Machine Learning
(for Trigger and Data Acquisition)

ISOTDAQ
Catania
21/6/2022
Sioni Summers (CERN)

Contents

- Introduction to Machine Learning

    - Neural networks - recap basics and different types

- Machine Learning in Trigger and DAQ

    - Examples

    - Fast ML: Hardware and software for ML

    - FPGA ML: hls4ml

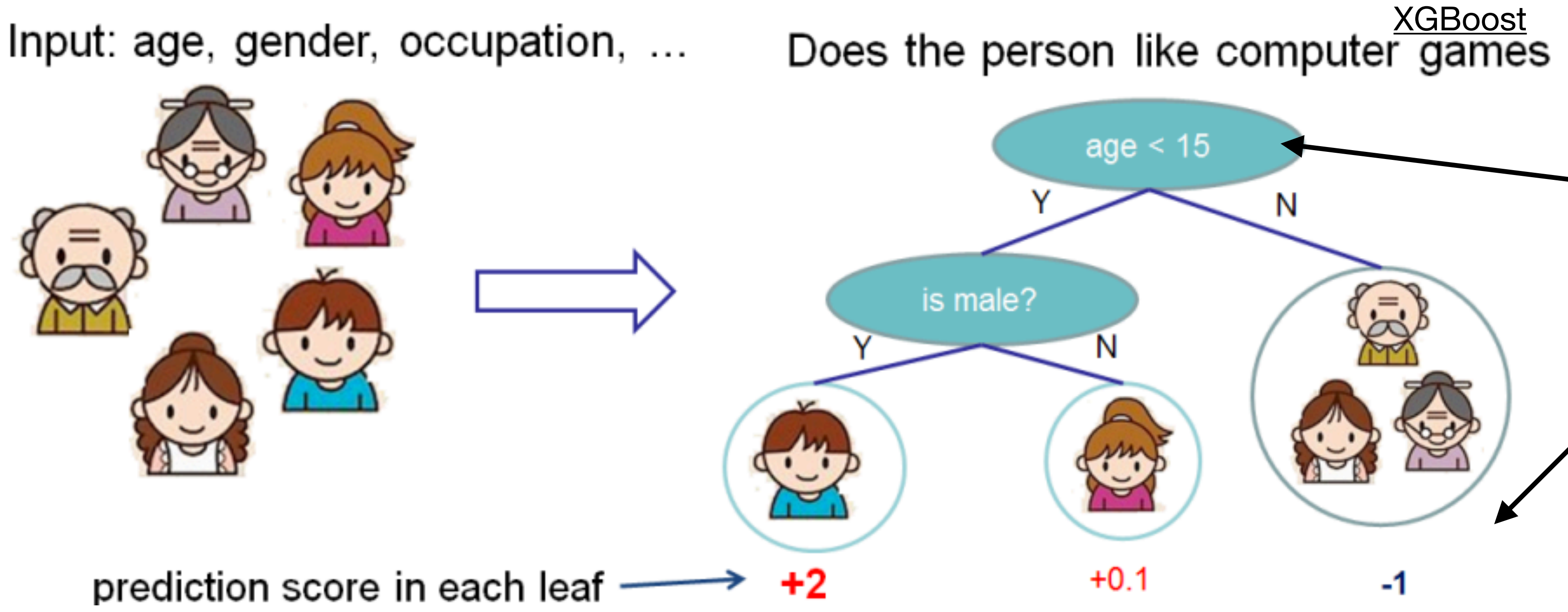
Motivation

> "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 machine learning

- Build models which learn patterns from data to later make predictions on unseen data
- e.g. predict whether a person will like computer games from characteristics

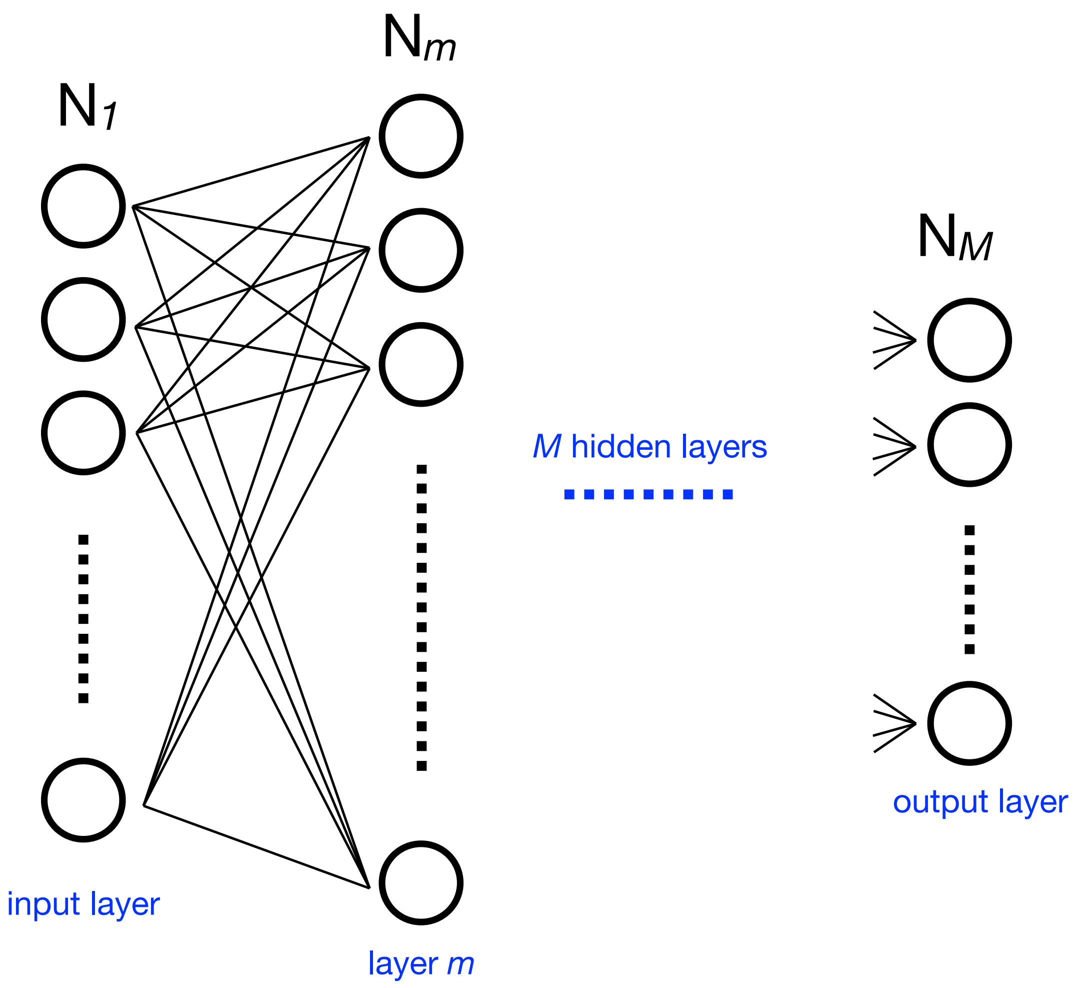

XGBoost
Input: age, gender, occupation, ...
Does the person like computer games
age < 15
Y
N
is male?
Y
N
prediction score in each leaf
+2
+0.1
-1

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

- 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

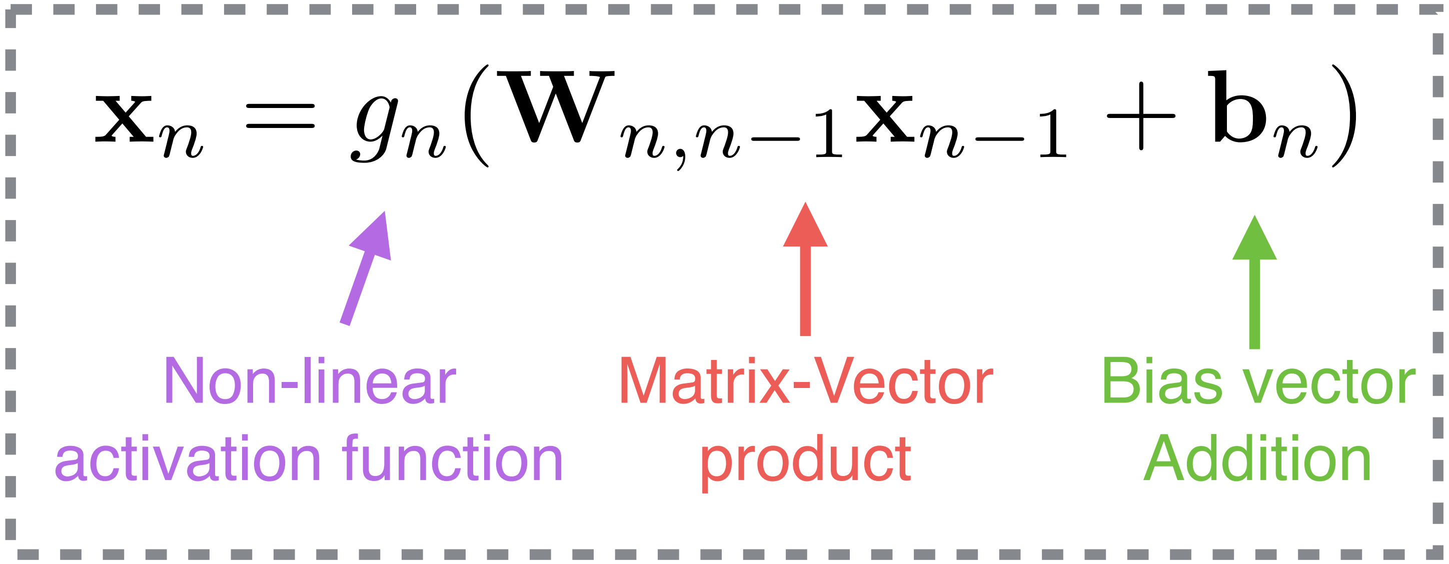
Neural Networks

- Model 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 prediction 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



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

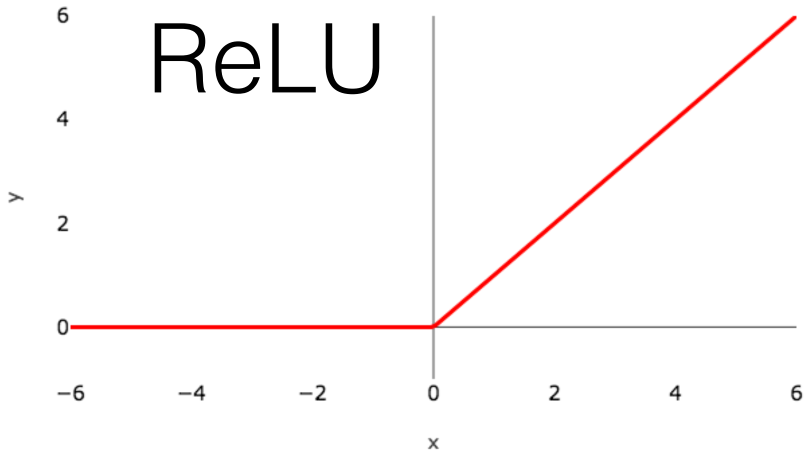


$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

Non-linear activation function

Matrix-Vector product

Bias vector Addition



- Without the activation function, we’re just doing linear transformations of our variables

- The actual values of these weights and biases are learned from data during training…

Training with Gradient Descent

- When training with *supervised learning* we start with a neural network with randomised weights and a collection of labelled *training data*

- We need to evaluate the performance of our network, using a loss function, e.g. mean squared error:

$$L(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

- y is the true value of the labelled example, i. ŷ is the value predicted by the neural network

- Would like to minimise the 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!
    - 'lr' is learning rate

$$w_j = w_j - lr\,\partial \frac{L}{\partial w_j}$$

Tools / Frameworks

- You don’t need to write all these algorithms yourself!
- 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`

K Keras
TensorFlow
PyTorch
pandas
$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$
scikit learn
ONNX
uproot
dmlc
XGBoost

A made up 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)
```

Convolutional Neural Networks

- The previous slides showed specifically *Fully Connected* or *Dense* Neural Networks
- Many other topologies exist for different types of problems
- 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

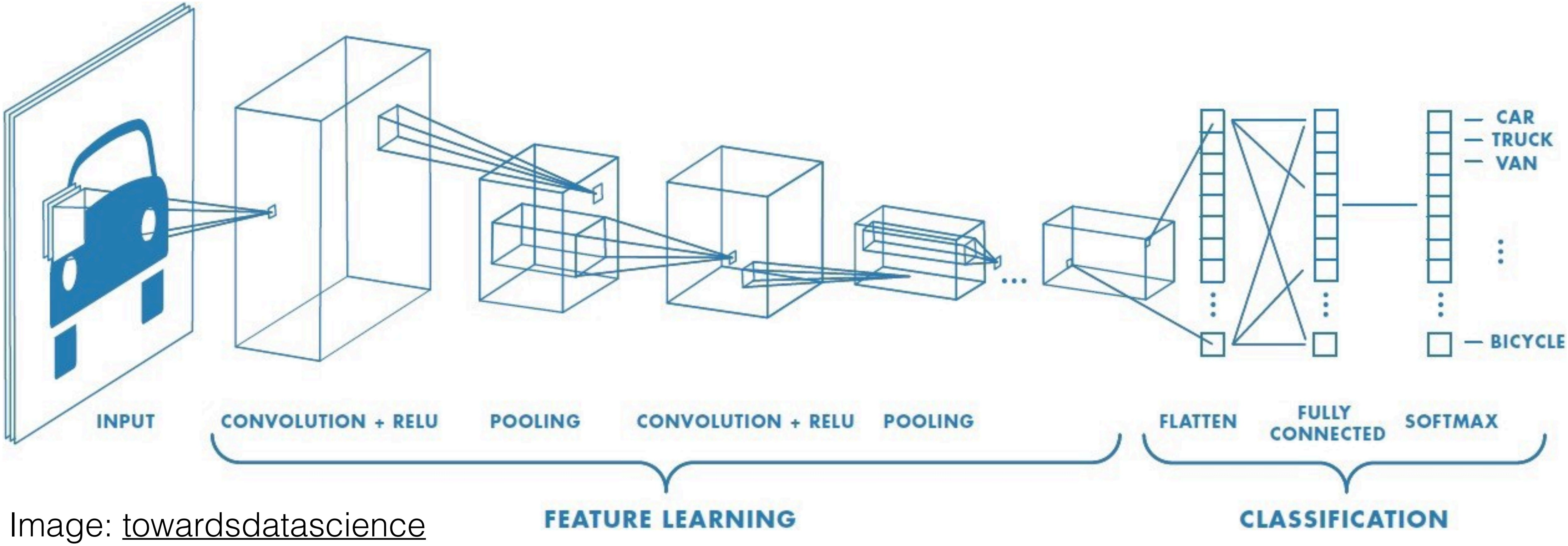


Image: towardsdatascience

Recurrent Neural Networks

- A Neural Network with a built in ‘memory’
- Used where there is ordered data, e.g. time series, natural language processing
- There are a few different flavours: Long Short Term Memory (LSTM), Gate Recurrent Unit (GRU)

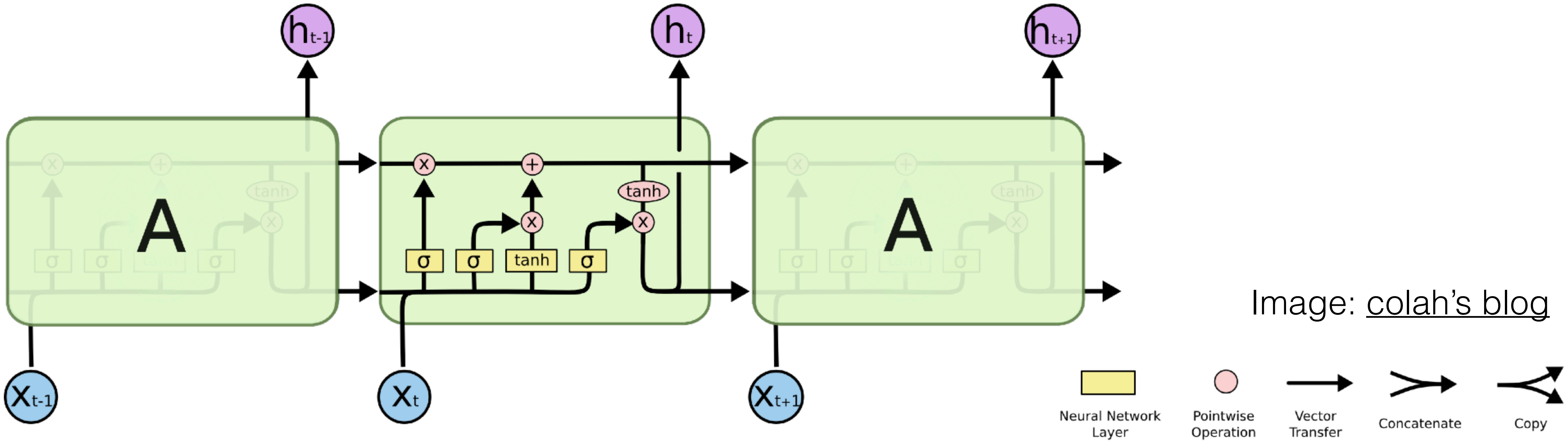
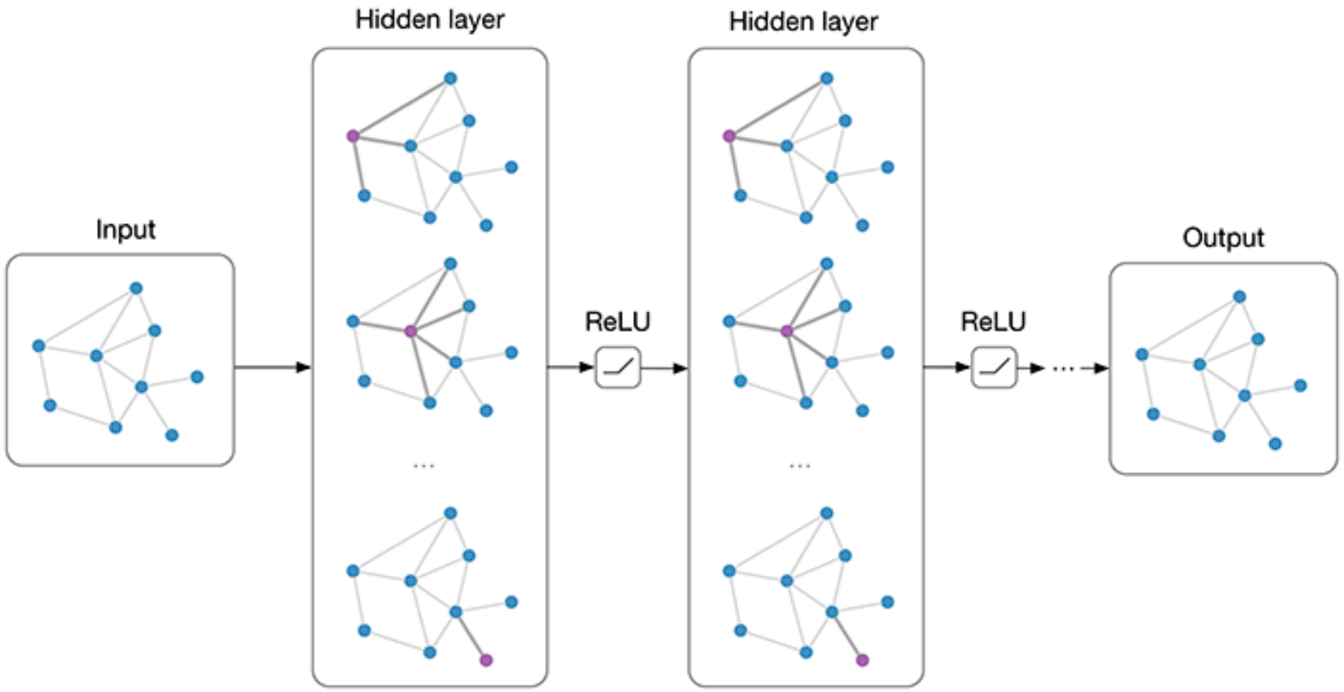


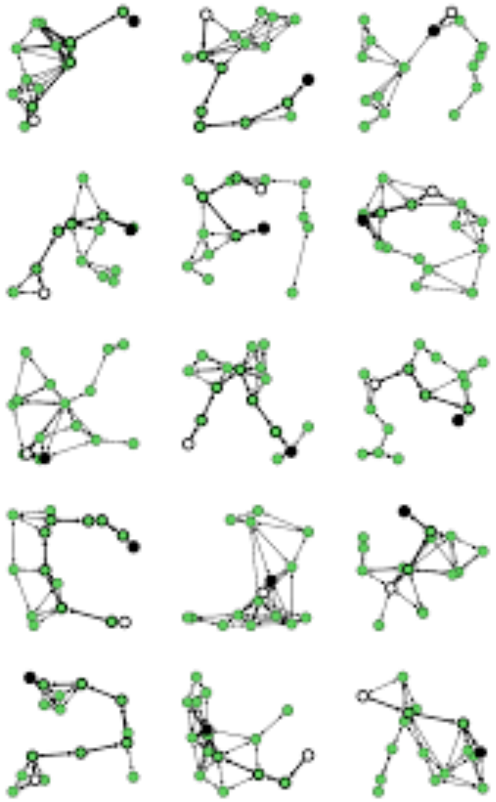
Image: colah’s blog

- 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

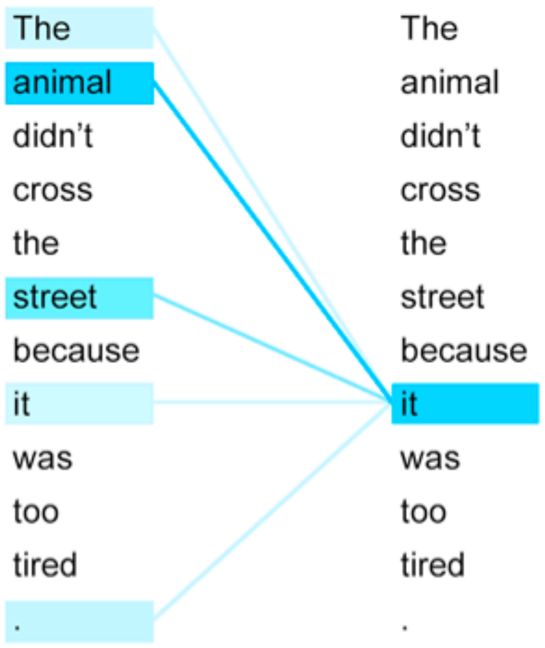
- We've seen NNs suitable for applying on 'high level features' (Fully Connected), images (Convolutional), and time series (Recurrent)
- Graph networks are well suited to problems described by graphs of vertices and edges
- Cluster / classify data not only according to its coordinates, but its neighbourhood
- 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, e.g. tracking, calorimetry in high pileup; hierarchical type problems, e.g. tracking, jets

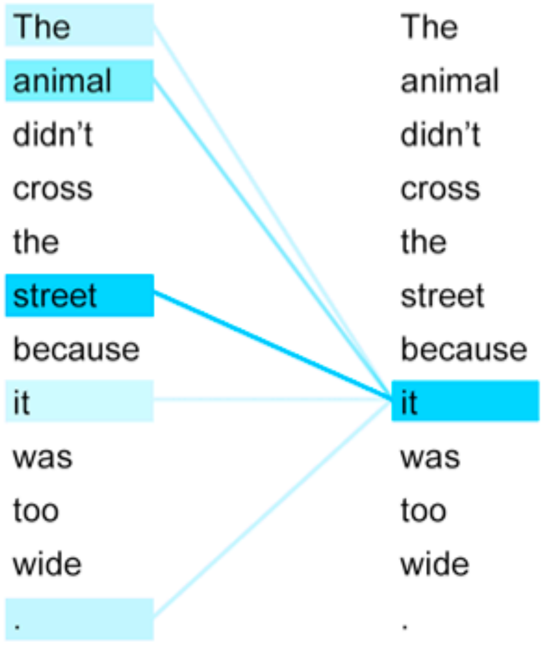


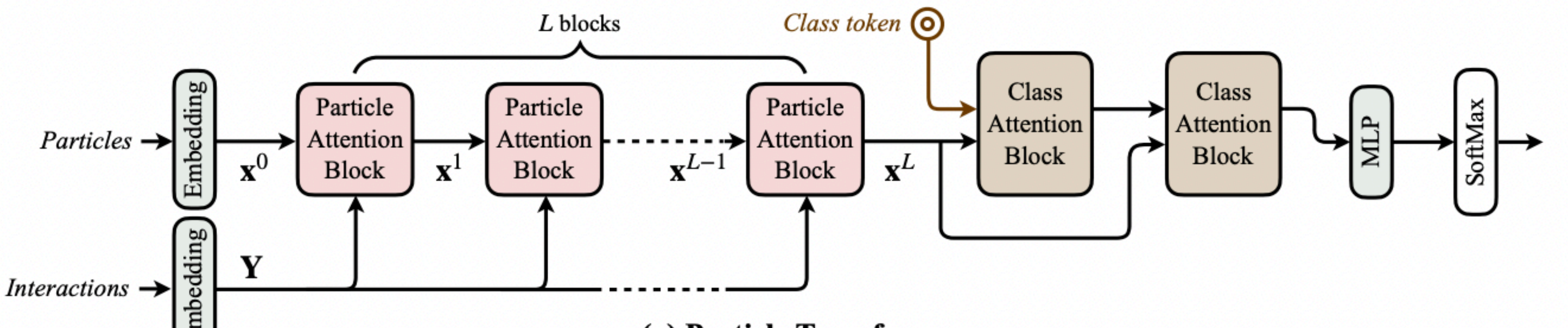


Transformers

- Sequence-to-sequence type problems
  - The big Natural Language Processing (NLP) models like BERT and GPT3
  - These big networks have 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)





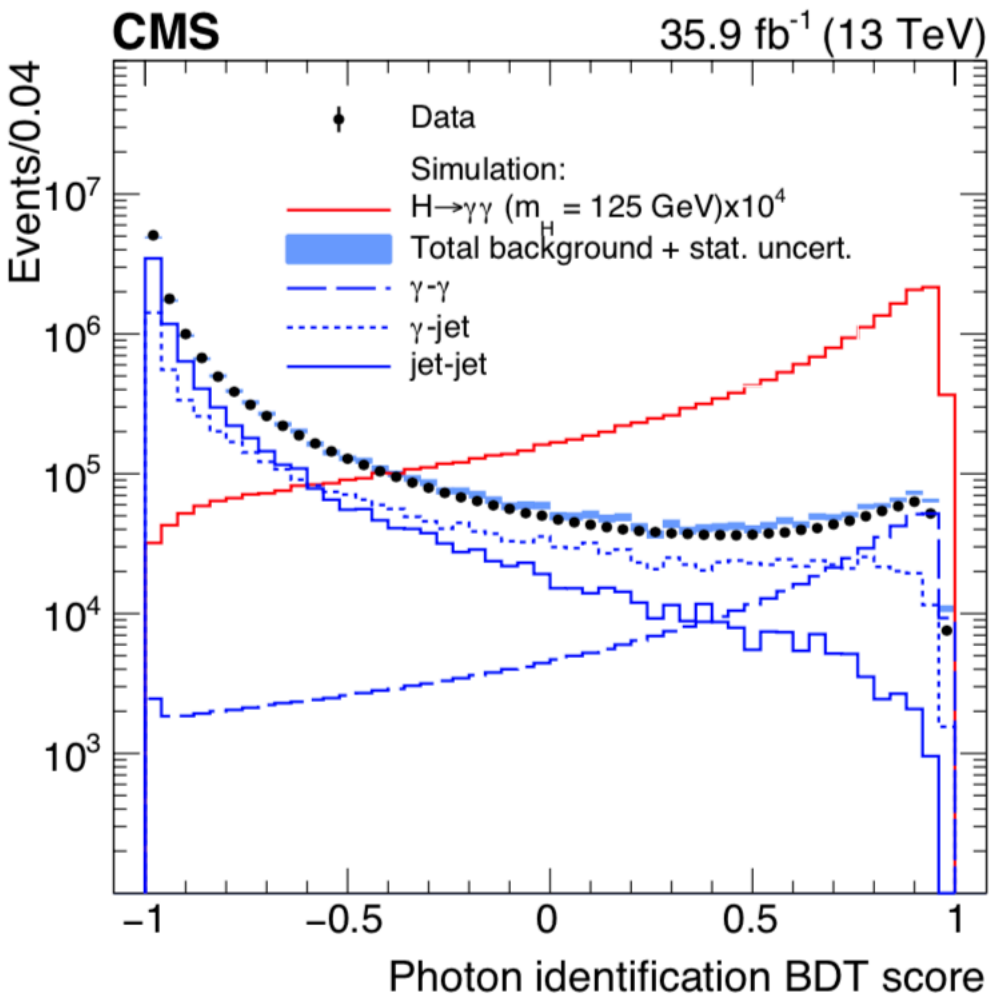


(a) Particle Transformer

Source

BDTs for Higgs

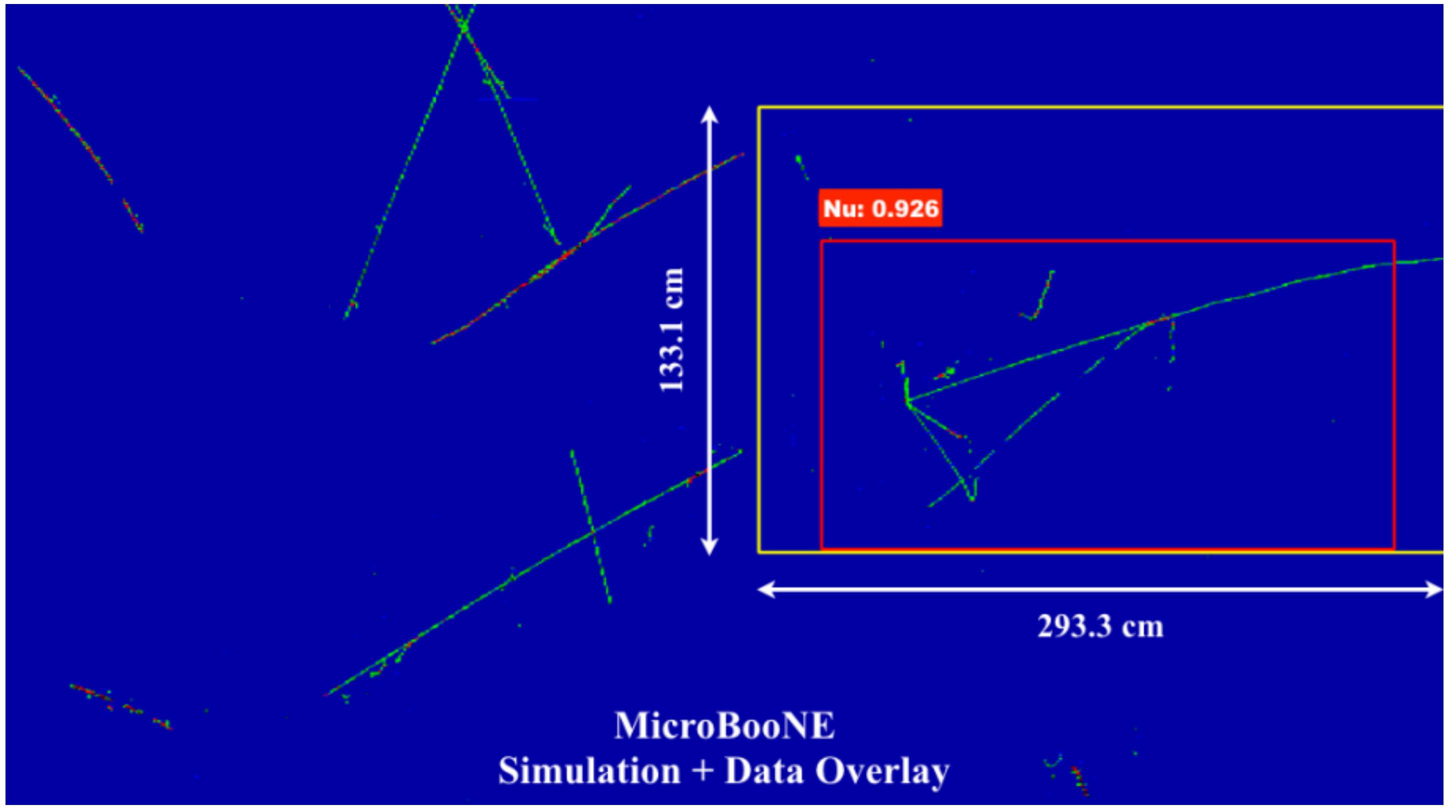
- Several BDTs involved in the analysis of Higgs boson decay to two photons using high-level variables
    - e.g. particle mass, η, isolation
- To separate signal photons from background (photons from jets)
- Choosing the most likely vertex for the photons (they are neutral, so no tracking)
- A diphoton 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

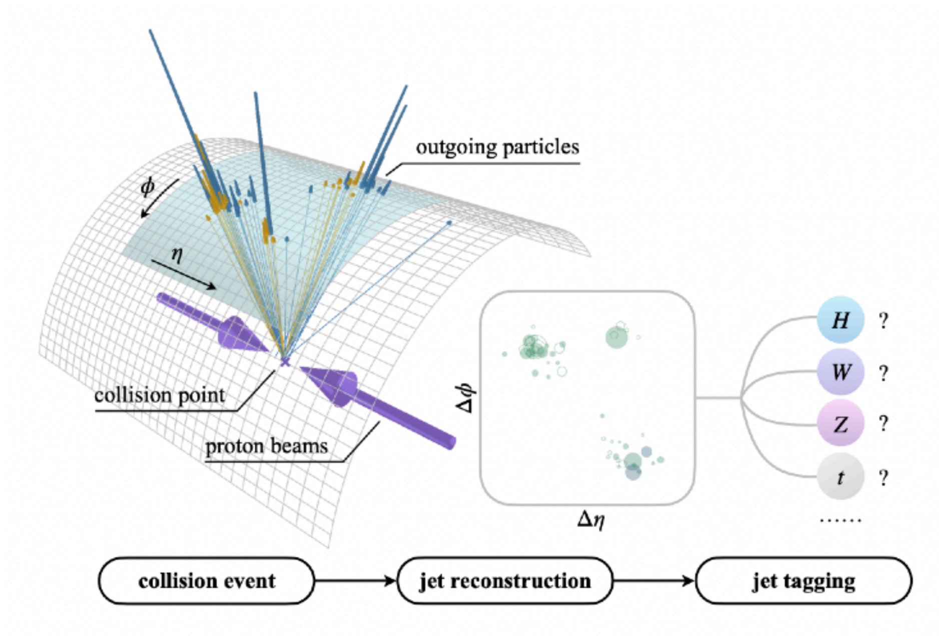


arxiv:1611.05531

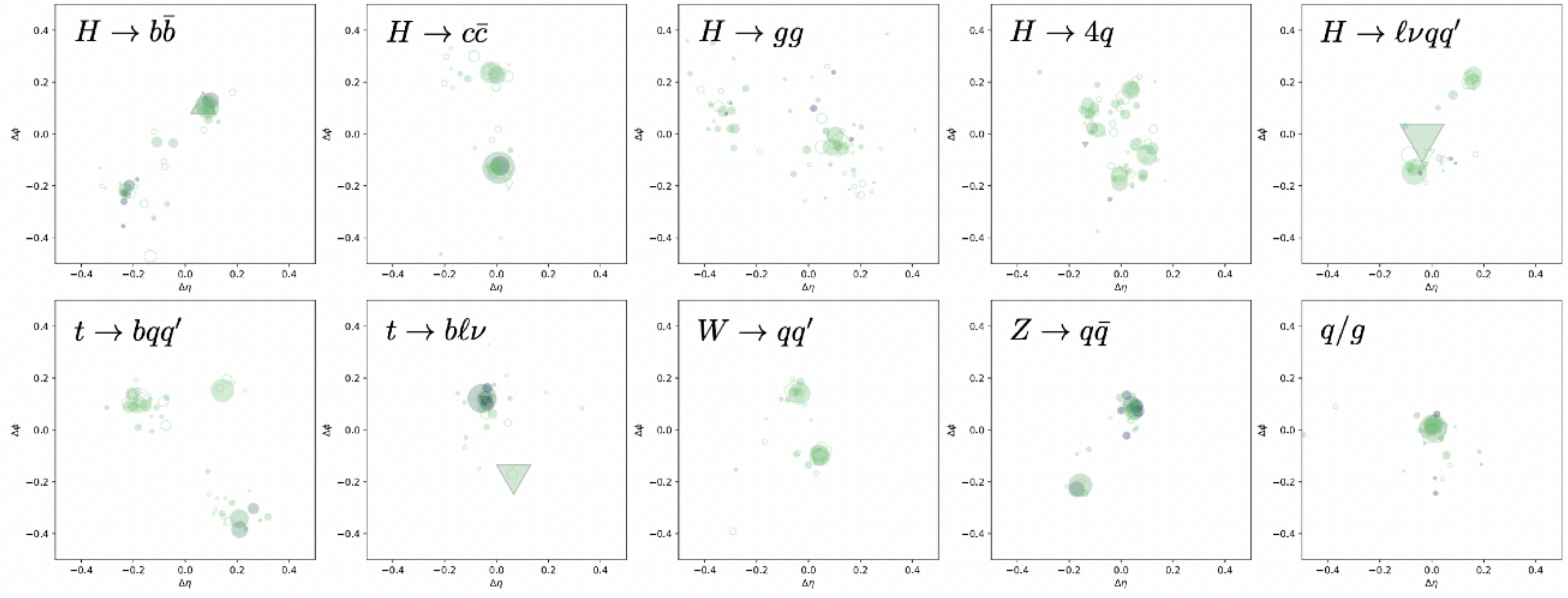
- Yellow box is ‘truth’ box containing all charge deposits from simulated interactions
- Red is bounding box predicted by CNN

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 pT ordered) -> Recurrent NN or Transformer
  - Represent particles as a graph (point cloud with connections) -> Graph NN

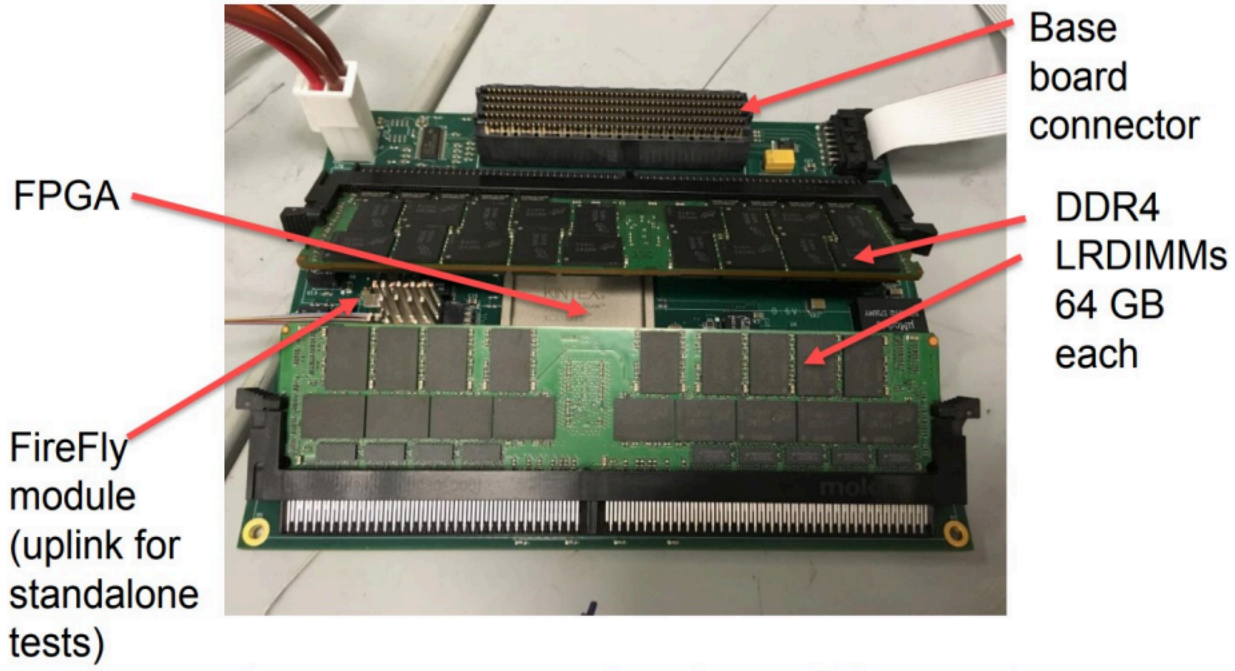


arXiv:2202.03772



Examples of ML in TDAQ

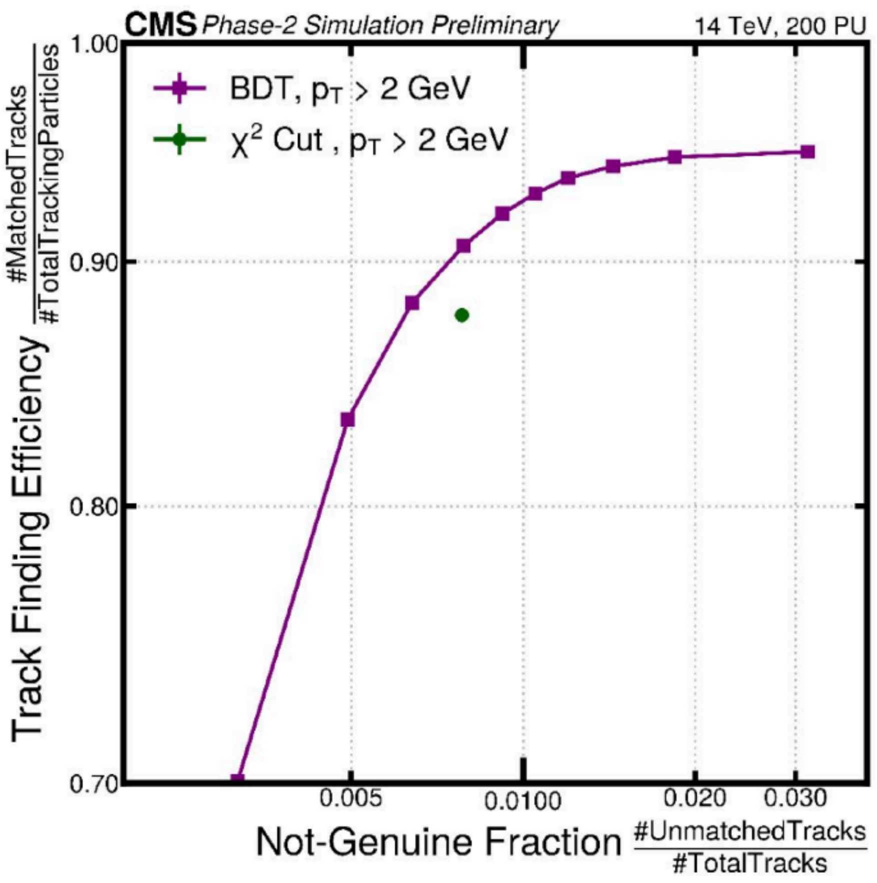
- **CMS Level 1 Trigger Endcap Muon** system uses a BDT to fit the muon momentum from hits in the muon stations
    - Complicated geometry and magnetic field makes an ML solution useful
- Deployed using a 'large LUT' implemented in DDR on a mezzanine card to the FPGA
- BDT is evaluated for every possible input, with the output written at that position in the LUT



- 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

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

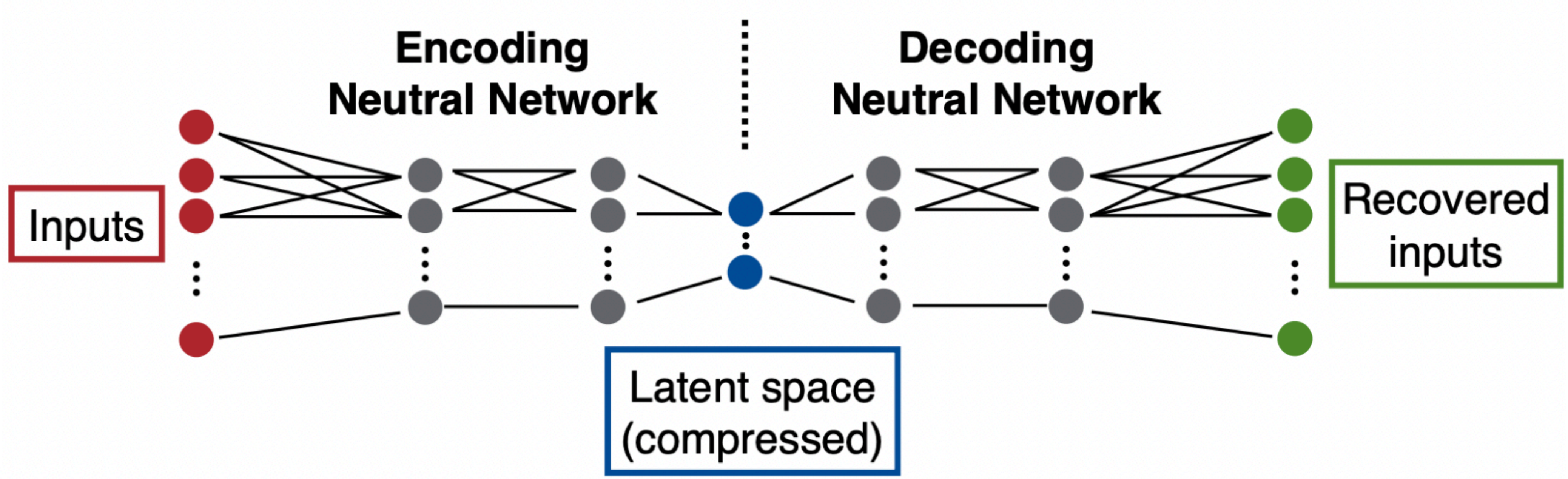


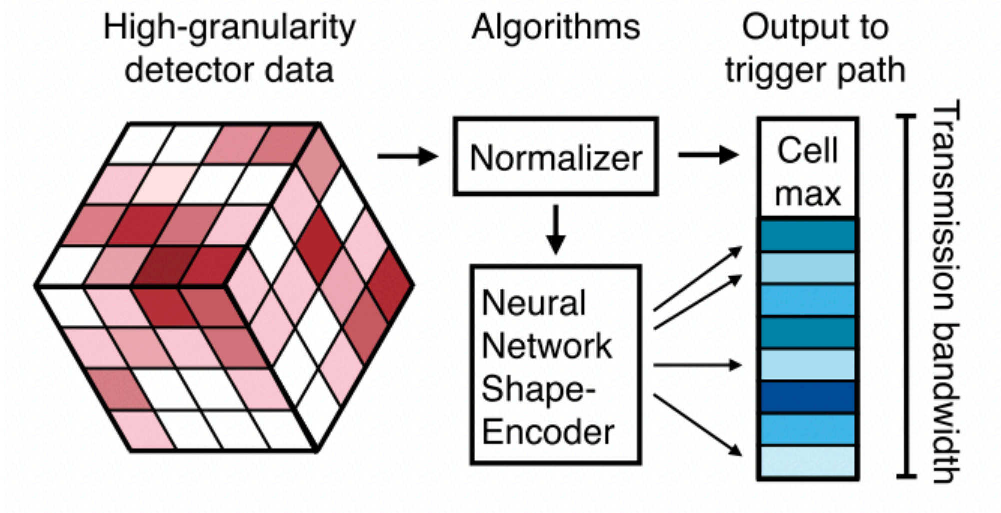
On-detector ML

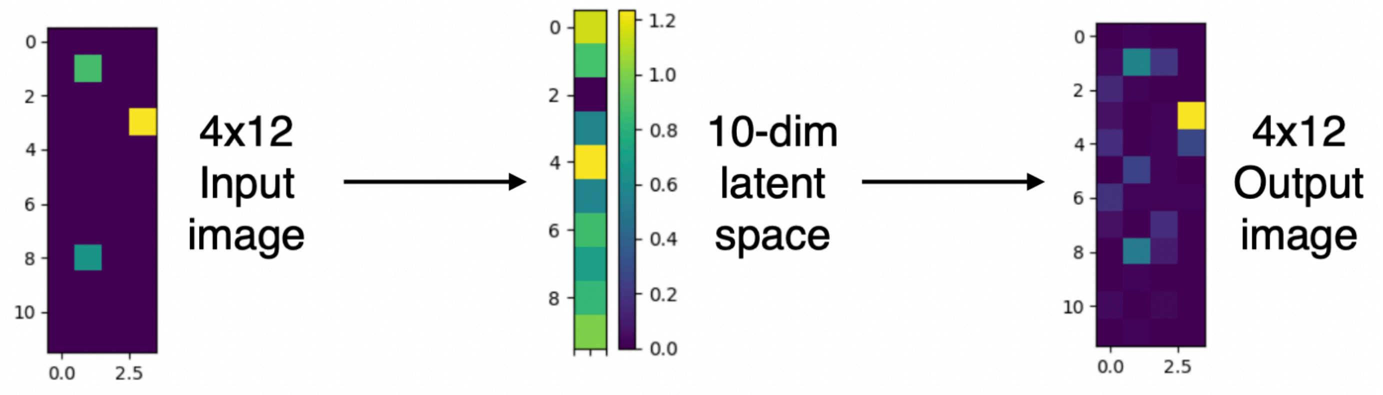
- ECON-T ASIC for CMS High Granularity Calorimeter
  - Compress data to be sent to trigger FPGAs with an AutoEncoder, decode off detector

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

On detector → lpGBT → Off detector (trigger)

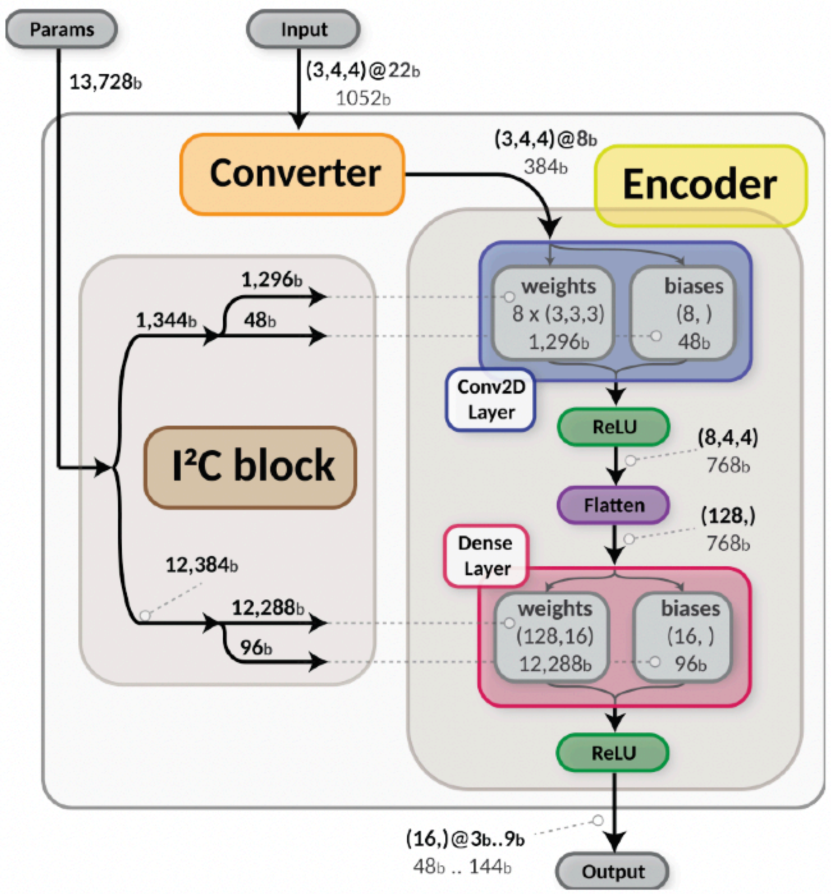


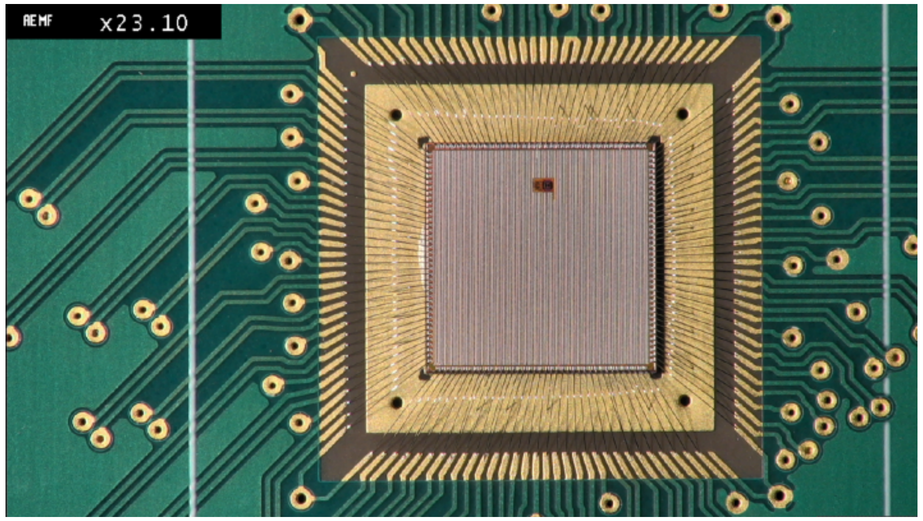


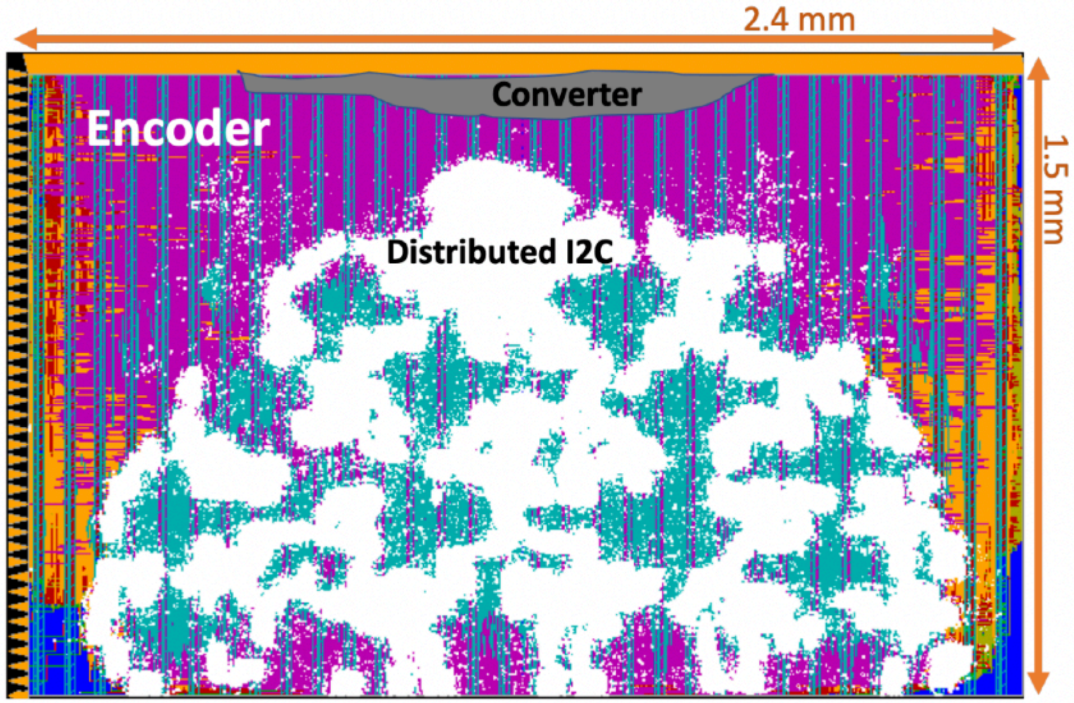


On-detector ML

- 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

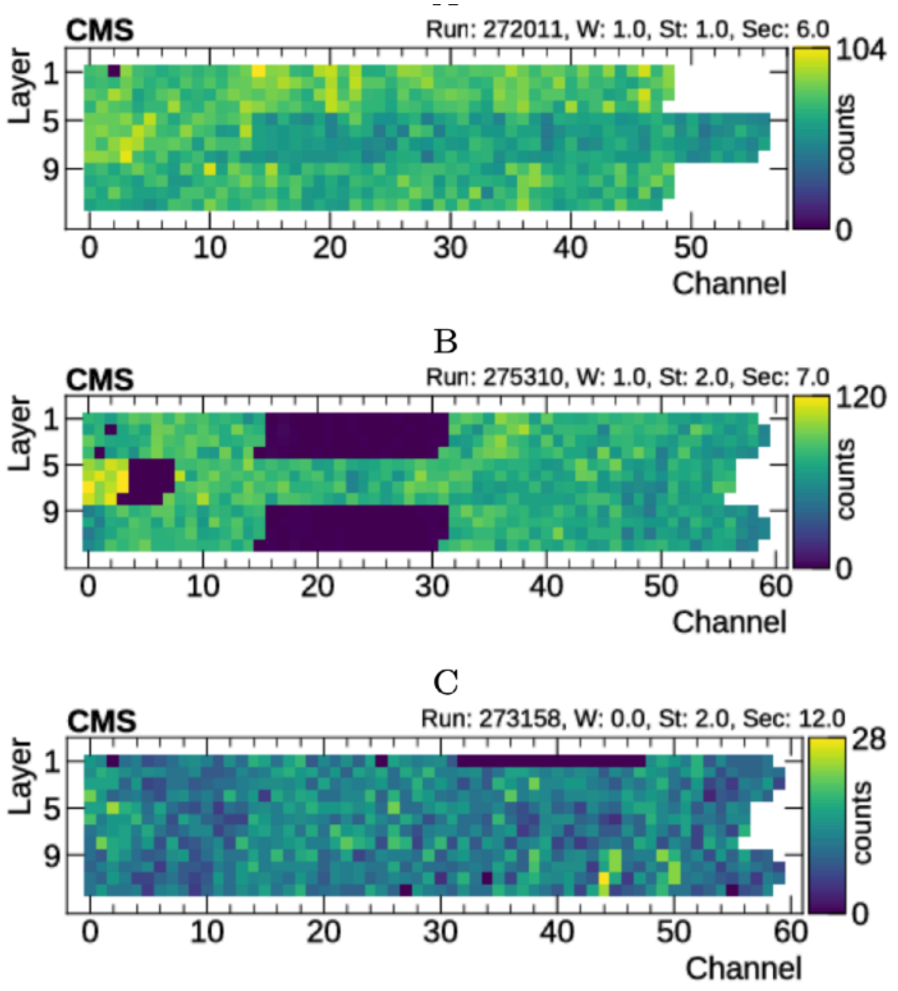


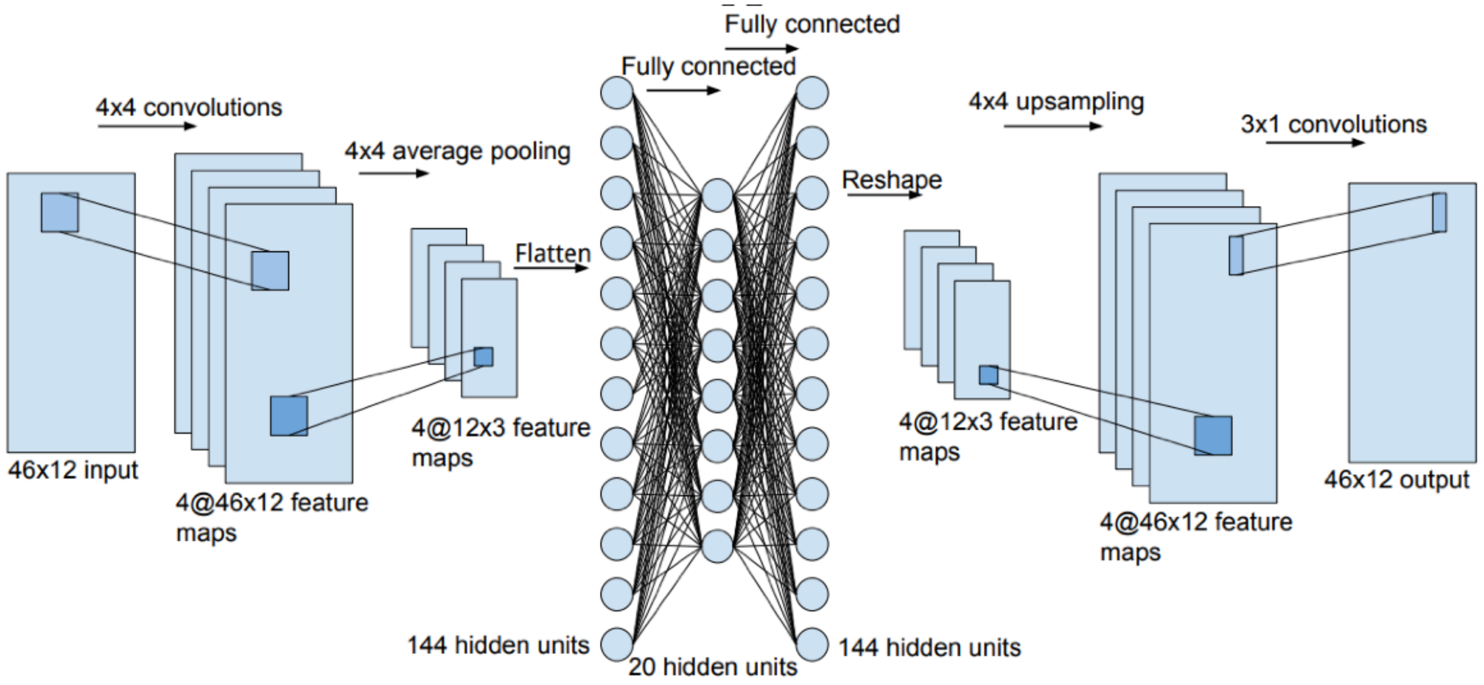




Data Quality Monitoring

- 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

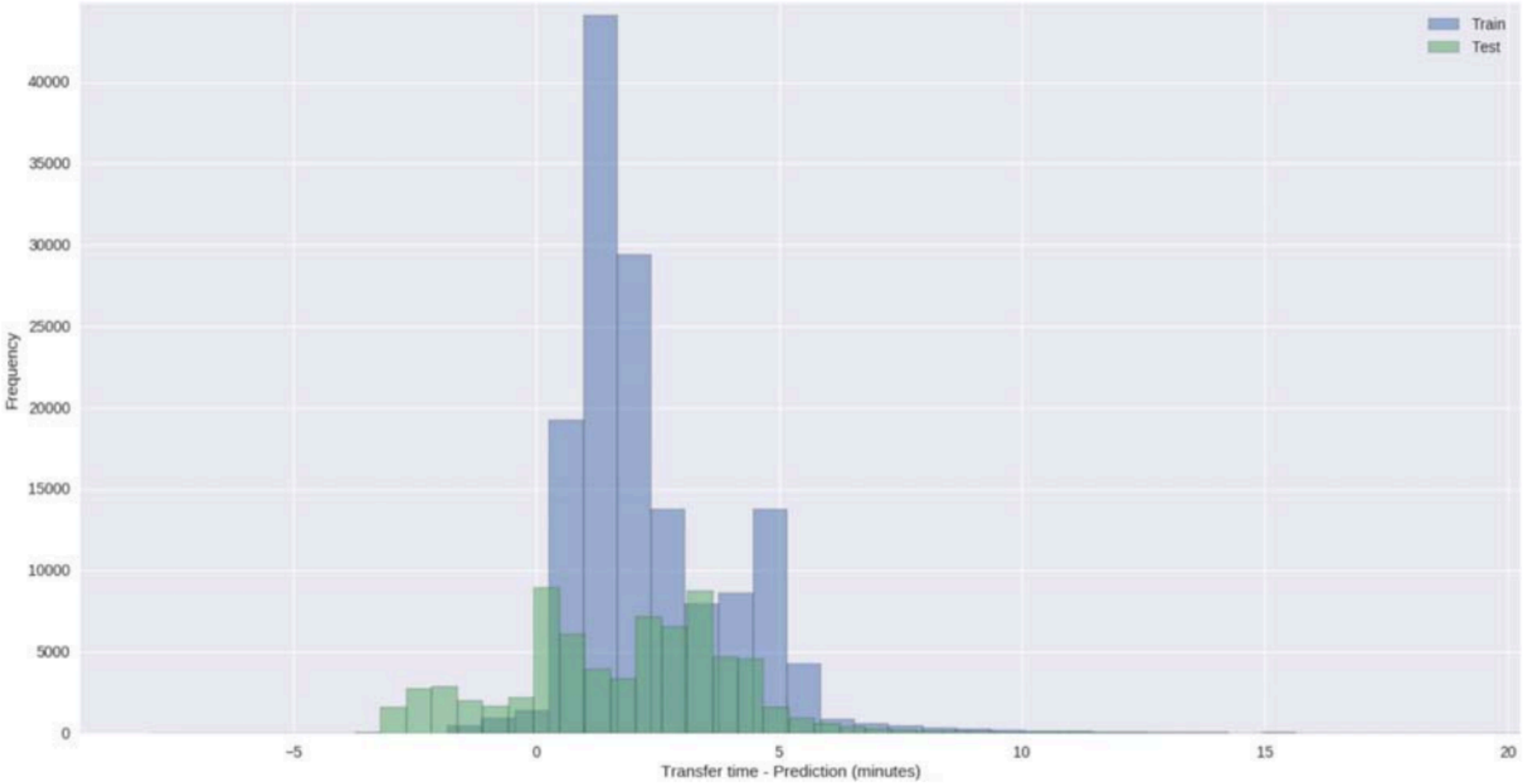




arXiv:1808.00911

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



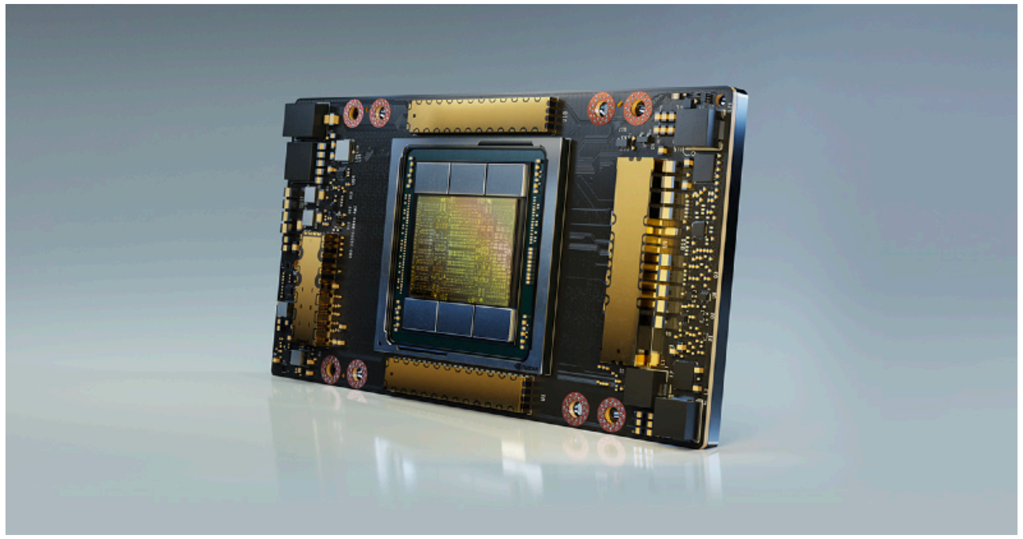
doi :10.1088/1742-6596/898/6/062009

ML for TDAQ: FastML

- Machine Learnings algorithms are highly parallelisable
  - Recall Neural Network forward pass is 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

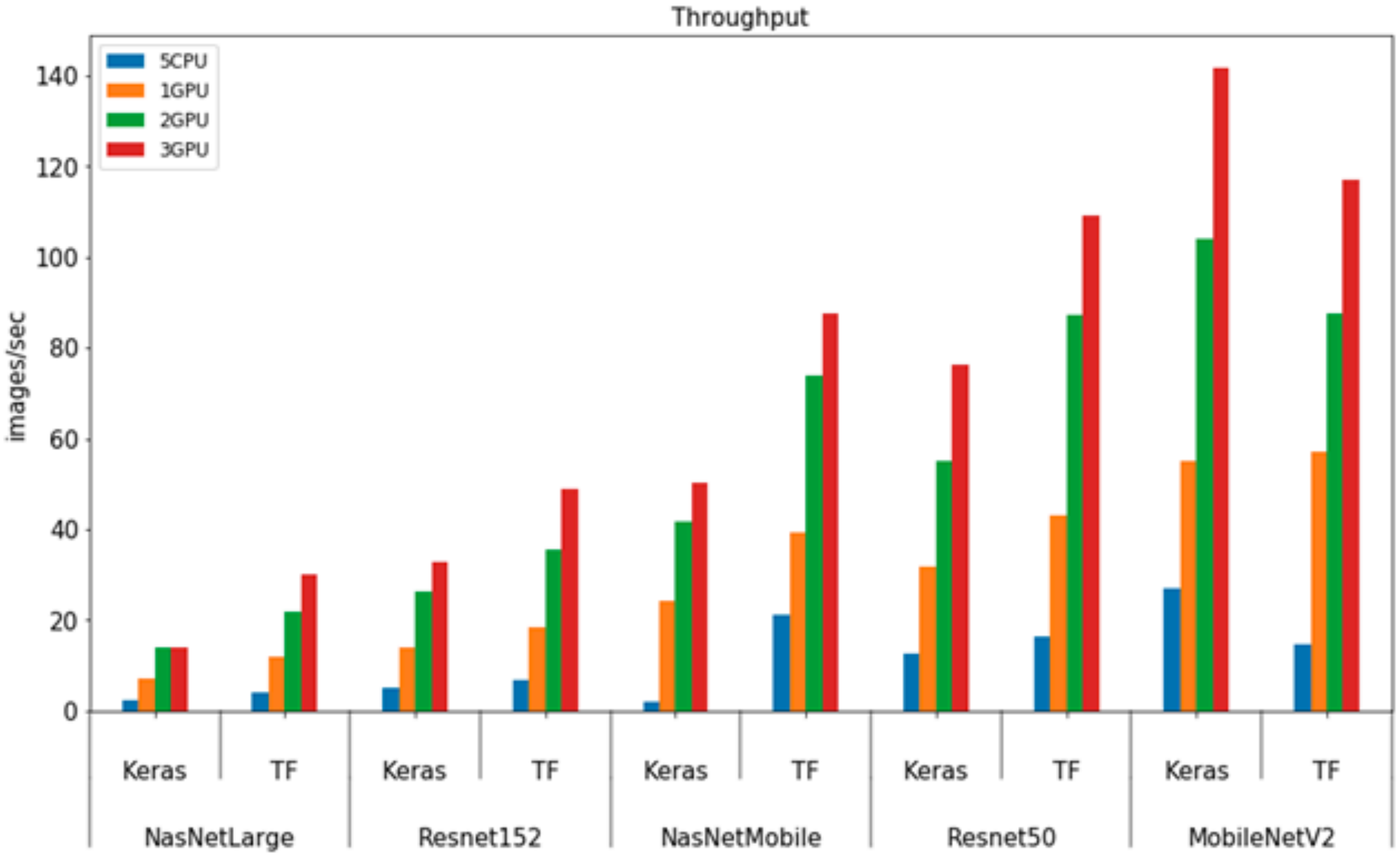
GPUs for ML

- 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
- See GPU lecture and Lab 14 from this school for more on programming GPUs



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 (ISOTDAQ: PCIexpress)
- Here, running inference on K80 GPUs, measuring images / second (throughput)



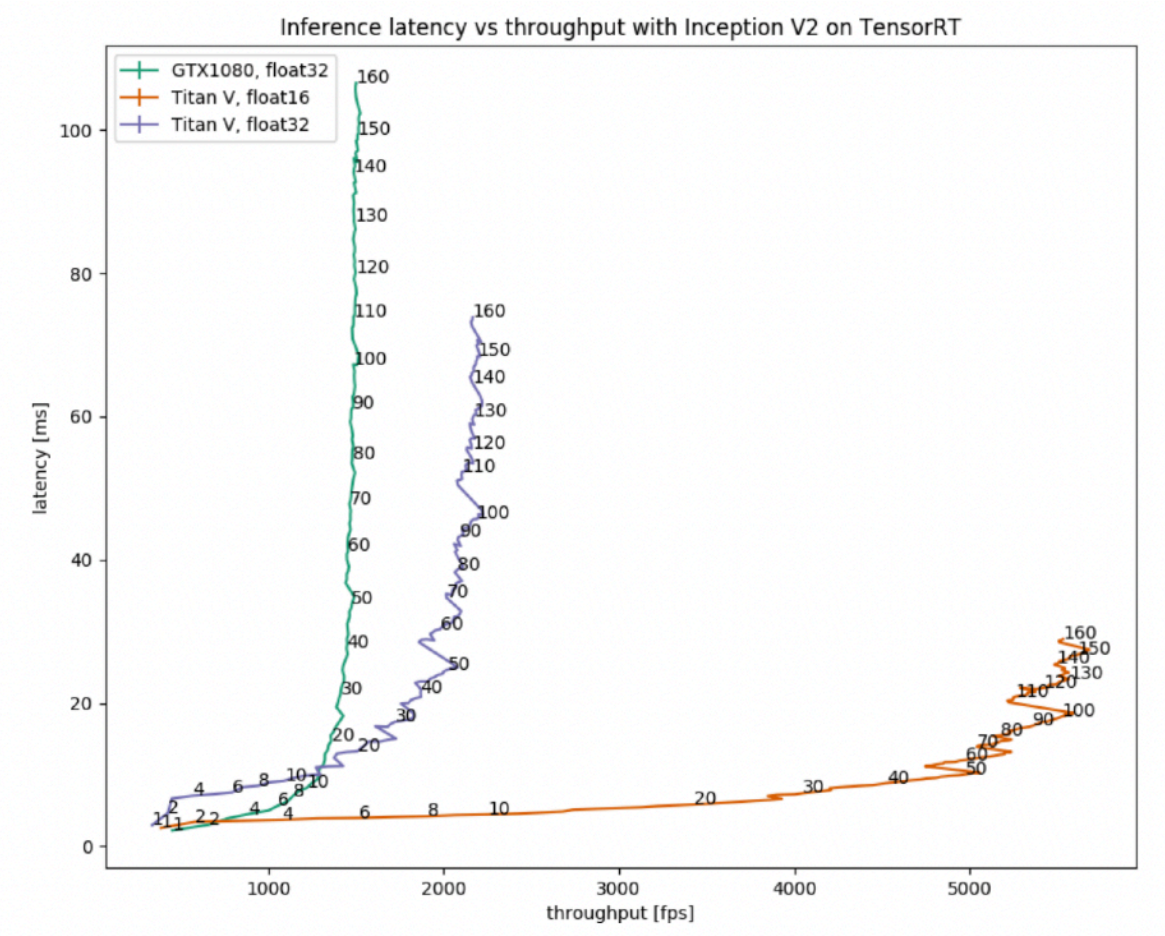
From Microsoft Azure

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

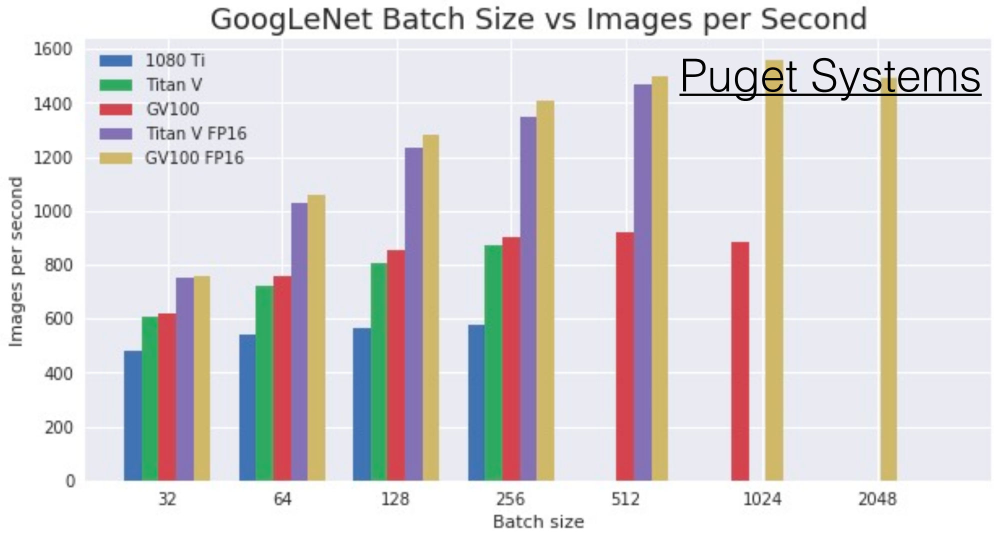
GPUs for ML - batching

- “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



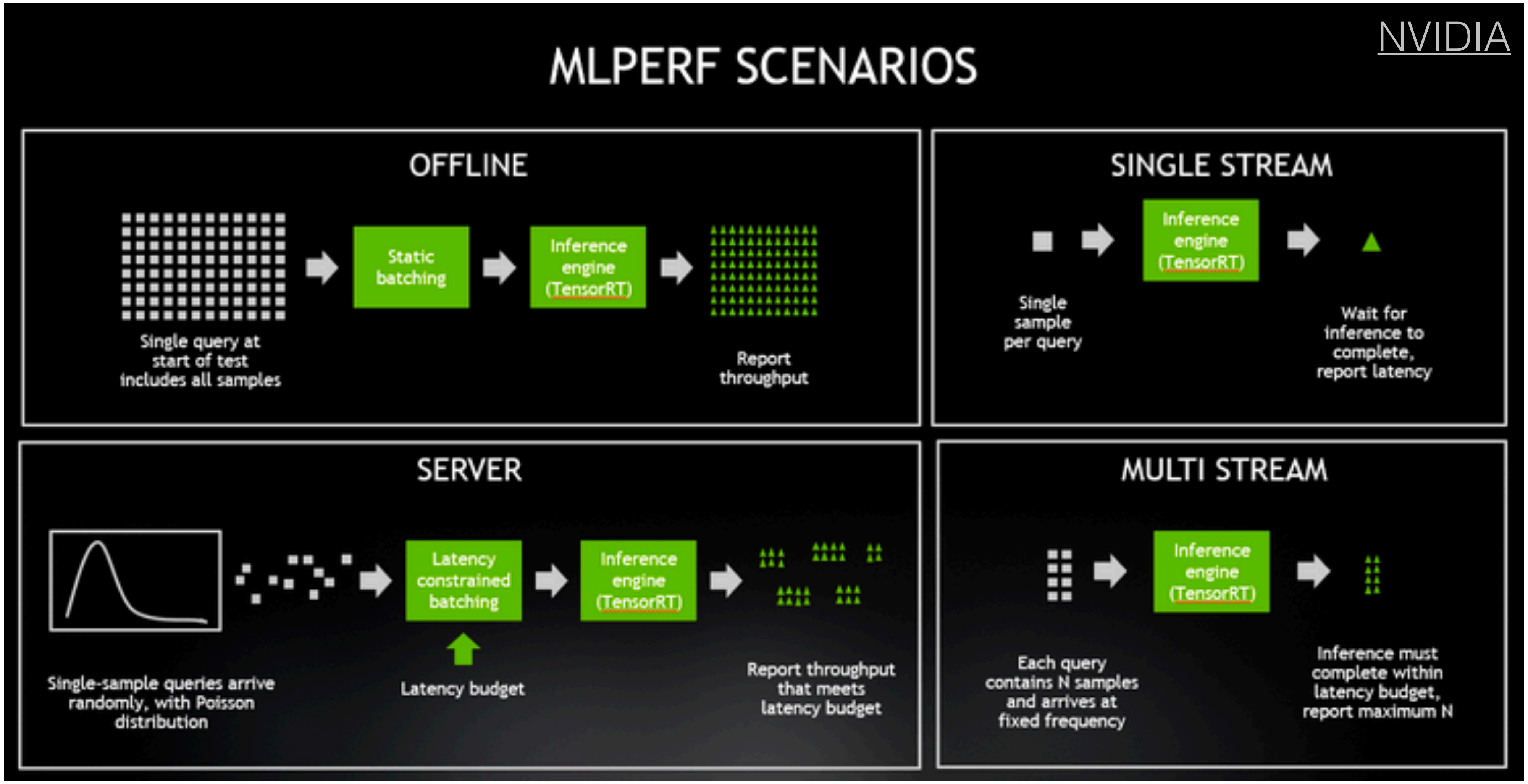
- Plot: throughput vs latency at different batch sizes for Inception V2 (large computer vision CNN)
    - On different GPUs and different precisions



Puget Systems

GPUs for ML - batching

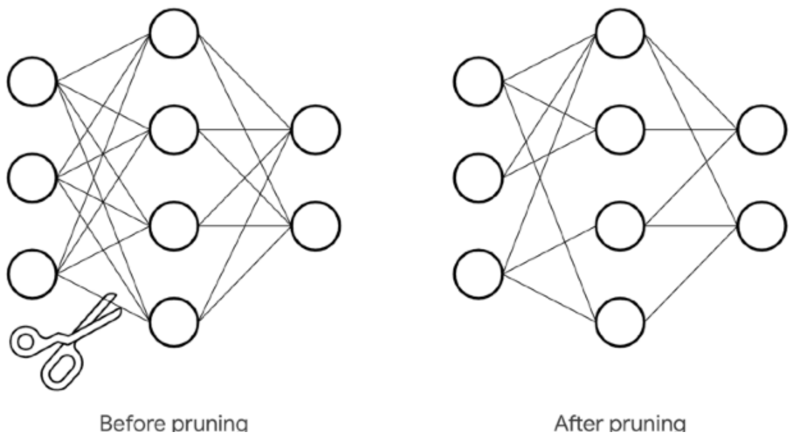
- Whether or not you can profit from batching depends also on:
  - Is the main constraint on throughput or latency? (Or both?)
  - The data source: do data arrive at fixed intervals (bottom right image), or stochastically (bottom left)?
  - Can you afford to wait to accumulate several samples before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

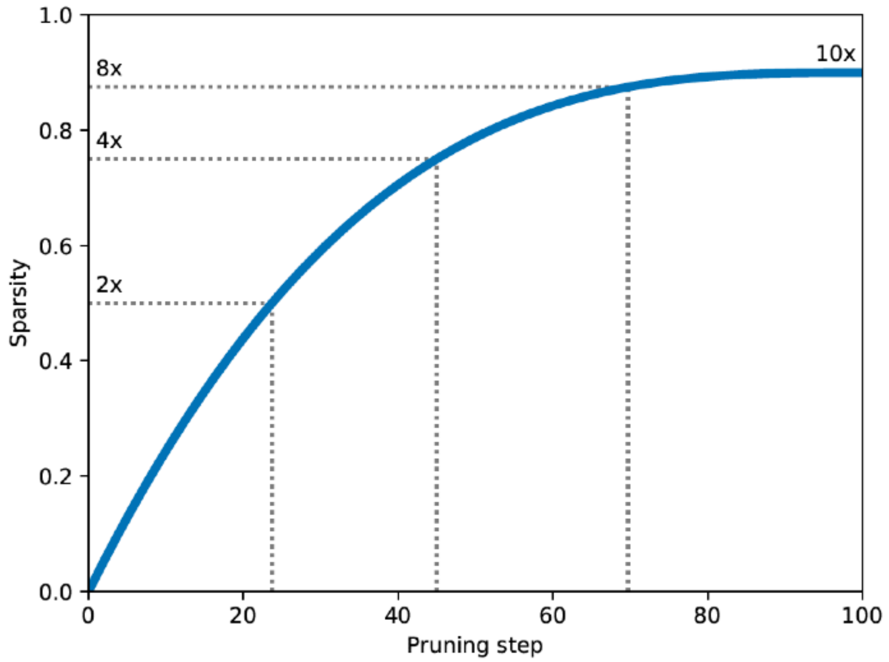

NVIDIA

MLPERF SCENARIOS
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

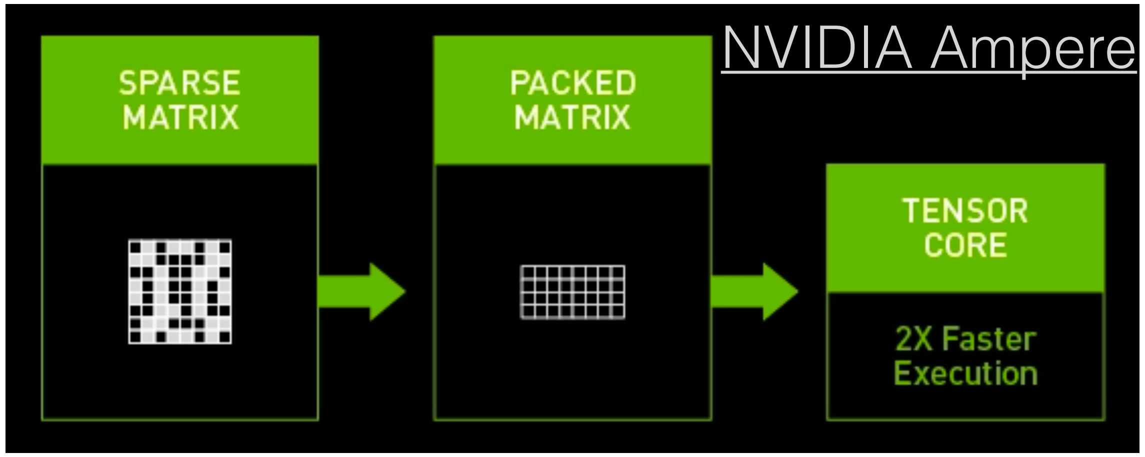
Pruning / Sparsity

- A Neural Network often contains many redundant connections
- Pruning methods generally remove some connections from the final model
    - Can improve generalisability also
- Can reduce the model size (memory footprint)
- Some processors can accelerate sparse networks
    - Basically - don’t do the x * 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 QAT (= QAP)





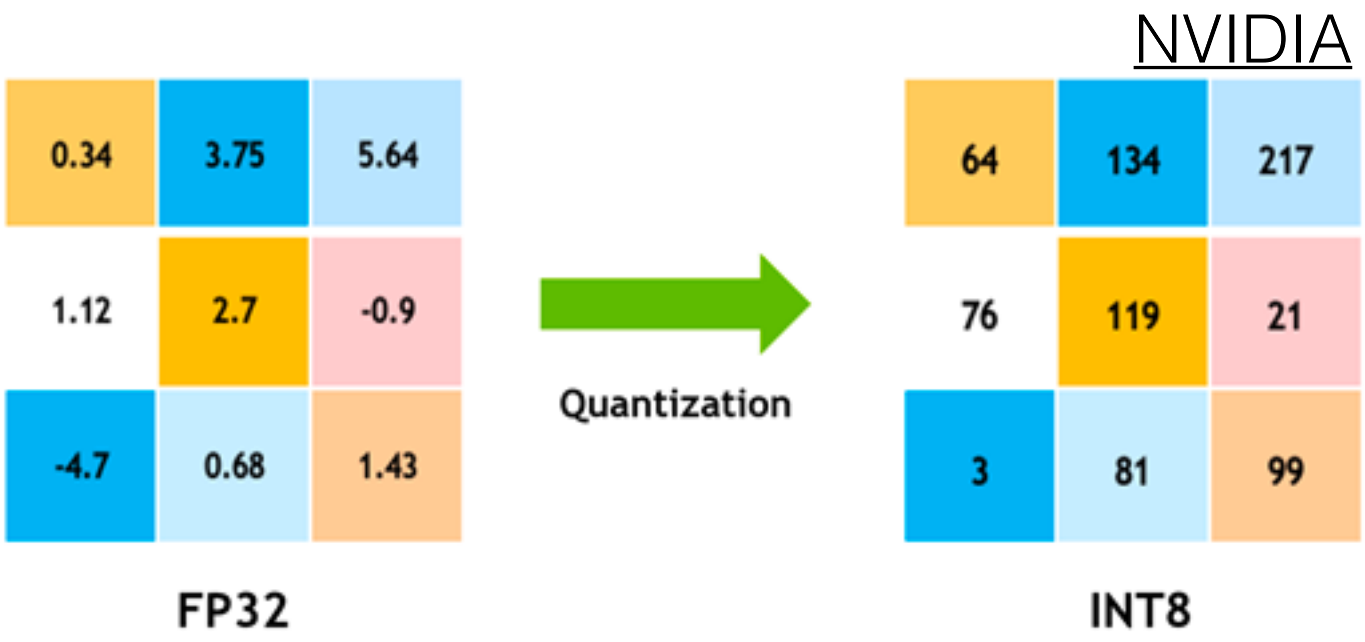
Images from Tensorflow blog



NVIDIA Ampere

Quantization

- Many GPUs support Int8, float16, bfloat16 precision with many more OPS than float32
    - Can do Post Training Quantization (PTQ) - train with FP32 then scale & round to lower precision
    - or Quantization Aware Training (QAT) - train with low precision (more on that later)
    - e.g. TensorRT (NVIDIA GPU), TensorFlow Lite (Google), torch.quantization (PyTorch)
- A method to make the most of the hardware: choices depend upon the target hardware

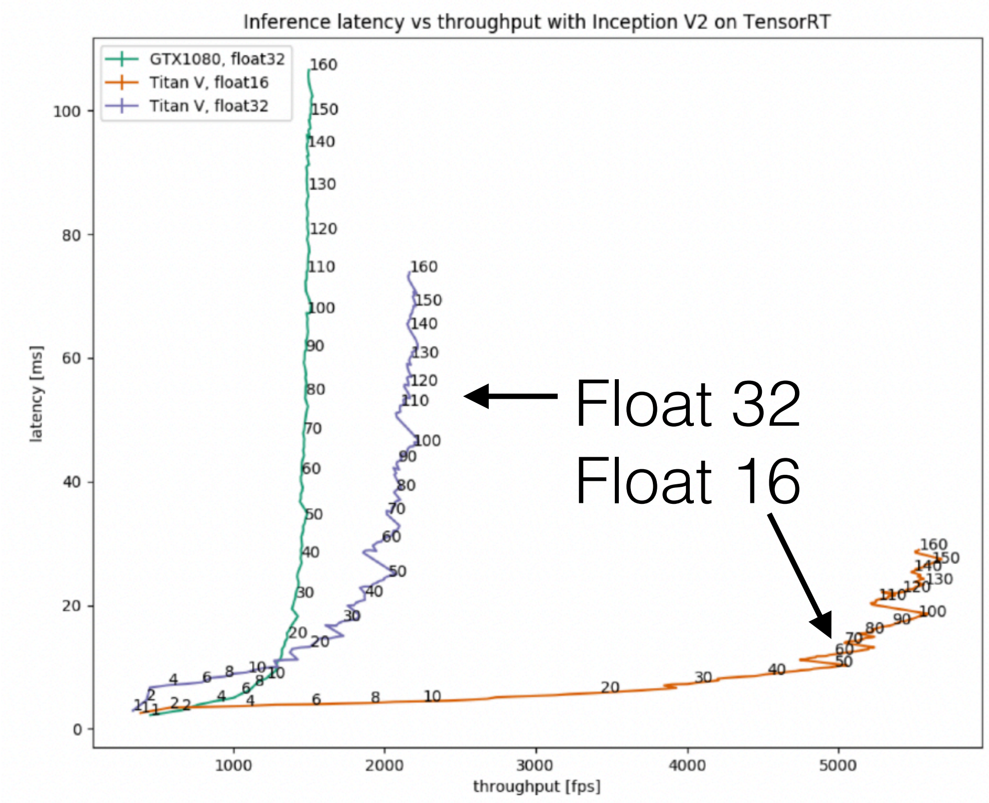

NVIDIA

0.34	3.75	5.64
1.12	2.7	-0.9
-4.7	0.68	1.43

FP32

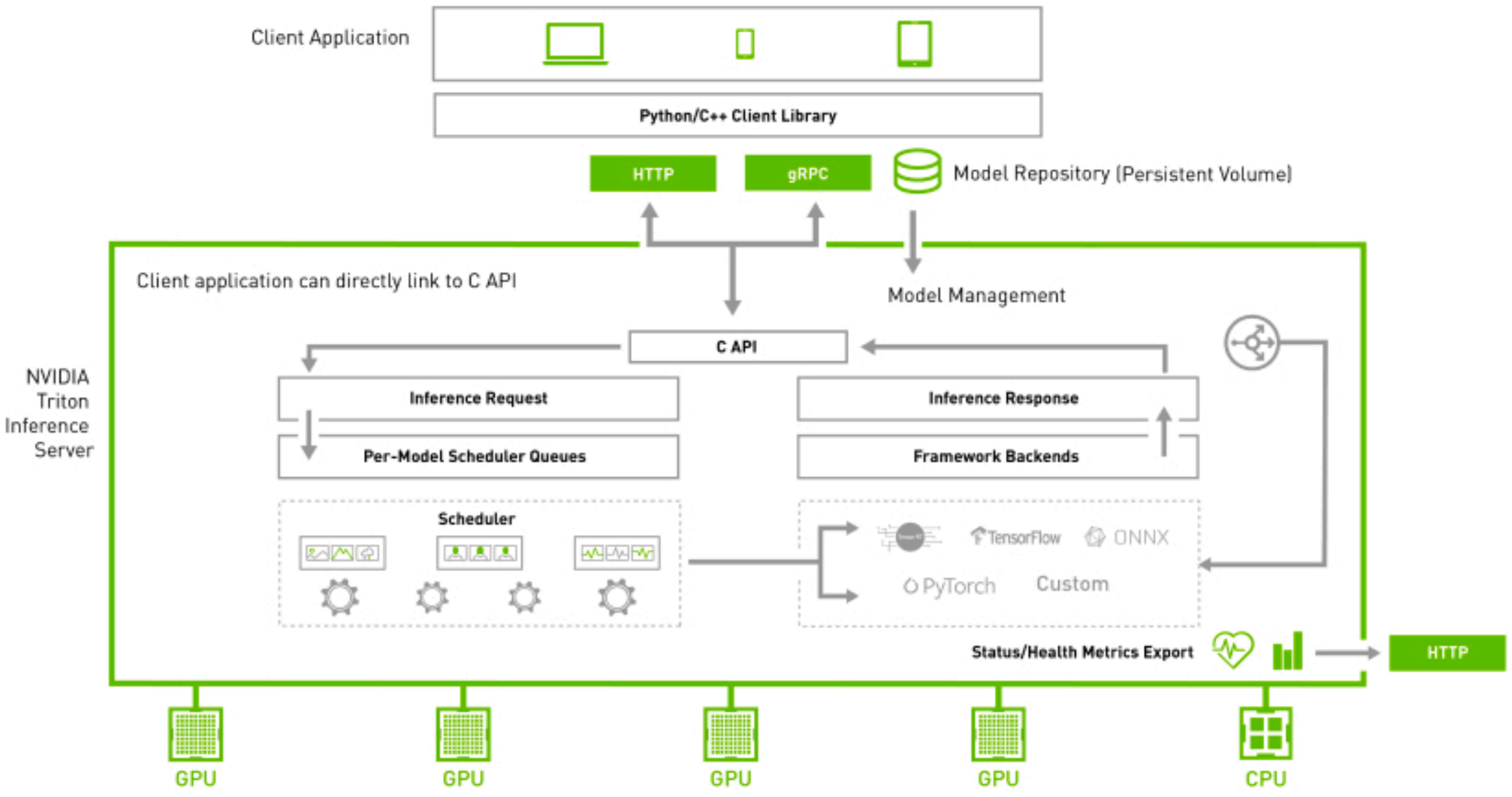
64	134	217
76	119	21
3	81	99

INT8



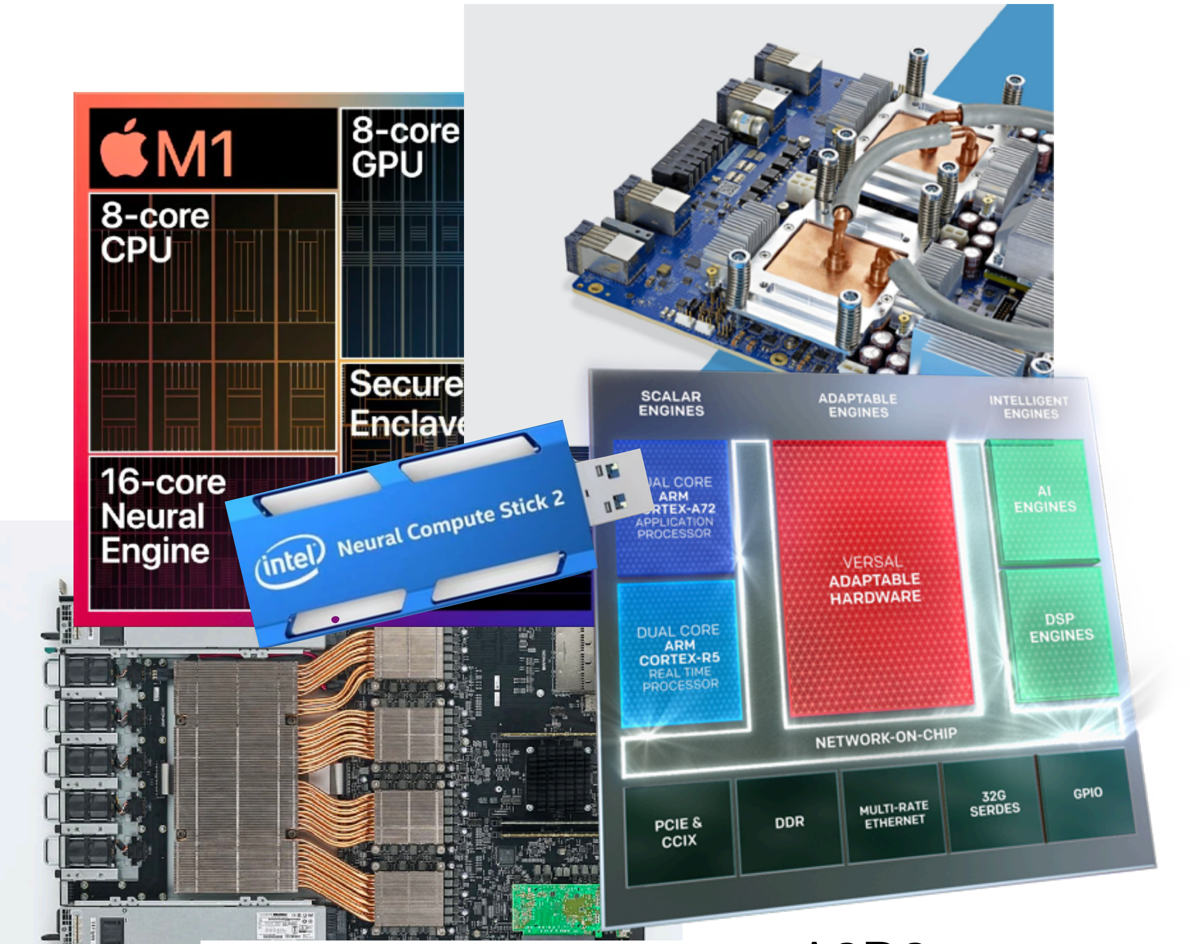
ML Deployment

- What about when you need to orchestrate many GPUs and many “clients” using them?
- Example: Triton inference server (NVIDIA, open source)
- Handles dynamic batching depending on requests to optimize latency/throughput performance
  - In HEP could be for varying event rate, or varying number inferences per event

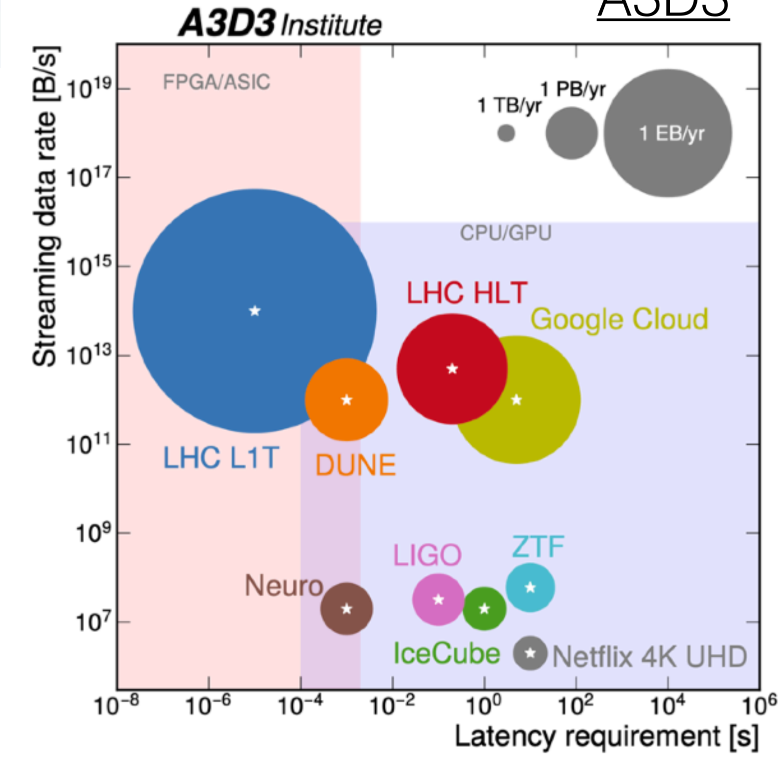

Client Application
Python/C++ Client Library
HTTP
gRPC
Model Repository (Persistent Volume)
Client application can directly link to C API
Model Management
NVIDIA Triton Inference Server
C API
Inference Request
Per-Model Scheduler Queues
Inference Response
Framework Backends
Scheduler
TensorFlow
ONNX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

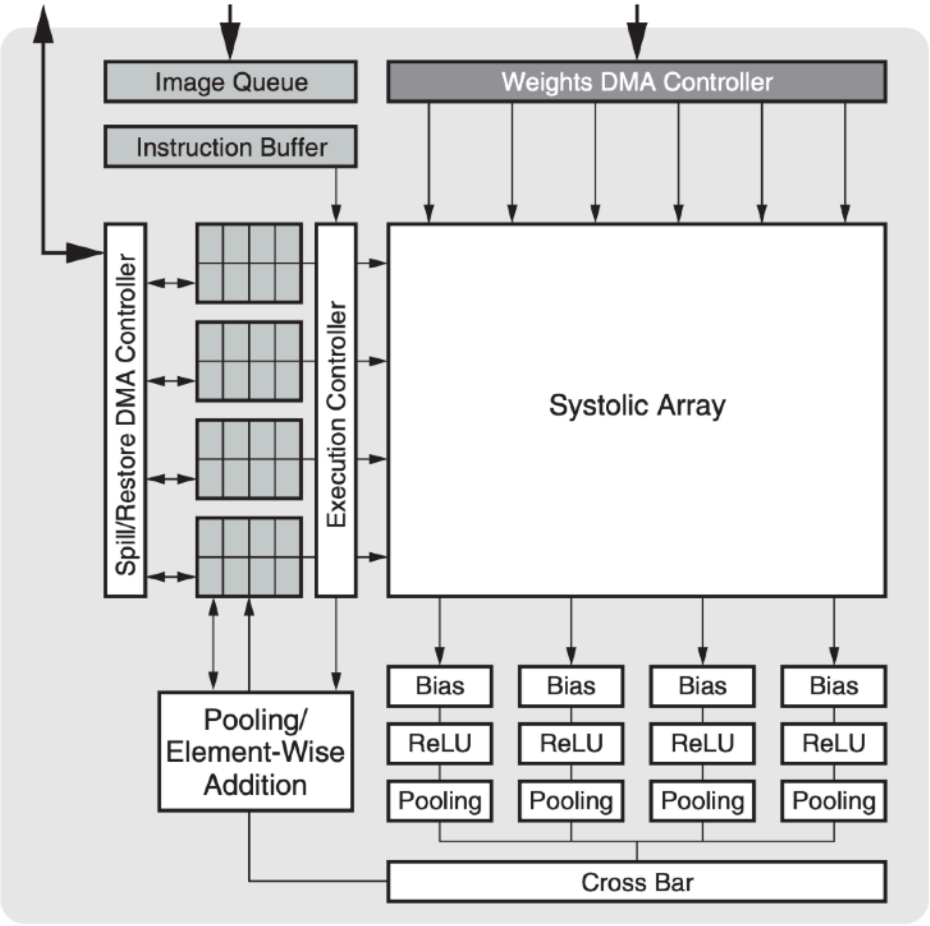


A3D3



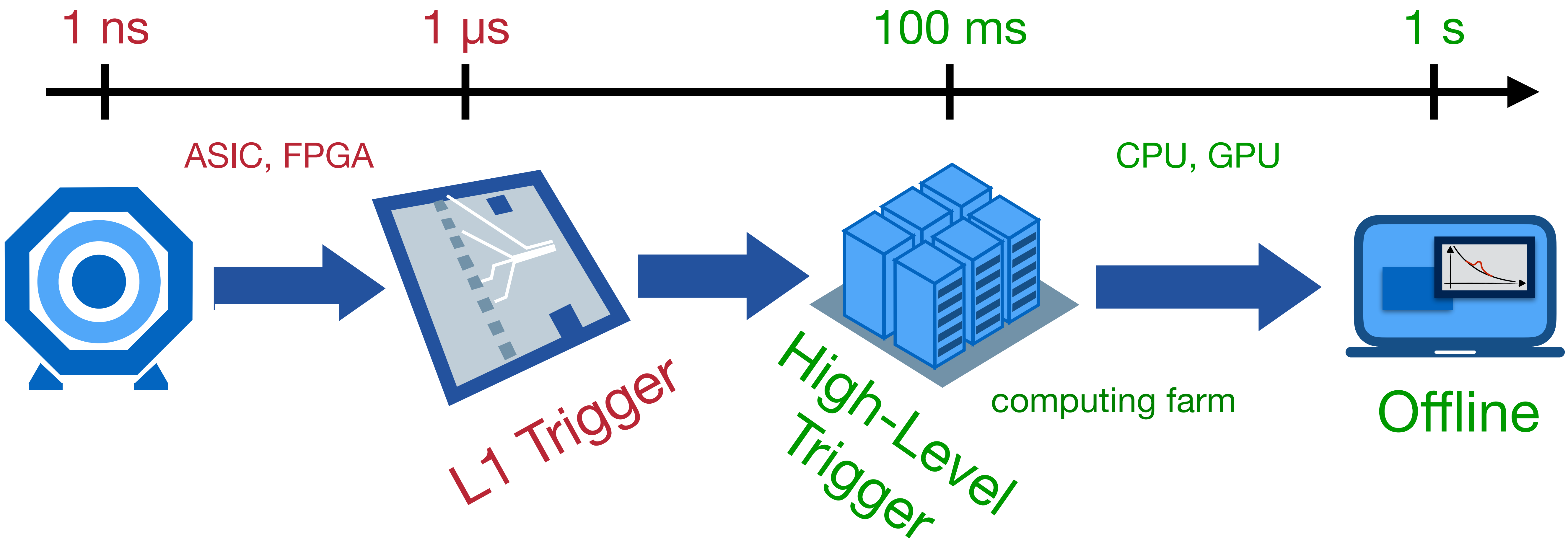
FPGAs for ML

- FPGAs are also highly suited to ML tasks - massive parallelism, high memory bandwidth
- There are 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

Machine Learning at L1 Trigger



- Typical ‘latency landscape’ of LHC experiment triggering
- To deploy Machine Learning at the L1 Trigger need to:
    - Be able to execute ML algorithms in O(1µs)
    - Execute these algorithms on FPGAs and ASICs

What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

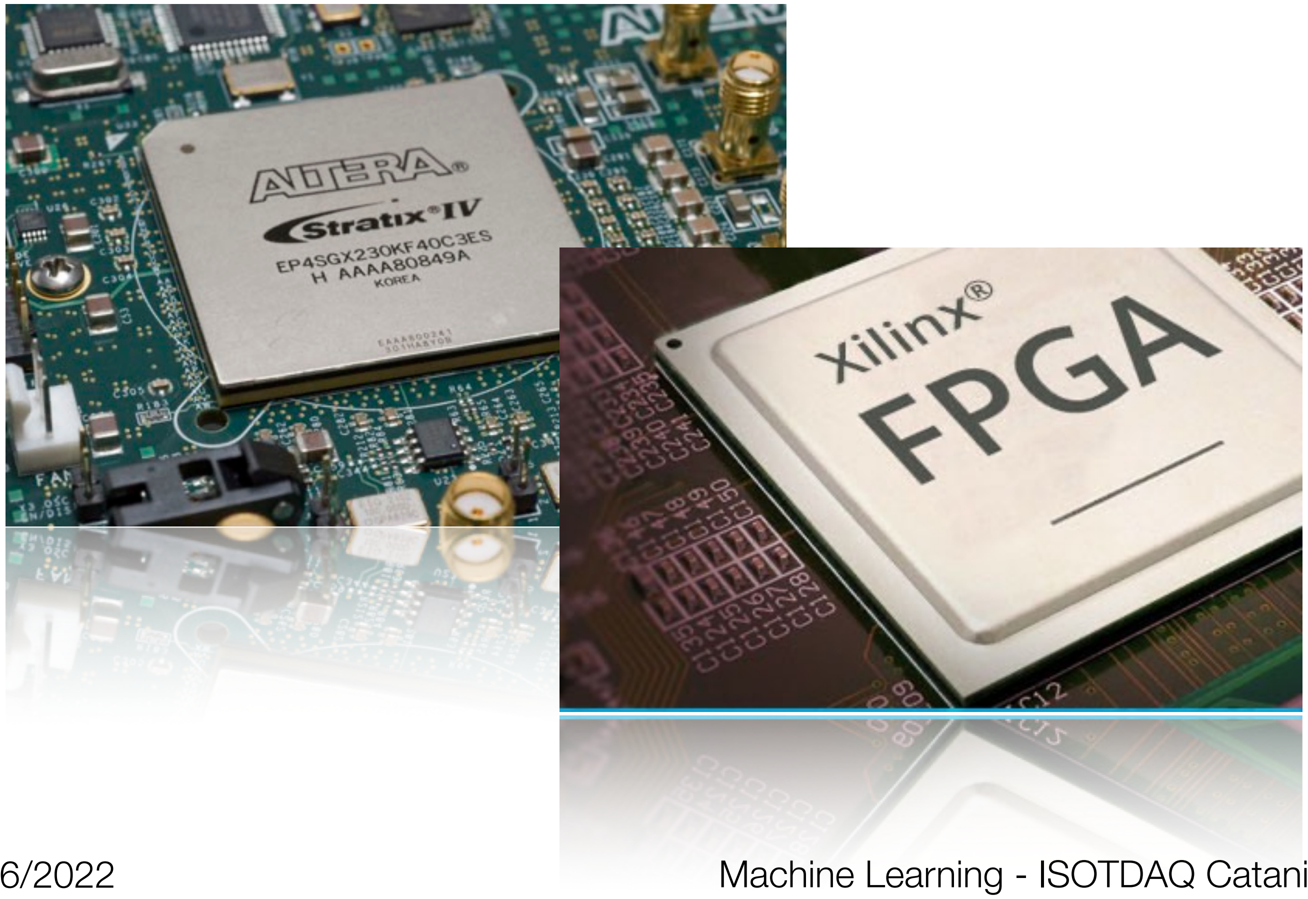
See talks "Introduction to FPGAs", "Advanced FPGA Programming", and Labs "FPGA Programming", "SoC FPGA" at this school

Contain many different building blocks ('resources') which are connected together as desired

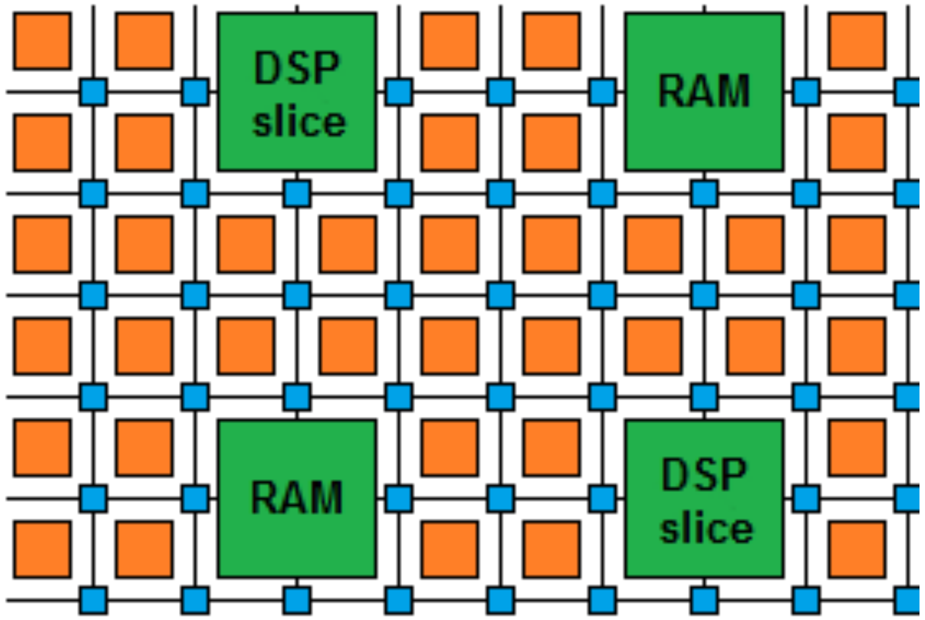
Extremely parallel processors

'Computing in space as well as time'

Processing workhorse of low level HEP triggers



FPGA diagram



LUTs - generic logic

DSPs - for multiplication

BRAM - for local, high-throughput storage

High Level Synthesis

- FPGA programming is hard
    - 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;
```

vs

```
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
- These also provide a powerful handle for optimisation not accessible to HDL developers

```
#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];
…
```

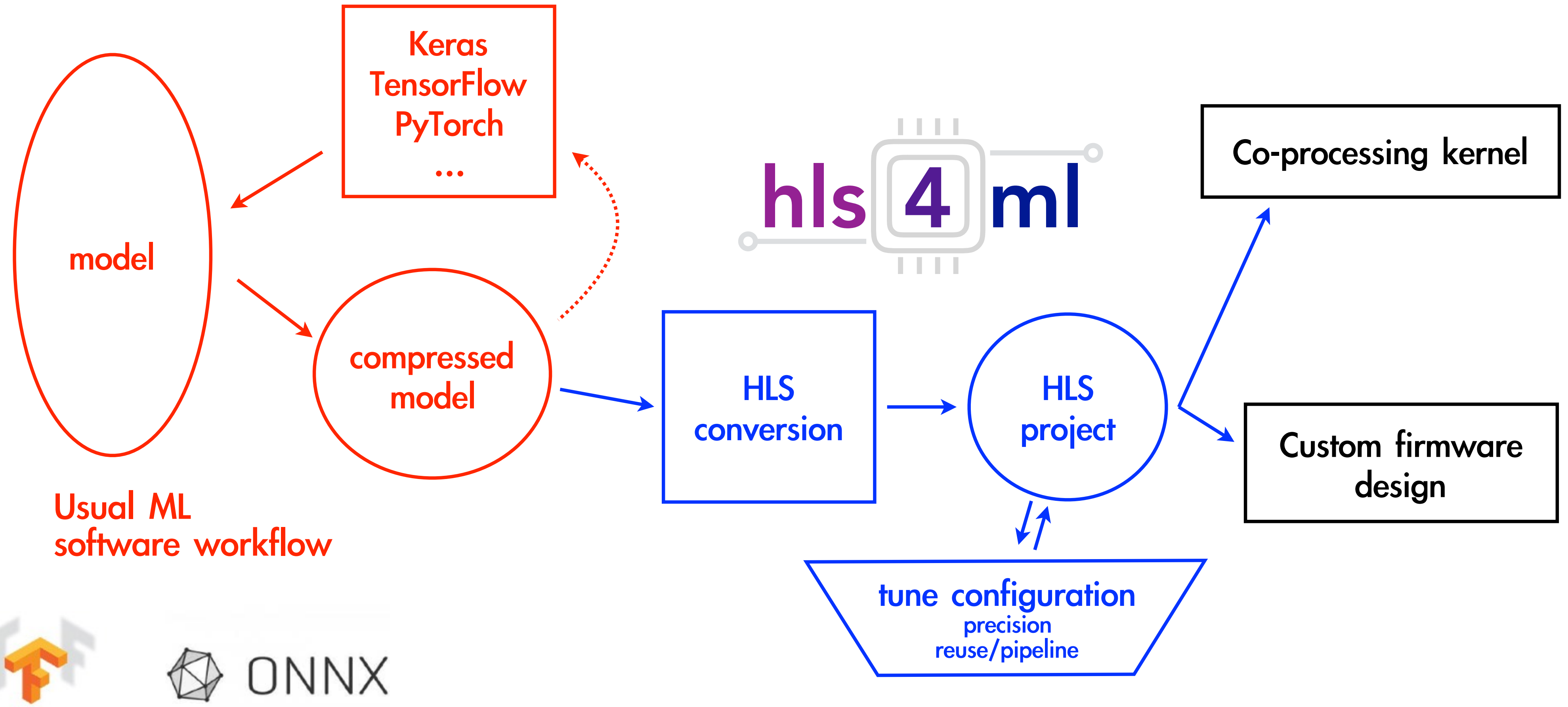
Use registers

Execute loop
iterations
in parallel

high level synthesis for machine learning

Implemented a user-friendly, open-source tool to develop and optimize FPGA firmware design for Machine Learning inference:

- Input models trained with standard ML libraries ((Q)Keras, PyTorch, (Q)ONNX)
- NN implementations using HLS C++
- comes with implementation of common ingredients - layer types, activation functions
- and novel ingredients for fast, efficient inference - binary/ternary NNs, network optimisations



Keras
TensorFlow
PyTorch
…

model

compressed model

Usual ML software workflow

hls 4 ml

HLS conversion

HLS project

tune configuration
precision
reuse/pipeline

Co-processing kernel

Custom firmware design

Vivado™ HLS

intel HLS COMPILER

Mentor A Siemens Business

Catapult HLS

K + TensorFlow

ONNX

PyTorch

https://fastmachinelearning.org/hls4ml/

https://arxiv.org/abs/1804.06913

hls4ml - NN implementation

- Dataflow architecture: each layer is an independent compute unit
  - With tunable parallelism and quantization
- Fully on-chip: NN must fit within available FPGA resources (pynq-z2 floorplan shown)
  - Example: small CNN trained on MNIST

Conv2D

ReLU

MaxPool2D

Conv2D

ReLU

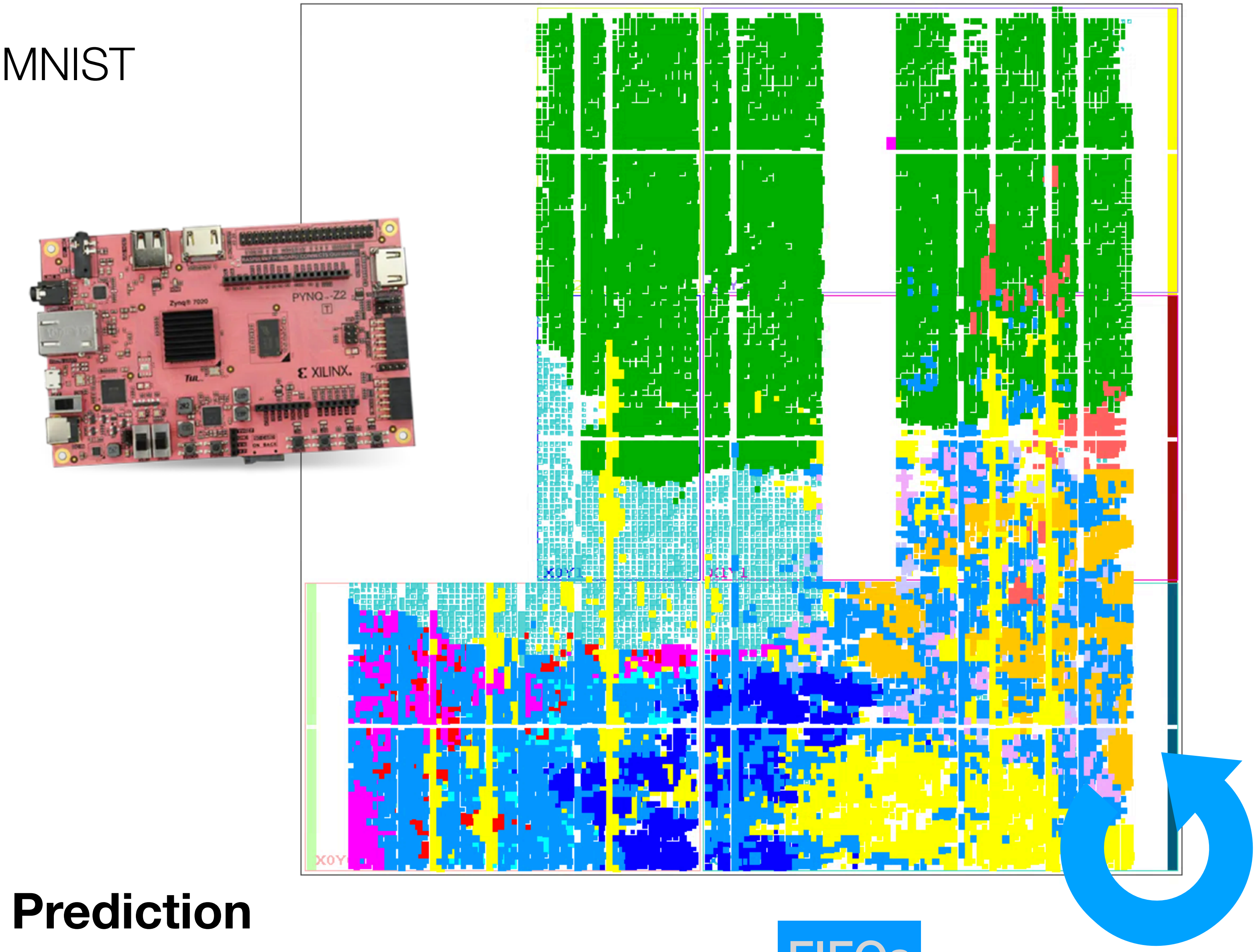
MaxPool2D

Flatten

Dense

Softmax

Prediction



FIFOs

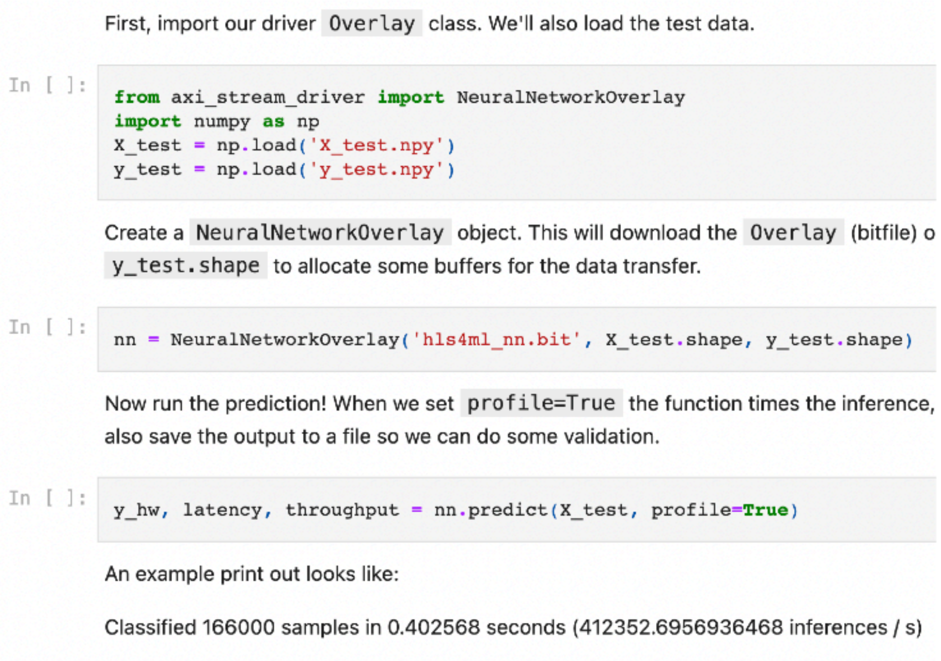
hls4ml one slide primer

- 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 - ReuseFactor, Strategy, parallel/streaming IO
- Much more information in the docs

```
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
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()
```

hls4ml deployment

- For custom FPGA board (like trigger): add NN IP core to project, connect signals in HDL
- For some popular boards hls4ml has a ‘VivadoAccelerator’ backend
  - Support for pynq-z2 (same as Lab 13: SoC at ISOTDAQ), ZCU102, Alveo coming soon
- NN IP has AXI Stream inputs & outputs, use Xilinx DMA IP to move data from PS
  - It could come straight from an IP reading a sensor
  - With driver for Xilinx PYNQ Python



First, import our driver `Overlay` class. We'll also load the test data.

```
In [ ]:  from axi_stream_driver import NeuralNetworkOverlay
         import numpy as np
         X_test = np.load('X_test.npy')
         y_test = np.load('y_test.npy')
```

Create a `NeuralNetworkOverlay` object. This will download the `Overlay` (bitfile) o `y_test.shape` to allocate some buffers for the data transfer.

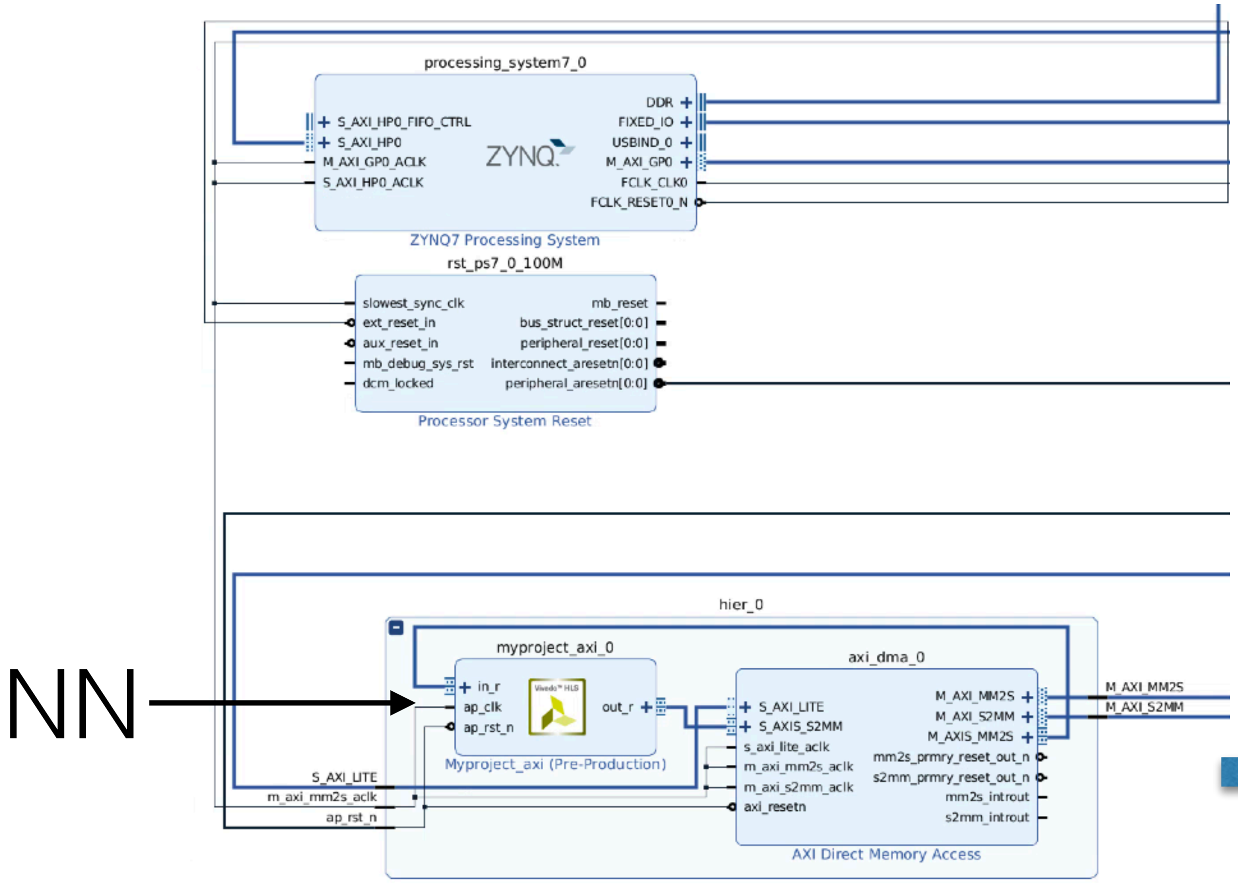
```
In [ ]:  nn = NeuralNetworkOverlay('hls4ml_nn.bit', X_test.shape, y_test.shape)
```

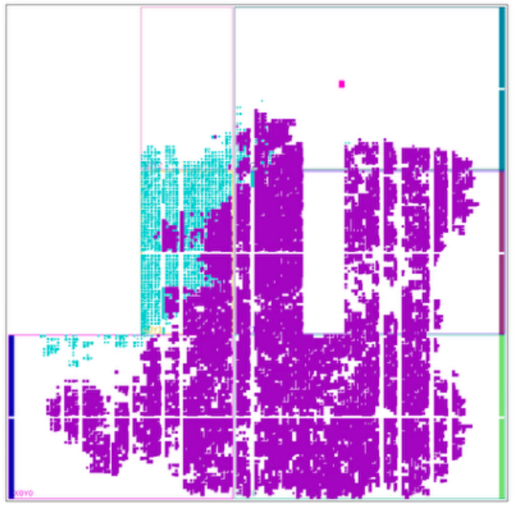
Now run the prediction! When we set `profile=True` the function times the inference, also save the output to a file so we can do some validation.

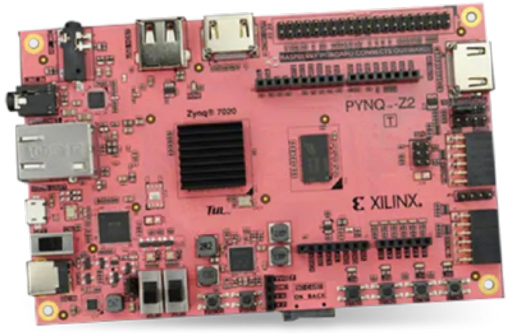
```
In [ ]:  y_hw, latency, throughput = nn.predict(X_test, profile=True)
```

An example print out looks like:

```
Classified 166000 samples in 0.402568 seconds (412352.6956936468 inferences / s)
```



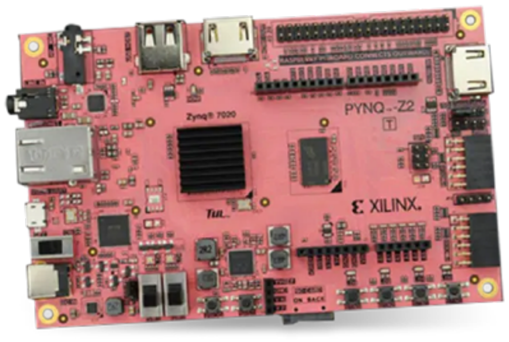


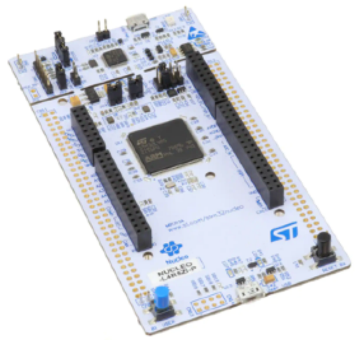


MLPerf™ Tiny Inference Benchmark

- 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

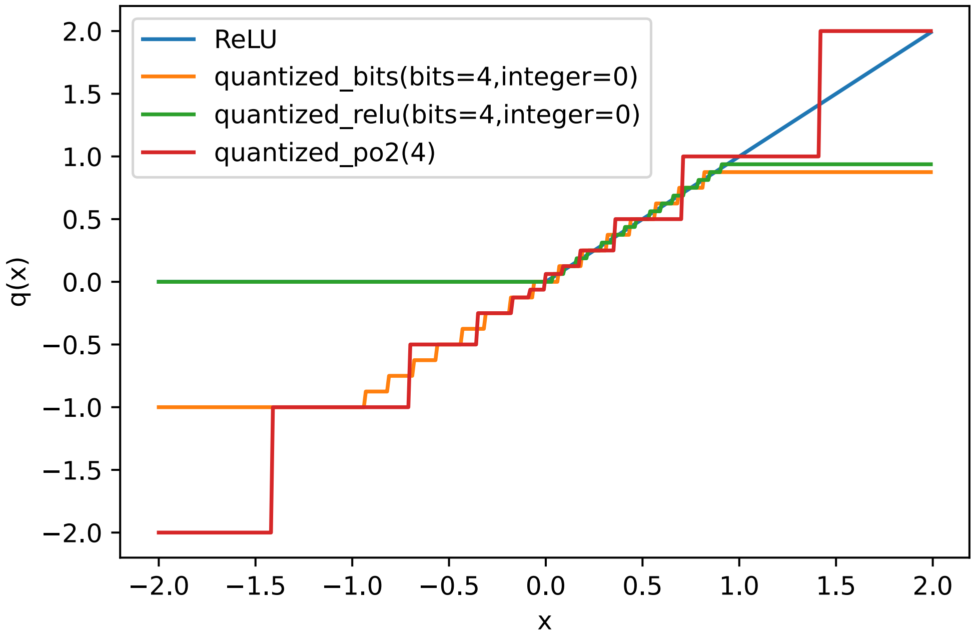
Benchmark		CIFAR-10			ToyADMOS		
Team	**Device**	**Accuracy**	**Latency (ms)**	**Power (W)***	**AUC**	**Latency (ms)**	**Power (W)***
hls4ml	Pynq-z2	77%	7.9	~ 1.5	0.82	0.096	~ 1.5
Latent AI	Raspberry Pi 4	85%	1.07	~ 4 - 5	0.85	0.17	~ 4 - 5
Harvard	Nucleo-L4R5ZI	85%	704		0.85	10.4	
Peng Cheng Lab	PCL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

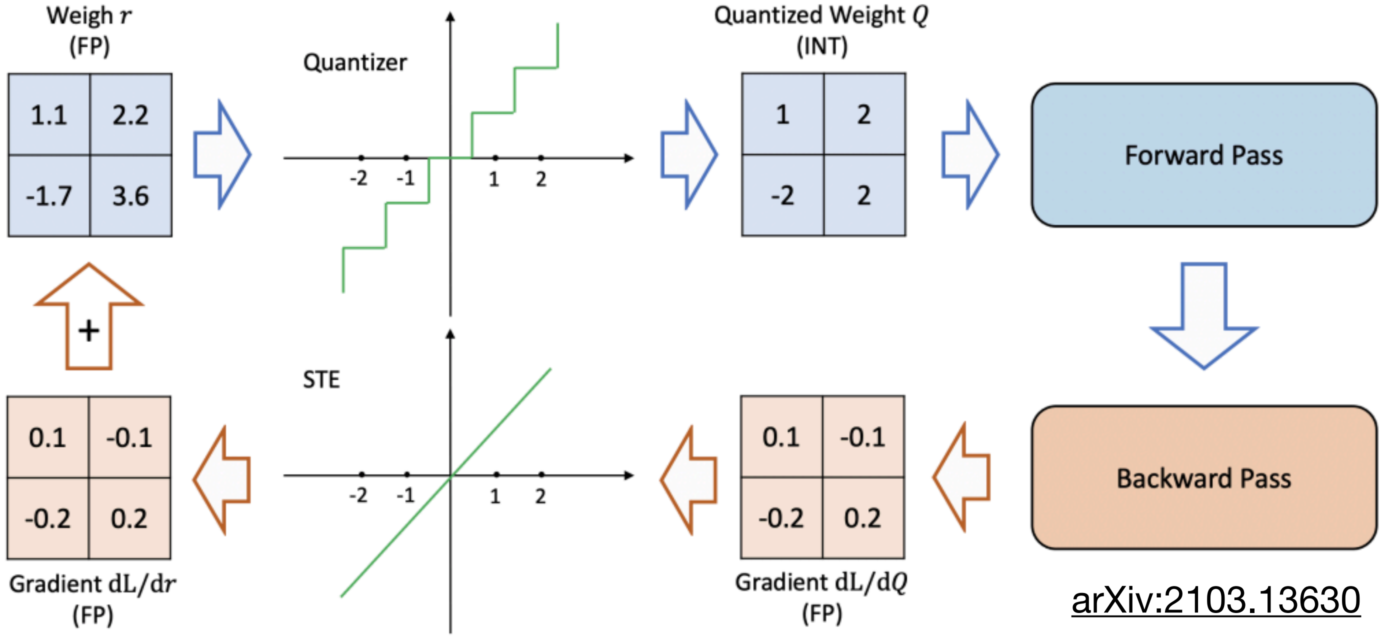




Quantization Aware Training

- Possibly the main technique for making NNs cheaper in FPGAs!
- 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
- 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





1.1	2.2
-1.7	3.6

1	2
-2	2

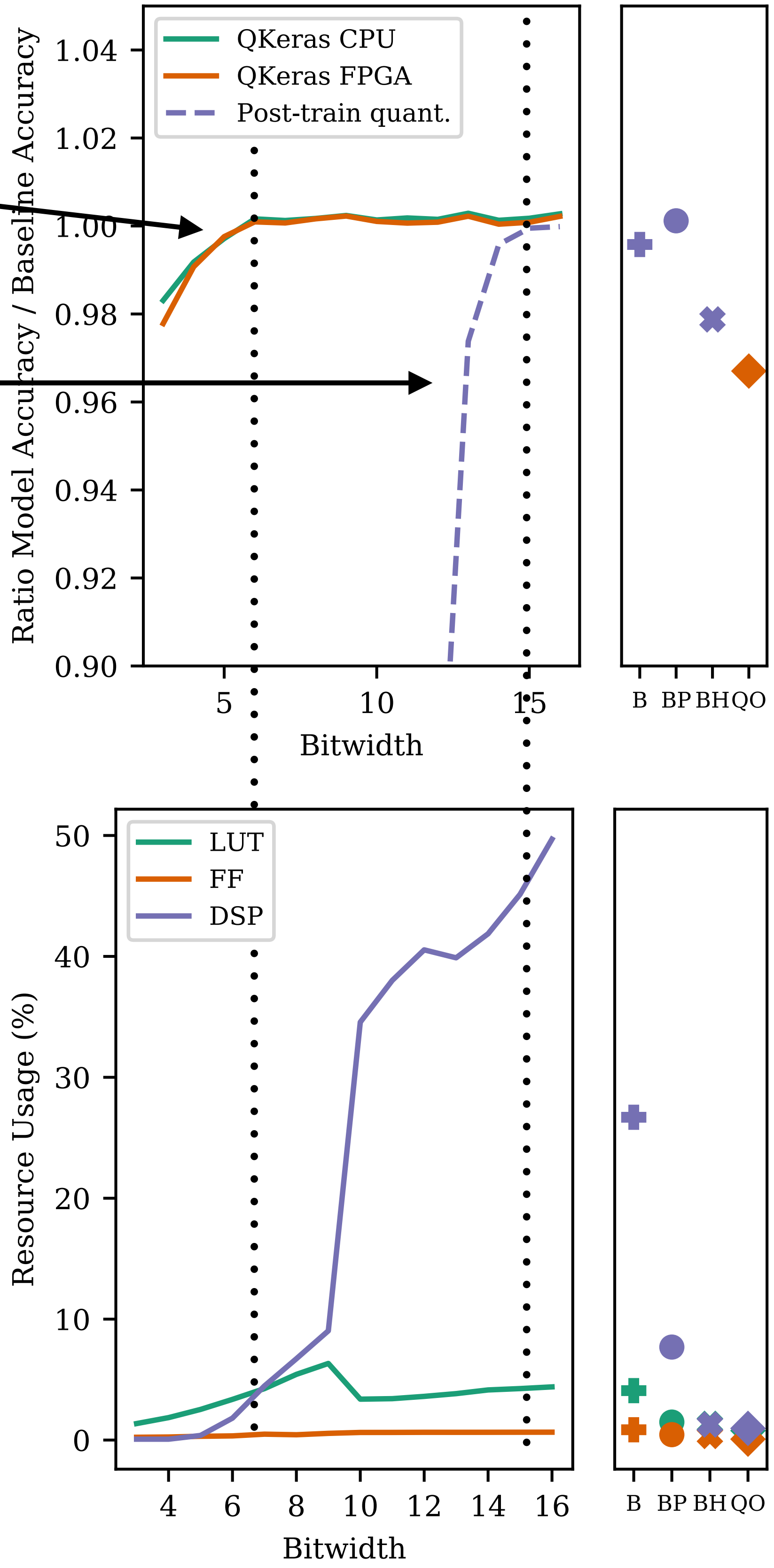
0.1	-0.1
-0.2	0.2

0.1	-0.1
-0.2	0.2

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)
  - Because of hls4ml’s dataflow architecture - we can take full advantage of that in device
- AutoQ tool for training NNs with hardware-cost constraints
- https://www.nature.com/articles/s42256-021-00356-5



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

- With Xilinx Research Labs we (hls4ml team) develop QONNX

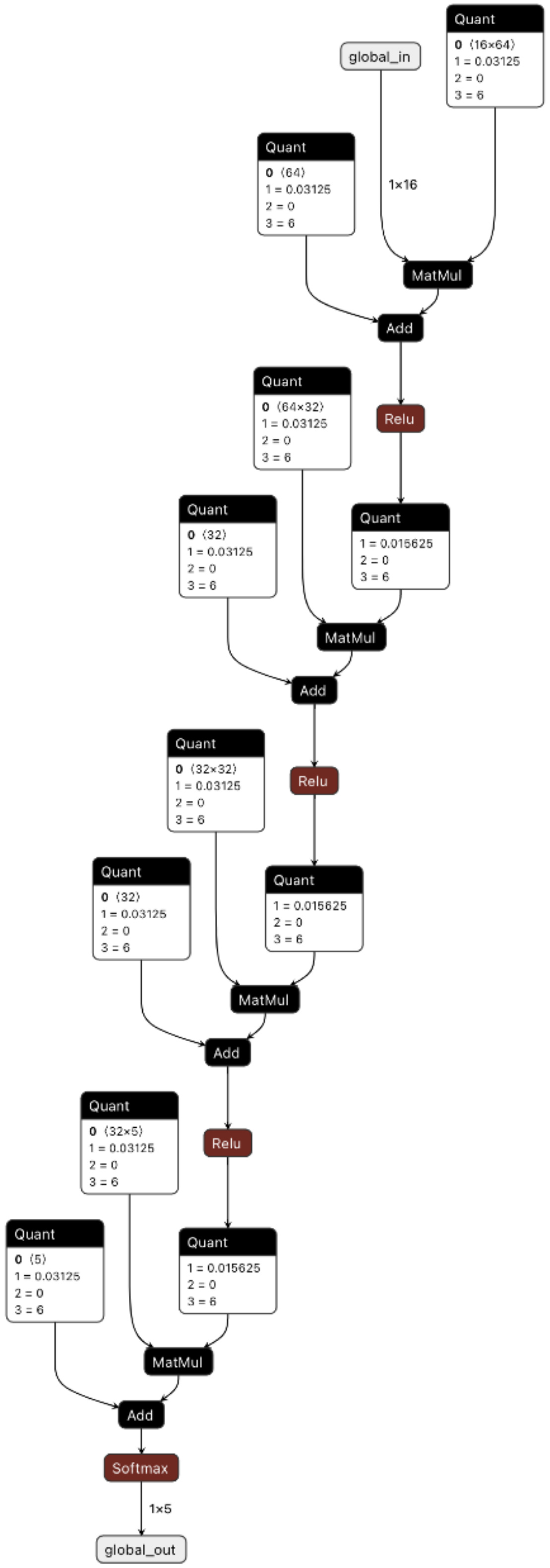
- Extend QONNX with `Quant` node

    - Flexible number of bits, zero-point, and per-channel scale factors

    - onnxruntime execution thanks to FINN (Xilinx RL NNs)

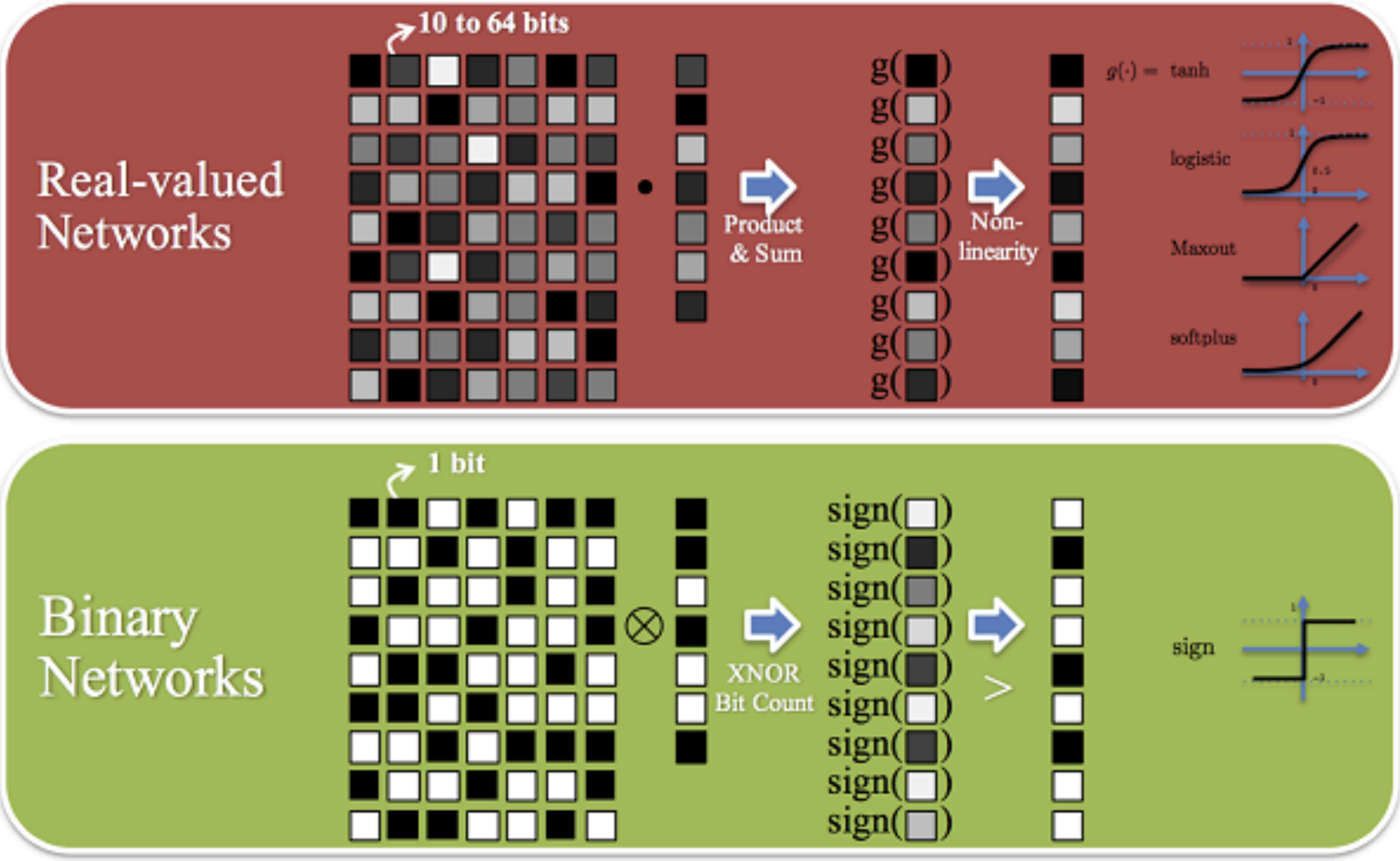
    - QONNX is exported by Brevitas, others are working on it, and we develop a QKeras to QONNX conversion

- github.com/fastmachinelearning/qonnx



Binary / Ternary neural networks

- DSPs (multipliers) often the limiting resource for our NN inference
- Can go down to event 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
- Batch Normalization
- BNN: arxiv.1602.02830
- TNN: arxiv.1605.04711



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

BNN - Dense Layer

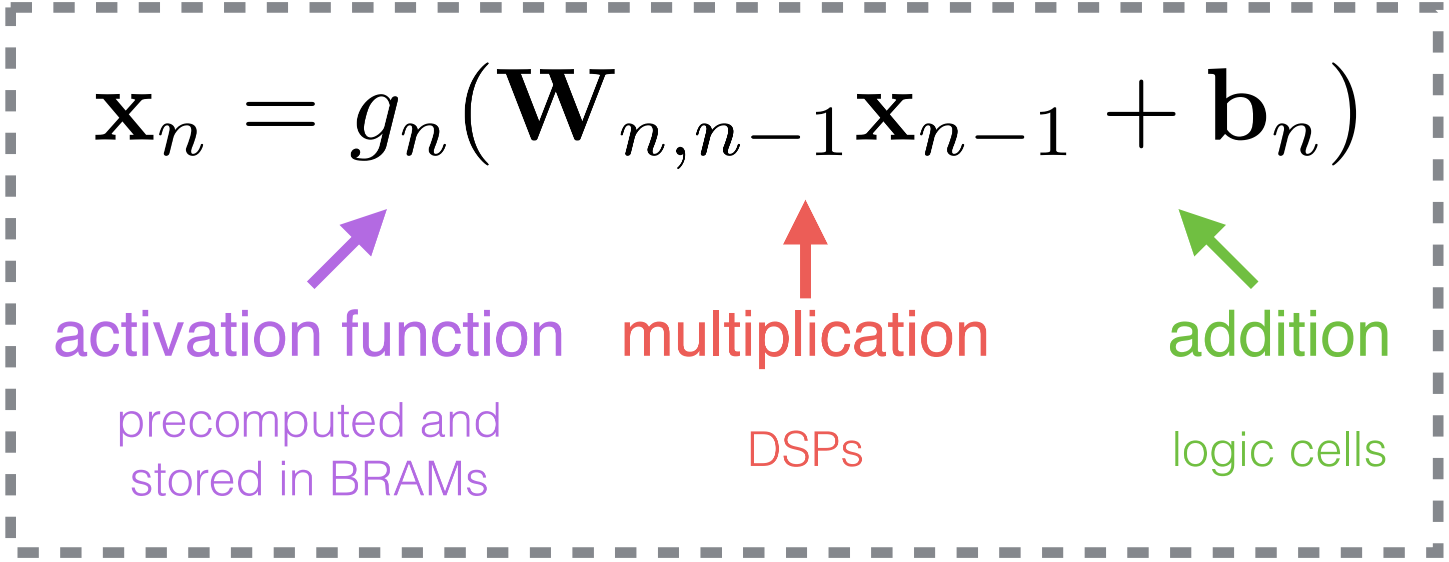
- DSPs often limiting FPGA resource for NNs
- Encode ‘-1’ as ‘0’
- Multiplication become XNOR, sum becomes bitcount

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



$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

activation function

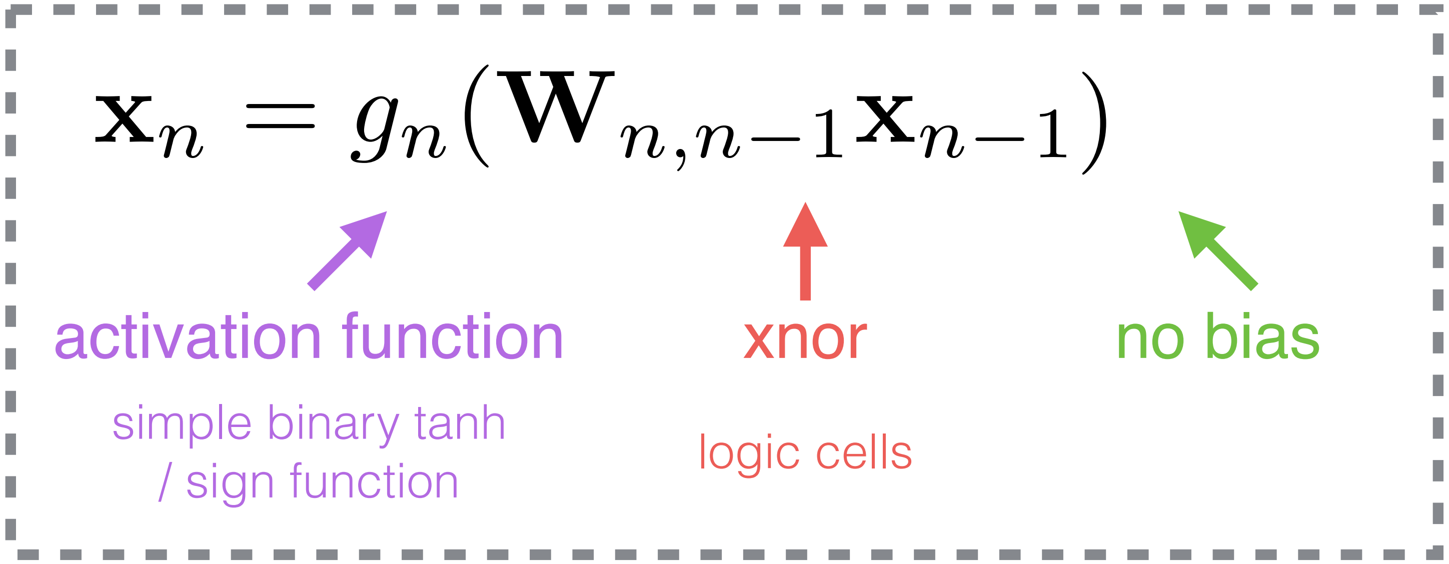
precomputed and
stored in BRAMs

multiplication

DSPs

addition

logic cells



$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1})$$

activation function

simple binary tanh
/ sign function

xnor

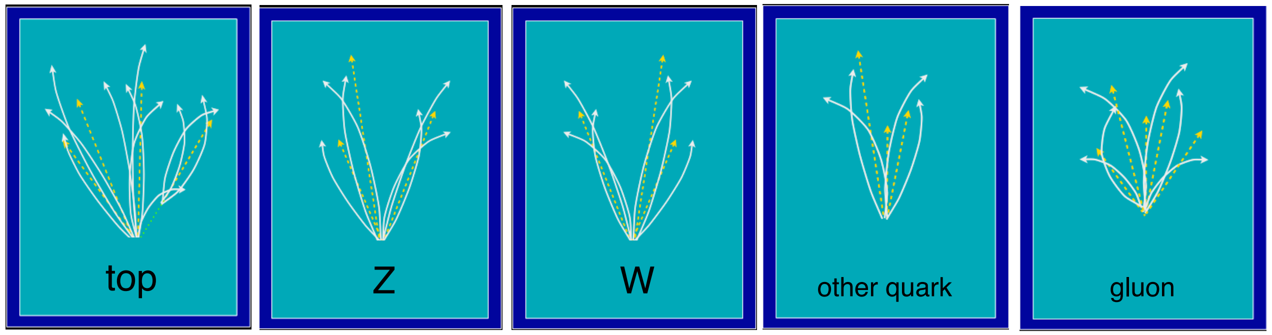
logic cells

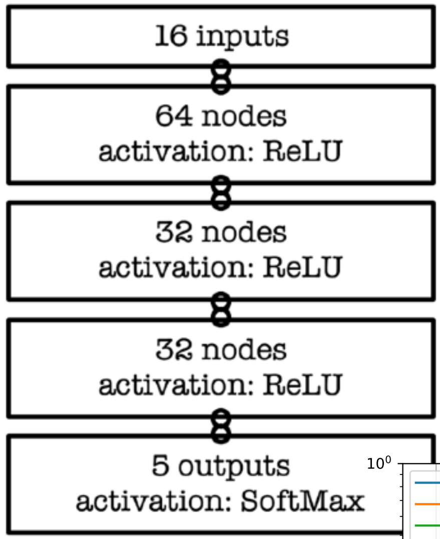
no bias

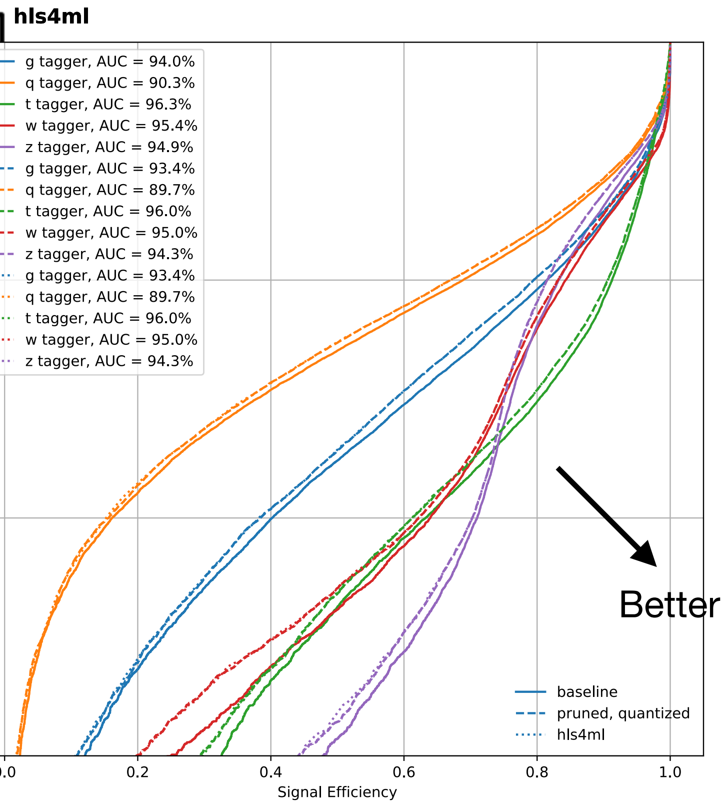
Binarized: 1-bit weights

Impact of QAT & Pruning

- From the hls4ml tutorial
  - Tagging jets (5 classes q/g/t/W/Z, 16 input variables)
- 3 hidden layer MLP (Dense layers):
  - 1) Keras floating point training, 16b inference
  - 2) QKeras with 6 bits for weights, biases, activations & 75% sparsity target with TFMOT
  - Minimal code changes required to go from 1) to 2)







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

Summary

- This was a whirlwind introduction to Machine Learning, its applications in HEP, and emerging use in Trigger and DAQ
- It 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
- GPUs are great ML accelerators, and starting to appear in DAQ systems
    - We went through some methods for deployment and optimization
- I've shown the hls4ml package for running ML inference in sub-microsecond latency on FPGAs (for L1T) and even ASICs
    - For more: fastmachinelearning.org/hls4ml