Machine Learning for Trigger and Data Acquisition

Thomas James (CERN)

thanks to Sioni Summers (CERN) for last year's slides

ISOTDAQ JUNE 2023, ISTANBUL

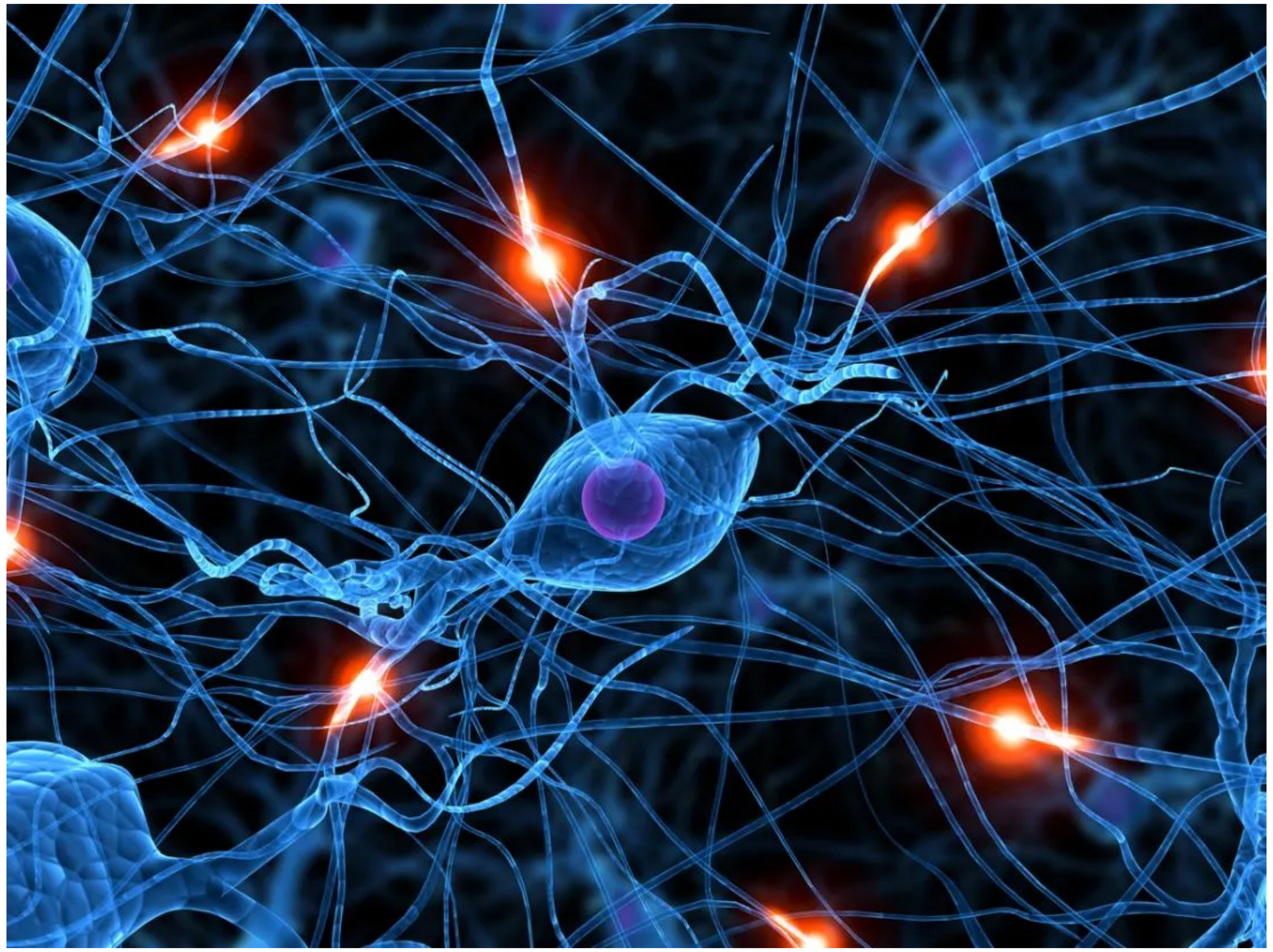
Contents

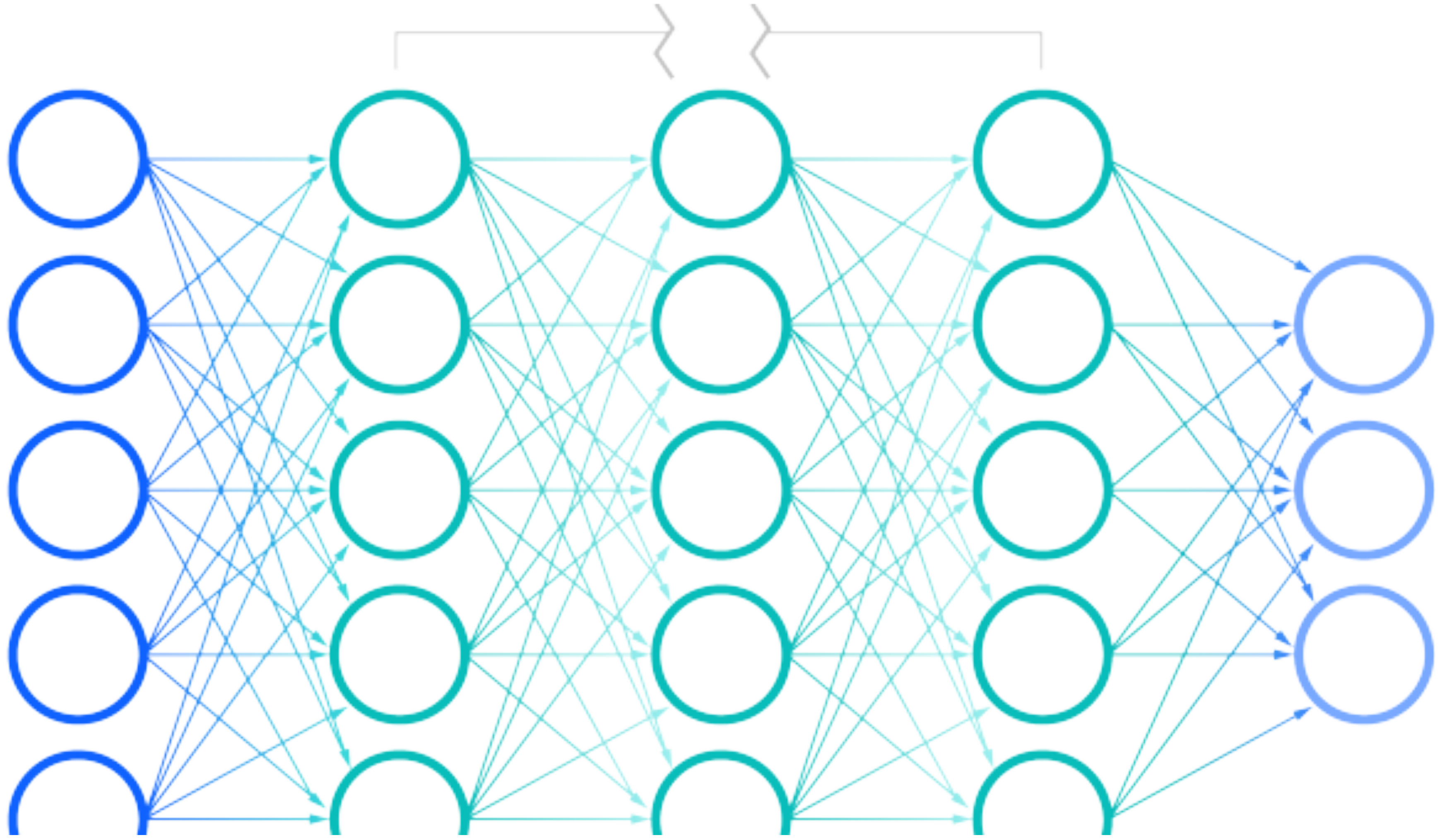
- Recap of ML
    - NN recap - CNN, RNN, GNN
    - Tools and Frameworks
- ML in HEP and TDAQ
    - Examples
- GPUs
- FPGAs
    - Tools, High level synthesis, Quantisation
    - Examples

Motivation

> “Scientific discoveries come from groundbreaking ideas and the capability to validate those ideas by testing nature at new scales—finer and more precise temporal and spatial resolution. This is leading to an explosion of data that must be interpreted, and ML is proving a powerful approach. The more efficiently we can test our hypotheses, the faster we can achieve discovery. To fully unleash the power of ML and accelerate discoveries, it is necessary to embed it into our scientific process, into our instruments and detectors.”

Applications and Techniques for Fast Machine Learning in Science

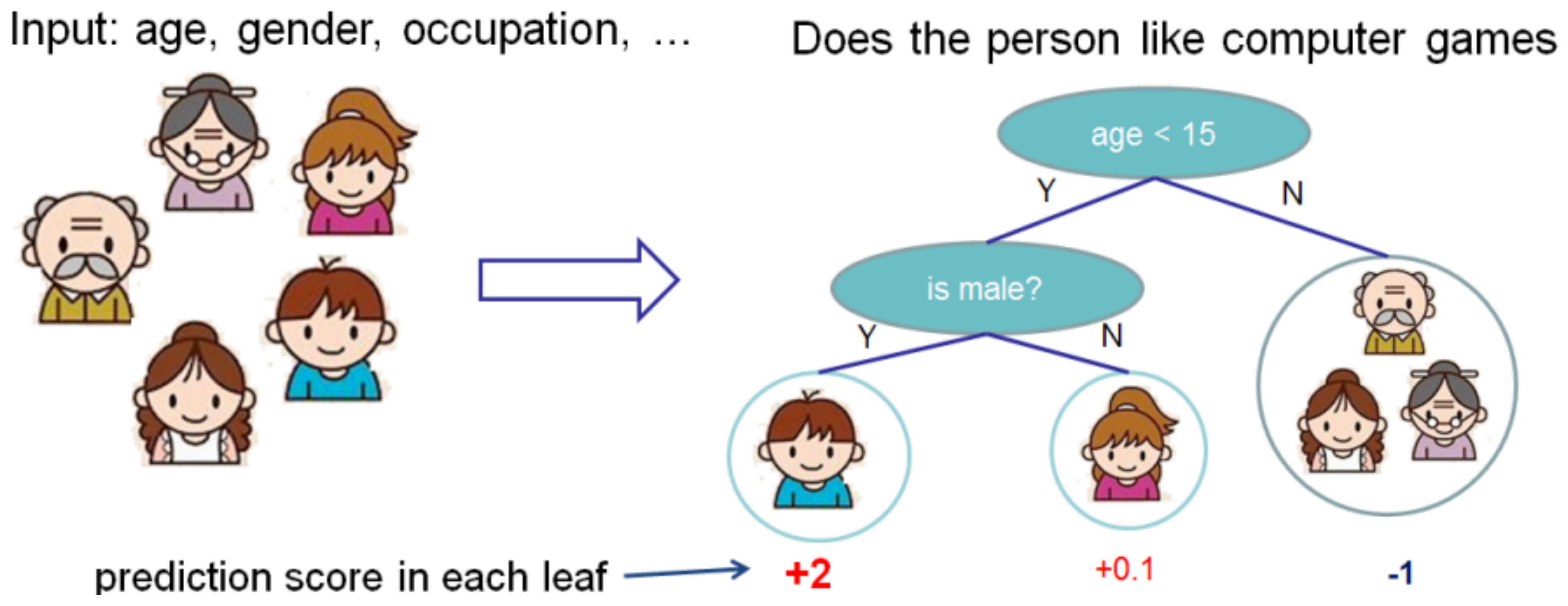




Introduction to ML & NN designs

Introduction to machine learning

- Build models which learn patterns from data to later make predictions on unseen data
  - Example: predict whether a person will like computer games from characteristics
- ML has been used to great effect in HEP, even since 1980s
  - Most commonly in offline analysis and reconstruction
  - But increasingly in realtime / trigger & DAQ
- ML, and Fast ML are extremely popular - lots of good tools out there

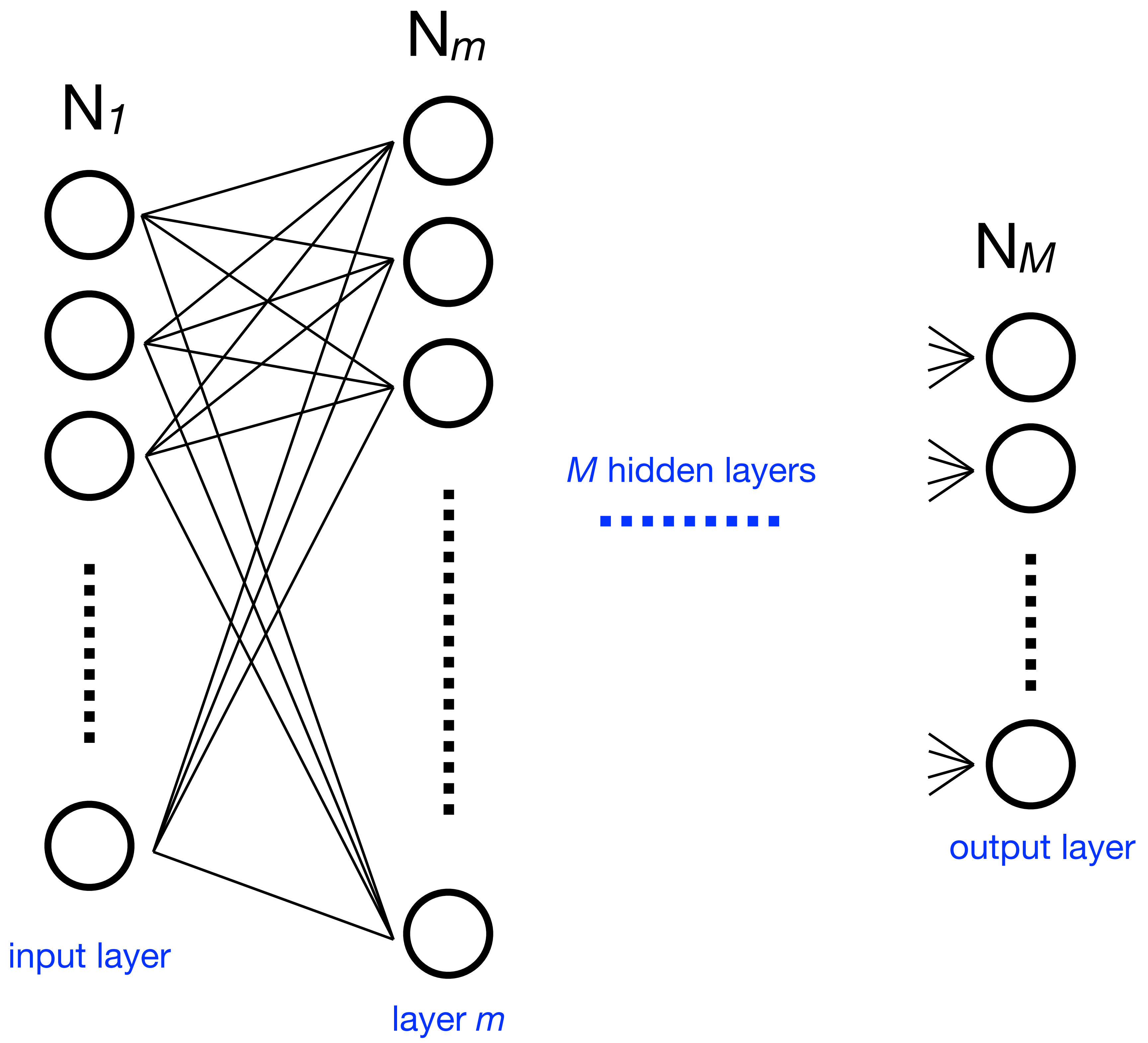

Input: age, gender, occupation, ...
Does the person like computer games
age < 15
Y
N
is male?
Y
N
prediction score in each leaf
+2
+0.1
-1

- Decision tree thresholds and prediction probabilities are learned from the training data

Neural Networks

- Loosely inspired by brain structure with neurons and synapses
  - Neurons are real valued representations of ‘something’
  - Synapses connect neurons (in one direction) with a *weight*
- Input neurons are your data variables
- Output neuron(s) are your predictions
  - class probabilities,
  - or continuous variables if performing a regression
- Hidden layers bring the performance of deep neural networks
  - Intermediate layers of neurons learn a more abstract representation of the data
  - More capable than ‘shallow’ networks on raw data
- Many topologies exist for different types of problems

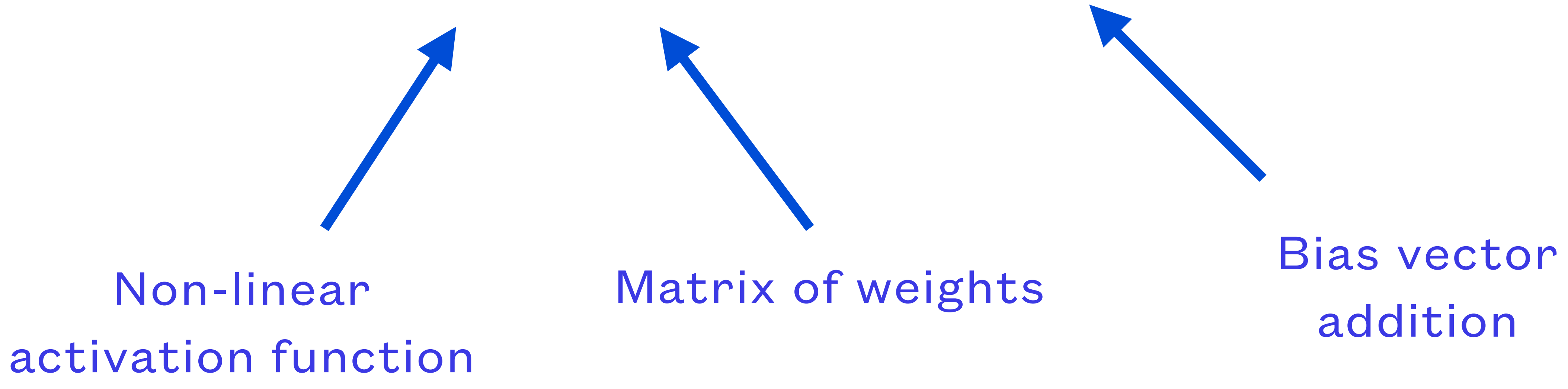
Fully Connected or *Dense* Neural Networks



Neural Networks

- The values of neurons in a layer is given by the product of the neuron values of the previous layer and the matrix of weights, with an added ‘bias’, and a non-linear ‘activation function’ applied

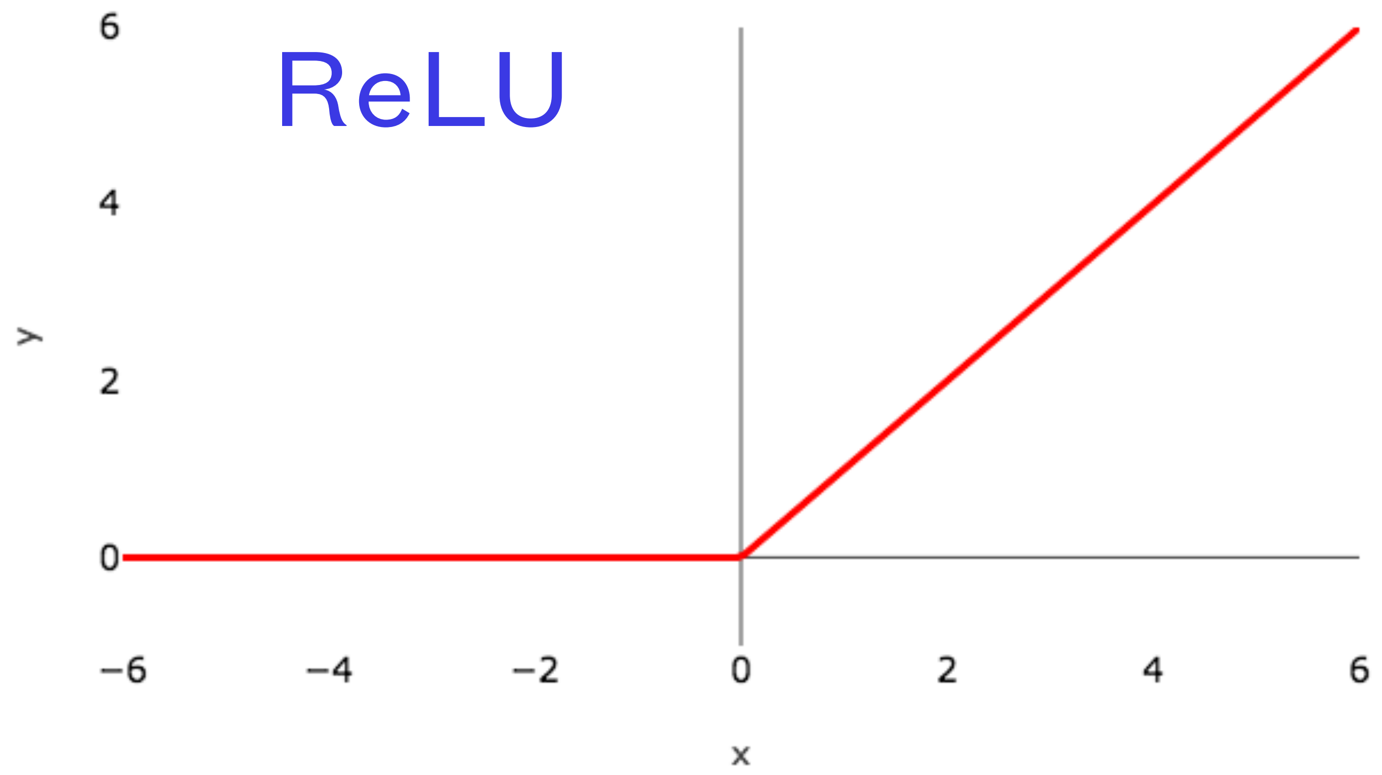
$$X_n = g_n(W_{n,n-1}x_{n-1} + b_n)$$



Non-linear activation function

Matrix of weights

Bias vector addition



- Without the activation function, we’re just doing linear transformations of our variables
- Values of weights and biases learned from data during training

Training with Gradient Descent

- *Supervised learning* - start with a NN of randomised weights and a collection of *training data*
- Evaluate performance network with a loss function, e.g. mean squared error:

$$\underset{\substack{\uparrow\\ \text{Loss}}}{L(y,\hat{y})} = \frac{1}{m}\sum_{i=1}^{m}(\underset{\substack{\uparrow\\ \text{Truth}}}{y_i} - \underset{\substack{\uparrow\\ \text{Prediction}}}{\hat{y}_i})^2 \qquad\qquad \underset{\substack{\nearrow\\ \text{Weights}}}{w_j} = w_j - \underset{\substack{\uparrow\\ \text{Learning rate}}}{l_r}\partial\frac{L}{\partial w_j}$$

- Minimise loss function to get the best performing network
    - Predictions as close to true labels as possible
- Update the (initially not very good) network parameters by evaluating the derivative of the loss function w.r.t those parameters, and iterate!

Tools / Frameworks

- Many excellent software tools and frameworks are out there for building ML models, training and deploying them
- There are particularly good sets of tools in `Python`

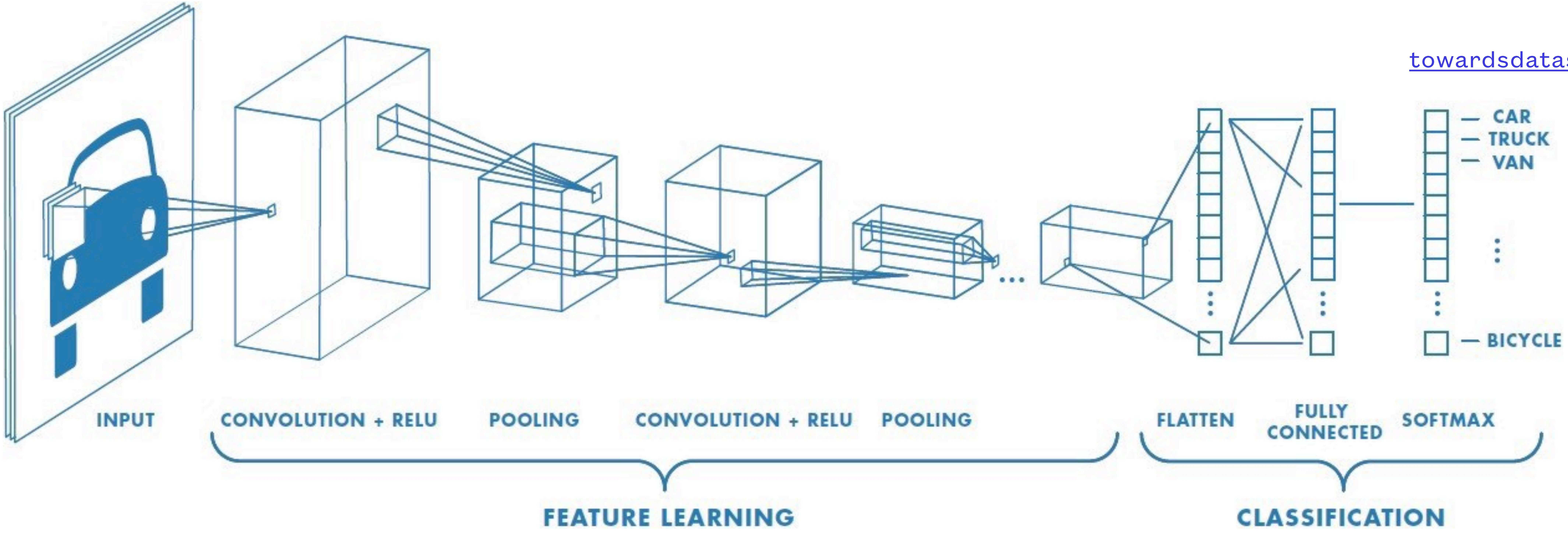
Dummy example - Keras NN

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from sklearn.model_selection import train_test_split
import uproot

X, y, = uproot.open(‘data.root’).arrays([…])
X_train, X_test, y_train, y_test = train_test_split(X, y)
inputs = Input(shape=(3,))
hidden = Dense(64, activation=‘relu’, input_shape=2, name=‘hidden’))(inputs)
output = Dense(1, activation=‘sigmoid’, name=‘output’))(hidden)
nn = Model(inputs=inputs, outputs=output)
nn.compile(optimizer=“Adam”, loss=“binary_crossentropy”, metrics=[“accuracy”])
nn.fit(X_train, y_train, batch_size=100, epochs=10)
nn.save(‘nn.h5’)
```

Convolutional Neural Networks

- Convolutional Neural Networks for images: apply *convolutional filters* - small neural networks - scanning over the pixels
  - Reduces the number of parameters compared to feeding the pixels into a Fully Connected NN
  - Adds translational invariance: the object in the image could be anywhere, and is filtered down by the convolutions

towardsdatascience.com [8]



Recurrent Neural Networks

- Built in *memory*
- Used for ordered data, e.g. time series, natural language processing
- Few different flavours: Long Short Term Memory (LSTM), Gate Recurrent Unit (GRU)

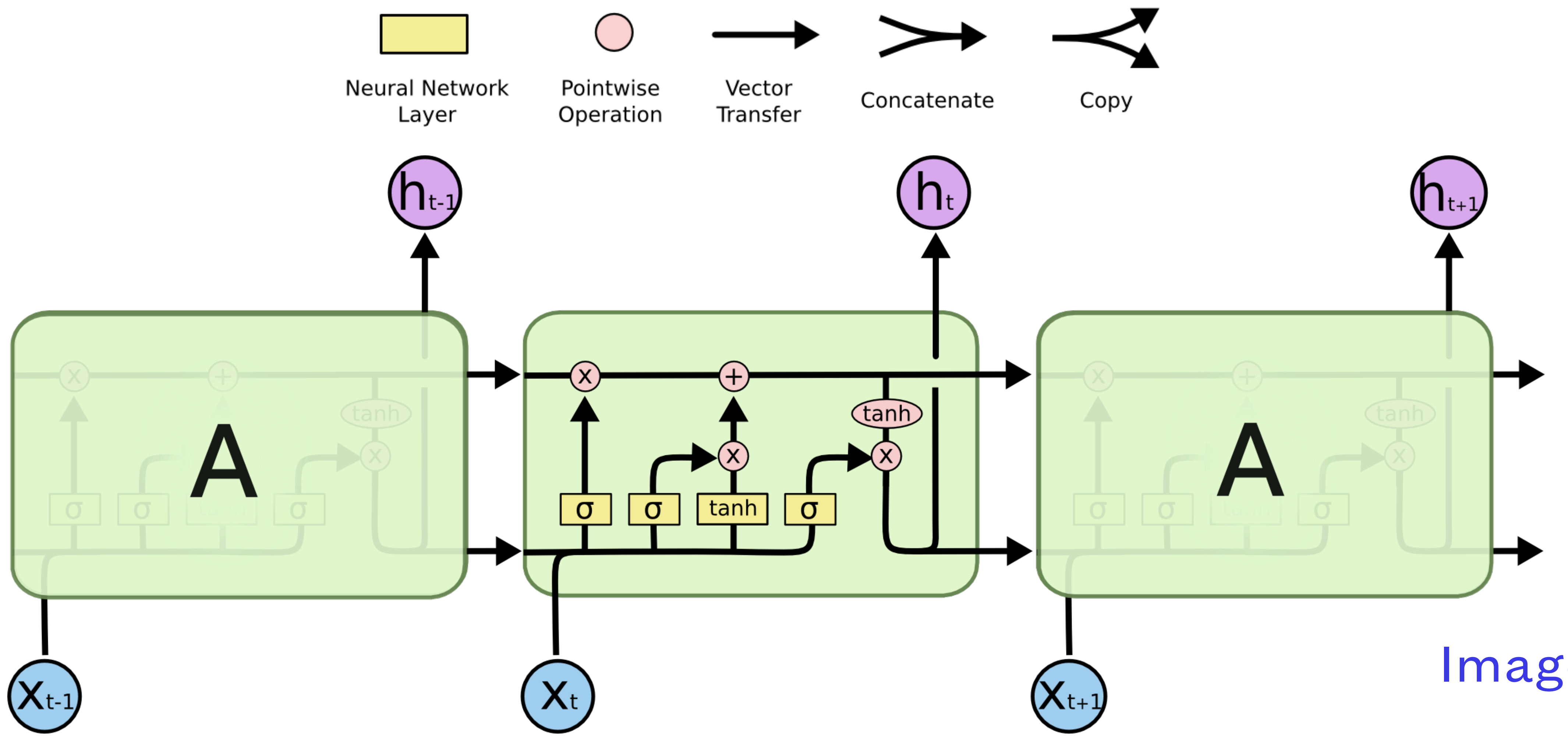
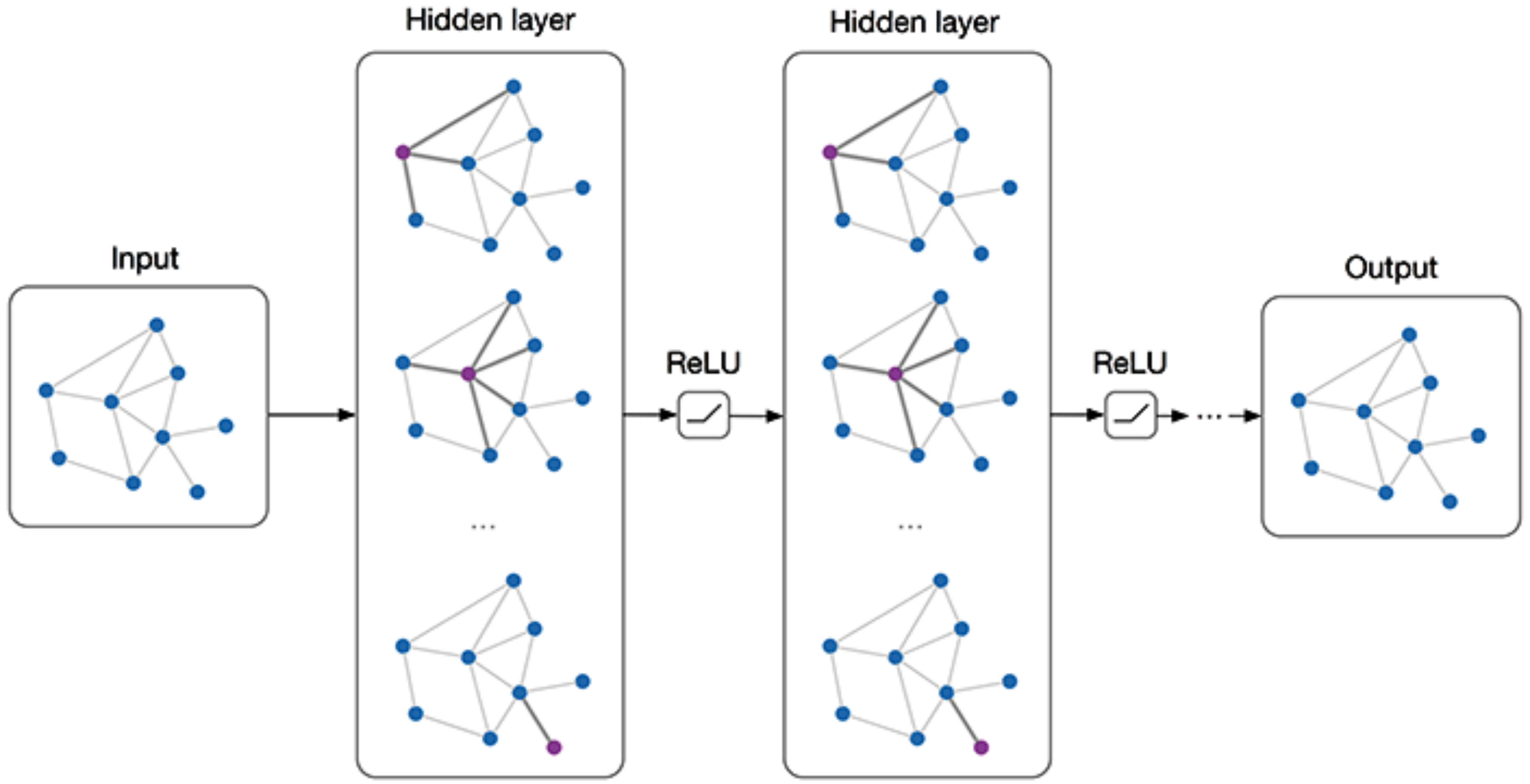


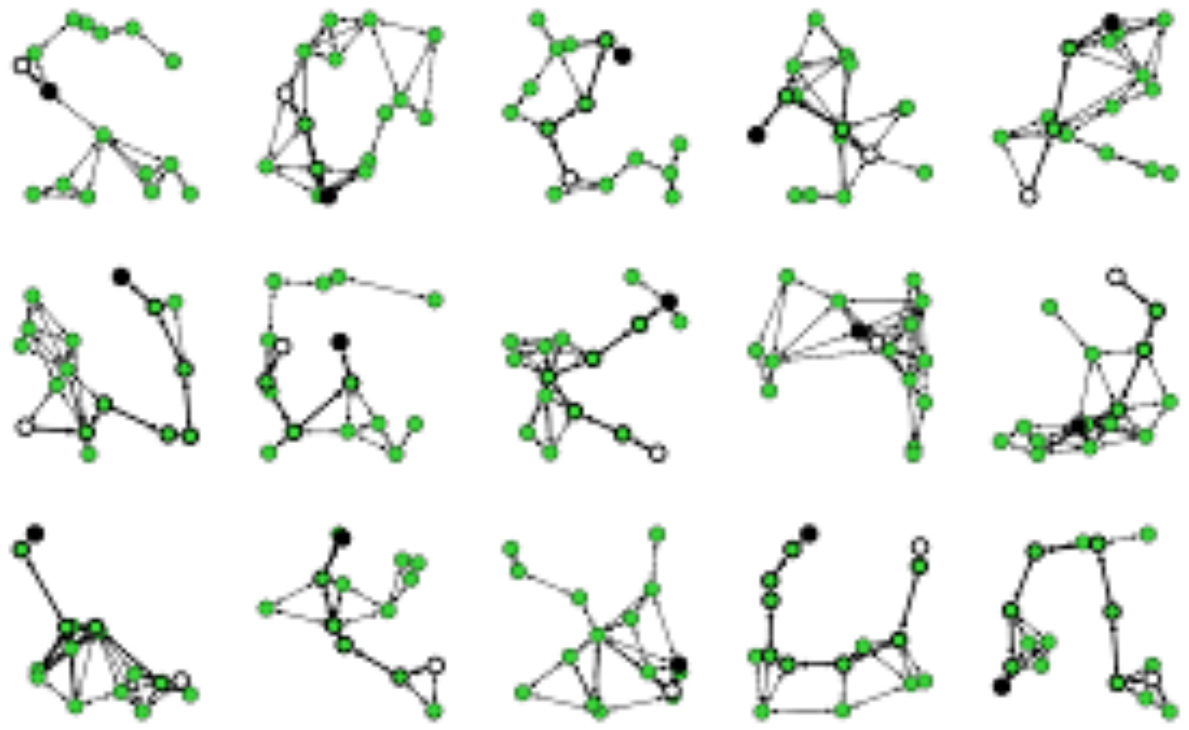
Image: colah's blog

- The LSTM cell has an internal state, and fully connected neural networks update this at each iteration
- Could be used, e.g. to predict the next word in a sentence

Graph Neural Networks

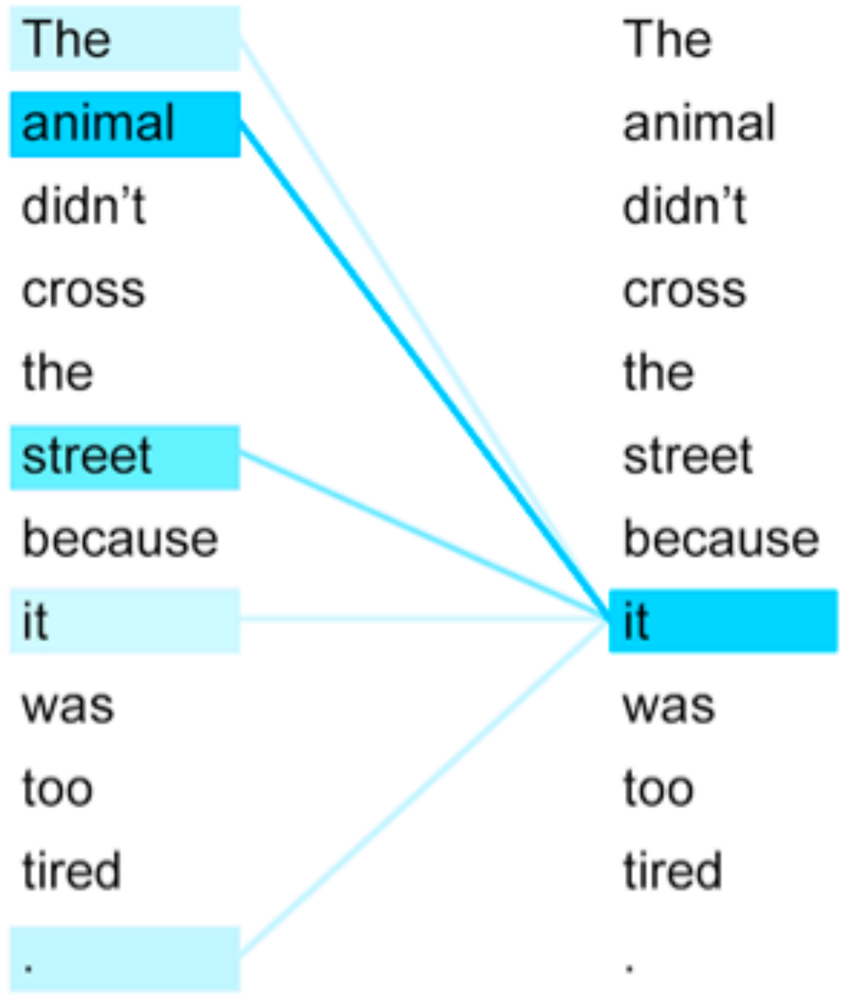
- Well suited to problems described by graphs of vertices and edges
- Cluster / classify data not only according to its coordinates, but its neighbourhood
- Iteratively update (strengthen/weaken) connections with fully connected or convolutional networks
- Used in, e.g., molecule synthesis for drug discovery
- Promising in HEP for multi-clusters in 'point cloud' like detectors (sparse images),
  - e.g. tracking, calorimetry in high pileup; hierarchical type problems, e.g. tracking, jets

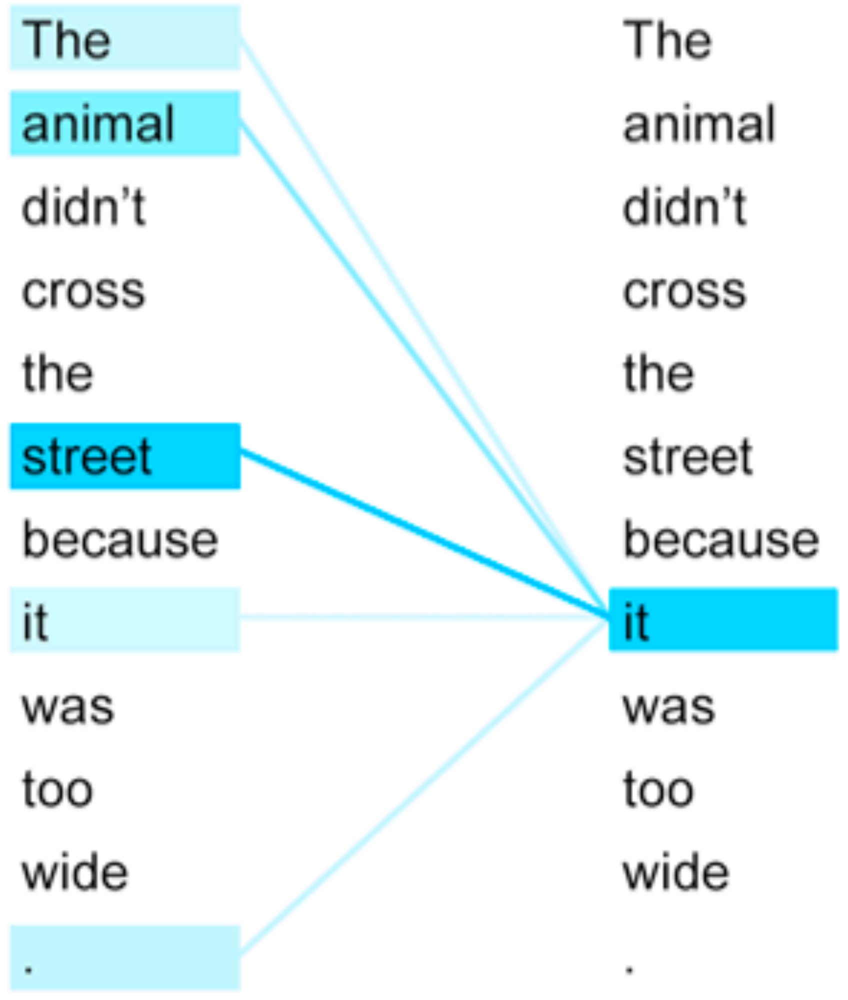


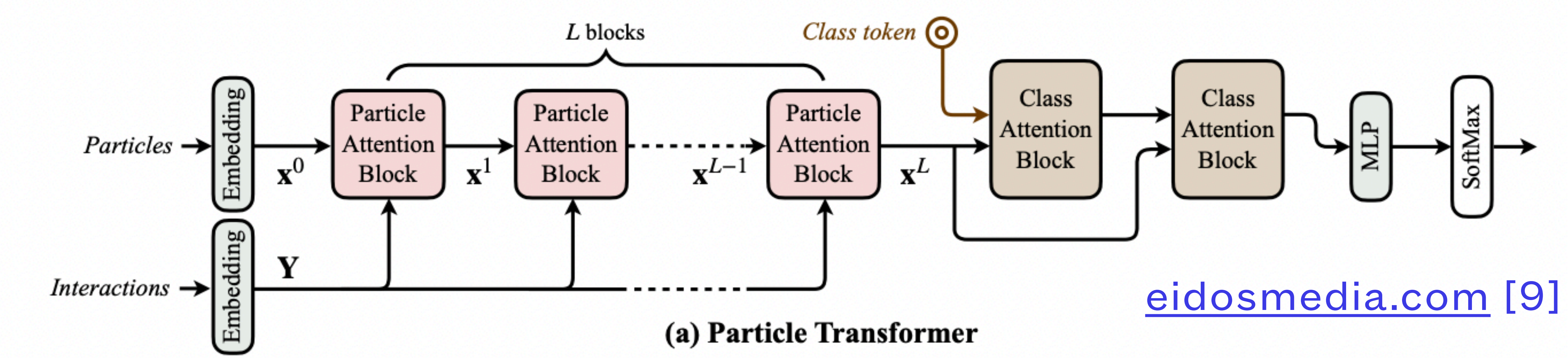


Transformers

- Sequence-to-sequence type problems
  - The big Natural Language Processing (NLP) models like BERT and GPT3
  - Billions of parameters
  - Unlike RNNs the full sequence enters at once - more paralellizable
- Attention mechanism - learning relationships / context
- Also relevant in HEP - Particle Transformer (ParT) (jet tagging)





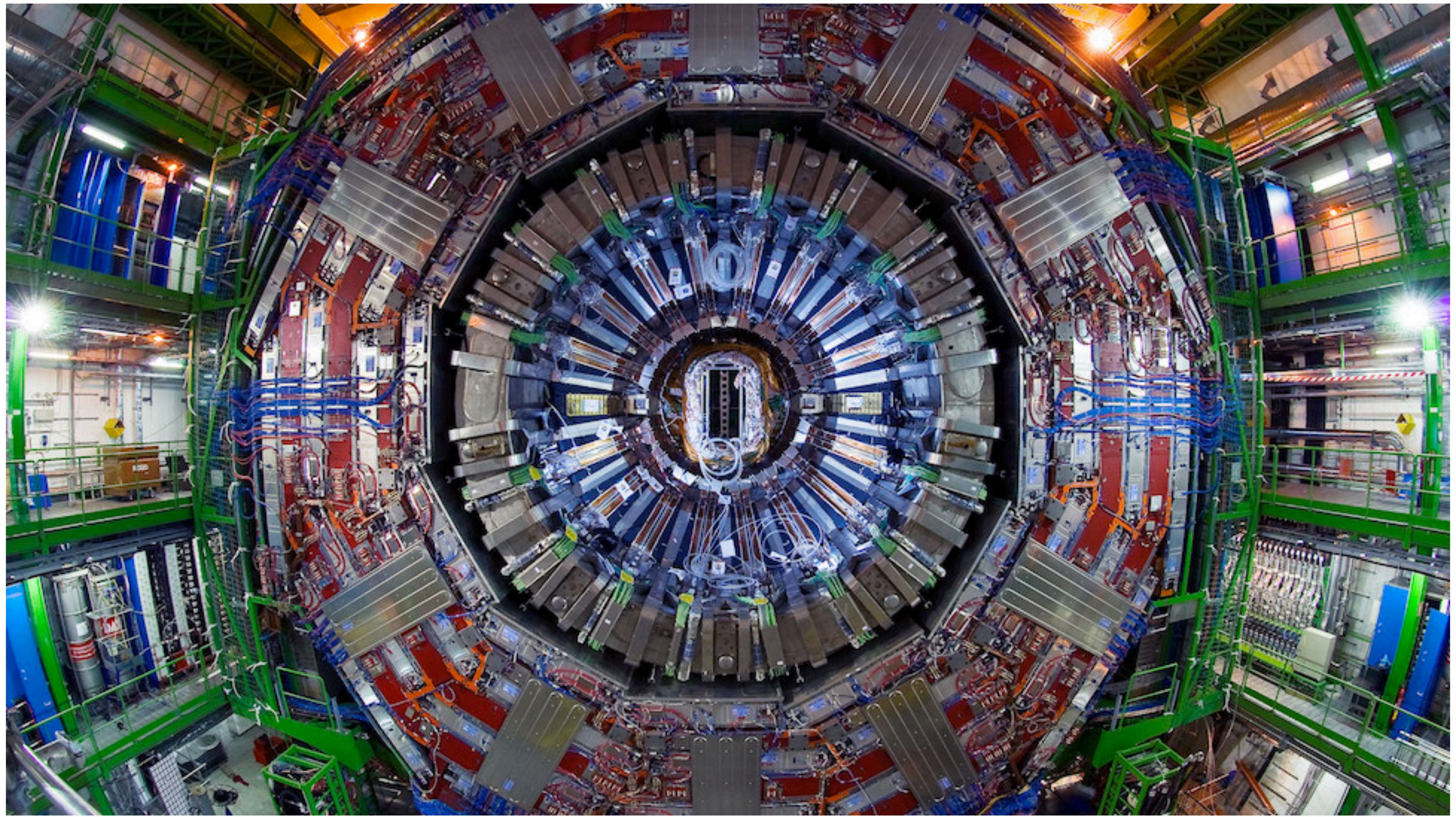


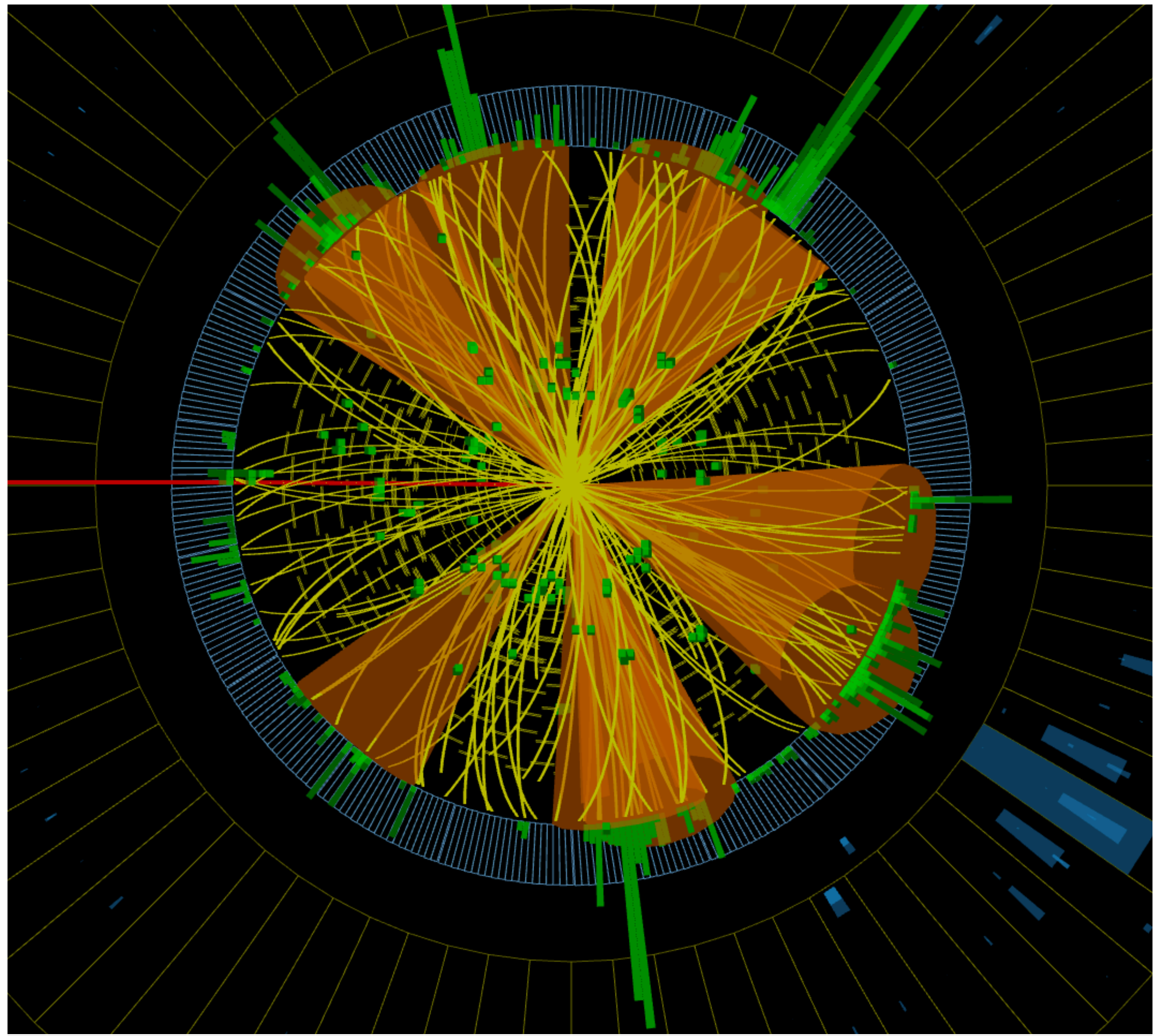
(a) Particle Transformer

eidosmedia.com [9]

General tips

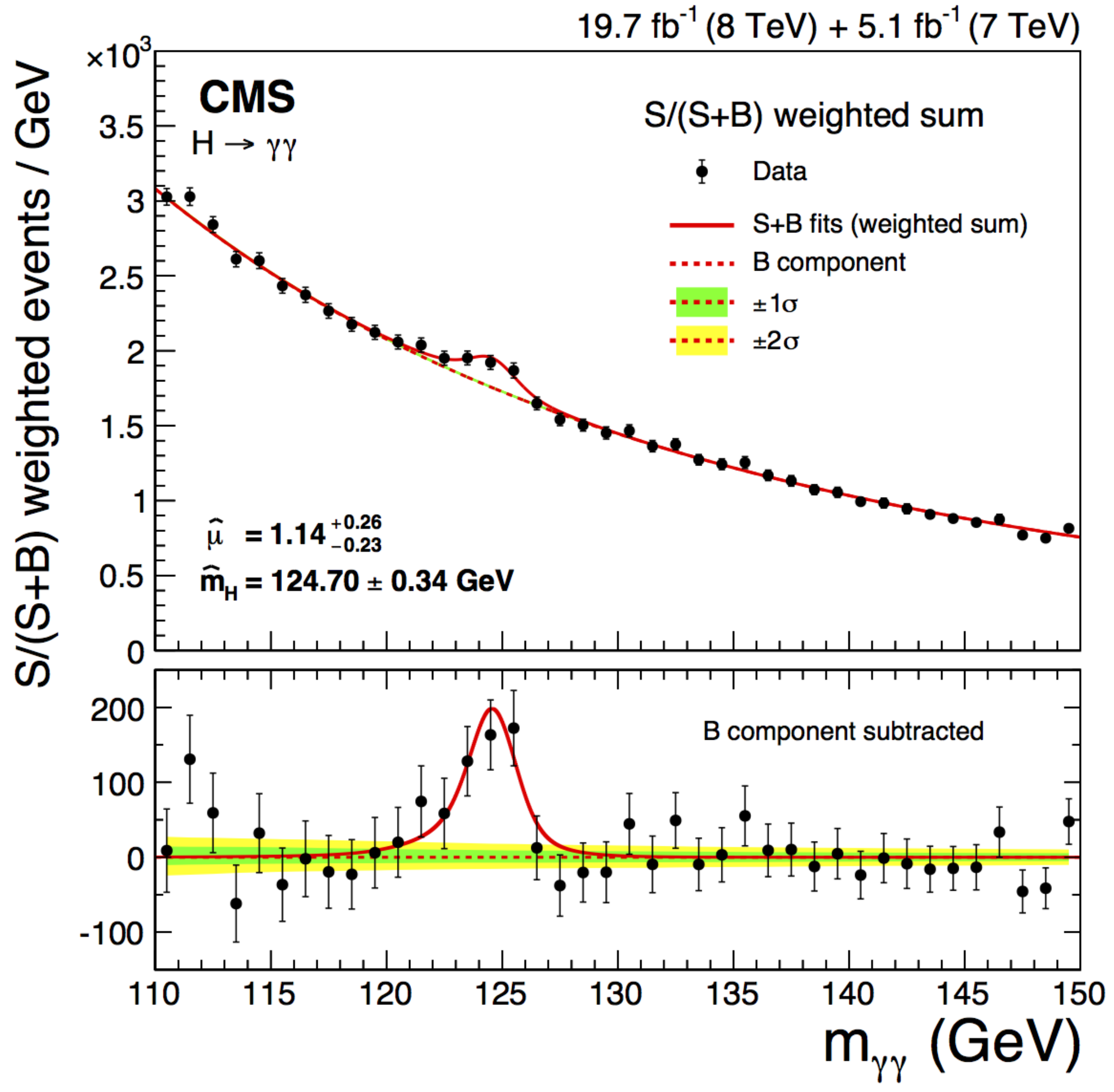
- There's a lot of tricks and a rich literature of best practises to get best performance (including computational):
- Training with *batches* - evaluate the gradient for the mean over a batch of samples rather than for every sample
- Tuning the learning rate, optimizer
- Choosing a loss function, activation function
- Choosing the best network architecture
  - Type of network, number of layers, number of neurons in each layer
- Hyperparameter scan / optimization - automatically search for the best solution to the above for your problem
  - e.g. Keras Tuner, Ray Tune
- Run network compression / pruning during training: improve robustness of your NN, *and* improve computational performance





Examples in High Energy Physics

Machine Learning @ LHC



- ML algorithms used offline for

- improving Higgs mass resolution with particle energy regression

- enhancing signal/background discrimination

- 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, ….

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

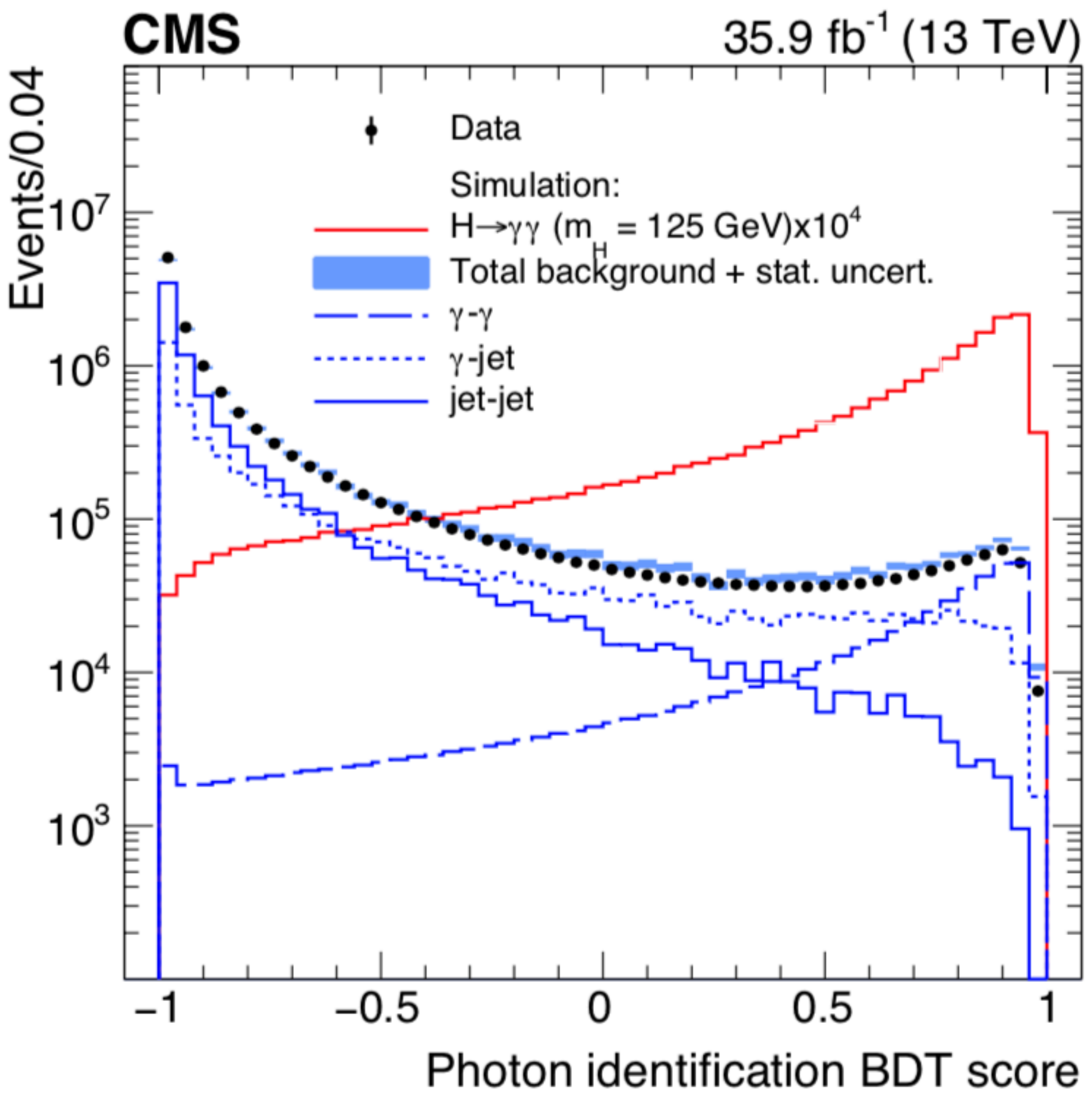
- How can we improve the trigger selection with ML?

- What can we do in ~ µs with an FPGA?

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

BDTs for Higgs

- Several BDTs involved in the analysis of Higgs boson decay to two photons using high-level variables
  - e.g. particle mass, η, isolation
- To separate signal photons from background (photons from jets)
- Choosing the most likely vertex for the photons (they are neutral, so no tracking)
- A diphoton quality BDT (separating signal like $\gamma\,\gamma$ events from background)
- Used to increase the purity of the selected diphoton dataset
- Increase in sensitivity due to ML equivalent to having 50% more data (and no ML)



arXiv:1804.02716v2

Neutrino Detector Reconstruction

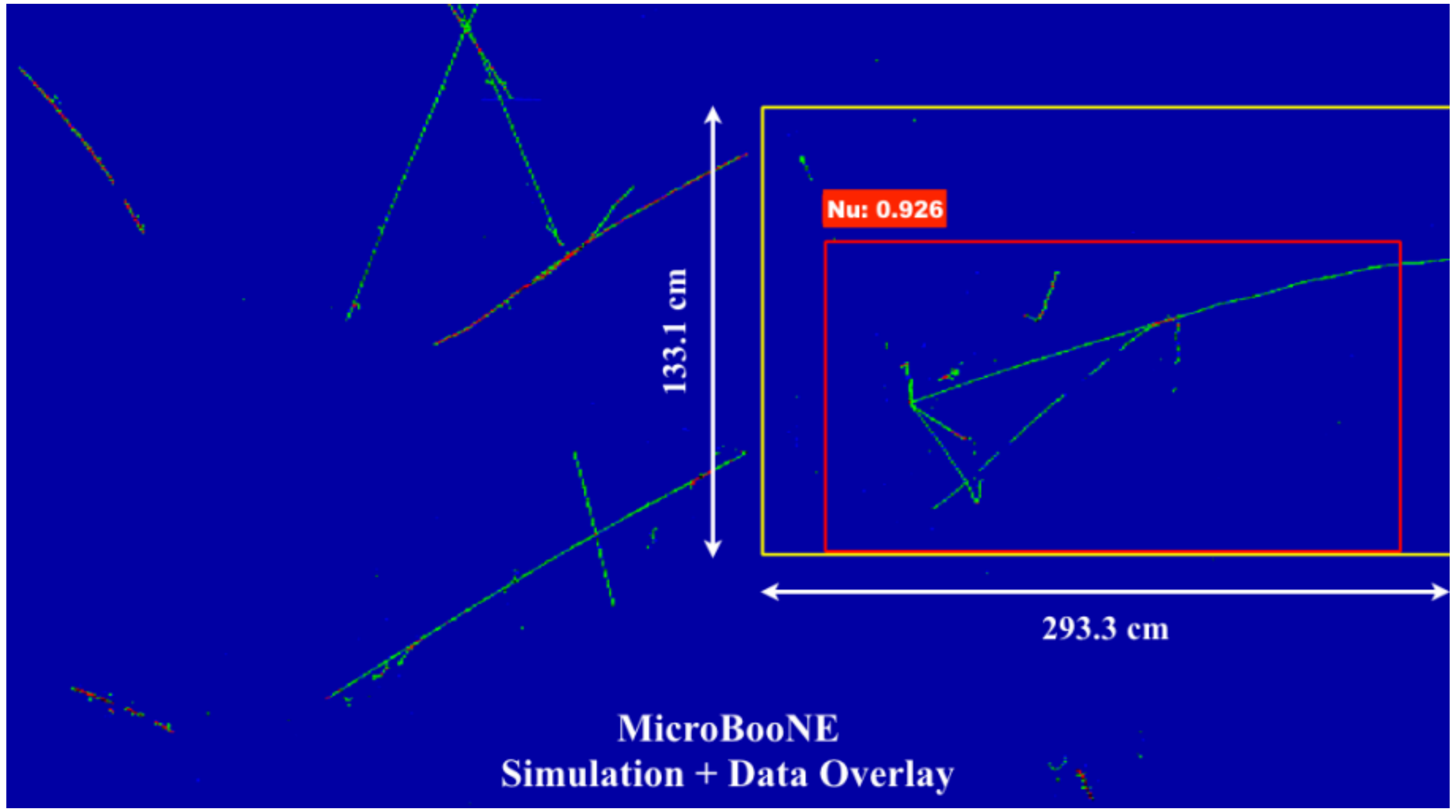
- From MicroBooNE, Liquid Argon time-projection chamber (LArTPC) neutrino experiment

- Using a CNN to identify neutrino interactions using a CNN

- e.g. simulated neutrino interaction yielding 1 μ, 3 p, 2 π. Background from cosmic data

- Yellow box is 'truth' box containing all charge deposits from simulated interactions

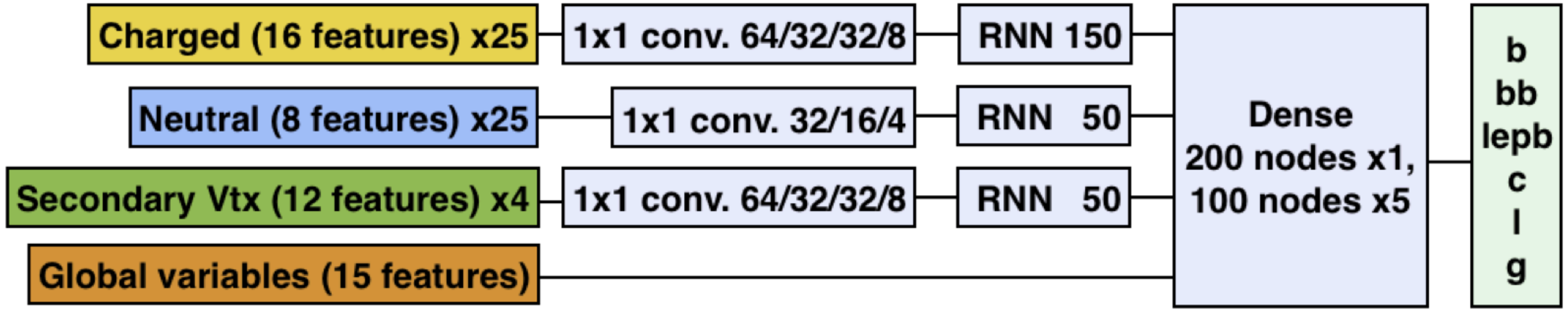
- Red is bounding box predicted by CNN



arxiv:1611.05531

Jet Tagging

- Big successes in HEP from ML for jet ID, example: DeepJet from CMS
- 1x1 CNN layers for ‘feature engineering’ (combining variables of single particles)
- LSTM recurrent networks iterate over particles sequentially
- Finally Dense layers combine features learned from the previous steps and the global variables



Charged (16 features) x25

Neutral (8 features) x25

Secondary Vtx (12 features) x4

Global variables (15 features)

1x1 conv. 64/32/32/8

1x1 conv. 32/16/4

1x1 conv. 64/32/32/8

RNN 150

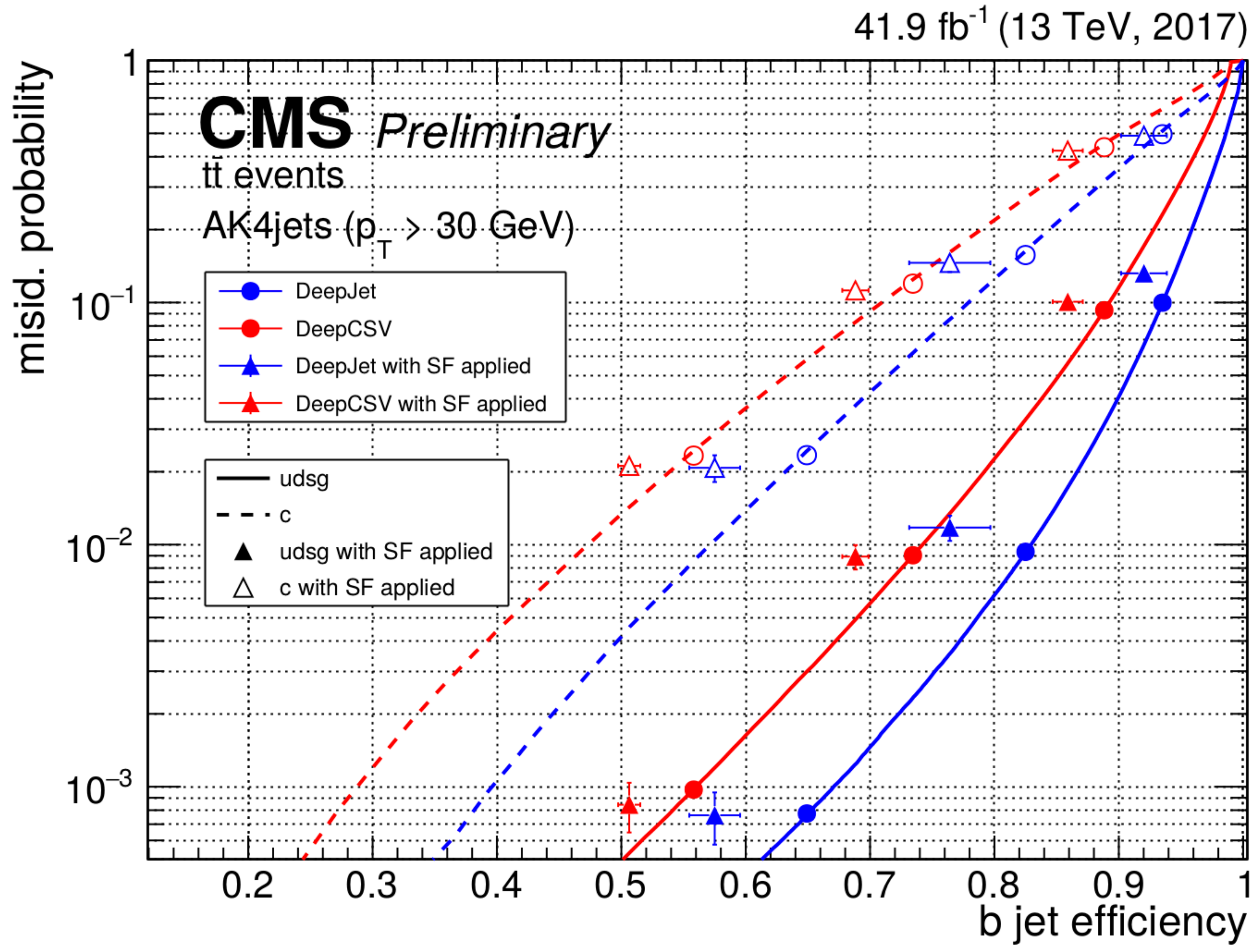
RNN 50

RNN 50

Dense 200 nodes x1, 100 nodes x5

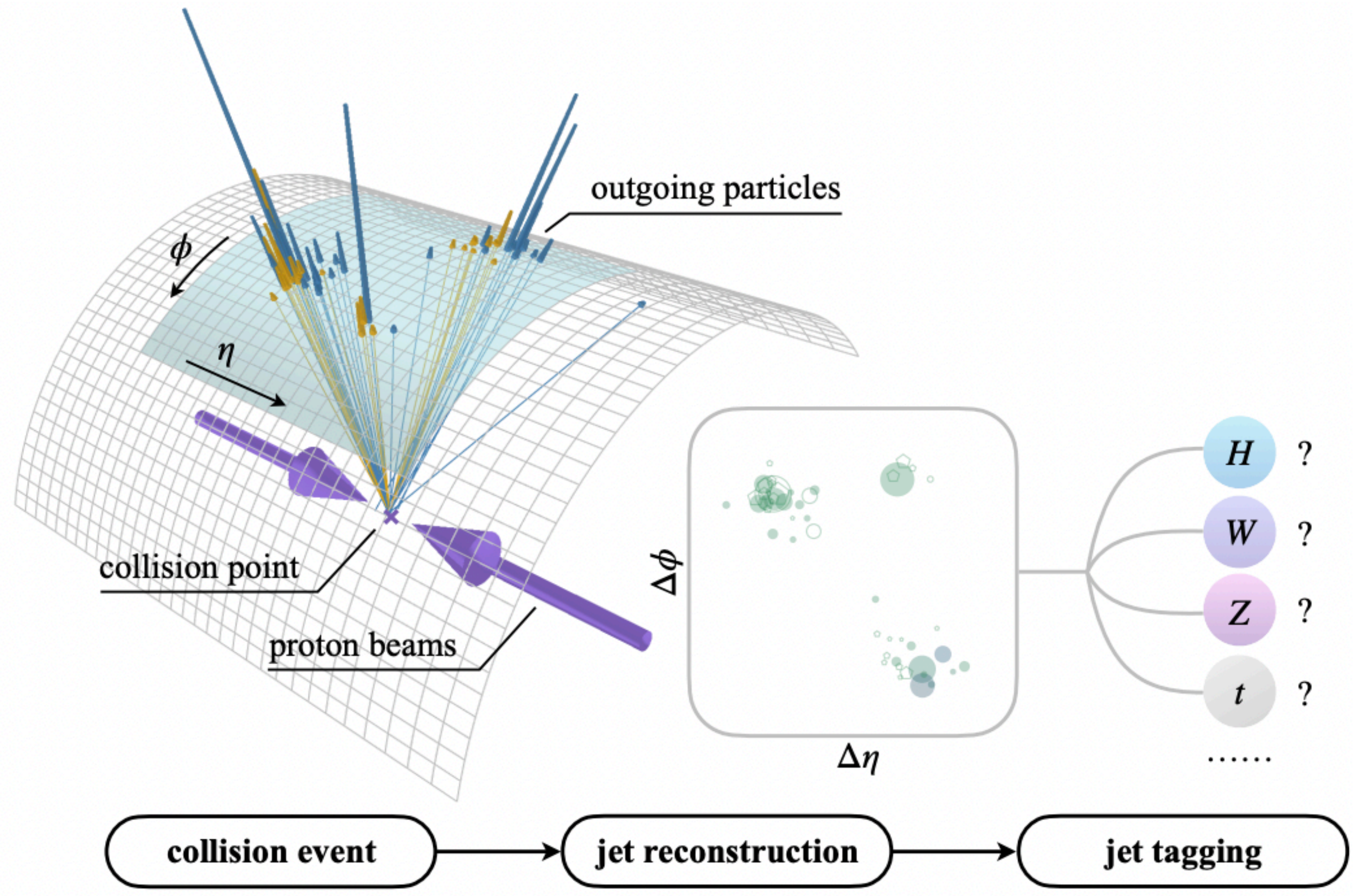
b bb lepb c l g

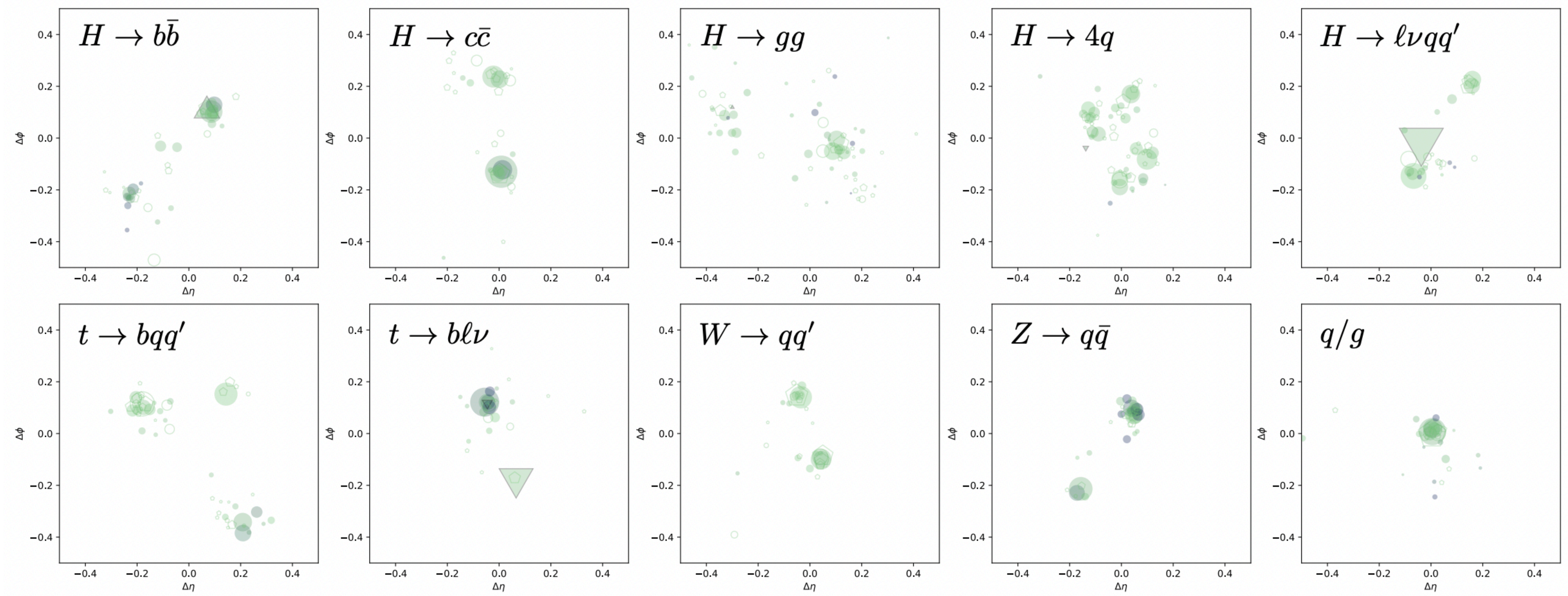
CMS-DP-2018-058



Jet Tagging

- Jet tagging is an area of HEP rich in ML: given the final state observables, what type of particle initiated the jet?
- How to represent the jet? Lots of approaches have been tried, relating to the different NN architectures
  - High-level observables reconstructed with *classical* means -> fed into MLP
  - Make images from individual particles by applying a grid -> Convolutional NN
  - Make lists of particles (often p_T ordered) -> Recurrent NN or Transformer
  - Represent particles as a graph (point cloud with connections) -> Graph NN

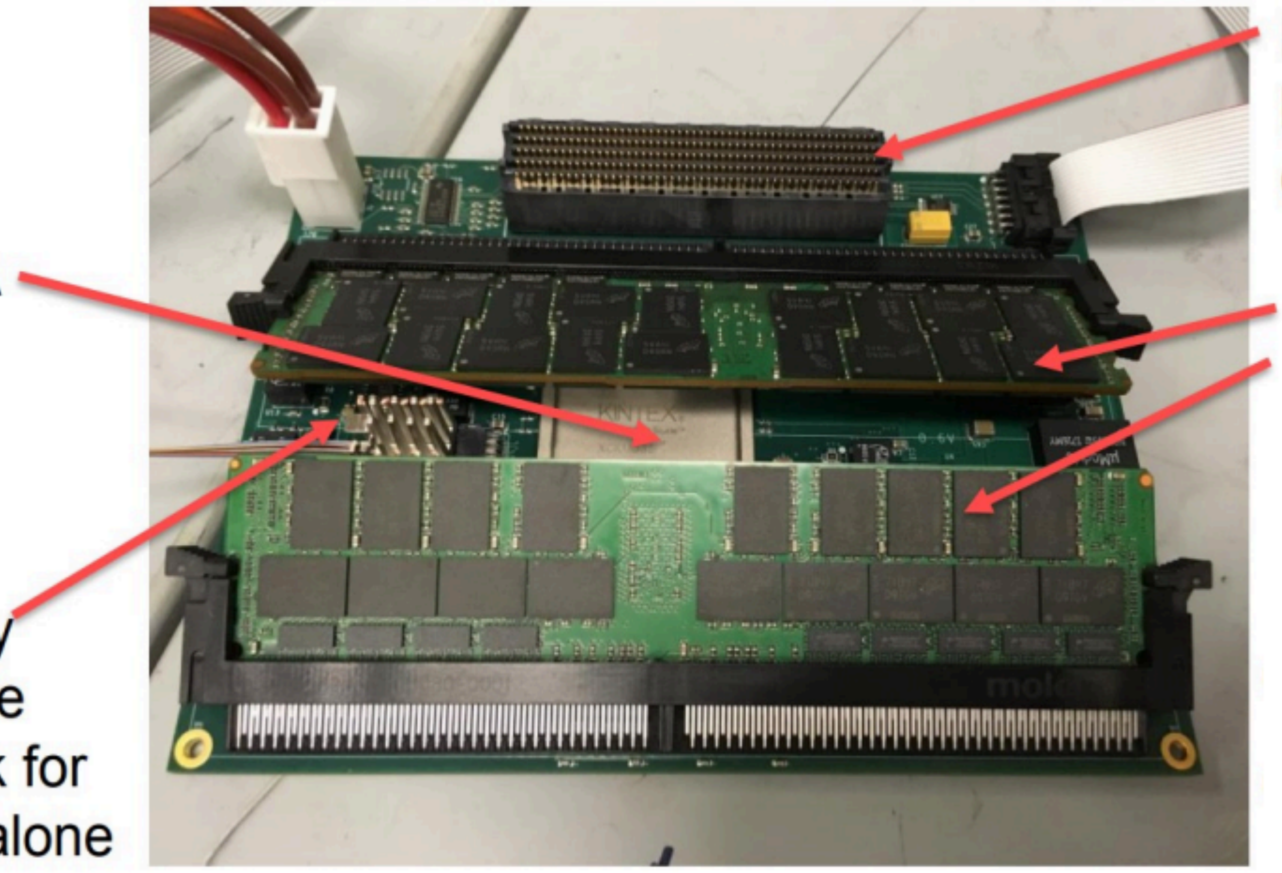




arXiv:2202.03772

CMS Level 1 Trigger Endcap Muon

- BDT to fit the muon momentum from hits in the muon stations
- Complicated geometry and magnetic field makes an ML solution useful
- Deployed using a ‘large LUT’ implemented in DDR on a mezzanine card to the FPGA
- BDT is evaluated for every possible input, with the output written at that position in the LUT

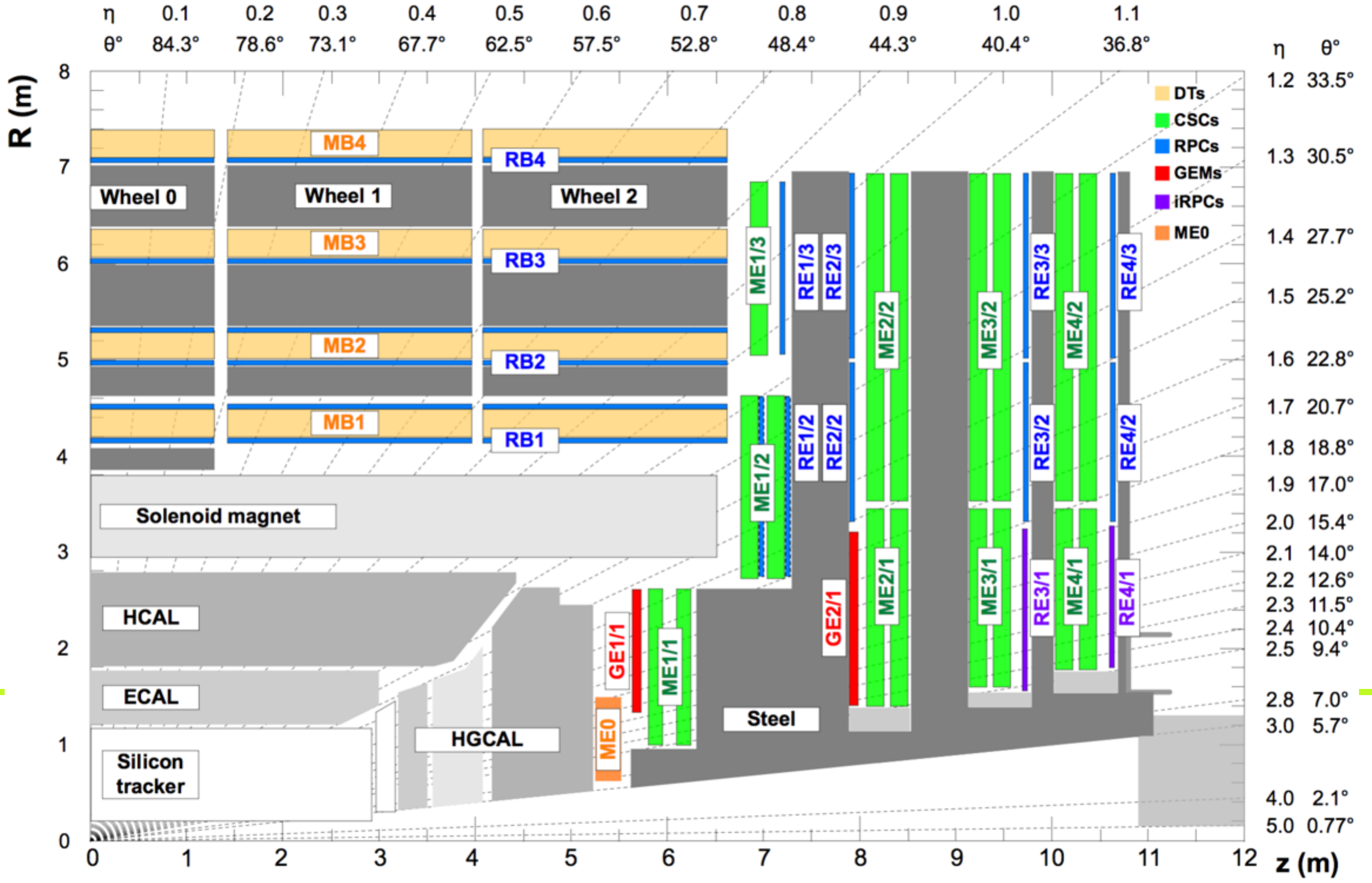


Base board connector

DDR4 LRDIMMs 64 GB each

FPGA

FireFly module (uplink for standalone tests)



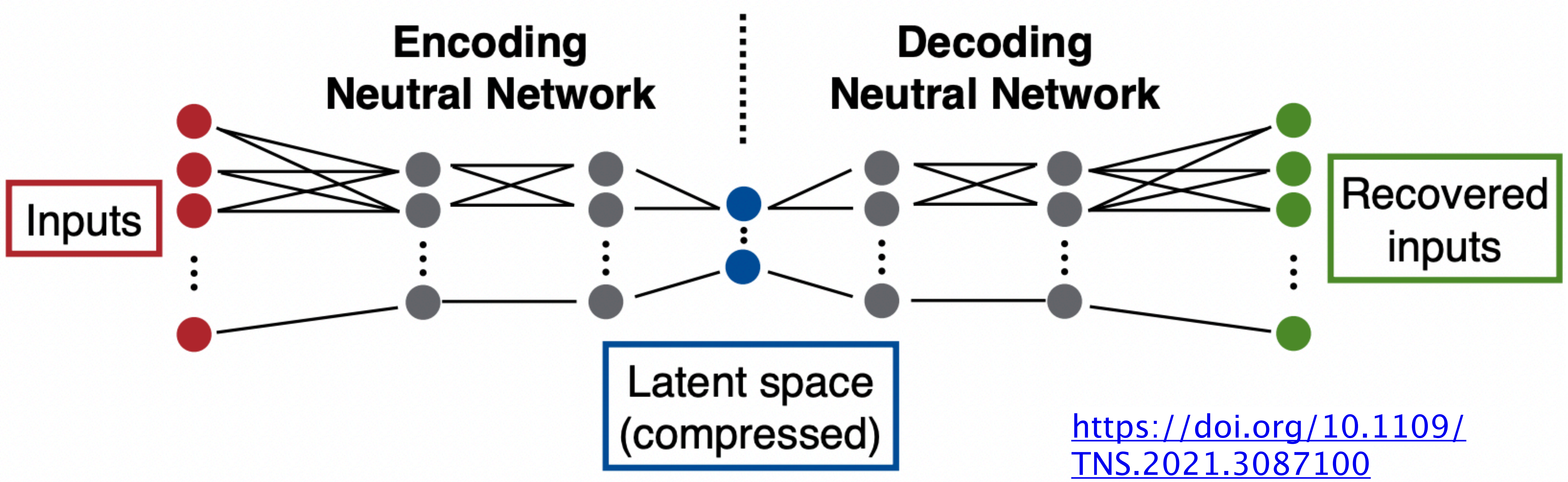
LHCb, Bonsai BDT

- In LHCb, Bonsai BDT has been used since the beginning of LHC data taking in their online software event selection
- Bonsai BDT is a technique to compress BDTs into a binned parameter space for faster execution
- Was used in the main selection path for most LHCb analyses

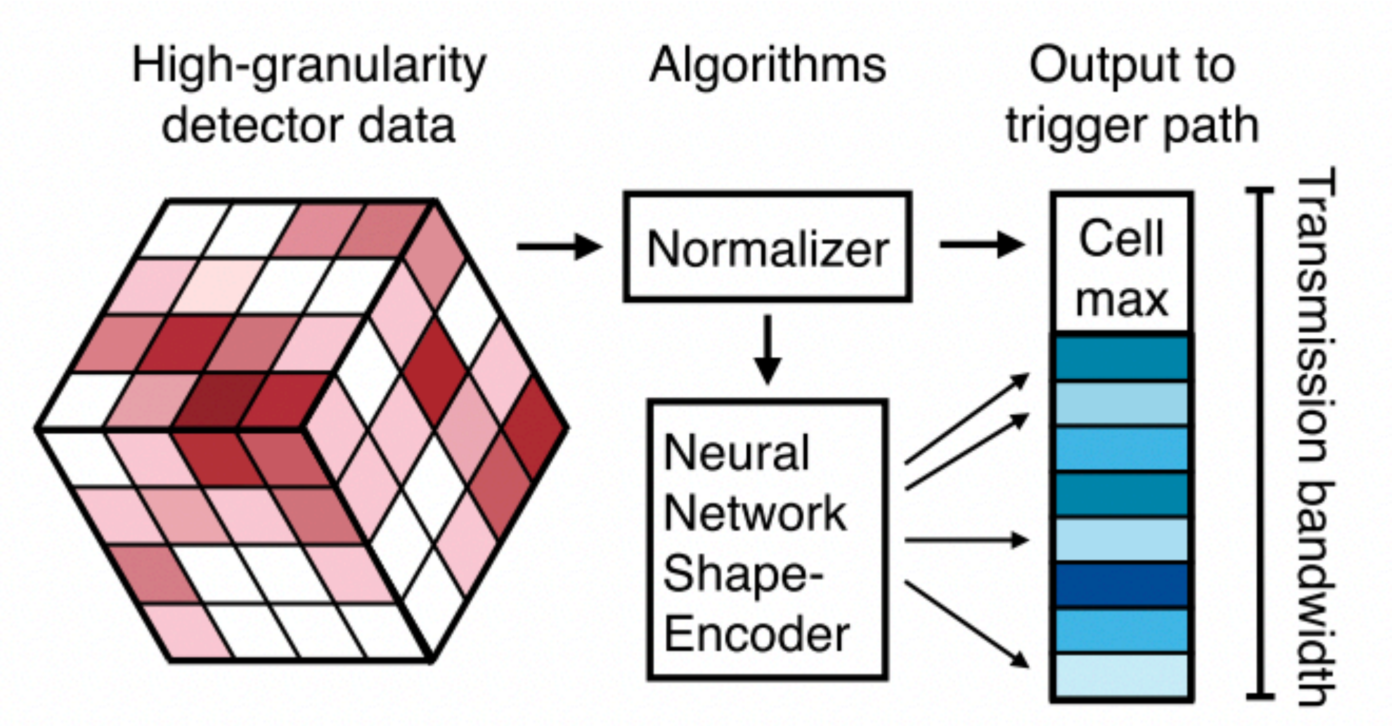
ECON-T ASIC for CMS HGCal

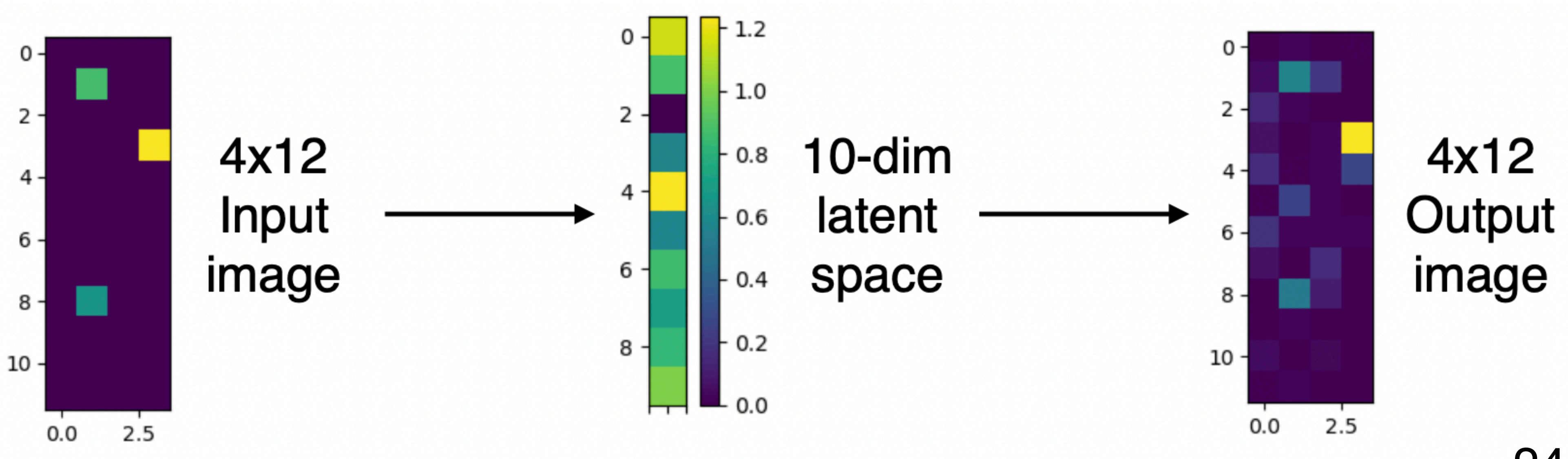
- Compress data to be sent to trigger FPGAs with an AutoEncoder, decode off detector

On detector → lpGBT → Off detector (trigger)



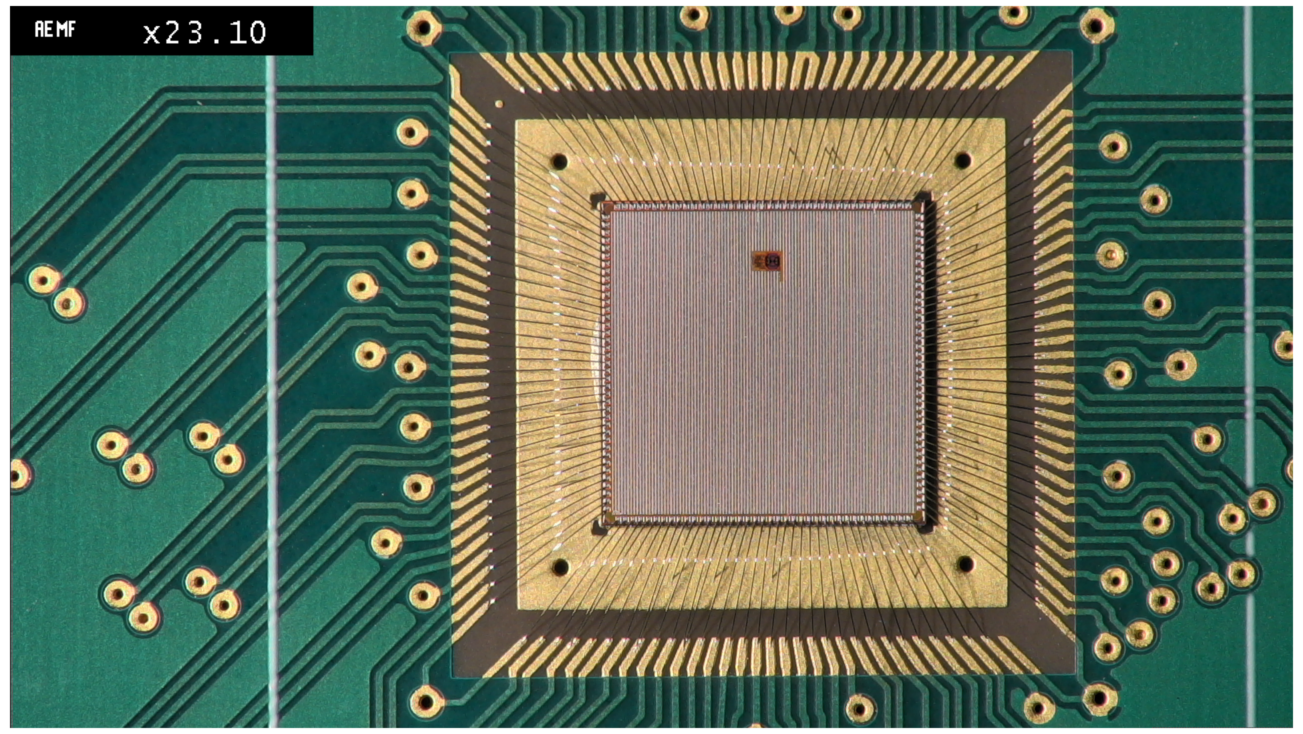
https://doi.org/10.1109/TNS.2021.3087100

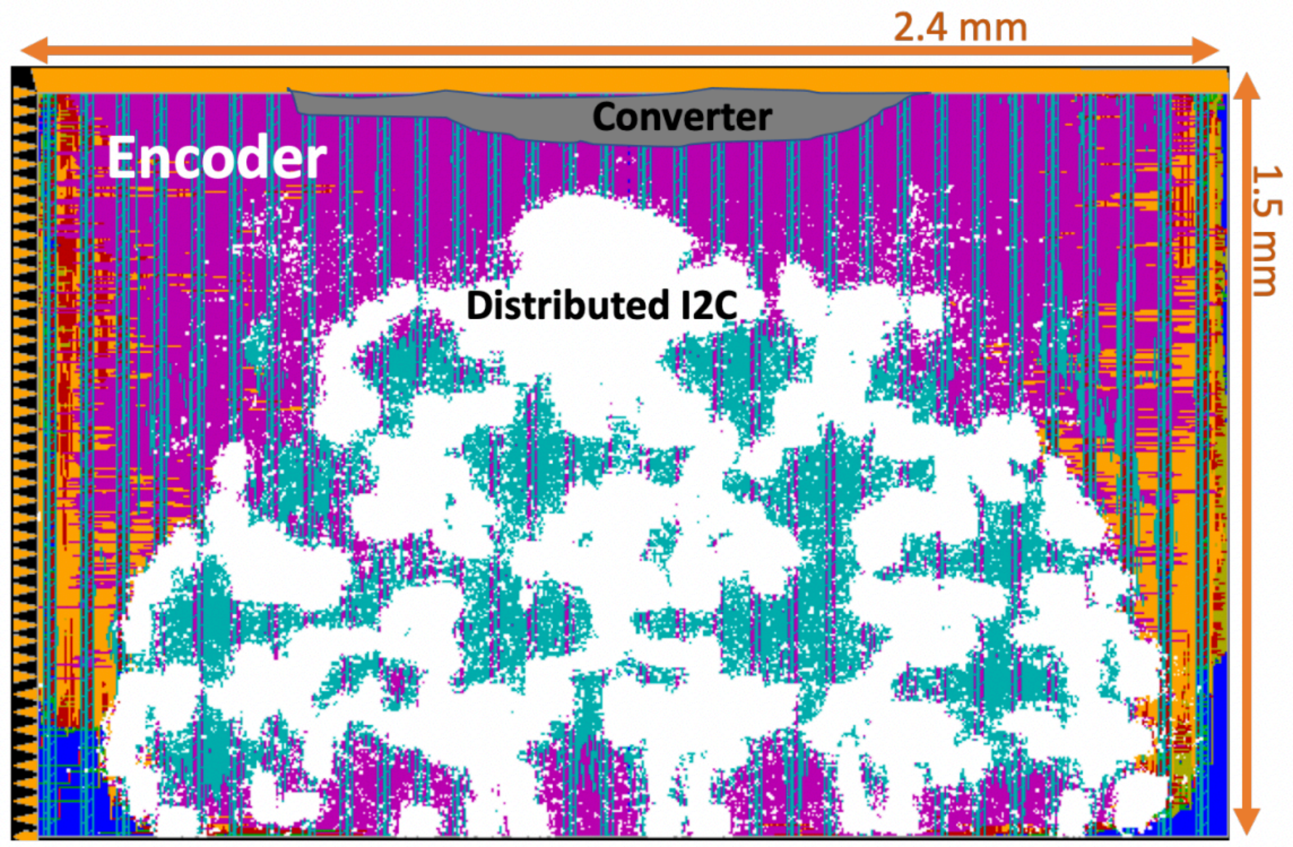


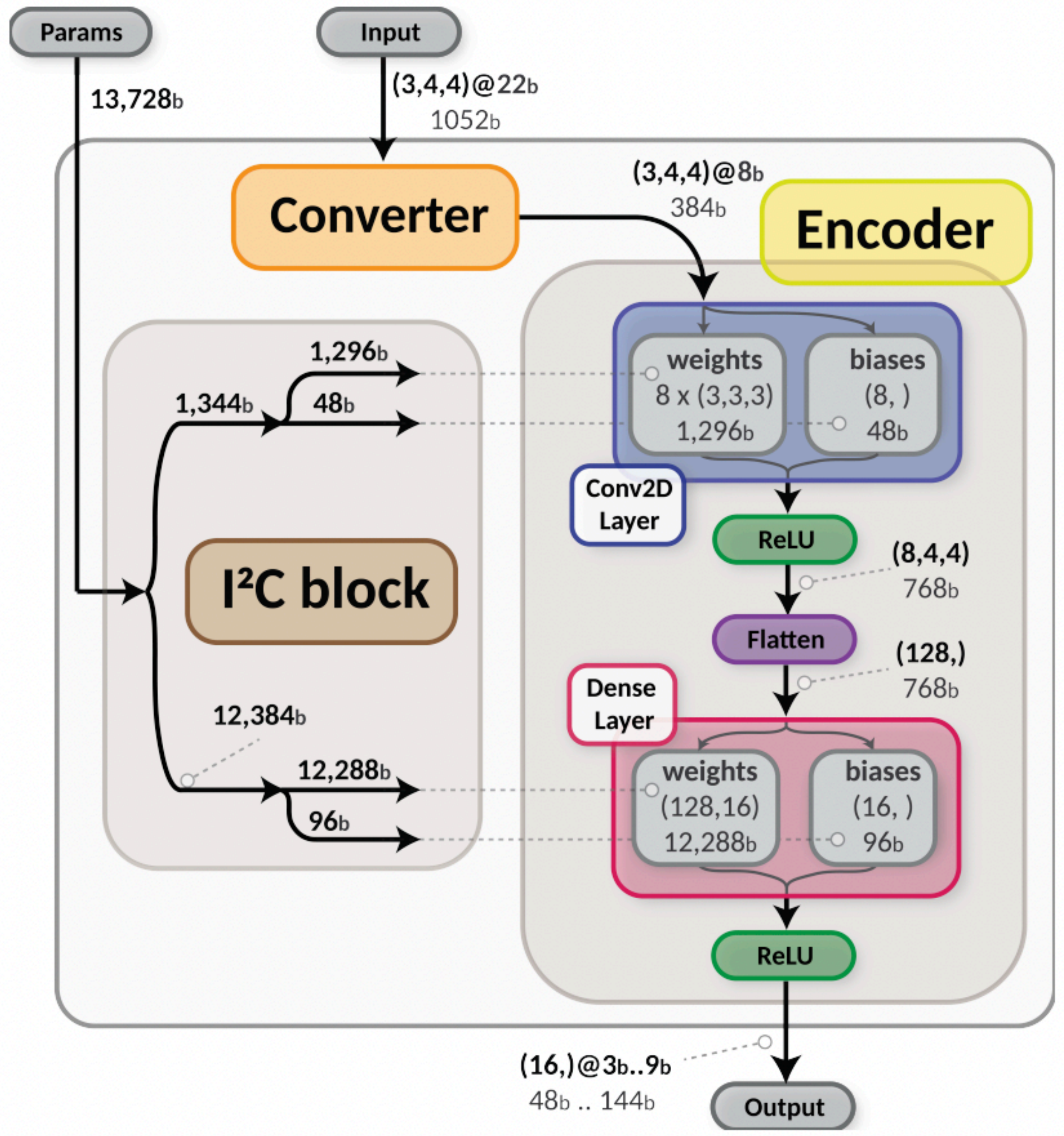


ECON-T ASIC for CMS HGCal

- Neural Net encoder IP block created for ECON-T ASIC with Catapult HLS (Mentor/Siemens) and hls4ml (more later)
    - NN architecture is fixed, weights can be reprogrammed (e.g. after NN retraining)
    - ECON-T also includes non-ML baseline compression algorithms
- Decoder block would run in trigger FPGAs
- Device manufactured and undergoing testing

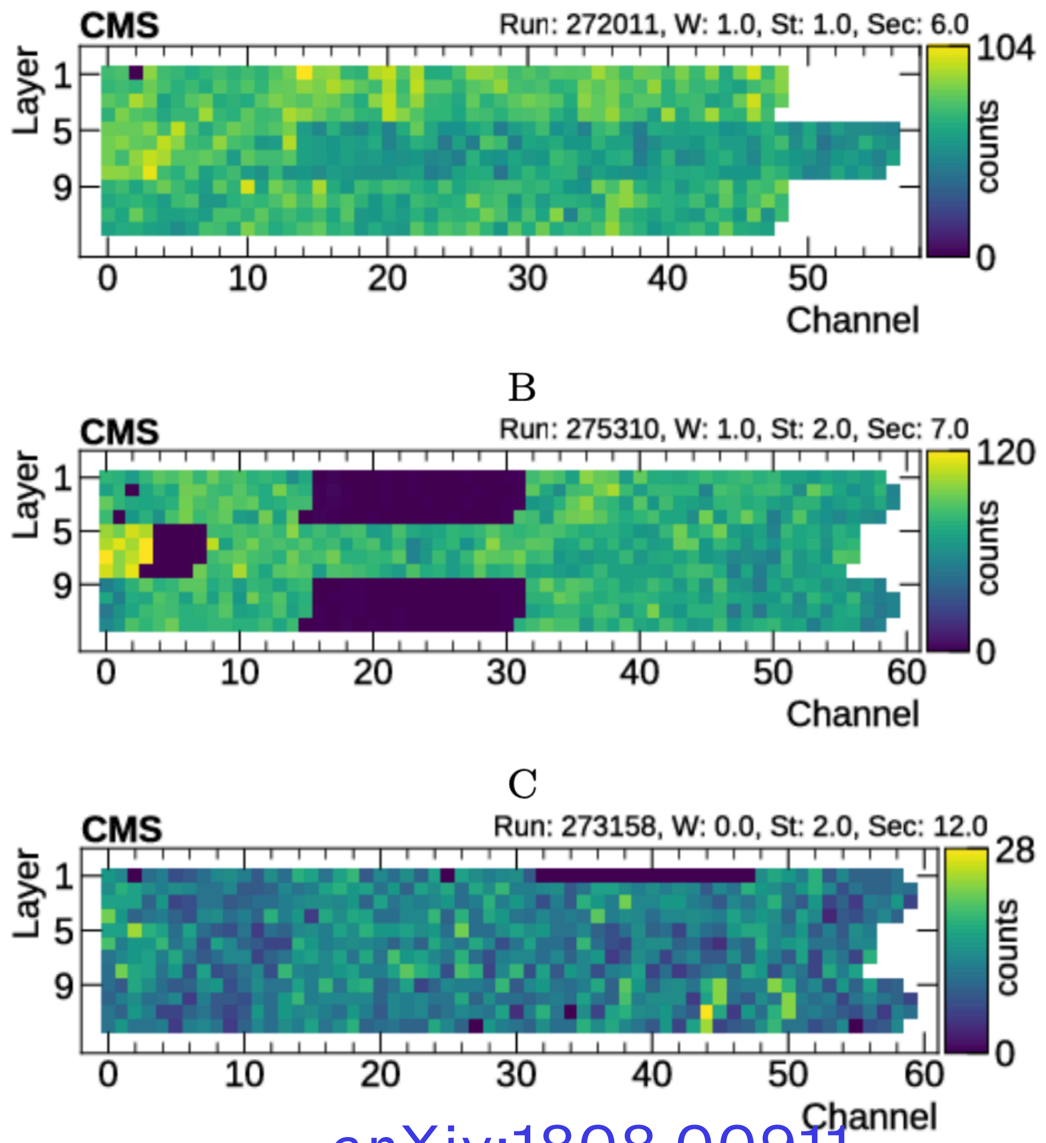




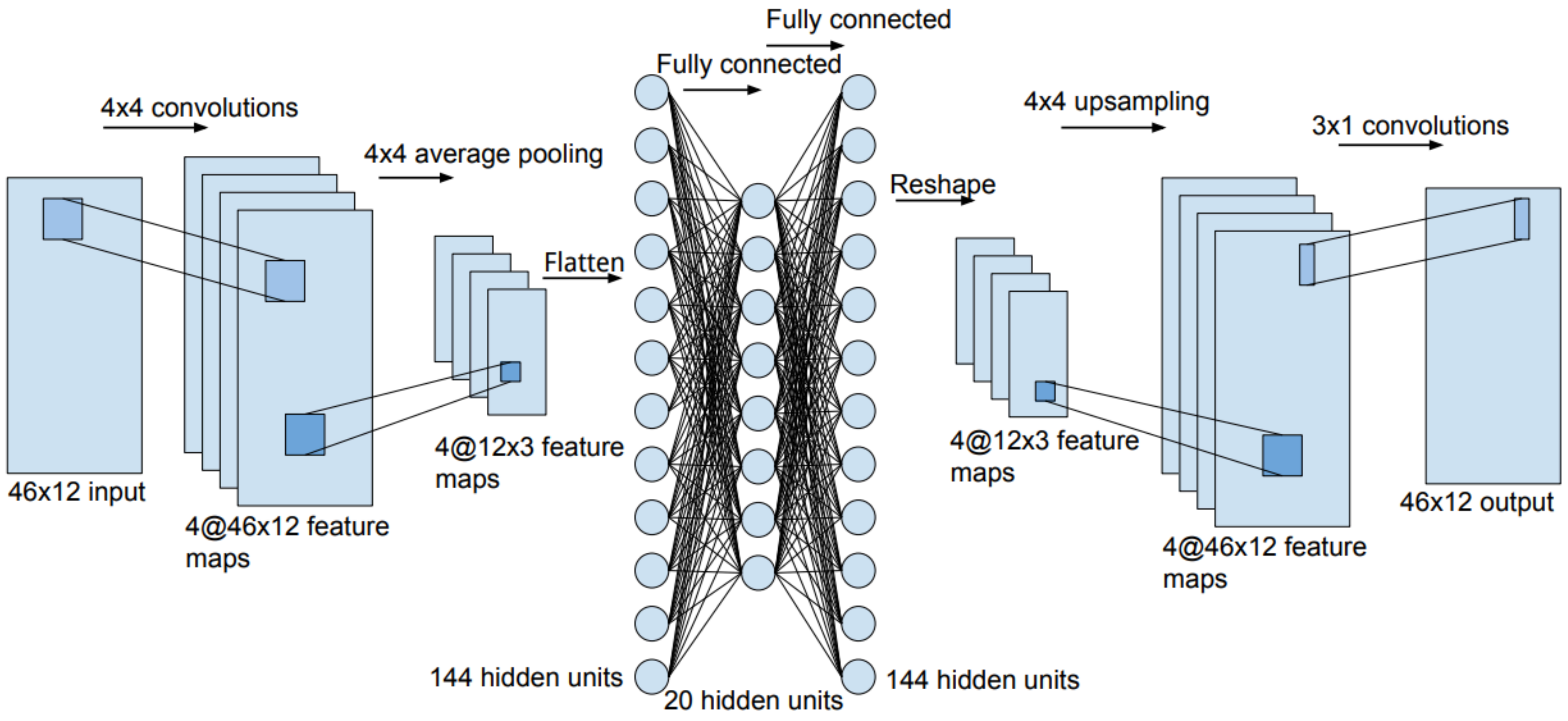


Data Quality Monitoring

- Using an Autoencoder for anomaly detection
  - Network has a ‘bottleneck’ that learns an abstract representation of the data
  - After bottleneck, decoder network tries to reproduce the input image
  - For anomalous input, the recreated image is not similar to the original input, and flagged
- Applied to CMS muon drift tube system, able to identify failures not spotted by previous, rule based system

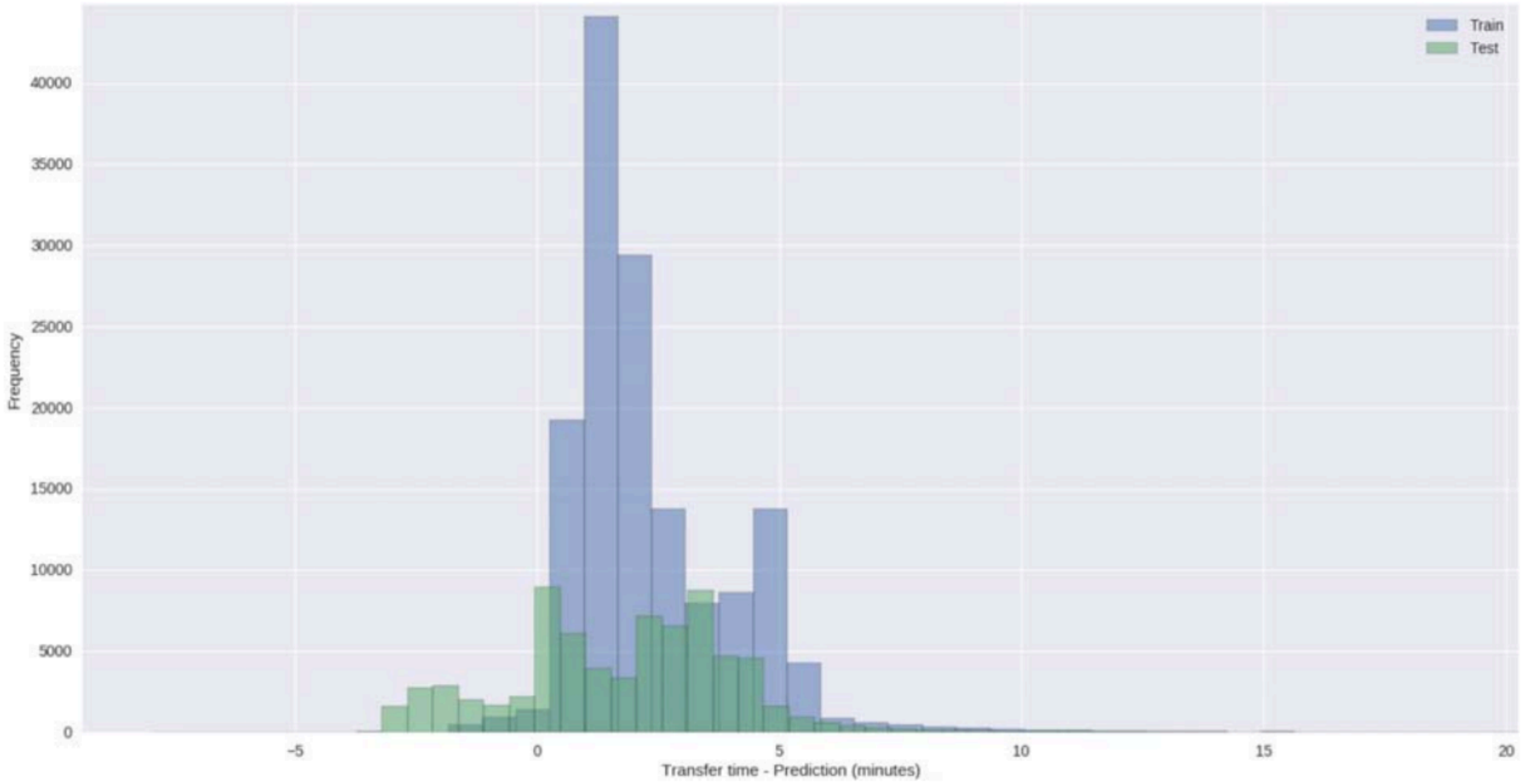


arXiv:1808.00911

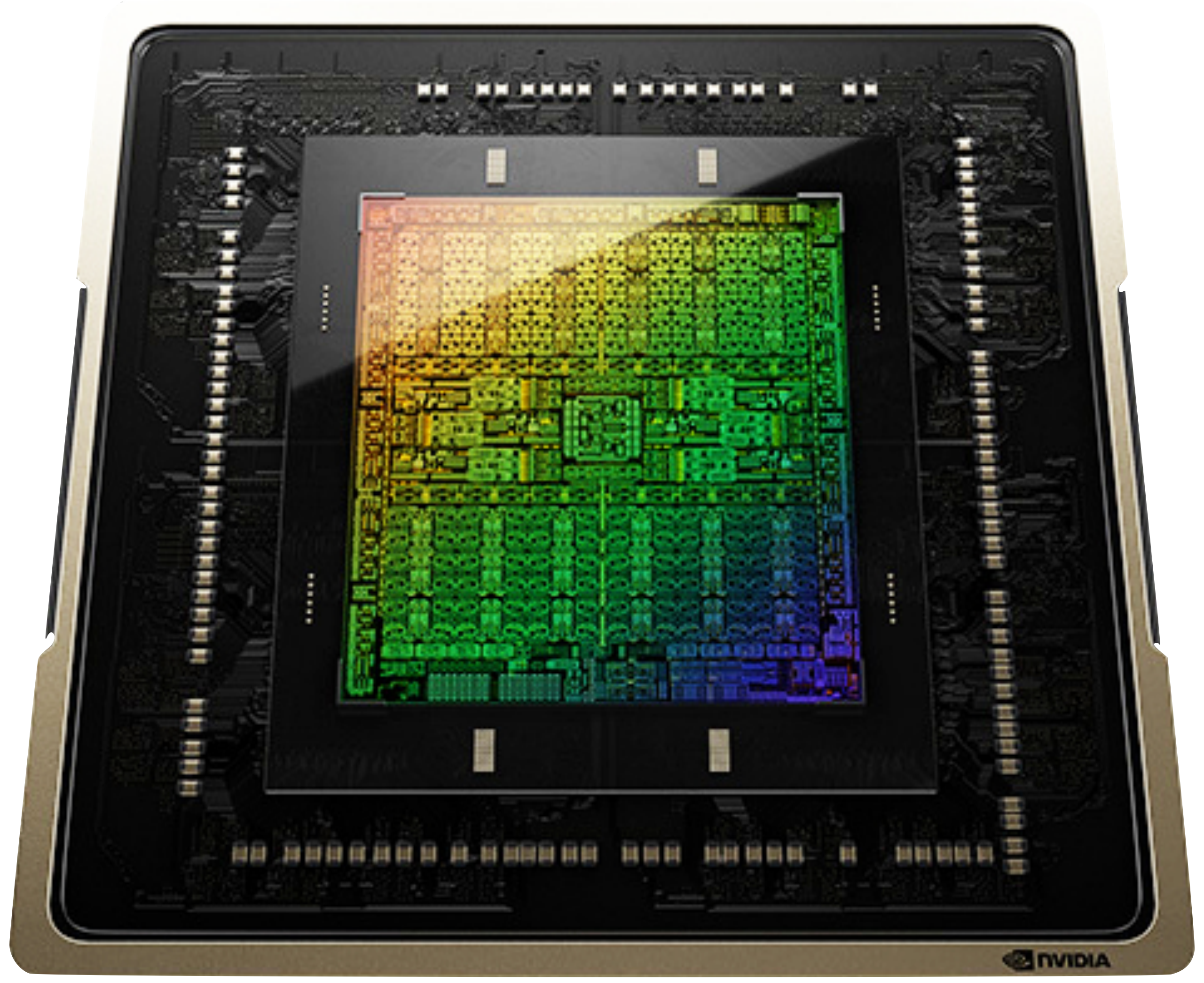


ML For Networking

- From ATLAS, predicting the transfer time of files between sites
- One metric in determining the network-aware scheduling of GRID jobs and file storage
- Uses a Long Short Term Memory (LSTM)
- Inputs: source, destination, activity, bytes, start timestamp, and end timestamp



doi :10.1088/1742-6596/898/6/062009

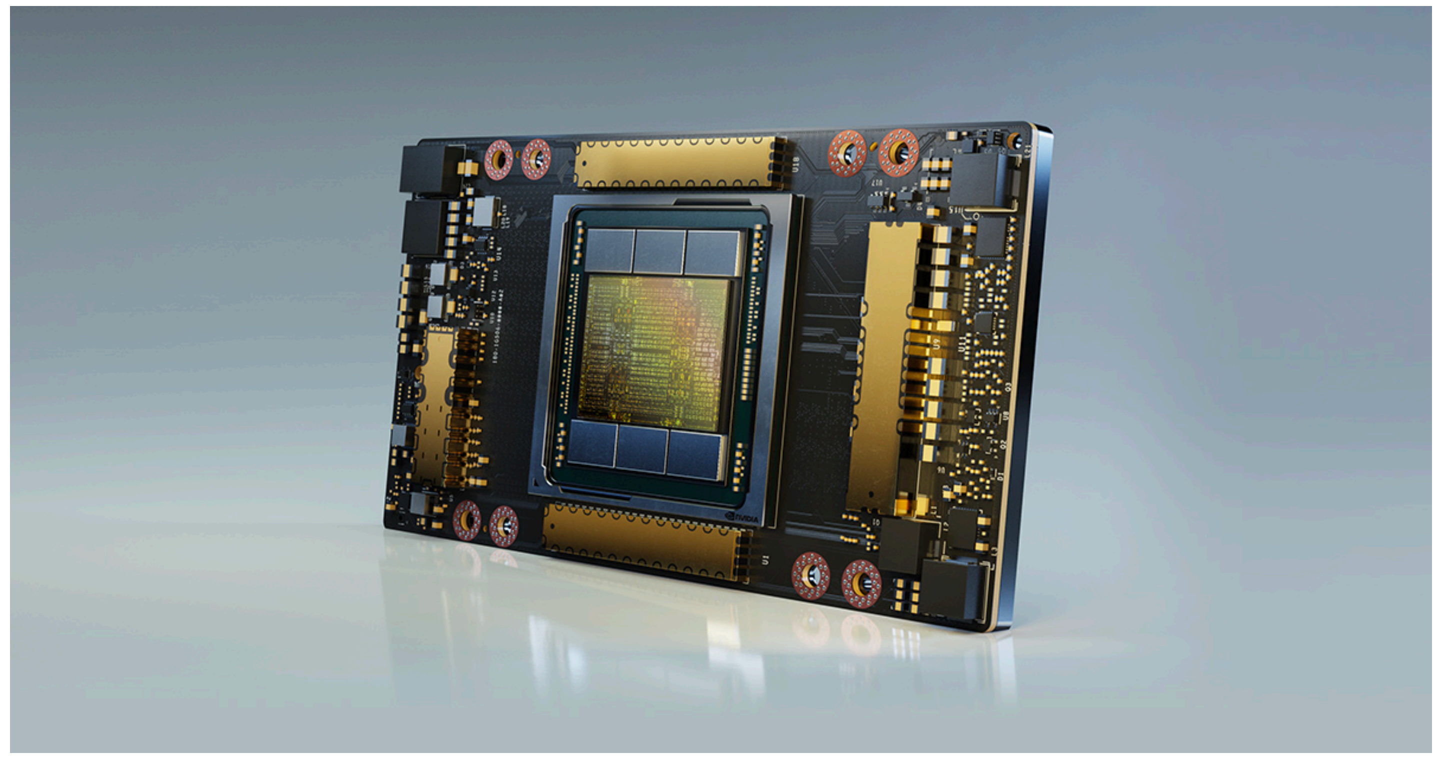


ML with GPUs

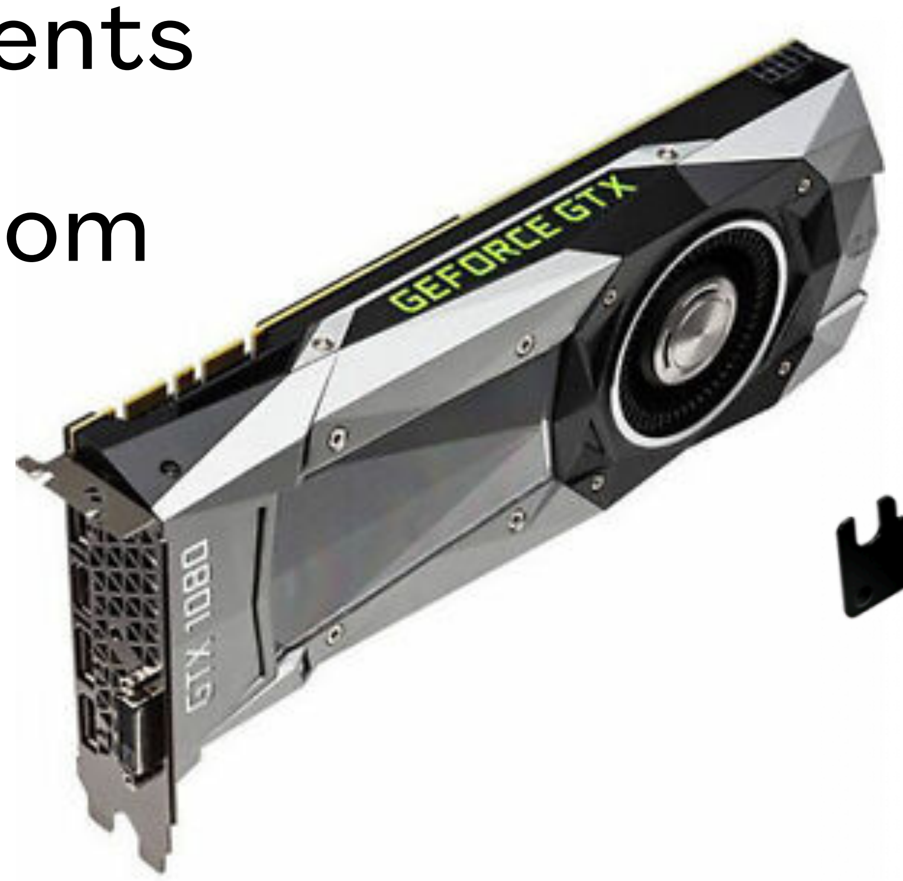
ML for TDAQ Overview

- ML algorithms highly parallelisable
  - NN forward pass just matrix-vector products and non-linear functions on vectors
- Can be accelerated with appropriate hardware:
  - CPUs with vector/SIMD units (e.g. AVX - get packages from Intel, for example)
  - GPU, FPGA, TPU (T = Tensor), IPU (I = Intelligence)
  - Need also good software and compilers to utilise hardware effectively
- ML is also big business, so lots of high performance solutions out there (incl open source)
- Often for Trigger and DAQ we can ‘train offline’, ‘predict online’
  - Different goals and hardware for each phase
  - May need to (re)optimize ML models for online performance

GPUs for ML

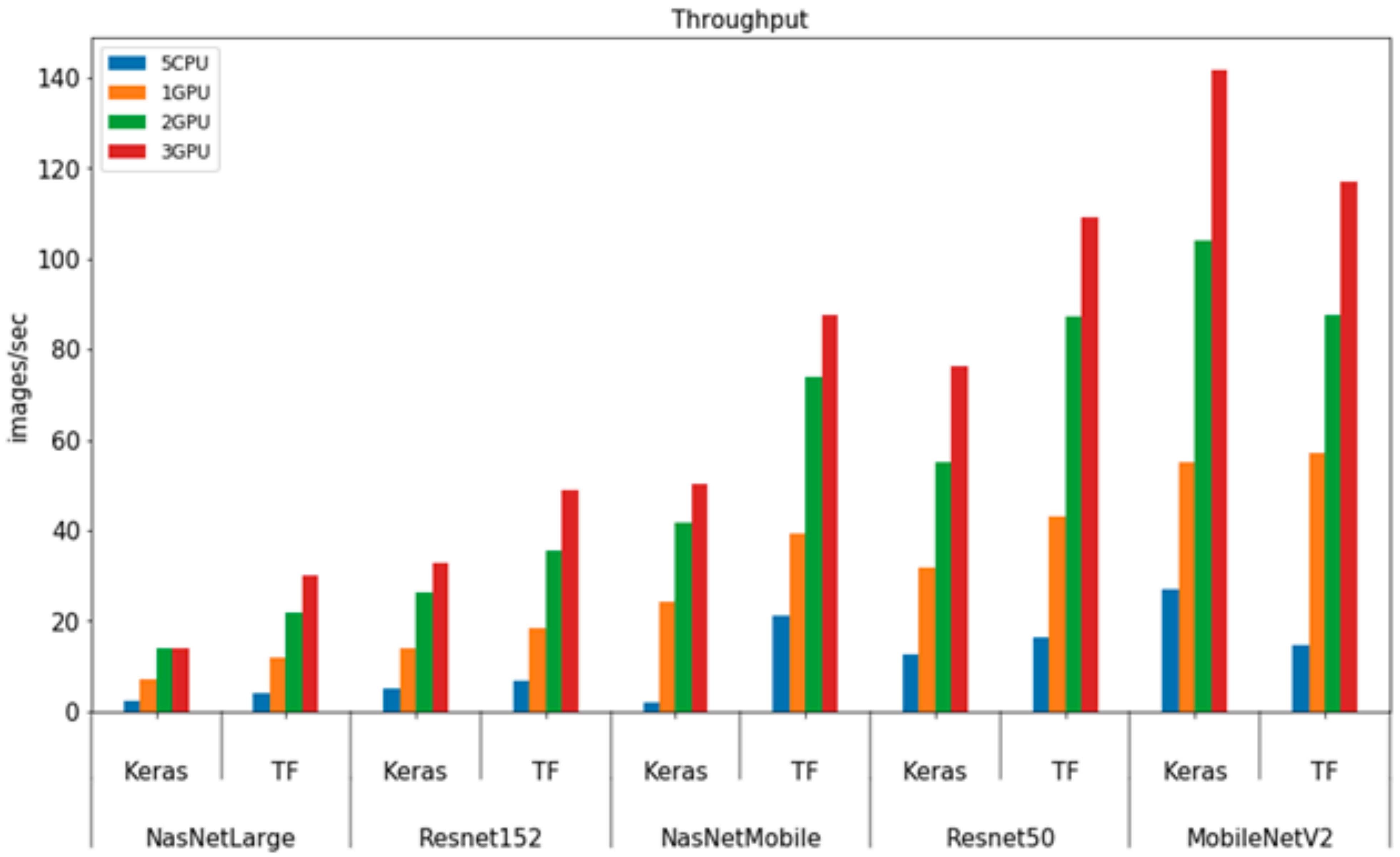


- GPUs are very powerful for machine learning
  - Many more parallel arithmetic ops than a CPU
  - Very high memory bandwidth
  - Training / predicting ML models on large datasets doesn’t involve much branching/control
  - Plus the GPU can be useful for other things
- Usually, using GPUs for ML, you don’t write CUDA code yourself but use a higher level framework like Tensorflow (or higher still with Keras, PyTorch)
  - Extremely easy to execute on a GPU with these environments
  - Exception might be when doing something extremely custom



GPUs for ML

- Biggest gains for GPUs are seen in training, but they also outcompute CPUs in inference
  - But remember you have to get the data to the device
- Here, running inference on K80 GPUs, measuring images / second (throughput)

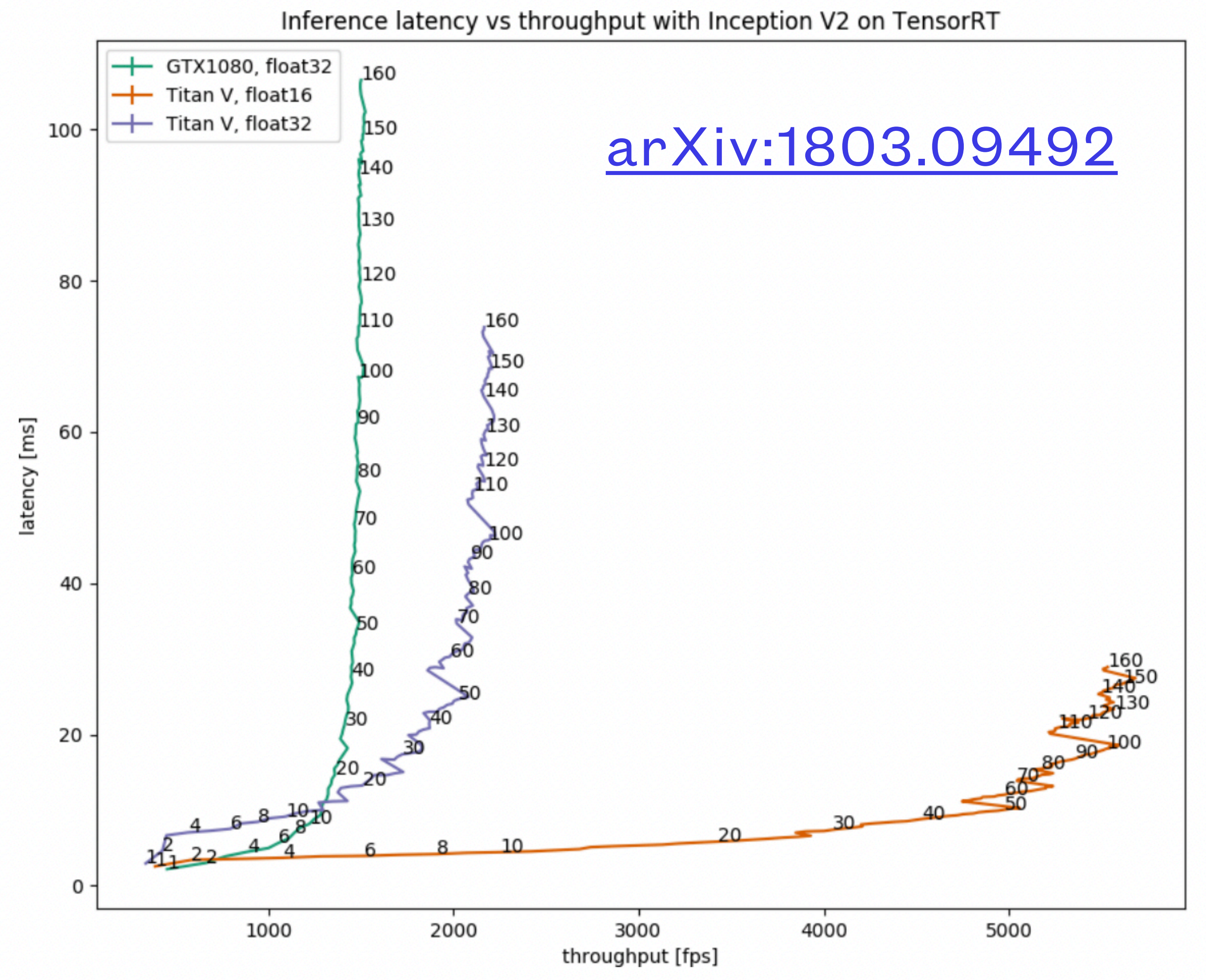


From Microsoft Azure

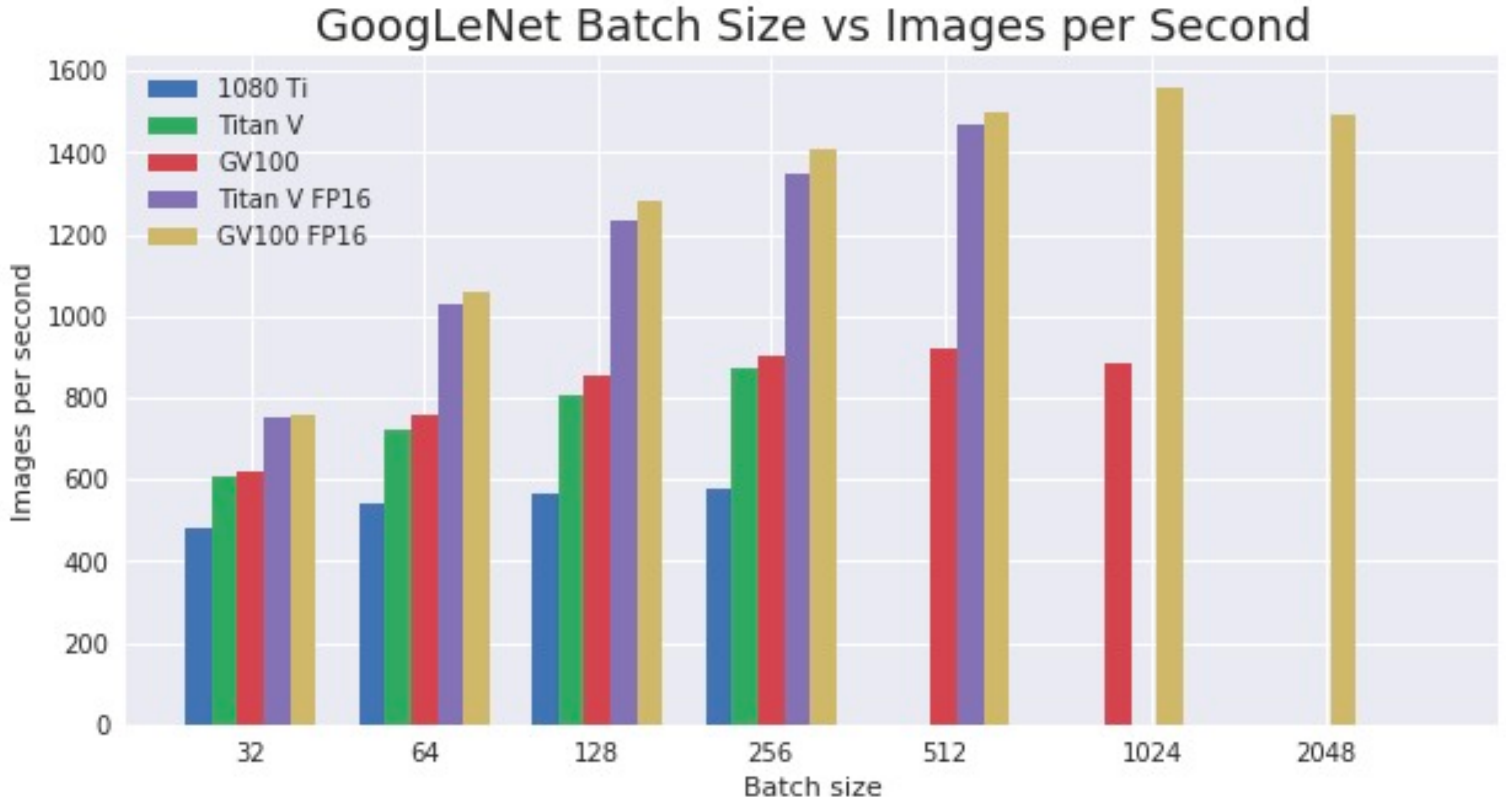
- mlperf.org has nice benchmarking of different hardware (not only GPUs) running on different models

GPUs for ML - batching

- “Batching” is a common technique for better hardware utilisation
  - Relevant both at training and inference time
- Send several data samples to the GPU in one batch to maximise use of memory bandwidth and compute
- Is the constraint latency or throughput?
  - If strictly latency: low batch size
  - If throughput: high batch size
  - Both: batch size where throughput saturates



arXiv:1803.09492

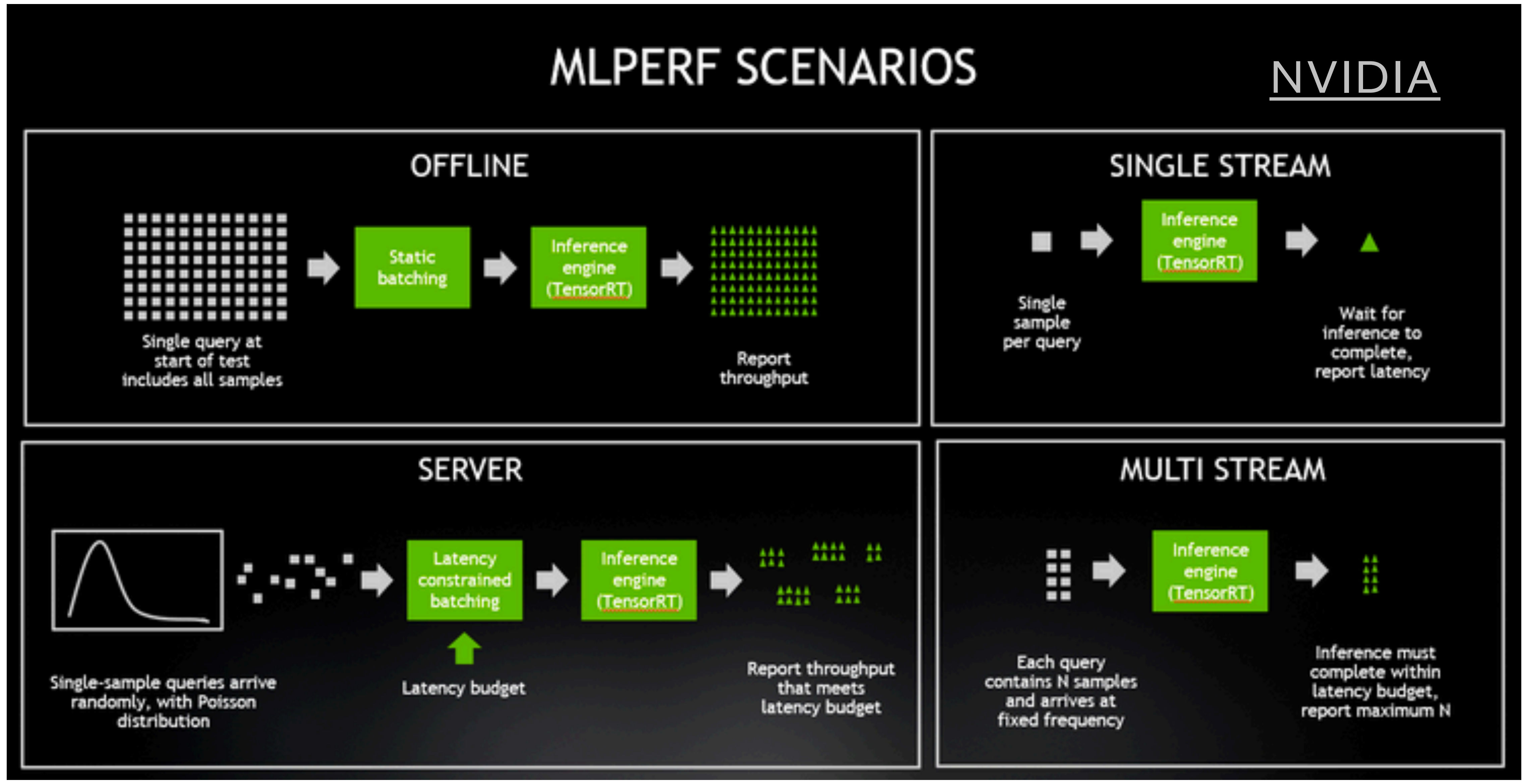


Puget Systems

- Plot: throughput vs latency at different batch sizes for Inception V2 (large computer vision CNN)
  - On different GPUs and different precisions

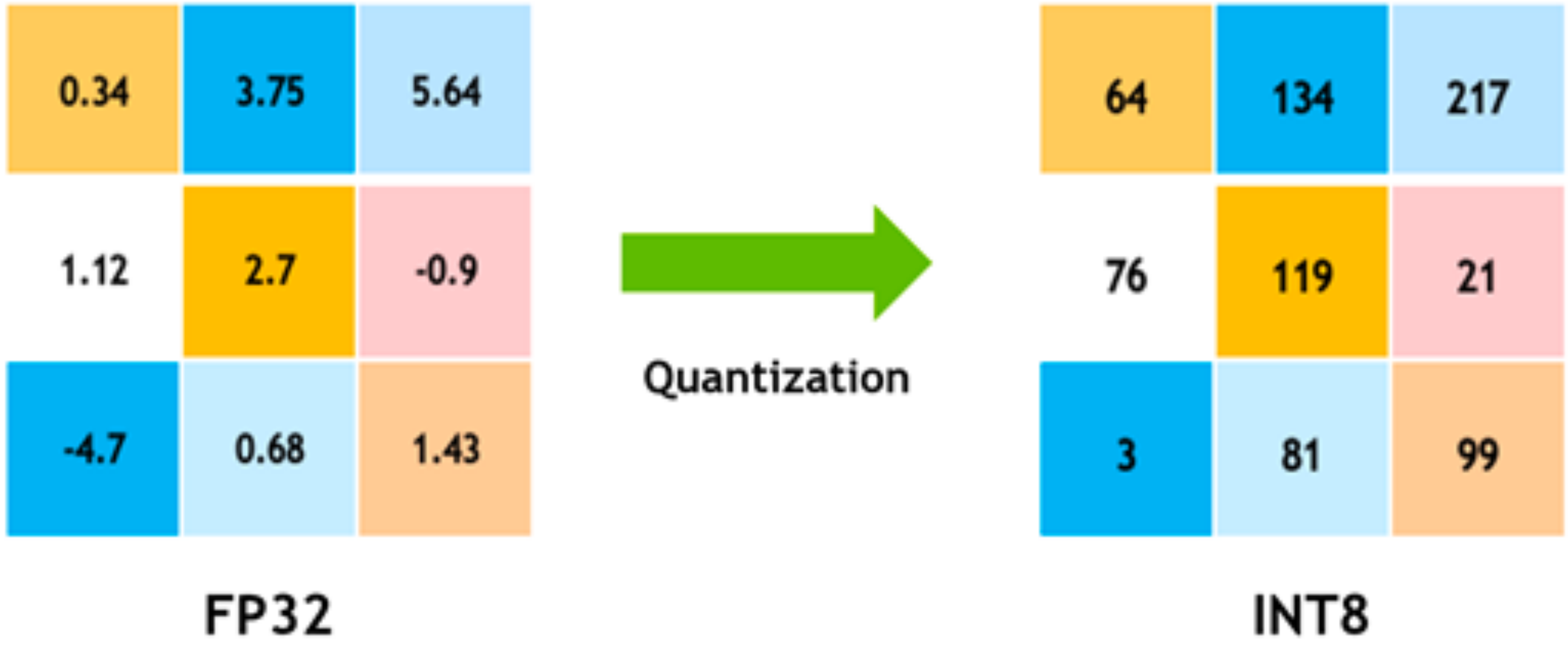
GPUs for ML - batching

- Whether or not you can profit from batching depends also on:
  - Is the main constraint on throughput or latency? (Or both?)
  - The data source: do data arrive at fixed intervals (bottom right image), or stochastically (bottom left)?
  - Can you afford to wait to accumulate several samples before sending them to the GPU?


MLPERF SCENARIOS
NVIDIA
OFFLINE
Static batching
Inference engine (TensorRT)
Single query at start of test includes all samples
Report throughput
SINGLE STREAM
Inference engine (TensorRT)
Single sample per query
Wait for inference to complete, report latency
SERVER
Latency constrained batching
Inference engine (TensorRT)
Single-sample queries arrive randomly, with Poisson distribution
Latency budget
Report throughput that meets latency budget
MULTI STREAM
Inference engine (TensorRT)
Each query contains N samples and arrives at fixed frequency
Inference must complete within latency budget, report maximum N

Quantization

- Many GPUs support Int8, float16, bfloat16 precision with many more OPS than float32
  - **Post Training Quantization (PTQ)** - train with FP32 then scale & round to lower precision
  - **Quantization Aware Training (QAT)** - train with lower precision
    - TensorRT (NVIDIA GPU),
    - TensorFlow Lite (Google),
    - torch.quantization (PyTorch)
- Choice of precision depends on target hardware and requirements



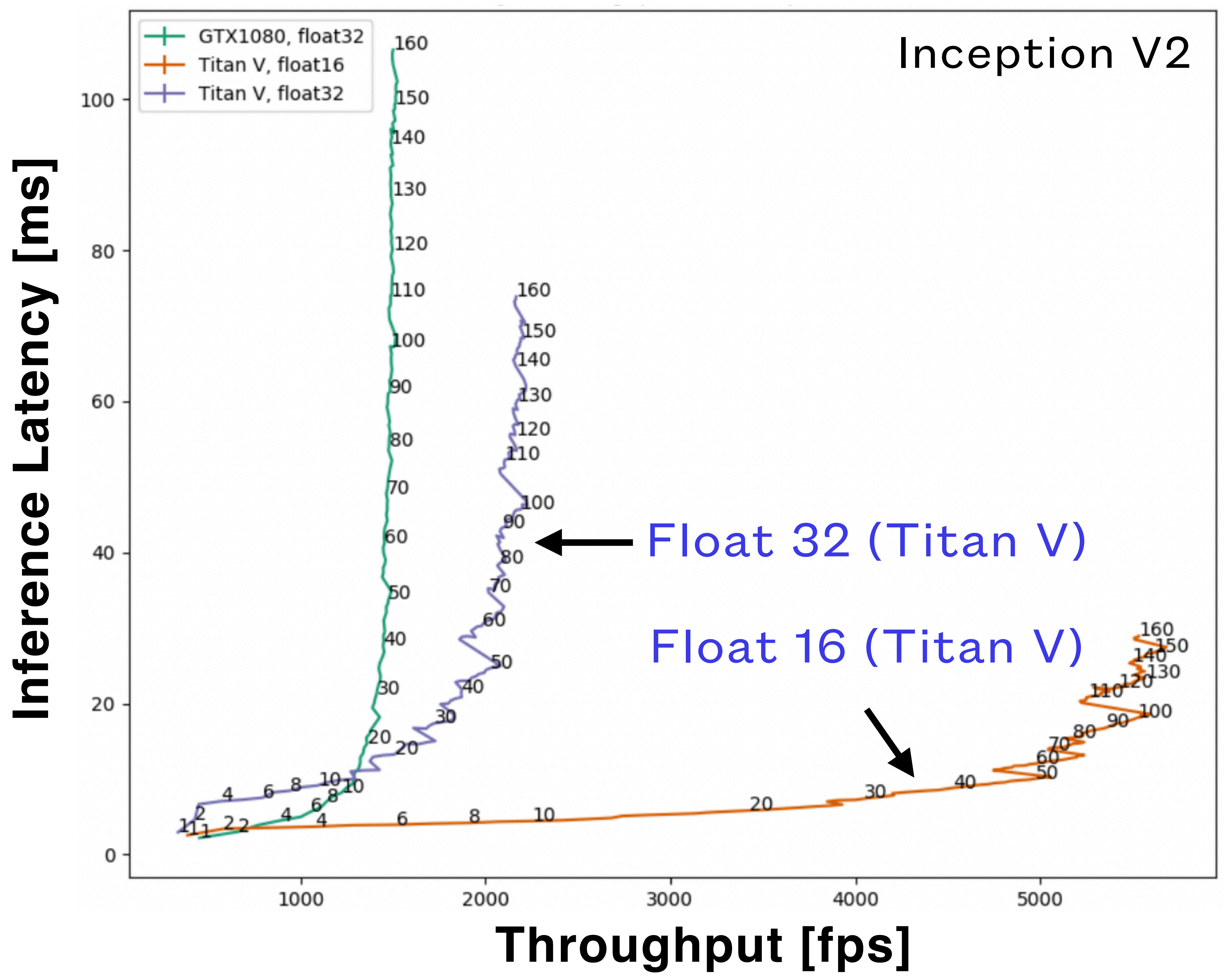
0.34	3.75	5.64
1.12	2.7	-0.9
-4.7	0.68	1.43

FP32

Quantization

64	134	217
76	119	21
3	81	99

INT8

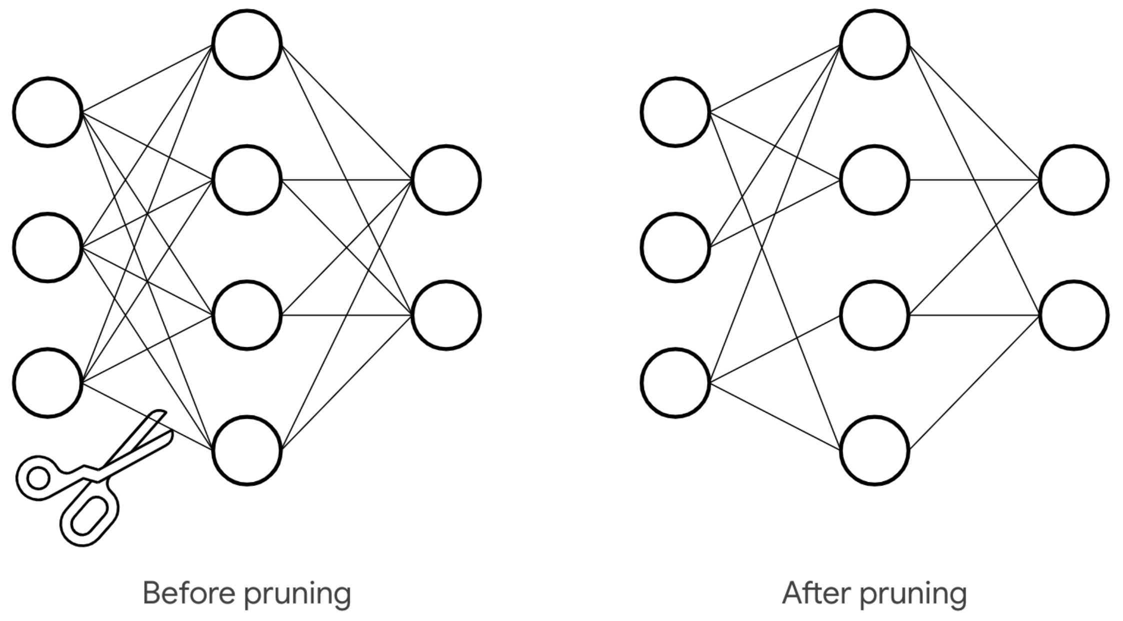


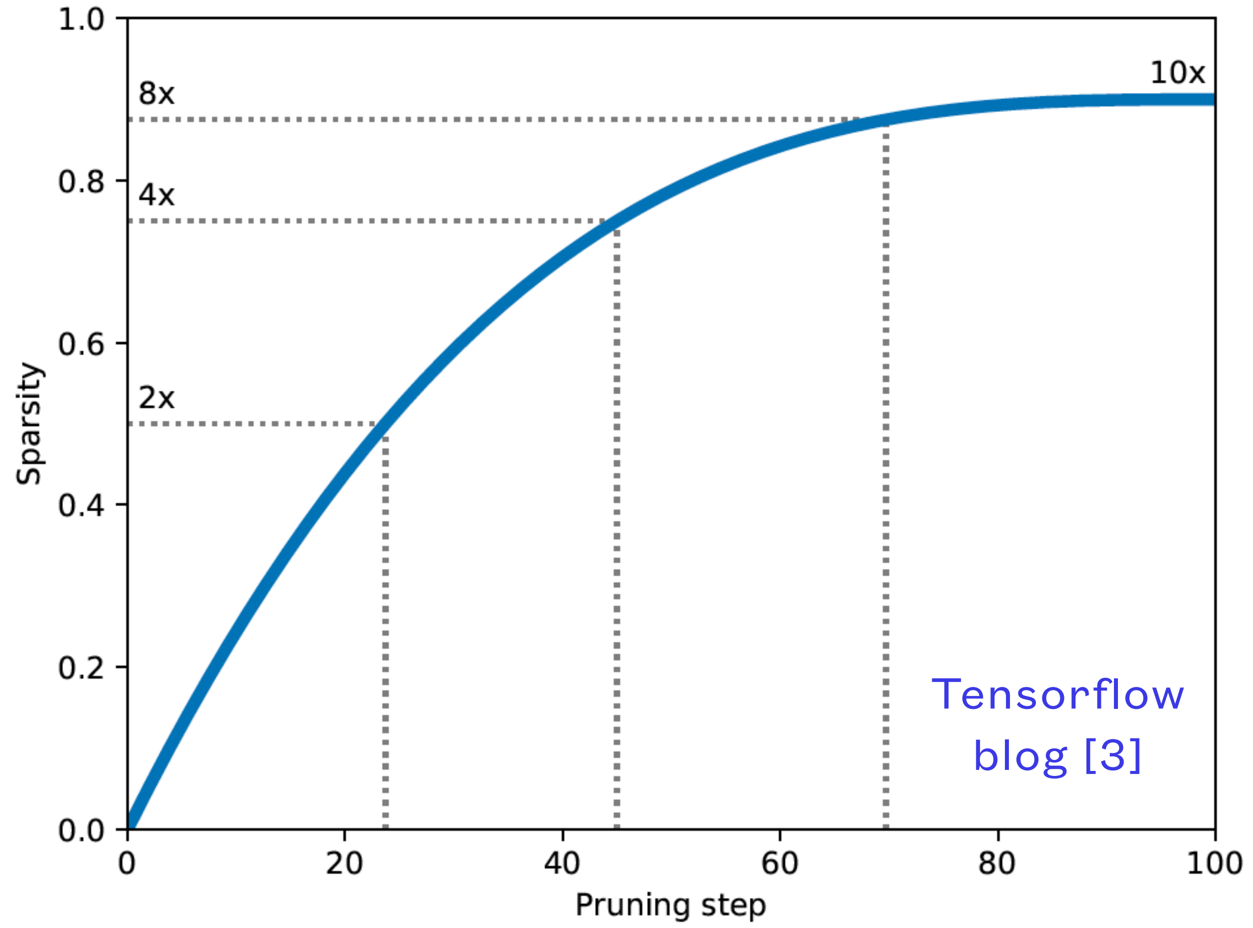
Float 32 (Titan V)

Float 16 (Titan V)

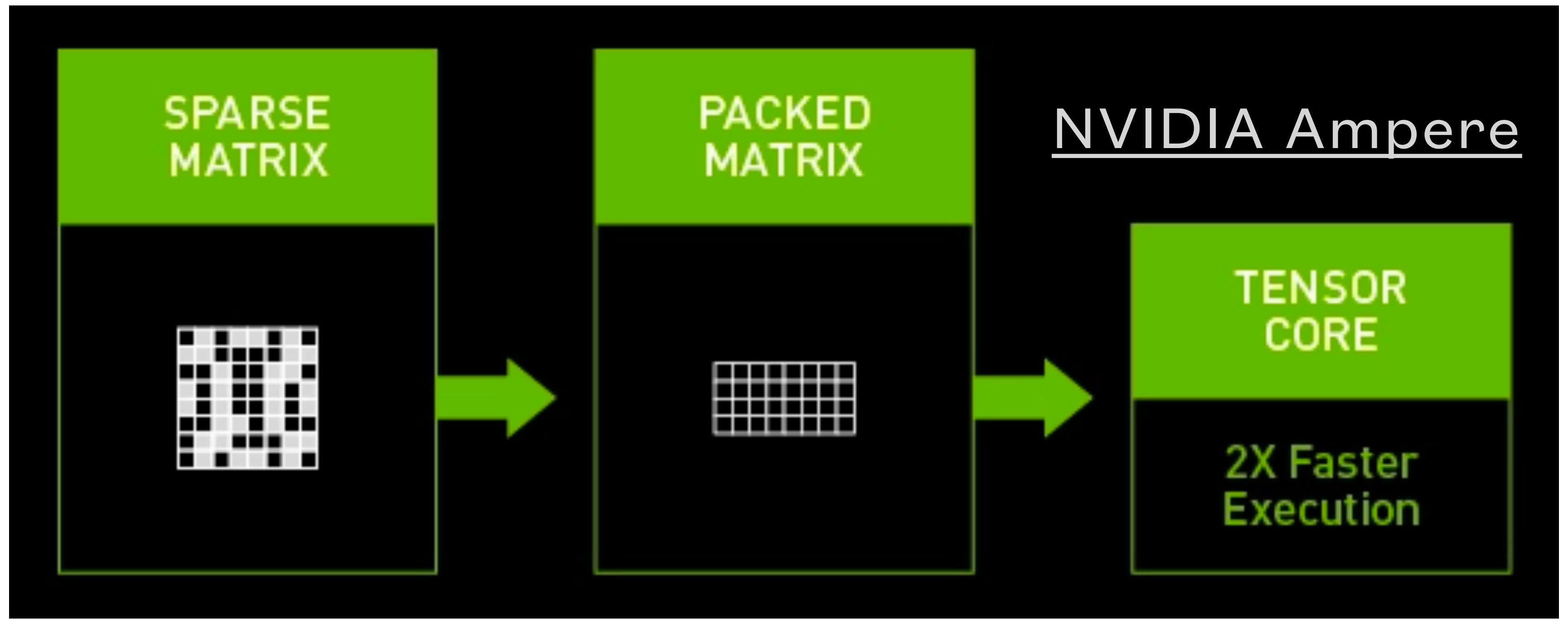
Pruning / Sparsity

- A Neural Network often contains many redundant connections
- Pruning = remove some connections from final model
- Can reduce the model size (memory footprint)
- Some processors can accelerate sparse networks
  - Basically - don’t do the *multiply by 0* computations
- Different methods:
  - Regularisation (penalise low value weights, then make them 0)
  - Target sparsity, e.g. sparsity ramp up with TFMOT
  - Structured pruning - remove continuous blocks of weights;
  - Filter pruning - entire filters of CNN
- Applies also to BDTs (λ, α in xgboost)
- Can be coupled with Quantisation Aware Training





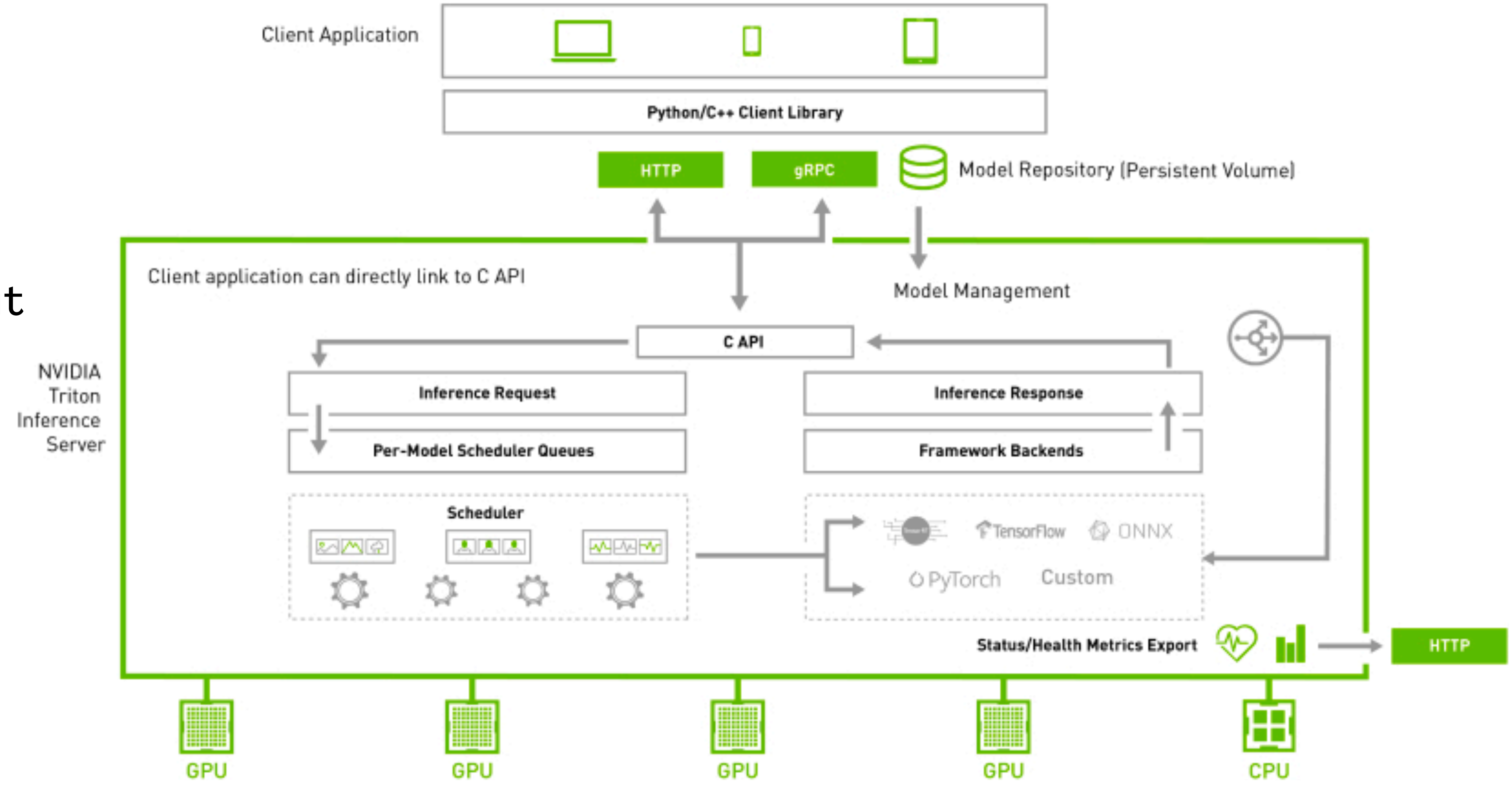
Tensorflow blog [3]



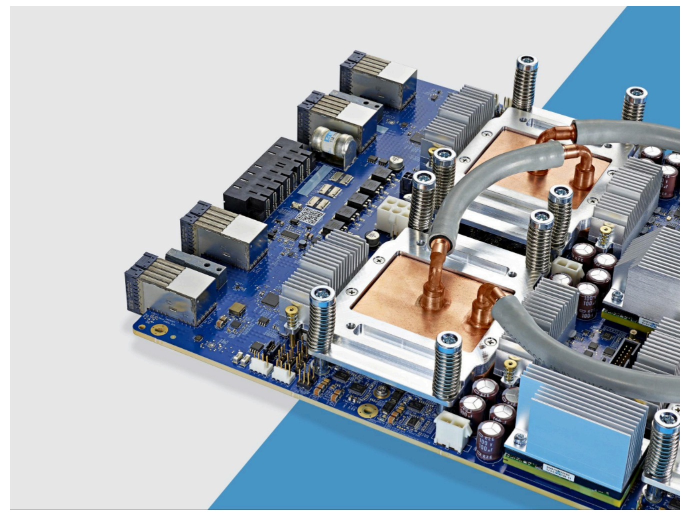
NVIDIA Ampere

Deployment: Triton inference server

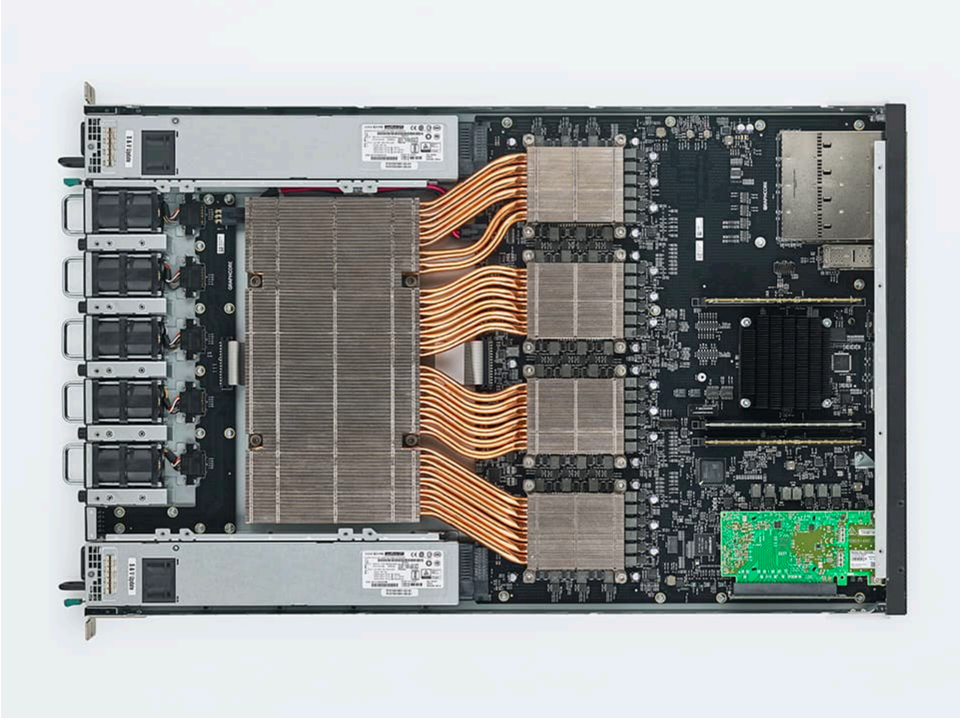
- Many GPUs / Many clients
- NVIDIA, open source
- Handles dynamic batching depending on requests to optimize latency/throughput performance
- Could be used for varying
  - event rate,
  - number of inferences / event

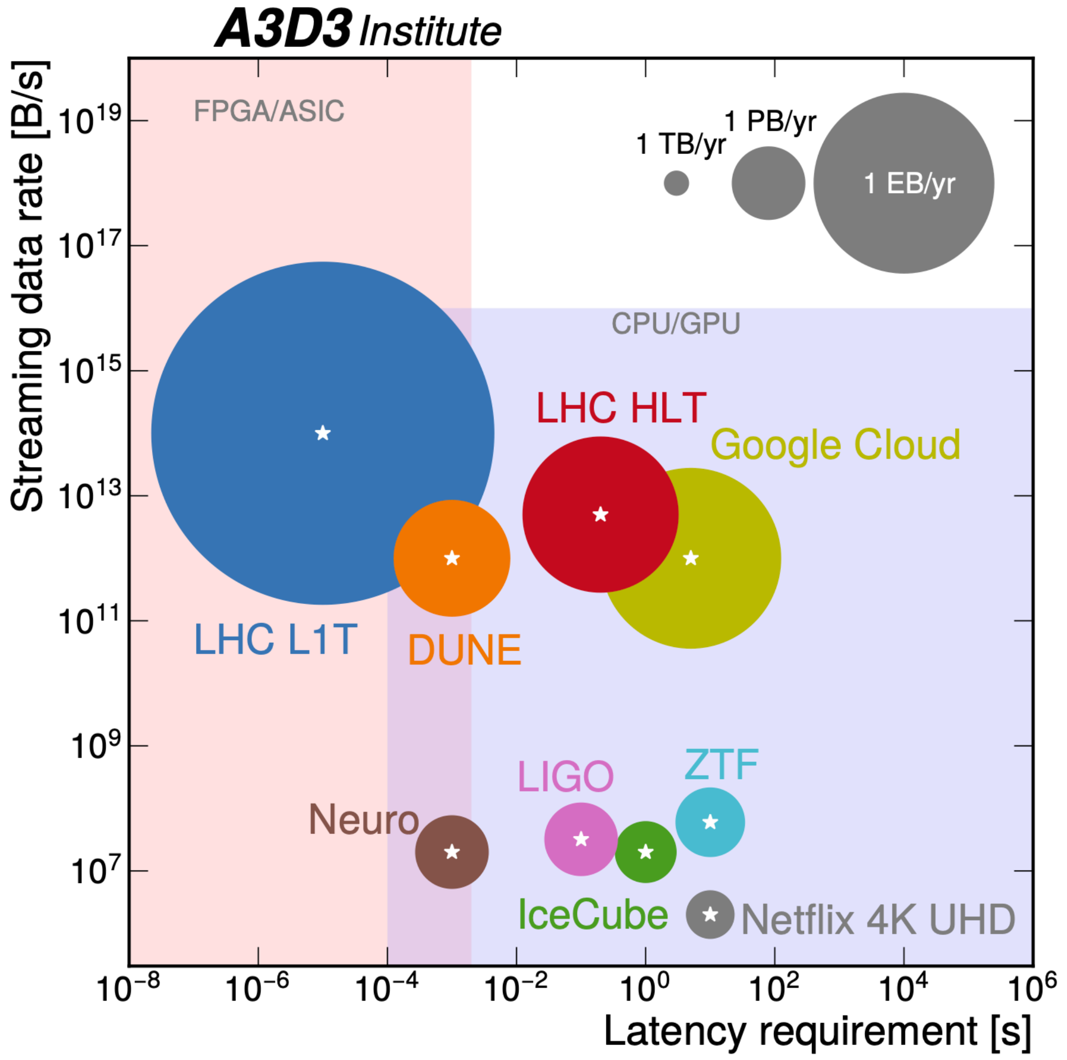

Client Application
Python/C++ Client Library
HTTP
gRPC
Model Repository (Persistent Volume)
NVIDIA Triton Inference Server
Client application can directly link to C API
Model Management
C API
Inference Request
Per-Model Scheduler Queues
Inference Response
Framework Backends
Scheduler
TensorFlow
ONNX
PyTorch
Custom
Status/Health Metrics Export
HTTP
GPU
GPU
GPU
GPU
CPU

ML Specific Processors

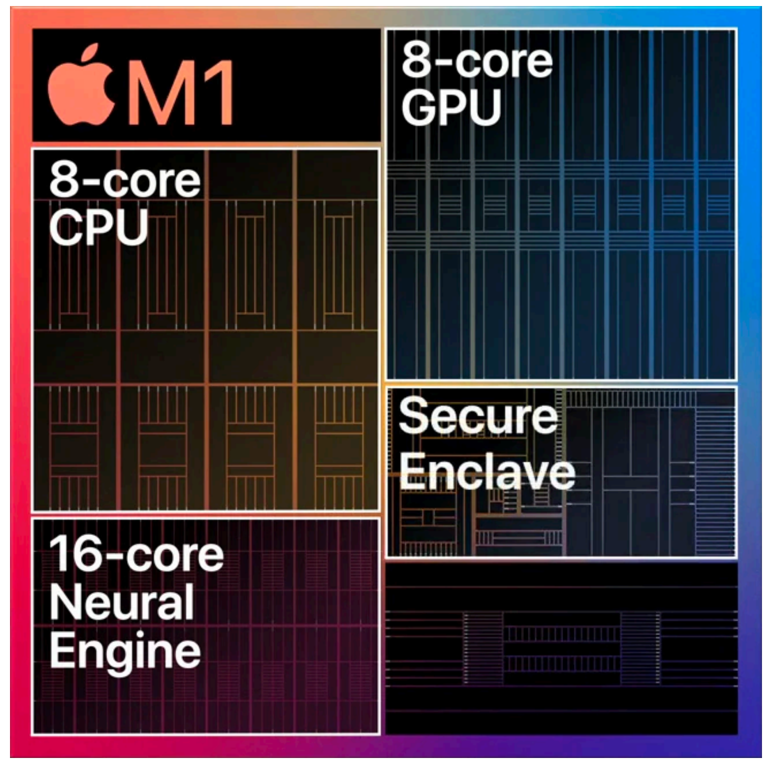


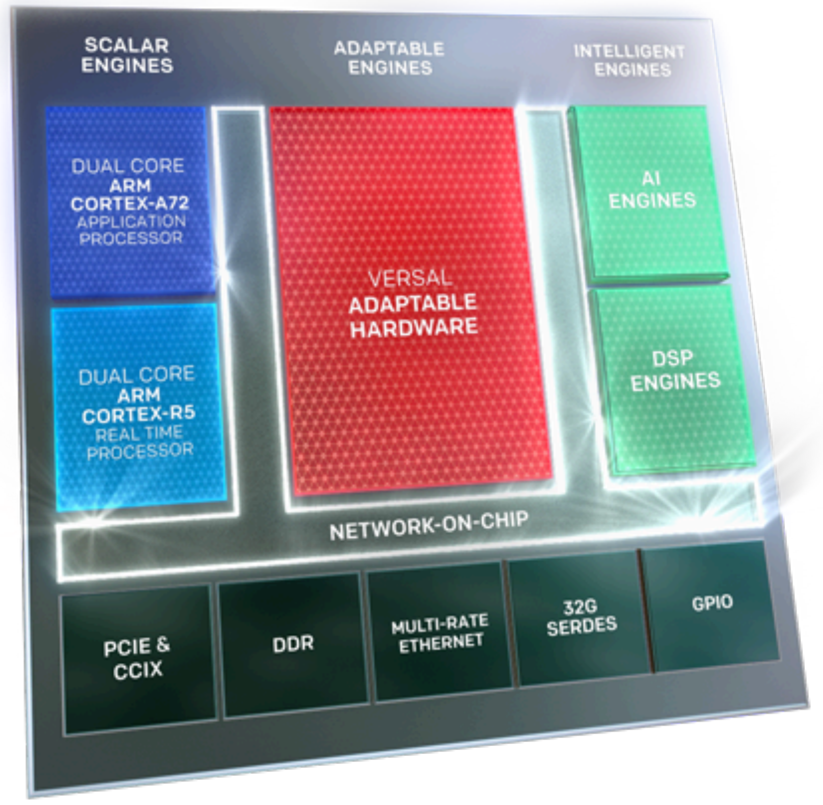
- There are some processors out there specifically designed for Machine Learning / AI
- e.g. Tensor Processing Unit (TPU) from Google, Intelligence Processing Unit (IPU) from Graphcore
- Devices aiming at low power embedded
  - Internet of Things, Smartphones
- Xilinx Versal ACAP for FPGAs with embedded Vector units, Vector/NN units in CPUs
- Many different things out there, each targeting a specific optimisation:
  - Best overall throughput
  - Lowest latency
  - Lowest power / smallest footprint
- Choose appropriate device for your task



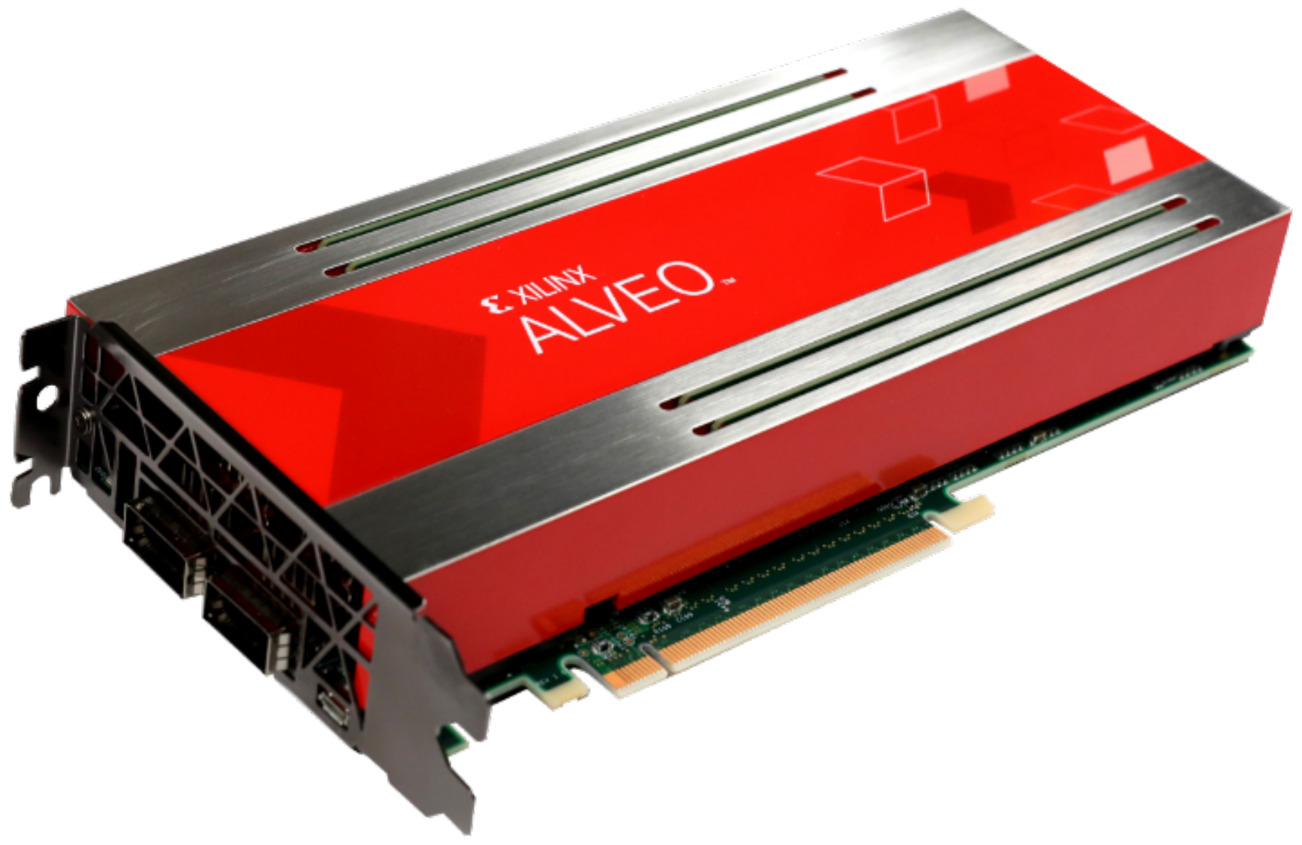


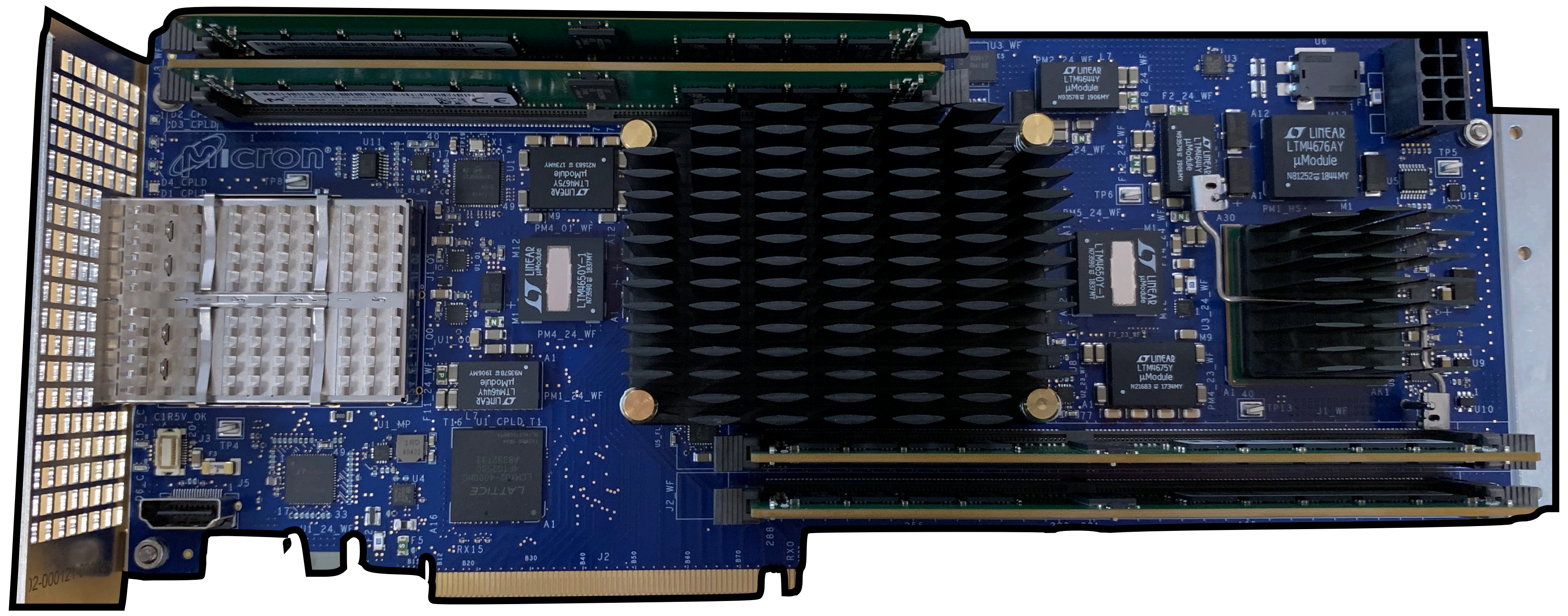
A3D3





S7t
VectorPath Accelerator Card
BittWare a molex company
Powered by Achronix Data Acceleration

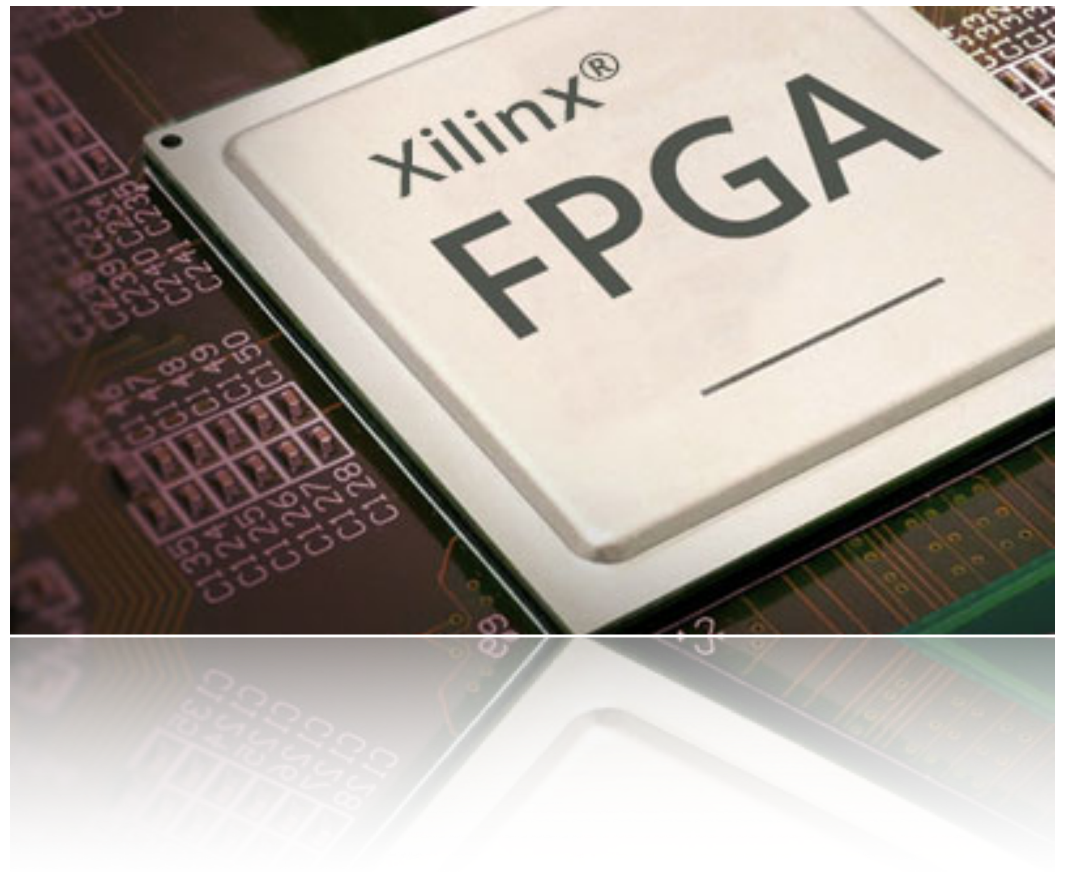




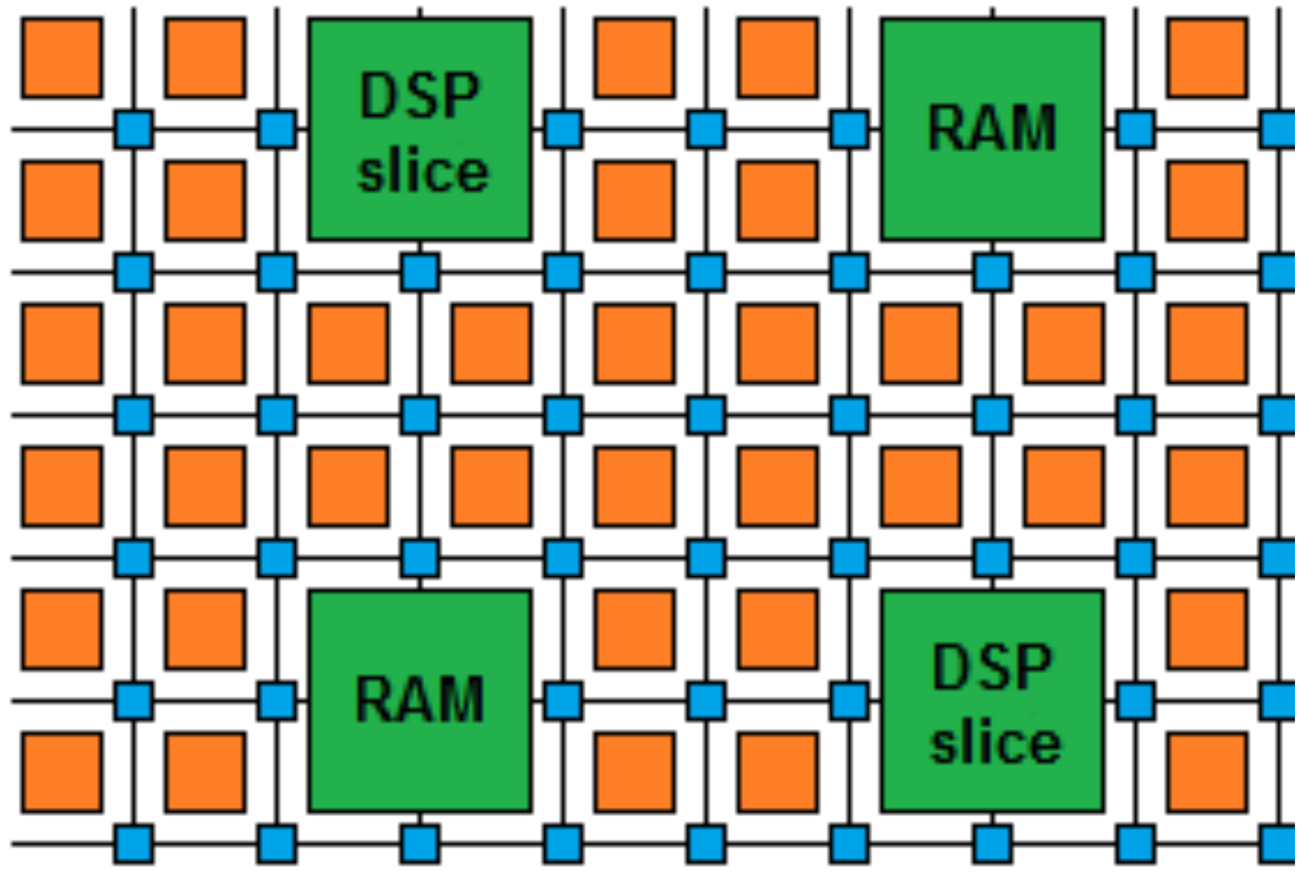
ML inference with FPGAs

What are FPGAs?

- Field Programmable Gate Arrays = reprogrammable integrated circuits
- Contain many different building blocks (*resources*) which are connected together as desired
- Extremely parallel processors
- *Computing in space as well as time*
- Utilised by most low level HEP triggers



FPGA diagram



LUTs - generic logic

DSPs - for multiplication

BRAM - for local, high-throughput storage

FPGAs in CMS

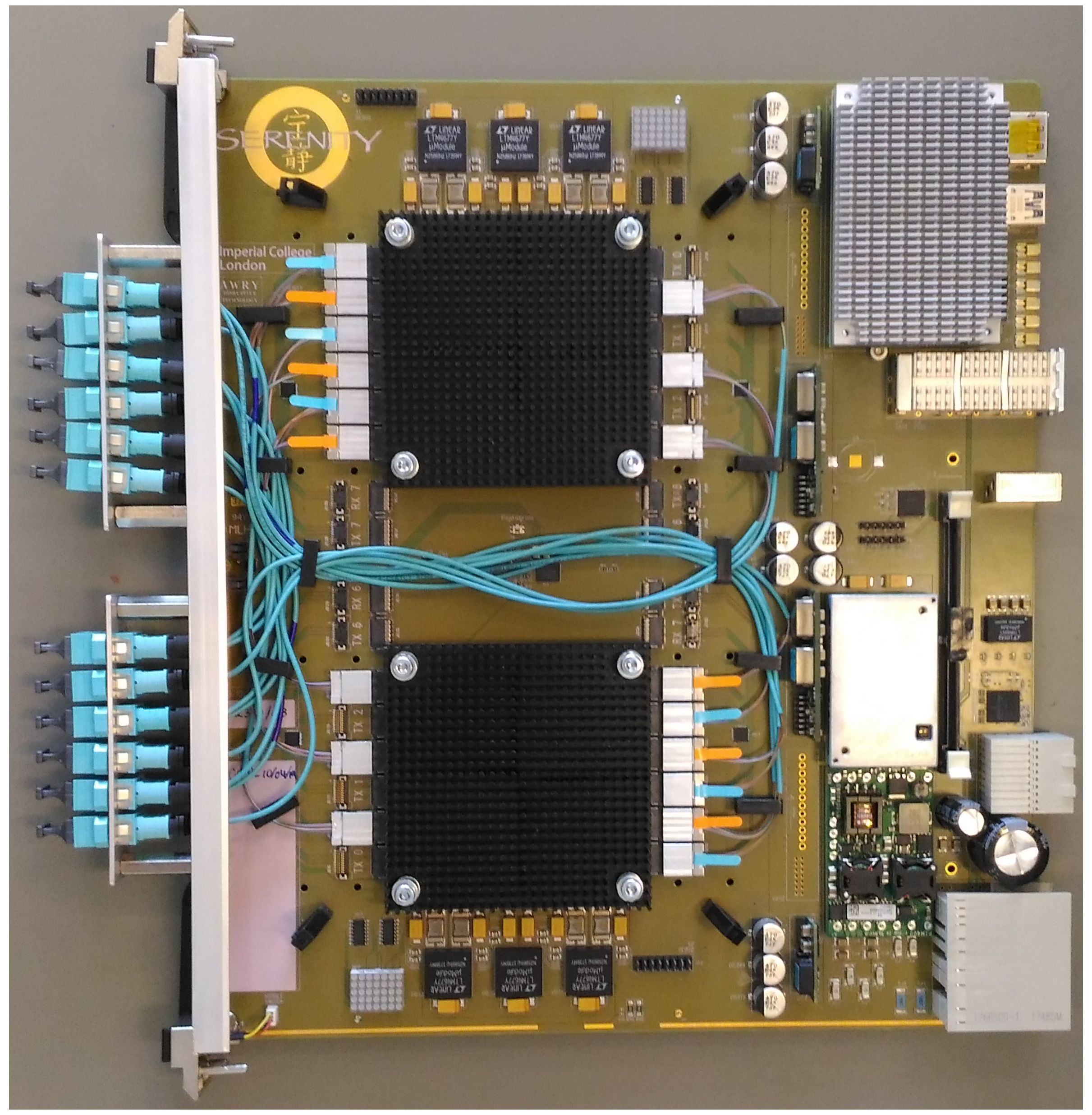
Possible uses of FPGAs:

Stream processor / real-time : fixed latency L1 trigger

- Primarily use custom hardware
- Very high IO bandwidth / optical inputs

CMS/UK: MP7

CMS/US: CTP7

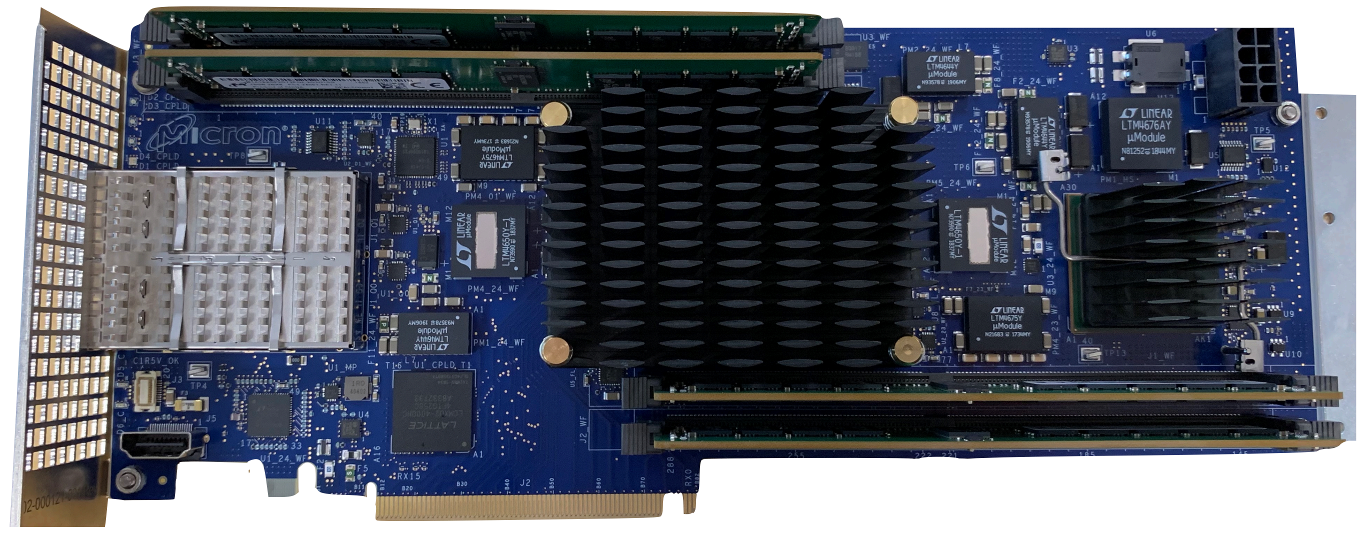


CMS/UK: Serenity

Form factor: VME / MTCA / ATCA

Accelerators

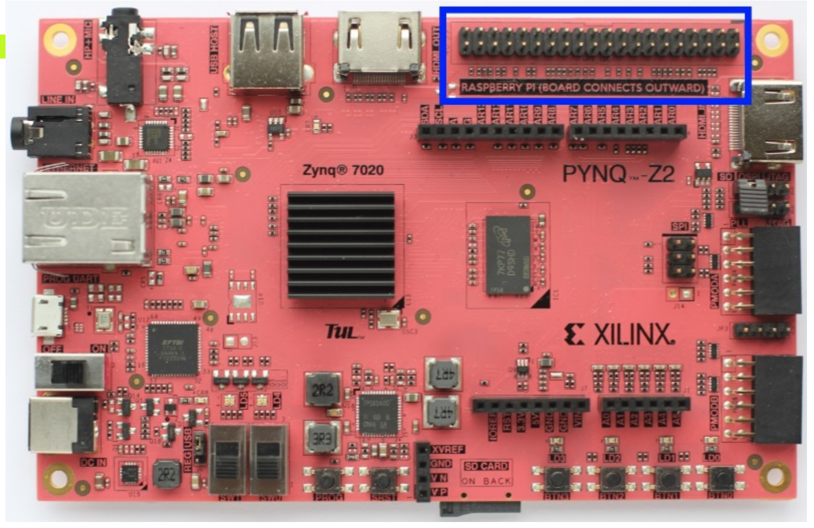
- Usually commercial hardware
- Mostly PCIe form-factor



Micron: SB852

Bitware: VectorPath S7t-VG6

Xilinx: Alveo U250



TUL: PYNQ

Embedded / low power

Why FPGA accelerators?

FPGA = high parallelism (like GPUs) + reprogrammable custom architectures

-> Typically (but not always) lower latency & improved power efficiency w.r.t GPU solution

Key benefit is bandwidth into device / in-network processing

Workloads:

- ML inference
- Video transcoding
- Database search / analytics
- Compression / decompression
- Encryption
- Computational storage / ‘smart’ SSD
- Low latency FINTECH

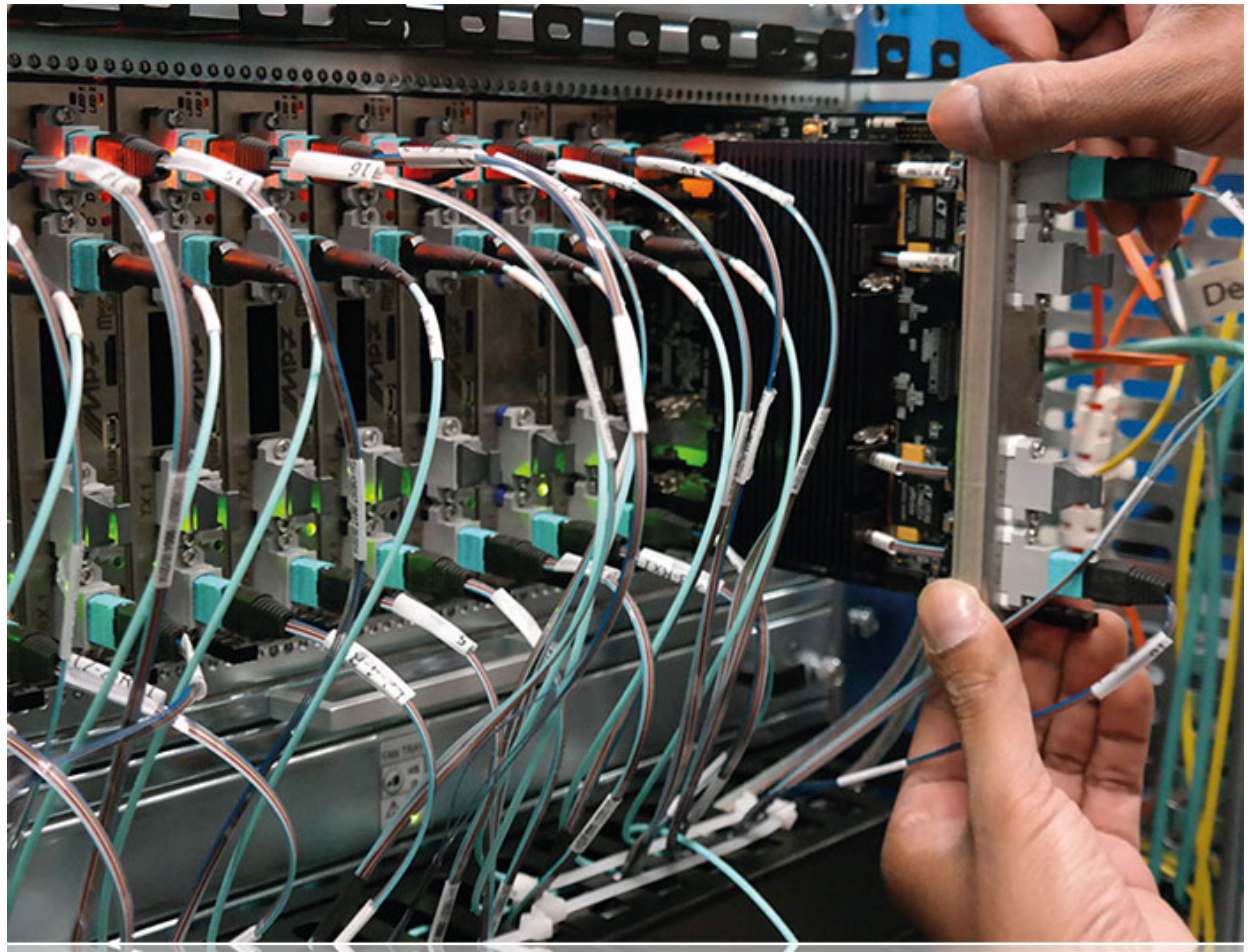
Requirements for successful acceleration:

- Sequential or stream-based processing with pre-determined data dependencies and deterministic execution
- Limited random-accesses
- Host-accelerator transfer overhead << processing time (large blocks of data preferred)

Note: will be covering primarily Xilinx devices. Intel devices also exist.

Algorithms running on FPGAS

- LHC Run 2 (2015-2018)
  - Clustering
  - Pattern Recognition
  - Energy Sums
  - Zero Suppression
  - Boosted Decision Trees

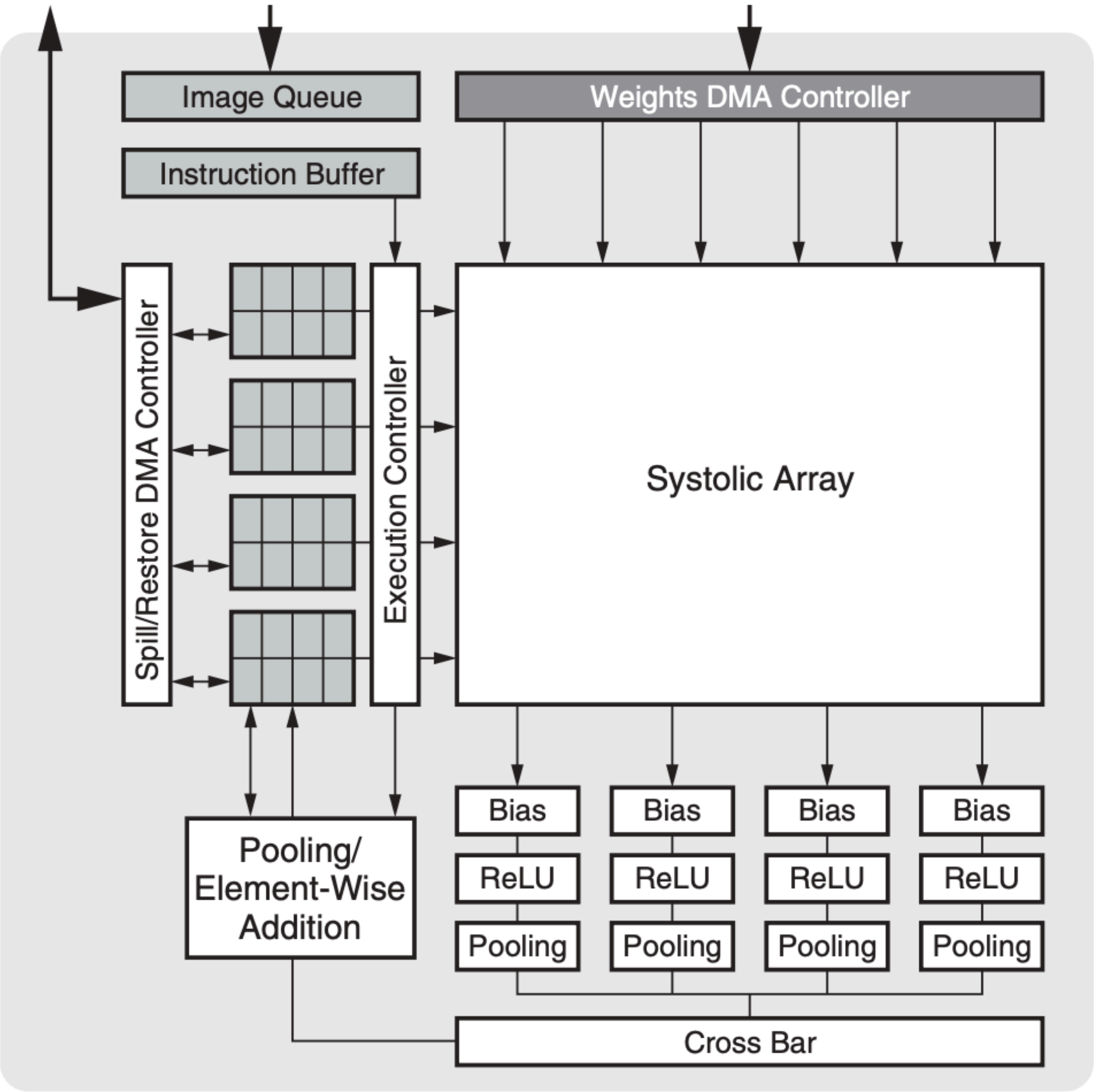


- LHC Run 3 (2022-2025)
  - Multi Layer Perceptrons: DNNs
  - Kalman Filters

- LHC Run 4 dev. 2029-
  - Hough Transform
  - Convolutional Neural Networks
  - ?????

FPGAs for ML

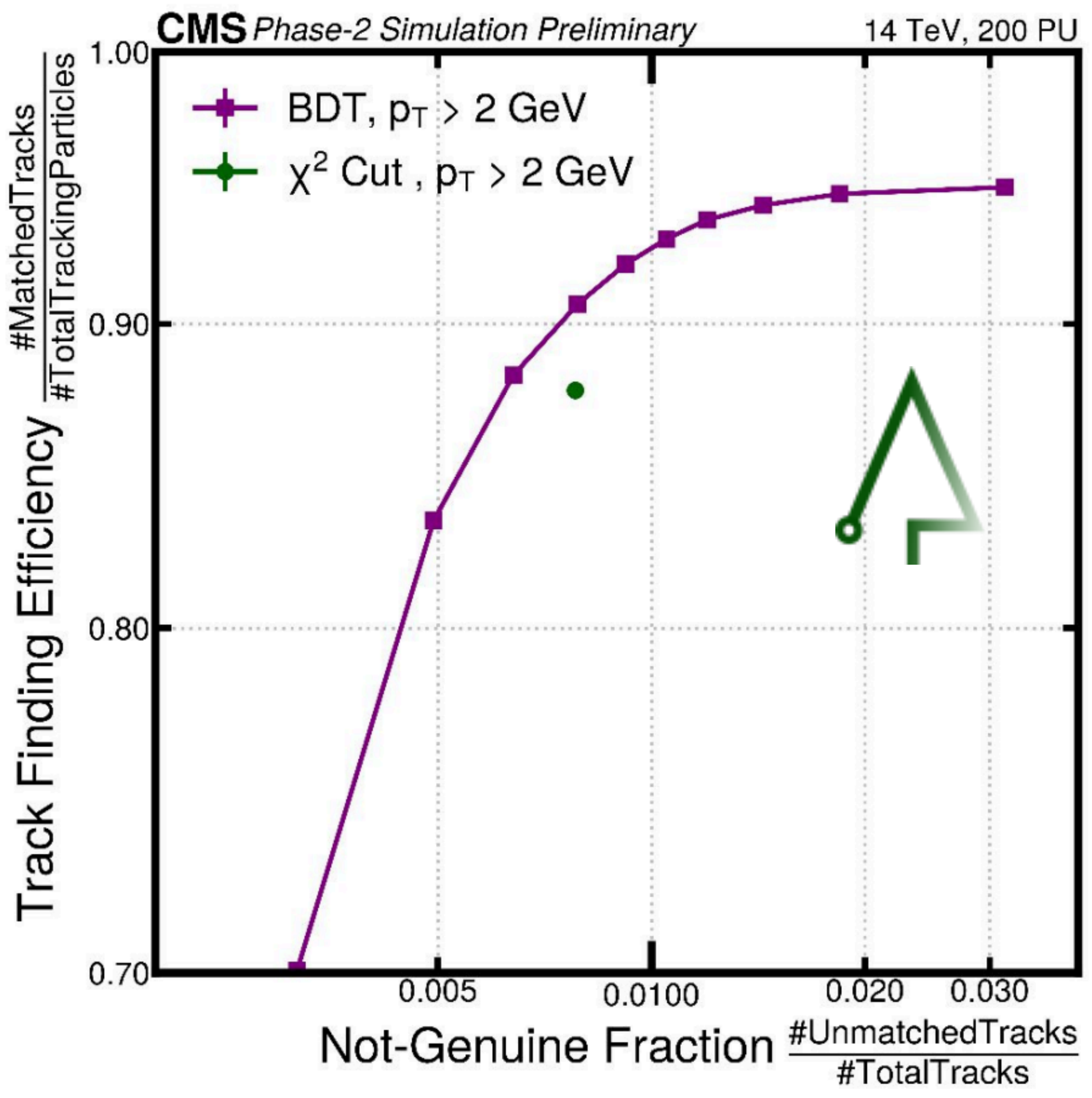
- FPGAs are highly suited to ML tasks - massive parallelism, high memory bandwidth
- Several big providers using FPGAs for ML in their datacentres
  - e.g. Microsoft with Bing and Azure, FPGA availability on Amazon Web Services
- Main way to execute ML on FPGAs:
  - Vendor libraries with fixed silicon designs and an instruction set - Deep Learning Processor Unit (DPU) for Xilinx Vitis AI, Deep Learning Acceleration (DLA) Suite for Intel
- Can outperform GPUs mostly at maintaining high-throughput with low latency (< 2ms)
- Able to achieve best 'performance per Watt'
- Can benefit from in-network processing with FPGA's high speed connectivity



Xilinx: xDNN

ML in L1T FPGAs

- Tools like hls4ml (more later) and conifer bring ML into FPGAs with sub-microsecond latency
- Example: identifying fake tracks from CMS Level 1 Track Finder (Phase 2 Upgrade)
- Fake tracks are identified in simulation as those not associated to a simulated particle
  - Often from combinatorics (200 pileup scenario), they harm trigger performance later
- A BDT with 60 trees and depth of 3 finds fakes better than simple cuts
- conifer library maps BDT onto FPGA logic
  - In this case 33 ns latency and < 1% resources (VU9P)
- Many algorithms in development for Phase 2
  - Improving object reconstruction (as here)
  - Improving event selection of difficult signatures



High Level Synthesis

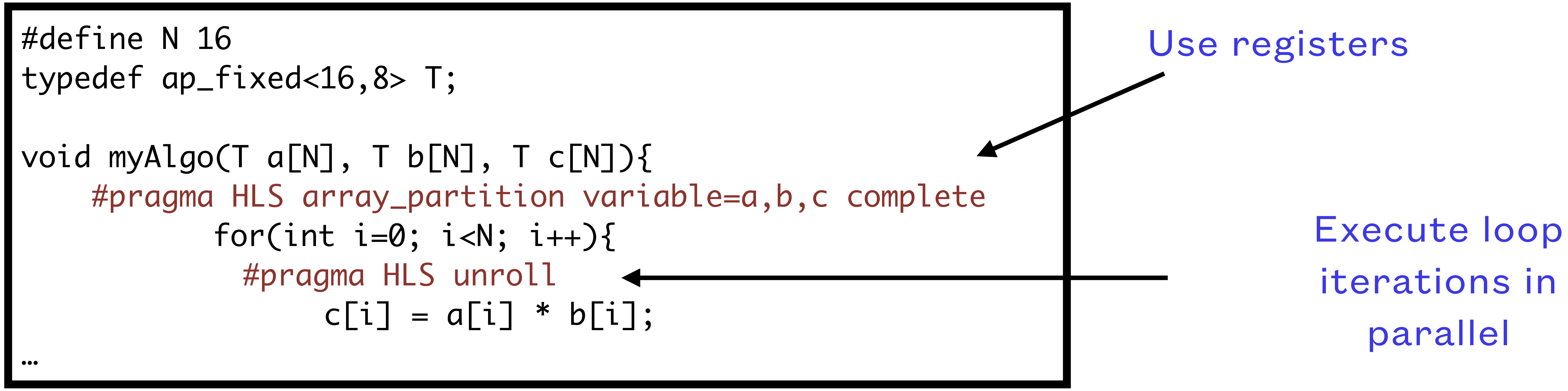
- FPGA programming has challenges
  - Requires a lot of expert engineering knowledge, long development cycles
- New design tools from the FPGA companies - ‘High Level Synthesis’ - make it a lot easier
  - Enabling more physicists to contribute
  - Enabling experienced FPGA designers to complete designs faster
- In HEP this is enabling us to bring more of the offline algorithms into the Level 1 Trigger
  - Kalman Filter for charged particle track reconstruction
  - Machine Learning…

```
entity add is
port(
  clk : in  std_logic;
  a   : in  signed(31 downto 0);
  b   : in  signed(31 downto 0);
  c   : out signed(31 downto 0)
)
end add;

architecture rtl of add is
  if rising_edge(clk) then
    c <= a + b;
  end if;
end rtl;
```

```
int add (int a, int b){
  return a + b;
}
```

High Level Synthesis

- With a Hardware Description Language (HDL), you write a description of a circuit
- With HLS, you write a description of your algorithm
  - The compiler decides the circuit
- Controlling how the compiler maps your algorithm to a circuit requires careful code design
  - And use of #pragma directives to guide the compiler

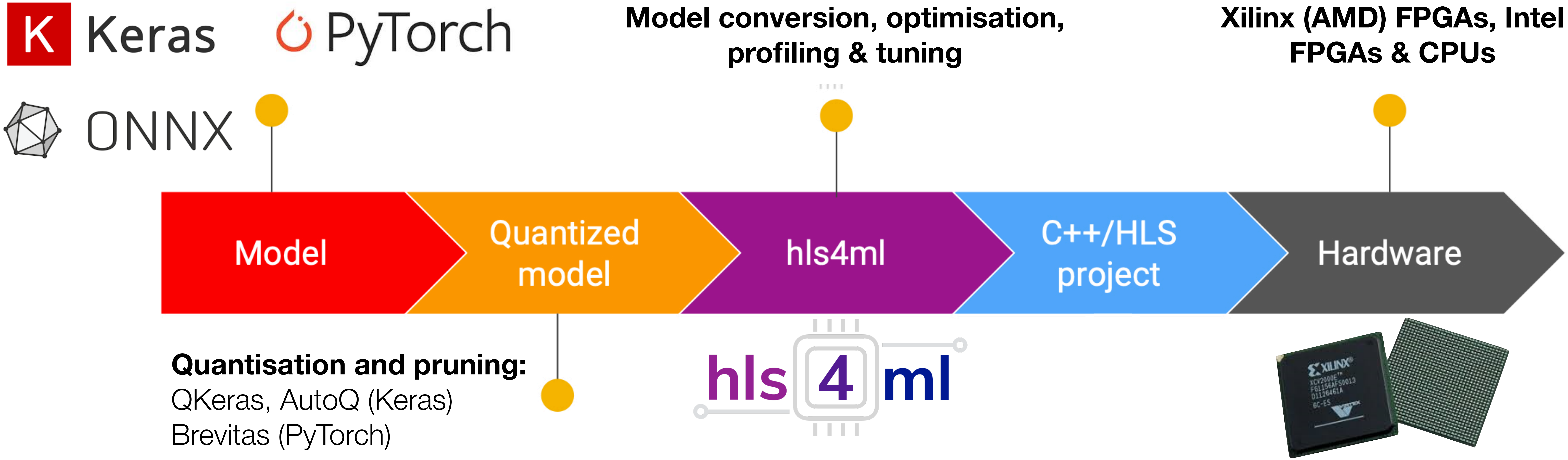


```
#define N 16
typedef ap_fixed<16,8> T;

void myAlgo(T a[N], T b[N], T c[N]){
    #pragma HLS array_partition variable=a,b,c complete
           for(int i=0; i<N; i++){
            #pragma HLS unroll
                 c[i] = a[i] * b[i];
…
```

Use registers

Execute loop iterations in parallel

- Open-source Python API & command line tool that translates trained NNs to synthesizable FPGA firmware [1, 5]


Keras
PyTorch
ONNX
Model conversion, optimisation, profiling & tuning
Xilinx (AMD) FPGAs, Intel FPGAs & CPUs
Model
Quantized model
hls4ml
C++/HLS project
Hardware
hls4ml
XILINX

Quantisation and pruning:
QKeras, AutoQ (Keras)
Brevitas (PyTorch)

- Implementations of common ingredients - layer types, activation functions
- Novel ingredients for fast, efficient inference - binary/ternary NNs, network optimisations

hls4ml

- Excels at very low latency applications
- Weights stored on-chip -> very fast access times, limited capacity
- Can tune latency vs resource utilisation with per-layer ‘reuse factor’
- Capability to utilise extremely heterogeneous quantisation techniques
- Relies on Xilinx HLS (tool that produces FPGA code from C++), blackbox that can produce non-optimal results
- Requires a bit more knowledge of FPGA design than some other solutions, but still accessible to non-Verilog/VHDL experts
- Work on support for new backends & off-chip weights ongoing
- Ideal for L1-trigger applications: expected to be widely used for CMS Phase II trigger

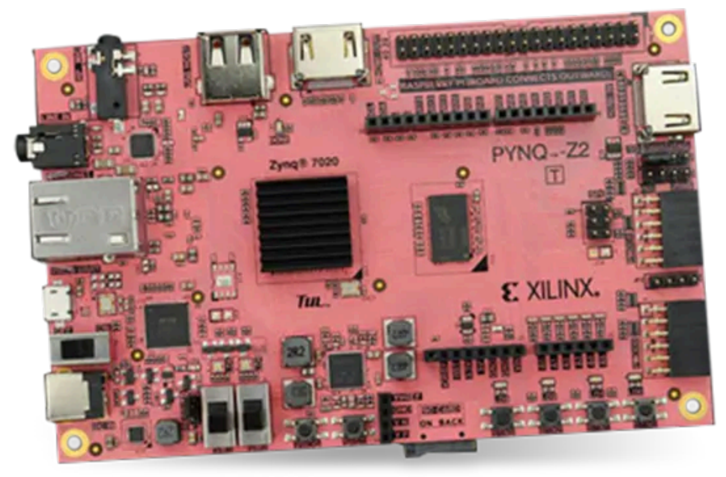
hls4ml

- Step 1: `pip install hls4ml`
- hls4ml is Python based, has Python API to:
  - convert NNs
  - write HLS projects
  - run emulation (execute the ap_fixed C++)
  - run synthesis (Vivado HLS)
  - Make accelerator bitfiles for some cards
  - There is also a command line tool
- Lots of user configuration is possible
  - Change data types (bitwidths) heterogeneously
  - Turn performance handles for each layer -
    - ReuseFactor, Strategy, parallel/streaming IO

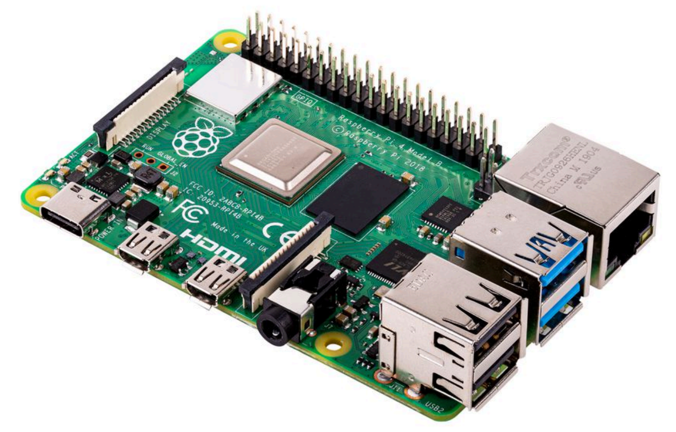
```
FROM HLS4ML IMPORT ...
IMPORT TENSORFLOW AS TF

# TRAIN OR LOAD A MODEL
MODEL = ... # E.G. TF.KERAS.MODELS.LOAD_MODEL(...)

# MAKE A CONFIG TEMPLATE
CFG = CONFIG_FROM_KERAS_MODEL(MODEL,
GRANULARITY='NAME')

# TUNE THE CONFIG - EACH LAYER INDEPENDENT
CFG['LAYERNAME']['LAYER2']['REUSEFACTOR'] = 4

# DO THE CONVERSION
HMODEL = CONVERT_FROM_KERAS_MODEL(MODEL, CFG)

# WRITE AND COMPILE THE HLS
HMODEL.COMPILE()

# RUN BIT ACCURATE EMULATION
Y_TF = MODEL.PREDICT(X)
Y_HLS = HMODEL.PREDICT(X)

# DO SOME VALIDATION
NP.TESTING.ASSERT_ALLCLOSE(Y_TF, Y_HLS)

# RUN HLS SYNTHESIS
HMODEL.BUILD()
```

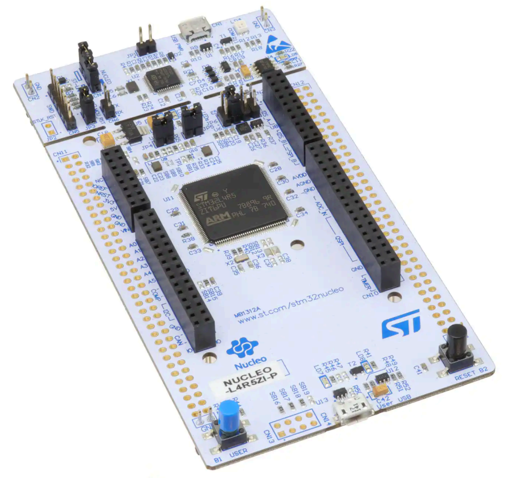
MLPerf™ Tiny Inference Benchmark

- MLCommons recently added 'Tiny' category to MLPerf benchmark (link)
- hls4ml submission targeted pynq-z2
- Fully on-chip hls4ml implementation is efficient for low power inference



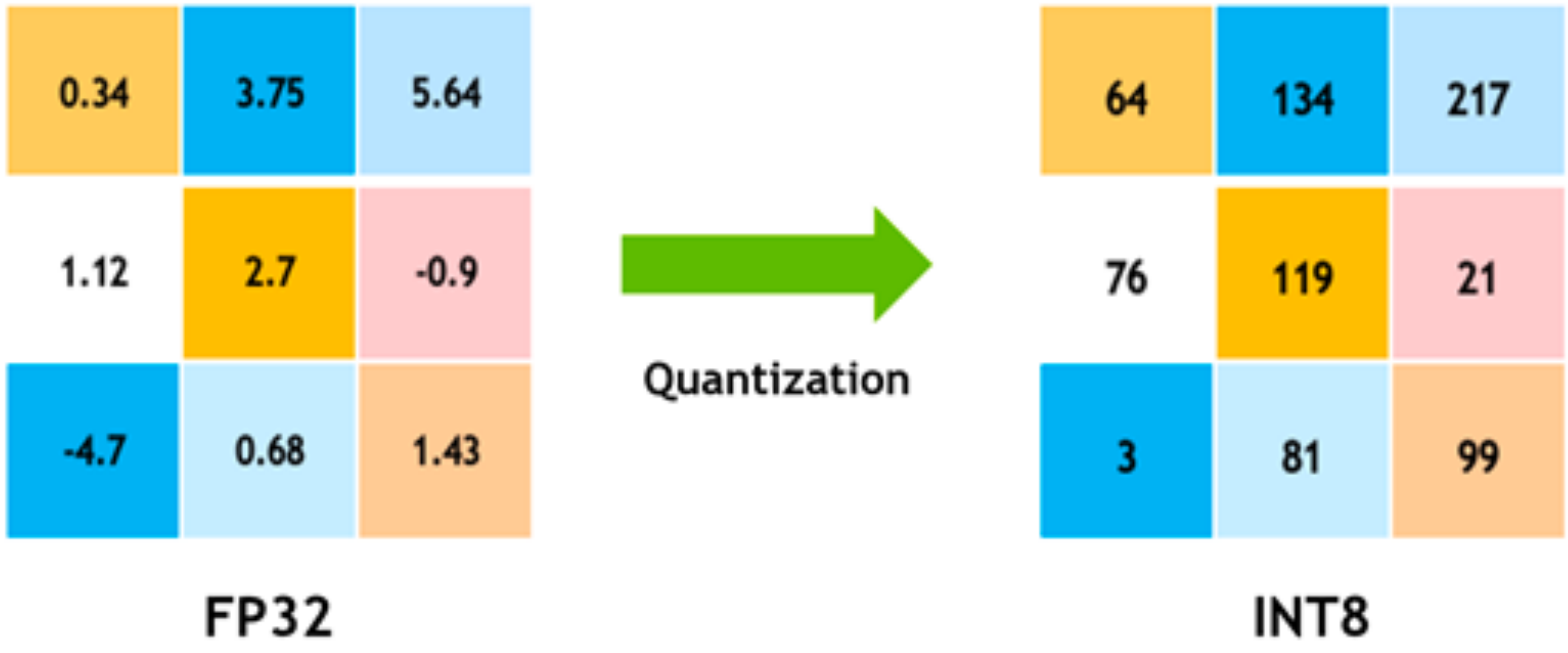
Benchmark		CIFAR-10			ToyADMOS		
Team	Device	Accuracy	Latency (ms)	Power (W)*	AUC	Latency (ms)	Power (W)*
hls4ml	Pynq-z2	77%	7.9	~ 1.5	0.82	0.096	~ 1.5
Latent AI	Raspberry Pi 4	85%	1.07	~ 4 - 5	0.85	0.17	~ 4 - 5
Harvard	Nucleo-L4R5ZI	85%	704		0.85	10.4	
Peng Cheng Lab	PCL Scepu02	85%	1239.16		0.85	13.65	





Quantization

- Using regular TensorFlow Keras or PyTorch, you train with floating point
  - We like to avoid floating point in FPGAs - much more resources & latency than fixed point
  - You can do post-training quantisation (PTQ) - represent the float values with some fixed point



0.34	3.75	5.64
1.12	2.7	-0.9
-4.7	0.68	1.43

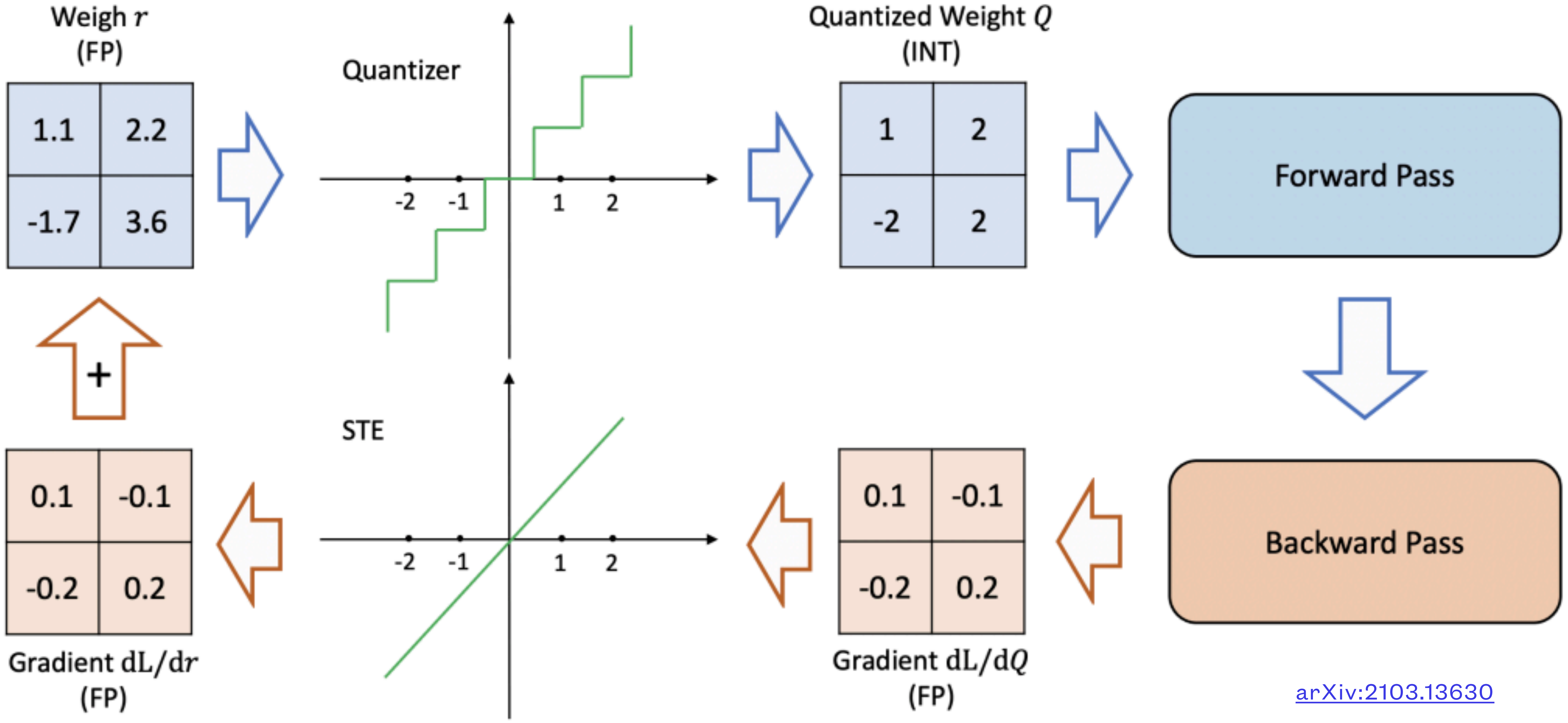
FP32

64	134	217
76	119	21
3	81	99

INT8

Quantization Aware Training (QAT)

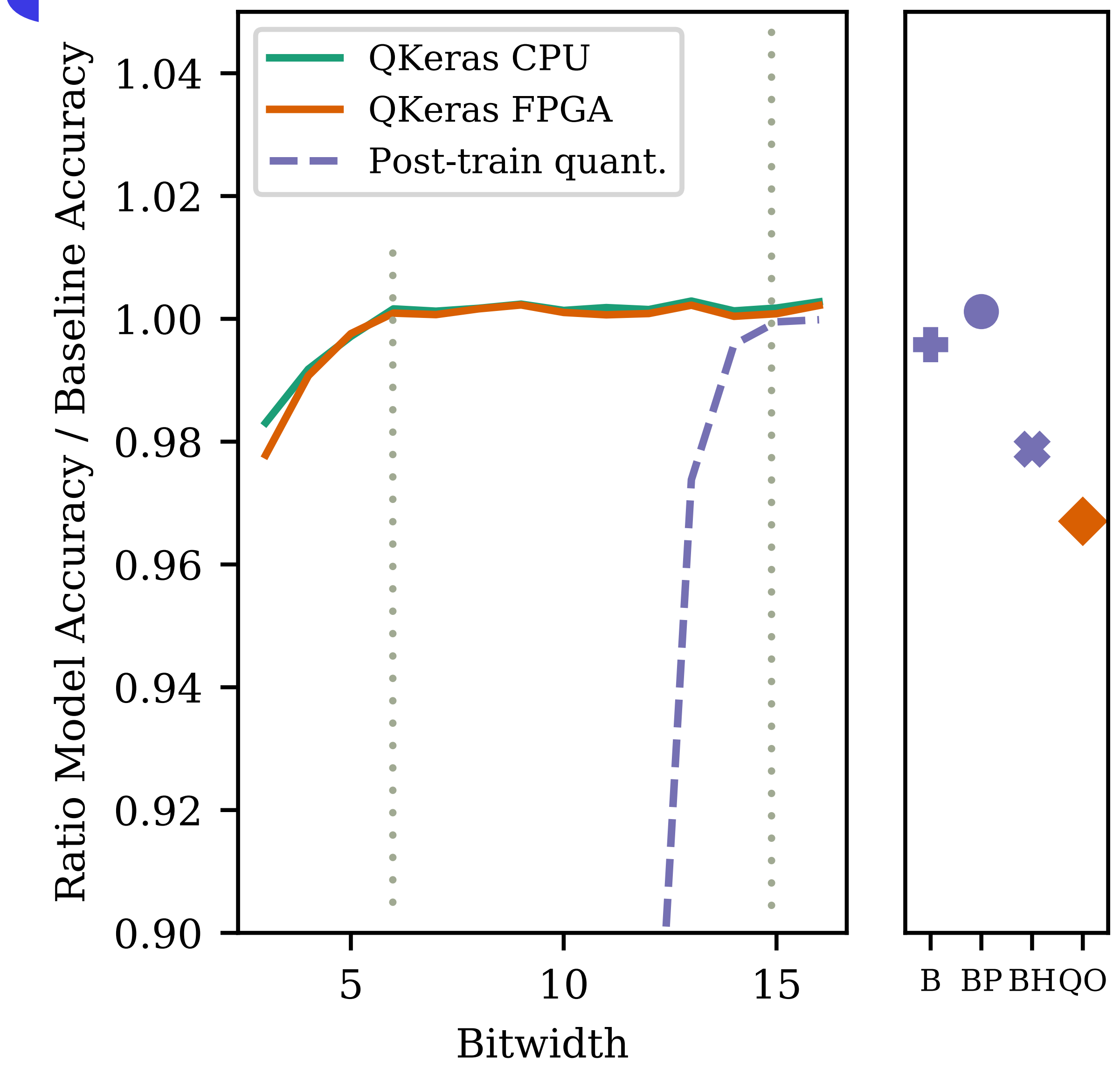
- With QAT, you constrain weights/biases/activations to fewer values (like fixed point)
  - Superior to PTQ for lower bitwidths - can go all the way down to 1 bit (representing ±1)
  - Often using ‘Straight Through Estimator’ for back propagation

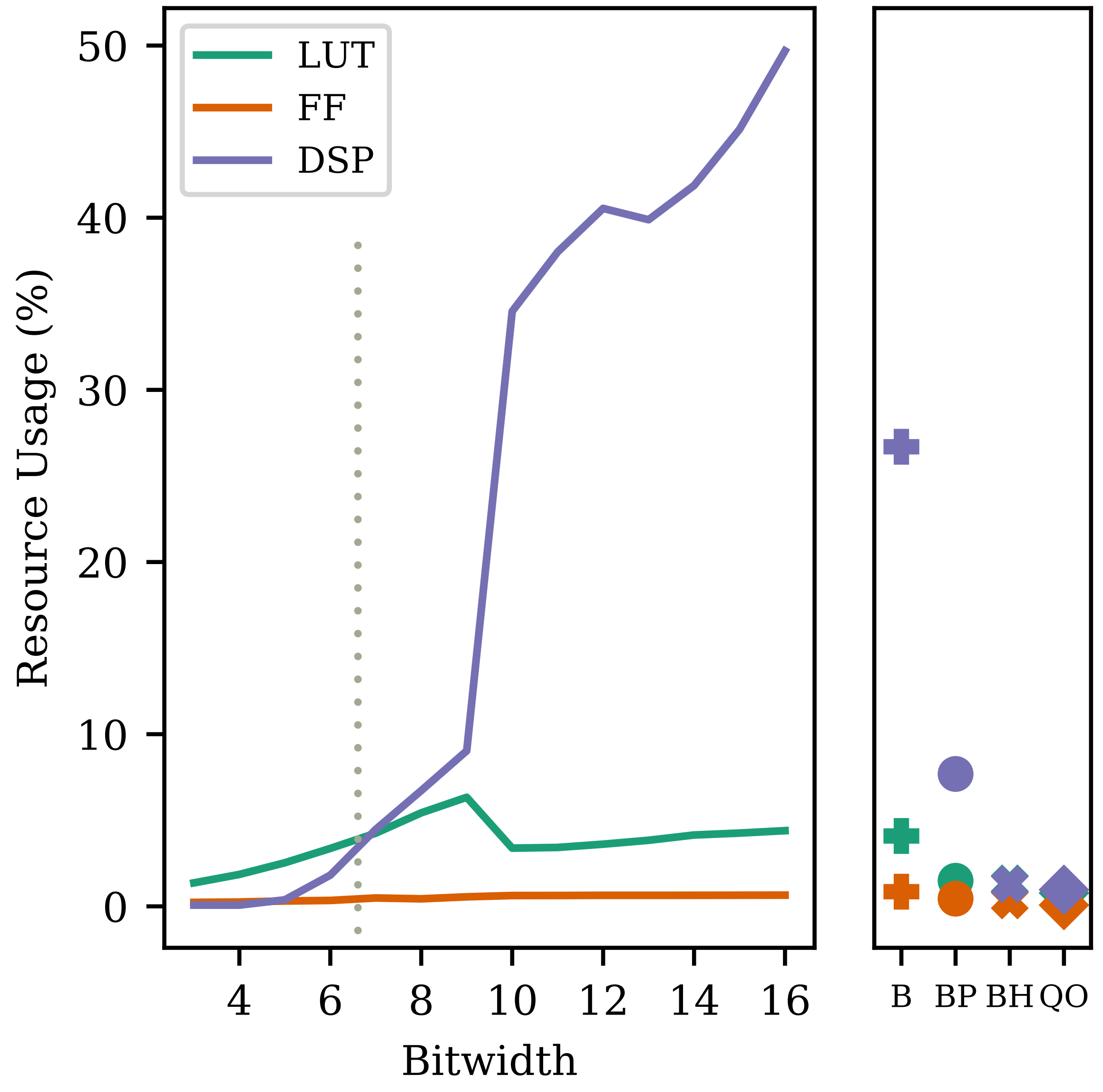


arXiv:2103.13630

Quantization Aware Training

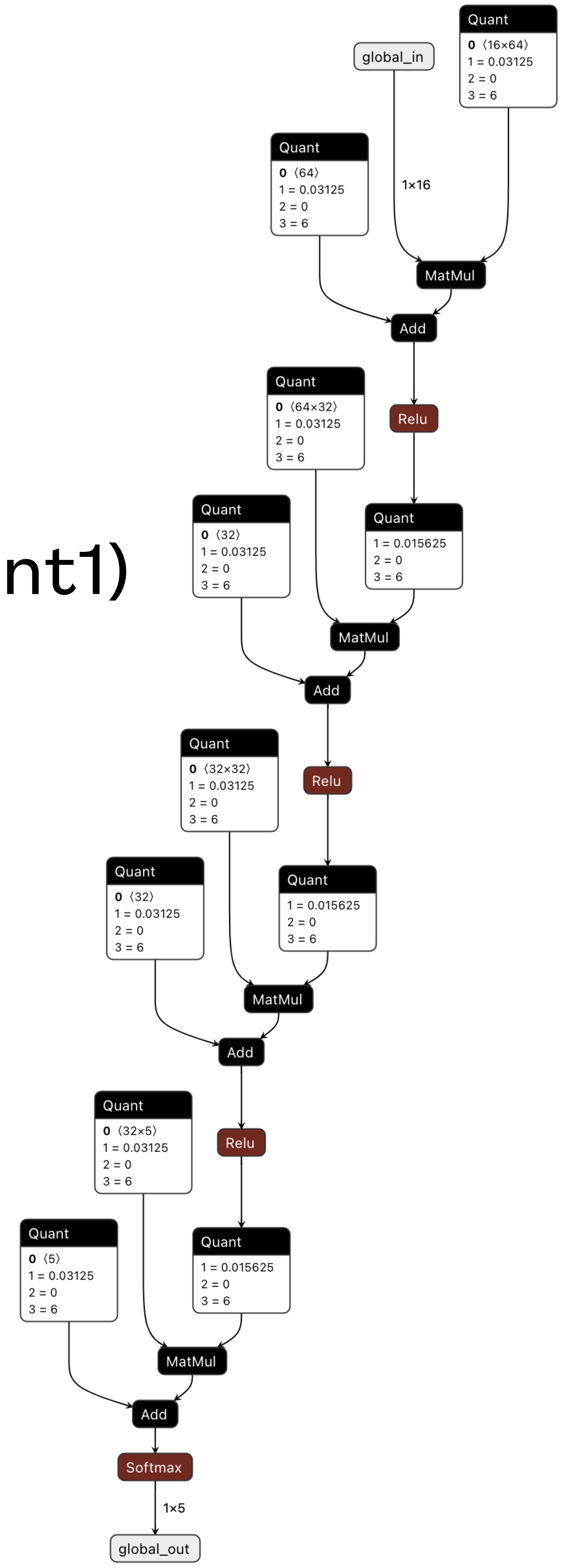
- QAT impact is significant - here w/QKeras & hls4ml
  - QAT maintains same accuracy until 6 bits, then drops slightly (not that much)
  - PTQ accuracy falls very fast reducing bitwidth
- Quantization can be heterogeneous
  - Different choices for weights vs activations, and for different layers
  - Wider “more expressive” activations can help
  - For autoencoders: higher precision at the bottleneck layers;
  - For regression: higher precision at the end (more continuous, less discrete output)
- AutoQ tool for training NNs with hardware-cost constraints [6]





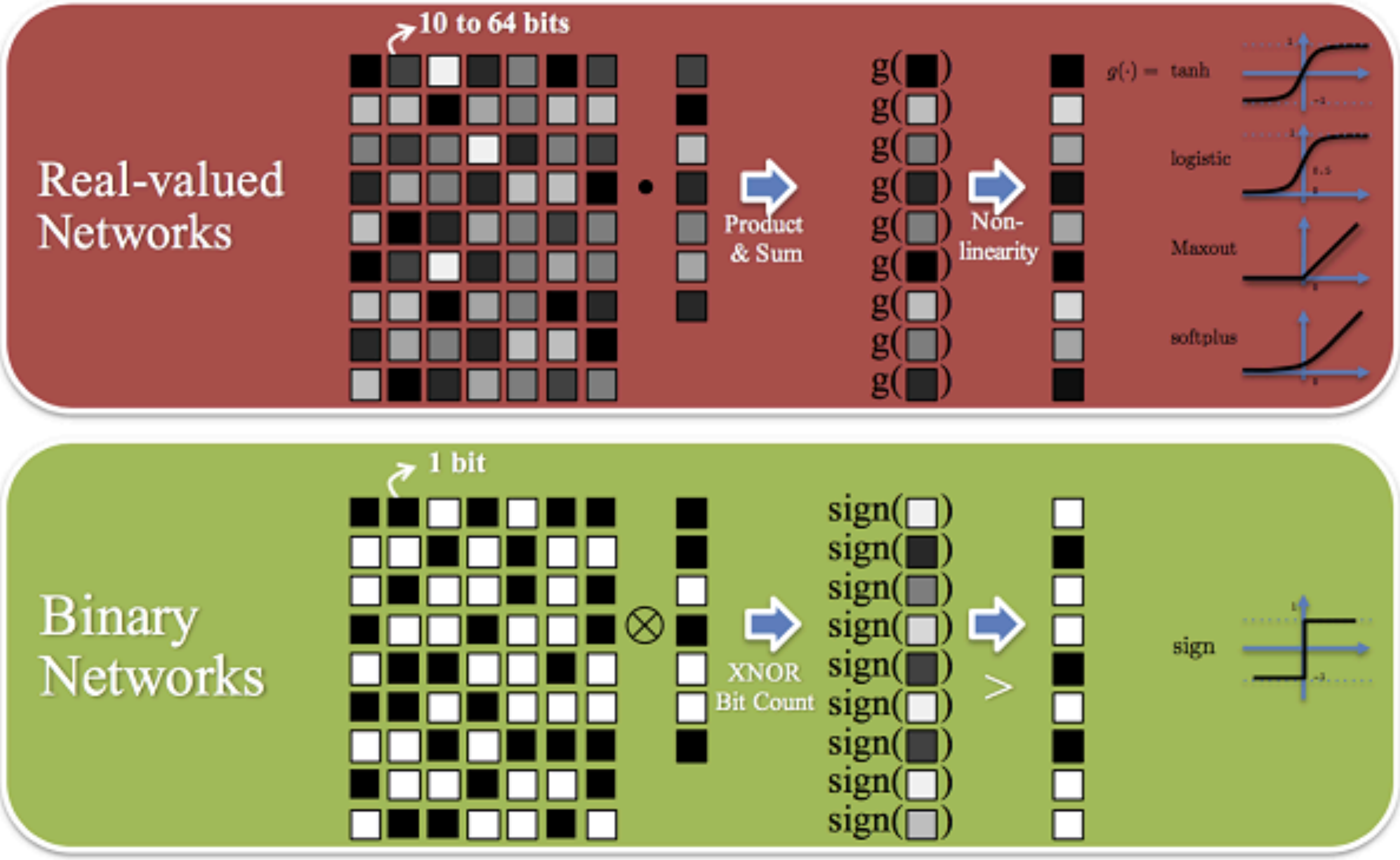
Representing Quantized NNs

- Lots of tools like Tensorflow, PyTorch, TensorRT have support for low precision (including QAT)
- But they are typically restricted to common CPU/GPU types (float16, int8, int4, int1)
    - For dataflow (layer unrolled) FPGA inference, we would like more flexibility
- Collab w/ Xilinx Research Labs: HLS4ML team develop QONNX [7]
- Extend QONNX with `Quant` node
    - Flexible number of bits, zero-point, and per-channel scale factors
    - onnxruntime execution thanks to FINN (Xilinx RL NNs)
    - QONNX is exported by Brevitas, others are working on it, and we develop a QKeras to QONNX conversion
- github.com/fastmachinelearning/qonnx



Binary / Ternary neural networks

- DSP multipliers often limiting resource
- Can often go down to 1- or 2-bit weights with limited performance loss
- Can have very efficient computation in the FPGA (and CPU/GPU/smartphone)
- Binarize weights **but not gradients** during backpropagation
- Use Binary Tanh, Ternary Tanh or ReLU activation
- BNN: arxiv.1602.02830
- TNN: arxiv.1605.04711



intel.com [4]

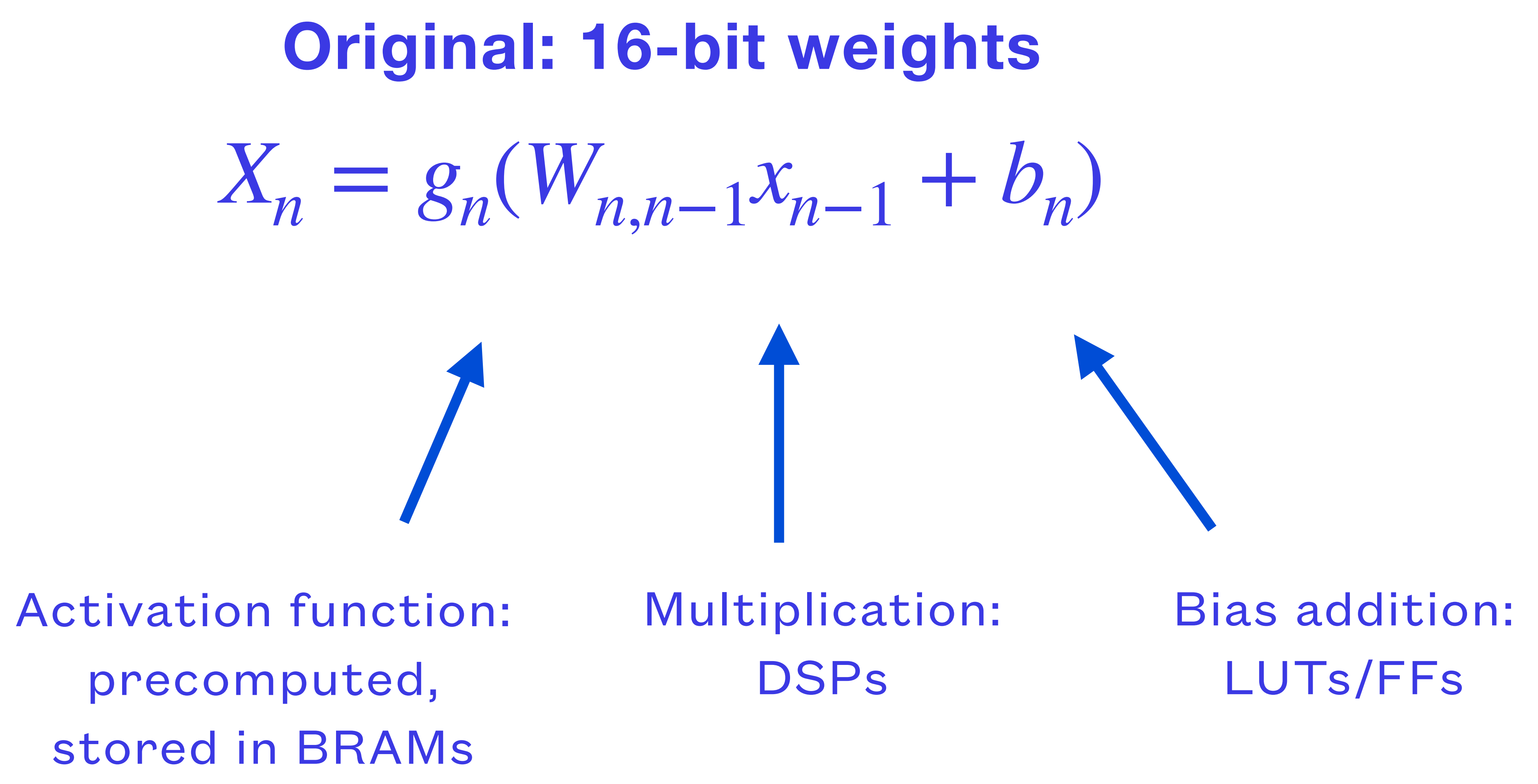
BNN - Dense Layer

- DSPs often limiting FPGA resource for NNs
- Encode ‘-1’ as ‘0’
- Multiplication become XNOR, sum becomes bitcount

A	B	A*B
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

A	A’
-1	0
1	1

A	B	A==B
0	0	1
0	1	0
1	0	0
1	1	1



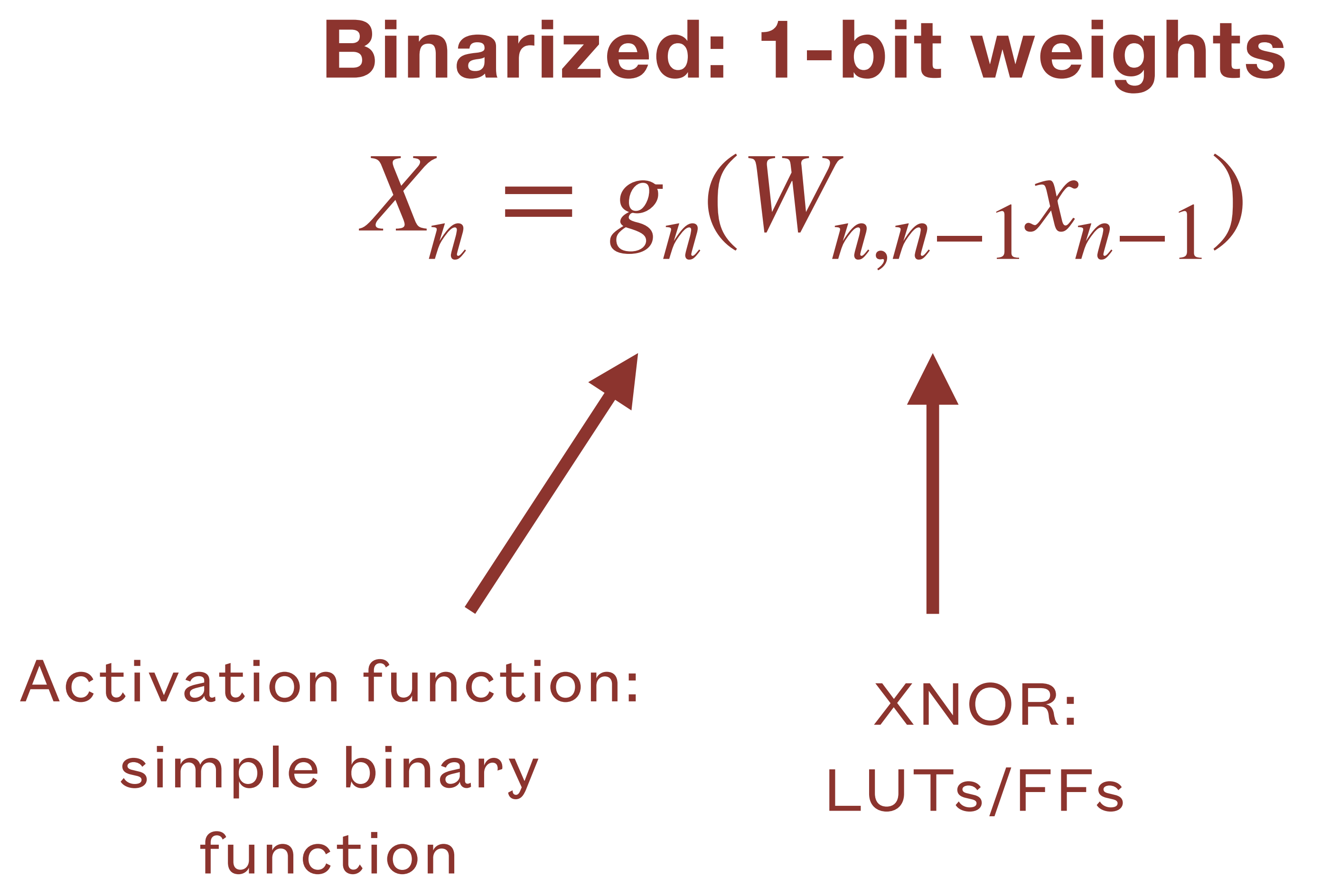
Original: 16-bit weights

$$X_n = g_n(W_{n,n-1}x_{n-1} + b_n)$$

Activation function: precomputed, stored in BRAMs

Multiplication: DSPs

Bias addition: LUTs/FFs



Binarized: 1-bit weights

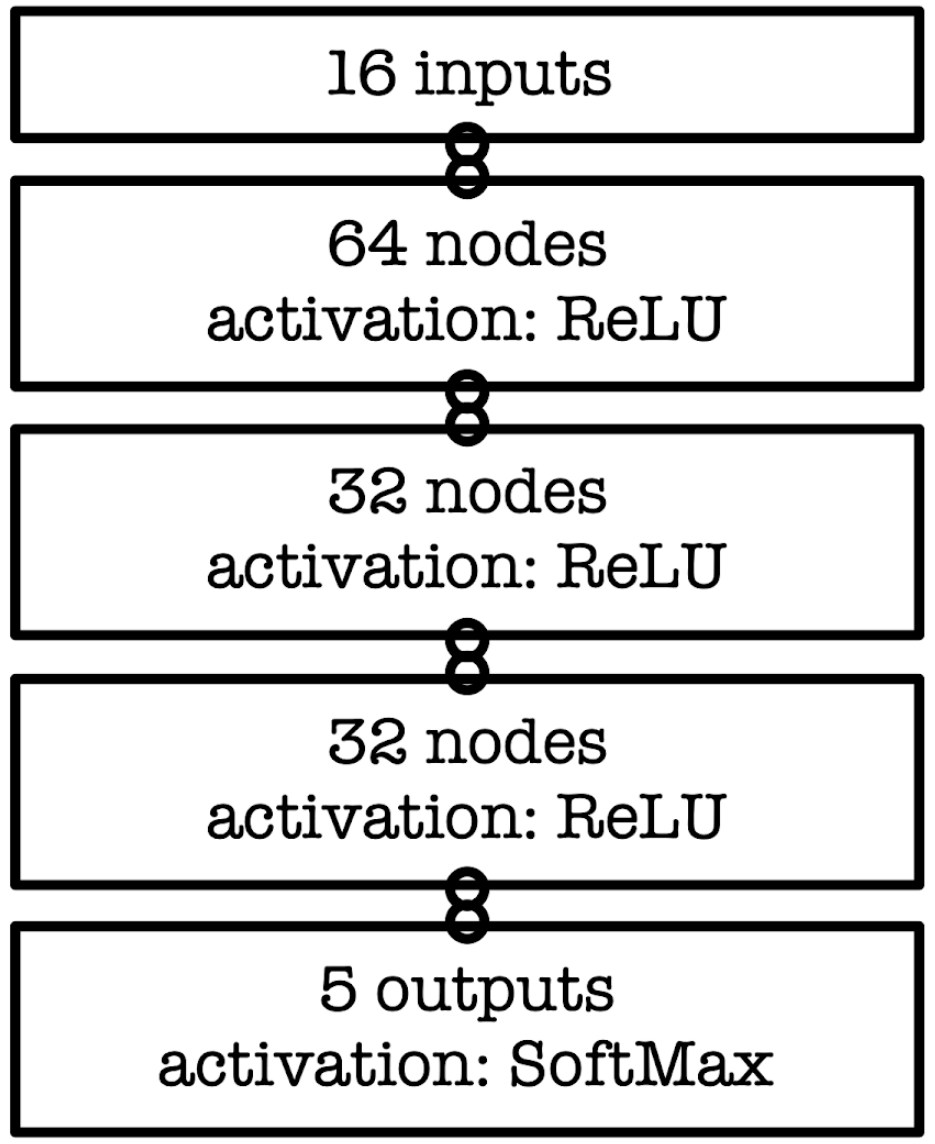
$$X_n = g_n(W_{n,n-1}x_{n-1})$$

Activation function: simple binary function

XNOR: LUTs/FFs

Jet tagging

- HLS4ml tutorial example [2]
  - Tagging jets (5 classes, 16 input variables)
- 3 fully connected layers



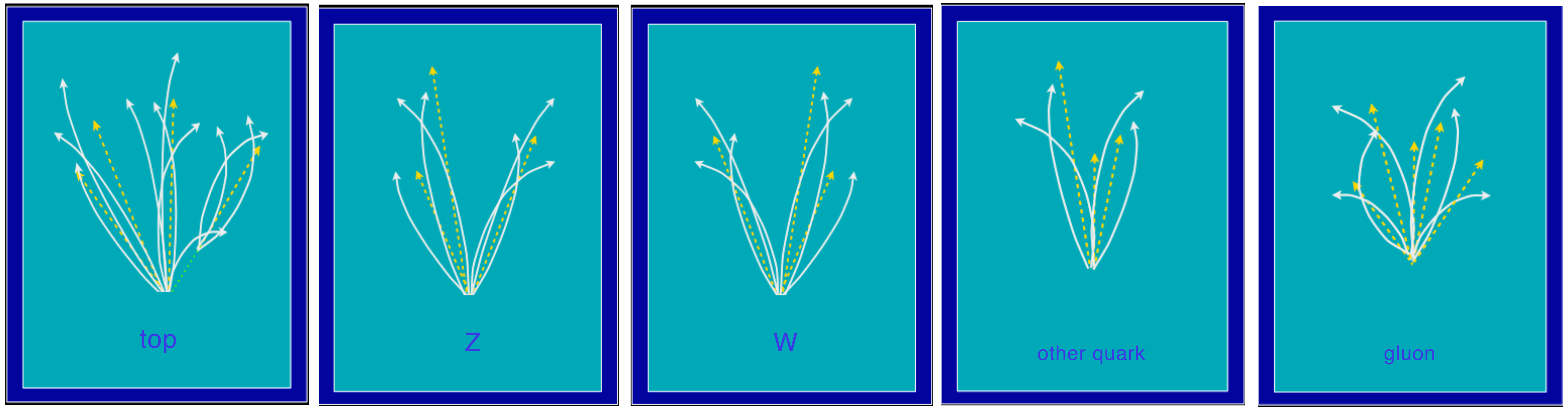
16 inputs

64 nodes
activation: ReLU

32 nodes
activation: ReLU

32 nodes
activation: ReLU

5 outputs
activation: SoftMax

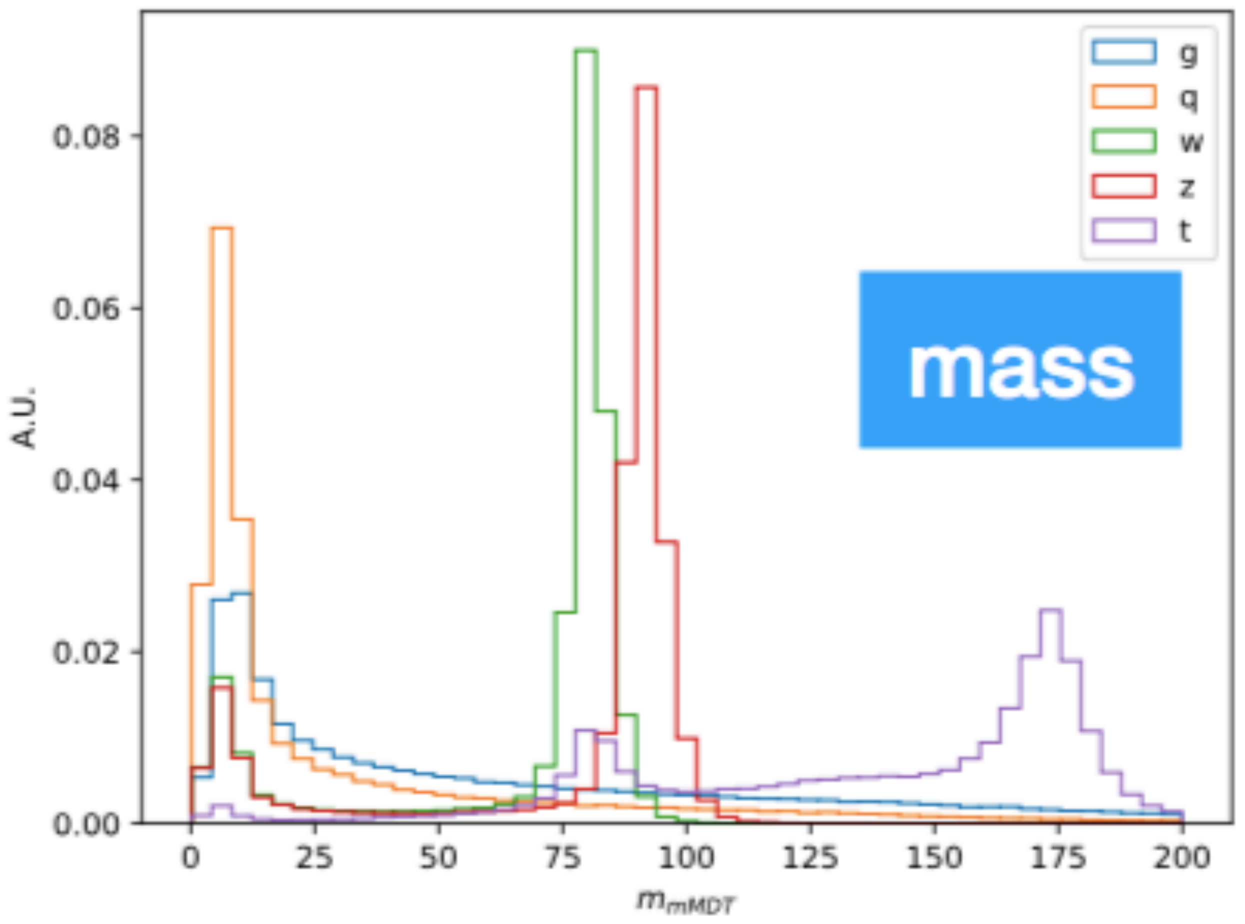


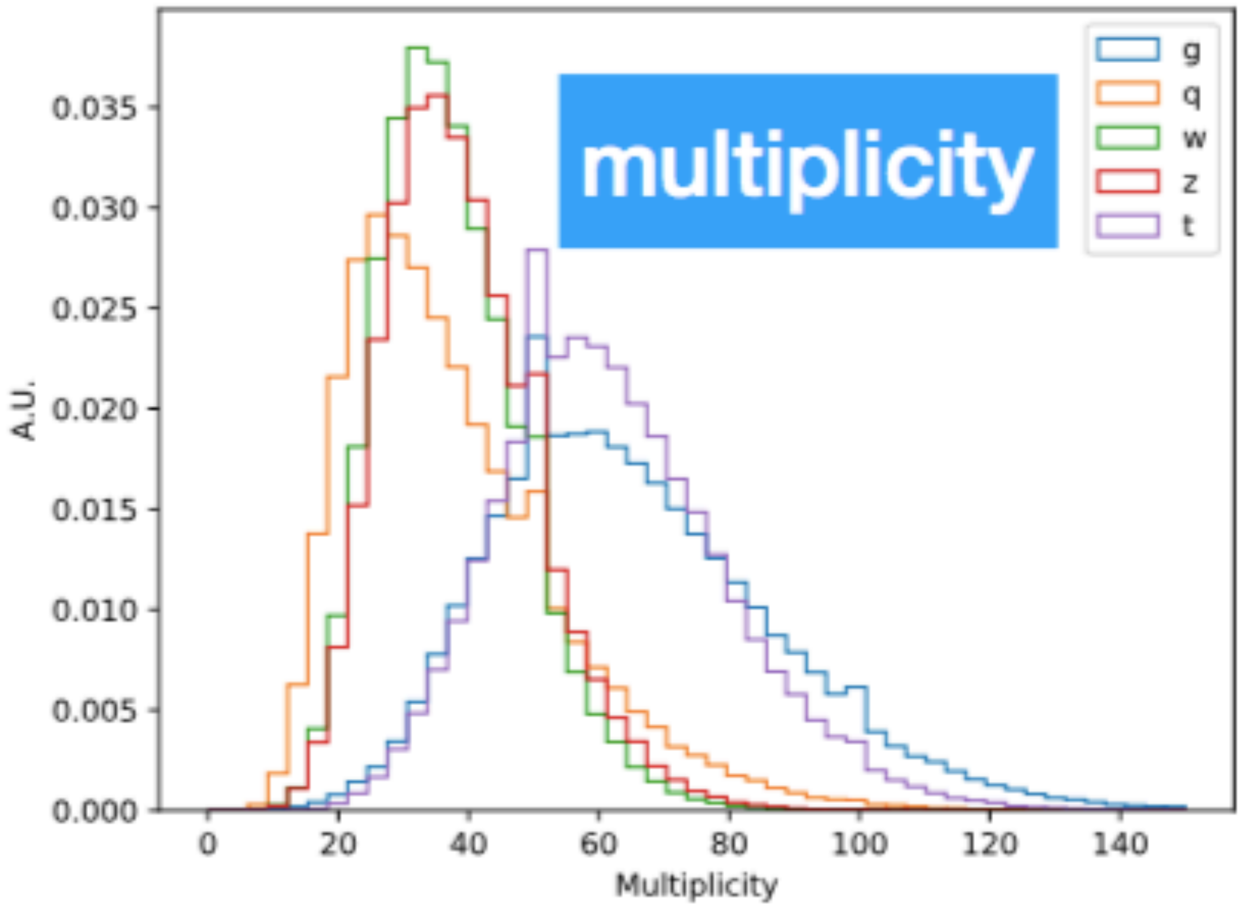
Jet tagging

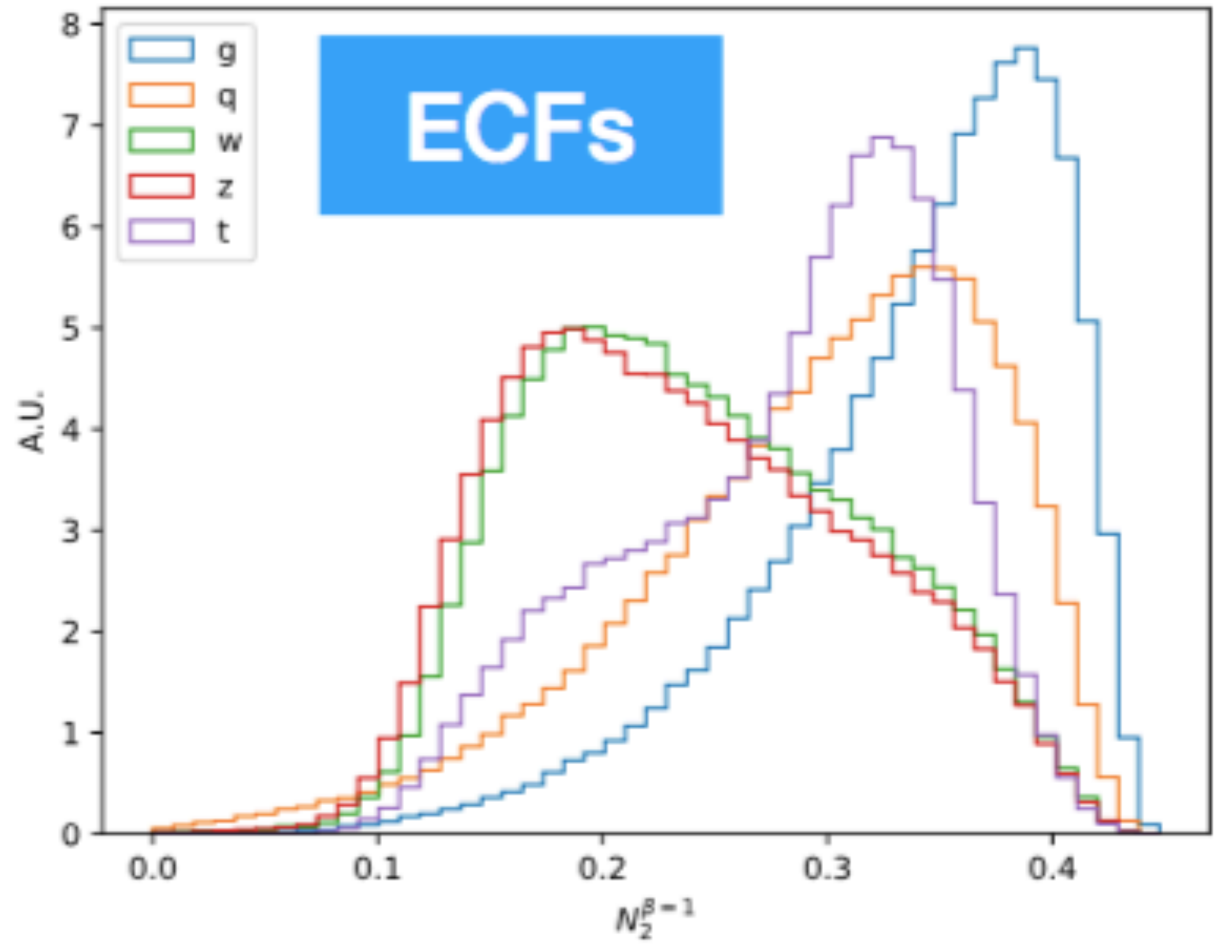
- Trained (on GPU) the five output multi-classifier on a sample of ~ 1M events with two boosted WW/ZZ/tt/qq/gg anti-kT jets

- 16 expert-level input variables, computed with FastJet:
  - known to have high discrimination power from offline data analyses and published studies

energy correlation functions

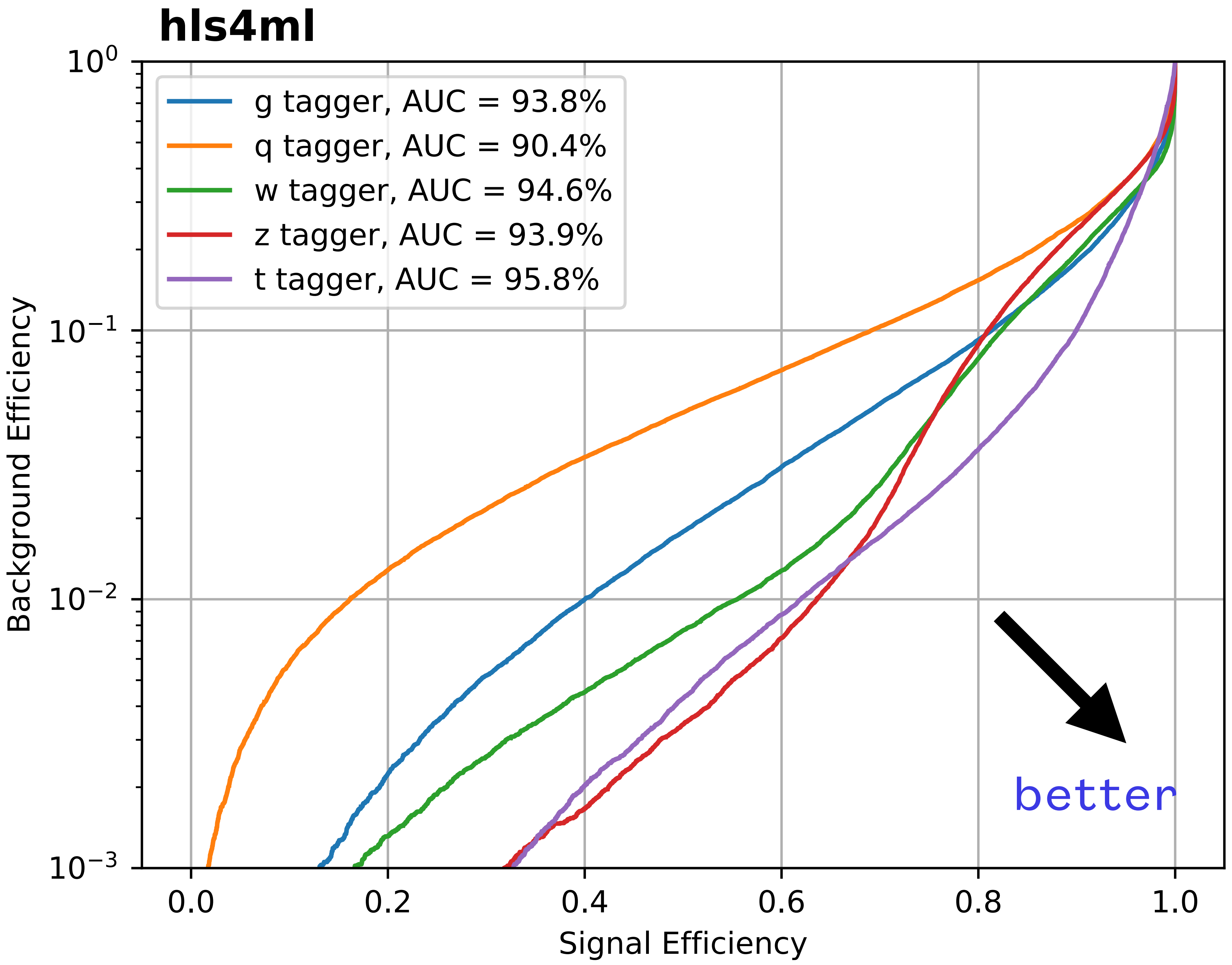


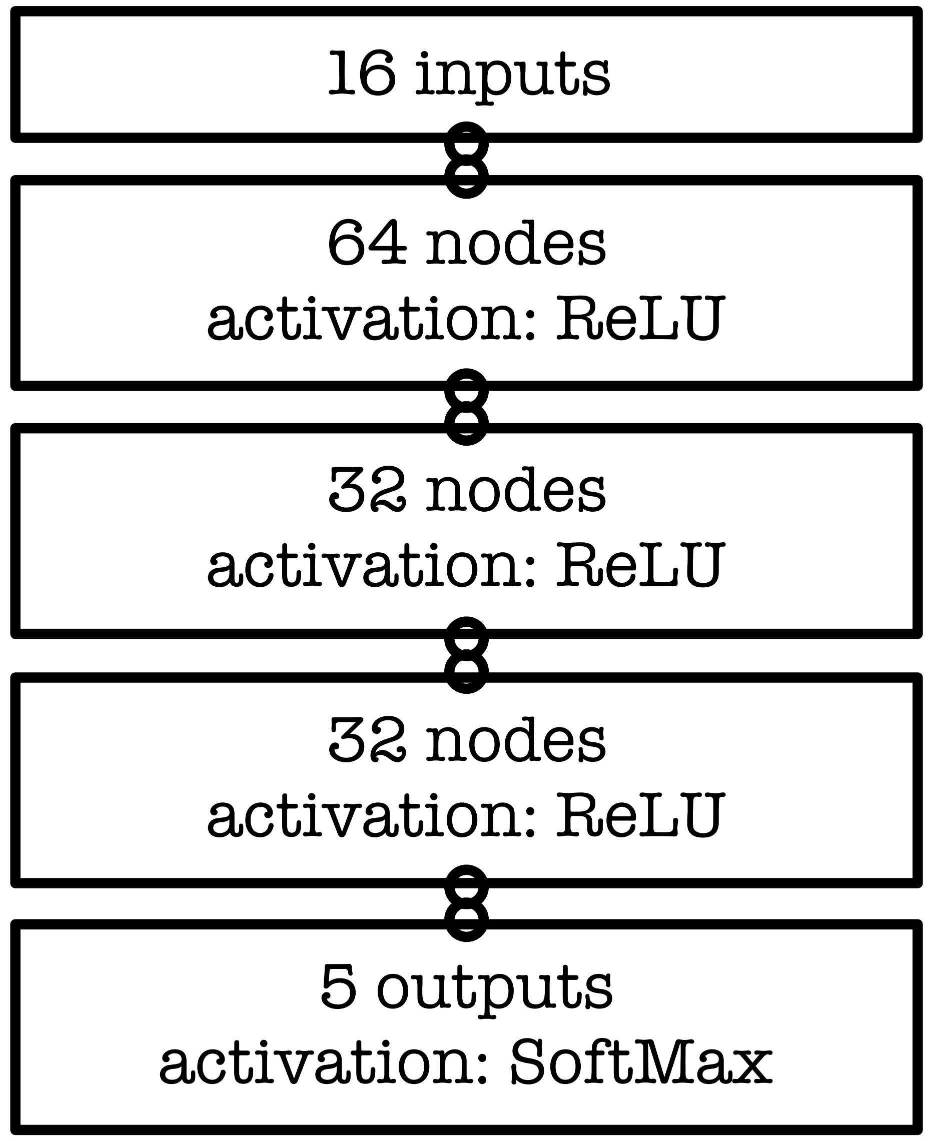




Jet tagging

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





K + TensorFlow

AUC = AREA UNDER ROC CURVE
(100% IS PERFECT, 20% IS RANDOM)

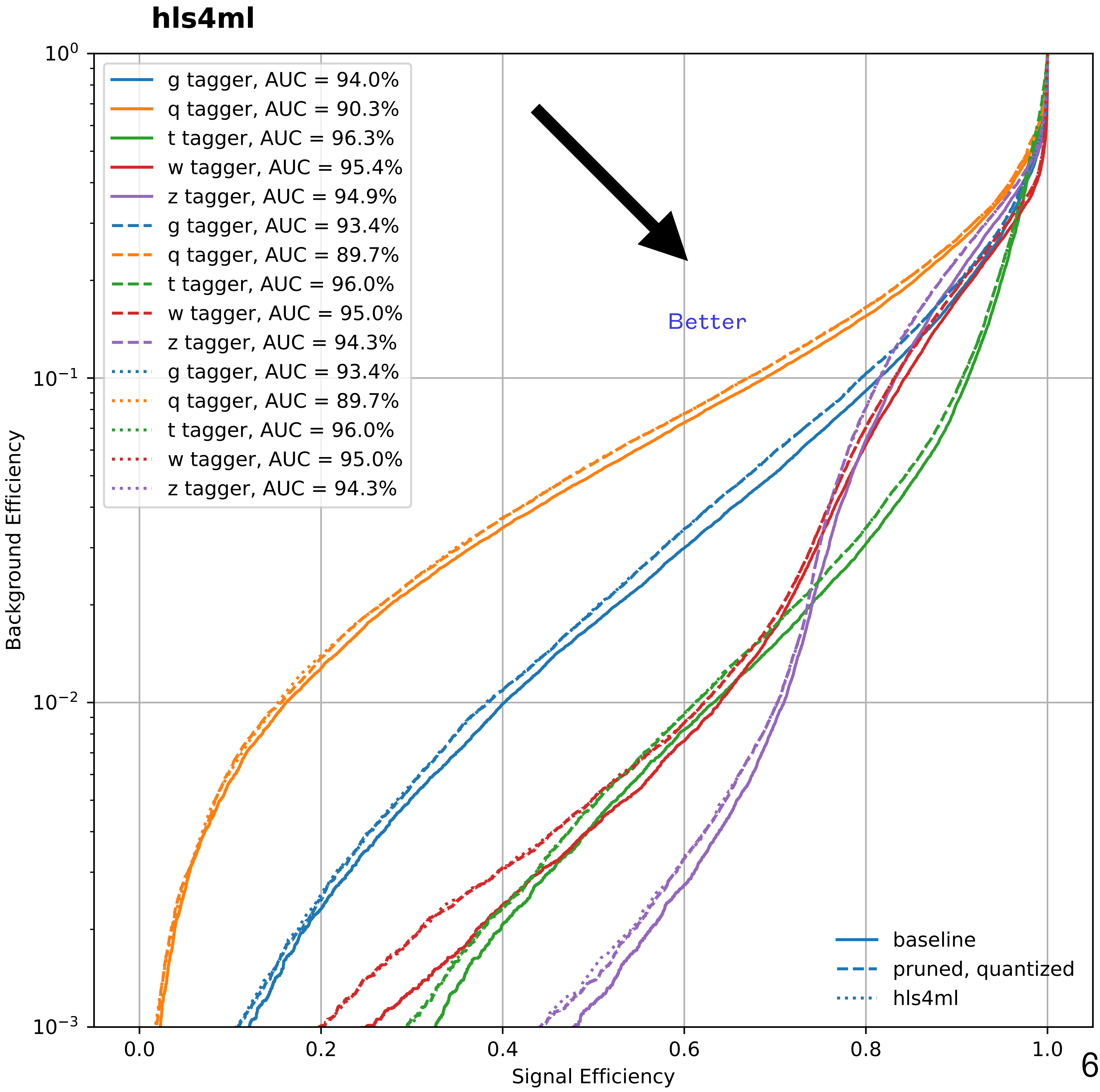
Jet tagging w/ QAT & Pruning

A. **Keras** floating point training, 16b inference

B. **QKeras** with 6 bits for weights, biases, activations & 75% sparsity target with TFMOT

Minimal code changes to go A to B

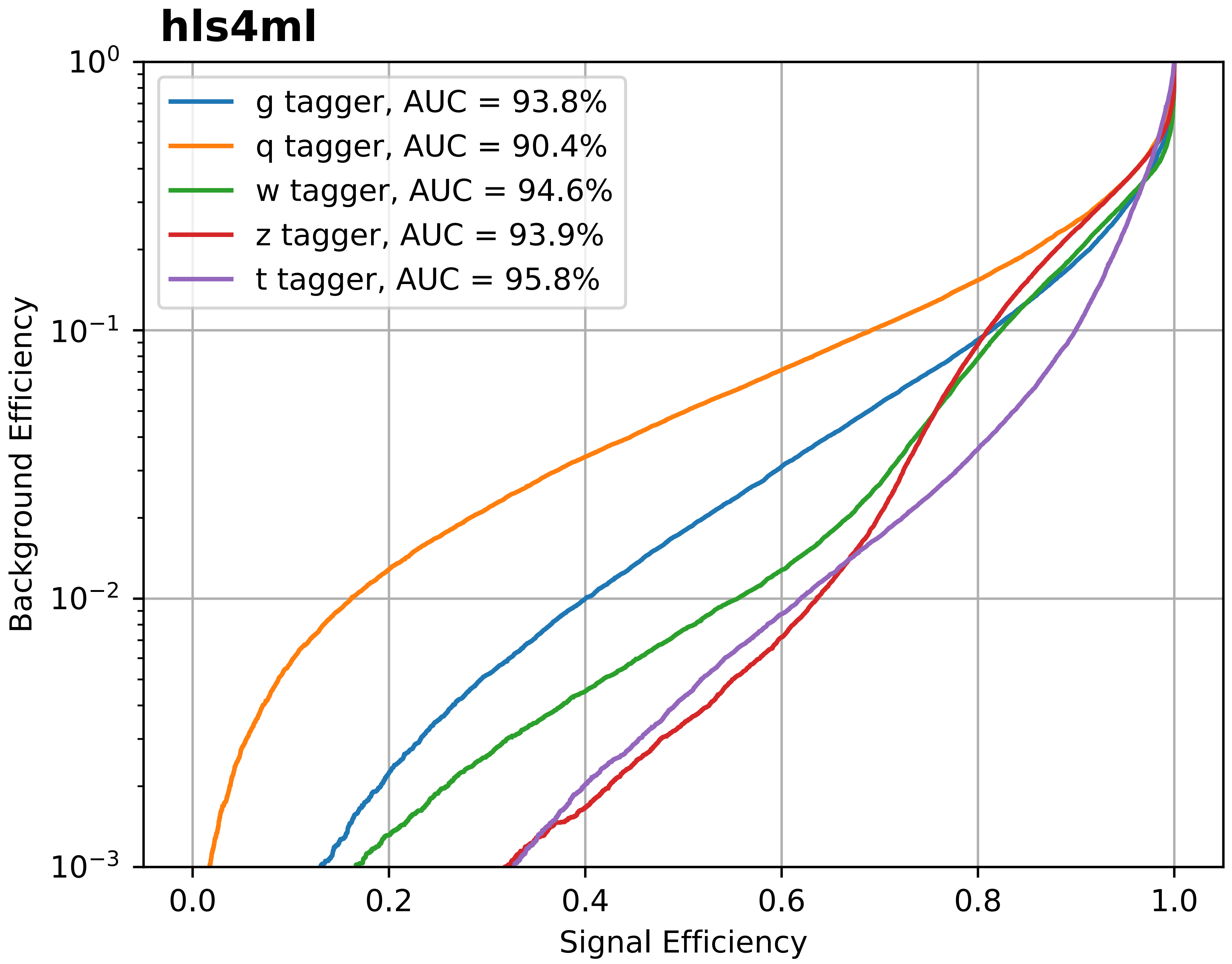
Xilinx VU9P	Latency	DSP	LUT
Keras 16b	50 ns	1890 (15%)	5%
QKeras 6b	40 ns	22 (~0%)	1%



BNN - Jet Classification

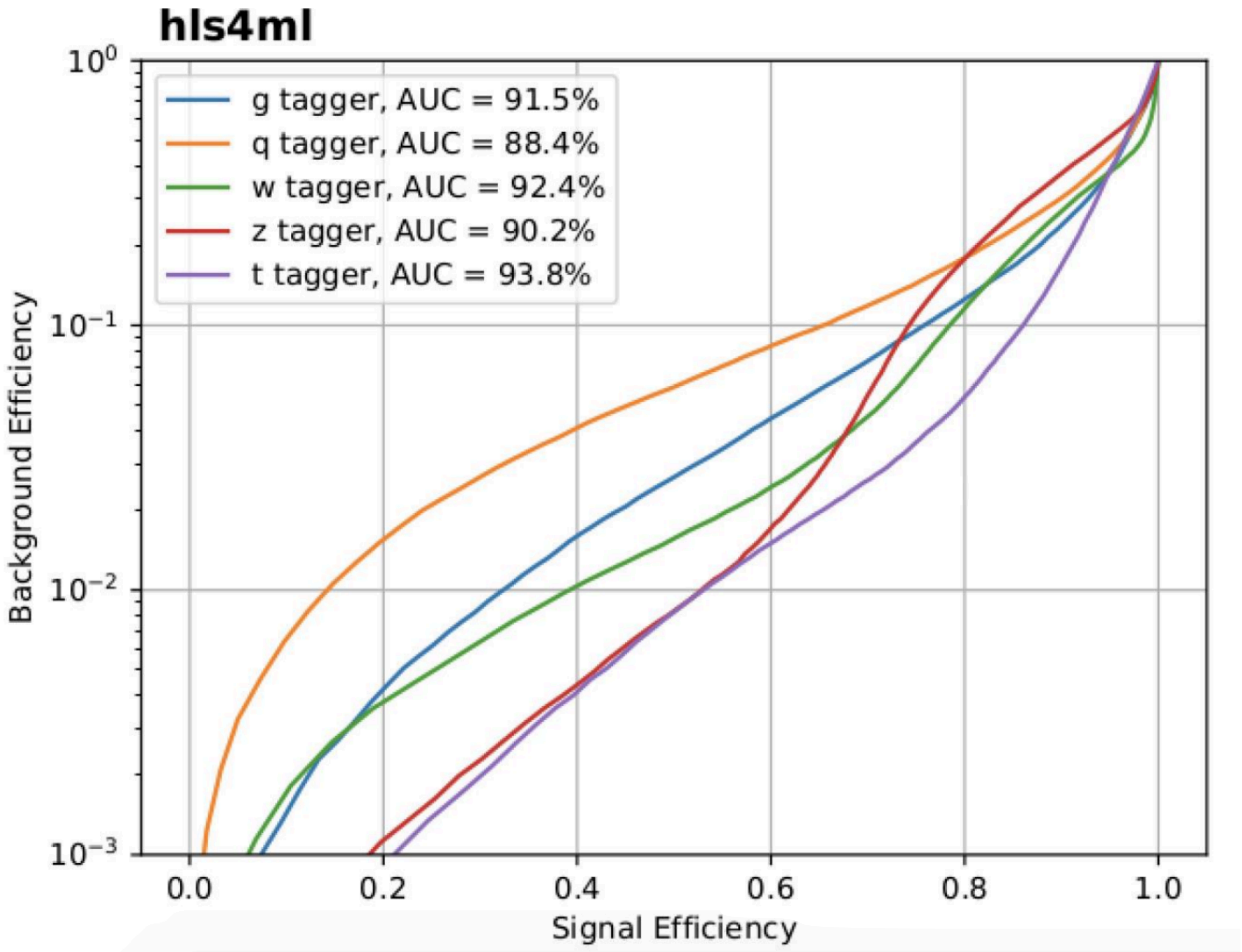
- Design an architecture to perform the same jet classification task but now with binary weights and activations - n neutrons 7x per layer

- Performed hyperparameter optimization to find most performant model within some constraints



Original: 16-bit weights

Average accuracy: 0.75

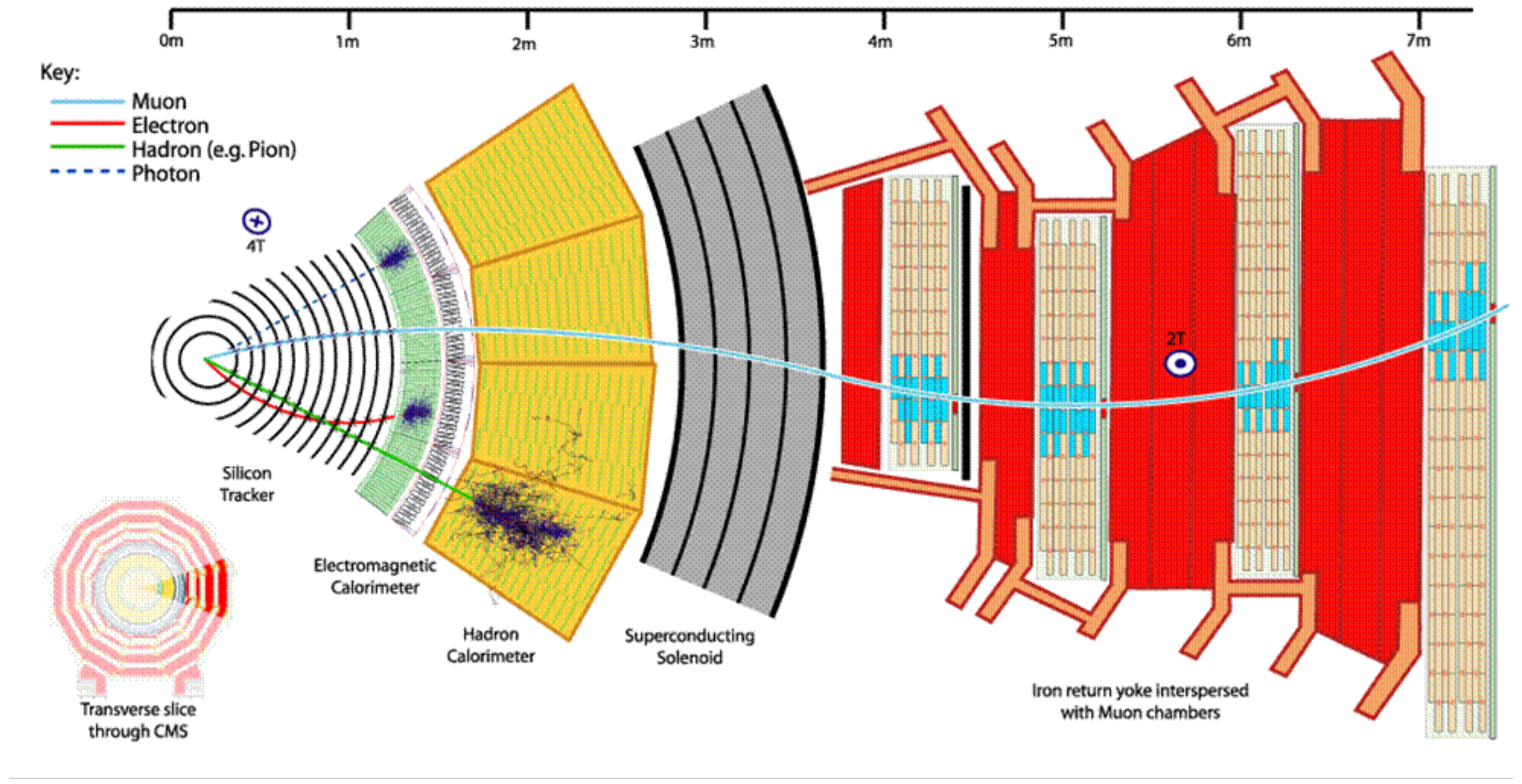


Binarized: 1-bit weights

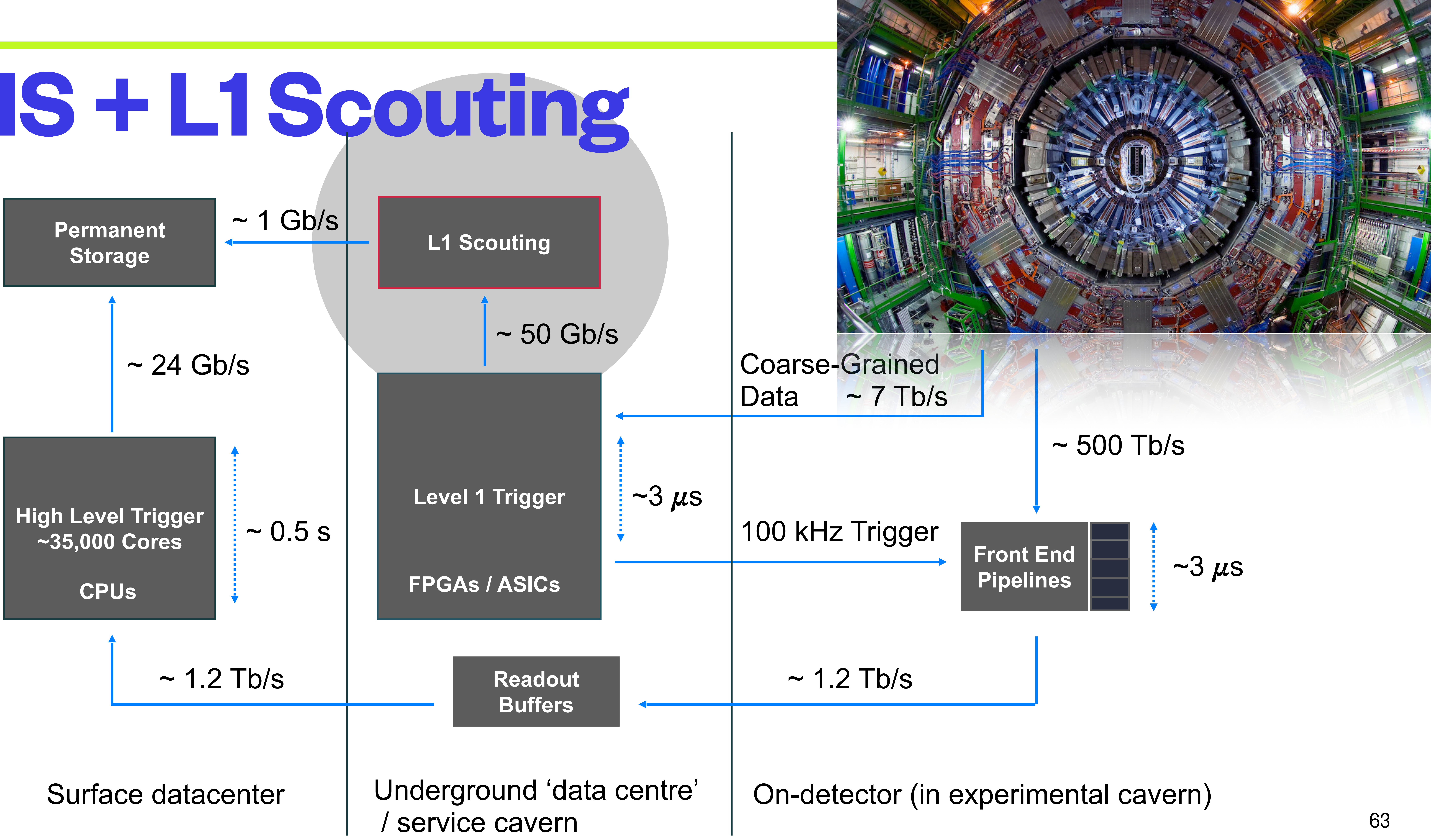
Average accuracy: 0.72

L1 Scouting

- Stream processor / accelerator hybrid
- What does L1 accept miss?
- Can we acquire L1 trigger data at full bunch crossing rate
    - subset of detector information, limited resolution
- Allows for analysis of certain topologies at full rate
    - semi real-time analysis and/or
    - storing of tiny event record
- Demonstrated for first time at end of 2018
- Upgraded w/ new boards in 2021 - validated with LHC test beams
- For LHC Run 3 (2022) - prompt & displaced muons, jets, electrons/photons, taus and global trigger outputs included



CMS + L1 Scouting



Permanent Storage

~ 1 Gb/s

L1 Scouting

~ 50 Gb/s

~ 24 Gb/s

Coarse-Grained Data ~ 7 Tb/s

~ 500 Tb/s

High Level Trigger
~35,000 Cores

CPUs

~ 0.5 s

Level 1 Trigger

FPGAs / ASICs

~3 μs

100 kHz Trigger

Front End Pipelines

~3 μs

~ 1.2 Tb/s

Readout Buffers

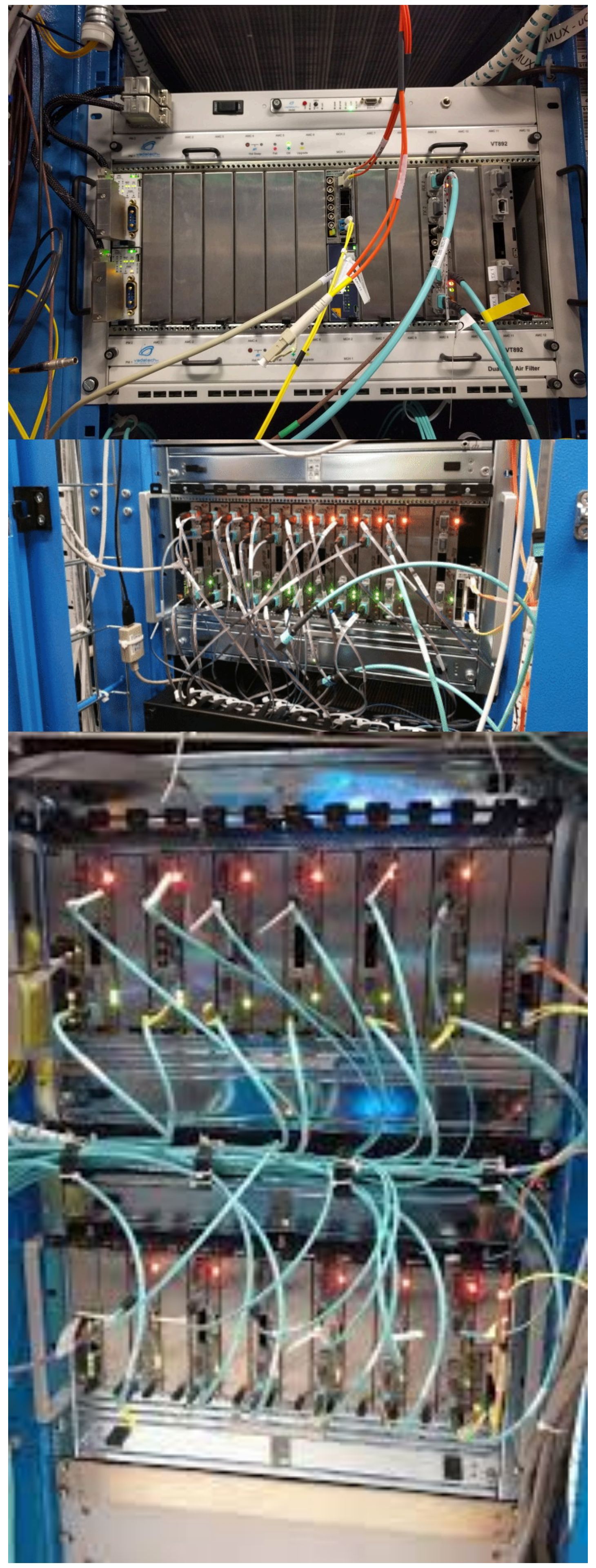
~ 1.2 Tb/s

Surface datacenter

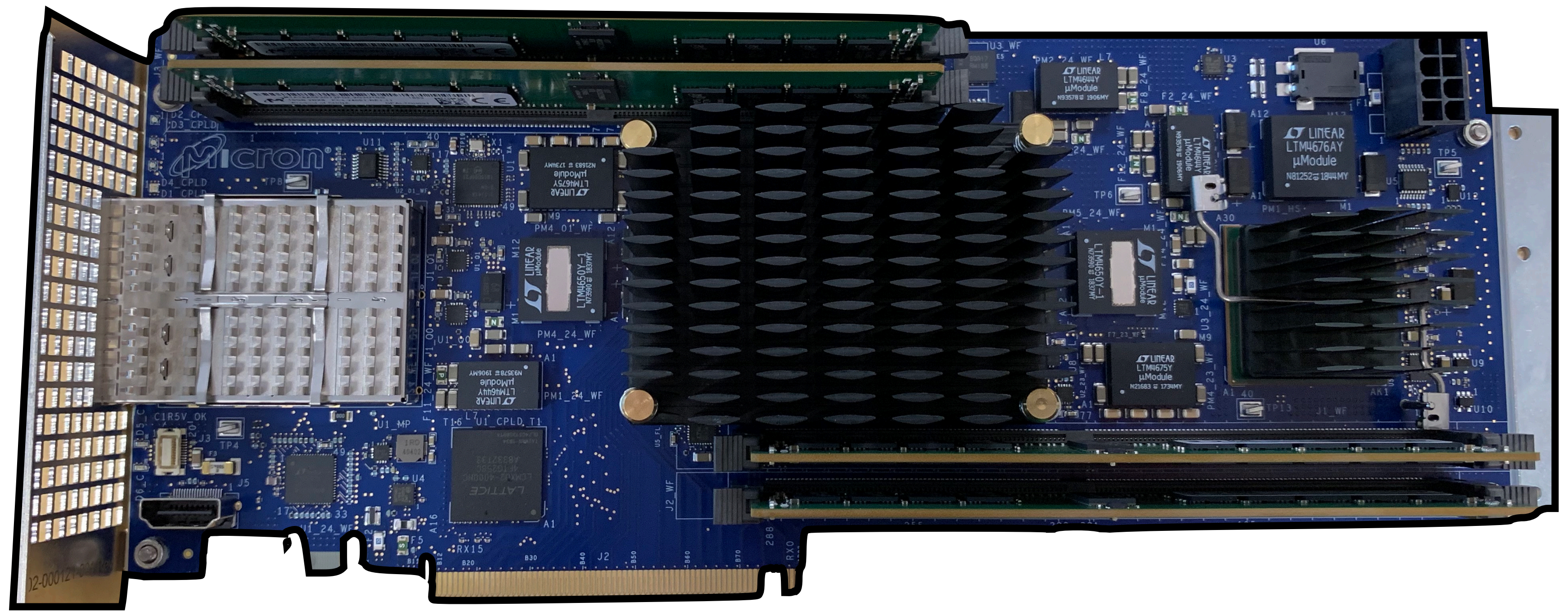
Underground ‘data centre’ / service cavern

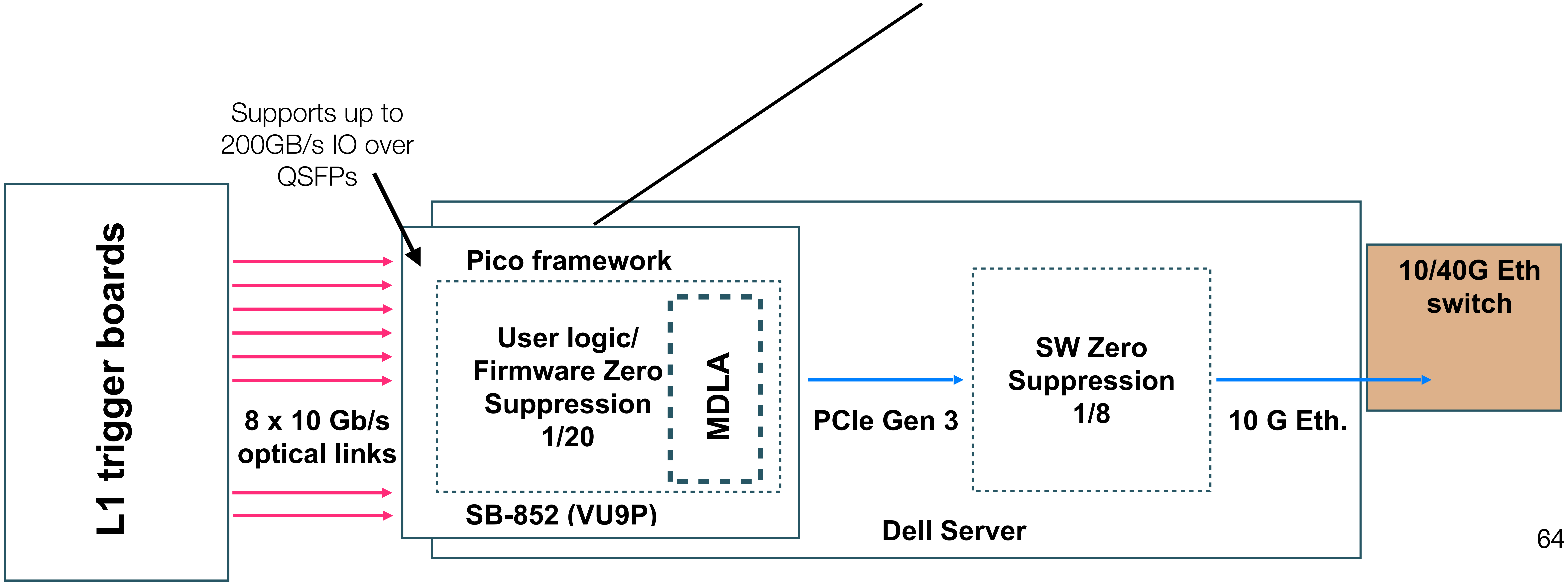
On-detector (in experimental cavern)

L1 Scouting with SB-852



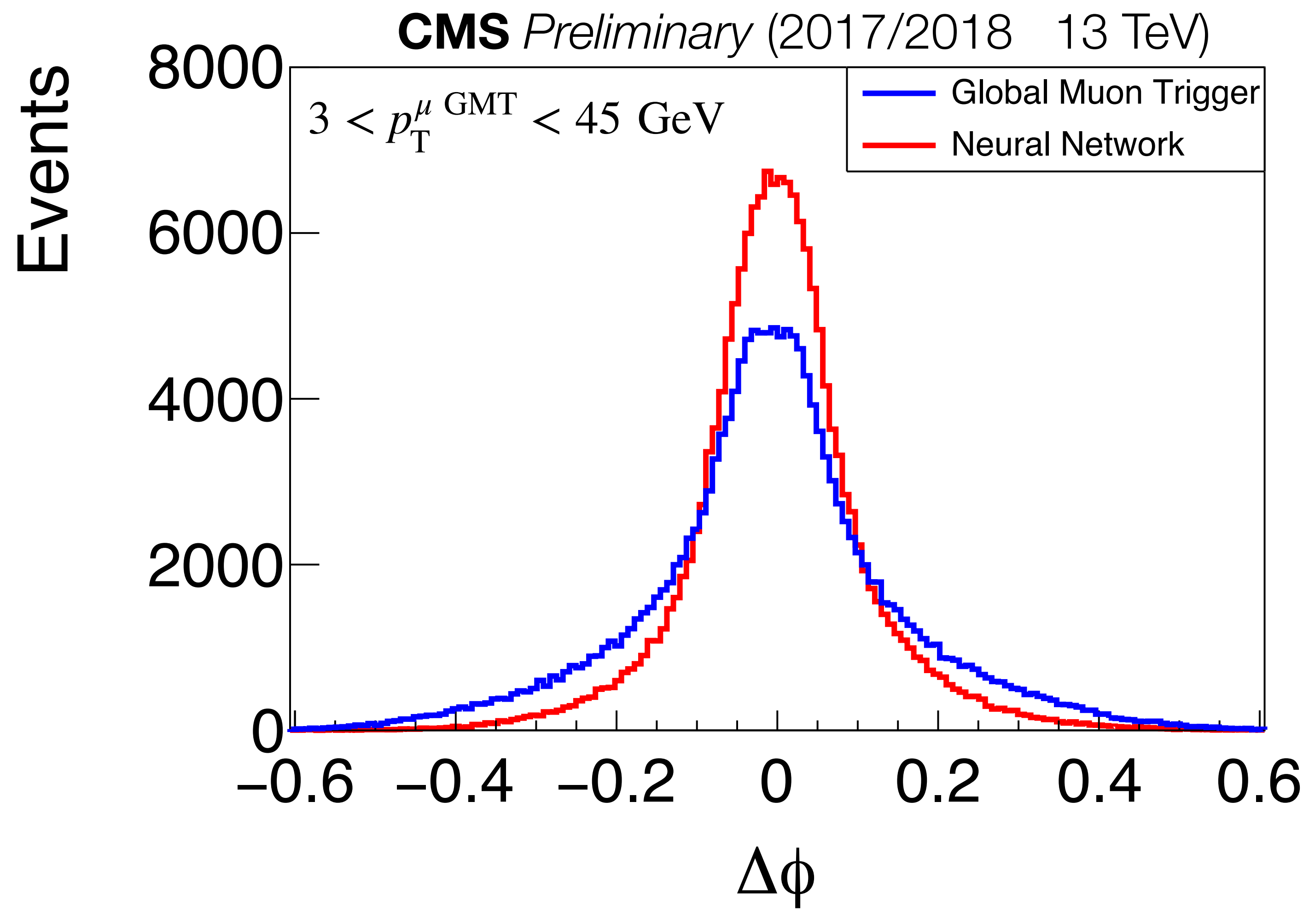
- Micron SB-852 for optical input -> DMA to PC
- Perform NN inference with Micron DLA after firmware ZS



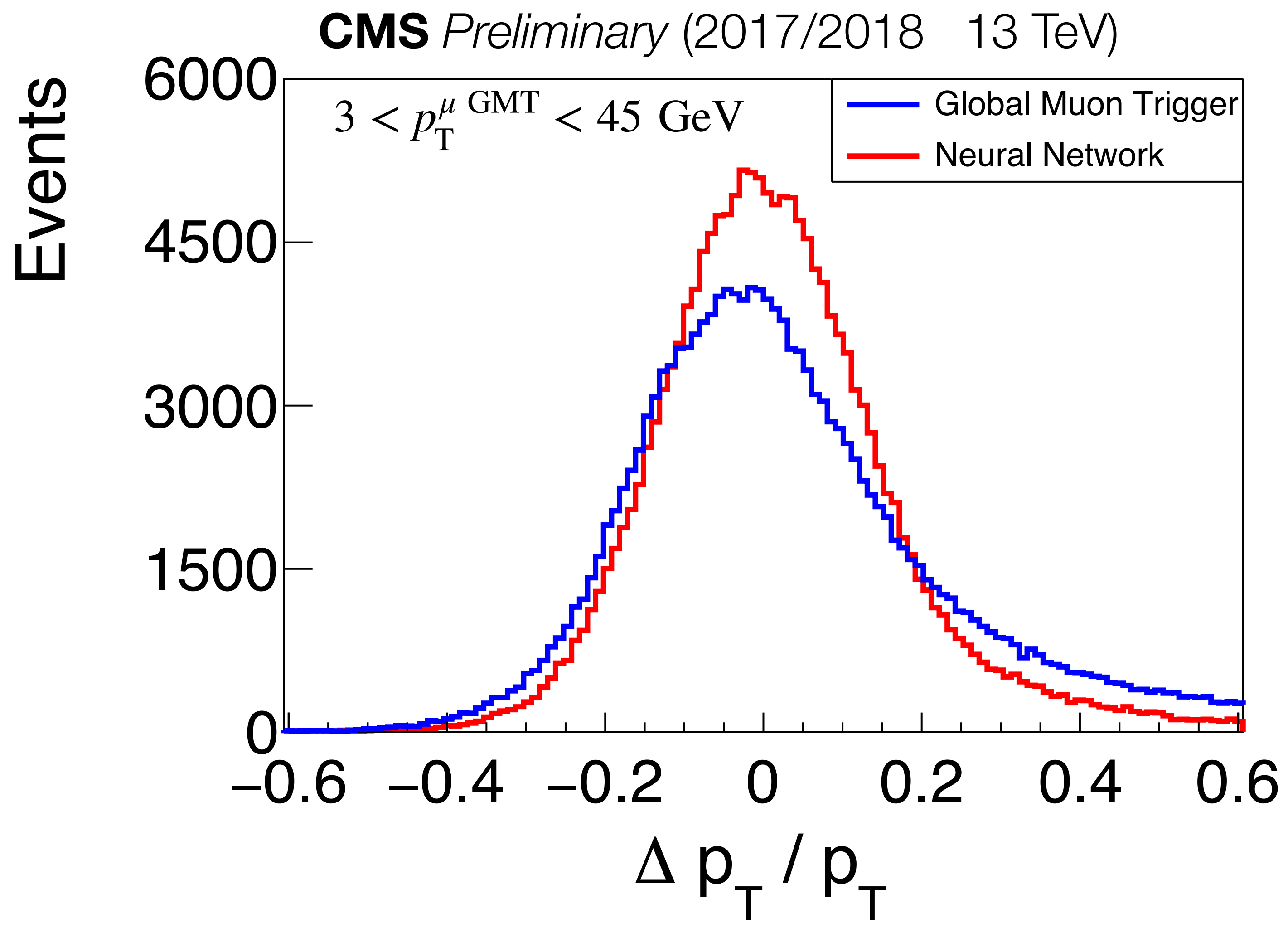


Why ML for L1 scouting?

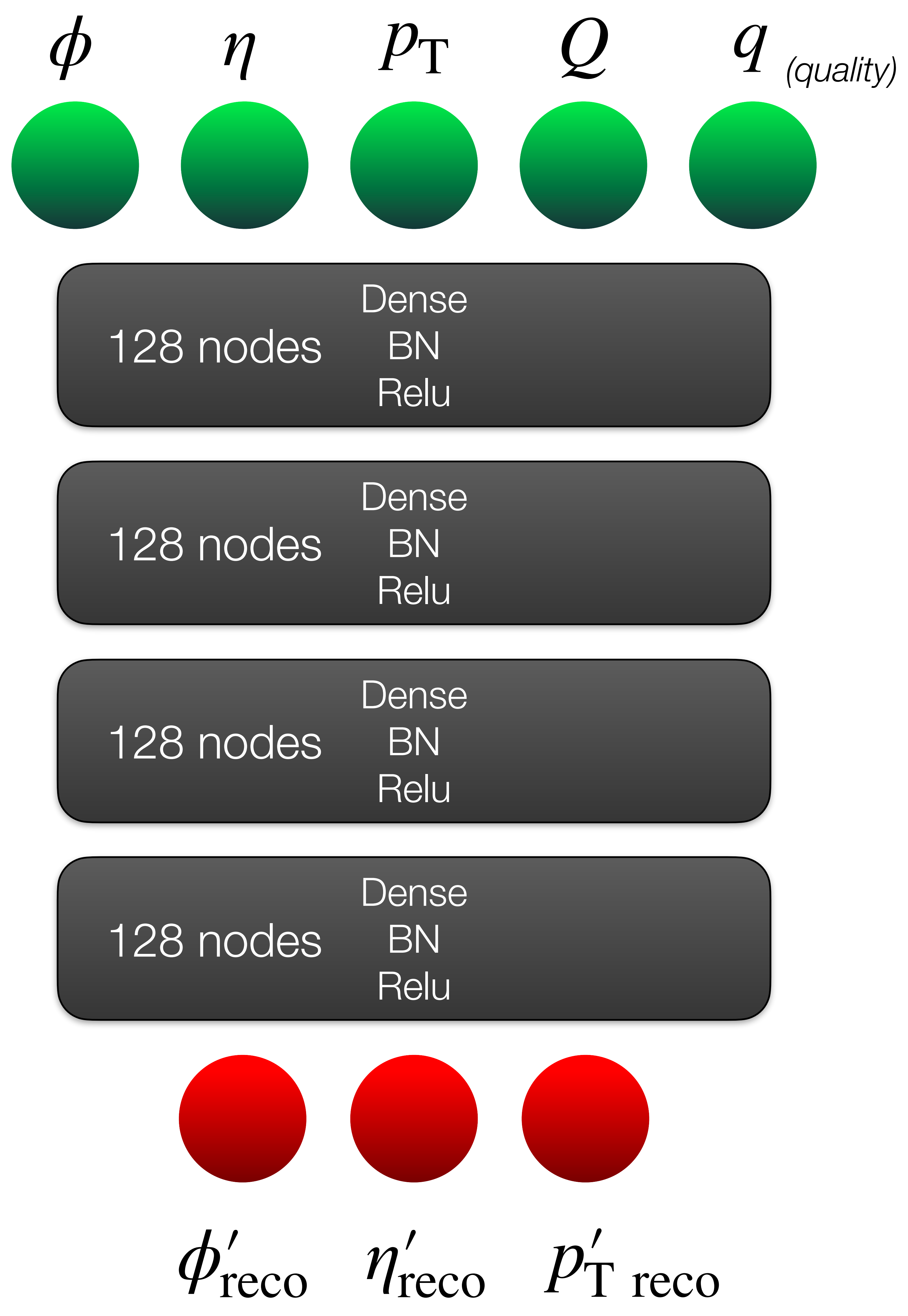
- Use of classical (FF-DNN) neural networks to ‘recalibrate’ L1 information to improve their utility for an online analysis
- Inputs - L1 objects e.g GMT muons:
- Target - Offline fully reconstructed objects



Particle angular position perpendicular to beam



Particle momentum in direction transverse to beam

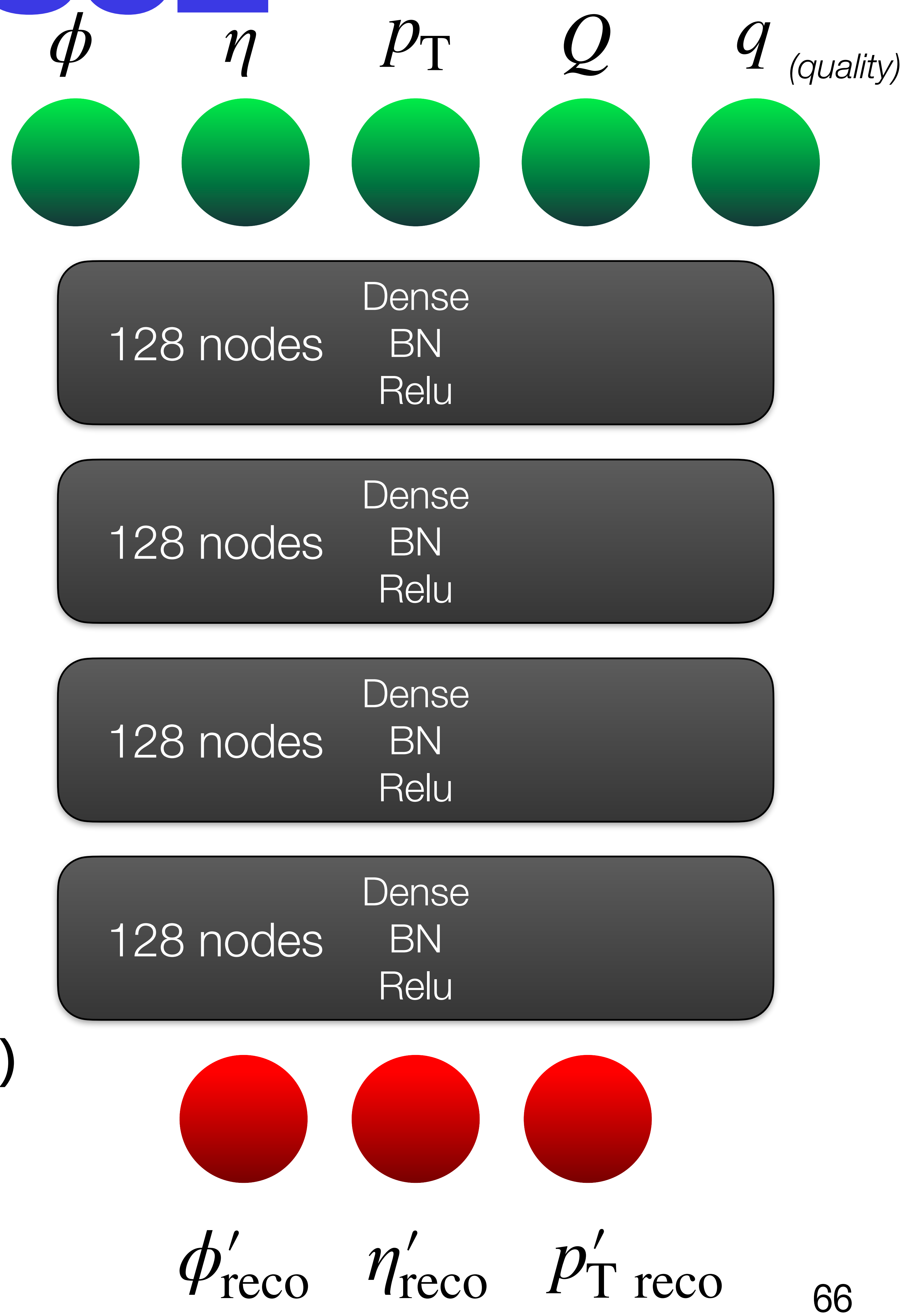


Muon recalibration on SB-852

N DLA clusters	Inference rate	Average latency / muon inference
4 cluster	5.6 MHz	171 ns
2 cluster	2.8 MHz	342 ns
1 cluster	1.4 MHz	683 ns

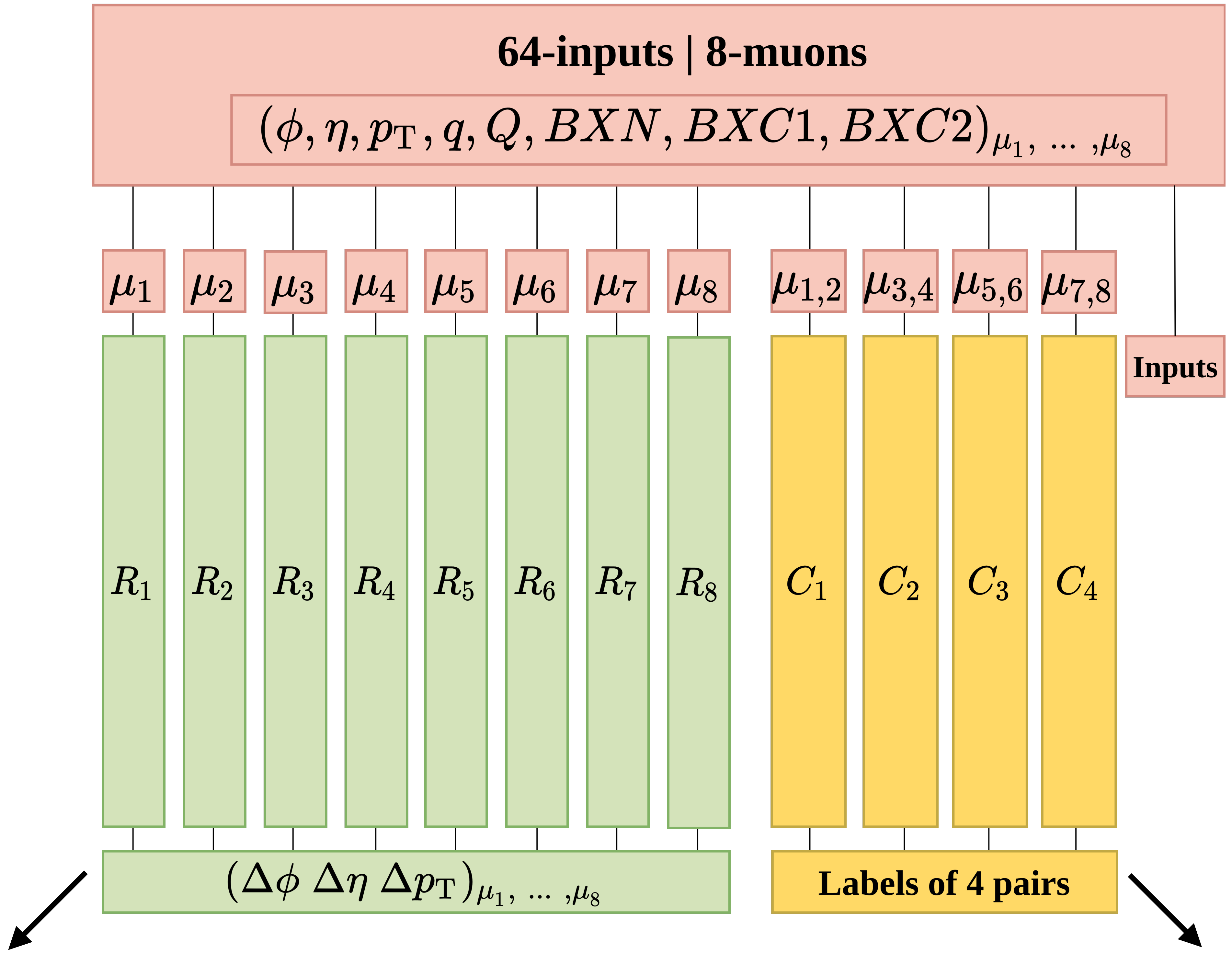
- 4 clusters maximum in VU9P FPGA
- Majority of latency from data/weights transfer RAM/FPGA, batching implemented to remove this bottleneck (batch size 1280)

Precision |hw - Keras sw|	Frac. < 1% diff
Model w/ integer inputs, no batch norm	99%



Fake muon pair classifier

- Network consists of 8 recalibration branches & 4 classification branches
- Trained/tested with Run 3 Zero-bias data



64-inputs | 8-muons

$(\phi, \eta, p_{\mathrm{T}}, q, Q, BXN, BXC1, BXC2)_{\mu_1, \ldots, \mu_8}$

μ_1 μ_2 μ_3 μ_4 μ_5 μ_6 μ_7 μ_8 $\mu_{1,2}$ $\mu_{3,4}$ $\mu_{5,6}$ $\mu_{7,8}$

R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8 C_1 C_2 C_3 C_4

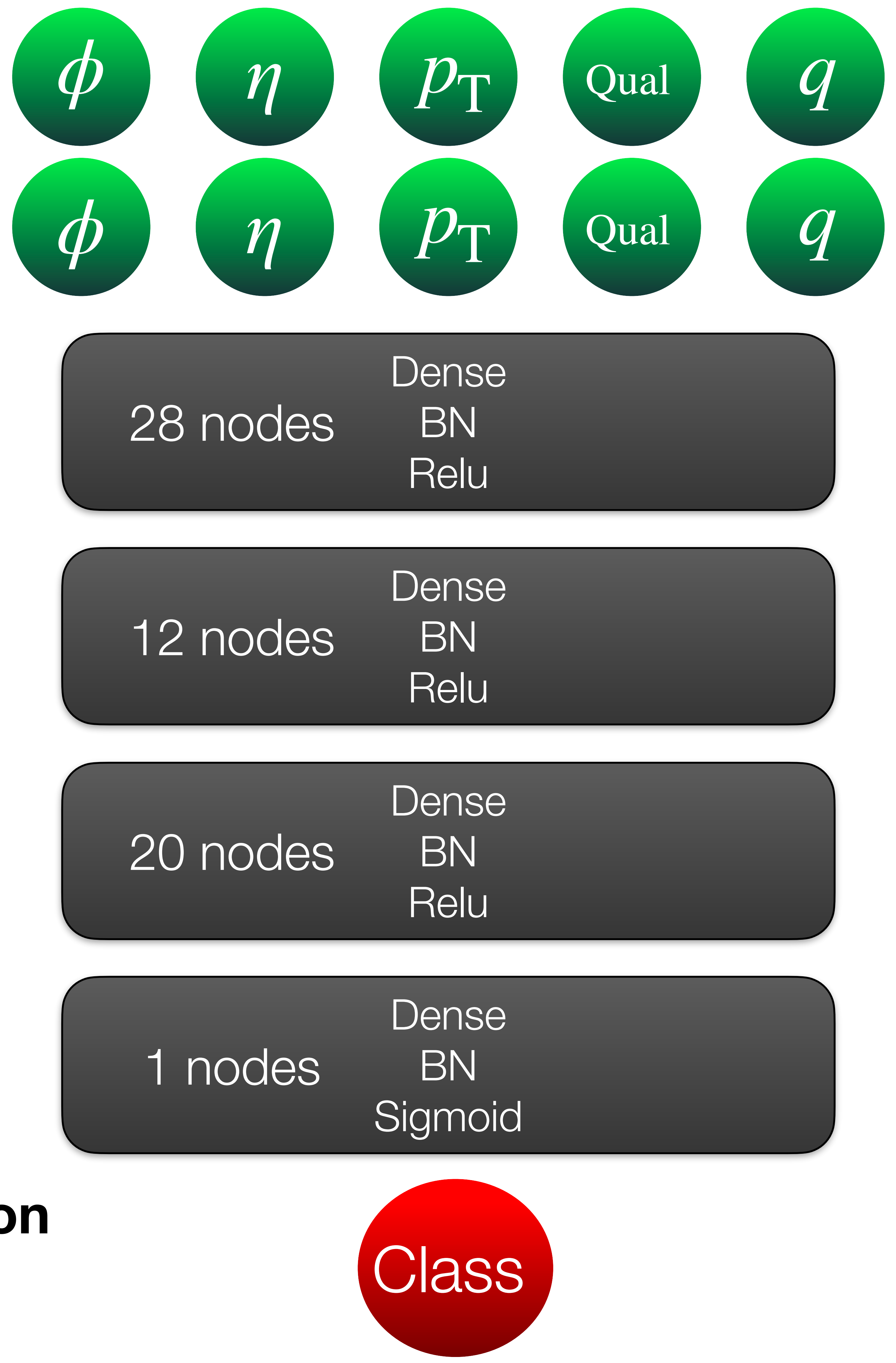
Inputs

$(\Delta\phi\ \Delta\eta\ \Delta p_{\mathrm{T}})_{\mu_1, \ldots, \mu_8}$

Labels of 4 pairs

Recalibration

Classification



ϕ η p_{T} Qual q

ϕ η p_{T} Qual q

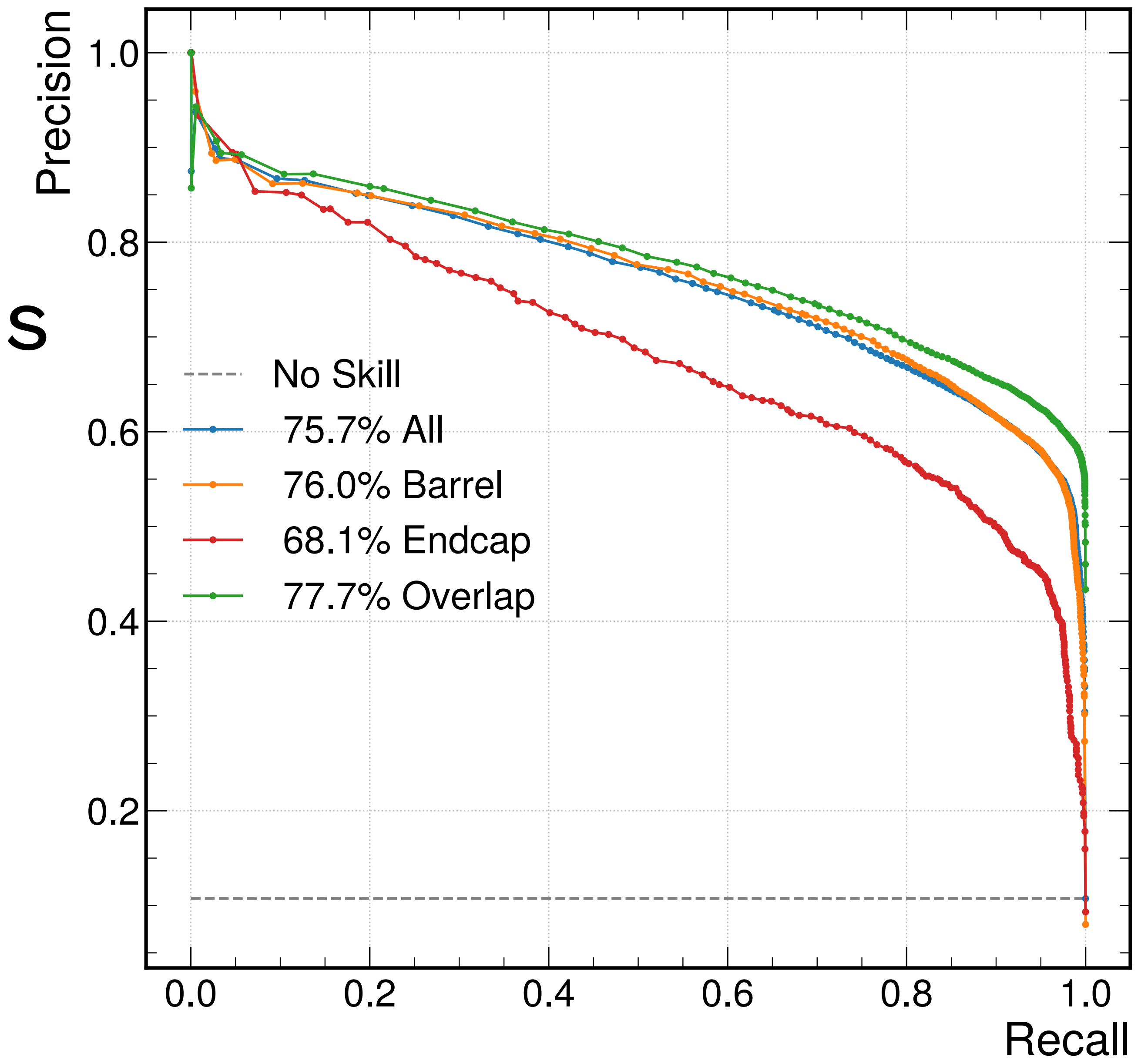
28 nodes — Dense, BN, Relu

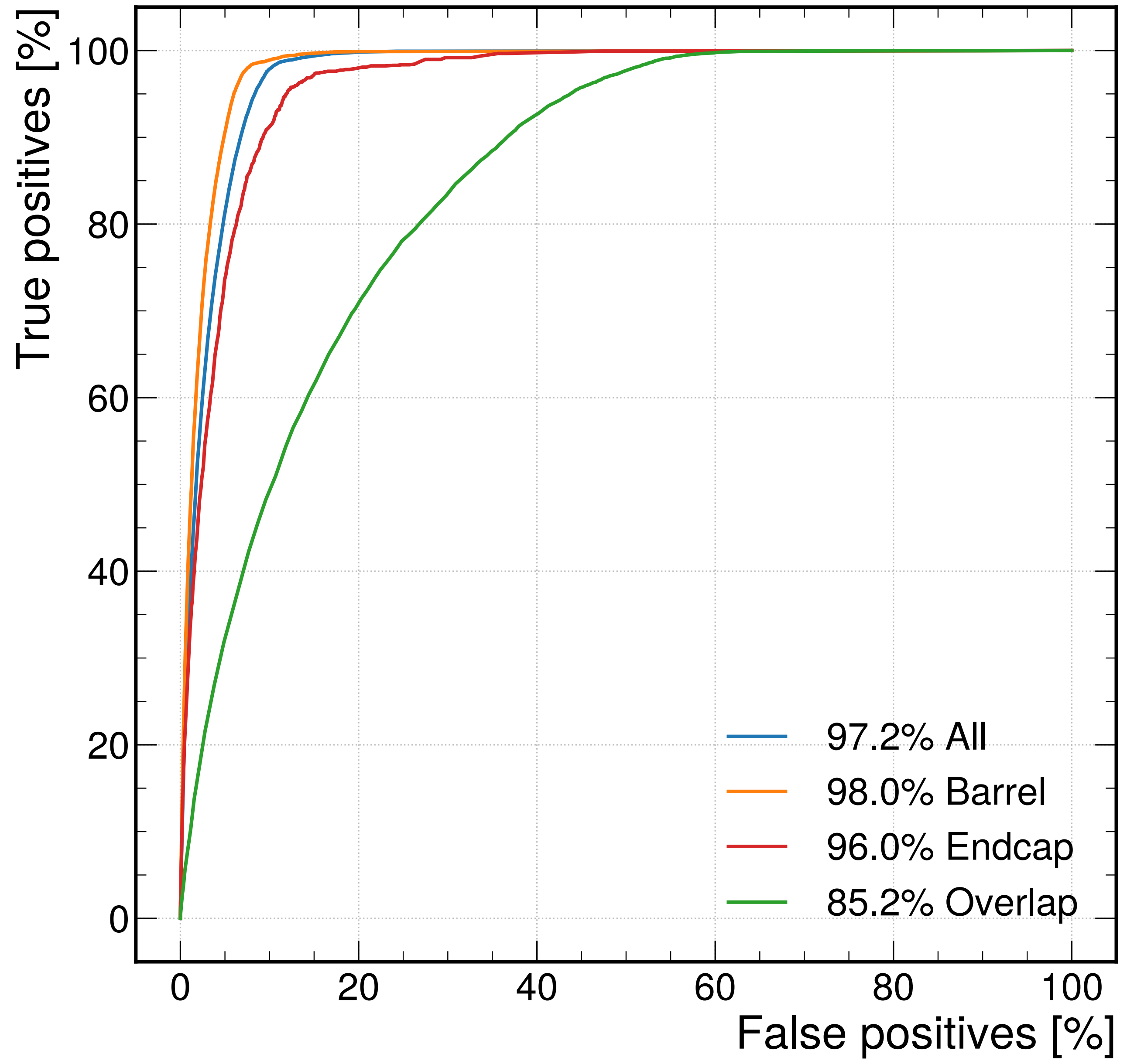
12 nodes — Dense, BN, Relu

20 nodes — Dense, BN, Relu

1 nodes — Dense, BN, Sigmoid

Class

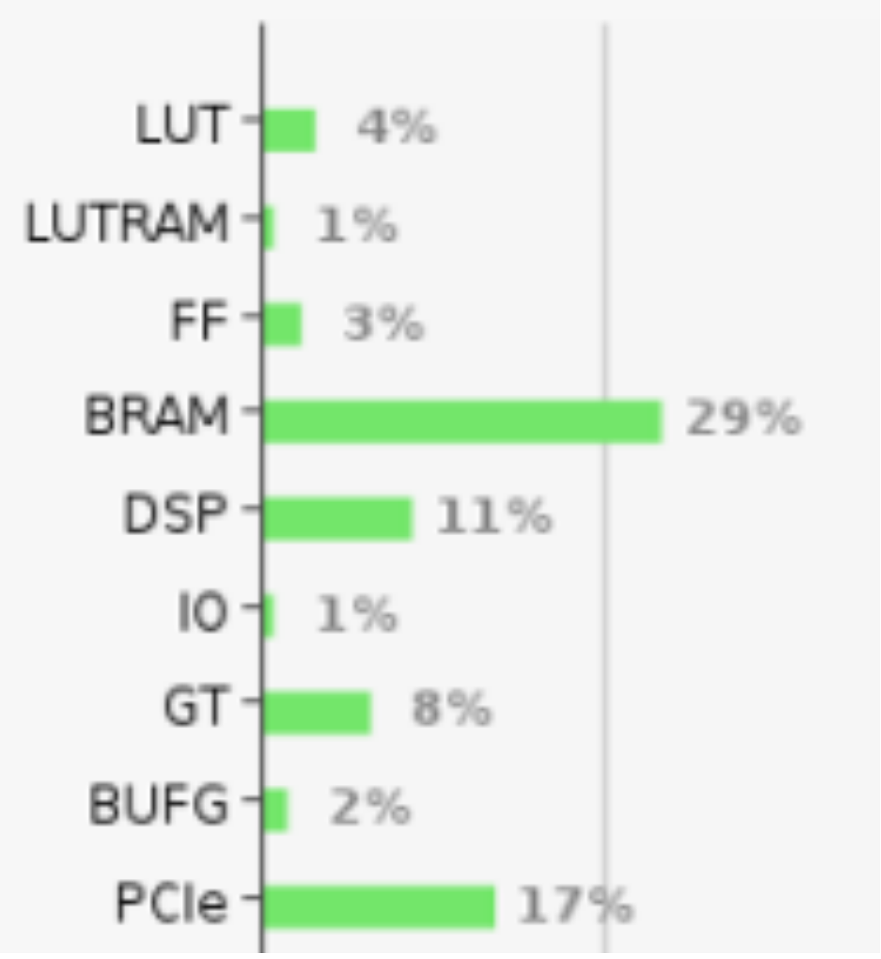


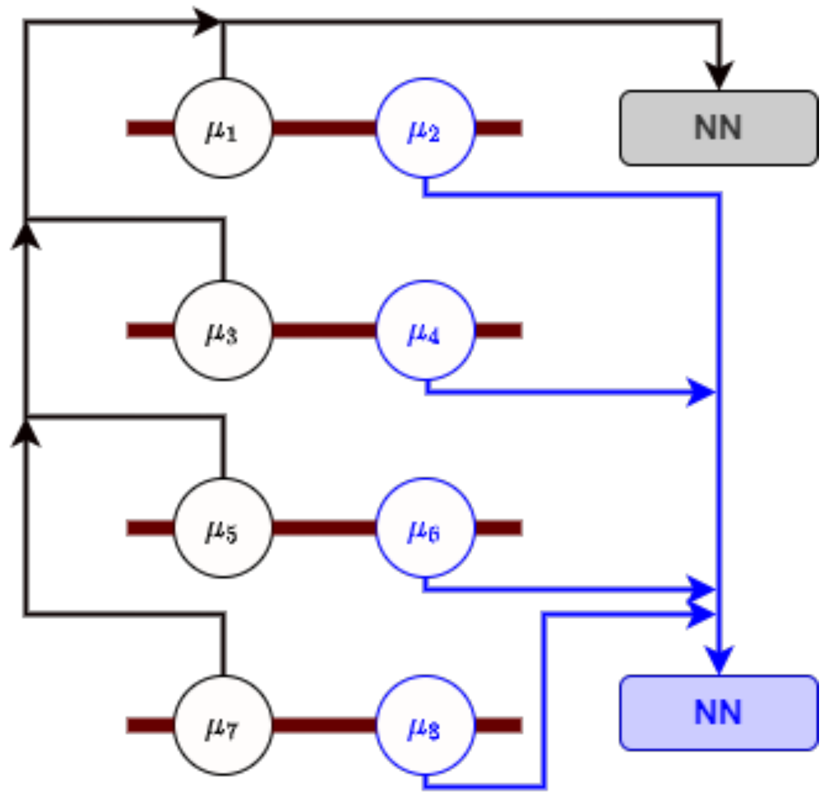


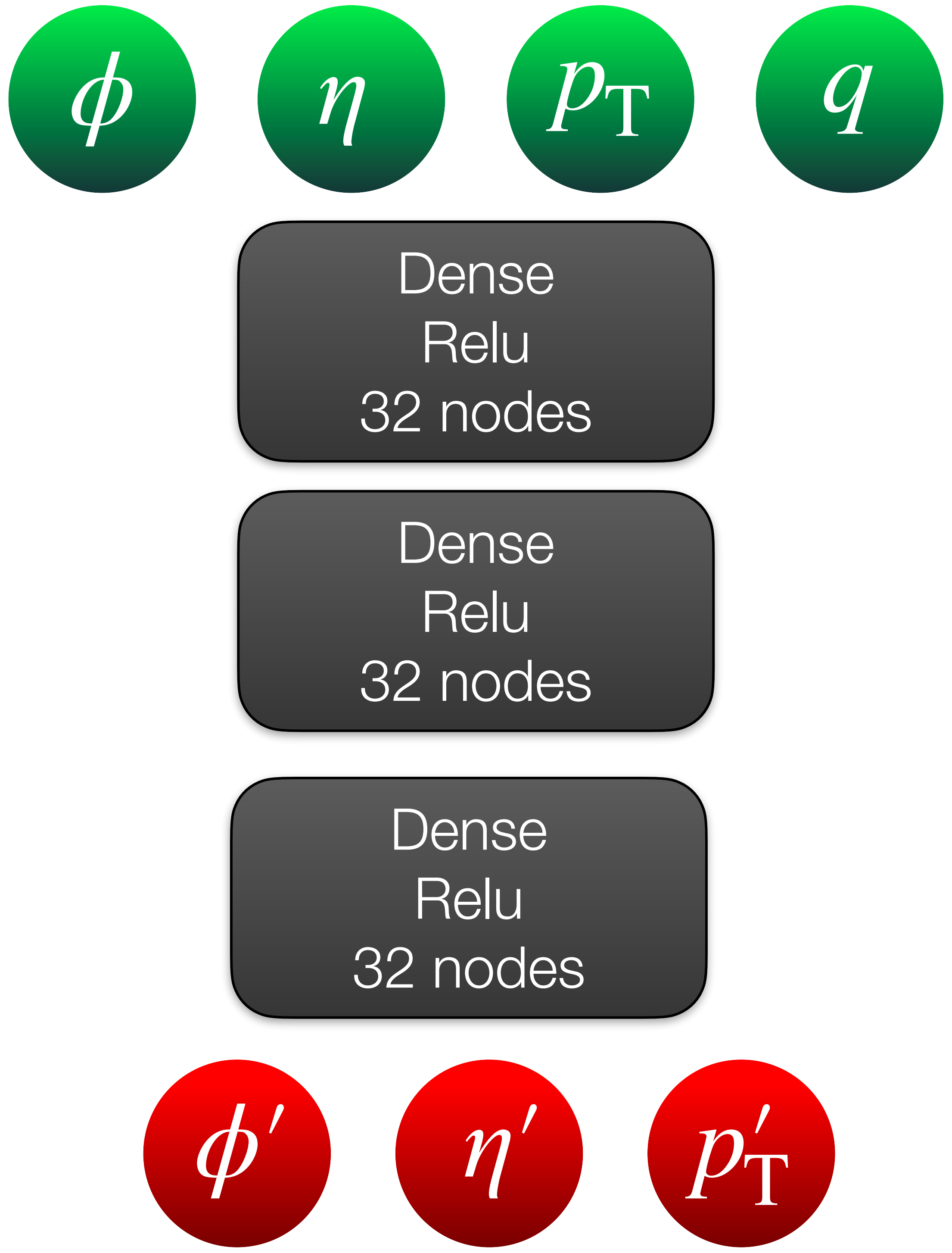
VCU128 - NN w/ hls4ml

- Integrated NN for muon recalibration generated w/ HLS4ML
- Q6.12 precision, pruning factor 0.5
- 2 NN each process 4 muons / BX
- Latency $\lesssim$ 100 ns FIFO latency, can accept 2 muons / clock

VU37P





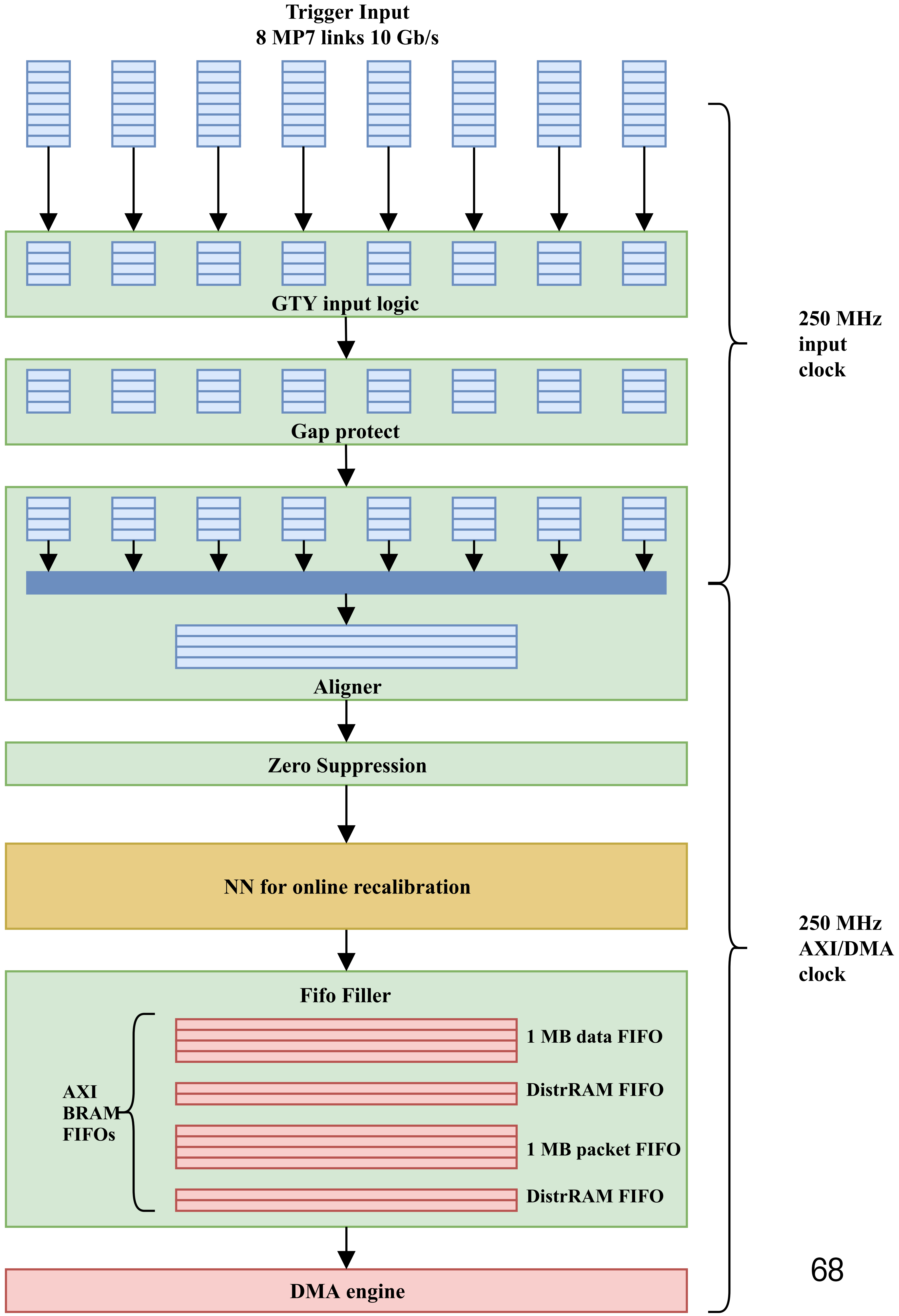

ϕ η p_T q

Dense
Relu
32 nodes

Dense
Relu
32 nodes

Dense
Relu
32 nodes

ϕ' η' p'_T


Trigger Input
8 MP7 links 10 Gb/s

GTY input logic

Gap protect

Aligner

Zero Suppression

NN for online recalibration

Fifo Filler

AXI
BRAM
FIFOs

1 MB data FIFO

DistrRAM FIFO

1 MB packet FIFO

DistrRAM FIFO

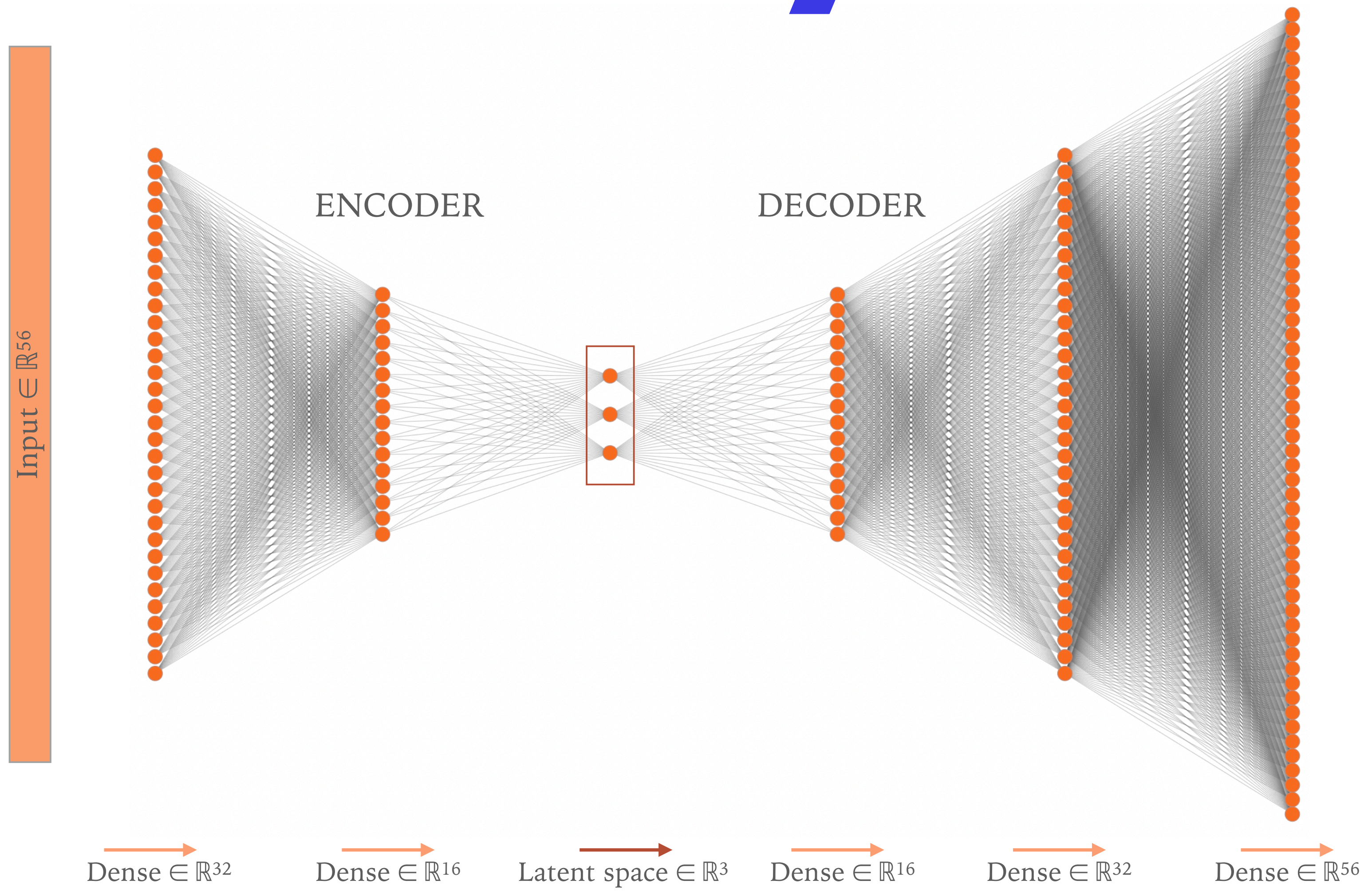
DMA engine

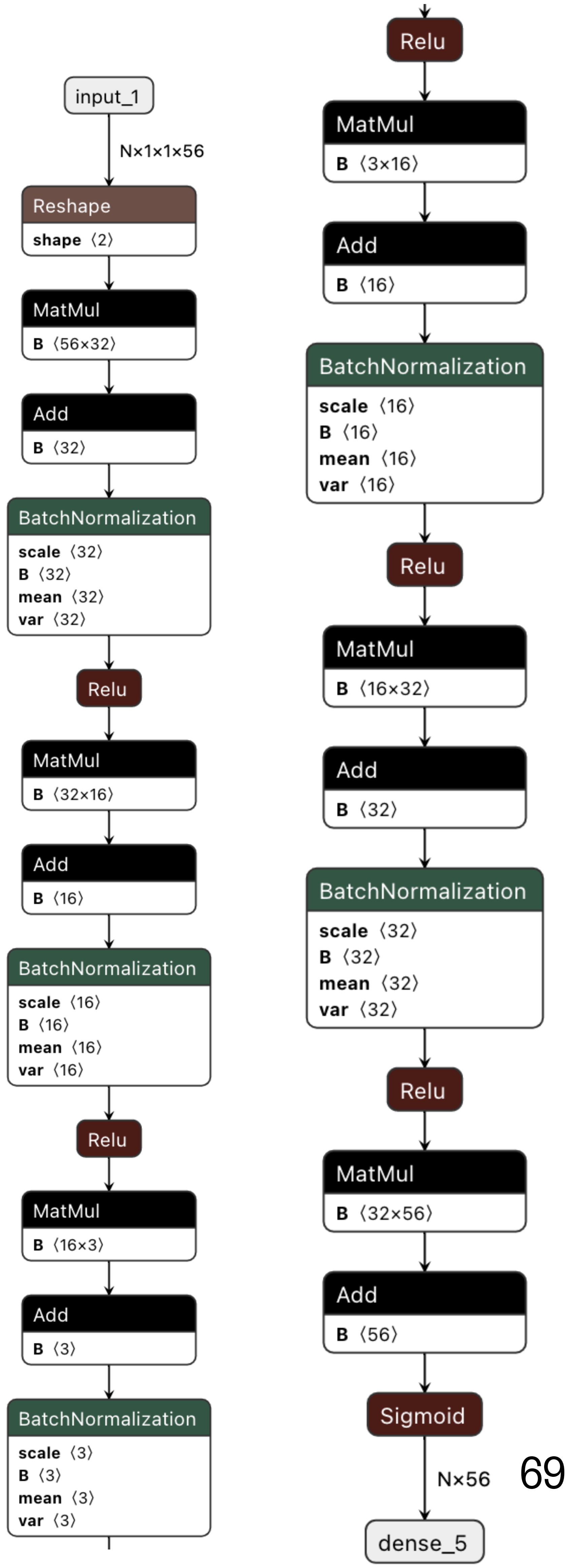
250 MHz
input
clock

250 MHz
AXI/DMA
clock

Auto-encoder for anomaly detection

- Train on Standard Model 'QCD' background
- Inputs: fixed size arrays of up to 10 jets, 4 muons, 4 electrons & Missing energy
  - (each with 3 parameters)
- Test with simulated BSM events
  - e.g new massive vector bosons, unusual Higgs decays





Xilinx VU9P	Latency (ns)	Clock freq. (MHz)	DSPs (%)	LUTs (%)	Flip Flops (%)
HLS4ML (8bit)*	48	200	20	8	0.4
MDLA	625	250	Four clusters, full chip used		

* Resources HLS estimates only.

See: T. Arrestad et al, LHC physics dataset for unsupervised New Physics detection at 40 MHz, arXiv:2107.02157

Summary

- ML is a rich and exciting field of research, constantly inventing new, more powerful techniques
- At the same time, device developers are supporting the growth of ML with faster, more parallel processors, and devices designed specifically for ML
- Deploying ML into the realtime processing for Trigger and DAQ is becoming increasingly possible and relevant

Thomas James (CERN): t.j@cern.ch

Special thanks to Sioni Summers (CERN) for last year's slides

Links and additional reading

[1] https://fastmachinelearning.org/hls4ml

[2] https://github.com/fastmachinelearning/hls4ml-tutorial

[3] https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html

[4] https://software.intel.com/en-us/articles/accelerating-neural-networks-with-binary-arithmetic

[5] https://arxiv.org/abs/1804.06913

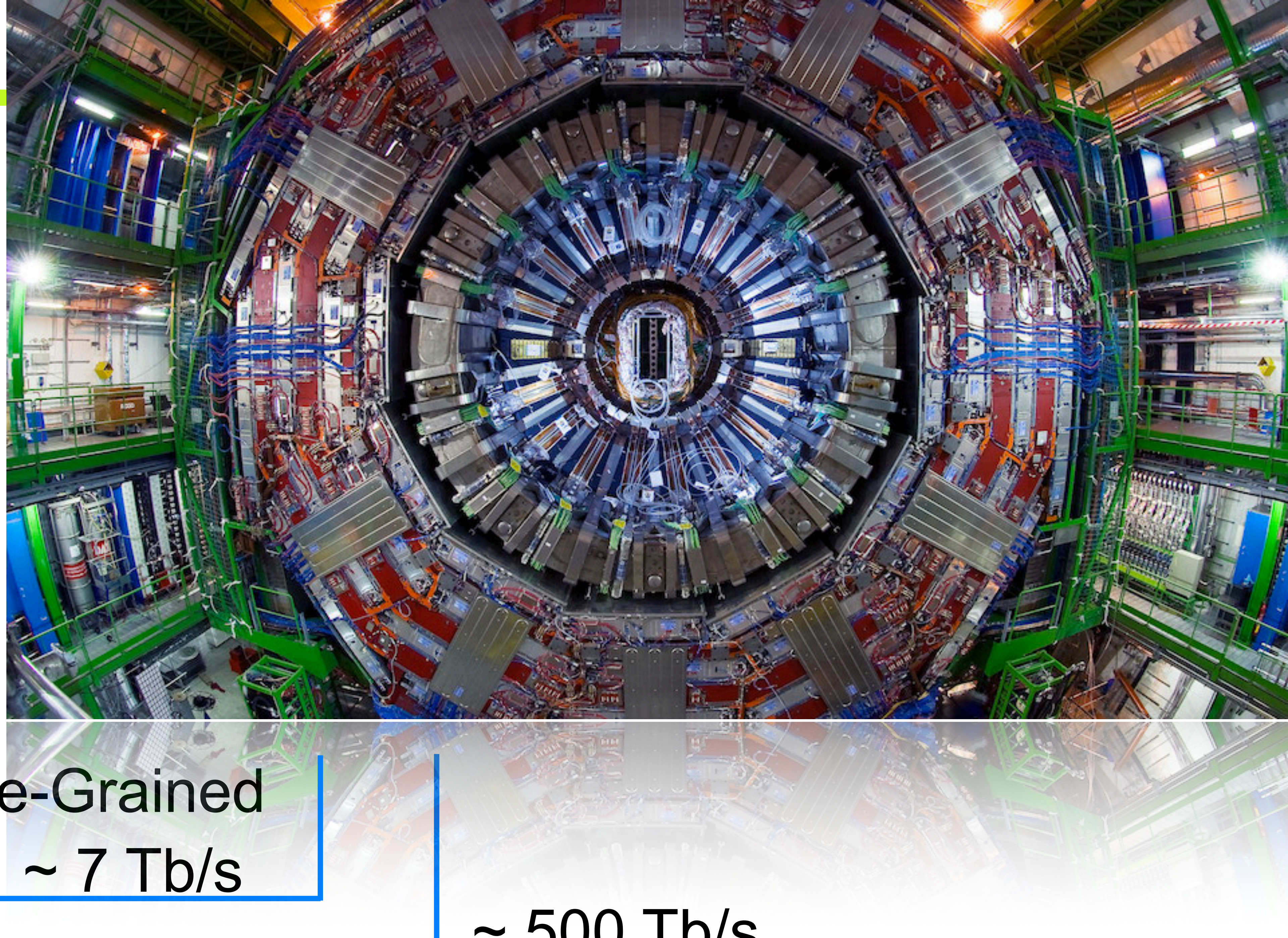
[6] https://www.nature.com/articles/s42256-021-00356-5

[7] github.com/fastmachinelearning/qonnx

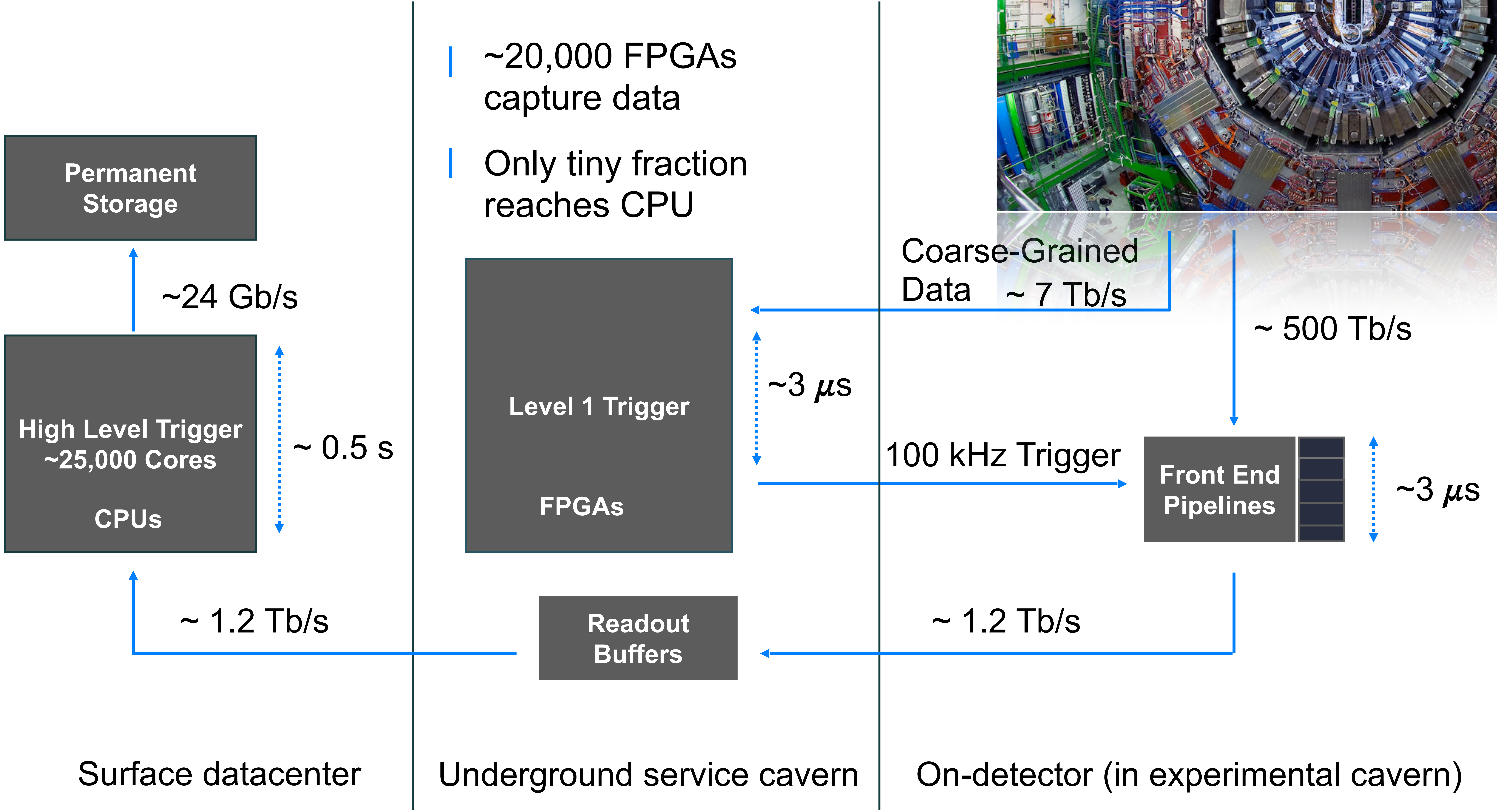
[8] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[9] https://www.eidosmedia.com/blog/technology/machine-learning-size-isn-t-everything

CMS 2015 - 2023



- ~20,000 FPGAs capture data
- Only tiny fraction reaches CPU



Permanent Storage
~24 Gb/s
High Level Trigger ~25,000 Cores
CPUs
~ 0.5 s
~ 1.2 Tb/s
Level 1 Trigger
FPGAs
~3 μs
Readout Buffers
Coarse-Grained Data
~ 7 Tb/s
~ 500 Tb/s
100 kHz Trigger
Front End Pipelines
~3 μs
~ 1.2 Tb/s
Surface datacenter
Underground service cavern
On-detector (in experimental cavern)