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

MACHINE LEARNING
Science and Technology

OPEN ACCESS

RECEIVED
26 September 2022

REVISED
15 February 2023

ACCEPTED FOR PUBLICATION
1 March 2023

PUBLISHED
10 April 2023

PAPER

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 Harris[2], Scott Hauck[1], Shih-Chieh Hsu[1], Michael Kagan[3], Vladimir Loncar[2], Chaitanya Paikara[1], Richa Rao[1], Sioni Summers[4], Caterina Vernieri[3] and Aaron Wang[1]

[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 rafaeltl@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 μs). 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.

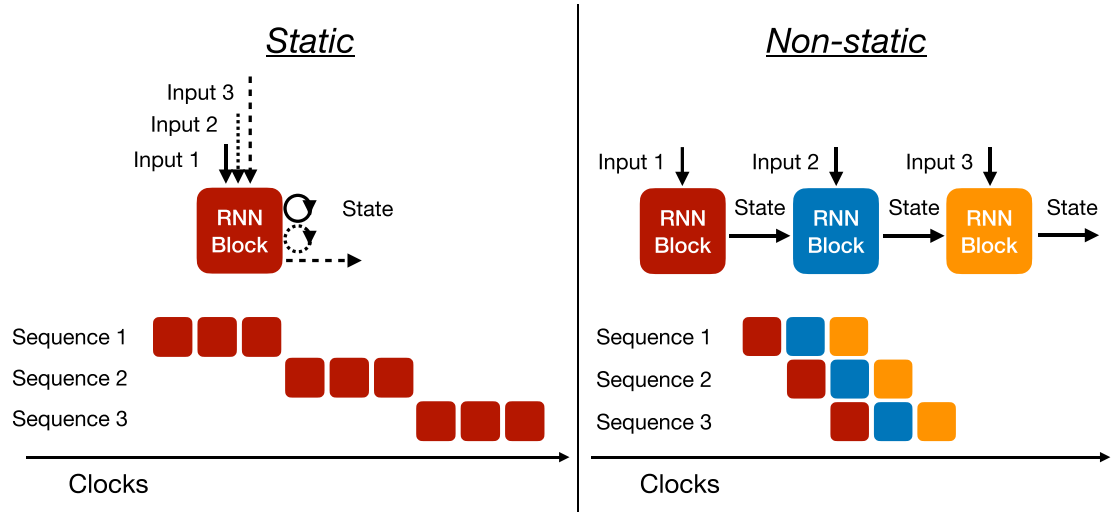


Figure 1. Schematics of the two RNN modes: static (left) and non-static (right). A sketch of the latency and pipelining is shown at the bottom for each mode, and illustrates the latency advantage of non-static mode at the cost of resources.

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 use: 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 ($p_{\mathrm{T}}(\mathrm{track})/p_{\mathrm{T}}(\mathrm{jet})$); $\Delta R(\mathrm{track}, \mathrm{jet})$; transverse and longitudinal impact parameters (d_0, d_z) and their significances ($\mathcal{S}(d_0)$, $\mathcal{S}(d_z)$).

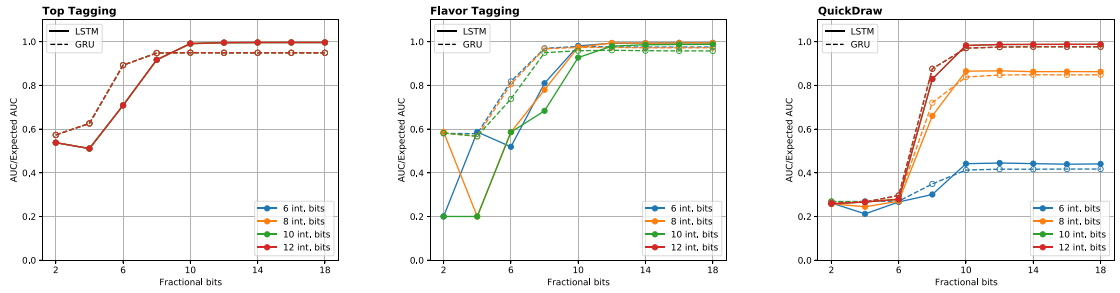
Flavor tagging models were constructed using Keras/TensorFlow, using either GRU or LSTM layers. The padded sequence of tracks, with maximum length of 15, is fed into either one recurrent layer (GRU or LSTM) with 120 hidden units. The recurrent layer outputs a representation of the padded sequence directly 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.



(a) Top quark tagging (b) Jet flavor tagging (c) QuickDraw

Figure 2. Ratios of the fixed-point and floating-point AUCs as function of fractional bits for the (a) top quark tagging, (b) jet flavor tagging, and (c) QuickDraw models. The precision of the integer part is kept fixed to 6 (blue), 8 (orange), 10 (green), and 12 (red) bits. All four lines overlap for the top quark tagging GRU (dashed) and LSTM (solid) models.

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-flvb2104-2-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.

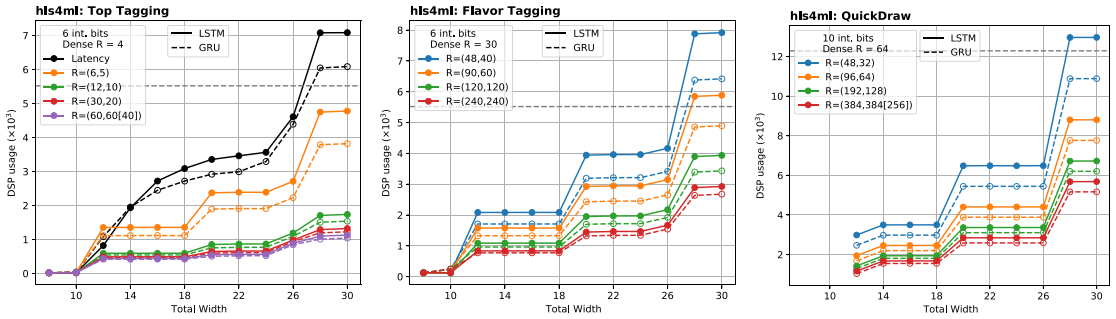
Model	Latency (μs)	$R = (6,5)$ (μs)	$R = (12,10)$ (μs)	$R = (30,20)$ (μs)	$R = (60,60\ [40])$ (μs)
GRU	1.7–1.7	2.4–6.5	3.2–7.3	5.0–9.1	8.0–12.1
LSTM	1.7–1.7	2.7–6.8	3.5–7.6	5.3–9.4	8.3–12.4

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

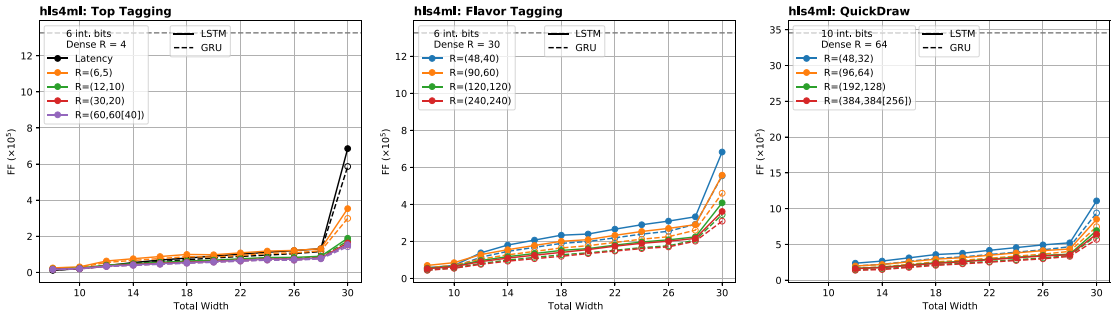


(a) Top quark tagging (b) Jet flavor tagging (c) QuickDraw

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.

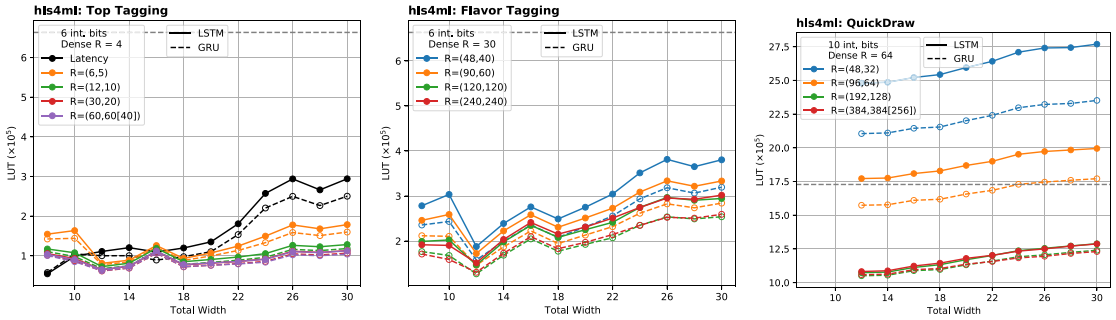
5.2. Parallelization

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



(a) Top quark tagging (b) Jet flavor tagging (c) QuickDraw

Figure 4. FF 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. FF utilization with latency strategy is shown only for the top quark tagging models and all other lines correspond to different reuse factors. The FFs available in the target FPGA for each model are shown in the black dashed horizontal line.



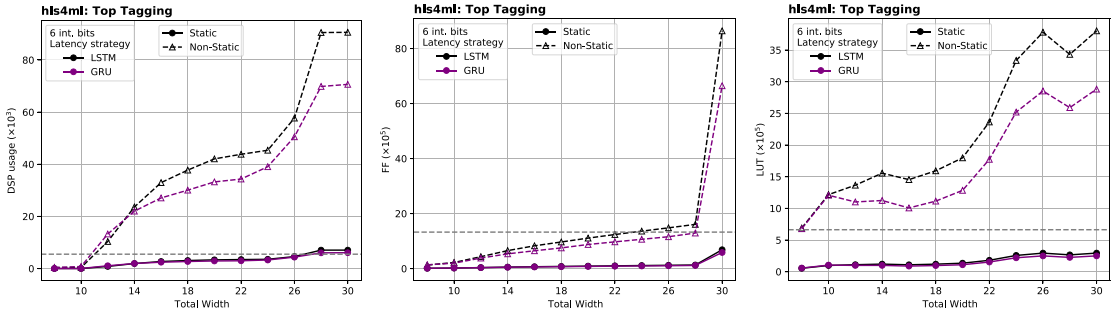
(a) Top quark tagging (b) Jet flavor tagging (c) QuickDraw

Figure 5. LUT 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. LUT utilization with latency strategy is shown only for the top quark tagging models and all other lines correspond to different reuse factors. The LUTs available in the target FPGA for each model are shown in the black dashed horizontal line.

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



(a) Top quark tagging: DSP (b) Top quark tagging: FF (c) Top quark tagging: LUT

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 $\mathcal{O}(1\,\mathrm{k})$ to $\mathcal{O}(100\,\mathrm{k})$ achieving latencies of $\mathcal{O}(1\,\mu\mathrm{s})$ to $\mathcal{O}(100\,\mu\mathrm{s})$. 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] Åarrestad 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 (FaaST) *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, Hacohen-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