Fast inference of deep neural networks in FPGAs for particle physics

To cite this article: J. Duarte *et al* 2018 *JINST* **13** P07027

View the article online for updates and enhancements.

Related content

- Using MaxCompiler for the high level synthesis of trigger algorithms
  S. Summers, A. Rose and P. Sanders

- An FPGA based track finder for the L1 trigger of the CMS experiment at the High Luminosity LHC
  R. Aggleton, L.E. Ardila-Perez, F.A. Ball et al.

- Software and firmware co-development using high-level synthesis
  N.P. Ghanathe, A. Madorsky, H. Lam et al.

Fast inference of deep neural networks in FPGAs for particle physics

J. Duarte,[a] **S. Han,**[b] **P. Harris,**[b] **S. Jindariani,**[a] **E. Kreinar,**[c] **B. Kreis,**[a] **J. Ngadiuba,**[d] **M. Pierini,**[d] **R. Rivera,**[a] **N. Tran**[a,1] **and Z. Wu**[e]

[a]*Fermi National Accelerator Laboratory, Batavia, IL 60510, U.S.A.*

[b]*Massachusetts Institute of Technology, Cambridge, MA 02139, U.S.A.*

[c]*HawkEye360, Herndon, VA 20170, U.S.A.*

[d]*CERN, CH-1211 Geneva 23, Switzerland*

[e]*University of Illinois at Chicago, Chicago, IL 60607, U.S.A.*

E-mail: hls4ml.help@gmail.com

ABSTRACT: Recent results at the Large Hadron Collider (LHC) have pointed to enhanced physics capabilities through the improvement of the real-time event processing techniques. Machine learning methods are ubiquitous and have proven to be very powerful in LHC physics, and particle physics as a whole. However, exploration of the use 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, among many other physics scenarios, searches for new dark sector particles and novel measurements of the Higgs boson. While we focus on a specific example, the lessons are far-reaching. A companion compiler package for this work 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 identify the problems in particle physics that would benefit from performing neural network inference with FPGAs. For our example jet substructure model, we fit well within the available resources of modern FPGAs with a latency on the scale of 100 ns.

[1]Corresponding author.

Contents

1 Introduction

Over the last several decades, as physicists have pursued a deeper understanding of particle physics phenomena, particle detectors have been made larger, more granular, and capable of processing data at ever increasing rates. This has led to a dramatic increase in data volumes that need to be efficiently analyzed in real time to reconstruct and filter events of interest. Machine learning (ML) methods deployed in the final stages of data processing have been demonstrated to be extremely effective in many different tasks across particle physics. Thus far, their use in real-time selection hardware, based on Field Programmable Gate Arrays (FPGAs), has been limited due to implementation complexity and FPGA resource demands. In this study, we explore the implementation of neural networks in FPGAs, mapping out resource usage and latency for various deep neural network architectures and hyperparameters, demonstrating the feasibility of deep learning techniques in very low-latency (sub-microsecond) FPGA applications.

The Large Hadron Collider (LHC) at CERN serves as a perfect example. The LHC is the world's highest energy particle accelerator, operating at the highest data rates ever achieved. Its goal is to understand the very basic laws of nature and the building blocks of the universe. In the first run of data-taking that concluded in 2012, the ground-breaking highlight was the discovery of the Higgs boson [1, 2]. The current data-taking campaigns are devoted to a full characterization of the Higgs-boson properties and to the search for physics phenomena beyond the standard model of particle physics, including the search for dark matter.

Due to the extreme frequency of 40 MHz at which proton bunches collide at the LHC, data rates at the two multipurpose experiments, CMS [3] and ATLAS [4], are of the order of hundreds of terabytes per second. With such high data rates, there are challenges for both real time and offline

processing of collision events. The task of the real-time processing is to filter events to reduce data rates to manageable levels for offline processing is called triggering. It is typically performed in multiple stages [5, 6]; two stages are used in the CMS detector. Because of the extreme input data rates and size of the data buffers, the first stage, Level-1 (L1), of data processing typically uses custom hardware with ASICs or, increasingly, FPGAs, to handle the initial data rate using pipelined algorithms with latencies of hundreds of nanoseconds totalling microseconds. The second stage of triggering, High Level Trigger (HLT), uses commercial CPUs to process the filtered data in software with longer latencies on the timescale of hundreds of milliseconds in total.

Machine learning methods, most recently deep learning, have a wide range of applications in event processing at the LHC [7–13], from lower level energy cluster calibration and regression to high level physics object classification, e.g. jet tagging with substructure information, and physics analyses. More sophisticated ML algorithms in the trigger will allow LHC experiments to preserve potential new physics signatures such as those related to the Higgs, dark matter, and hidden sectors [14] that would otherwise be lost. This can be achieved through overall more performant trigger algorithms or fast data analysis techniques such as data scouting or trigger level analysis [15–17]. In this study, we explore the implementation of neural network inference in FPGAs by mapping out resource usage and latency for various network architectures and hyperparameters. The goal is to understand the range of possible neural network designs that may be implemented in FPGAs given the resource and latency constraints for different LHC event processing applications. As a case study, we consider the case of using ML methods for jet substructure [18] classification which, when employed in the trigger, enable searches for new dark sector particles and important measurements of the Higgs transverse momentum spectrum. The lessons learned in this case study are broadly applicable.

An important complementary result of this work is the companion compiler, `hls4ml`,[1] which translates ML models from common open-source software packages such as `Keras` [19] and `PyTorch` [20] into RTL (Register-Transfer Level) abstraction for FPGAs using High-Level Synthesis (HLS) tools, which have demonstrated considerable progress in recent years [21]. There are many types of HLS, and our particular package and the results of this study are based on Vivado HLS `2017.2` [22] though the general techniques can be used in other applications with other FPGAs.

In high energy physics, engineering support with long development cycles is required to translate physics-motivated data processing algorithms into firmware. However, engineering is a scarce and valuable resource. The `hls4ml` tool allows physicists to rapidly prototype ML algorithms for both firmware feasibility and physics performance without extensive Verilog/VHDL experience, thus greatly decreasing the time for algorithm development cycles while preserving engineering resources. We focus on the task of the FPGA-based triggers of the ATLAS and CMS experiments with algorithm latencies in the microsecond range, fully pipelined to handle the 40 MHz LHC collision rate. For this task, solutions with either CPUs or GPUs are not possible due to the severe time limitation imposed. Such latencies are unique to LHC trigger challenges, and therefore few general tools exist for this application. Nonetheless, the `hls4ml` package is a general purpose tool and is designed to serve a broad range of applicatons in particle physics and beyond, from trigger and data acquisition tasks (DAQ) to longer latency trigger tasks (milliseconds) and CPU-FPGA co-processor hardware.

---

[1]The project can be accessed at https://hls-fpga-machine-learning.github.io/hls4ml.

The rest of the paper is organized as follows. In section 2, we describe the essential concepts in implementing neural networks in FPGAs for trigger and DAQ. This includes a case study for developing a jet substructure ML algorithm for FPGA implementation. In section 3, we detail the HLS synthesis for various neural network architectures and hyperparameters and the resulting implementation on an FPGA. Finally, we summarize our findings, detail follow-up studies, and discuss potential broader applications in physics and other fields in section 4.

1.1 Related work

The inference of neural networks in FPGAs is a rapidly developing and high interest field. There exists an extensive literature on the topic. However, as it pertains to the particular task for particle physics where networks can be smaller but latency constraints are much more severe, this is the first dedicated general purpose study of this kind in the field. Nonetheless, some ML techniques have been deployed in the LHC trigger already, including a first implementation of a boosted decision tree (BDT) for muon momentum measurement [23]. An early attempt to deploy convolutional neural networks (CNNs) on FPGAs for particle physics was presented at NIPS 2017 [24].

We borrow inspiration from other works, including the RFNoC Neural Network Library [25], on which `hls4ml` is based. An overview of existing toolflows for mapping CNNs on FPGAs is given in [26]. Snowflake [27] is a scalable and efficient CNN accelerator with models specified in `Torch` [28] and a single computation architecture (sequential IO) designed to perform at near-peak hardware utilization targeting Xilinx System-on-Chips (SoCs). Caffeine [29] is another CNN accelerator for `Caffe`-specified models targeting Xilinx devices that support a SDAccel15 environment and a PCIe interface between the FPGA and a host. fpgaConvNet [30–33] converts CNNs specified in `Caffe` [34] or `Torch` formats into generated Xilinx Vivado HLS code with a streaming architecture (parallel IO). FP-DNN (Field Programmable Deep Neural Networks) [35] is a framework that takes `TensorFlow` [36]-described DNNs (CNNs, LSTM-RNNs [37], and Residual Nets) as input, and generates the hardware implementations on FPGA boards with RTL-HLS hybrid templates. DNNWeaver [38] is an open-source alternative, which also supports DNNs specified in `Caffe` format and automatically generates the accelerator Verilog code using hand-optimized Verilog templates with a high degree of portability.

The physics problem we take as a benchmark for our discussion is substructure-based jet tagging, on which there is a rich literature of deep-learning applications [39–46]. In this context, there were attempts to use CNN, RNNs, as well as physics-inspired network architectures. Jets have been represented as grayscale images, RGB images, sequences of particles, or a set of physics-inspired high-level features, as we do in this study.

2 Building neural networks with `hls4ml`

In this section, we give an overview of translating a given neural network model into a FPGA implementation using HLS. We then detail a specific jet substructure case study, but the same concepts are applicable for a broad class of problems. We conclude this section by discussing how to create an efficient and optimal implementation of a neural network in terms of performance, resource usage, and latency.

2.1 `hls4ml` concept

The task of automatically translating a trained neural network, specified by the model's architecture, weights, and biases, into HLS code is performed by the `hls4ml` package. A schematic of a typical workflow is illustrated in figure 1.
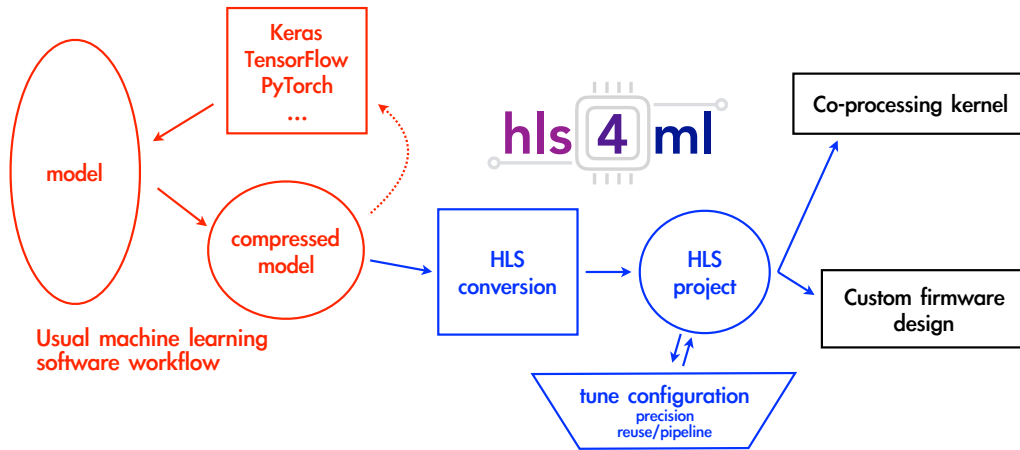
Figure 1. A typical workflow to translate a model into a FPGA implementation using `hls4ml`.

The part of the workflow illustrated in red indicates the usual software workflow required to design a neural network for a specific task. This usual machine learning workflow, with tools such as `Keras` and `PyTorch`, involves a training step and possible compression steps (more discussion below in section 2.3) before settling on a final model. The blue section of the workflow is the task of `hls4ml`, which translates a model into an HLS project that can be synthesized and implemented to run on an FPGA.

At a high level, FPGA algorithm design is unique from programming a CPU in that independent operations may run fully in parallel, allowing FPGAs to achieve trillions of operations per second at a relatively low power cost with respect to CPUs and GPUs. However, such operations consume dedicated resources onboard the FPGA and cannot be dynamically remapped while running. The challenge in creating an optimal FPGA implementation is to balance FPGA resource usage with achieving the latency and throughput goals of the target algorithm. Key metrics for an FPGA implementation include:

1. **latency**, the total time (typically expressed in units of "clocks") required for a single iteration of the algorithm to complete.

2. **initiation interval**, the number of clock cycles required before the algorithm may accept a new input. Initiation interval (often expressed as "II") is inversely proportional to the inference rate, or throughput; an initiation interval of 2 achieves half the throughput as an initiation interval of 1. Consequently, data can be pipelined into the algorithm at the rate of the initiation interval.

– 4 –

3. **resource usage**, expressed as the following FPGA resource categories: onboard FPGA memory (BRAM), digital signal processing (arithmetic) blocks (DSPs), and registers and programmable logic (flip-flops, or FFs, and lookup tables, or LUTs).

The `hls4ml` tool has a number of configurable parameters which can help the user explore and customize the space of latency, initiation interval, and resource usage tradeoffs for their application. Because every application is different, the goal of the `hls4ml` package is to empower the user to perform this optimization through automated neural network translation and FPGA design iteration. In practice, the time required to perform `hls4ml` translation of a neural netowrk is much shorter (minutes to hours) than a designing a specific neural network architecture for an FPGA, and may be used to rapidly prototype machine learning algorithms without dedicated engineering support for the FPGA implementation. For physicists, this makes designing physics algorithms for the trigger or DAQ significantly more accessible and efficient, thus has the potential for the *"time to physics"* to be greatly reduced.

We first introduce some terminology and concepts for the inference of deep, fully connected neural networks. Consider the network illustrated in figure 2 with M layers, where each layer m has N_m neurons. The input layer has N_1 input neurons and the output layer has N_M output neurons. The vector of neuron output values at each layer are denoted by $\boldsymbol{x}_m$. For the mth fully connected layer ($m > 1$),

$$\boldsymbol{x}_m = g_m \left(\boldsymbol{W}_{m,m-1} \boldsymbol{x}_{m-1} + \boldsymbol{b}_m \right) , \tag{2.1}$$

where $\boldsymbol{W}_{m,m-1}$ is the matrix of weights between layers $m-1$ and m, $\boldsymbol{b}_m$ are the bias values, and g_m is the activation function for layer m. The size of matrix $\boldsymbol{W}_{m,m-1}$ is $N_m \times N_{m-1}$ and thus the number of multiplications required to compute the neuron values of layer m is implicitly also $N_m \times N_{m-1}$.
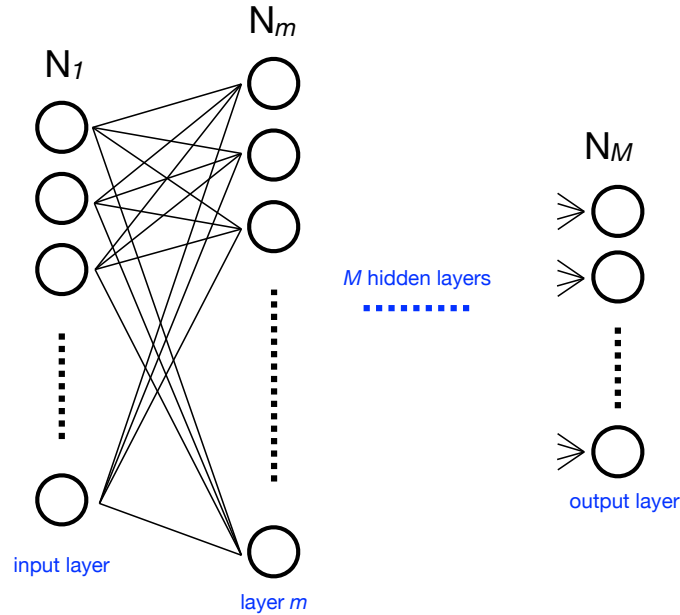
Figure 2. A cartoon of a deep, fully connected neural network illustrating the description conventions used in the text.

In `hls4ml`, the calculation of each layer x_m is performed independently and sequentially. The inference is pipelined and accepts a new set of inputs after its initiation interval, as described above. The total number of multiplications required to infer a given neural network is:

$$N_{\text{multiplications}} = \sum_{m=1}^{M} N_{m-1} \times N_m. \tag{2.2}$$

Non-trivial activation functions, such as sigmoid, softmax, and hyperbolic tangent, are precomputed for a range of input values and stored in BRAMs. The ReLU activation function is implemented in programmable logic. The effect of the neural network hyperparameters on the latency, throughput, and resource usage informs the optimal network implementation for any given application.

2.2 Case study: jet substructure

Jets are collimated showers of particles that result from the decay and hadronization of quarks q and gluons g. At the LHC, due to the high collision energy, a particularly interesting jet signature emerges from overlapping quark-initiated showers produced in decays of heavy standard model particles. For example, the W and Z bosons decay to two quarks ($q\bar{q}$) 67%–70% of the time and the Higgs boson is predicted to decay to two b-quarks ($b\bar{b}$) approximatly 58% of the time. The top quark decays to two light quarks and a b-quark ($q\bar{q}b$). It is the task of jet substructure [18, 47] to distinguish the various radiation profiles of these jets from backgrounds consisting mainly of quark (u, d, c, s, b) and gluon-initiated jets. The tools of jet substructure have been used to distinguish interesting jet signatures from backgrounds that have production rates hundreds of times larger than the signal [48].

Jet substructure at the LHC has been a particularly active field for machine learning techniques as jets contain $O(100)$ particles whose properties and correlations may be exploited to identify physics signals. The high dimensionality and highly correlated nature of the phase space makes this task an interesting testbed for machine learning techniques. There are many studies that explore this possibility, both in experiment and theory [18, 39–47, 49–51]. For this reason, we choose to benchmark our FPGA studies using the jet substructure task.

We give two examples in figure 3 where jet substructure techniques in the trigger can play an important role: low mass hidden hadronic resonances [52] and boosted Higgs produced in gluon fusion [53]. Both processes are overwhelmed by backgrounds and current trigger strategies would select only on the energy of the jet. By introducing jet substructure techniques in the hardware trigger, we can further greatly reduced backgrounds and preserve significantly more of these types of signals in the future. There are many other physics signatures which could benefit from jet substructure in the trigger. In this case study, we focus on the task of classifying a jet as either a quark (q), gluon (g), W boson (W), Z boson (Z), or top quark (t) jet .[2]

Input generation and features. Events are generated at $\sqrt{s} = 13$ TeV for comparison to LHC performance. Parton-level (unshowered quark) W^+W^-, ZZ, $t\bar{t}$, $q\bar{q}$, and gg events are first produced at leading-order using `MadGraph5_aMC_at_NLO` [54] (version 2.3.1) with the NNPDF23LO1 parton

---

[2]The Higgs boson is not included in this study as its mass and substructure are quite similar to W and Z bosons and are otherwise difficult to distinguish in absence of track vertexing information, a situation common to current FPGA-based triggers.
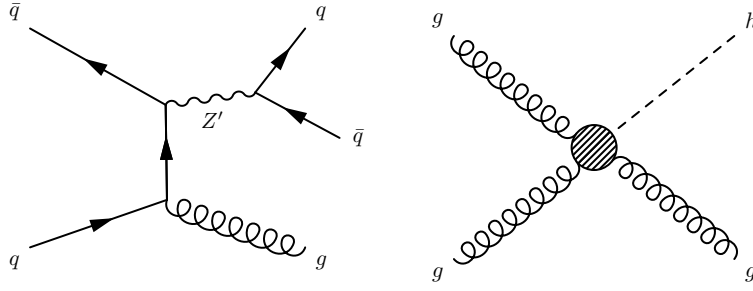
Figure 3. Example Feynman diagrams of interesting physics signatures that would benefit from jet substructure algorithms in the hardware trigger.

**Table 1**. A summary of the observables used in the analysis.

Observables
m_{mMDT}
$N_2^{\beta=1,2}$
$M_2^{\beta=1,2}$
$C_1^{\beta=0,1,2}$
$C_2^{\beta=1,2}$
$D_2^{\beta=1,2}$
$D_2^{(\alpha,\beta)=(1,1),(1,2)}$
$\sum z \log z$
Multiplicity

distribution functions (PDFs) [55]. To focus on a relatively narrow kinematic range, the transverse momenta of the partons and undecayed gauge bosons are generated in a window with energy spread given by $\delta p_{\mathrm{T}}/p_{\mathrm{T}} = 0.01$, centered at 1 TeV. These parton-level events are then decayed and showered in PYTHIA8 [56] (version 8.212) with the Monash 2013 tune [57], including the contribution from the underlying event. For each final state, 200,000 events are generated.

To build a complete list of expert features, we implement a variety of jet recombination algorithms and substructure tools via the FASTJET 3.1.3 and FASTJET CONTRIB 1.027 packages [58, 59]. As a baseline, all jets are clustered using the anti-k_{T} algorithm [60], with a distance parameter of $R = 0.8$. Even though the parton-level p_{T} distribution is narrow, the jet p_{T} spectrum is significantly broadened by kinematic recoil from the parton shower and energy migration in and out of the jet cone. We apply a cut on the reconstructed jet p_{T} to remove extreme events from the analysis, vetoing those outside a window of $0.8\,\mathrm{TeV} < p_{\mathrm{T}} < 1.6\,\mathrm{TeV}$ for the $p_{\mathrm{T}} = 1$ TeV bin.

The jet substructure community has developed a wide variety of observables to identify the origin of a jet based on the structure of its radiation pattern. In table 1, we list all the observables used in this study [61–64]. A brief description of each of these variables is presented in ref. [65]. These are used as expert-level inputs to a neural network classifier which is near optimal.[3]

---

[3]More sophisticated approaches exist, but the goal of this study is not to achieve better performance than existing algorithms. Instead, the goal is to examine the implementation of several effective neural network architectures in FPGAs.
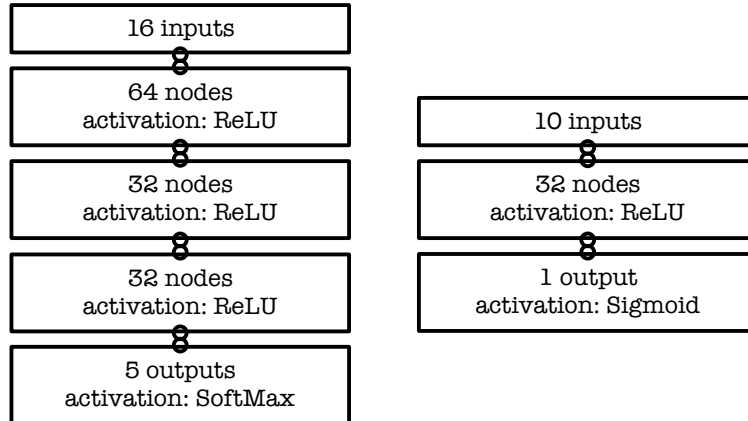
Figure 4. Two neural network architectures for jet substructure classification. (Left) A three-hidden-layer model we use to categorize five classes of jets (q, g, W, Z, and t). (Right) A one-hidden-layer model used to identify top quarks, simplified for the FPGA implementation described in section 3.3.

Benchmark networks and floating point performance. We train a neural network for the classification task of q, g, W, Z, and t discrimination. The data are randomly split into training (60%), validation (20%), and testing (20%) datasets. The input features are standardized by removing the mean and scaling to unit variance. The architecture, illustrated in figure 4 (left), is a fully-connected neural network with three hidden layers. The activation function for the hidden layers is ReLU [66] while the output layer activation function is a softmax function to provide probabilities for each class. The categorical cross-entropy loss function is minimized with and without L_1 regularization of the weights (section 2.3) using the Adam algorithm [67] with an initial learning rate of 10^{-4} and a minibatch size of 1024. The learning rate is halved if the validation loss fails to improve over 10 epochs. Training is performed on an AWS EC2 P2 GPU instance [68] with `Keras`. We also consider a simpler architecture with one hidden layer, see figure 4 (right), when studying the final FPGA implementation on a specific device. This is described further in section 3.3.

The performance of the neural network classifier is shown in figure 5. The plot is presented using the `matplotlib` package [69]. The general features of this performance plot are typical of jet substructure classification tasks. Top-quark jets, by virtue of their large mass and three-prong nature, have the best separation from the rest of the jet types. The W and Z jets are similar in performance because of their masses and two-prong nature while quark and gluon jets are notoriously challenging to classify [48]. Given this multi-jet classifier performance, we explore how to implement such a neural network architecture in an FPGA using `hls4ml`.

2.3 Efficient network design

We now present a general strategy for the translation of a deep neural network into an FPGA implementation for a specific network designed for the task of jet substructure classification which was introduced in section 2.2. We focus on tuning the network inference in a way that uses the FPGA resources efficiently and meets latency and pipelining constraints.
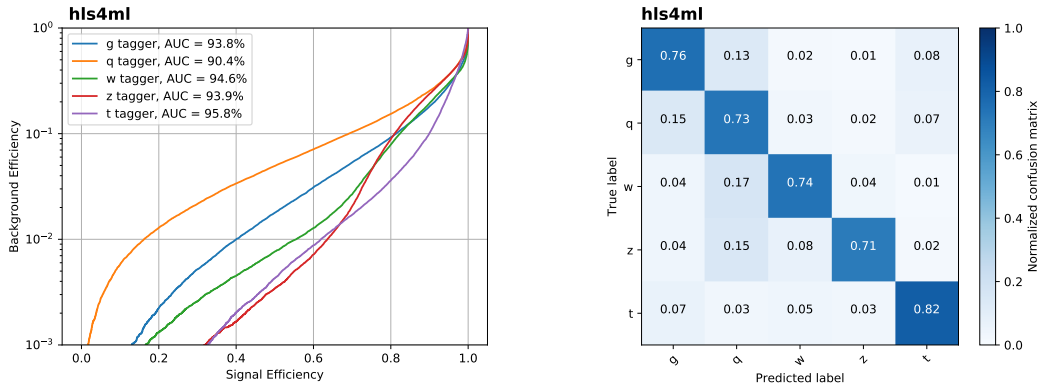
Figure 5. Performance of the deep neural network classifier: (Left) signal efficiency versus mis-identification rate for quark, gluon, *W* boson, *Z* boson, and top quark jet identification. The mis-identification rate is based on an equal admixture of the other non-signal jet types. (Right) The corresponding normalized confusion matrix for the five classes.

Neural network inference can be made efficient with the following techniques: compression, quantization, and parallelization. We summarize these ideas briefly:

- **compression**: neural network synapses and neurons can be redundant; compression attempts to reduce the number of synapses or neurons thereby effectively reducing $N_{\text{multipliers}}$ without suffering any performance loss;

- **quantization**: often 32-bit floating point calculations are not needed in the inference of a network to achieve optimal performance; quantization can reduce the precision of the calculations (weights, biases, etc.) in the neural network with no loss in performances;

- **parallelization**: one can tune how much to parallelize the multiplications required for a given layer computation; 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 optimize algorithm throughput versus resource usage.

In the following subsections, we describe in more detail the implementation and effect of these optimizations.

One important topic that we do not discuss is how the input features are computed before being fed to the neural network as this depends on the specific application. For example, in the jet substructure case study we consider, the pre-computation of the expert features can be quite time consuming and resource intensive. However, in other cases, the neural network may take raw detector inputs which require little preprocessing. For a more realistic scenario, it is important to consider the computation time of the inputs to the neural network. Additionally, it is important to consider the precision and range of the inputs; bit-shifting or translating the inputs to the appropriate range may be important to the performance of the algorithm.

Compression. Network compression is a widespread technique to reduce the size, energy consumption, and overtraining of deep neural networks [70]. Several approaches have been successfully deployed to compress networks, including [71]:

- **parameter pruning**: selective removal of weights based on a particular ranking [70, 72, 73],

- **low-rank factoriation**: using matrix/tensor decomposition to estimate informative parameters [74–78],

- **transferred/compact convolutional filters**: special structural convolutional filters to save parameters [79], and

- **knowledge distillation**: training a compact network with distilled knowledge of a large network [80].

Our approach is a simplified version of iterative parameter pruning and retraining [70, 81] with L_1 regularization, where the loss function L is augmented with an additional penalty term,

$$L_\lambda(\boldsymbol{w}) = L(\boldsymbol{w}) + \lambda \|\boldsymbol{w}\|_1 . \tag{2.3}$$

L_1 regularization is known to produce sparse models, provide built-in feature selection [82], and is a readily available option in many machine learning workflows. In principle, training with L_p regularization with $0 \leq p < 1$ [73] may improve the sparsity and performance of the model, but these regularizers are not always easy to implement. While we take this simplified approach, we note that there are other, more sophisticated, approaches to compression in the literature which may yield even better results.

We train the model with L_1 regularization with $\lambda = 10^{-4}$. We then sort the weights based on their absolute value relative to the maximum absolute value of the weights in a particular layer. With L_1 regularization we see two separate sub-populations of weights with one at smaller values and one at larger values. Weights falling below a certain percentile, corresponding to the smaller-value sub-population, are removed. Next, we retrain the model again with L_1 regularization while constraining the previously pruned weights to remain zero. We stop after seven iterations of this procedure at which point the sum of the pruned weight sub-population is 3% of the original summed weight population and the model is compressed by 70% (3051 weights pruned out of 4389 original weights and biases). Figure 6 illustrates this procedure. The top left of figure 6 shows the distribution of the weights before compression. From the top left to the bottom right, the arrows indicate the following steps of the pruning and retraining procedure and the resulting distribution of weights is shown. Finally, in the bottom right, we present the final distribution of the weights after compression. We observe no significant change in the pruned network performance when compared with the original.

Quantization. Quantized [70, 83–86] and even binarized [87–90] neural networks have been studied in detail as an additional way to compress neural networks by reducing the number of bits required to represent each weight. FPGAs provide considerable freedom in the choice of data type and precision. Both are important to consider to prevent wasting FPGA resources and incurring
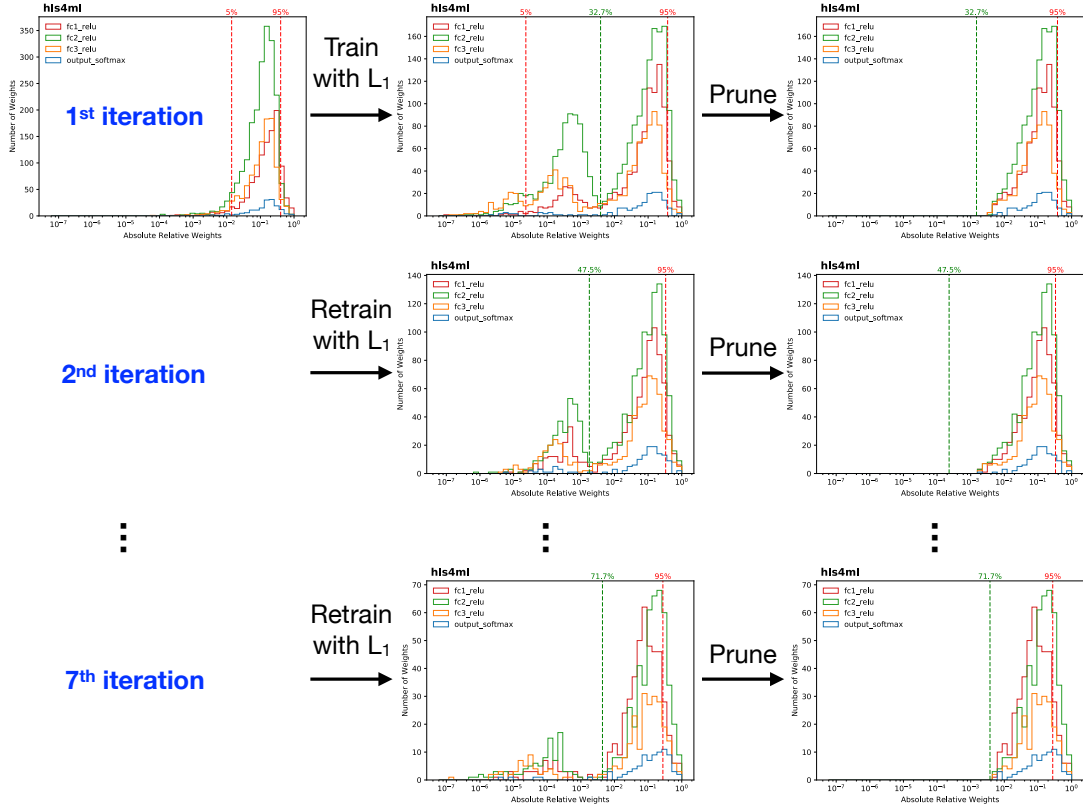
Figure 6. Illustration of the iterative parameter pruning and retraining with L_1 regularization procedure. The distribution of the absolute value of the weights relative to the maximum absolute value of the weights is shown after each step of the pruning and retraining procedure. In the top left, the distribution before compression is shown, while in the bottom right, the distribution after compression is displayed.

additional latency. In `hls4ml` we use fixed point arithmetic, which uses less resources and latency than floating point arithmetic.

The inputs, weights, biases, sums, and outputs of each layer (see eq. 2.1) are all represented as fixed point numbers. For each, the number of bits above and below the binary point can be configured for the use case. It is broadly observed that precision can be reduced significantly without causing a loss in performance [86], but this must be done with care. In figure 7, we show the distribution of the absolute value of the weights after the compression described in section 2.3. In this case, to avoid underflow/overflow in the weights, at least three bits should be assigned above the binary point — two to envelope the largest absolute value and one for the sign. The neuron values, x_m, and intermediate signals in the FPGA used to compute them may require more bits to avoid underflows/overflows. We determine the number of bits to assign *below* the binary point by scanning physics performance as a function of the bit precision.

Reducing precision saves resources used for signal routing as well as resources and latency used for mathematical operations. For many applications, the limiting FPGA resource will be the number of DSPs, which are used primarily for multiplications. The number of DSPs used per multiplier depends on the precision of the numbers being multiplied and can change abruptly. For
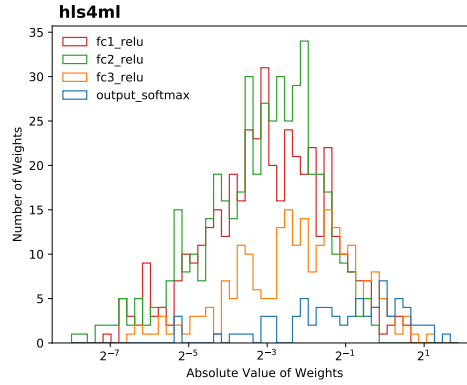
Figure 7. Distribution of the absolute value of the weights after compression.

example, one Xilinx DSP48E1 block [91] can multiply a 25-bit number with an 18-bit number, but two are required to multiply a 25-bit number with a 19-bit number. Similarly, the latency of multipliers increases with precision, though they can remain pipelined. Detailed exploration of the effect of calculation precision is presented in section 3.

As mentioned in section 2.1, non-trivial activation functions are precomputed for a range of input values and stored in BRAMs. The binning within this range and the output bit width are configurable in `hls4ml`. Lastly, we note that additional methods exist to further compress the network architecture through quantization that have not been explored in this paper [83, 89]. In particular, retraining the network with a quantized precision in the training can lead to equivalent performance with significantly smaller weight precision [92]. We leave investigations of these approaches for further work.

Parallelization. The trade-off between latency, throughput, and FPGA resource usage is determined by the parallelization of the inference calculation. In `hls4ml`, this is configured with a multiplier "reuse factor" that sets the number of times a multiplier is used in the computation of a layer's neuron values. With a reuse factor of one, the computation is fully parallel. With a reuse factor of R, $1/R$ of the computation is done at a time with a factor of $1/R$ fewer multipliers. This is illustrated in figure 8.

FPGA multpliers are pipelined; therefore, the latency of one layer computation, L_m, is approximately

$$L_m = L_{\text{mult}} + (R - 1) \times \text{II}_{\text{mult}} + L_{\text{activ}}, \qquad (2.4)$$

where L_{mult} is the latency of the multiplier, II_{mult} is the initiation interval of the multiplier, and L_{activ} is the latency of the activation function computation. Equation 2.4 is approximate because, in some cases, additional latency can be incurred for signal routing, for instance in the addition of multiplication results contributing to a neuron value.

As discussed in section 2.1, we implement each layer calculation independently and sequentially. The calculation of one layer cannot be initiated until the calculation of the previous layer has completed. Therefore, the total latency is equal to the sum of latencies of each layer plus
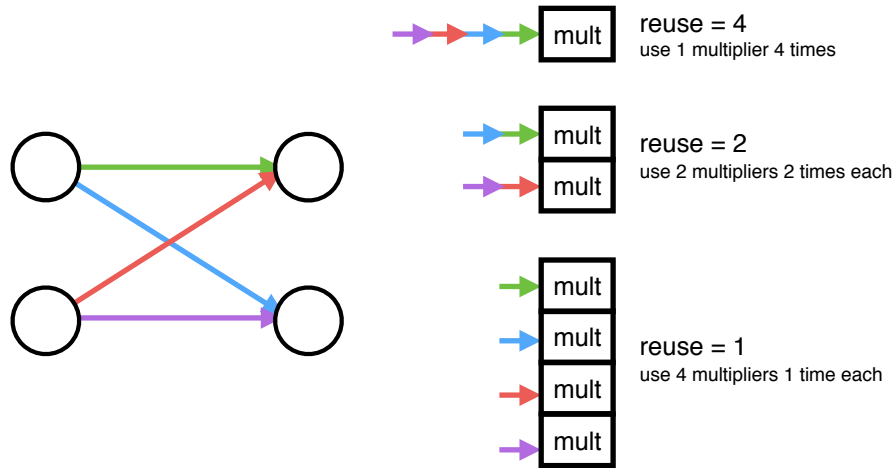
Figure 8. Illustration of multiplier resource usage for different values of reuse factor. The left drawing shows the case of two neuron pairs being connected by 4 connections, implying four multiplications to be performed. The right drawing shows how to perform these multiplications, from fully serial (top) to fully parallelized (bottom).

the latency required to connect the layers. The number of inferences completed per unit time is inversely proportional to the reuse factor.

3 Performance and implementation

In this section, we quantify the results from the HLS translation and optimization of the jet substructure neural network described in section 2.2 as a function of the three basic principles described in the previous section: compression, quantization, and parallelization. First we discuss the classification performance of the neural network when implemented in firmware in section 3.1. Then in section 3.2, we quantify the HLS synthesis in terms of FPGA resource usage and latency. The combination of these two metrics, classification and firmware performance, define how to optimally implement neural networks into FPGA hardware for a given application. Finally, in section 3.3, we discuss the implementation for a specific FPGA and compare the actual resource usage to the estimates from Vivado HLS, which can be obtained much more quickly.

3.1 Classification performance

To quantify the performance of our five-output classifier, we use the AUC metric, or area under the Receiver Operating Characteristic (ROC) curve. The ROC curve is given by the background rejection versus signal efficiency computed from sequential cuts on the classifier output, where background rejection is (1 − background efficiency), as illustrated in figure 5. We denote the AUC achieved by a full 32-bit floating point inference of the neural network as Expected AUC. We evaluate the neural network with fixed point precision denoted by <X,Y> where Y is the number of bits representing the signed number above the binary point (i.e. the integer part), and X is the total number of bits. We perform two scans — one where we fix the number of integer bits and one
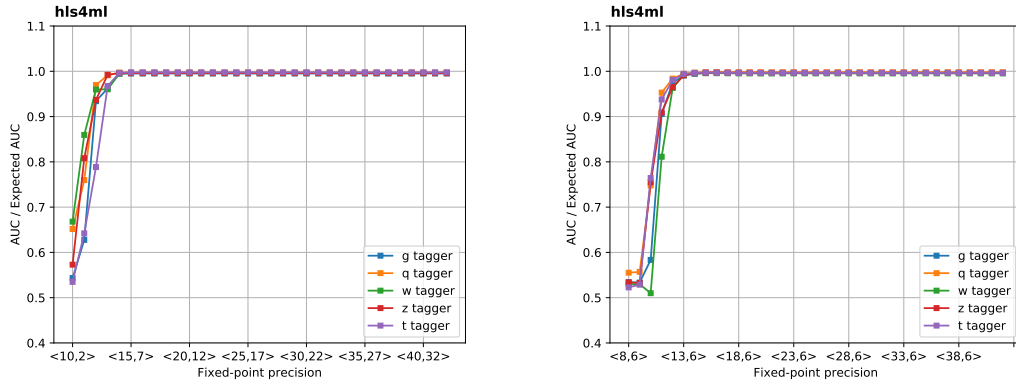
Figure 9. Ratios of the fixed point AUC and Expected AUC versus fixed point precision for the fully connected three-hidden-layer network. Optimal performance with no loss of classification power corresponds to ratios of 1. (left) The number of integer bits is scanned. (right) The number of integer bits is fixed to six, and the number of fractional bits is scanned. The various colored lines are AUC performance for different jet substructure taggers (q,g,W,Z,t).

where we fix the number fractional bits. The results are illustrated in figure 9 where the scan of the integer bits is on the left and the scan of the fractional bits is on the right.

Optimal performance with no loss of classification power corresponds to AUC/Expected AUC = 1. Figure 9 shows that with fixed point calculations and a sufficient number of bits, the Expected AUCs can be reproduced with negligible loss in performance. The number of integer bits is chosen to be just above the point where underflows/overflows do not occur and AUC/Expected AUC = 1. With this number of integer bits, we then scan in the number of fractional bits. Optimal performance is achieved with about 16 bits in total.

We perform similar scans to compare the compressed three-hidden-layer model AUC with that of the uncompressed model. Agreement with the Expected AUC occurs at roughly the same precision, as shown in figure 10.

3.2 Latency and resource estimates in HLS

We now explore how the FPGA resources required by the model are influenced by

- **compression**, the three-hidden-layer model with 70% of the parameters pruned;

- **quantization**, the precision of the inputs, weights, and biases; for this particular network we focus on scans of fixed point precision `<X,6>` based on our discussion in section 3.1. We scan above the point where we reach optimal performance to show the benefits of quantization and the resource usage one would expect when higher precision is required.

- **parallelization**, the number of times a given multiplier is used for a layer computation; using a multiplier once is the most parallel (and quickly) a layer can be computed and is what we call a *reuse factor* of 1.

With these variables as handles on how to control the implementation of the network, we monitor the following firmware implementation metrics:
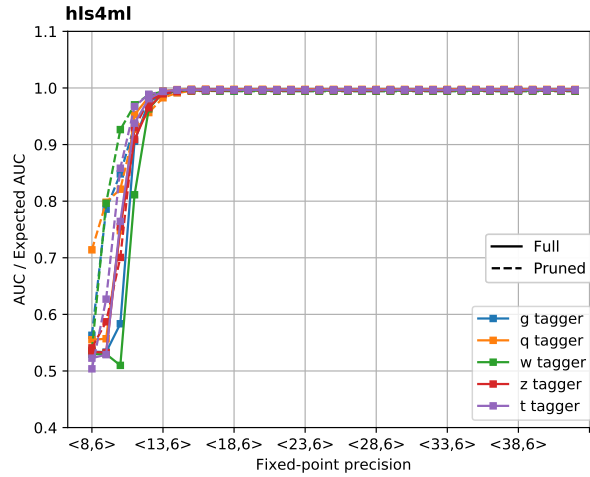
Figure 10. Ratios of the fixed point AUC and expected AUC versus fixed point precision. The solid lines represent the fully connected network, while the dash lines denote the compressed network.

- **resources**: DSPs, FFs, and LUTs;

- **latency**: the time it takes to compute the full network;

- **initiation interval**: the time before a new set of inputs can be accepted.

At the moment we do not probe the block RAM (BRAM) usage, which is only used to store pre-computed activation function values. Storing and accessing neural network weights from BRAMs, for instance, leads to latencies longer than the requirements for the first stages of LHC L1 triggers. For longer latency tasks, e.g. HLT applications, the capabilities of `hls4ml` can be expanded to allow for weight storage in BRAMs.

The results presented below are synthesized for a Xilinx Kintex Ultrascale FPGA with part number xcku115-flvb2104-2-i. The clock frequency is fixed at 200 MHz, which is typical for the first stages of LHC triggers. There can be variations in results versus clock frequency, but in the $O(100\,\text{MHz})$ range, we find variations are negligible. Resource usage estimates are taken from the Vivado HLS synthesis step and are found to be conservative in general when compared to implementation as we will discuss in section 3.3. While conservative, the short time required to make HLS resource estimates makes them useful for rapidly prototyping different network designs and deriving useful trends. Discussion of our results' dependence on the version of Vivado HLS, the specific FPGA, and the final implementation in the FPGA are discussed in section 3.3.

Resources with compression. We first explore the effect of compression on the FPGA resources required by the neural network. Because the compression is typically part of the training workflow, we consider it separately from the other optimization handles. Looking at the DSP usage and the algorithm latency in figure 11, we show the difference between our compressed and uncompressed neural network models. In both cases, we consider the network maximally parallelized (*reuse factor* of 1). With the weights stored in programmable logic, sparse matrix multiplication is handled trivially and zero-weight multiplications are optimized out of the network FGPA implementation.
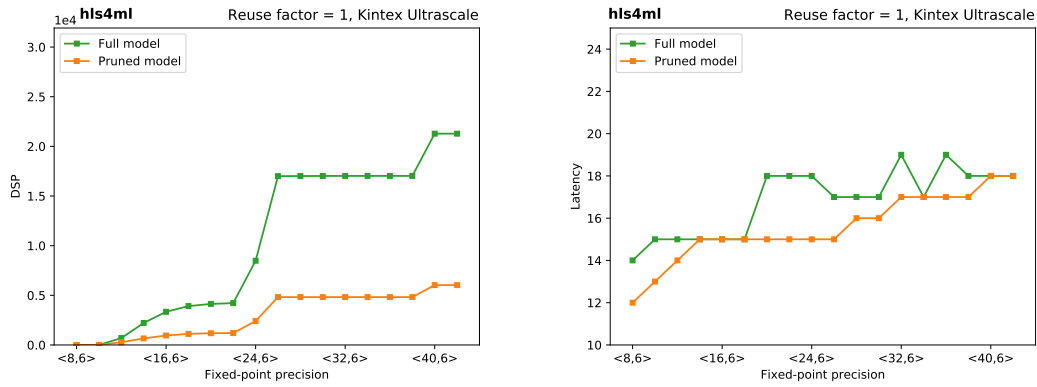
Figure 11. A comparison between the compressed and uncompressed models, with a reuse factor of 1 for DSP usage (left) and latency in clock cycles for a 200 MHz clock frequency (right). The x-axis is a scan in the fixed-point precision of the model and demonstrates how resource usage changes as a function of the precision of the calculations in the network inference.

We find this to be a very attractive feature of HLS though more sophisticated compression techniques like those described in [93] may require more study.

As shown in figure 11 (left), the DSP usage is drastically reduced for the compressed model compared to the original network by an amount that is proportional to the 70% compression rate described in section 2.3. In addition, the DSP usage increases as the fixed-point precision increases. The increases are not smoothly varying because they depend on the DSP design precisions. On the right of figure 11, we present the latency of the algorithm in clock cycles for a 200 MHz clock frequency. Because the network still has the same structure, in terms of the number of hidden layers, the latency is approximately the same in the compressed and uncompressed models. Note that the total latency to infer the model is approximately 15 clock cycles which translates to 75 ns, well within the latency budgets of the first stages of LHC triggers.

To summarize the results of the HLS synthesis of the compressed and uncompressed models, we report some vital statistics in table 2. We note the reduced resources while maintaining the same performance, latency, and initiation interval.

Table 2. A summary of the vital statistics and HLS resource estimates of the uncompressed and compressed jet substructure tagging model with a network precision of fixed-point `<16,6>` and fully pipelined with clock frequency of 200 MHz synthesized on a Xilinx Kintex Ultrascale FPGA.

Network	Uncompressed network	Compressed network
AUC / Expected AUC	99.68%	99.55%
Parameters	4389	1338
Compression factor	-	3.3×
DSP48E	3329	954
Logic (LUT + FF)	263,234	88,797
Latency	75 ns	75 ns

Compressed three-hidden-layer Model Results. We now consider our compressed three-hidden-layer neural network model as the benchmark model for our use case and perform detailed scans of FPGA resources versus network precision and reuse factor. In figure 12 and figure 13, we examine the DSP, FF, and LUT usage as a function of precision of the fixed point calculations, <X,6>. From the findings in section 3.1, we scan the number of fractional bits by scanning X while fixing the integer bits at $Y = 6$, guaranteeing no underflows/overflows. Different curves are shown for different values of reuse factor.
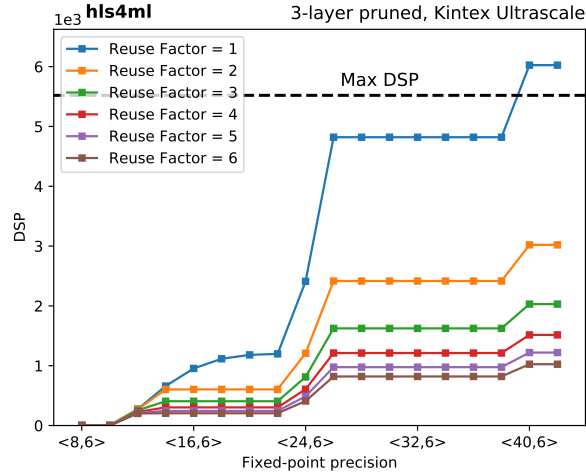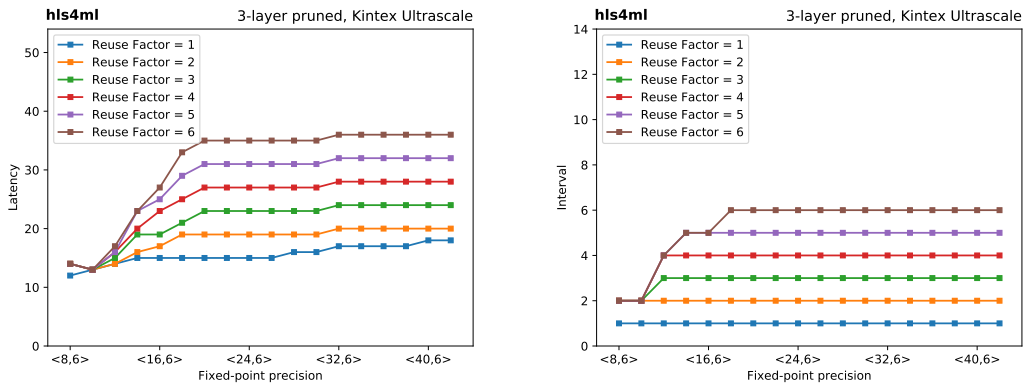

Figure 12. DSP usage in the compressed three-hidden-layer model as a function of the network precision. The various curves illustrate resource usage for different resource usage factors.
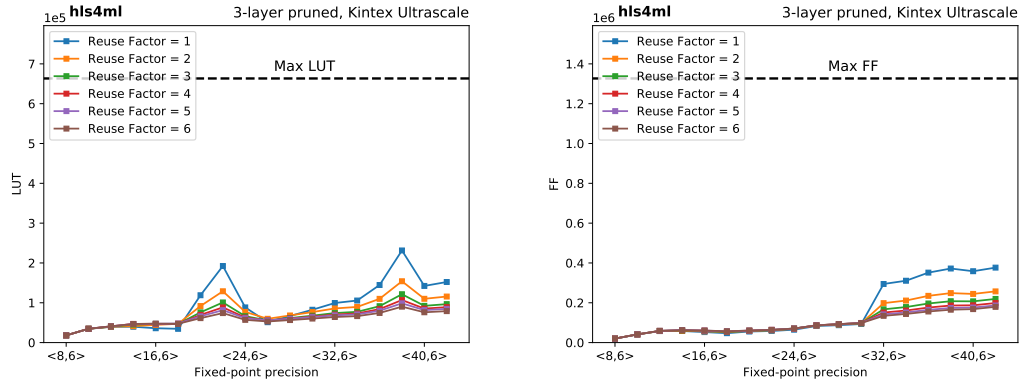
Figure 13. LUT and FF usage in the compressed three-hidden-layer model as a function of the network precision. The various curves illustrate resource usage for different resource usage factors.

In figure 12, we show how the reuse factor is used to control the number of times a multiplier is used in the neural network. As the reuse factor increases, we are able to control the DSP usage proportionally to the reuse factor. The DSP resource usage has steps in the resource usage as a function of the network precision. This is consistent for all values of `reuse` and comes from the

precision of the Xilinx FPGA DSPs. In the figure, we also indicate the maximum number of DSPs available in this particular Xilinx Kintex Ultrascale FPGA. In figure 13, the LUT (left) and FF (right) usage is shown. For both the LUTs and the FFs, the resource usage relative to the FGPA's capacity is small compared to that of the DSPs. Additionally, we observe spikes in FF usage at the DSP precision limits. We find that they are removed when performing the implementation (discussed in section 3.3). We note a general trend across figure 12 and figure 13 that the `reuse = 1` case can tend to deviate from the trends for the other cases when `reuse > 1`. We believe in the `reuse = 1` case, HLS is able to do further optimizations on a single multiplier for a given network design, and it does not have that optimization freedom when a multiplier is tasked with multiple operations when `reuse > 1`.

Figure 14. Latency (left) and initiation interval (right) in the compressed three-hidden-layer model as a function of the network precision. The various curves illustrate resource usage for different resource usage factors. The latency is given in clock cycles for a 200 MHz clock frequency.

Next, we examine the aspects of the FPGA implemenetation related to latency and throughput. In figure 14, on the left (right), we show the latency (initiation interval) of the algorithm versus precision for a number of different reuse factors. The latency of the network inference increases by 4 clocks, corresponding to the four layers of neuron values that must be computed, with each increment in reuse factor. This is in line with expectations from eq. 2.4 where additional reuse of multipliers in a given layer calculation incurs added latency. By design, the initiation interval and the reuse factor match as a new input can be introduced to the algorithm only when all multiplications for a given multiplier are completed. At very low network precision, the HLS synthesis initiation interval is smaller than the reuse factor as multiplications are no longer implemented in DSPs but through FFs and LUTs.

3.3 FPGA implementation

To get a qualitative understanding of the differences between Vivado HLS resource estimates and a final "placed and routed" implementation, we use a "bare" firmware design that uses minimal resources beyond those required by the neural network. This "bare" implementation consists of a simple VHDL wrapper that connects the `hls4ml` firmware block directly to the FPGA's general purpose input/output pins to prevent Vivado from optimizing out the logic we are trying to

characterize. Including the VHDL wrapper, we perform the firmware implementation and compare the resulting resource usage to the Vivado HLS estimates.

When performing the implementation, we note that the clock period targeted by HLS was not initially achieved due to the design not meeting timing constraints during the implementation step, so we increased the clock period in the final FPGA implementation to meet the timing constraints. The required increase in clock period became larger with NN complexity; algorithms that took a large part of the FPGA required longer clock periods. For the three-hidden-layer pruned neural network at 32-bit precision, a clock period of 8 ns was needed to implement an HLS block designed for 5 ns. This was observed for all reuse factors. A simple solution to overcome this issue is to synthesize the HLS design for a slightly smaller clock period than intended. We also note that different versions of Vivado HLS have varying degrees of success meeting timing constraints. We had more success meeting timing constraints at the HLS target with Vivado 2016.4 than 2017.2.

The power usage is shown in figure 15. For all implementations, a clear trend towards more power usage for larger network precision is present. As expected, as the throughput is decreased by increasing the reuse factor, power usage also goes down.
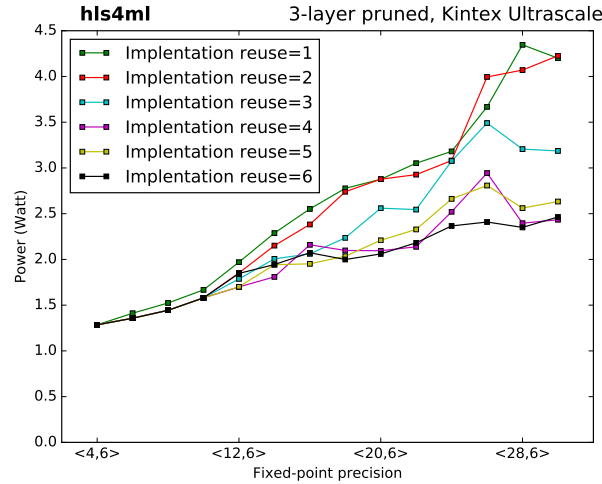

Figure 15. Power usage (W) for the baseline three-hidden-layer model as a function of precision.

We connect each bit of input/output directly to a unique pin under the assumption that all inputs are delivered on the same clock edge. In this "bare" implementation, we do not use the high speed transceivers that would allow significantly higher bandwidth input/ouput. Due to the limited number of pins, we now consider a different neural network model with fewer inputs. The model, illustrated in figure 4 (right), has ten input neurons and one output neuron with one hidden layer between. We also tested the three-hidden-layer pruned network and find similar quantitative conclusions in the regime where the number of pins was sufficient for implementation. For the rest of this subsection, we present results with the one-hidden-layer network using an 8 ns clock at implementation.

Figure 16 shows the DSP usage for the implemented design compared with the DSP estimate obtained from the HLS synthesis. For all cases, the DSP usage in the implemented design is

observed to be smaller than the HLS estimate, and in particular, we find the HLS synthesis estimate and the implementation agree well for multiplications which require one DSP (< 24 bits). Deviations between HLS estimates and Vivado implementation are evident above 24 bits, however, as other FF and LUT resources can further be used to optimize DSP usage in the final implementation.
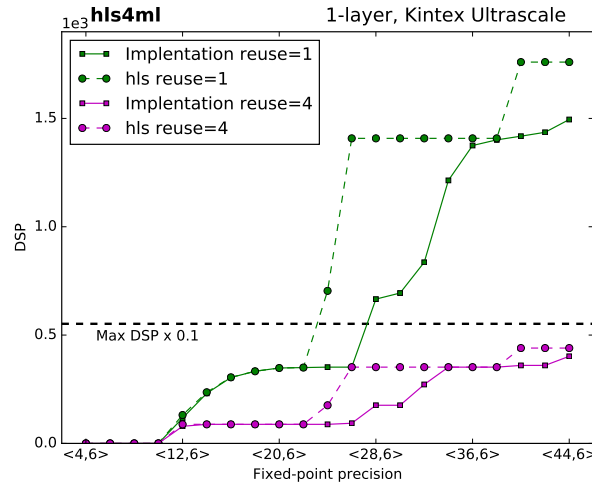
Figure 16. Comparsion of the DSP usage of the one-hidden-layer implemenation on the Xilinx Kintex Ultrascale FPGA as a function of the precision for various reuse factors.

Figure 17 compares the FF and LUT usage between the HLS estimate and the implementation. Large differences are present between the HLS estimates and implemented FF usage. The HLS estimates are typically factors of 2–4 high, becoming particularly large for implementations using more than 32 bits. The LUT usage is similarly overestimated by the HLS calculation, with the spikes at 22 bits and 38 bits optimized away during the implentation step. For all points, excluding the 26-bit implemenation of the LUTs, the HLS estimate is more conservative than the firmware implementation.
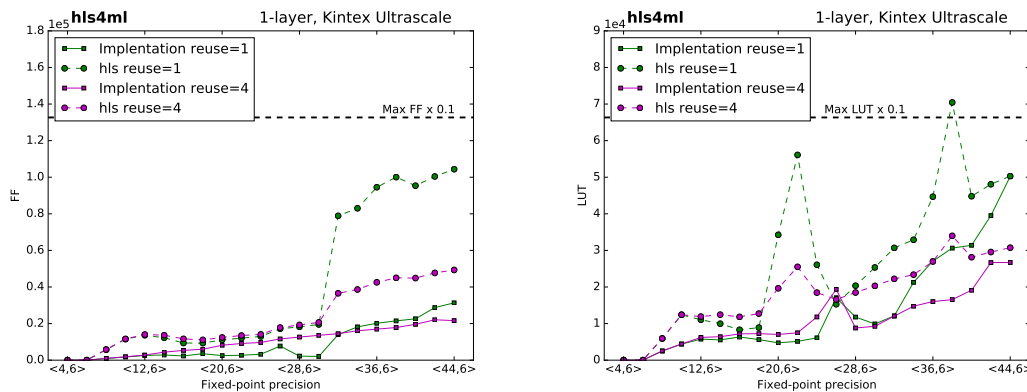
Figure 17. Comparison of the FF performance (left) and the LUT performance (right) for the Kintex Ultrascale processor as a function of the precision for 1 and 4 reuse factors.

4 Summary and outlook

We introduce `hls4ml`, a deep neural network compiler based on HLS capable of porting fully connected networks to an FPGA trained from conventional training frameworks such as `Keras` and `PyTorch`. For the first result using this framework, we focus on the application of real-time event reconstruction and filtering at the LHC in custom electronics with FPGAs. This requires pipelined network inference with latencies on the scale of 1 μs. For such low latencies, networks necessarily have a smaller number of parameters. For this paper, we consider a specific case study and train a fully connected neural network to identify jets as originating from a light quark, gluon, W boson, Z boson, or top quark. The original model has 4389 parameters, and applying network compression and reduced precision, it is possible to implement a fully-connected three-hidden-layer network in a Xilinx Kintex Ultrascale using roughly 10% of the available DSPs, with results varying with the initiation interval. The latency of the inference is approximately 75–150 ns with a clock frequency of 200 MHz. This fits well into the allowed hardware trigger reconstruction budget of LHC detectors such as ATLAS and CMS.

The accessibility and ease of configurability in HLS allows for physicists to quickly develop and optimize machine learning algorithms targeting FPGA hardware. This greatly decreases both firmware development time over traditional VHDL/Verilog-based algorithms and engineering resources in physics which are scarce. We discuss generic techniques such as network compression, parallelization, and reduced precision, which can be applied to design efficient neural network implementations tailored for different applications at the LHC and beyond. The results presented use `hls4ml` to scan the network precision and parallelization to optimize DSP and other resource usage. We compare results of estimated resources from HLS synthesis with the resource usage at implementation. HLS resource estimates are conservative, particularly for FFs and LUTs. The HLS resource estimate for DSP usage is comparable to the implemented design although it can be conservative across designed DSP precisions.

While we have demonstrated the `hls4ml` framework in the context of fully-connected neural networks, we intend for `hls4ml` to be a general tool for translating many types of neural network architectures that are commonly used in physics. For example, we envision expanding the framework to include convolutional neural networks (CNNs) and recurrent neural networks (RNNs), such as Long Short Term Memory (LSTM) units [37]. CNNs are typically used in image based problems including calorimeter cluster reconstruction [39, 94]. LSTMs are used to process a dynamic list of objects; for jet substructure tagging, they have been used to process lists of particle properties belonging to a single jet. At their core, both network architectures would build on the existing `hls4ml` framework, and the same efficient network design principles apply. In addition, `hls4ml` can be extended to target FPGAs from other vendors, such as Intel FPGAs using the Quartus HLS compiler. We currently support `Keras` and `PyTorch` based implementations.

There are further interesting extensions of this line of research using FPGA co-processors for machine learning, pioneered by efforts such as Micrsoft Brainwave [95, 96] and others. At experiments like CMS and ATLAS, the first tier of reconstruction is limited to the microsecond timescale. However, the second tier of reconstruction, the high-level trigger, is limited to reconstruct and understand events on the 100 ms timescale. This second tier currently reads the output of the first tier at a rate reduced by a factor of 400 from the original collision rate. At these timescales, the

inference times can be as long as 10 ms. For such long timescales, a large reuse factor can allow for large machine learning algorithms to placed on FPGAs. FPGAs can consequently be used as a co-processor accelerator to significantly reduce the time needed to perform complex, core LHC reconstruction algorithms such as track reconstruction. With the ability to infer $O(100)$ times faster than CPUs [97], FPGAs can be employed as a low-power, low-cost co-processor in conjunction with CPUs that can be used to significantly speed up the high-level trigger, and potentially improve its performance. In future works, we look forward to more detailed comparisons of neural networks for physics applications on CPU, GPU, and FPGA hardware.

Beyond the LHC, the scope is very broad. We believe this tool can be used in many different scientific applications. Increasingly in nuclear and particle physics, more intense beams and higher rate experiments are being developed, and readout and processing in these experiments often require high speed inference of complex data inputs.

Acknowledgments

References

[1] ATLAS collaboration, *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** (2012) 1 [arXiv:1207.7214].

[2] 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].

[3] CMS collaboration, *The CMS Experiment at the CERN LHC*, 2008 *JINST* **3** S08004.

[4] ATLAS collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, 2008 *JINST* **3** S08003.

[5] C. Bernius, *The ATLAS Trigger Algorithms Upgrade and Performance in Run-2*, 2017, arXiv:1709.09427, http://inspirehep.net/record/1626107/files/arXiv:1709.09427.pdf.

[6] M. Tosi, *The CMS trigger in Run 2*, PoS(EPS-HEP2017)523.

[7] CMS collaboration, *Measurements of Higgs boson properties in the diphoton decay channel in proton-proton collisions at $\sqrt{s}$ = 13 TeV*, arXiv:1804.02716.

[8] ATLAS collaboration, *Measurement of the Higgs boson mass in the $H \to ZZ^* \to 4\ell$ and $H \to \gamma\gamma$ channels with $\sqrt{s} = 13\,TeV\,pp$ collisions using the ATLAS detector*, `arXiv:1806.00242`.

[9] CMS collaboration, *Identification of heavy-flavour jets with the CMS detector in pp collisions at 13 TeV*, 2018 *JINST* **13** P05011 [`arXiv:1712.07158`].

[10] ATLAS collaboration, *Measurements of b-jet tagging efficiency with the ATLAS detector using $t\bar{t}$ events at $\sqrt{s} = 13\,TeV$*, `arXiv:1805.01845`.

[11] ATLAS collaboration, M. Lassnig, W. Toler, R. Vamosi and J. Bogado, *Machine learning of network metrics in ATLAS Distributed Data Management*, *J. Phys. Conf. Ser.* **898** (2017) 062009.

[12] CMS collaboration, *Observation of t̄tH production*, *Phys. Rev. Lett.* **120** (2018) 231801 [`arXiv:1804.02610`].

[13] ATLAS collaboration, *Observation of Higgs boson production in association with a top quark pair at the LHC with the ATLAS detector*, `arXiv:1806.00425`.

[14] Y. Gershtein, *CMS Hardware Track Trigger: New Opportunities for Long-Lived Particle Searches at the HL-LHC*, *Phys. Rev.* **D 96** (2017) 035027 [`arXiv:1705.04321`].

[15] CMS collaboration, *Search for narrow resonances in dijet final states at $\sqrt{s} = 8\,TeV$ with the novel CMS technique of data scouting*, *Phys. Rev. Lett.* **117** (2016) 031802 [`arXiv:1604.08907`].

[16] ATLAS collaboration, *Search for low-mass dijet resonances using trigger-level jets with the ATLAS detector in pp collisions at $\sqrt{s} = 13\,TeV$*, `arXiv:1804.03496`.

[17] CMS collaboration, *Search for dijet resonances in proton-proton collisions at $\sqrt{s} = 13\,TeV$ and constraints on dark matter and other models*, *Phys. Lett.* **B 769** (2017) 520 [*Erratum ibid.* **B 772** (2017) 882] [`arXiv:1611.03568`].

[18] A.J. Larkoski, I. Moult and B. Nachman, *Jet Substructure at the Large Hadron Collider: A Review of Recent Advances in Theory and Machine Learning*, `arXiv:1709.04464`.

[19] F. Chollet et al., *Keras*, https://keras.io, 2015.

[20] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito et al., *Automatic differentiation in PyTorch*.

[21] G. Inggs, S. Fleming, D. Thomas and W. Luk, *Is high level synthesis ready for business? a computational finance case study*, in proceedings of the *2014 International Conference on Field-Programmable Technology (FPT)*, December 2014, pp. 12–19.

[22] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons and F. Morgan, *Xilinx vivado high level synthesis: Case studies*, in proceddings of *Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET Year*, 2014, pp. 352–356.

[23] CMS collaboration, *Boosted Decision Trees in the Level-1 Muon Endcap Trigger at CMS*, Tech. Rep., CMS-CR-2017-357, CERN, Geneva, 2017.

[24] T. Boser, P. Calafiura and I. Johnson, *Convolutional neural networks for track reconstruction on fpgas*, https://docs.google.com/presentation/d/1mTqsm5TronnB8MFD6yyq3CSPCV4bfck_aQ-ericM9cQ/edit#slide=id.p3, 2017.

[25] E. Kreinar, *RFNoC neural network library using Vivado HLS*, *Proceedings of the GNU Radio Conference* **2** (2017) 7.

[26] S.I. Venieris, A. Kouris and C.-S. Bouganis, *Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions*, `arXiv:1803.05900`.

[27] V. Gokhale, A. Zaidy, A.X.M. Chang and E. Culurciello, *Snowflake: A model agnostic accelerator for deep convolutional neural networks*, `arXiv:1708.02579`.

[28] R. Collobert, K. Kavukcuoglu and C. Farabet, *Torch7: A matlab-like environment for machine learning*, talk given at *BigLearn, NIPS Workshop*, 2011.

[29] C. Zhang, Z. Fang, P. Zhou, P. Pan and J. Cong, *Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks*, in proceedings of the *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 2016, pp. 1–8.

[30] S.I. Venieris and C.-S. Bouganis, *fpgaConvNet: A Toolflow for Mapping Diverse Convolutional Neural Networks on Embedded FPGAs*, talk givn at *NIPS 2017 Workshop on Machine Learning on the Phone and other Consumer Devices*, 2017.

[31] S.I. Venieris and C.-S. Bouganis, *fpgaConvNet: Automated Mapping of Convolutional Neural Networks on FPGAs (Abstract Only)*, in proceedings of the *2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 291–292.

[32] S.I. Venieris and C.-S. Bouganis, *Latency-Driven Design for FPGA-based Convolutional Neural Networks*, in proceedings of the *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Septemper 2017, pp. 1–8.

[33] S.I. Venieris and C.-S. Bouganis, *fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs*, in proceedings of the *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, May 2016, pp. 40–47.

[34] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick et al., *Caffe: Convolutional Architecture for Fast Feature Embedding*, `arXiv:1408.5093`.

[35] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen et al., *Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates*, in proceedings of the *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2017, pp. 152–159.

[36] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro et al., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015.

[37] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, *Neural Comput.* **9** (1997) 1735.

[38] H. Sharma, J. Park, D. Mahajan, E. Amaro, J.K. Kim, C. Shao et al., *From high-level deep neural models to fpgas*, in proceedings of the *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[39] L. de Oliveira, M. Kagan, L. Mackey, B. Nachman and A. Schwartzman, *Jet-images — deep learning edition*, *JHEP* **07** (2016) 069 [`arXiv:1511.05190`].

[40] D. Guest, J. Collado, P. Baldi, S.-C. Hsu, G. Urban and D. Whiteson, *Jet Flavor Classification in High-Energy Physics with Deep Neural Networks*, *Phys. Rev.* **D 94** (2016) 112002 [`arXiv:1607.08633`].

[41] S. Macaluso and D. Shih, *Pulling Out All the Tops with Computer Vision and Deep Learning*, `arXiv:1803.00107`.

[42] K. Datta and A.J. Larkoski, *Novel Jet Observables from Machine Learning*, *JHEP* **03** (2018) 086 [`arXiv:1710.01305`].

[43] A. Butter, G. Kasieczka, T. Plehn and M. Russell, *Deep-learned Top Tagging with a Lorentz Layer*, `arXiv:1707.08966`.

[44] G. Kasieczka, T. Plehn, M. Russell and T. Schell, *Deep-learning Top Taggers or The End of QCD?*, *JHEP* **05** (2017) 006 [arXiv:1701.08784].

[45] P.T. Komiske, E.M. Metodiev and M.D. Schwartz, *Deep learning in color: towards automated quark/gluon jet discrimination*, *JHEP* **01** (2017) 110 [arXiv:1612.01551].

[46] A. Schwartzman, M. Kagan, L. Mackey, B. Nachman and L. De Oliveira, *Image Processing, Computer Vision and Deep Learning: new approaches to the analysis and physics interpretation of LHC events*, *J. Phys. Conf. Ser.* **762** (2016) 012035.

[47] J.M. Butterworth, A.R. Davison, M. Rubin and G.P. Salam, *Jet substructure as a new Higgs search channel at the LHC*, *Phys. Rev. Lett.* **100** (2008) 242001 [arXiv:0802.2470].

[48] L. Asquith et al., *Jet Substructure at the Large Hadron Collider: Experimental Review*, arXiv:1803.06991.

[49] J. Cogan, M. Kagan, E. Strauss and A. Schwarztman, *Jet-Images: Computer Vision Inspired Techniques for Jet Tagging*, *JHEP* **02** (2015) 118 [arXiv:1407.5675].

[50] J. Pearkes, W. Fedorko, A. Lister and C. Gay, *Jet Constituents for Deep Neural Network Based Top Quark Tagging*, arXiv:1704.02124.

[51] G. Louppe, K. Cho, C. Becot and K. Cranmer, *QCD-Aware Recursive Neural Networks for Jet Physics*, arXiv:1702.00748.

[52] CMS collaboration, *Search for low mass vector resonances decaying into quark-antiquark pairs in proton-proton collisions at $\sqrt{s} = 13$ TeV*, *JHEP* **01** (2018) 097 [arXiv:1710.00159].

[53] CMS collaboration, *Inclusive search for a highly boosted Higgs boson decaying to a bottom quark-antiquark pair*, *Phys. Rev. Lett.* **120** (2018) 071802 [arXiv:1709.05543].

[54] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer et al., *The automated computation of tree-level and next-to-leading order differential cross sections and their matching to parton shower simulations*, *JHEP* **07** (2014) 079 [arXiv:1405.0301].

[55] R.D. Ball et al., *Parton distributions with LHC data*, *Nucl. Phys.* **B 867** (2013) 244 [arXiv:1207.1303].

[56] T. Sjöstrand, S. Ask, J.R. Christiansen, R. Corke, N. Desai, P. Ilten et al., *An Introduction to PYTHIA 8.2*, *Comput. Phys. Commun.* **191** (2015) 159 [arXiv:1410.3012].

[57] P. Skands, S. Carrazza and J. Rojo, *Tuning PYTHIA 8.1: the Monash 2013 Tune*, *Eur. Phys. J.* **C 74** (2014) 3024 [arXiv:1404.5630].

[58] M. Cacciari, G.P. Salam and G. Soyez, *FastJet User Manual*, *Eur. Phys. J.* **C 72** (2012) 1896 [arXiv:1111.6097].

[59] M. Cacciari and G.P. Salam, *Dispelling the N^3 myth for the k_t jet-finder*, *Phys. Lett.* **B 641** (2006) 57 [hep-ph/0512210].

[60] M. Cacciari, G.P. Salam and G. Soyez, *The Anti-k(t) jet clustering algorithm*, *JHEP* **04** (2008) 063 [arXiv:0802.1189].

[61] D. Adams et al., *Towards an Understanding of the Correlations in Jet Substructure*, *Eur. Phys. J.* **C 75** (2015) 409 [arXiv:1504.00679].

[62] A.J. Larkoski, S. Marzani, G. Soyez and J. Thaler, *Soft Drop*, *JHEP* **05** (2014) 146 [arXiv:1402.2657].

[63] A.J. Larkoski, G.P. Salam and J. Thaler, *Energy Correlation Functions for Jet Substructure*, *JHEP* **06** (2013) 108 [arXiv:1305.0007].

[64] I. Moult, L. Necib and J. Thaler, *New Angles on Energy Correlation Functions*, *JHEP* **12** (2016) 153 [arXiv:1609.07483].

[65] E. Coleman, M. Freytsis, A. Hinzmann, M. Narain, J. Thaler, N. Tran et al., *The importance of calorimetry for highly-boosted jet substructure*, 2018 *JINST* **13** T01003 [arXiv:1709.08705].

[66] V. Nair and G.E. Hinton, *Rectified linear units improve restricted Boltzmann machines*, in Proceedings of *the 27th International Conference on Machine Learning*, vol. 27, 2010, pp. 807–814.

[67] D.P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980.

[68] AWS, *Amazon EC2 P2 Instances*, https://aws.amazon.com/ec2/instance-types/p2/, 2018.

[69] J.D. Hunter, *Matplotlib: A 2D graphics environment*, *Comput. Sci. Eng.* **9** (2007) 90.

[70] S. Han, H. Mao and W.J. Dally, *Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding*, arXiv:1510.00149.

[71] Y. Cheng, D. Wang, P. Zhou and T. Zhang, *A survey of model compression and acceleration for deep neural networks*, arXiv:1710.09282.

[72] Y. LeCun, J.S. Denker and S.A. Solla, *Optimal brain damage*, in *Advances in Neural Information Processing Systems 2*, D.S. Touretzky ed., Morgan-Kaufmann (1990), pp. 598–605.

[73] C. Louizos, M. Welling and D.P. Kingma, *Learning Sparse Neural Networks through L$_0$ Regularization*, arXiv:1712.01312.

[74] R. Rigamonti, A. Sironi, V. Lepetit and P. Fua, *Learning separable filters*, in proceedings of the *2013 IEEE Conference on Computer Vision and Pattern Recognition*, June 2013, pp. 2754–2761.

[75] E.L. Denton, W. Zaremba, J. Bruna, Y. LeCun and R. Fergus, *Exploiting linear structure within convolutional networks for efficient evaluation*, in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence and K.Q. Weinberger eds., Curran Associates, Inc. (2014), pp. 1269–1277.

[76] M. Jaderberg, A. Vedaldi and A. Zisserman, *Speeding up convolutional neural networks with low rank expansions*, arXiv:1405.3866.

[77] M. Denil, B. Shakibi, L. Dinh, M.A. Ranzato and N. de Freitas, *Predicting parameters in deep learning*, in *Advances in Neural Information Processing Systems 26*, C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani and K.Q. Weinberger eds., Curran Associates, Inc. (2013), pp. 2148–2156.

[78] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy and B. Ramabhadran, *Low-rank matrix factorization for deep neural network training with high-dimensional output targets*, in procedings of the *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 6655–6659.

[79] T.S. Cohen and M. Welling, *Group Equivariant Convolutional Networks*, arXiv:1602.07576.

[80] C. Bucilǎ, R. Caruana and A. Niculescu-Mizil, *Model compression*, in proceedings of the *12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, U.S.A. 2006, pp. 535–541.

[81] S. Han, J. Pool, J. Tran and W.J. Dally, *Learning both weights and connections for efficient neural networks*, arXiv:1506.02626.

[82] A. Y. Ng, *Feature selection, l1 vs. l2 regularization, and rotational invariance*, in proceedings of the *Twenty-first International Conference on Machine Learning*, New York, NY, U.S.A., 2004, p. 78.

[83] Y. Gong, L. Liu, M. Yang and L. D. Bourdev, *Compressing deep convolutional networks using vector quantization*, `arXiv:1412.6115`.

[84] J. Wu, C. Leng, Y. Wang, Q. Hu and J. Cheng, *Quantized convolutional neural networks for mobile devices*, in proceedings of the *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 4820–4828.

[85] V. Vanhoucke, A. Senior and M.Z. Mao, *Improving the speed of neural networks on cpus*, talk given at the *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[86] S. Gupta, A. Agrawal, K. Gopalakrishnan and P. Narayanan, *Deep learning with limited numerical precision*, `arXiv:1502.02551`.

[87] M. Courbariaux, Y. Bengio and J.-P. David, *Binaryconnect: Training deep neural networks with binary weights during propagations*, in *Advances in Neural Information Processing Systems 28*, C. Cortes, N.D. Lawrence, D.D. Lee, M. Sugiyama and R. Garnett eds., Curran Associates, Inc. (2015), pp. 3123–3131.

[88] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio, *Binarized neural networks*, in *Advances in Neural Information Processing Systems 29*, D.D. Lee, M. Sugiyama, U.V. Luxburg, I. Guyon and R. Garnett eds., Curran Associates, Inc. (2016), pp. 4107–4115.

[89] M. Rastegari, V. Ordonez, J. Redmon and A. Farhadi, *Xnor-net: Imagenet classification using binary convolutional neural networks*, `arXiv:1603.05279`.

[90] P. Merolla, R. Appuswamy, J.V. Arthur, S.K. Esser and D.S. Modha, *Deep neural networks are robust to weight binarization and other non-linear distortions*, `arXiv:1606.01981`.

[91] Xilinx, *7 Series DSP48E1 Slice User Guide*.

[92] C. Zhu, S. Han, H. Mao and W.J. Dally, *Trained ternary quantization*, `arXiv:1612.01064`.

[93] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M.A. Horowitz et al., *EIE: Efficient Inference Engine on Compressed Deep Neural Network*, `arXiv:1602.01528`.

[94] M. Paganini, L. de Oliveira and B. Nachman, *Accelerating Science with Generative Adversarial Networks: An Application to* 3D *Particle Showers in Multilayer Calorimeters*, *Phys. Rev. Lett.* **120** (2018) 042003 [`arXiv:1705.02355`].

[95] D. Burger, *Microsoft unveils Project Brainwave for real-time AI*, https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/, 2017.

[96] A.M. Caulfield, E.S. Chung, A. Putnam, H. Angepat, D. Firestone, J. Fowers et al., *Configurable clouds*, *IEEE Micro* **37** (2017) 52.

[97] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M.A. Horowitz et al., *EIE: efficient inference engine on compressed deep neural network*, `arXiv:1602.01528`.