Deep learning on FPGAs for L1 trigger and Data Acquisition

DESY Instrumentation Seminar, October 12, 2018

Javier Duarte, Sergo Jindariini, Ben Kreis, Ryan Rivera, Nhan Tran (Fermilab)

Jennifer Ngadiuba, Maurizio Pierini (CERN)

Edward Kreinar (Hawkeye 360)

Phil Harris, Song Han (MIT)

Zhenbin Wu (University of Illinois at Chicago)
Motivation
The challenge: triggering at (HL-)LHC
The challenge: triggering at (HL-)LHC

Extreme bunch crossing frequency of 40 MHz → extreme data rates $O(100 \text{ TB/s})$

“Triggering” = filter events to reduce data rates to manageable levels
The challenge: triggering at (HL-)LHC

Extreme bunch crossing frequency of 40 MHz → extreme data rates O(100 TB/s)

“Triggering” = filter events to reduce data rates to manageable levels

Squeeze the beams to increase data rates
→ multiple pp collisions per bunch crossing (pileup)

2016: <PU> ~ 20-50
2017 + Run 3: <PU> ~ 50-80
HL-LHC: 140-200

CHALLENGE: maintain physics in increasingly complex collision environment
→ untriggered events lost forever!

Sophisticated techniques needed to preserve the physics!
A typical trigger system

Triggering typically performed in multiple stages @ ATLAS and CMS

- **L1 Trigger**: 40 MHz, 1 kHz, 1 MB/evt
- **High-Level Trigger**: 100 kHz, 1 kHz, 1 MB/evt

- **Custom hardware**: Level-1 Trigger (hardware)
  - 99.75% rejected
  - Decision in ~4 μs
- **High-Level Trigger (software)**
  - 99% rejected
  - Decision in ~100s ms

After trigger, 99.99975% of events are gone forever

- **Offline**: Computing farm for detailed analysis of the full event
  - Latency O(100 ms)

Absorbs 100s TB/s

Trigger decision to be made in O(μs)

Latencies require all-FPGA design

For HL-LHC upgrade: latency and output rates will increase by ~ 3 (ex: for CMS 3.8 → 12.5 μs @ L1)
New trigger algorithms

The detector and its trigger system

Output: trigger primitives
(calo energy clusters, muons, tracks)

Particle-flow algorithm @ L1

Output: particle candidates

Trigger decision

CMS-TDR-017, JINST 12 (2017) P10003
New trigger algorithms

The detector and its trigger system

Output: trigger primitives
\(\text{(calo energy clusters, muons, tracks)}\)

Particle-flow algorithm @ L1

Output: particle candidates

Machine learning

Trigger decision

---

CMS-TDR-017, JINST 12 (2017) P10003
New trigger algorithms

The detector and its trigger system

Output: trigger primitives
(calo energy clusters, muons, tracks)

Particle-flow algorithm @ L1

Output: particle candidates

Machine learning

CMS-TDR-017, JINST 12 (2017) P10003

Trigger decision

THIS TALK!
The latency landscape @ LHC

100 ms

1 s

High-Level Trigger

computing farm

Offline

ML methods typically employed in offline analysis or longer latency trigger tasks

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ….
The latency landscape @ LHC

100 ms

1 s

CMS Simulation Preliminary
\( \sqrt{s} = 13 \text{ TeV}, 2016 \)

<table>
<thead>
<tr>
<th>Method</th>
<th>Misid. Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSVv2</td>
<td>( 10^{-1} )</td>
</tr>
<tr>
<td>DeepCSV</td>
<td>( 10^{-2} )</td>
</tr>
<tr>
<td>cMVAv2</td>
<td>( 10^{-3} )</td>
</tr>
<tr>
<td>uds, c</td>
<td></td>
</tr>
</tbody>
</table>

b-jet efficiency

Deep neural network based on high-level features
both offline and @ HLT

ML methods typically employed in offline analysis or longer latency trigger tasks

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ....

Level-1 Trigger (hardware)
- 99.75% rejected
- decision in ~4 \( \mu \text{s} \)

High-Level Trigger (software)
- 99% rejected
- decision in ~100s ms

After trigger, 99.99975% of events are gone forever

Off-line computing farm

ex, identification of b-quark jets

Deep neural network based on high-level features both offline and @ HLT
The latency landscape @ LHC

ML algorithms used offline for
* improving Higgs mass resolution with particle energy regression
* enhancing signal/background discrimination

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ....
The latency landscape @ LHC

1 ns 1 μs 100 ms 1 s

FPGAs
L1 Trigger
High-Level Trigger
computing farm
Offline

Exploration of ML algorithms in low-latency, real-time processing has just begun!

What can we do in < us on one FPGA?

ML methods typically employed in offline analysis or longer latency trigger tasks

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ....
Muon reconstruction @ L1

First implementation of a ML algo for CMS L1 trigger on FPGAs [*]

A BDT is used to improve the momentum of muons in the forward region of the detector

based on curvature angles in the magnetic fields \((\Delta \phi, \Delta \theta)\) and few other variables

Prediction of BDT for every possible input stored into pre-computed 1.2 GB Look-Up Table (LUT) on FPGA

Achieved reduction of background rates by factor 3 w/o efficiency losses

Usage of LUTs does not scale nicely with ML algo complexity → quickly use all resources

**Can we improve this approach?**

The rise of specialized hardware for ML

GPUs excell at parallel processing
- Good for complex NN training of huge amount of data!
- Notoriously power-hungry

Sub-optimal for fast and simple NN inference
- Optimize resources utilization for less intensive tasks
The rise of specialized hardware for ML

GPUs excel at parallel processing
Good for complex NN training of huge amount of data!
Notoriously power-hungry

Sub-optimal for fast and simple NN inference
Optimize resources utilization for less intensive tasks

New developments in FPGAs and ASICs making #RealTimeAI possible!
The rise of specialized hardware for ML

New developments in FPGAs and ASICs making #RealTimeAI possible!

Intel Arria 10 already at cloud scale for Microsoft Bing, Azure, etc..

Custom AI hardware for Google on the cloud

Google Tensor Processing Unit
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

Contain array of logic cells used to configure low level operations (bit masking, shifting, addition)
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

Contain array of logic cells used to configure low level operations (bit masking, shifting, addition)

FPGA diagram

Also contain embedded components:

- **Digital Signal Processors (DSPs):** logic units used for multiplications
- **Random-access memories (RAMs):** embedded memory elements
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

Contain array of logic cells embedded with DSPs, BRAMs, etc.

High speed input/output to handle the large bandwidth

Support highly parallel algorithm implementations

Low power (relative to CPU/GPU)

Digital Signal Processors (DSPs): logic units used for multiplications

Random-access memories (RAMs): embedded memory elements

Flip-flops (FF) and look up tables (LUTs) for additions
How are FPGAs programmed?

**Hardware Description Languages**

HDLs are programming languages which describe electronic circuits.

**High Level Synthesis**

generate HDL from more common C/C++ code

pre-processor directives and constraints used to optimize the timing

*drastic decrease in firmware development time!*

We use here **Xilinx Vivado HLS** [*], but plan also Intel/Altera Quartus HLS

Machine learning & FPGAs

FPGAs used broadly across particle physics experiments for DAQ and trigger development

- becoming more accessible thanks to use of HLS
- the hardware structures maps nicely onto ML architectures
- early adoption of ML algorithms for L1 trigger uses BDT

Extensive literature on deep learning in FPGAs

- mainly targeting acceleration of large networks, relaxed latency constraints
- support of ML architectures in Keras/TensorFlow, Caffe, Torch

This is the first dedicated study on inference of deep neural networks in FPGAs for low-latency application

arxiv.1804.06913
Neural network inference

\[ x_n = g_n\left( W_{n,n-1} x_{n-1} + b_n \right) \]

- **Activation function**: ReLU
- **Multiplication**
- **Addition**: DSPs, logic cells

- **Precomputed and stored in BRAMs**

**Diagram**:
- **Input Layer**: 16 inputs
- **Hidden Layers**: 64 nodes, 32 nodes, 32 nodes, 5 outputs
- **Output Layer**: 5 outputs, activation: SoftMax

**Equation**:
\[ N_{\text{multiplications}} = \sum_{n=2}^{N} L_{n-1} \times L_n \]
Neural network inference

\[ x_n = g_n(W_{n,n-1}x_{n-1} + b_n) \]

- **Activation function**: \( g_n \)
- **Multiplication**: \( W_{n,n-1} \) precomputed and stored in BRAMs
- **Addition**: DSPs, logic cells

**How many resources? DSPs, LUTs, FFs?**

- **16 inputs**
- **64 nodes**
- **5 outputs**

**Does the model fit in the latency requirements?**

**Layer m**

- **Input layer**: \( L_1 \)
- **Output layer**: \( L_n \)
- **M hidden layers**

**Layer formulation**

\[ N_{\text{multiplications}} = \sum_{n=2}^{N} L_{n-1} \times L_n \]
Case study: jet tagging

Study a multi-classification task: discrimination between highly energetic (boosted) \( q, g, W, Z, t \) initiated jets

Signal: reconstructed as one massive jet with substructure

Jet substructure observables used to distinguish signal vs background [*]

Jet substructure features

Jet substructure observables provide large discrimination power between these types of jets

mass, multiplicity, energy correlation functions, …
(computed with FastJet [*])


These are expert-level features

Not necessarily realistic for L1 trigger
“Raw” particle candidates more suitable (to be studied next)
But lessons here are generic

One more case: H→bb discrimination vs W/Z→qq requires more “raw” inputs for b-tagging information
Case study: jet tagging

- We train (on GPU) a **five output multi-classifier**: sample of events with **two boosted WW/ZZ/tt/qq/gg anti-$k_T$ jets**, generated with Madgraph and showered with Pythia8.

- Fully connected neural network with **16 expert inputs**:
  - **Relu activation function** for intermediate layers
  - **Softmax activation function** for output layer

---

AUC = area under ROC curve (100% is perfect, 20% is random)

---

### Diagram:

- 16 inputs
- 64 nodes, activation: ReLU
- 32 nodes, activation: ReLU
- 32 nodes, activation: ReLU
- 5 outputs, activation: SoftMax

---

12.10.2018

Jennifer Ngadiuba - **hls4ml**: deep neural networks in FPGAs
Efficient NN design for FPGAs

FPGAs provide huge flexibility

*Performance depends on how well you take advantage of this*

We have three handles:

- **compression**: reduce number of synapses or neurons
- **quantization**: reduces the precision of the calculations (inputs, weights, biases)
- **parallelization**: tune how much to parallelize to make the inference faster/slower versus FPGA resources

Constraints:

- Input bandwidth
- FPGA resources
- Latency
Efficient NN design for FPGAs

FPGAs provide huge flexibility

*Performance depends on how well you take advantage of this*

We have three handles:

- **compression**: reduce number of synapses or neurons

- **quantization**: reduces the precision of the calculations (inputs, weights, biases)

- **parallelization**: tune how much to parallelize to make the inference faster/slower versus FPGA resources

Constraints:

- Input bandwidth
- FPGA resources
- Latency
Efficient NN design: compression

- Iterative approach:
  - train with L1 regularization (loss function augmented with penalty term):
    \[ L_\lambda(\tilde{\mathbf{w}}) = L(\tilde{\mathbf{w}}) + \lambda||\tilde{\mathbf{w}}_1|| \]
  - sort the weights based on the value relative to the max value of the weights in that layer

![Graph showing iterative approach to compression](image)

1st iteration

Train with L1

...
Efficient NN design: compression

• Iterative approach:
  
  - train with **L1 regularization** (loss function augmented with penalty term):

    \[ L_\lambda(\vec{w}) = L(\vec{w}) + \lambda ||\vec{w}_1|| \]

  - sort the weights based on the value relative to the max value of the weights in that layer

  - prune weights falling below a certain percentile and retrain

---

**Diagrams:**

- **Train with L1 with Weights:**
  - Sort the weights based on the value relative to the max value of the weights in that layer.
  - Prune weights falling below a certain percentile and retrain.

- **Prune Weights:**
  - Shows the distribution of weights after pruning.

12.10.2018

Jennifer Ngadiuba - **hls4ml**: deep neural networks in FPGAs
Efficient NN design: compression

Prune and repeat the train for 7 iterations
Efficient NN design: compression

Prune and repeat the train for 7 iterations

→ 70% reduction of weights and multiplications w/o performance loss

7th iteration

Prune

Retrain with L1

before pruning

after pruning

pruning synapses

pruning neurons

Train with L1

Prune

Figure 3.1: Pruning the synapses and neurons of a deep neural network.

The phases of pruning and retraining may be repeated iteratively to further reduce network complexity. In effect, this training process learns the network connectivity in addition to the weights — this parallels the human brain development [109] [110], where excess synapses formed in the first few months of life are gradually "pruned", with neurons losing little-used connections while preserving the functionally important connections.

On the ImageNet dataset, the pruning method reduced the number of parameters of AlexNet by a factor of $9 \times$ (61 to 6.7 million), without incurring accuracy loss. Similar experiments with VGG-16 found that the total number of parameters can be reduced by $13 \times$ (138 to 10.3 million), again with no loss of accuracy. We also experimented with the more efficient fully-convolutional neural networks: GoogleNet (Inception-V1), SqueezeNet, and ResNet-50, which have zero or very thin fully connected layers. From these experiments we find that they share very similar pruning ratios before the accuracy drops: 70% of the parameters in those fully-convolutional neural networks can be pruned. GoogleNet is pruned from 7 million to 2 million parameters, SqueezeNet from 1.2 million to 0.38 million, and ResNet-50 from 25.5 million to 7.47 million, all with no loss of Top-1 and Top-5 accuracy on ImageNet.

In the following sections, we provide solutions on how to prune neural networks and how to retrain the pruned model to recover prediction accuracy. We also demonstrate the speedup and energy efficiency improvements of the pruned model when run on commodity hardware.
Efficient NN design for FPGAs

FPGAs provide huge flexibility

*Performance depends on how well you take advantage of this*

We have three handles:

- **compression**: reduce number of synapses or neurons

- **quantization**: reduces the precision of the calculations (inputs, weights, biases)

- **parallelization**: tune how much to parallelize to make the inference faster/slower versus FPGA resources

Constraints:

- Input bandwidth
- FPGA resources
- Latency
Efficient NN design: quantization

- In FPGAs use **fixed point data types** → less resources and latency than 32-bit floating point

- NN inputs, weights, biases, outputs represented as `ap_fixed<width,integer>`

```
0101.10111010101
```

- To avoid overflow/underflow of weights at least 3 bits needed

- But need more bits for neurons as computed with multiplications and sums → we perform a scan of physics performance versus bit precision

```
• 0101.10111010101
```

```
weights
```

- `ap_fixed<14,4>`

- Integer bits = 2 + 1 for sign (need more for neurons)
Efficient NN design for FPGAs

FPGAs provide huge flexibility

*Performance depends on how well you take advantage of this*

Constraints:
- Input bandwidth
- FPGA resources
- Latency

We have three handles:

- **compression**: reduce number of synapses or neurons

- **quantization**: reduces the precision of the calculations (inputs, weights, biases)

- **parallelization**: tune how much to parallelize to make the inference faster/slower versus FPGA resources
Efficient NN design: parallelization

- Trade-off between latency and FPGA resource usage determined by the parallelization of the calculations in each layer.
- Configure the “reuse factor” = number of times a multiplier is used to do a computation.

**Reuse factor:** how much to parallelize operations in a hidden layer.

- **Fully serial:**
  - reuse = 4
  - use 1 multiplier 4 times

- **Fully parallel:**
  - reuse = 1
  - use 4 multipliers 1 time each

- **Less resources/ Less throughput:**
  - reuse = 2
  - use 2 multipliers 2 times each

Diagram shows the trade-off between latency and resource usage with different reuse factors.
In this section we give an overview of the basic task of translating a given neural network model into a firmware implementation using HLS. We then pick a specific use-case to study, though the study will be discussed in a way that is meant to be applicable for a broad class of problems. We conclude this section by discussing how to create an efficient and optimal firmware implementation of a neural network in terms of not only performance but also resource usage and latency.

2.1 **hls4ml concept**

Our basic task is to translate a trained neural network by taking a model architecture, weights, and biases and implementing them in HLS in an automated fashion. This automated procedure is the task of the software/firmware package, **hls4ml**. A schematic of a typical workflow is illustrated in Fig. 1.

The part of the workflow that is 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 Sec. 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 produces a firmware block. This automated tool has a number of configurable parameters which can help the user customize the network translation for their application.

The time to perform the hls4ml translation is much shorter (minutes to hours) than a custom design of a neural network and can be used to rapidly prototype machine learning algorithms without dedicated engineering support. For physicists, this makes designing physics algorithms for the trigger or DAQ significantly more accessible and efficient, thus allowing the "time to physics" to be greatly reduced.

The `hls4ml` package

Translation

```python
python keras-to-hls.py -c keras-config.yml
```

**Translation**

**Inputs**

**Keras**

**HDF5**

**Config**

- IOType: parallelize or serialize
- ReuseFactor: how much to parallelize
- DefaultPrecision: inputs, weights, biases

```
KerasJson: example-keras-model-files/KERAS_1layer.json
KerasH5: example-keras-model-files/KERAS_1layer_weights.h5
OutputDir: my-hls-test
ProjectName: myproject
XilinxPart: xc7vx690tffg1927-2
ClockPeriod: 5

IOType: io_parallel # options: io_serial/io_parallel
ReuseFactor: 1
DefaultPrecision: ap_fixed<18,8>
```

```
my-hls-test/
build_prj.tcl
firmware
myproject_test.cpp
```
The **hls4ml** package

**Build HLS project**

```
# HLS4ML #
#
open_project -reset myproject_prj
set_top myproject
add_files firmware/myproject.cpp -cflags "-I[file normalize ../../nnet_utils]"
add_files -tb myproject_test.cpp -cflags "-I[file normalize ../../nnet_utils]"
add_files -tb firmware/weights
open_solution -reset "solution1"
set_part {xcku115-flvf1924-2-i}
csim_design -period 5 -name default
csim_design
 csynth_design
 cosim_design -trace_level all
export_design -format ip_catalog
exit
```

**vivado_hls -f build_prj.tcl**

**Vivado™ HLS**

Produce a firmware block in ~ minutes!
Study details

GOAL
Map out FPGA performance, resource usage and latency versus compression, quantization, and parallelization hyperparameters

SETUP
Xilinx Vivado 2017.2
HLS target clock frequency: 200 MHz (5 clocks/BX)
Kintex Ultrascale, xcku115-flvb2104-2-i
  • 1.4M logic cells, 5,520 DSPs, 1.3M FFs, 700k LUTs, 2200 BRAMs

RESULTS
First examine resource usage coming from HLS estimate
Then discuss the exact resources given by the final implementation
Efficient NN design: quantization

- Quantify the performance of the classifier with the AUC
- Expected AUC = AUC achieved by 32-bit floating point inference of the neural network

```
ap_fixed<width,integer>
```

0101.1011101010

Scan integer bits
- Fractional bits fixed to 8
- Full performance at 6 integer bits

Scan fractional bits
- Integer bits fixed to 6
- Full performance at 8 fractional bits
DSPs (used for multiplication) are often limiting resource. - maximum use when fully parallelized - DSPs have a max size for input (e.g. 27x18 bits), so number of DSPs per multiplication changes with precision

70% compression ~ 70% fewer DSPs

### Efficient NN design: compression

**Fully parallelized (max DSP use)**

- Number of DSPs available
- Compression

![Graph showing compression and DSP usage](image)

- **70% compression ~ 70% fewer DSPs**

**Algorithm 1**

1. Training connectivity
2. Pruning connections
3. Retraining the remaining weights

In the following sections, we provide solutions on how to prune neural networks and how to retrain the pruned model to recover prediction accuracy.

#### 3.2 Pruning Methodology

Our pruning method employs a three-step process: training connectivity, pruning connections, and retraining the remaining weights. The last two steps can be done iteratively to obtain better compression ratios. The process is illustrated in Figure 3.2.
Parallelization: DSPs usage

Figure 11: DSP usage in the pruned 3-layer model as a function of the network precision. The various curves illustrate resource usage for different resource usage factors.

Figure 12: FF and LUT usage in the pruned 3-layer model as a function of the network precision. The various curves illustrate resource usage for different resource usage factors.

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. In the right plot of Fig. 13, the initiation interval is shown for different reuse factors. 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 DSP multiplier are completed.

At very low network precision, the HLS synthesis initiation interval is smaller than the reuse factor. This is because multiplications are no longer implemented in DSPs but through FFs and LUTs.
Parallelization: Timing

- **Initiation interval**: number of clocks before accepting new inference inputs
  - scales with reuse factor
  - for very low data precisions multiplications implemented through FFs and LUTs

- **Additional latency** introduced by reusing the multiplier

\[
L_m = L_{\text{mult}} + (R - 1) \times I_{\text{mult}} + L_{\text{activ}}
\]
Other resources: FFs and LUTs

- Fairly linear increase with precision
- Small percentage of total available
- Spikes present at steep transitions in LUTs usage as artifacts of HLS synthesis
  
  *Not observed in implementation*
  
  *Found also dependence on Vivado HLS version*
Firmware implementation

- Final implementation gives actual resource usage and timing estimate
  - how optimal is the HLS design?
- Power usage increases with precision, it goes down for less throughput (higher reuse factor)
Firmware implementation

- Final implementation gives actual resource usage and timing estimate
  - How optimal is the HLS design?

- Power usage increases with precision, it goes down for less throughput (higher reuse factor)

- Implement a 1-layer NN, simply routing all firmware block’s inputs and outputs to FPGA available pins

- HLS estimate on resource usage are conservative
  - DSPs usage agree well below DSP precision transition (27 bit), implementation does further optimization
  - FFs and LUTs overestimated by a factor 2-4
Summary and outlook
The latency landscape @ LHC

Focused on L1 trigger as first application → pure FPGAs

What can we do in < us on one FPGA?
The latency landscape @ LHC

Focused on L1 trigger as first application → pure FPGAs

What can we do in < us on one FPGA?

More time means
More resource reuse (x1000)
Bigger networks
→ acceleration with FPGAs
FPGA Co-Processor Acceleration Card

Offload a CPU from the computational heavy parts to a FPGA “accelerator”

- Increased computational speed of 10x-100x
- Reduced system size of 10x
- Reduced power consumption of 10x-100x

General use cases

- Autonomous vehicles
- Edge computing (phones)
- Big data analytics
- Medical applications (health monitoring, medical imaging) ….

Use case @ LHC

- FPGA accelerators on-site for HLT
- FPGA accelerators for offline computing resources (Cloud: Microsoft, Amazon, etc.)
Amazon Web Service provides cloud based system consisting of CPU/GPU/FPGAs

- AWS F1 instances include up to 8 Xilinx Virtex Ultrascale+

Used hls4ml through SDAccel to create a firmware implementation of 1D CNN

- 5 layers with 10 four-channel inputs, latency of 116 ns
- successfully run on an AWS F1 instance
New possibilities for HEP

Convolutional neural networks

CNNs primarily developed for computer vision/image recognition

- Designed to recognize visual patterns from pixel images with minimal preprocessing
- Exploit strong spatially local correlation present in natural images
- Share weights within a layer thanks to locality and translation invariance

ex, GoogLeNet (2014)

22 layers
5 million parameters

arxiv.1409.4842
New possibilities for HEP

Adaptation of GoogLeNet used offline by NOvA experiment:

Readout detector as a (multidimensional) image
Identify neutrino interactions based on their topology from “raw” data

Can we fit this on a FPGA?

Benefits in large reduction of recorded data

Beyond LHC…

Noble gas TPCs for dark matter and neutrino physics increasing target size and facing increased data rates
**hls4ml**: other NN architectures

- **Convolutional Neural Networks**
  - active implementation of small Conv1D and Conv2D with hls4ml
  - resources reuse and compression supported
  - work is ongoing to ensure large scale networks

- **Boosted Decision Tree** *(work in progress)*
  - each node in decision tree compares element against a threshold $\rightarrow$ boolean logic, thresholds in LUT, suitable for FPGA
  - each tree is independent $\rightarrow$ high parallelization

- **Binary/Ternary Neural Networks** *(work in progress)*
  - weights are binary/ternary in the inference $= \pm 1,0$
  - ternary NN does not need pruning/compression
  - similar performance and latency with 0% DSPs used

- **Recurrent NN and LSTM** under testing *(work in progress)*
Summary

We introduced a new software/firmware package **hls4ml**

Automated translation of everyday machine learning inference into firmware in ~ minutes

Tunable configuration for optimization of your use case

First application is single FPGA, <1 us latency for L1 trigger or DAQ

Explore also applications for acceleration with CPU-FPGA co-processors for long latency trigger tasks

For more info


Backup
CPUs, GPUs, FPGAs, and ASICs

FPGAs are the middle ground of latency, energy efficiency and flexibility