1502.03167

- [78] E. Sari, M. Belbahri, V.P. Nia, *How does batch normalization help binary training?* (2020), 1909.09139
- [79] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., *PyTorch: An Imperative Style, High-Performance Deep Learning Library* (2019)
- [80] X. Glorot, Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in *Proceedings of the thirteenth international conference on artificial intelligence and statistics (JMLR Workshop and Conference Proceedings, 2010)*, pp. 249–256
- [81] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, Li Fei-Fei, *ImageNet: A large-scale hierarchical image database*, in *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), pp. 248–255
- [82] J. Davis, M. Goadrich, *The relationship between Precision-Recall and ROC curves*, in *Proceedings of the 23rd international conference on Machine learning* (2006), pp. 233–240
- [83] J. Rabbi, N. Ray, M. Schubert, S. Chowdhury, D. Chao, *Remote Sensing* **12**, 1432 (2020)

PAPER • OPEN ACCESS

Neural Network-Based Primary Vertex Reconstruction with FPGAs for the Upgrade of the CMS Level-1 Trigger System

To cite this article: C. Brown *et al* 2023 *J. Phys.: Conf. Ser.* **2438** 012106

View the [article online](#) for updates and enhancements.

You may also like

- [Level-1 pixel based tracking trigger algorithm for LHC upgrade](#)
C.-S. Moon and A. Savoy-Navarro
- [The scientific potential and technological challenges of the High-Luminosity Large Hadron Collider program](#)
Oliver Brüning, Heather Gray, Katja Klein et al.
- [Analysis and improvement of verifiable blind quantum computation](#)
Min Xiao, , Yannan Zhang et al.

Neural Network-Based Primary Vertex Reconstruction with FPGAs for the Upgrade of the CMS Level-1 Trigger System

C. Brown¹, A. Bundock², M. Komm³, V. Loncar³, M. Pierini³, B. Radburn-Smith¹, A. Shtipliyski¹, S. Summers³, J.-S. Dancu¹, and A. Tapper¹ for the CMS Collaboration

¹ Imperial College London (Blackett Laboratory, Prince Consort Rd, South Kensington, London, SW7 2BW, United Kingdom)

² University of Bristol (HH Wills Physics Laboratory, Tyndall Ave, Bristol BS8 1TL, United Kingdom)

³ CERN (CH-1211 Geneva 23, Switzerland)

E-mail: cebrown@cern.ch

Abstract.

The CMS experiment will be upgraded to maintain physics sensitivity and exploit the improved performance of the High Luminosity LHC. Part of this upgrade will see the first level (Level-1) trigger use charged particle tracks reconstructed within the full outer silicon tracker volume as an input for the first time and new algorithms are being designed to make use of these tracks. One such algorithm is primary vertex finding which is used to identify the hard scatter in an event and separate the primary interaction from additional simultaneous interactions. This work presents a novel approach to regress the primary vertex position and to reject tracks from additional soft interactions, which uses an end-to-end neural network. This neural network possesses simultaneous knowledge of all stages in the reconstruction chain, which allows for end-to-end optimisation. The improved performance of this network versus a baseline approach in the primary vertex regression and track-to-vertex classification is shown. A quantised and pruned version of the neural network is deployed on an FPGA to match the stringent timing and computing requirements of the Level-1 Trigger.

1. Introduction

The HL-LHC will produce up to 200 simultaneous proton-proton interactions per bunch crossing (pile-up) in the CMS detector. While most proton-proton interactions are inelastic, a hard scatter, which reveals the interactions CMS aims to probe, is far rarer making the identification of this primary interaction key for triggering. Due to the increased Pile-Up (PU), the CMS Level-1 (L1) Trigger is to be upgraded [1] and novel algorithms are being developed to maintain the physics sensitivity of the detector. Part of the L1 Trigger upgrade is to introduce track finding, which will use outer tracker modules [2] to reconstruct tracks with a transverse momentum (p_T) > 2 GeV. This information can be used to separate the Primary Vertex (PV) from PU. The main downstream user of the PV is Pile-Up Per Particle Identification (PUPPI) [3] which will perform calculations on the tracks associated to this vertex. This makes the PV regression and the association of tracks to this vertex important to utilise the physics performance of the PUPPI algorithm while reducing the impact of PU [1]. As with the current system, the L1 trigger will



be implemented on custom hardware running FPGAs with strict resource limitations. Thanks to larger front-end buffers, the latency will be increased to $12.5\ \mu\text{s}$ which when combined with more powerful FPGAs allows for more complex algorithms to be used.

While simple histogramming and cut-based methods can lead to effective vertexing strategies and are well within the latency budget they do not take into account all the information from the track finder and so are susceptible to non-genuine tracks, called fakes, and the differences in track parameter resolution in different regions of the detector [4]. Modern Deep Neural Networks (DNNs) are able to find optimal solutions from low-level information such as track features thus skipping the lengthy development processes of more traditional approaches. Tools such as hls4ml [5] and QKeras [6] allow these DNNs to be compressed to fit in FPGA hardware.

2. Baseline Approach

The PV is the location of the hard proton-proton scatter in an event. Offline, it is defined as the reconstructed vertex with the highest sum of track p_T^2 [7].

The baseline approach to vertex finding bins all tracks in z_0 weighted by their p_T in a 256-bin histogram spanning a z_0 range of -15 to 15 cm, as is shown in Fig. 1. Where z_0 is defined as the distance of a reconstructed track from the beamspot, along the beam line. A three-bin window is then passed across this histogram to find the three consecutive bins with the highest combined p_T . The centre of the middle of these three bins is returned as the PV. While this method is fast (a latency of 30 clock cycles at 360 MHz) and has low resource usage, it has some key issues. The first is the lack of correction for the degradation in z_0 resolution for high η tracks, which leads to a worsening of the resolution of the PV. Secondly, it does not account for high p_T fakes which, when associated with clusters of PU tracks, can appear as high p_T vertices.

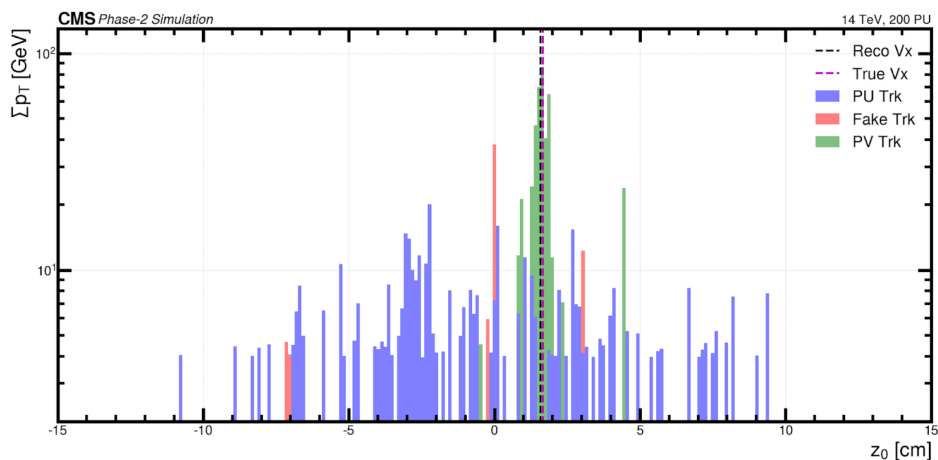


Figure 1. A single simulated event of a hadronic $t\bar{t}$ decay with PU of 200 showing all tracks histogrammed in z_0 weighted by p_T and coloured by their track type. Also shown is the reconstructed PV using the baseline approach and true generator PV.

The baseline approach to track-to-vertex association uses an η -dependent window in z_0 around the PV. This is reasonably effective with a true positive rate (correctly assigning PV tracks to the PV) of 91% and a false positive rate (assigning either a PU or fake track to the vertex) of 10%. Again, this method is fast and simple but fails to take into account more complex track features, such as the quality of the track fit, and is therefore heavily dependent on the resolution of the tracks it is provided.

3. End-to-End Neural Network Approach

The end-to-end Neural Network (NN) approach uses the same concepts as the baseline approach and expands on them with interconnected neural networks that are trainable end-to-end as shown in Fig. 2. Instead of weighting by p_T in a 256 bin histogram, a three layer DNN is used to learn an ideal weight function per track from input track features. The features used are the track p_T , η and the output of a BDT trained to distinguish non-genuine and real tracks [8] (labelled as MVA in eq. 1). These learned track weights are then used in combination with the track's z_0 to fill a histogram. This histogram is used as the input to a 1D convolution of kernel size three, depth one and stride one. The convolved histogram is passed through an ArgMax to obtain the bin position with the largest entry, as in the baseline approach.

Instead of a cut-based approach to track-to-vertex association a three layer DNN is used, which uses the same input track features as the weight network and additionally the distance from the PV to the track in z_0 . Using a SoftMax final output activation, a likelihood that a track belongs to the PV is returned.

3.1. Back-Propagation

The end-to-end network is trained in one cycle with a two part loss function. The first is a Huber loss [9] for the event level regression of the PV versus the true generator-level vertex. The second is a binary cross entropy loss that is used at the track level comparing the output track-to-vertex association probability to the simulation truth track label. These two losses are equally weighted.

Part of the end-to-end network is a histogram that is filled with a learnt track weight, which is convolved and the peak found. This contains two custom operations where the differential of the loss function with respect to the network weights are needed. The first is the histogram where each bin h_i has the input of the learnt weight w and the track's z_0 . The bins are filled as:

$$h_i = \sum_{j=0}^{tracks} \delta(j \in \text{bin } i) w(p_{T,j}, \eta_j, MVA_j) \quad (1)$$

resulting in the following gradients

$$\frac{\partial h_i}{\partial z_0} = 0 \quad \text{and} \quad \frac{\partial h_i}{\partial w} = \sum_{j=0}^{tracks} \delta(j \in \text{bin } i) \quad (2)$$

which are implemented as custom TensorFlow [10] operations.

The second part of the PV regression is the peak finding of a convolved histogram. This in a forward pass is simply an ArgMax operation that finds the index of the highest member of a 256 element vector. However, in order to back-propagate the regression loss function the differential of this with respect to its inputs is needed which, for a standard ArgMax, is undefined. Instead, a soft ArgMax is used which combines a SoftMax, a linear layer, and a final sum to find the ArgMax of the input vector. The soft ArgMax of a vector x with N total elements is defined as:

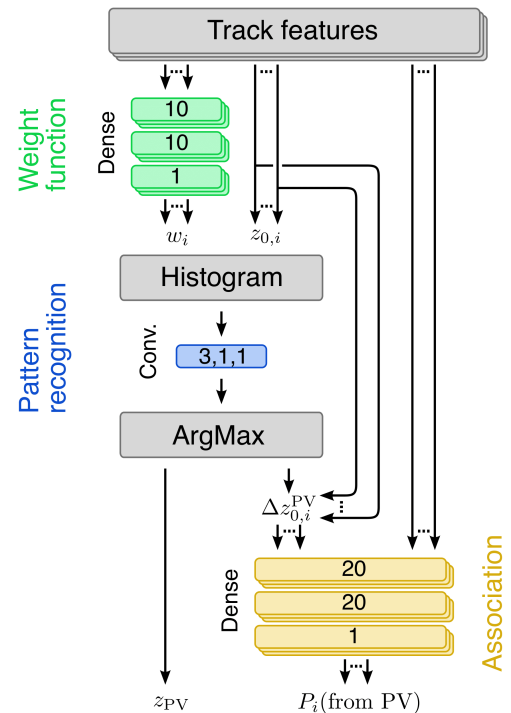


Figure 2. End-to-end network architecture showing the three distinct networks in colour as well as the position of the histogram layer and ArgMax.

$$\sum_{i=0}^N i \frac{e^{x_i/T}}{\sum_{j=0}^N e^{x_j/T}} \quad (3)$$

where T is a tuned hyperparameter of the network which allows this layer to return an approximate one-hot encoding.

4. Performance

The end-to-end approach outperforms the baseline approach in several key metrics. The first is the PV regression. Figure 3 shows the NN approaches in red and blue outperform the baseline in black especially in the tails of the residual where the improved filtering of fake tracks has reduced the number of high p_T clusters appearing to be the PV. Secondly, the NN outperforms the baseline approach in assigning tracks to this PV. The receiver operating characteristic (ROC) curve in Fig. 4 demonstrates that for a fixed false positive rate of 10 % the NN approach has a true positive rate of 96 % versus the baseline rate of 91 %.

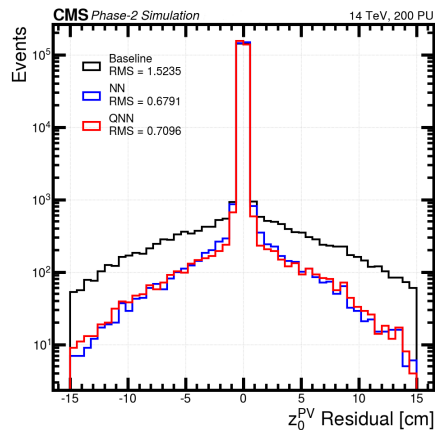


Figure 3. True PV - Reconstructed PV for the Baseline and NN approaches. NN refers to the floating point approach while QNN is the quantised approach described in Section 5.

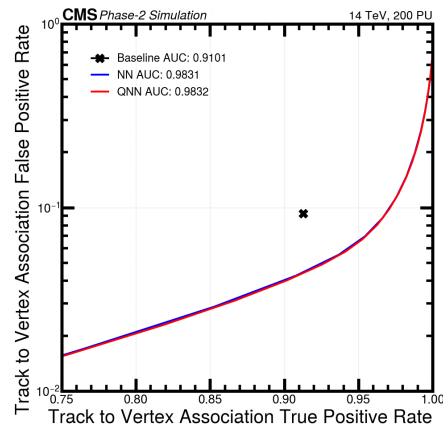


Figure 4. Receiver Operating Characteristic (ROC) curve for the Baseline and NN approaches to track to vertex association. Shown as true positive rate versus false positive rate.

5. Firmware Implementation

The hls4ml [5] package is used to realise this network in FPGA firmware. As the network has custom histogram layers, these were not converted with hls4ml but instead existing VHDL firmware from the baseline approach was reused. This means the network is split into three parts when it is converted: a weight network that takes input tracks and outputs a learnt weight; a pattern network that convolves the histogram created from the tracks; and an association network that outputs a probability the track is from the vertex, these separate networks are highlighted in Fig. 2. As the latency budget is small, parts of the network will be implemented multiple times to exploit the parallelism of the L1 architecture, notably the weight and association networks that work on a track-by-track level and so will be replicated 18 times. The replication of elements of the network means the size in FPGA resources of the partial networks is critical for their use in firmware. A variety of tools were used to reduce the size of the network. Firstly, regularization introduced a loss function that penalizes the

absolute value of the weights [11]. Secondly, pruning iteratively removed weights close to zero to remove unnecessary weights, keeping the overall network size small [12]. Finally, quantization aware training using the QKeras [6] package uses fixed point numbers for network parameters with restricted bitwidths, which, when passed to hls4ml, reduced the required resources for the network.

The final resource usages for a Xilinx UltraScale+ VU9P with a clock frequency of 360 MHz are shown in Table 1. Both an unquantised and quantised version of each part of the network are shown, demonstrating the effectiveness of quantised aware training and pruning of the networks, especially in reducing the Digital Signal Processor (DSP) usage which is the limiting factor in these FPGAs. Also shown in Figs. 3 and 4 is the performance of the full quantised network in red, demonstrating no loss in performance when moving from an unquantised to quantised network.

Table 1. Resource usage and latencies of a Xilinx VU9P running at 360 MHz for the floating point Neural Network (NN) and the quantised and pruned version (Q) with their expected number of replications. Also included is the baseline approach, the NN approaches are additional to these resources and latency as they use existing parts of the baseline firmware. These resource usages are estimates from a Vivado synthesis of the networks and the latencies from a C-Simulation.

Network	Latency (ns)	Initiation Interval (ns)	LUTs %	DSPs %	BRAMs %	FFs %
NN (Q) Weights	22 (14)	2.7 (2.7)	2.52 (0.90)	19.98 (0.00)	0.00 (0.00)	0.72 (0.36)
NN (Q) Pattern	58 (42)	51 (35)	4.27 (4.43)	3.74 (0.00)	5.28 (5.28)	3.22 (3.15)
NN (Q) Assoc.	30 (25)	2.7 (2.7)	0.54 (7.92)	107.64 (0.54)	0.00 (0.00)	2.70 (2.34)
Baseline	44	2.7	2.40	0.00	1.90	1.40

6. Conclusion

The HL-LHC will see up to 200 PU conditions for the LHC experiments. To maintain the physics performance of the detector and exploit the high integrated luminosity, the CMS experiment is being upgraded. Upgrades to the L1 Trigger system will see charged particle tracks within the full outer silicon tracker volume used for track matching and global event variables such as the primary vertex, which is necessary to separate the hard interaction from pile-up. This work introduces a novel approach to PV finding and association of tracks to the PV using an end-to-end neural network that learns both the PV position and the likelihood of a track originating from this PV. The network uses a custom histogram layer and soft ArgMax to ensure that the loss functions can be back-propagated and is shown to outperform the baseline approach in key metrics. Finally, the implementation of this network in an FPGA is discussed and the effective use of QKeras and pruning to reduce the overall resource usage is demonstrated.

References

- [1] CMS Collaboration, “The Phase-2 Upgrade of the CMS Level-1 Trigger Technical Design Report,” Tech. Rep. CERN-LHCC-2020-009. CMS-TDR-019-002, CERN, Geneva, 2020. <https://cds.cern.ch/record/2272264>.
- [2] G. Hall, M. Raymond, and A. Rose, “2-d PT module concept for the SLHC CMS tracker,” *Journal of Instrumentation*, vol. 5, no. 07, p. C07012, 2010.
- [3] D. Bertolini, P. Harris, M. Low, and N. Tran, “Pileup Per Particle Identification,” *JHEP*, vol. 10, p. 059, 2014.
- [4] S. Mersi, “Phase-2 Upgrade of the CMS Tracker,” *Nuclear and Particle Physics Proceedings*, vol. 273-275, pp. 1034 – 1041, 2016. 37th Int. Conf. on High Energy Physics (ICHEP).

- [5] J. Duarte *et al.*, “Fast inference of deep neural networks in FPGAs for particle physics,” , *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.
- [6] C. N. Coelho *et al.*, “Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors,” *Nature Machine Intelligence*, vol. 8, no. 3, pp. 675–686, 2021.
- [7] CMS Collaboration, “Description and performance of track and primary-vertex reconstruction with the CMS tracker,” *Journal of Instrumentation*, vol. 9, no. 10, p. P10009, 2014.
- [8] S. Summers, “Application of FPGAs to Triggering in High Energy Physics.” Ph.D. thesis Imperial College, London, 2018.
- [9] P. J. Huber, “Robust Estimation of a Location Parameter,” *The Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73 – 101, 1964.
- [10] M. Abadi *et al.*, “TensorFlow,” 2015. Software available from tensorflow.org.
- [11] R. Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.
- [12] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning both weights and connections for efficient neural networks,” in *Proc. of the 28th Int. Conf. on Neural Information Processing Systems - Vol. 1*, NIPS’15, (Cambridge, MA, USA), p. 1135–1143, MIT Press, 2015.



Contribution ID: 460

Type: Oral

Generative Adversarial Networks for fast simulation: generalisation and distributed training in HPC

Thursday, 14 March 2019 18:20 (20 minutes)

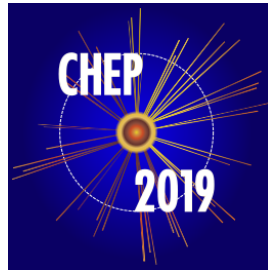
Deep Learning techniques have are being studied for different applications by the HEP community: in this talk, we discuss the case of detector simulation. The need for simulated events, expected in the future for LHC experiments and their High Luminosity upgrades, is increasing dramatically and requires new fast simulation solutions. We will describe an R&D activity within CERN openlab, aimed at providing a configurable tool capable of training a neural network to reproduce the detector response and replace standard Monte Carlo simulation. This represents a generic approach in the sense that such a network could be designed and trained to simulate any kind of detector in just a small fraction of time. We will present the first application of three-dimensional convolutional Generative Adversarial Networks to the simulation of high granularity electromagnetic calorimeters. We will describe detailed validation studies comparing our results to Geant4 Monte Carlo simulation, showing, in particular, the very good agreement we obtain for high level physics quantities (such as energy shower shapes) and detailed calorimeter response (single cell response). Finally we will show how this tool can easily be generalized to describe a larger class of calorimeters, opening the way to a generic machine learning based fast simulation approach. To achieve generalization we will leverage advanced optimization algorithms (using Bayesian and/or Genetic approach) and apply state of the art data parallel strategies to distribute the training process across multiple nodes in HPC and Cloud environment. Performance of the parallelization of GAN training on HPC clusters will also be discussed in details.

Primary authors: VALLECORSA, Sofia (CERN); VLIMANT, Jean-Roch (California Institute of Technology (US)); LONCAR, Vladimir (University of Belgrade (RS)); NGUYEN, Thong (California Institute of Technology (US)); KHATTAK, Gulrukh (CERN); PIERINI, Maurizio (CERN); CARMINATI, Federico (CERN); PANTALEO, Felice (CERN)

Presenter: VALLECORSA, Sofia (CERN)

Session Classification: Track 1: Computing Technology for Physics Research

Track Classification: Track 1: Computing Technology for Physics Research



Contribution ID: 176

Type: Oral

hls4ml: deploying deep learning on FPGAs for L1 trigger and Data Acquisition

Thursday, 7 November 2019 12:15 (15 minutes)

Machine learning is becoming ubiquitous across HEP. There is great potential to improve trigger and DAQ performance with it. However, the exploration of such techniques within the field in low latency/power FPGAs has just begun. We present hls4ml, a user-friendly software, based on High-Level Synthesis (HLS), designed to deploy network architectures on FPGAs. As a case study, we use hls4ml for boosted-jet tagging with deep networks at the LHC. We map out resource usage and latency versus network architectures, to identify the typical problem complexity that hls4ml could deal with. We discuss current applications in HEP experiments and future applications. We also report on recent progress in the past year on newer neural network architectures and networks with orders of magnitude more parameters.

Consider for promotion

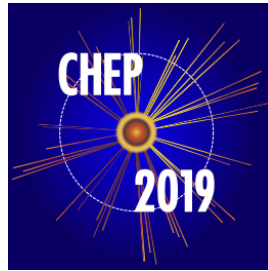
Yes

Primary authors: TRAN, Nhan Viet (Fermi National Accelerator Lab. (US)); LONCAR, Vladimir (University of Belgrade (RS))

Presenter: LONCAR, Vladimir (University of Belgrade (RS))

Session Classification: Track 1 –Online and Real-time Computing

Track Classification: Track 1 –Online and Real-time Computing



Contribution ID: 35

Type: **Oral**

MPI-based tools for large-scale training and optimization at HPC sites

Tuesday, 5 November 2019 11:00 (15 minutes)

MPI-learn and MPI-opt are libraries to perform large-scale training and hyper-parameter optimization for deep neural networks. The two libraries, based on Message Passing Interface, allows to perform these tasks on GPU clusters, through different kinds of parallelism. The main characteristic of these libraries is their flexibility: the user has complete freedom in building her own model, thanks to the multi-backend support. In addition, the library supports several cluster architectures, allowing a deployment on multiple platforms. This generality can make this the basis for a train & optimise service for the HEP community. We present scalability results obtained from two typical HEP use-case: jet identification from raw data and shower generation from a GAN model. Results on GPU clusters were obtained at the ORNL TITAN supercomputer and other HPC facilities, as well as exploiting commercial cloud resources and OpenStack. A comprehensive comparisons of scalability performance across platforms will be presented, together with a detailed description of the libraries and their functionalities.

Consider for promotion

Yes

Primary authors: LONCAR, Vladimir (University of Belgrade (RS)); VLIMANT, Jean-Roch (California Institute of Technology (US)); Dr VALLECORSIA, Sofia (CERN); KHATTAK, Gul Rukh (University of Peshawar (PK)); PIERINI, Maurizio (CERN); NGUYEN, Thong (California Institute of Technology (US)); CARMINATI, Federico (CERN)

Presenter: LONCAR, Vladimir (University of Belgrade (RS))

Session Classification: Track 9 –Exascale Science

Track Classification: Track 9 –Exascale Science

Fast Inference of Deep Neural Networks for Real-time Particle Physics Applications

Authors: [Javier Duarte](#), [Song Han](#), [Philip Harris](#), [Sergo Jindariani](#), [Edward Kreinar](#), [Benjamin Kreis](#), [Vladimir Loncar](#), [Jennifer Ngadiuba](#), [Maurizio Pierini](#), [Dylan Rankin](#), [Ryan Rivera](#), [Sioni Summers](#), [Nhan Tran](#), [Zhenbin Wu](#) [Authors Info & Claims](#)

FPGA '19: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays • February 2019 • Pages 305
• <https://doi.org/10.1145/3289602.3293986>

Published: 20 February 2019 [Publication History](#) [Check for updates](#)

3 0



ABSTRACT



Machine learning methods are ubiquitous and have proven to be very powerful in LHC physics, and particle physics as a whole. However, exploration of such techniques in low-latency, low-power FPGA (Field Programmable Gate Array) hardware has only just begun. FPGA-based trigger and data acquisition systems have extremely low, sub-microsecond latency requirements that are unique to particle physics. We present a case study for neural network inference in FPGAs focusing on a classifier for jet substructure which would enable many new physics measurements. While we focus on a specific example, the lessons are far-reaching. A compiler package is developed based on High-Level Synthesis (HLS) called HLS4ML to build machine learning models in FPGAs. The use of HLS increases accessibility across a broad user community and allows for a drastic decrease in firmware development time. We map out FPGA resource usage and latency versus neural network hyperparameters to allow for directed resource tuning in the low latency environment and assess the impact on our benchmark Physics performance scenario. For our example jet substructure model, we fit well within the available resources of modern FPGAs with latency on the scale of 100~ns.

Cited By

[View all](#) [↗](#)

Rothmann M and Pormann M. (2023). STANN – Synthesis Templates for Artificial Neural Network Inference and Training. *Advances in Computational Intelligence*. 10.1007/978-3-031-43085-5_31. (394-405).

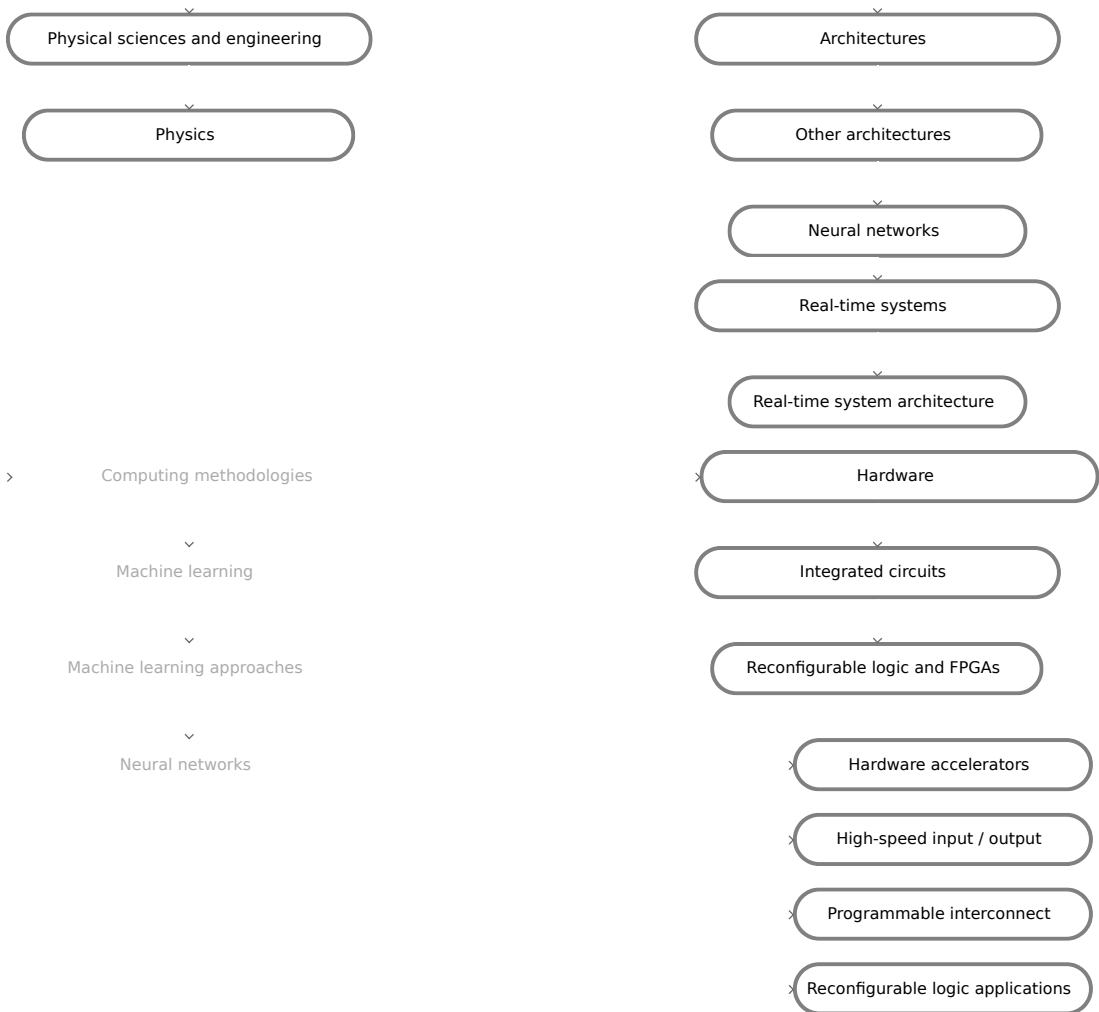
https://link.springer.com/10.1007/978-3-031-43085-5_31

Tourad E and Eleuldj M. (2022). Generic Automated Implementation of Deep Neural Networks on Field Programmable Gate Arrays. *Innovations in Smart Cities Applications Volume 5*. 10.1007/978-3-030-94191-8_80. (989-1000).

https://link.springer.com/10.1007/978-3-030-94191-8_80

Nechi A, Groth L, Mulhem S, Merchant F, Buchty R and Berekovic M. (2023). FPGA-based Deep Learning Inference Accelerators: Where Are We Standing?. *ACM Transactions on Reconfigurable Technology and Systems*. 10.1145/3613963.

<https://dl.acm.org/doi/10.1145/3613963>



Recommendations

Behavioural synthesis of SGD using the CCC framework: a simple XOR-solving MLP

Abstract

Behavioural synthesis enables the automation of the design process by generating task-specific hardware configured for either FPGA and SoC platforms or custom silicon...

[Read More](#)

HLS-Based Acceleration Framework for Deep Convolutional Neural Networks

Applied Reconfigurable Computing. Architectures, Tools, and Applications

Abstract

Deep Neural Networks (DNNs) have been successfully applied in many fields. Considering performance, flexibility, and energy efficiency, Field Programmable Gate Array (FPG...

[Read More](#)

A Runtime Programmable Accelerator for Convolutional and Multilayer Perceptron Neural Networks on FPGA

Applied Reconfigurable Computing. Architectures, Tools, and Applications

Abstract

Deep neural networks (DNNs) are prevalent for many applications related to classification, prediction and regression. To perform different applications with better...

[Read More](#)

Comments



0 Comments

Share

Best Newest Oldest

Nothing in this discussion yet.

[Privacy](#)

[Do Not Sell My Data](#)

[View Table Of Contents](#)

Categories

- [Journals](#)
- [Magazines](#)
- [Books](#)
- [Proceedings](#)
- [SIGs](#)
- [Conferences](#)
- [Collections](#)
- [People](#)







Join

- [Join ACM](#)
- [Join SIGs](#)
- [Subscribe to Publications](#)
- [Institutions and Libraries](#)

About

- [About ACM Digital Library](#)
- [ACM Digital Library Board](#)
- [Subscription Information](#)
- [Author Guidelines](#)
- [Using ACM Digital Library](#)
- [All Holdings within the ACM Digital Library](#)
- [ACM Computing Classification System](#)
- [Digital Library Accessibility](#)

Connect

-  [Contact](#)
-  [Facebook](#)
-  [Twitter](#)
-  [Linkedin](#)
-  [Feedback](#)
-  [Bug Report](#)

The ACM Digital Library is published by the Association for Computing Machinery. Copyright © 2023 ACM, Inc.

[Terms of Usage](#) | [Privacy Policy](#) | [Code of Ethics](#)



Large and compressed Convolutional Neural Networks on FPGAs with hls4ml

Monday, 30 November 2020 14:58 (6 minutes)

We present ultra low-latency Deep Neural Networks with large convolutional layers on FPGAs using the hls4ml library. Taking benchmark models trained on public datasets, we discuss various options to reduce the model size and, consequently, the FPGA resource consumption: pruning, quantization to fixed precision, and extreme quantization down to binary or ternary precision. We demonstrate how inference latencies of $O(10)$ micro seconds can be obtained while high accuracy is maintained

Primary author: LONCAR, Vladimir (CERN)

Presenter: LONCAR, Vladimir (CERN)

A OneAPI backend of hls4ml to speed up Neural Network inference on CPUs

Monday, 30 November 2020 15:30 (6 minutes)

A recent effort to explore a neural network inference in FPGAs using High-Level Synthesis language (HLS), focusing on low-latency applications in triggering subsystems of the LHC, resulted in a framework called hls4ml. Deep Learning model converted to HLS using the hls4ml framework can be executed on CPUs, but have subpar performance. We present an extension of hls4ml using the new Intel oneAPI toolkit that converts deep learning models into high-performance Data Parallel C++ optimized for Intel x86 CPUs. We show that inference time on Intel CPUs is improved hundreds of times over previous HLS-based implementation, and several times over unmodified Keras/TensorFlow.

Primary author: LONCAR, Vladimir (CERN)

Presenter: LONCAR, Vladimir (CERN)

AIgean: An Open Framework for Machine Learning on Heterogeneous Clusters

Naif Tarafdar^{*}, Giuseppe Di Guglielmo[†], Philip C Harris[‡], Jeffrey D Krupa[‡],
Vladimir Loncar[§], Dylan S Rankin[‡], Nhan Tran[¶], Zhenbin Wu^{||}, Qianfeng Shen^{*}, Paul Chow^{*}

^{*} University of Toronto

[†] Columbia University

[‡] Massachusetts Institute of Technology

[§] CERN

[¶] Fermilab

^{||} University of Illinois

Email: {naif.tarafdar, qianfeng.shen}@mail.utoronto.ca, giuseppe.diguglielmo@columbia.edu,
{pcharris, jkrupa, drankin}@mit.edu

{vladimir.loncar, zhenbin.wu}@cern.ch, ntran@fnal.gov, pc@eecg.toronto.edu

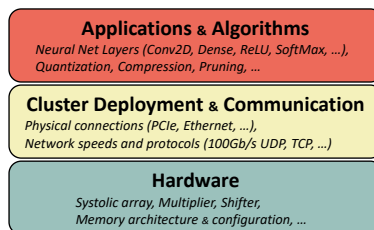


Fig. 1. Abstraction Stack for common Machine Learning Frameworks.

Machine learning (ML) in the past decade has been one of the most popular topics of research within the computing community. Interest within the computing field ranges across all levels of the computation stack. We show this stack in Figure 1. This work introduces an open framework, called AIgean, to build and deploy machine learning (ML) algorithms on a heterogeneous cluster of devices (CPUs and FPGAs). Users can flexibly modify any layer of the machine learning stack in Figure 1 to suit their need. This allows both machine learning domain experts to focus on higher algorithmic layers, and distributed systems experts to create the communication layers below.

We leverage two open-source projects: Galapagos [3], for multi-FPGA deployment and hls4ml [1], for generating machine learning kernels synthesizable using Vivado HLS. We use particle detection in the physics domain to provide the first driving applications that help us to characterize the framework. To use AIgean, the user provides a machine learning algorithm and the resources of their cluster. Then AIgean converts the algorithm into appropriate IP cores and provides the off-chip communication between devices. HLS4ml was adapted to provide streaming interfaces. This fits the Galapagos model and works well with single inference. We designed a bridge from hls4ml to Galapagos to convert fixed-point streams into Galapagos streams that can be received from any compute

kernel within our cluster.

We demonstrate the effectiveness of AIgean with two use cases: a small network running on a single network-connected FPGA and an autoencoder running on three FPGAs, and compare to SDAccel [4]. Our small neural-network single-FPGA implementation can implement a single inference in 0.08 ms as opposed to 2.9 ms in SDAccel, highlighting the efficacy of a network-connected accelerator for a single inference case. Our 3-FPGA autoencoder implementation performs a batch-size of 2400 inferences in 0.08 ms as opposed to 0.26 ms on a single FPGA in SDAccel, showing the need for multi-FPGA fabrics as it allows users to target large implementations of their machine learning circuitry, these implementations can perform better than smaller implementations. Multi-FPGA fabrics also make it possible to implement large networks such as ResNet-50, which is work in progress. Preliminary results before any optimizations have been applied shows that we can achieve a throughput of 200 images/s using 5 FPGAs, which can be compared to Brainwave, which has a throughput of 559 images/s [2]. We expect our results to improve significantly once we apply optimizations.

REFERENCES

- [1] J. Duarte, P. Harris, S. Hauck, B. Holzman, S.-C. Hsu, S. Jindariani, S. Kha, B. Krei, B. Le, M. Liu *et al.*, "FPGA-accelerated machine learning inference as a service for particle physics computing," *arXiv preprint arXiv:1904.08986*, 2019.
- [2] J. Fowers *et al.*, "A Configurable Cloud-scale DNN Processor for Real-time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 1–14. [Online]. Available: <https://doi.org/10.1109/ISCA.2018.00012>
- [3] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow, "Galapagos: A full stack approach to FPGA integration in the cloud," *IEEE Micro*, vol. 38, no. 6, pp. 18–24, 2018.
- [4] Xilinx Inc., "SDAccel Development Environment," <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>, 2020.

Contribution ID: 617 Contribution code: **contribution ID 617**Type: **Oral**

Looking for new physics in the LHC hardware trigger with Deep Autoencoders

Monday, 29 November 2021 17:20 (20 minutes)

We show how to adapt and deploy anomaly detection algorithms based on deep autoencoders, for the unsupervised detection of new physics signatures in the extremely challenging environment of a real-time event selection system at the Large Hadron Collider (LHC). We demonstrate that new physics signatures can be enhanced by three orders of magnitude, while staying within the strict latency and resource constraints of a typical LHC event filtering system. This would allow for collecting datasets potentially enriched with high-purity contributions from new physics processes. Through per-layer, highly parallel implementations of network layers, support for autoencoder-specific losses on FPGAs and latent space based inference, we demonstrate that anomaly detection can be performed in as little as 80 ns using less than 3% of the logic resources in the Xilinx Virtex VU9P FPGA. Opening the way to real-life applications of this idea during the next data-taking campaign of the LHC.

Significance

This talk covers the material in the paper <https://arxiv.org/abs/2108.03986>, where we explore for the first time in the LHC hardware trigger the potential of unsupervised and semi-supervised techniques for detecting new physics signatures, most notably anomaly detection algorithms enhanced with deep learning. Using such algorithms, one can learn how to identify out-of-distribution events directly from the LHC data. One can then select the most anomalous events, which are the most likely to contain new physics signatures, into a special data stream of anomalous events. The complications come from the fact that such algorithms have to fit into the strict restrictions of Level-1 trigger, in particular, it should have latency of $O(1) \mu\text{s}$, take up a small fraction of the FPGA resources and have an initial interval smaller than LHC bunch-crossing (i.e. 25 ns).

We have successfully designed such an algorithm and tested its performance on several potential new physics scenarios with the ability to enhance such signals by three orders of magnitude, while keeping the algorithm's performance as fast as 80 ns, with an initial interval of just 5 ns using less than 3% of FPGA resources.

References

Speaker time zone

Compatible with Europe

Primary authors: POL, Adrian Alan (CERN); PULJAK, Ema (Rudjer Boskovic Institute (HR)); GOVORKOVA, Katya (CERN); DUARTE, Javier Mauricio (Univ. of California San Diego (US)); NGADIUBA, Jennifer (FNAL); GRACZYK, Maksymilian (Imperial College (GB)); PIERINI, Maurizio (CERN); GHIEMMETTI, Nicolo (CERN); SUMMERS, Sioni

Paris (CERN); AARRESTAD, Thea (CERN); JAMES, Thomas Owen (CERN); NGUYEN, Thong (California Institute of Technology (US)); LONCAR, Vladimir (CERN); WU, Zhenbin (University of Florida (US))

Presenter: GOVORKOVA, Katya (CERN)

Session Classification: Track 2: Data Analysis - Algorithms and Tools

Track Classification: Track 2: Data Analysis - Algorithms and Tools



Contribution ID: 623 Contribution code: **contribution ID 623**

Type: **Poster**

Machine Learning at 40 MHz with hls4ml

The hls4ml project started to bring Neural Network inference to the L1 trigger system of the LHC experiments. Since its initial proposal, the library has grown, integrating support for multiple backends, multiple network architectures (convolutional, recurrent, graph), extreme quantization (binary and ternary networks), and multiple applications (classification, regression, anomaly detection). Thanks to a collaboration with Google, it was interfaced to QKeras to enhance network compression capabilities through quantization aware training. In this talk we review the status of the project and its new features and its possible applications for LHC Run III and High-Luminosity LHC.

Significance

References

Speaker time zone

No preference

Primary authors: POL, Adrian Alan (CERN); RANKIN, Dylan Sheldon (Massachusetts Inst. of Technology (US)); DUARTE, Javier Mauricio (Fermi National Accelerator Lab. (US)); NGADIUBA, Jennifer (FNAL); GOVORKOVA, Katya (CERN); GRACZYK, Maksymilian (Imperial College (GB)); PIERINI, Maurizio (CERN); TRAN, Nhan (Fermi National Accelerator Lab. (US)); GHIEMMETTI, Nicolo (CERN); HARRIS, Philip Coleman (Massachusetts Inst. of Technology (US)); SUMMERS, Sioni Paris (CERN); AARRESTAD, Thea (CERN); LONCAR, Vladimir (CERN); WU, Zhenbin (University of Florida (US))

Presenter: LONCAR, Vladimir (CERN)

Session Classification: Posters: Crystal

Track Classification: Track 2: Data Analysis - Algorithms and Tools

Contribution ID: 573 Contribution code: **contribution ID 573**Type: **Poster**

Lightweight Jet Identification and Reconstruction as an Object Detection Task

In this contribution, we apply deep learning object detection techniques based on convolutional blocks to jet identification and reconstruction problem encountered at the CERN Large Hadron Collider. Particles reconstructed through the Particle Flow algorithm can be represented as an image composed of calorimeter and tracker cells as an input to a Single Shot Detection network. The algorithm, called PFJet-SSD is able to perform localization, classification and additional regression tasks to measure jet features in a single feed-forward pass. Besides this parallelization, we gain additional acceleration from network slimming, homogeneous quantization and optimized runtime for meeting memory and latency constraints. We compare the Ternary Weight Network (TWN) with weights constrained to $\{-1, 0, 1\}$ with per layer- and channel-dependent scaling factors to networks running with an 8-bit fixed point and a 32-bit floating-point precision. We show that the Ternary Weight Network closely matches the performance of its full-precision equivalent while all the variants of PFJet-SSD outperform the physics baseline. Finally, we report the inference latency on different hardware platforms and discuss future applications.

Significance

The proposed PFJet-SSD solution builds on the previously proposed Jet-SSD algorithm (in Reference). Besides multiple architectural changes (e.g. grouped convolutions, attention modules), PFJet-SSD tackles a more challenging dataset (with added pile-up). We introduce a physics baseline and present accuracy for multiple quantization setups. We compare inference latency estimates for multiple hardware platforms/runtimes. With these results, we are able to discuss the feasibility of the deployment of PFJet-SSD in the production system.

References

<https://arxiv.org/abs/2105.05785>

Speaker time zone

Compatible with Europe

Primary authors: POL, Adrian Alan (CERN); Mrs KLEMPNER, Anat (CEVA Inc.); NGADIUBA, Jennifer (FNAL); GOVORKOVA, Katya (CERN); PIERINI, Maurizio (CERN); Mrs SIRKIN, Olya (CEVA Inc.); Mr HALILY, Roi (CEVA Inc.); SUMMERS, Sioni Paris (CERN); Mr KOPETZ, Tal (CEVA Inc.); AARRESTAD, Thea (CERN); LONCAR, Vladimir (CERN)

Presenter: POL, Adrian Alan (CERN)

Session Classification: Posters: Broccoli

Track Classification: Track 2: Data Analysis - Algorithms and Tools



Contribution ID: 244

Type: Poster

Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml

Thursday, 27 October 2022 11:00 (30 minutes)

Recurrent neural networks have been shown to be effective architectures for many tasks in high energy physics, and thus have been widely adopted. Their use in low-latency environments has, however, been limited as a result of the difficulties of implementing recurrent architectures on field-programmable gate arrays (FPGAs). In this paper we present an implementation of two types of recurrent neural network layers—long short-term memory and gated recurrent unit—within the hls4ml [1] framework. We demonstrate that our implementation is capable of producing effective designs for both small and large models, and can be customized to meet specific design requirements for inference latencies and FPGA resources. We show the performance and synthesized designs for multiple neural networks, many of which are trained specifically for jet identification tasks at the CERN Large Hadron Collider.

[1] J. Duarte et al., “Fast inference of deep neural networks in FPGAs for particle physics”, JINST 13 (2018) P07027, arXiv:1804.06913

Significance

RNNs have shown substantial success for many tasks in particle physics. They are particularly well-suited to those problems involving sequences of particle or detector signals, outperforming densely connected deep neural networks (DNNs) and convolution neural networks (CNNs) on certain jet classification tasks. In spite of this success, RNNs have not seen widespread adoption in ultra-low latency environments in physics when compared to DNNs and CNNs. This difference is owed in part to tools such as hls4ml that simplify the adaptation of the latter models from Keras to HLS. The support for GRUs and LSTMs in hls4ml that we present in this work represents the removal of a major barrier to the use of RNNs in ultra-low latency environments. This has ramifications not only for high energy physics but also other research areas where RNNs have become popular. While we have focused on the usage of hls4ml with FPGAs, it is important to note that hls4ml can also be used to create ASIC designs, and thus this work also allows for the possibility of RNN usage on ASICs as well. The recurrent or repeating nature of many modern algorithms, such as RNNs, transformers and graph neural networks, make them very difficult to be run, particularly at low latency, on FPGAs. In this work, we present the successful deployment of RNNs in models with number of trainable parameters ranging from $O(1\text{ k})$ to $O(100\text{ k})$ achieving latencies of $O(1\text{ s})$ to $O(100\text{ s})$. This represents an important step in enabling support in hls4ml for more complex architectures with recursive computations.

References

<https://arxiv.org/abs/2207.00559>

Experiment context, if any

Primary authors: WANG, Aaron; VERNIERI, Caterina (SLAC National Accelerator Laboratory (US)); PAIKARA, Chaitanya (University of Washington); RANKIN, Dylan Sheldon (Massachusetts Inst. of Technology (US)); KHODA, Elham E (University of Washington (US)); KAGAN, Michael Aaron (SLAC National Accelerator Laboratory (US)); HARRIS, Philip Coleman (Massachusetts Inst. of Technology (US)); TEIXEIRA DE LIMA, Rafael (SLAC National Accelerator Laboratory (US)); RAO, Richa (University of Washington); HAUCK, Scott; HSU, Shih-Chieh (University of Washington Seattle (US)); SUMMERS, Sioni Paris (CERN); LONCAR, Vladimir (CERN)

Presenter: KHODA, Elham E (University of Washington (US))

Session Classification: Poster session with coffee break

Track Classification: Track 1: Computing Technology for Physics Research

Design and first test results of a reconfigurable autoencoder on an ASIC for data compression at the HL-LHC

Monday, 3 October 2022 14:00 (5 minutes)

The High Granularity Calorimeter (HGCAL) is a new subdetector of the CMS experiment in development as part of the upgrades for the High Luminosity LHC. The HGCAL readout system includes the Endcap Trigger Concentrator (ECON-T) ASIC, responsible for algorithmically reducing the immense data volume associated with the trigger patch of this six-million channel “imaging” calorimeter. To accomplish the data reduction, a reconfigurable autoencoder algorithm has been implemented in the ECON-T. The design, optimization, and implementation of this neural network encoder and first test results of the functionality within the ECON-T ASIC prototype are presented.

Primary authors: SHENAI, Alpana (Fermi National Accelerator Lab. (US)); SYAL, Chinar (Fermi National Accelerator Lab. (US)); HERWIG, Christian (Fermi National Accelerator Lab. (US)); MANTILLA SUAREZ, Cristina Ana (Fermi National Accelerator Lab. (US)); Dr GINGU, Cristinel Veniamin (Fermi National Accelerator laboratory); NOONAN, Danny (Fermi National Accelerator Lab. (US)); BRAGA, Davide (FERMILAB); COKO, Duje (University of Split. Fac.of Elect. Eng., Mech. Eng. and Nav.Architect. (HR)); FAHIM, Farah (Fermilab); DI GUGLIELMO, Giuseppe (Fermilab); HOFF, James (Fermi National Accelerator Lab. (US)); DUARTE, Javier Mauricio (Univ. of California San Diego (US)); NGADIUBA, Jennifer (FNAL); HIRSCHAUER, Jim (Fermi National Accelerator Lab. (US)); WILSON, Jon (Baylor University (US)); KWOK, Ka Hei Martin (Fermi National Accelerator Lab. (US)); MIRANDA, Llovizna (Northwestern University); Mr VALENTIN, Manuel (Northwestern University); LUPI, Matteo (CERN); PIERINI, Maurizio (CERN); TRAN, Nhan (Fermi National Accelerator Lab. (US)); KLABBERS, Pamela (Fermi National Accelerator Laboratory); RUBINOV, Paul Michael (Fermi National Accelerator Lab. (US)); HARRIS, Philip Coleman (Massachusetts Inst. of Technology (US)); WICKWIRE, Ralph Owen (Fermilab); MEMIK, Seda (Northwestern University); SUMMERS, Sioni Paris (CERN); LONCAR, Vladimir; WANG, Xiaoran (Fermi National Accelerator Lab. (US)); LUO, Yingyi (Northwestern University)

Presenter: NOONAN, Danny (Fermi National Accelerator Lab. (US))

Session Classification: Contributed Talks

Quantized ONNX (QONNX)

Monday, 3 October 2022 14:15 (15 minutes)

One of the products of the cooperation between the hls4ml and FINN groups is `Quantized ONNX (QONNX)`, a simple but flexible method to represent uniform quantization in ONNX. Its goal is to provide a high-level representation that can be targeted by training frameworks while minimizing reliance on implementation-specific details. It should also be lightweight, only adding a small number of operators. QONNX accomplishes this by being in the fused quantize-dequantize (QDQ) style. The main operator is the `Quant` operator, which takes a bitwidth, scale, and zero-offset to quantize an input tensor and then immediately dequantizes it, undoing the scale and zero offset. The resulting values are (quantized) floating point numbers, which can be used by standard ONNX operators. There is also a `BipolarQuant` operator, which is like the regular `Quant` operator but specialized for binary quantization. Finally there is a `Trunc` operator to truncate the least significant bits. Currently `Brevitas`, a PyTorch research library for quantization-aware training (QAT), and `QKeras`, a Keras library for QAT, can produce QONNX. `HAWQ` support is being added, and is the focus of a separate abstract.

The FINN and hls4ml groups also worked on a common set of utilities to ease the ingestion of QONNX by the FINN and hls4ml software. These utilities simplify the ONNX graphs by doing such things as folding constants, inferring dimensions, making sure nodes are named—commonly referred to as cleaning. FINN and hls4ml also prefer convolution data to be in a channels-last format, so we have a common pass to convert the ONNX graphs to a channels-last format using custom operators. We also have some common optimizers to, for example, change Gemm operators to lower level MatMul and Add operators so that FINN and hls4ml do not need to handle Gemm explicitly.

We will also present how hls4ml ingests QONNX. Given the high-level nature of QONNX, a direct implementation, dequantizing right after quantizing, does not map well to hardware. Instead, hls4ml makes use of optimizers to convert the abstract graph to something that can be more easily implemented on an FPGA or ASIC. In particular, the scale and zero-point in a quantization and in dequantization, if not one and zero respectively, are logically stripped from the quantization operation, resulting in three operations: scale and offset, quantization, and unscale and de-offset. The unscaling can then often be propagated down across linear operations like matrix multiplies or convolutions, to produce quantized dense or convolution layers. As an optimization, for power-of-two scales and zero offsets, we can offload the scale propagation to the HLS compiler by using fixed precision numbers, and for quantized constant weights, we can merge the scale/offset and quantization into the weights, only leaving an unscale and de-offset node if needed.

We also introduce a QONNX model zoo to share quantized neural networks in the QONNX format.

Primary authors: PAPPALARDO, Alessandro (AMD Adaptive and Embedded Computing Group (AECG) Labs); HAWKS, Benjamin (Fermi National Accelerator Lab); BORRAS, Hendrik (Uni Heidelberg); DUARTE, Javier Mauricio (Univ. of California San Diego (US)); MITREVSKI, Jovan (Fermi National Accelerator Lab. (US)); MUHIZI, Jules (Fermilab/Harvard University); TRAHMS, Matthew (UW ACME Lab); BLOTT, Michaela (AMD Adaptive and Embedded Computing Group (AECG) Labs); TRAN, Nhan (Fermi National Accelerator Lab. (US)); HAUCK, Scott; HSU, Shih-Chieh (University of Washington Seattle (US)); SUMMERS, Sioni Paris (CERN); LONCAR, Vladimir; UMUROGLU, Yaman

Presenter: MITREVSKI, Jovan (Fermi National Accelerator Lab. (US))

Session Classification: Contributed Talks

Fast recurrent neural networks on FPGAs with hls4ml

Tuesday, 4 October 2022 14:45 (15 minutes)

Recurrent neural networks have been shown to be effective architectures for many tasks in high energy physics, and thus have been widely adopted. Their use in low-latency environments has, however, been limited as a result of the difficulties of implementing recurrent architectures on field-programmable gate arrays (FPGAs). In this paper we present an implementation of two types of recurrent neural network layers—long short-term memory and gated recurrent unit—within the hls4ml [1] framework. We demonstrate that our implementation is capable of producing effective designs for both small and large models, and can be customized to meet specific design requirements for inference latencies and FPGA resources. We show the performance and synthesized designs for multiple neural networks, many of which are trained specifically for jet identification tasks at the CERN Large Hadron Collider.

[1] J. Duarte et al., “Fast inference of deep neural networks in FPGAs for particle physics”, JINST 13 (2018) P07027, arXiv:1804.06913

Primary authors: WANG, Aaron; VERNIERI, Caterina (SLAC National Accelerator Laboratory (US)); Mr PAIKARA, Chaitanya (University of Washington); RANKIN, Dylan Sheldon (Massachusetts Inst. of Technology (US)); KHODA, Elham E (University of Washington (US)); KAGAN, Michael Aaron (SLAC National Accelerator Laboratory (US)); HARRIS, Philip Coleman (Massachusetts Inst. of Technology (US)); TEIXEIRA DE LIMA, Rafael (SLAC National Accelerator Laboratory (US)); Ms RAO, Richa (University of Washington); HAUCK, Scott; HSU, Shih-Chieh (University of Washington Seattle (US)); SUMMERS, Sioni Paris (CERN); LONCAR, Vladimir

Presenter: KHODA, Elham E (University of Washington (US))

Session Classification: Contributed Talks

End-to-End Vertex Finding for the CMS Level-1 Trigger

Monday, 3 October 2022 16:00 (15 minutes)

The High Luminosity LHC provides a challenging environment for fast trigger algorithms; increased numbers of proton-proton interactions per collision will introduce more background energy in the detectors making triggering on interesting physics signatures more challenging. To help mitigate the effect of this higher background the highest energy interaction in an event can be found and other detector signatures can be associated with it. This primary vertex finding at the CMS Level-1 trigger will be performed within a latency of 250 ns. This work presents an end-to-end neural network based approach to vertex finding and track to vertex association. The network possesses simultaneous knowledge of all stages in the reconstruction chain, which allows for end-to-end optimisation. A quantised and pruned version of the neural network, split into three separate sub networks, is deployed on an FPGA using the hls4ml tools rerun through Xilinx vitis hls to take advantage of optimised pipelining. A custom hls4ml tool for convolutional neural networks that allows fully parallel input is used to ensure the strict latency requirements are met. Hardware demonstration of the network on a prototype Level-1 trigger processing board will also be shown.

Primary author: BROWN, Christopher Edward (Imperial College (GB))

Co-authors: BUNDOCK, Aaron (University of Bristol (GB)); TAPPER, Alex (Imperial College London); RAD-BURN-SMITH, Benjamin (Imperial College (GB)); KOMM, Matthias (Deutsches Elektronen-Synchrotron (DE)); PIERINI, Maurizio (CERN); SUMMERS, Sioni Paris (CERN); LONCAR, Vladimir (CERN)

Presenters: RADBURN-SMITH, Benjamin (Imperial College (GB)); BROWN, Christopher Edward (Imperial College (GB))

Session Classification: Contributed Talks



Contribution ID: 144

Type: Poster

Design and first test results of the CMS HGCAL ECON-T ASIC including an autoencoder-inspired neural network for on-detector data compression

Tuesday, 20 September 2022 16:40 (20 minutes)

The CMS experiment will replace its endcap calorimeters with a High Granularity Endcap Calorimeter (HGCAL) as part of the upgrades for High Luminosity LHC. The HGCAL readout system includes the Endcap Trigger Concentrator (ECON-T) ASIC to help manage the immense data volume associated with the trigger path of this six-million channel “imaging” calorimeter. Each ECON-T ASIC handles 15.36 Gbps of HGCROC trigger data and performs up to 12x data reduction by means of four user-selectable algorithms for data selection or compression. The design and first test results of the ECON-T ASIC are presented.

Summary (500 words)

The HGCAL is a 47-layer sampling calorimeter composed of a front electromagnetic (ECAL) section and rear hadronic section, including both silicon and plastic scintillator as active materials. The trigger readout system consists of the HGCROC ASIC for digitization, the ECON-T ASIC for data reduction, and the lpGBT ASIC for data serialization to 10.24 Gbps. With approximately 6 million readout channels, 10 bits of charge and 10 bits of time information per channel per LHC bunch crossing, the inherent data volume is approximately 5 petabits per second. This volume is reduced to about 300 Tb/s by reading out every other layer in the ECAL section, 4x or 9x ganging of sensor channels into trigger cells (TC) within the HGCROC, and using a 7-bit floating point encoding for each TC. The ECON-T ASIC further reduces the data volume to approximately 40 Tb/s by means of four user-selectable algorithms for data selection or compression, which allows readout of the entire HGCAL trigger path with about 9k optical links at 10.24 Gbps each. The ECON-T ASIC is required to operate in a radiation environment up to 200 Mrad, with power consumption of 2.5 mW per channel (500 mW per ASIC), and latency of 500 ns. The ECON-T algorithms include a threshold algorithm, which reads out TC exceeding a programmable threshold; a super-TC algorithm which combines data from adjacent TC; a ranked-choice algorithm which sorts and reads out the largest TC up to a programmable number of TC; and an autoencoder-inspired, configurable neural network which provides lossy data compression up to 7x. The ECON-T ASIC was fabricated in 2021 as a full functionality prototype. Functionality and radiation testing began in December 2021. The design as well as results of full functionality testing and radiation characterization are presented.

Primary authors: SHENAI, Alpana (Fermi National Accelerator Lab. (US)); GINGU, Cristian (Fermilab); BRAGA, Davide (FERMILAB); COKO, Duje (University of Split. Fac.of Elect. Eng., Mech. Eng. and Nav.Architect. (HR)); HIRSCHAUER, Jim (Fermi National Accelerator Lab. (US)); SYAL, Chinara (Fermi National Accelerator Lab. (US)); MANTILLA SUAREZ, Cristina Ana (Fermi National Accelerator Lab. (US)); NOONAN, Danny (Fermi National Accelerator Lab. (US)); HOFF, James (Fermi National Accelerator Lab. (US)); WILSON, Jonathan (Baylor University); LUPI, Matteo (CERN); KLABBERS, Pamela (Fermi National Accelerator Laboratory); RUBINOV, Paul Michael (Fermi National Accelerator Lab. (US)); WICKWIRE, Ralph Owen (Fermilab); WANG, Xiaoran (Fermi National Accelerator Lab. (US)); MEMIK, Seda (Northwestern University); HARRIS, Philip Coleman (Massachusetts Inst. of Technology)

(US)); DUARTE, Javier Mauricio (Univ. of California San Diego (US)); HERWIG, Christian (Fermi National Accelerator Lab. (US)); VALENTIN, Manuel (Northwestern University); PIERINI, Maurizio (CERN); NGADIUBA, Jennifer (FNAL); MIRANDA, Llovizna (Northwestern University); GUGLIELMO, Giuseppe Di; LONCAR, Vladimir (CERN); LUO, Yingyi (Northwestern University); TRAN, Nhan (Fermi National Accelerator Lab. (US)); SUMMERS, Sioni Paris (CERN); KWOK, Ka Hei Martin (Fermi National Accelerator Lab. (US)); FAHIM, Farah (Fermilab); DE OLIVEIRA, Rui (CERN); DATAO, Gong (University of Minnesota); KREMASTIOTIS, Iraklis (CERN); KULIS, Szymon (CERN); VICENTE LEITAO, Pedro (CERN); LEROUX, Paul (Katholieke Universiteit Leuven); RODRIGUES SIMOES MOREIRA, Paulo (CERN); PRINZIE, Jeffrey; Prof. GUO, Di (Central China Normal University); Dr SUN, Quan (Fermi National Accelerator Lab.); YE, Jingbo (Southern Methodist University (US)); BRAM, Feas (Katholieke Universiteit Leuven); DONGXU, Yang (Southern Methodist University); HAMMER, Mike (Argonne National Laboratory)

Presenter: HOFF, James (Fermi National Accelerator Lab. (US))

Session Classification: Tuesday posters session

Track Classification: ASIC



Contribution ID: 55

Type: **Lightning Talk**

Graph Neural Networks on FPGAs with HLS4ML

Monday, 25 September 2023 18:10 (5 minutes)

Graph structures are a natural representation of data in many fields of research, including particle and nuclear physics experiments, and graph neural networks (GNNs) are a popular approach to extract information from that. Simultaneously, there is often a need for very low-latency evaluation of GNNs on FPGAs. The HLS4ML framework for translating machine learning models from industry-standard Python implementations into optimized HLS code suitable for FPGA applications has been extended to support GNNs constructed using PyTorch Geometric (PyG). To that end, the parsing of general PyTorch models using symbolic tracing using the torch.FX package has been added to HLS4ML. This approach has been extended to enable parsing of PyG models and support for GNN-specific operations has been implemented. To demonstrate the performance of the GNN implementation in HLS4ML, a network for track reconstruction in the sPHENIX experiment is used. Future extensions, such as an interface to quantization-aware training with Brevitas, are discussed.

Primary authors: SCHULTE, Jan-Frederik (Purdue University (US)); LIU, Miaoyuan (Purdue University (US)); HARRIS, Philip Coleman (Massachusetts Inst. of Technology (US)); LONCAR, Vladimir (Massachusetts Inst. of Technology (US))

Presenter: SCHULTE, Jan-Frederik (Purdue University (US))

Session Classification: Contributed Talks

Track Classification: Contributed Talks



Contribution ID: 12

Type: **Lightning Talk**

Jets as sets or graphs: Fast jet classification on FPGAs for efficient triggering at the HL-LHC

Monday, 25 September 2023 17:40 (5 minutes)

The upcoming high-luminosity upgrade of the LHC will lead to a factor of five increase in instantaneous luminosity during proton-proton collisions. Consequently, the experiments situated around the collider ring, such as the CMS experiment, will record approximately ten times more data. Furthermore, the luminosity increase will result in significantly higher data complexity, thus making more sophisticated and efficient real-time event selection algorithms an unavoidable necessity in the future of the LHC.

One particular facet of the looming increase in data complexity is the availability of information pertaining to the individual constituents of a jet at the first stage of the event filtering system, known as the level-1 trigger. Therefore, more intricate jet identification algorithms that utilise this additional constituent information can be designed if they meet the strict latency, throughput, and resource requirements. In this work, we construct, deploy, and compare fast machine-learning algorithms, including graph- and set-based models, that exploit jet constituent data on field-programmable gate arrays (FPGAs) to perform jet classification. The latencies and resource consumption of the studied models are reported. Through quantization-aware training and efficient FPGA implementations, we show that $O(100)$ ns inference of complex models like graph neural networks and deep sets is feasible at low resource cost.

Primary authors: SZNAJDER, Andre (Universidade do Estado do Rio de Janeiro (BR)); ODAGIU, Denis-Patrick (ETH Zurich (CH)); Prof. DISSERTORI, Guenther (ETH Zurich (CH)); DUARTE, Javier Mauricio (Univ. of California San Diego (US)); AARRESTAD, Thea (ETH Zurich (CH)); , Vladimir (); LUK, Wayne; QUE, Zhiqiang (Walkie) (Imperial College London)

Presenter: ODAGIU, Denis-Patrick (ETH Zurich (CH))

Session Classification: Contributed Talks

Track Classification: Contributed Talks



Contribution ID: 8

Type: **Standard Talk**

Hardware-aware pruning of real-time neural networks with hls4ml Optimization API

Wednesday, 27 September 2023 13:30 (15 minutes)

Neural networks achieve state-of-the-art performance in image classification, medical analysis, particle physics and many more application areas. With the ever-increasing need for faster computation and lower power consumption, driven by real-time systems and Internet-of-Things (IoT), field-programmable gate arrays (FPGAs) have emerged as suitable accelerators for deep learning applications. Due to the high computational complexity and memory footprint of neural networks, various compression techniques, such as pruning, quantisation and knowledge distillation, have been proposed in literature. Pruning sparsifies a neural network, reducing the number of multiplications and memory. However, unstructured pruning often fails to capture properties of the underlying hardware, bottlenecking improvements and causing load-balance inefficiency on FPGAs.

We propose a hardware-centric formulation of pruning, by formulating it as a knapsack problem with parallelisation-aware tensor structures. The primary emphasis is on real-time inference, with latencies of order $1\mu\text{s}$. We evaluate our method on a range of tasks, including jet tagging at CERN's Large Hadron Collider and fast image classification (SVHN, Fashion MNIST). The proposed method achieves reductions ranging between 55% and 92% in digital signal processing blocks (DSPs) and up to 81% in block memory (BRAM), with inference latencies ranging between 105ns and $205\mu\text{s}$.

The proposed algorithms are integrated with hls4ml and open-sourced with an Apache 2.0 licence, enabling an end-to-end tool for hardware-aware pruning and real-time inference. Furthermore, the tools are readily integrated with QKeras, enabling pruning and inference of models trained with quantisation-aware training. Compared to TensorFlow Model Optimization, hls4ml Optimization API offers advanced functionality, including support for structured pruning, gradient-based ranking methods and integration with model reduction libraries, such as Keras Surgeon. Furthermore, by enabling multiple levels of pruning granularity, the software can target a wide range of hardware platforms. Through integration with hls4ml, an open-source, end-to-end system is built, allowing practitioners from a wide range of fields to compress and accelerate neural networks suited for their applications.

Primary authors: RAMHORST, Benjamin (Imperial College London); Prof. CONSTANTINIDES, George (Imperial College London); Dr LONČAR, Vladimir (Massachusetts Institute of Technology)

Presenter: RAMHORST, Benjamin (Imperial College London)

Session Classification: Contributed Talks

Track Classification: Contributed Talks

Adapting Skip Connections for Resource-Efficient FPGA Inference

Authors: [Olivia Weng](#), [Gabriel Marcano](#), [Vladimir Loncar](#), [Alireza Khodamoradi](#), [Nojan Sheybani](#), [Farinaz Koushanfar](#), [Kristof Denolf](#), [Javier Mauricio Duarte](#), [Ryan Kastner](#) [Authors Info & Claims](#)

FPGA '23: Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays • February 2023 • Pages 229
• <https://doi.org/10.1145/3543622.3573172>

Published: 12 February 2023 [Publication History](#) [Check for updates](#)

0 0

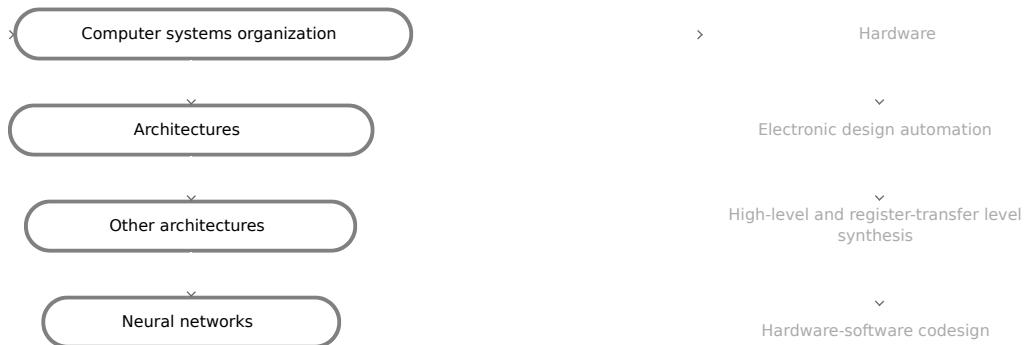


ABSTRACT

Deep neural networks employ skip connections - identity functions that combine the outputs of different layers-to improve training convergence; however, these skip connections are costly to implement in hardware. In particular, for inference accelerators on resource-limited platforms, they require extra buffers, increasing not only on- and off-chip memory utilization but also memory bandwidth requirements. Thus, a network that has skip connections costs more to deploy in hardware than one that has none. We argue that, for certain classification tasks, a network's skip connections are needed for the network to learn but not necessary for inference after convergence. We thus explore removing skip connections from a fully-trained network to mitigate their hardware cost. From this investigation, we introduce a fine-tuning/retraining method that adapts a network's skip connections - by either removing or shortening them-to make them fit better in hardware with minimal to no loss in accuracy. With these changes, we decrease resource utilization by up to 34% for BRAMs, 7% for FFs, and 12% LUTs when implemented on an FPGA.

Index Terms

Adapting Skip Connections for Resource-Efficient FPGA Inference



Recommendations

