Machine Learning
(for Trigger and Data Acquisition)

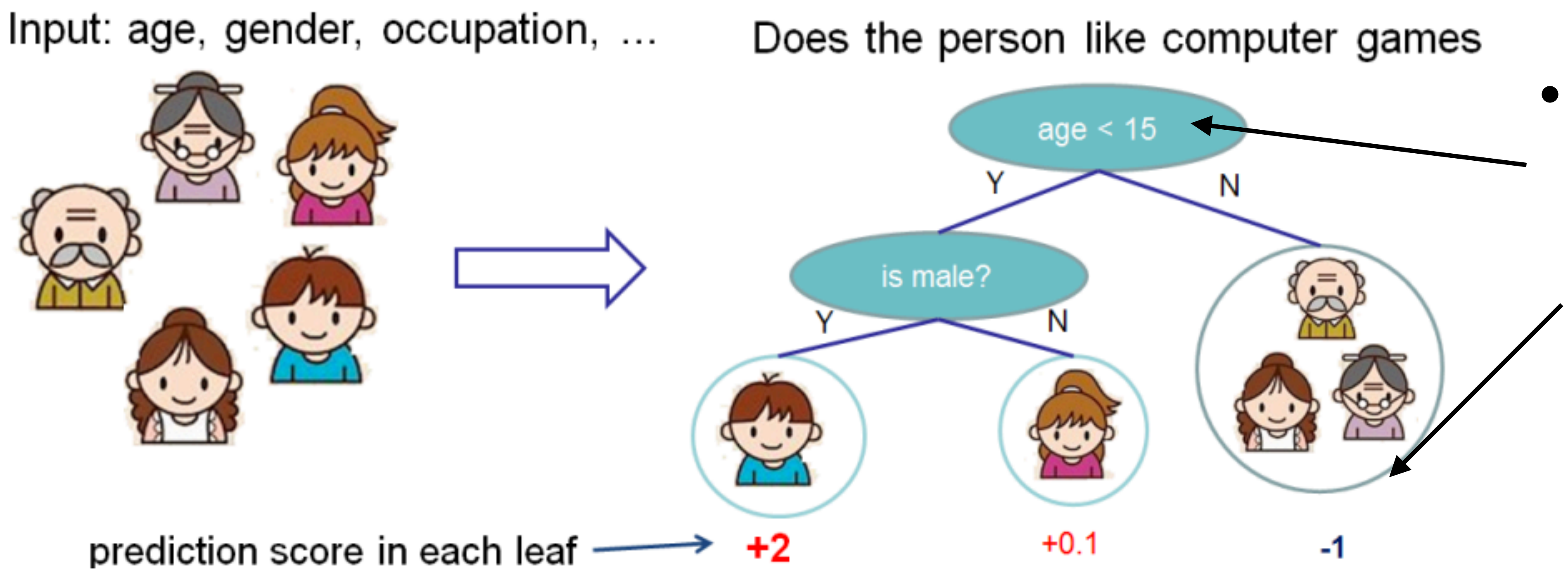
ISOTDAQ
Valencia, Spain
17/01/2020
Sioni Summers (CERN)

Contents

- Introduction to Machine Learning
  - Neural networks - basics and different types
  - Examples in HEP
- Machine Learning in Trigger and DAQ
  - Hardware and software for ML
  - Examples
  - Fast Machine Learning: hls4ml

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


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

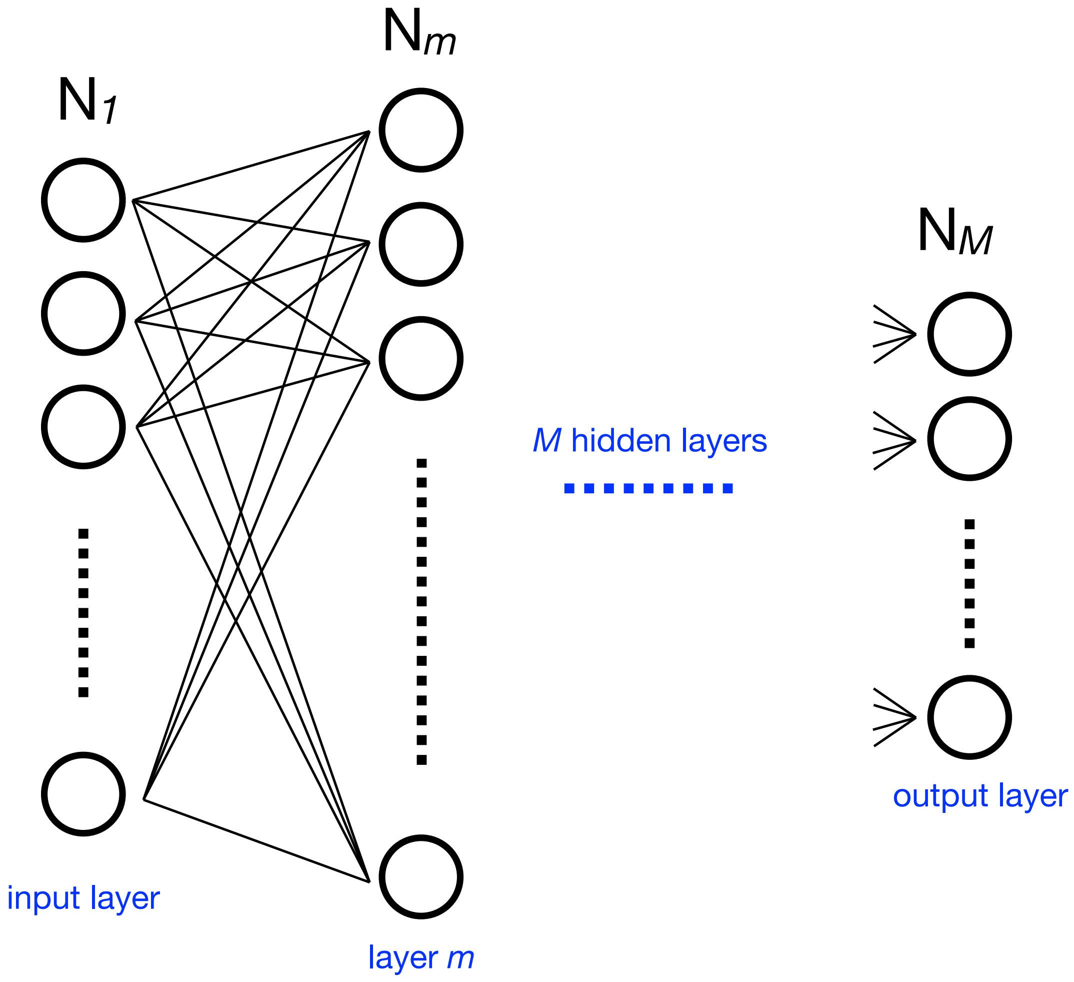
- Decision tree thresholds and prediction probabilities are learned from the training data

https://arxiv.org/abs/1603.02754

- 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

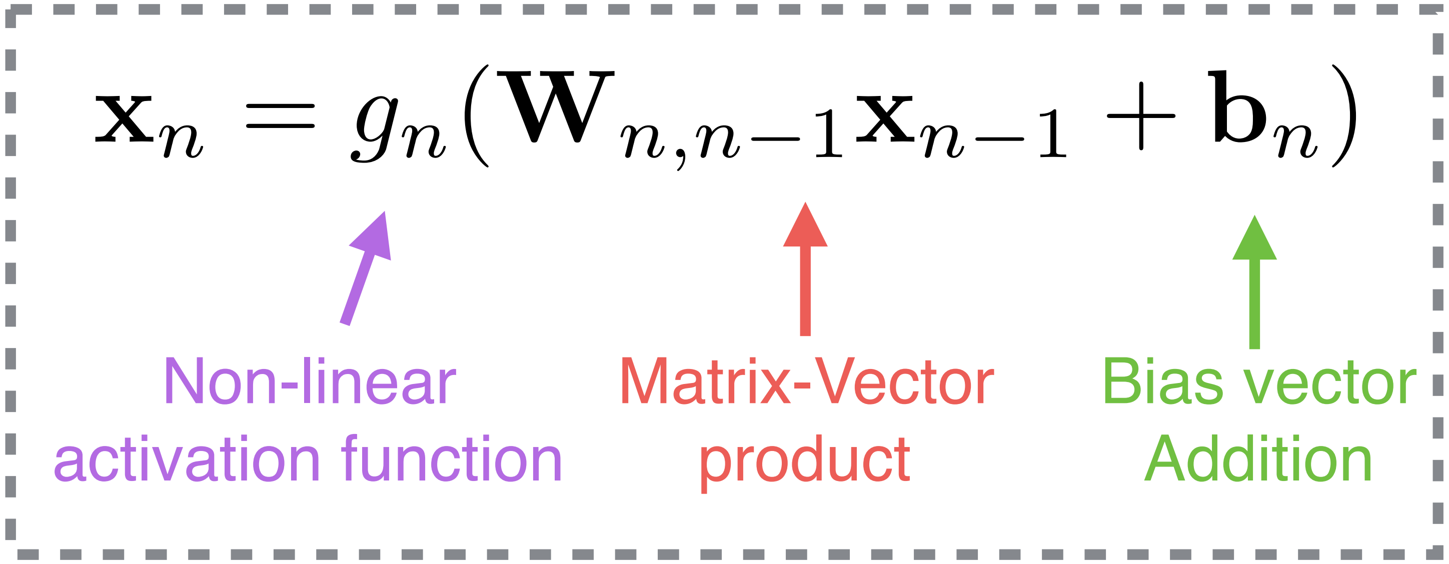
Neural Networks

- Model 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

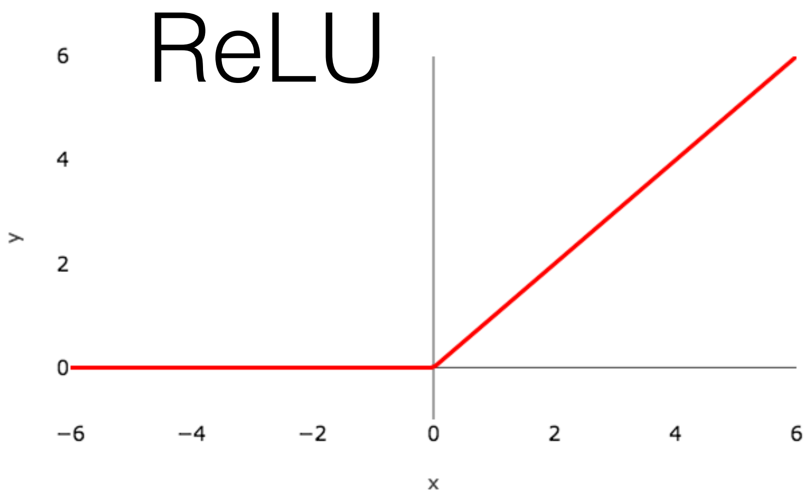


$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

Non-linear activation function

Matrix-Vector product

Bias vector Addition



ReLU

- 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

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

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
- Run network compression / pruning after training: improve robustness of your NN, *and* improve computational performance

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

```
from keras.models import Sequential
from keras.layers import 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)
nn = Sequential()
nn.add(Dense(64, activation=‘relu’, input_shape=2,
name=‘hidden’))
nn.add(Dense(2, activation=‘softmax’, name=‘output’))
nn.compile()
nn.fit(X_train, y_train, batch_size=100)
nn.save(‘nn.h5’)
```

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

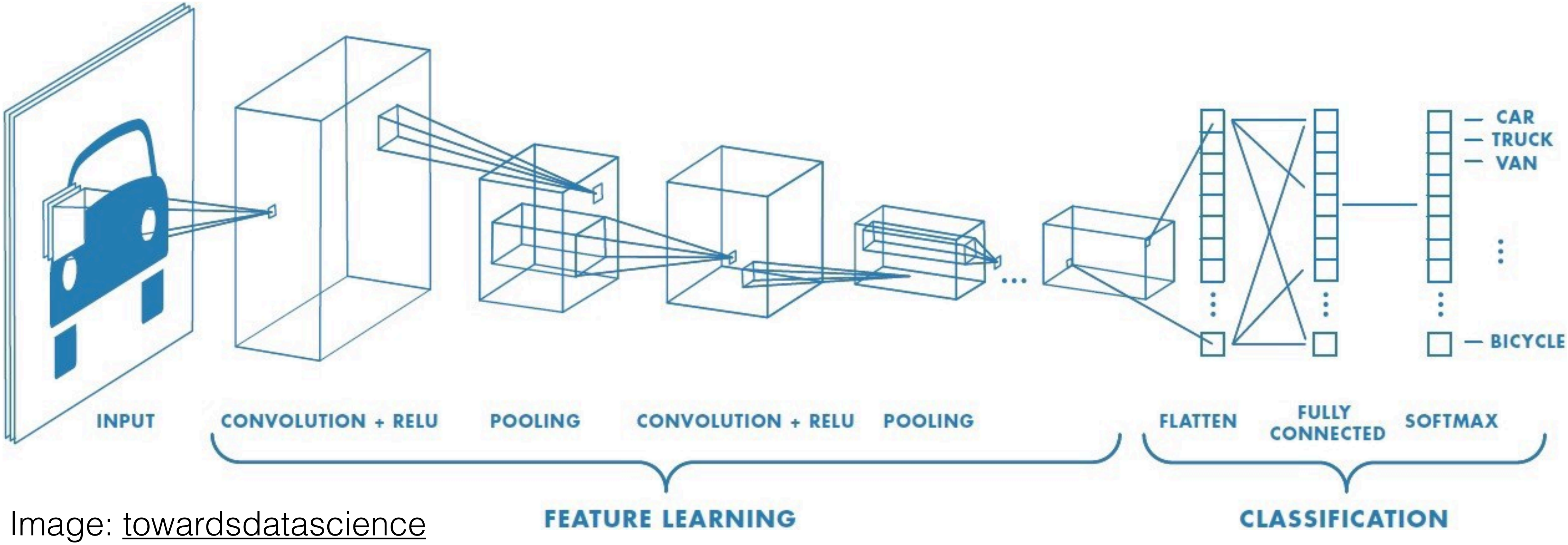


Image: towardsdatascience

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)

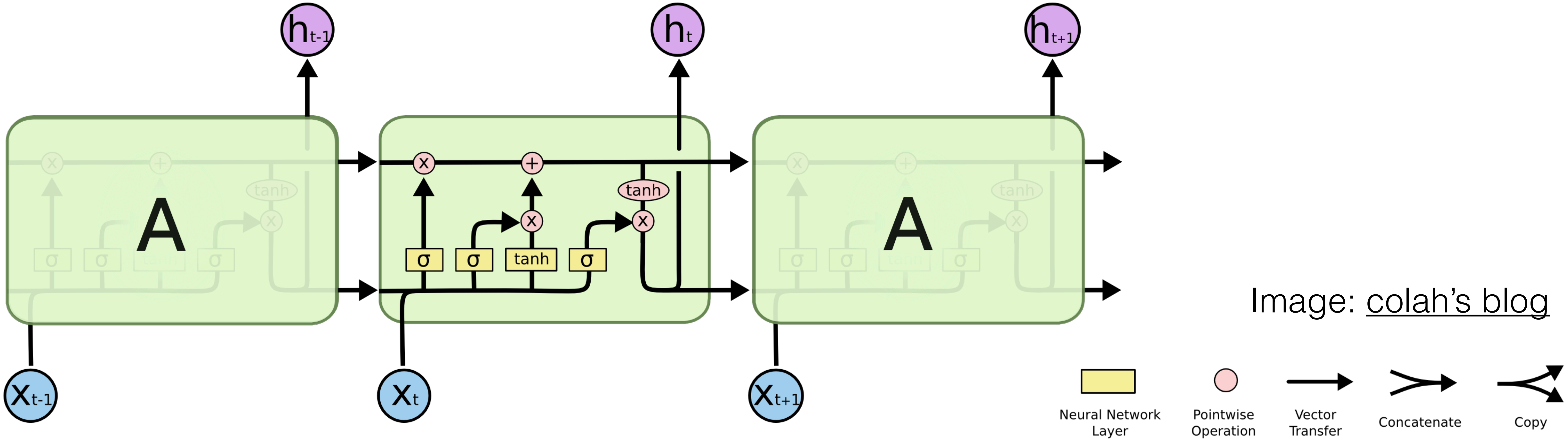
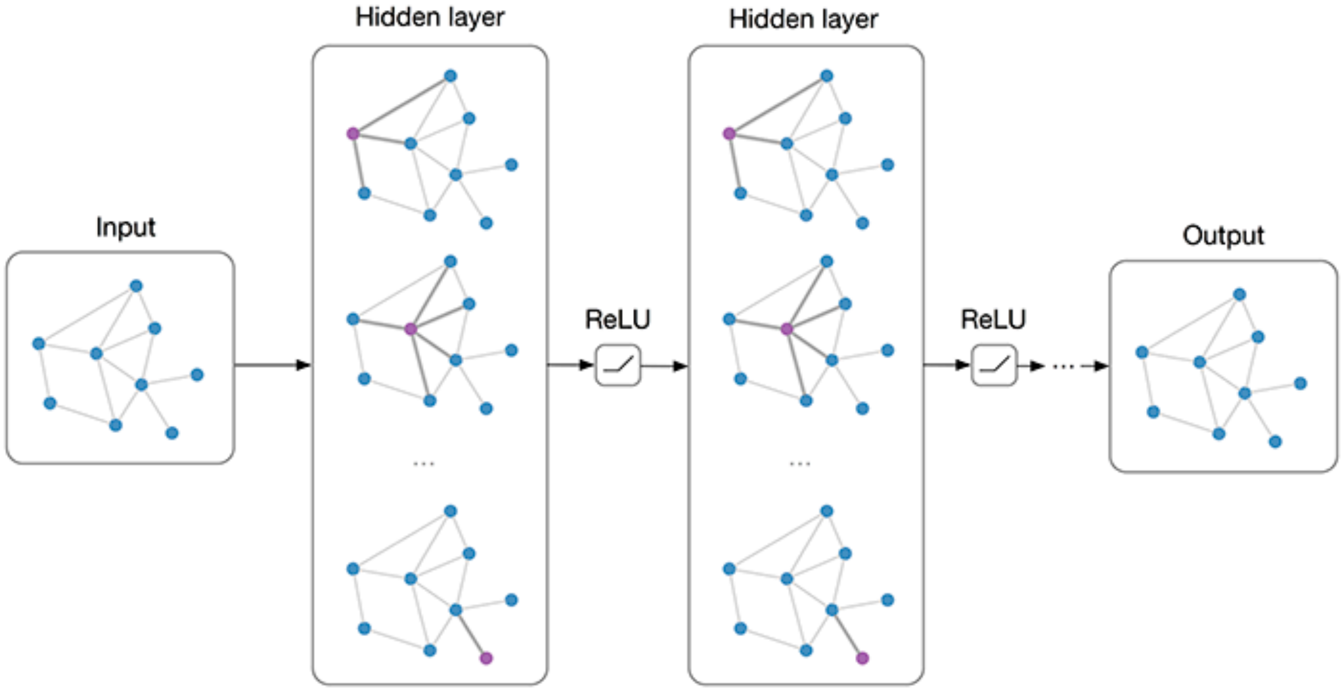


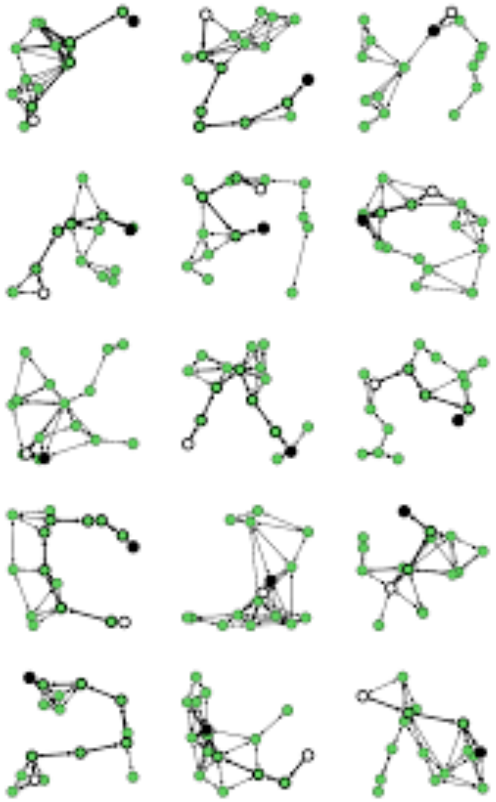
Image: colah’s blog

- 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

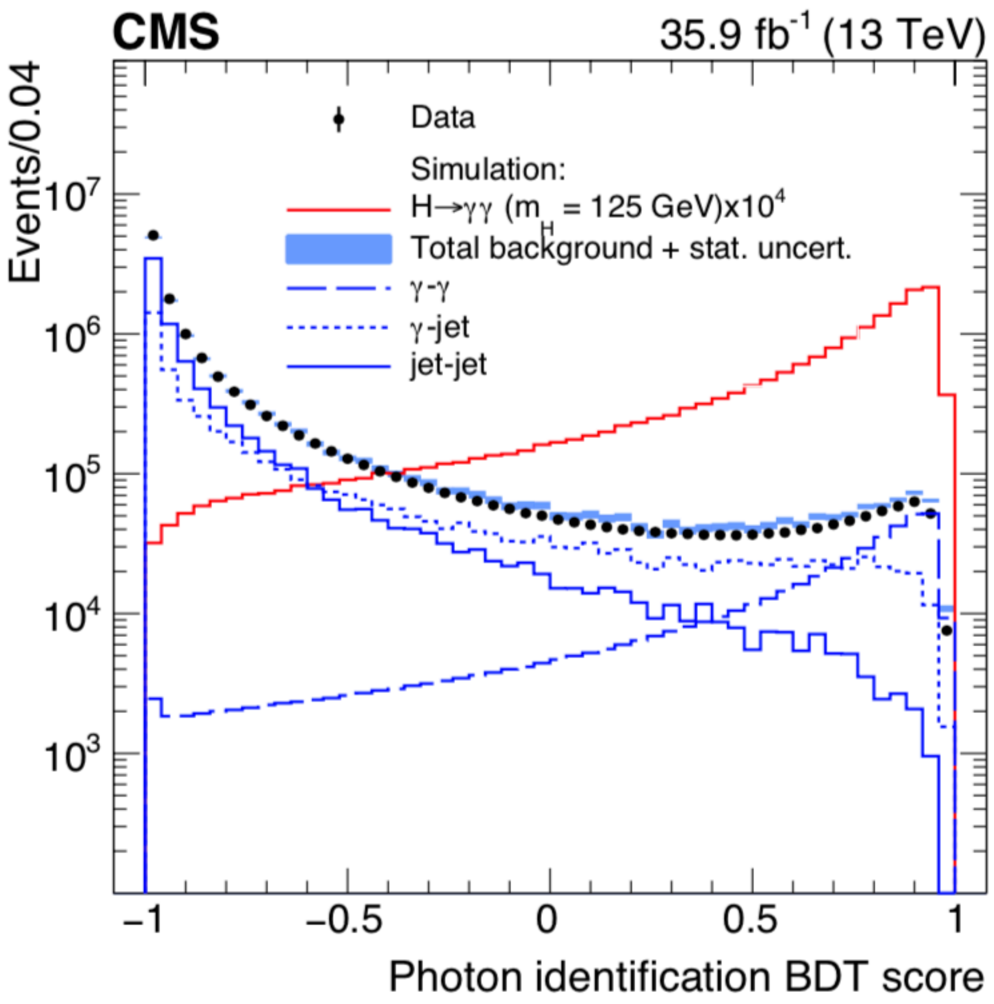
- We've seen NNs suitable for applying on 'high level features' (Fully Connected), images (Convolutional), and time series (Recurrent)
- Graph networks are well suited to problems described by graphs of vertices and edges
- Cluster / classify data not only according to its coordinates, but its neighbourhood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





BDTs for Higgs

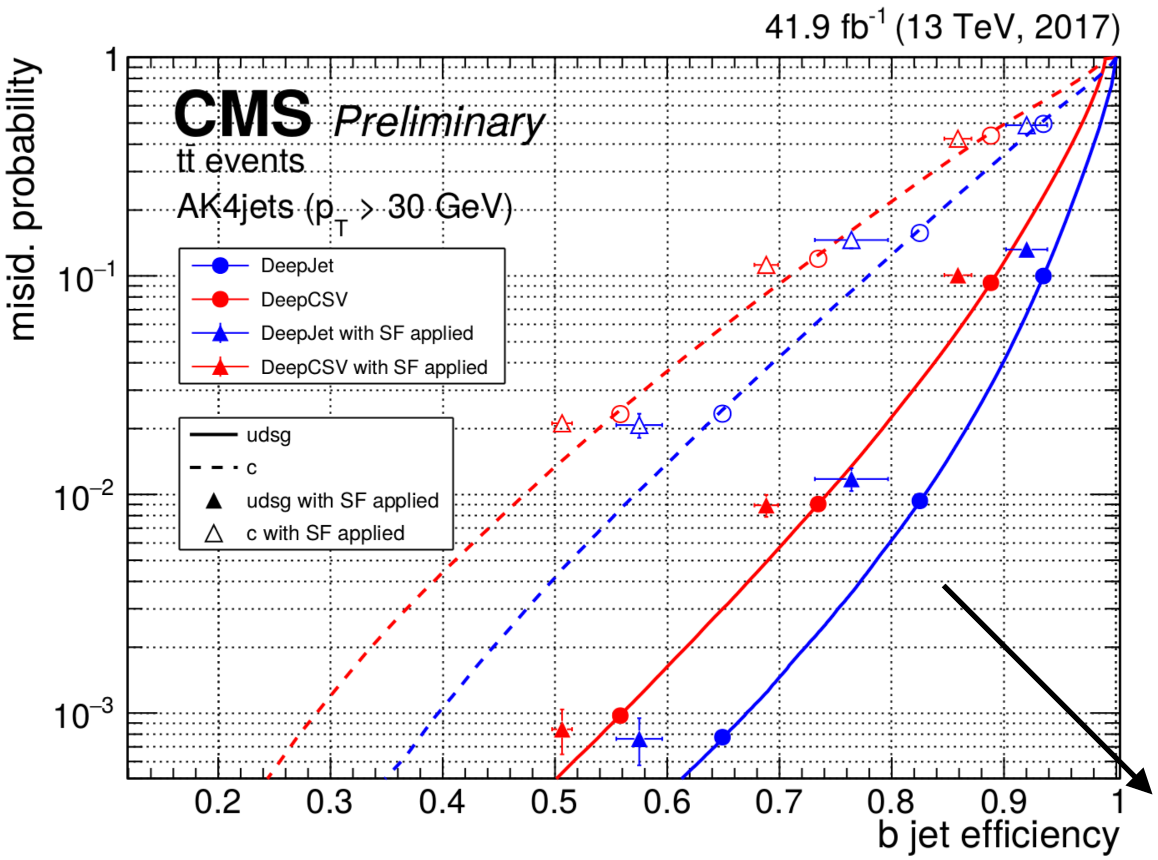
- Several BDTs involved in the analysis of Higgs boson decay to two photons using high-level variables
    - e.g. particle mass, η, isolation
- To separate signal photons from background (photons from jets)
- Choosing the most likely vertex for the photons (neutral, so no tracking)
- A diphoton quality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

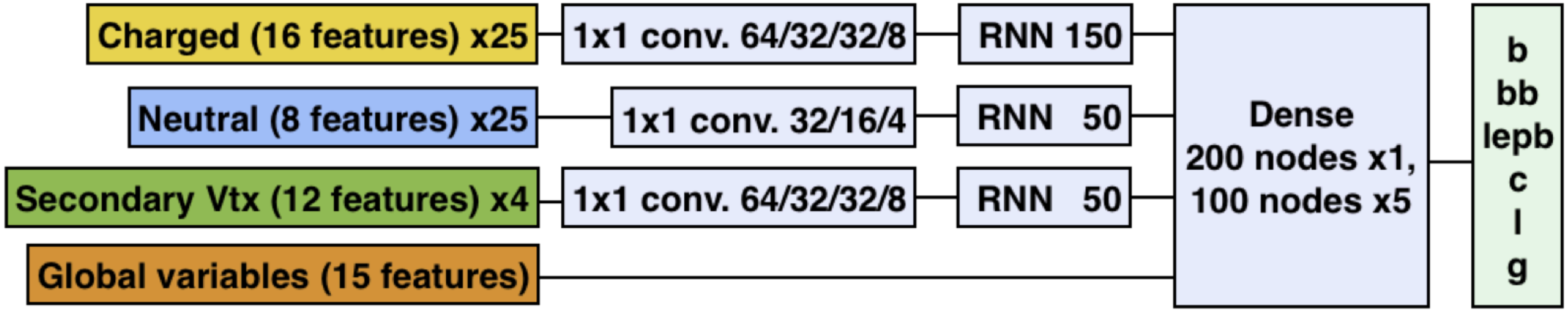
Deep NN Jet Tagging

- 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



better

CMS-DP-2018-058



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

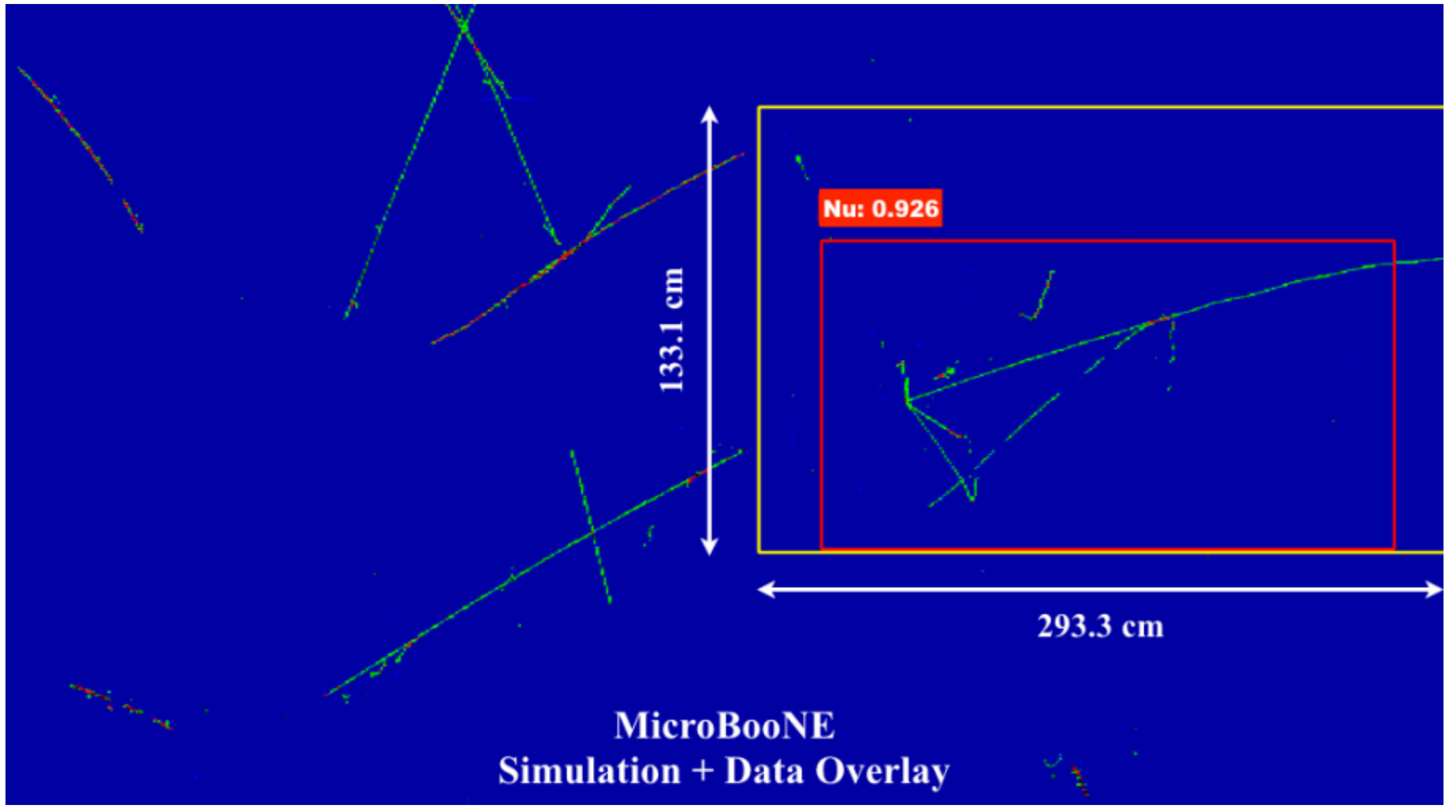
RNN 50

Dense 200 nodes x1, 100 nodes x5

b
bb
lepb
c
l
g

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

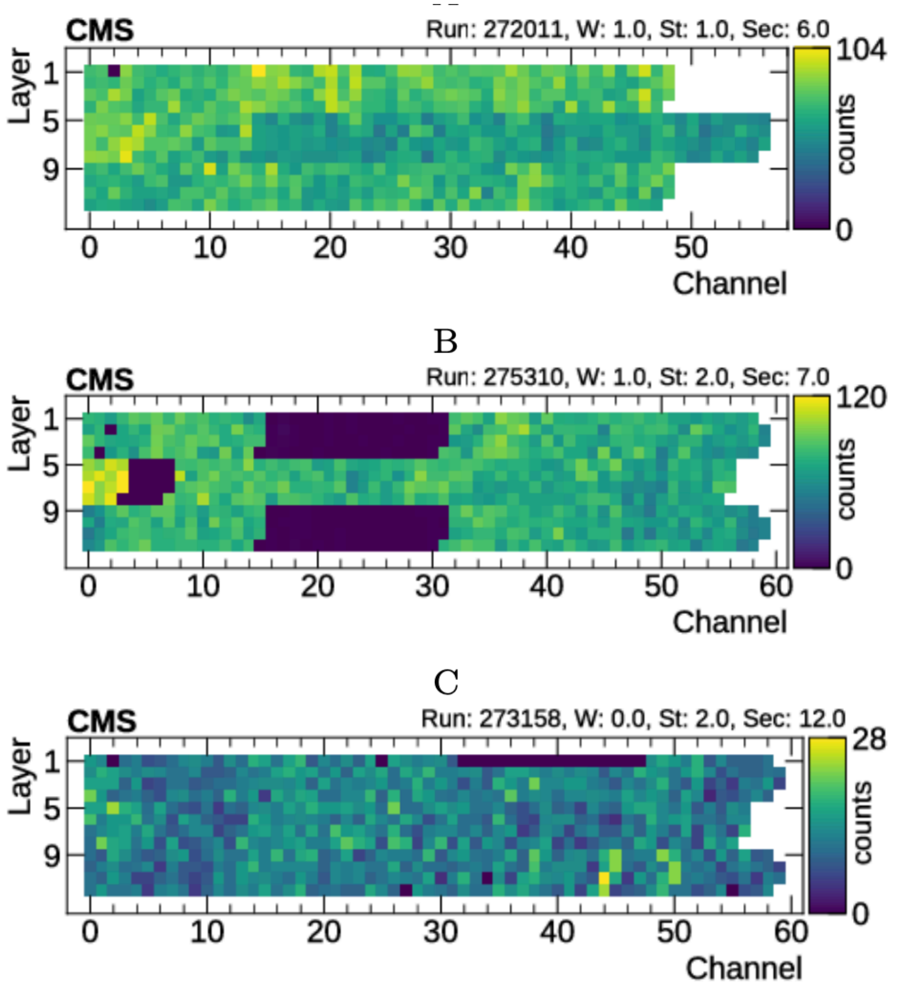


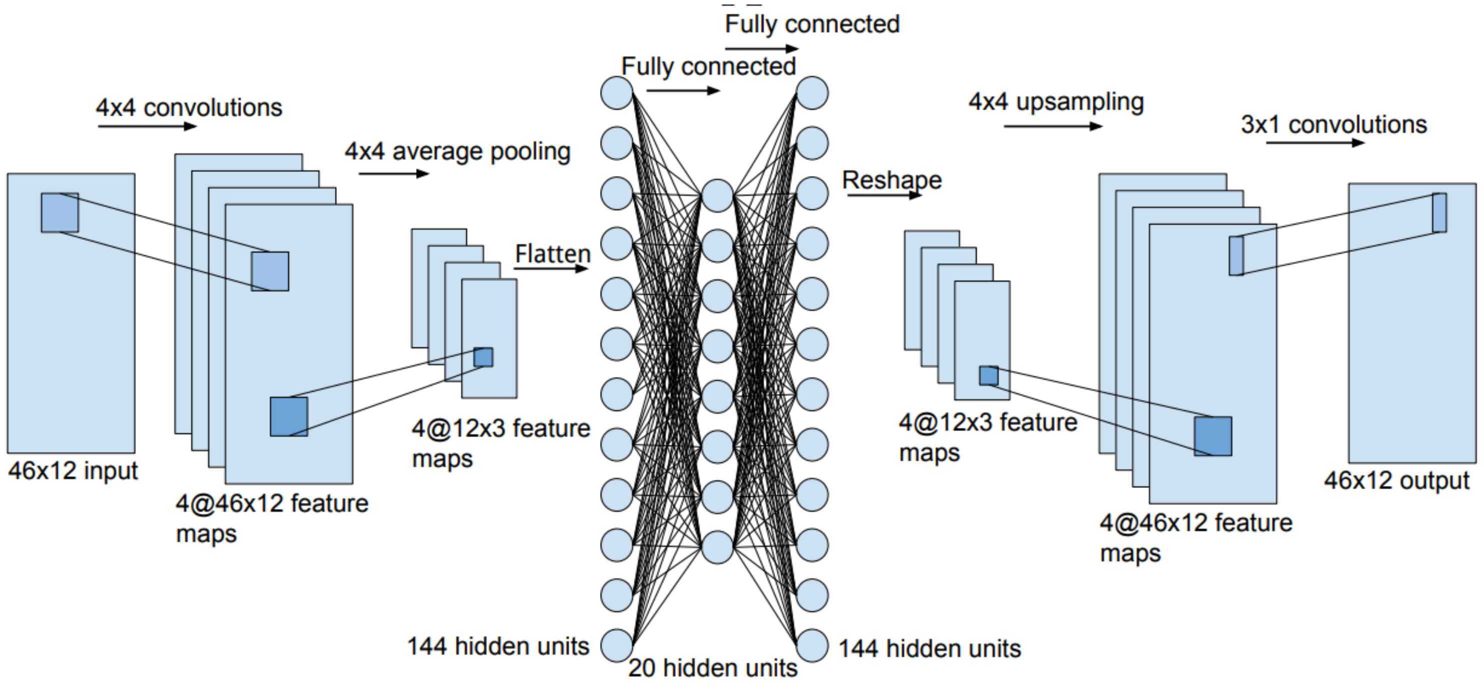
arxiv:1611.05531

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

Data Quality Monitoring

- Using an Autoencoder for anomaly detection
  - Network has a ‘bottleneck’, learning an abstract representation of the data
  - After bottleneck, 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

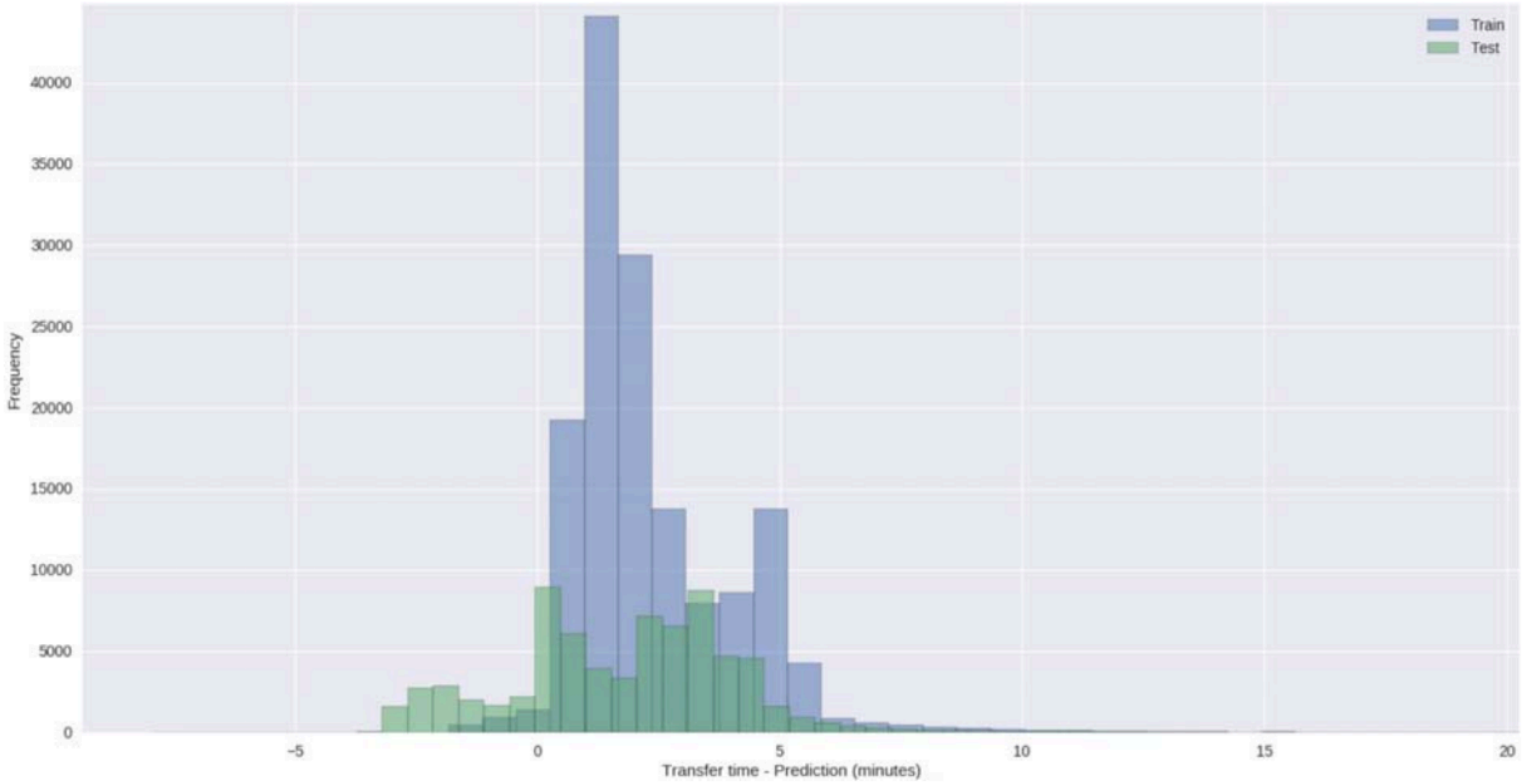




arxiv:1808.00911

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: Hardware

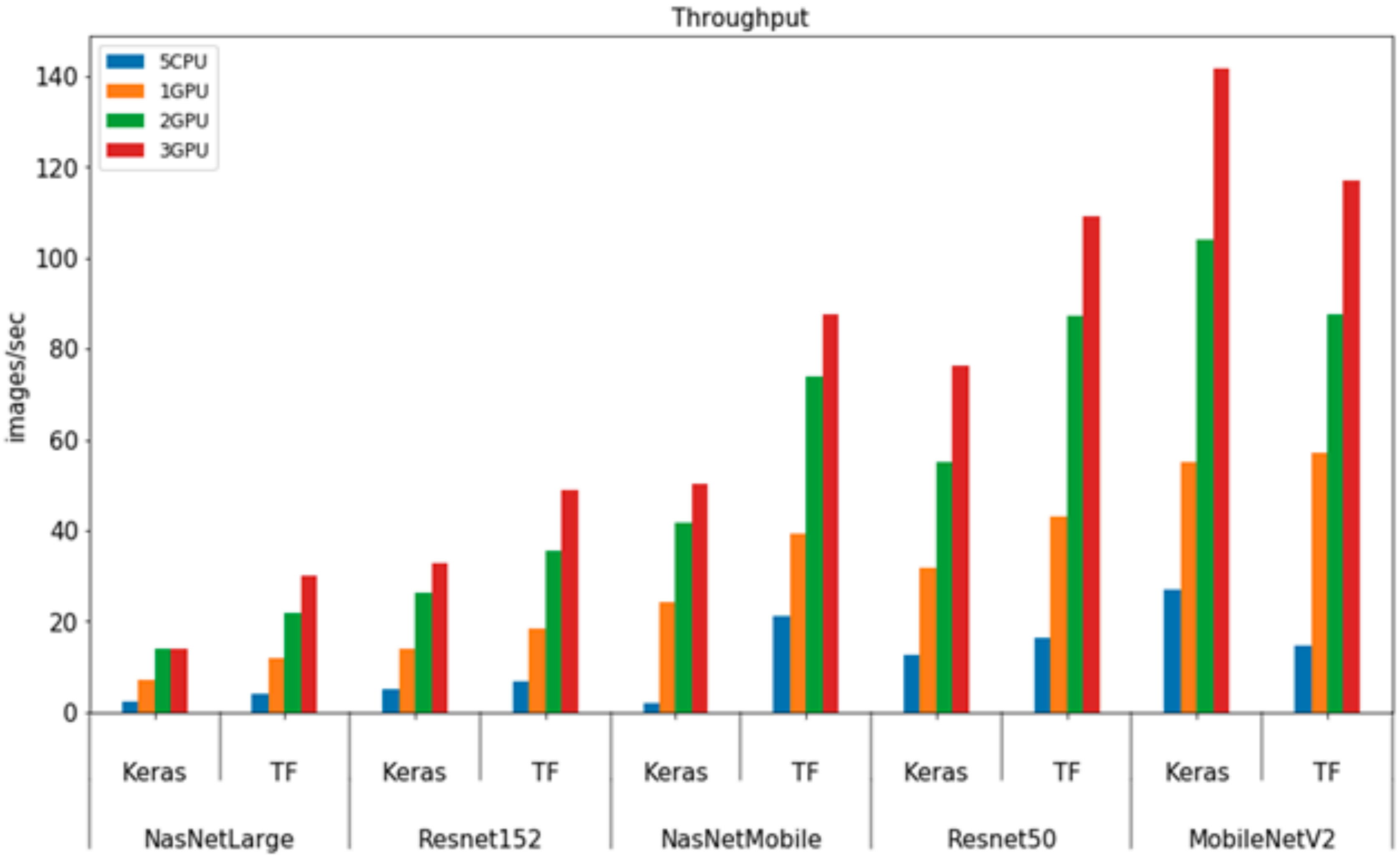
- Machine Learnings algorithms are highly parallelisable
    - Recall Neural Network forward pass is matrix-vector products and non-linear functions on vectors
- Can be accelerated with appropriate hardware:
    - CPUs with vector/SIMD units (e.g. AVX)
    - GPU, FPGA, TPU (T = Tensor)
- ML is also big business, so lots of high performance solutions out there
- Often for Trigger and DAQ we can ‘train offline’, ‘predict online’
    - Could use different hardware for each phase
- In the context of a possible heterogenous compute future in HEP, will need to accommodate ML processors

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
- 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 can also outcompete CPUs in inference
- Here, running inference on K80 GPUs, measuring images / second (throughput)

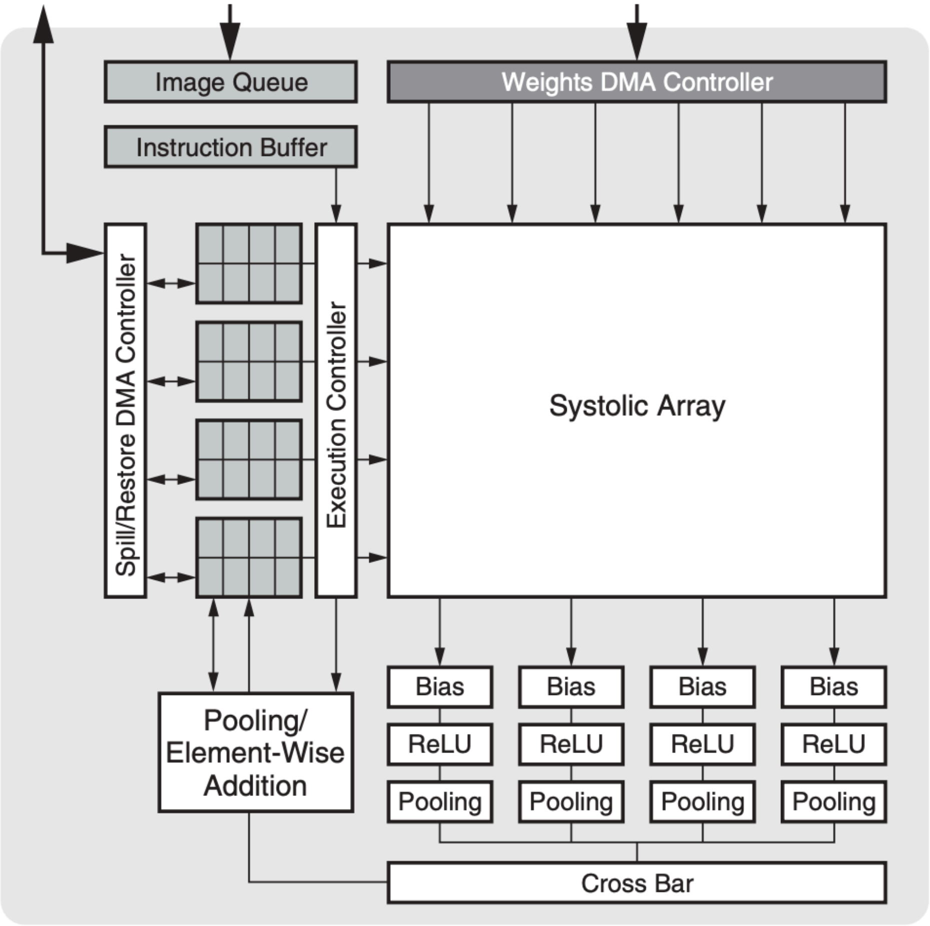


From Microsoft Azure

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

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, 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 Specific Processors

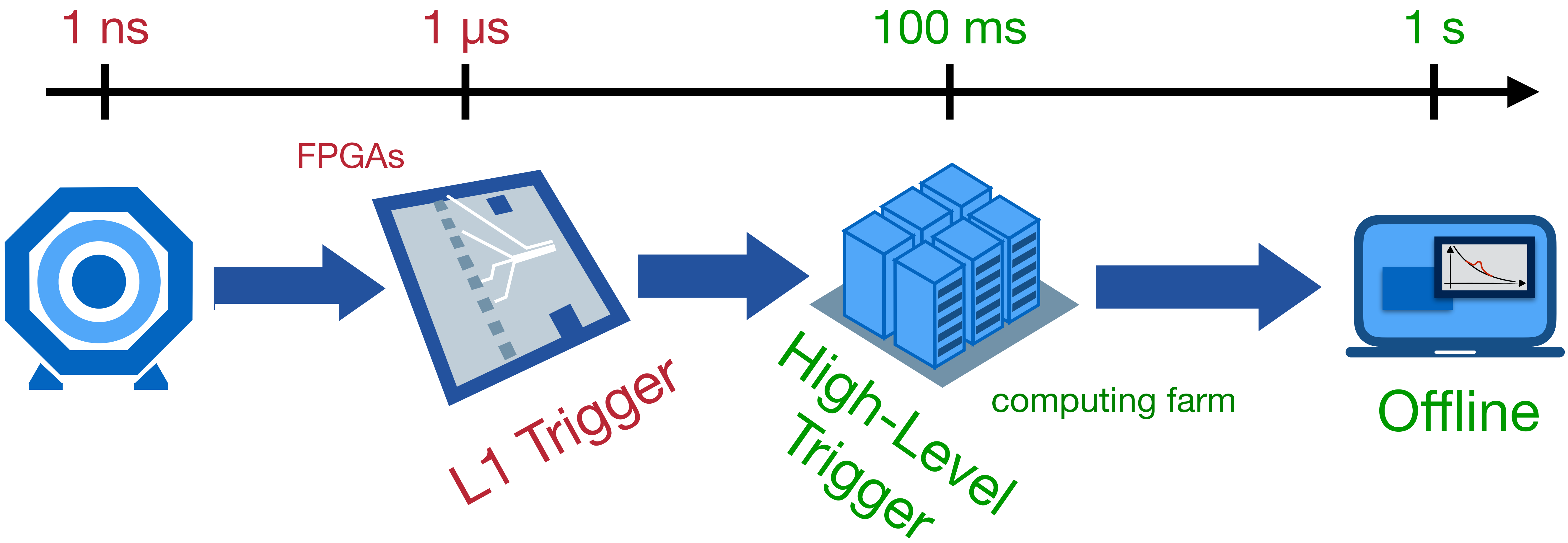
- There are some processors out there specifically designed for Machine Learning / AI
- e.g. Tensor Processing Unit (TPU) from Google
- Devices aiming at low power embedded
  - Internet of Things, Smartphones
- Soon: Xilinx Versal ACAP for FPGAs with embedded Vector units (or vice versa?)
- Many different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

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 as the main selection path for most LHCb analyses

Machine Learning at L1 Trigger

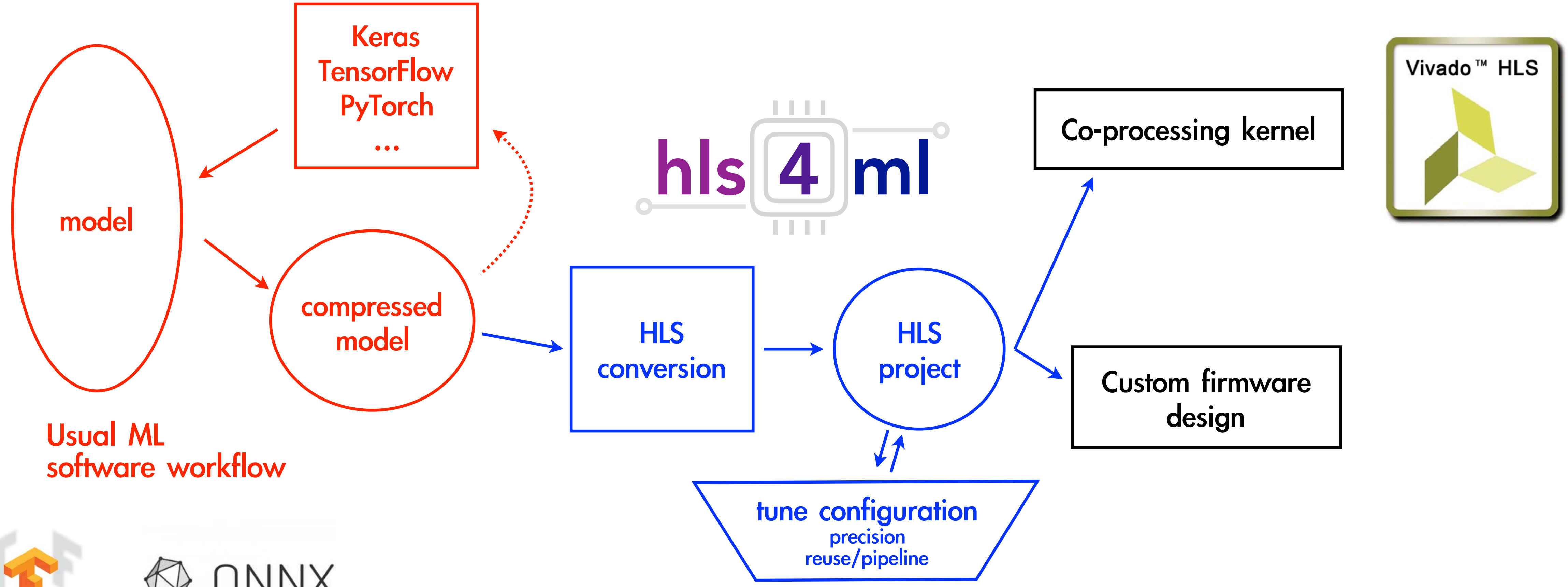


- Typical 'latency landscape' of LHC experiment
- To deploy Machine Learning at the L1 Trigger need to:
  - Be able to execute ML algorithms in O(1µs)
  - Execute these algorithms on FPGAs

high level synthesis for machine learning

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

- reads as input models trained with standard ML libraries (keras, PyTorch, onnx)
- uses Xilinx HLS software (more accessible to non-FPGA-expert)
- 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

hls4ml

HLS conversion

HLS project

tune configuration
precision
reuse/pipeline

Co-processing kernel

Custom firmware design

Vivado™ HLS

https://hls-fpga-machine-learning.github.io/hls4ml/

https://arxiv.org/abs/1804.06913

What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

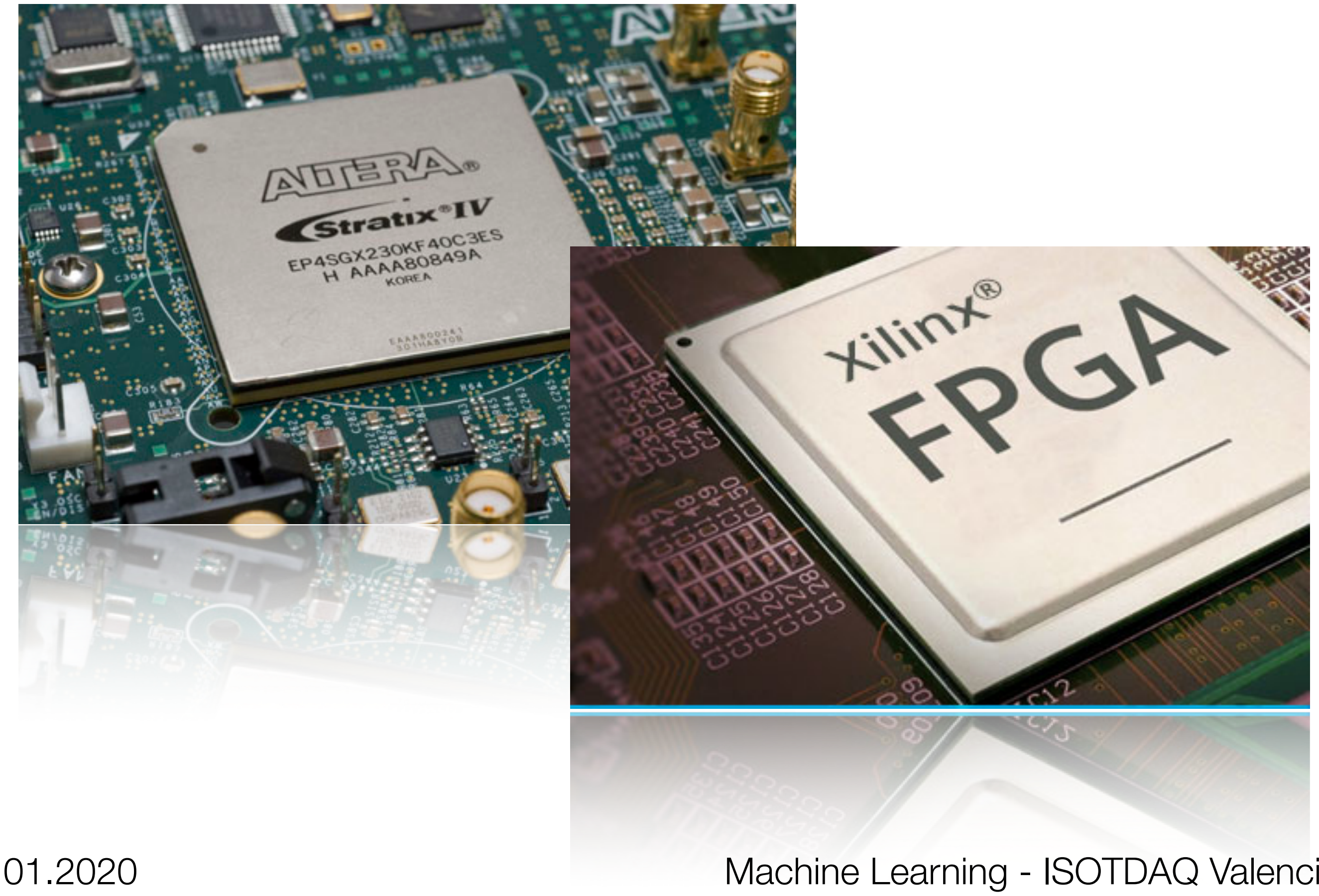
See talks “Introduction to Field Programmable Gate Arrays”, “Advanced FPGA Programming”, and Labs “FPGA Programming”, “SoC FPGA” at this school

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

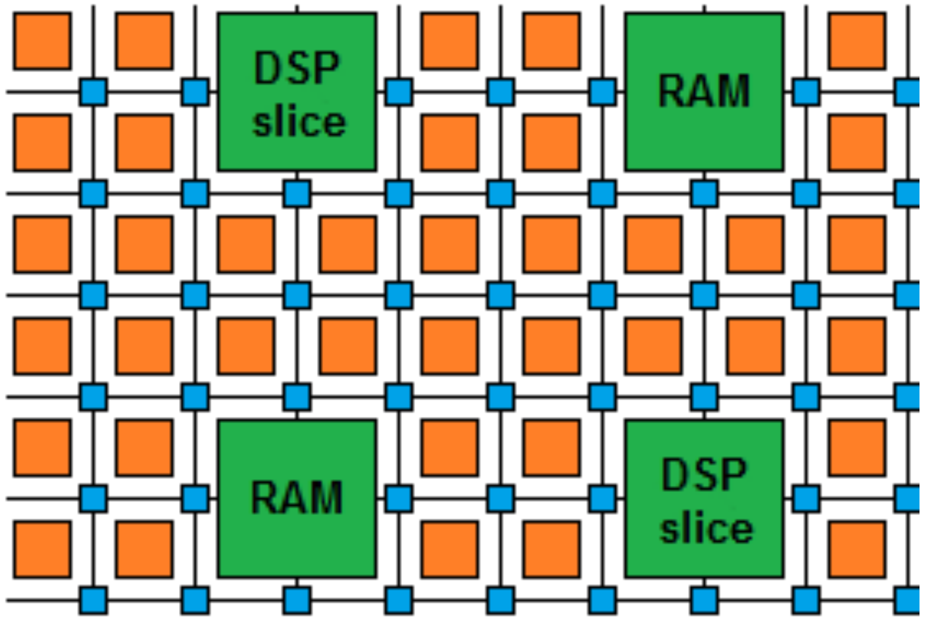
Extremely parallel processors

‘Computing in space as well as time’

Processing workhorse of LHC triggers



FPGA diagram



LUTs - generic logic

DSPs - for multiplication

BRAM - for local, high-throughput storage

High Level Synthesis

- Usually, 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), one writes 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 compiler directives to guide the compiler
- These also provide a powerful handle for optimisation not accessible to HDL developers

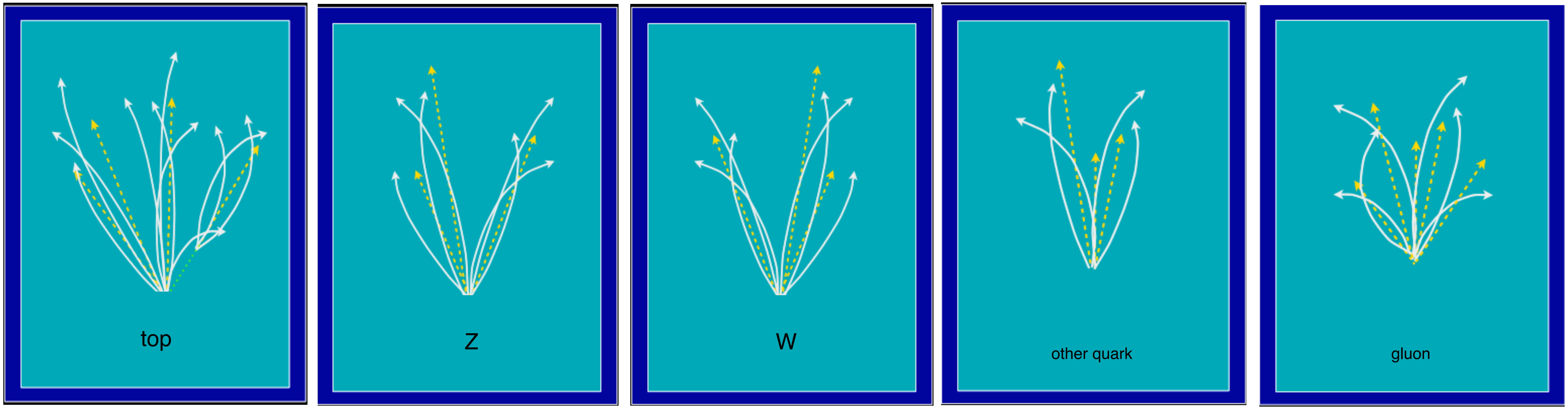
```
#define N 16
typedef ap_fixed<16,8> T;

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

Physics case: jet tagging

Study a **multi-classification task to be implemented on FPGA:** discrimination between highly energetic (boosted) ***q, g, W, Z, t*** initiated jets



$t \to bW \to bqq$

3-prong jet

$Z \to qq$

2-prong jet

$W \to qq$

2-prong jet

q/g background

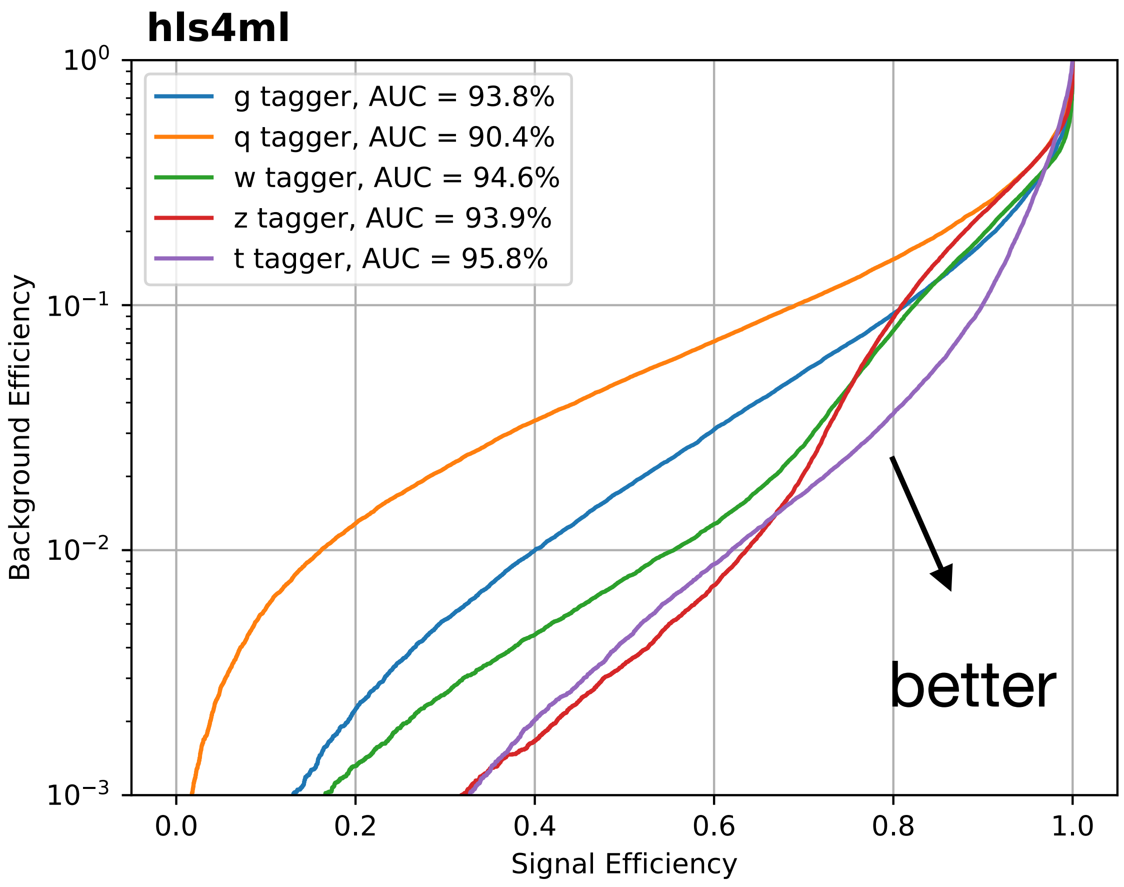
no substructure
and/or mass ~ 0

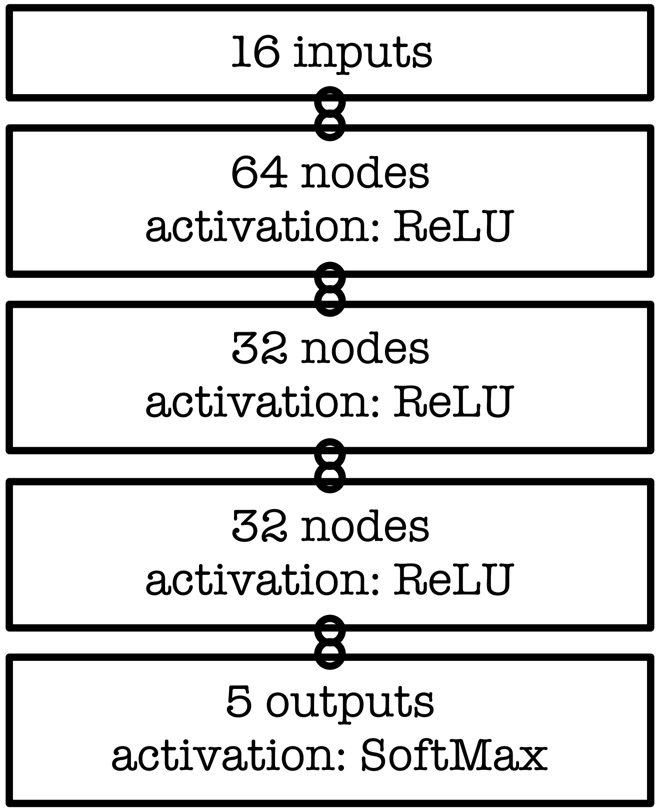
Reconstructed as one massive jet with substructure

Physics case: jet tagging

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

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




16 inputs

64 nodes
activation: ReLU

32 nodes
activation: ReLU

32 nodes
activation: ReLU

5 outputs
activation: SoftMax

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

Efficient NN design for FPGAs

FPGAs provide huge flexibility

Performance depends on how well you take advantage of this

Constraints:

Input bandwidth

FPGA resources

Latency

With `hls4ml` package we have studied/optimized the FPGA design through:

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

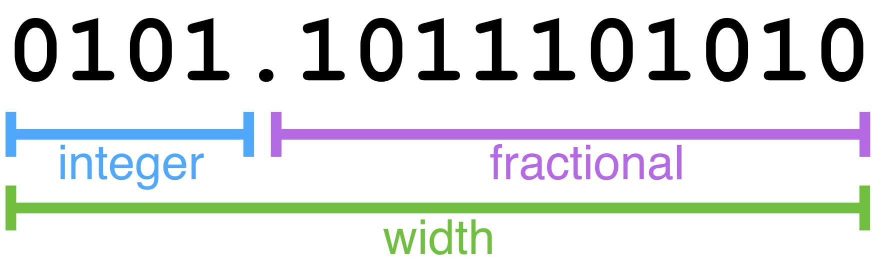
NN TRAINING

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

FPGA PROJECT DESIGNING

Efficient NN design: quantization

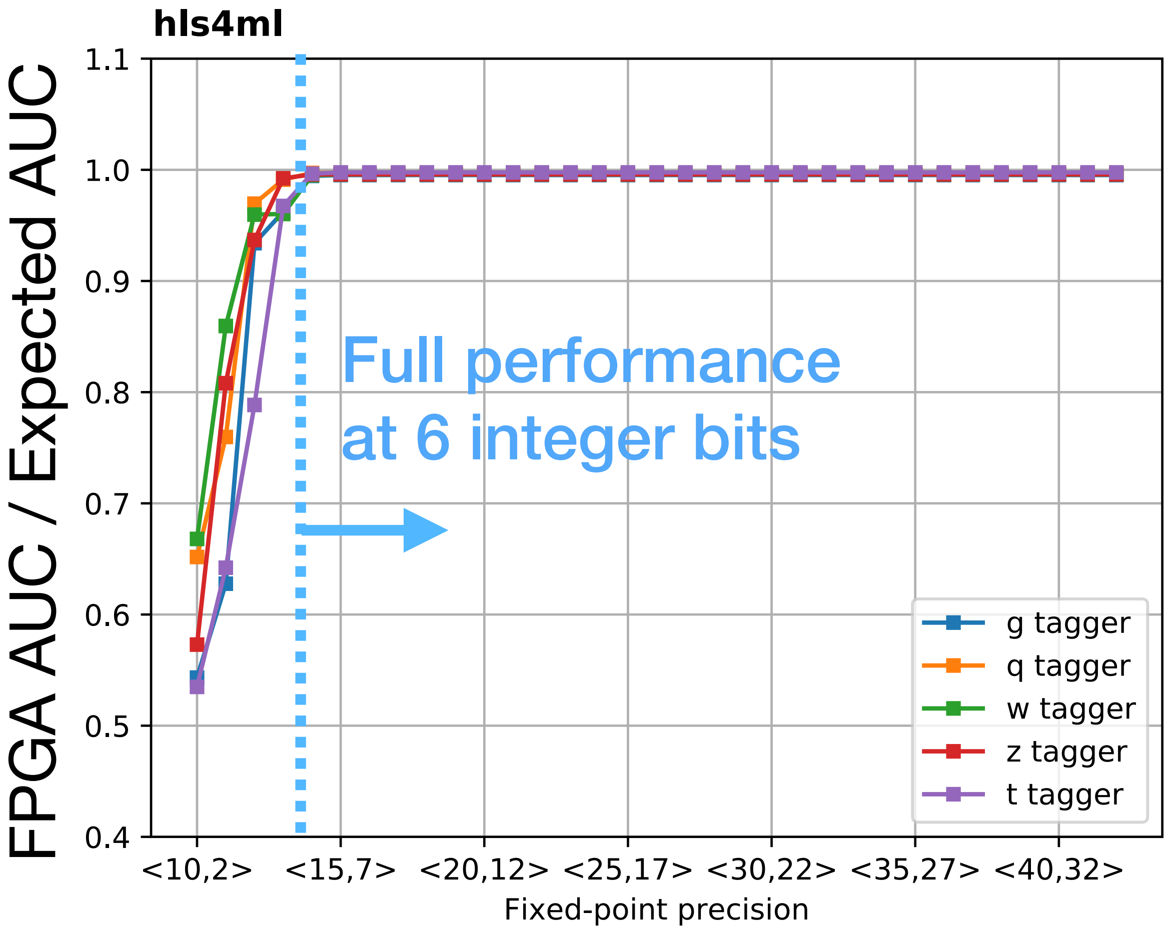
ap_fixed<width,integer>



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

Scan integer bits

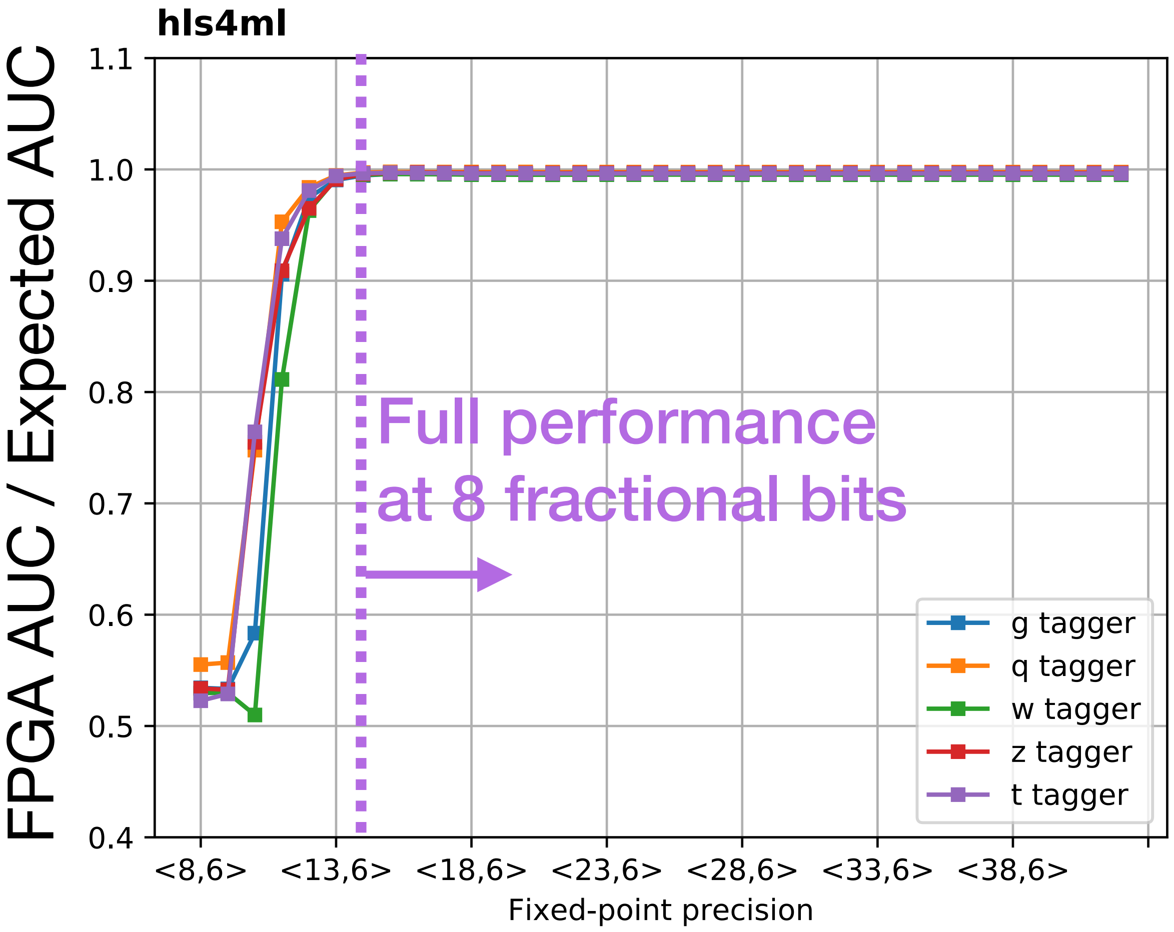
Fractional bits fixed to 8



Full performance at 6 integer bits

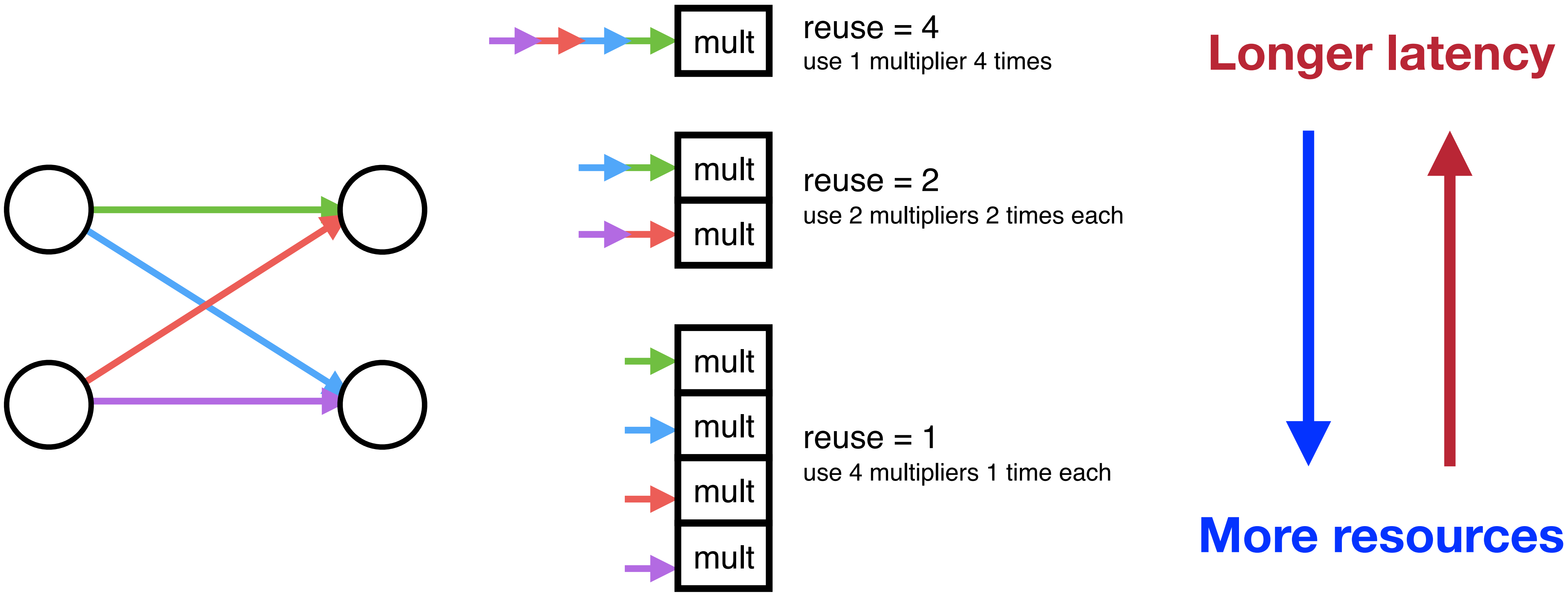
Scan fractional bits

Integer bits fixed to 6



Full performance at 8 fractional bits

Efficient NN design: reuse



mult

reuse = 4
use 1 multiplier 4 times

mult
mult

reuse = 2
use 2 multipliers 2 times each

mult
mult
mult
mult

reuse = 1
use 4 multipliers 1 time each

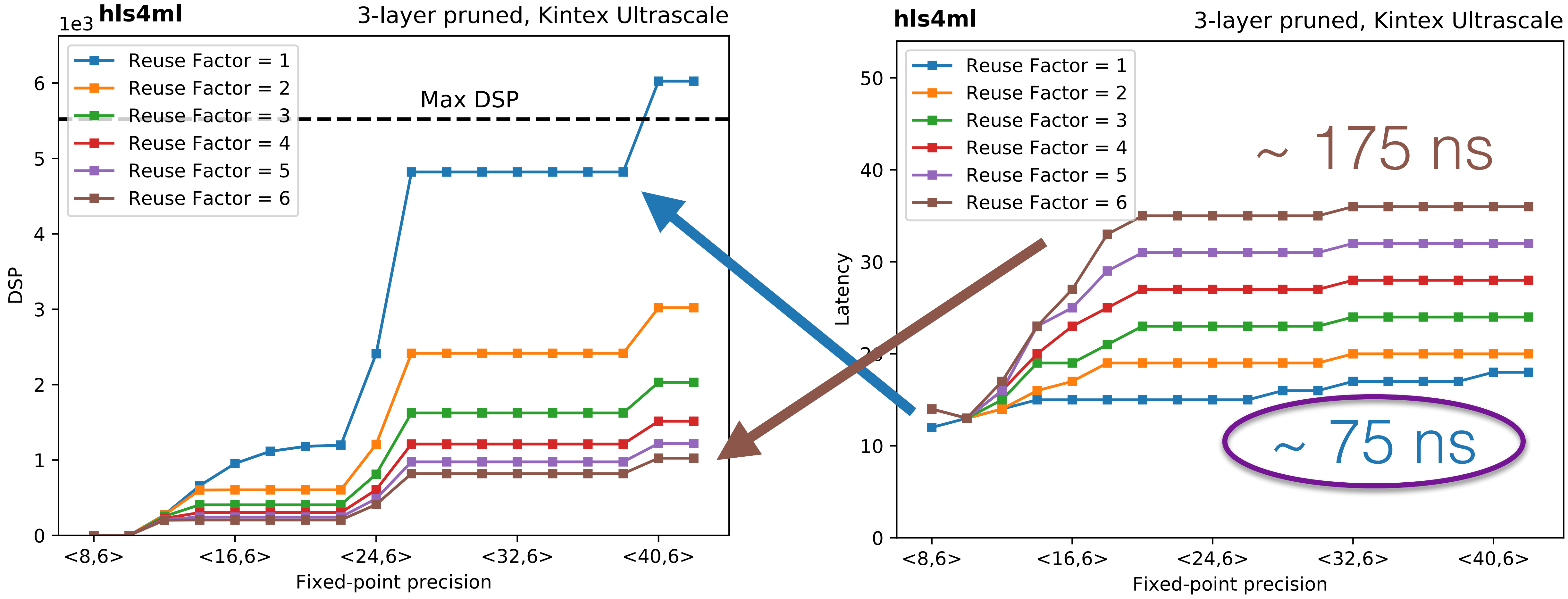
Longer latency

More resources

- Key feature of **hls4ml**: a handle to trade resource usage and latency/throughput
- Reuse = 1: fully unroll everything onto different resources
  - Fastest, most resource intensive
- Reuse > 1: one resource used sequentially for several operations
  - Slower, but save resources

Parallelisation

- Low reuse gives lowest latency, most resource usage
- High reuse gives longer latency, lower resource usage
- *Throughput* decreases with increasing reuse
- A large enough model will use all of the resources with reuse=1, so sometimes must increase it

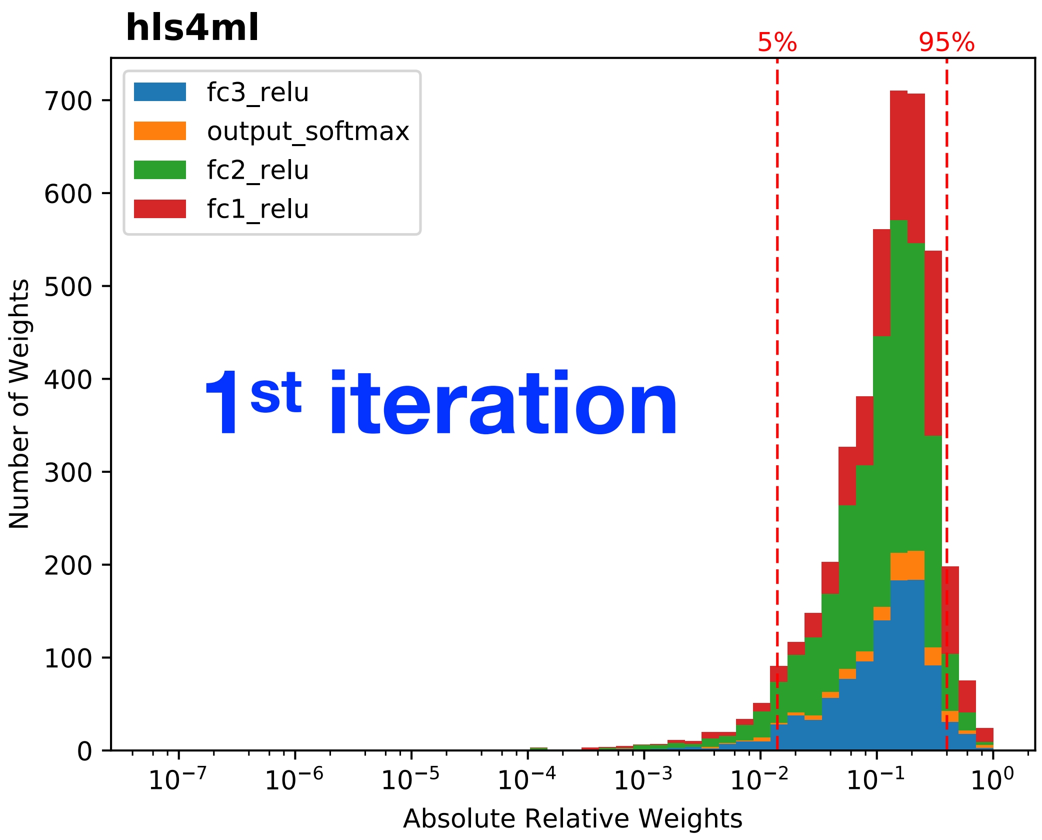


Efficient NN design: compression

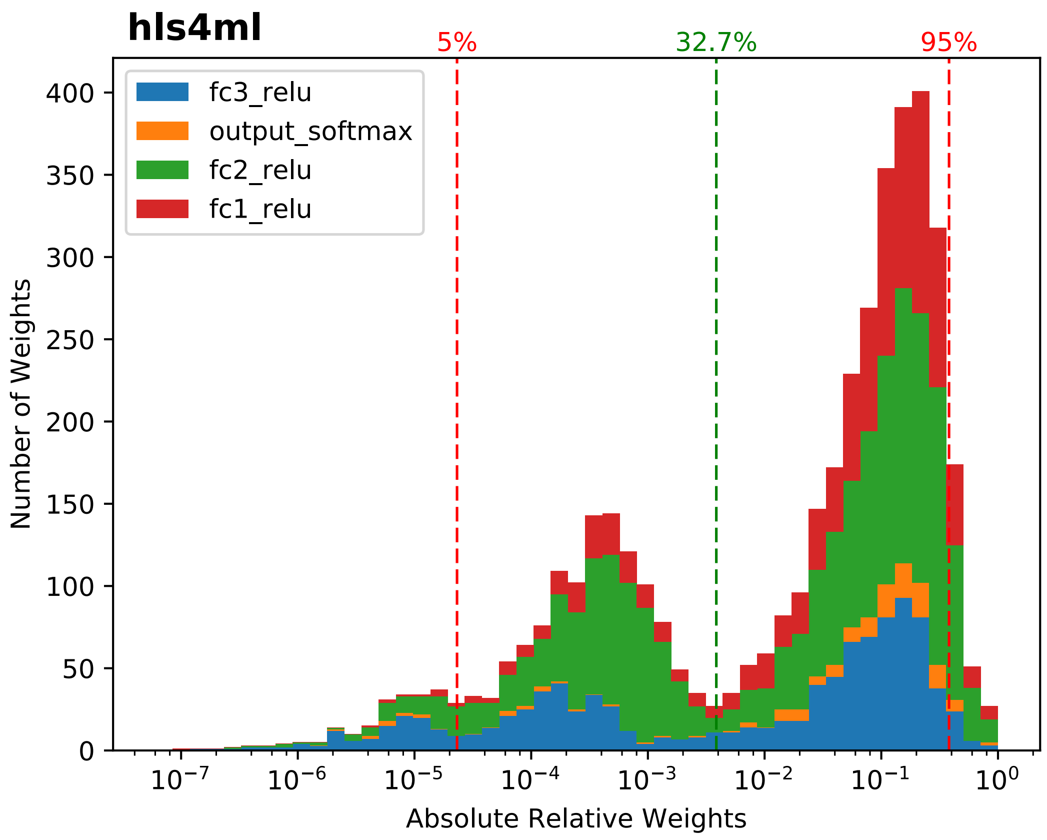
- Iterative approach:
    - train with **L1 regularization** (loss function augmented with penalty term):

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

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



Train with L1 →



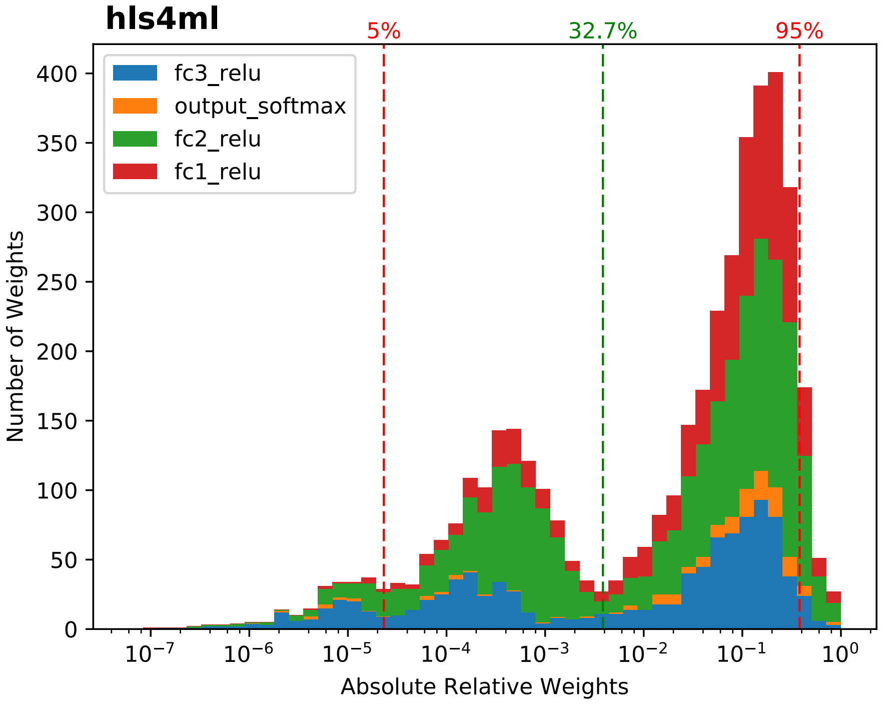
Efficient NN design: compression

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

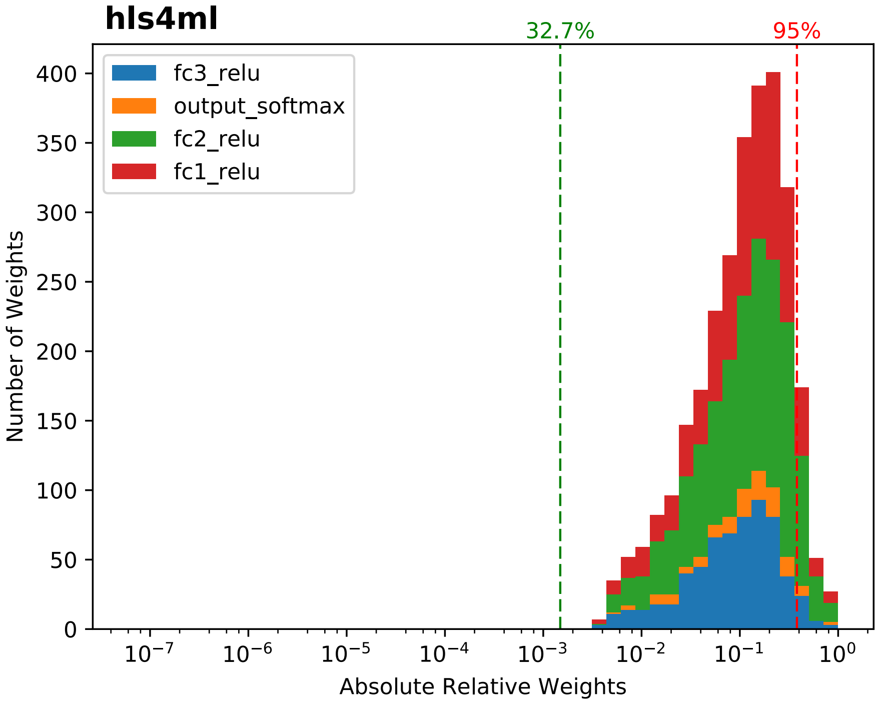
$$L_\lambda(\vec{w}) = L(\vec{w}) + \lambda||\vec{w}_1||$$

  - sort the weights based on the value relative to the max value of the weights in that layer
  - prune weights falling below a certain percentile and retrain

Train with L_1 →

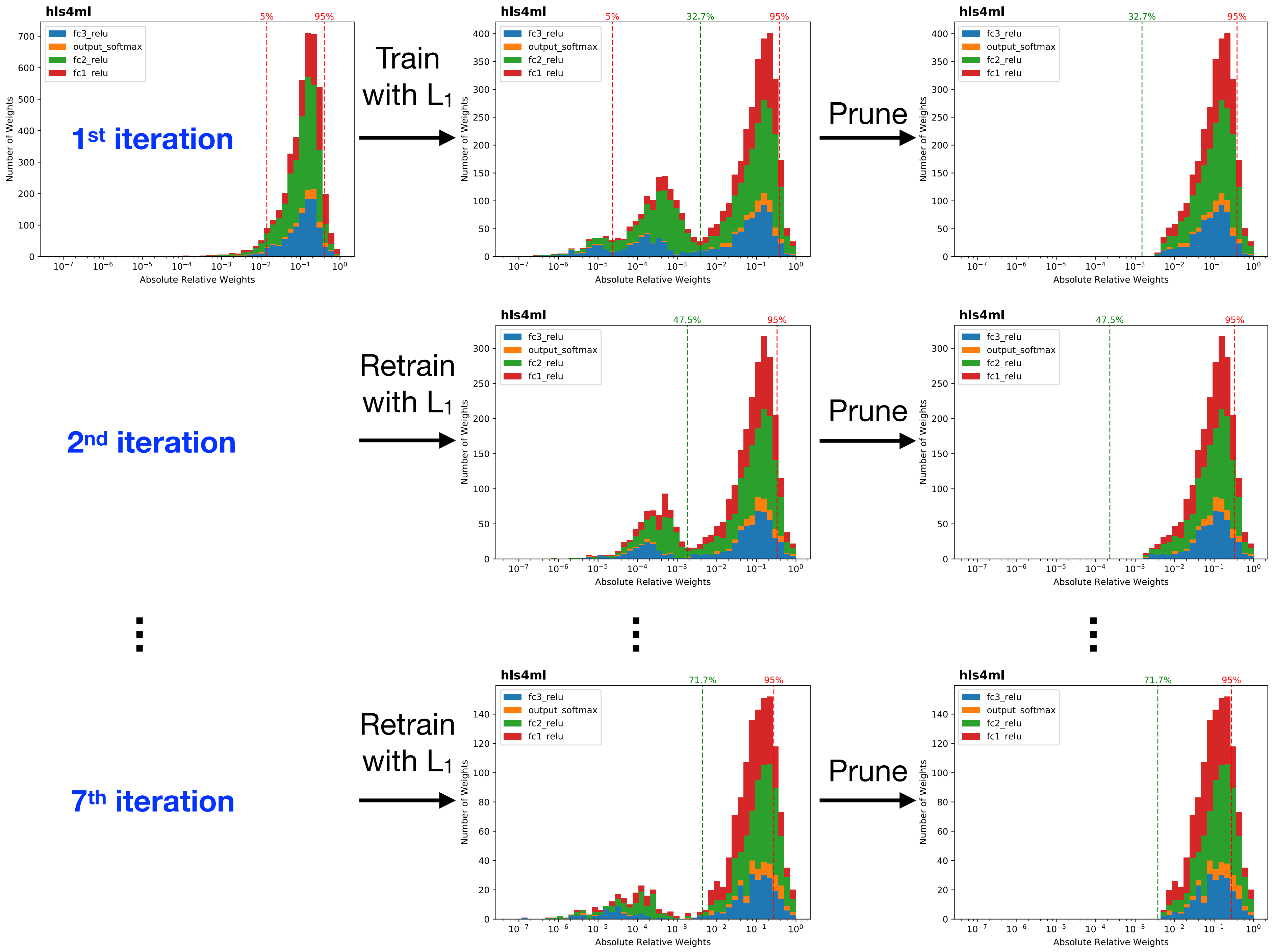


Prune →



Efficient NN design: compression

Prune and repeat the train for 7 iterations



1st iteration

Train with L_1

Prune

2nd iteration

Retrain with L_1

Prune

⋮

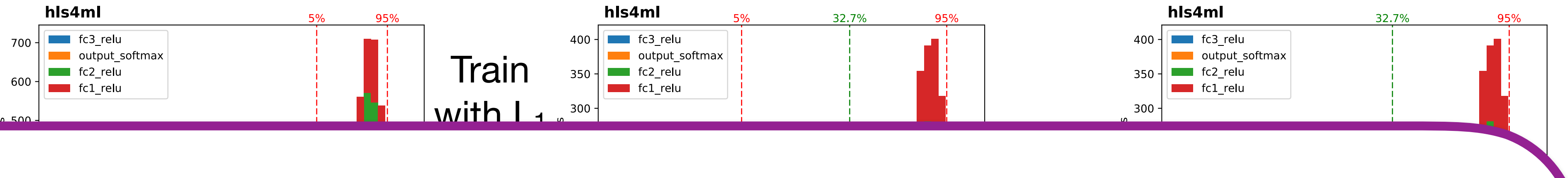
7th iteration

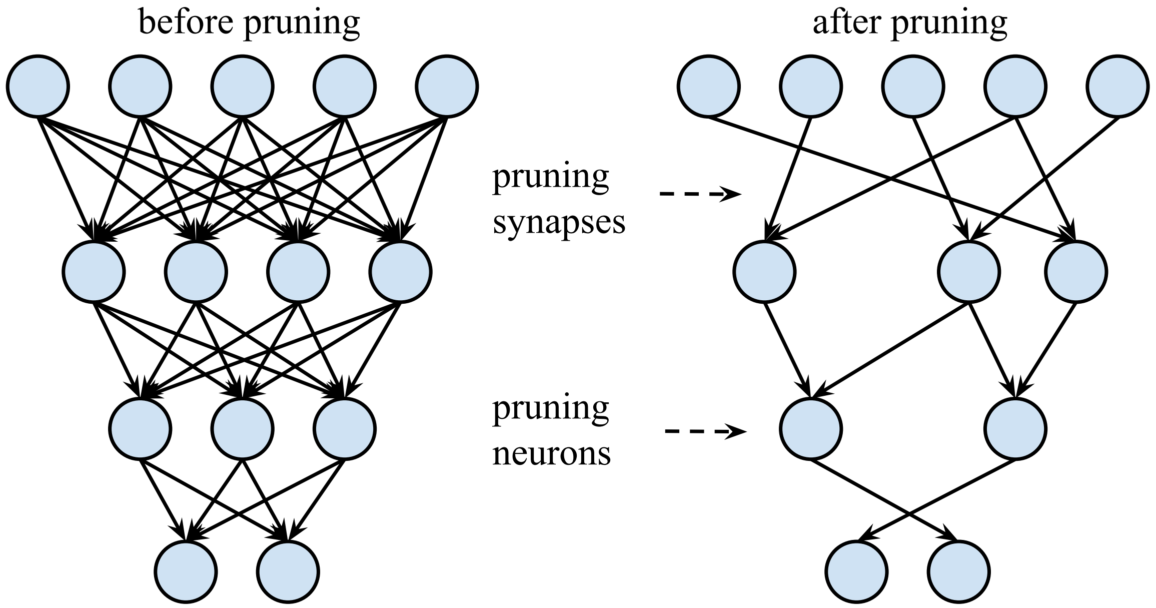
Retrain with L_1

Prune

Efficient NN design: compression

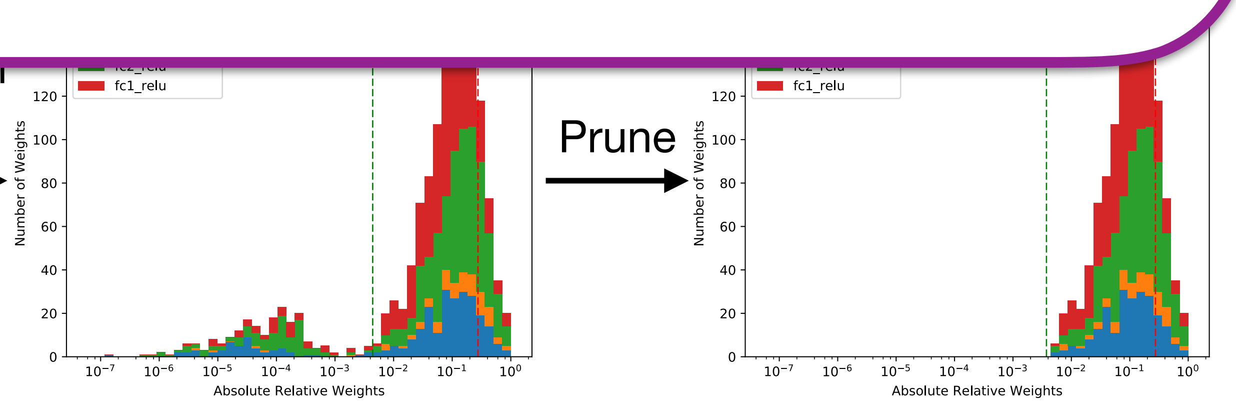
Prune and repeat the train for 7 iterations





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

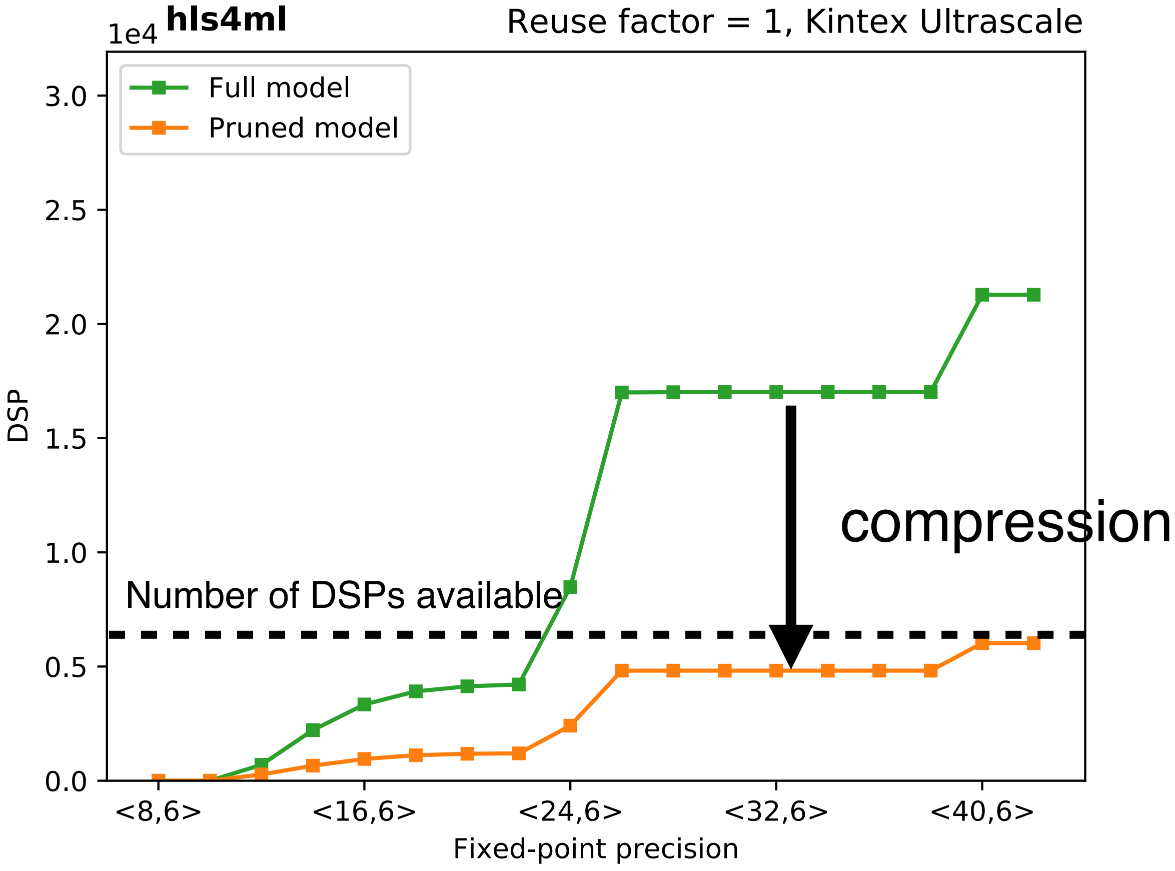
Retrain with L_1



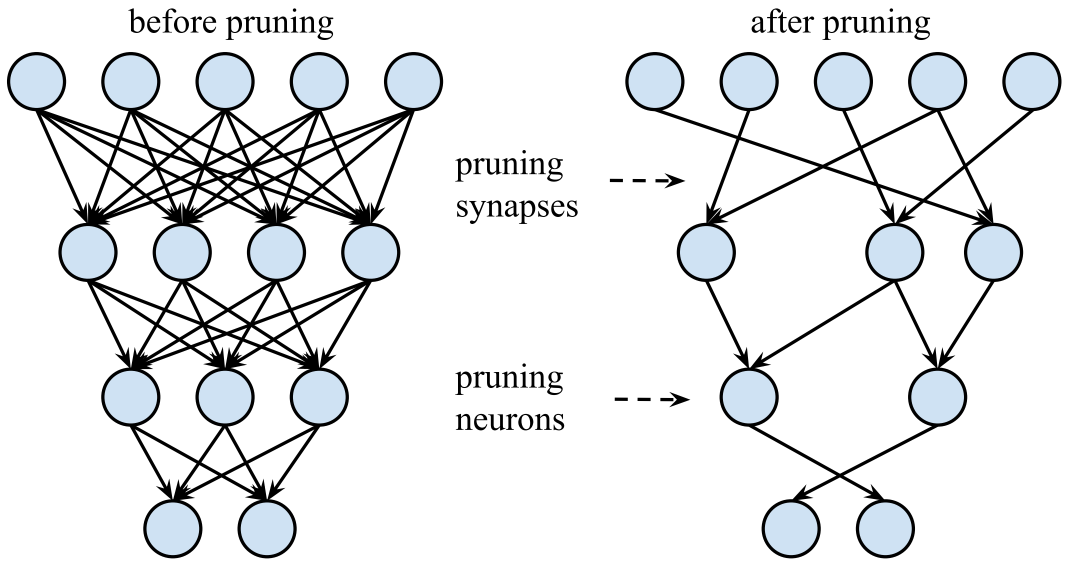
Prune

7th iteration

Efficient NN design: compression



70% compression ~ 70% fewer DSPs



- DSPs (used for multiplication) are often limiting resource
  - DSPs have a max size for input (e.g. 27x18 bits), so number of DSPs per multiplication changes with precision

Using hls4ml

- The model to translate
- Some test vectors for simulation (check precision)
- Output directory / name
- Target FPGA, clock speed

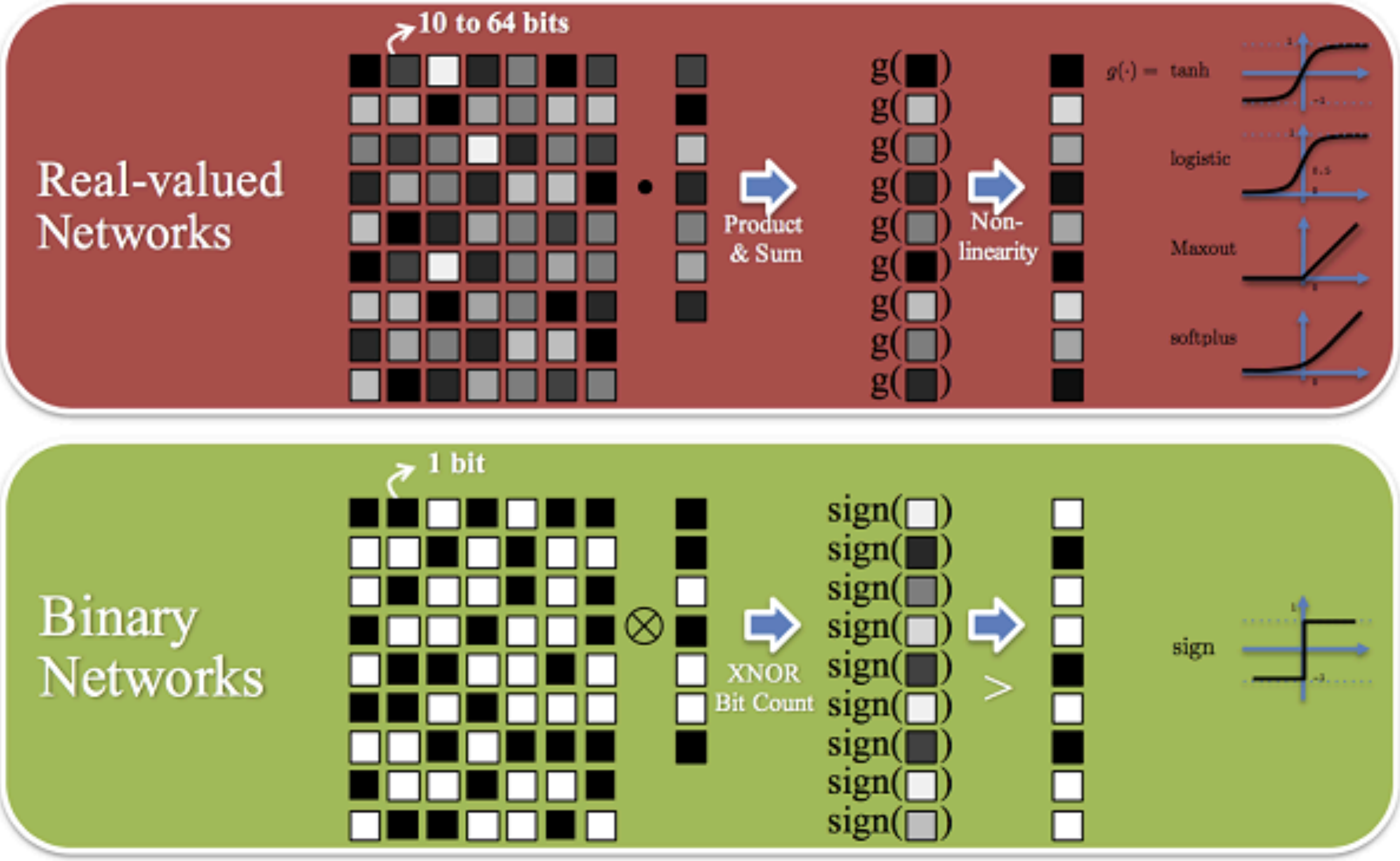
```
KerasJson: keras/KERAS_3layer.json
KerasH5:   keras/KERAS_3layer_weights.h5
#InputData: keras/KERAS_3layer_input_features.dat
#OutputPredictions: keras/KERAS_3layer_predictions.dat
OutputDir: my-hls-test
ProjectName: myproject
XilinxPart: xcku115-flvb2104-2-i
ClockPeriod: 5

IOType: io_parallel # options: io_serial/io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,6>
    ReuseFactor: 1
#   LayerType:
#     Dense:
#       ReuseFactor: 2
#       Strategy: Resource
#       Compression: True
```

- Model data precision and parallelisation
- More fine grained data precision and parallelisation
  - Per-layer, or per-layer type
- Then:

```
hls4ml convert -c my_model.yml
hls4ml build -p my-hls-test
```

Binary / Ternary neural networks

- DSPs (multipliers) usually the limiting resource for our NN inference
- Instead, use 1- or 2-bit weights with limited performance loss
- Can have very efficient computation in the FPGA
- Binarize weights but not gradients during backpropagation
- Use Binary Tanh, Ternary Tanh or ReLu activation
- Batch Normalization



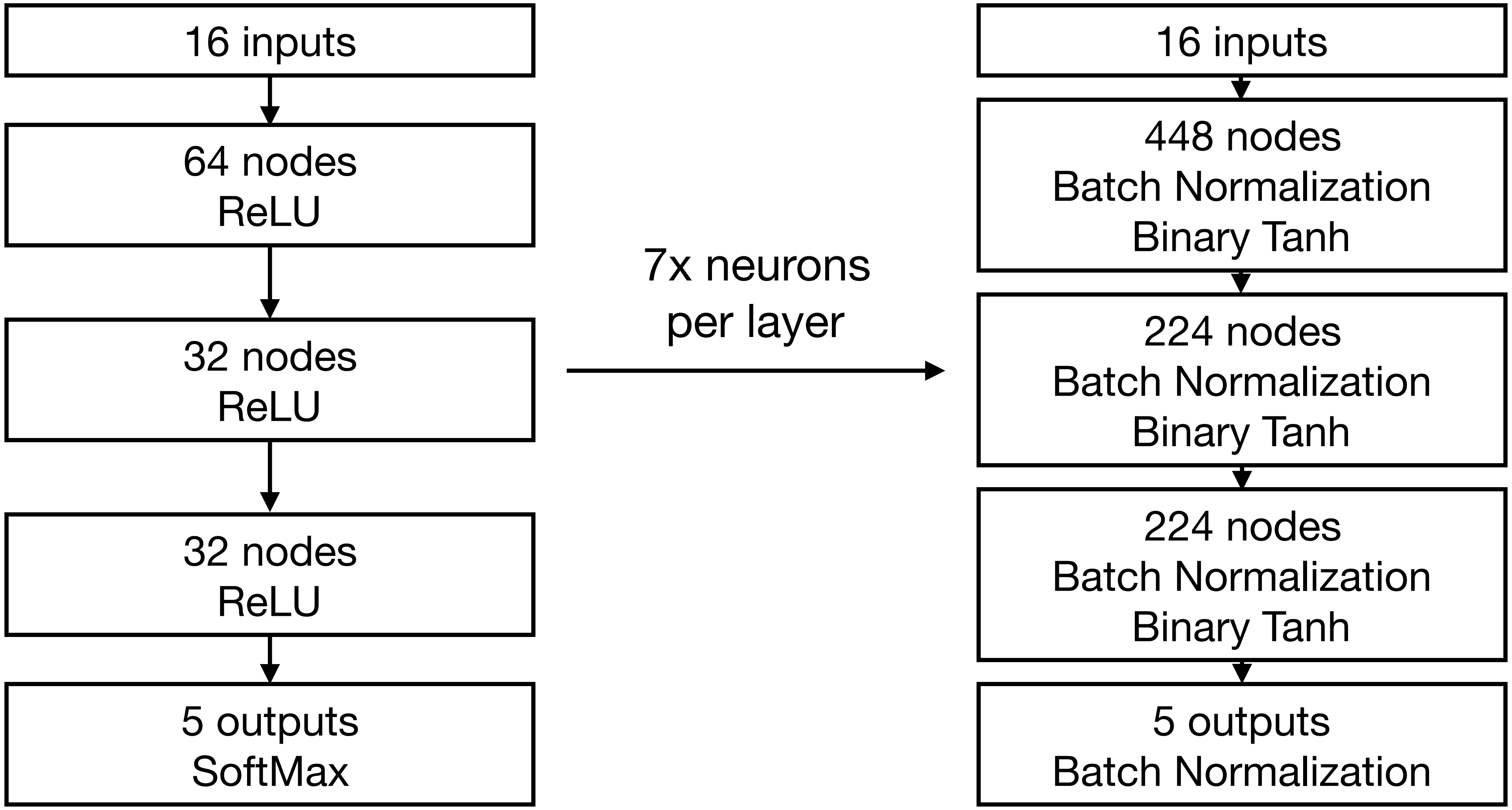
https://software.intel.com/en-us/articles/accelerating-neural-networks-with-binary-arithmetic

- BNN: arxiv.1602.02830
- TNN: arxiv.1605.04711

BNN - Jet Classification

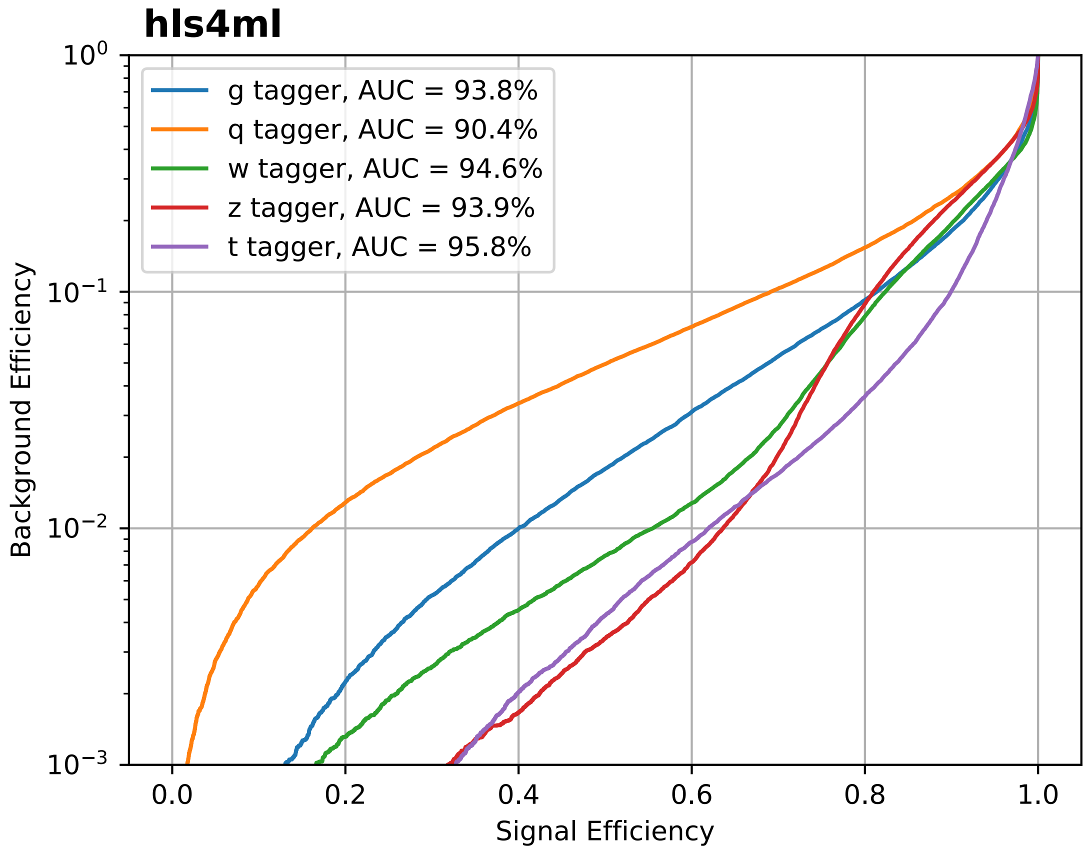
- Design an architecture to perform the same jet classification task but now with binary weights and activations

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

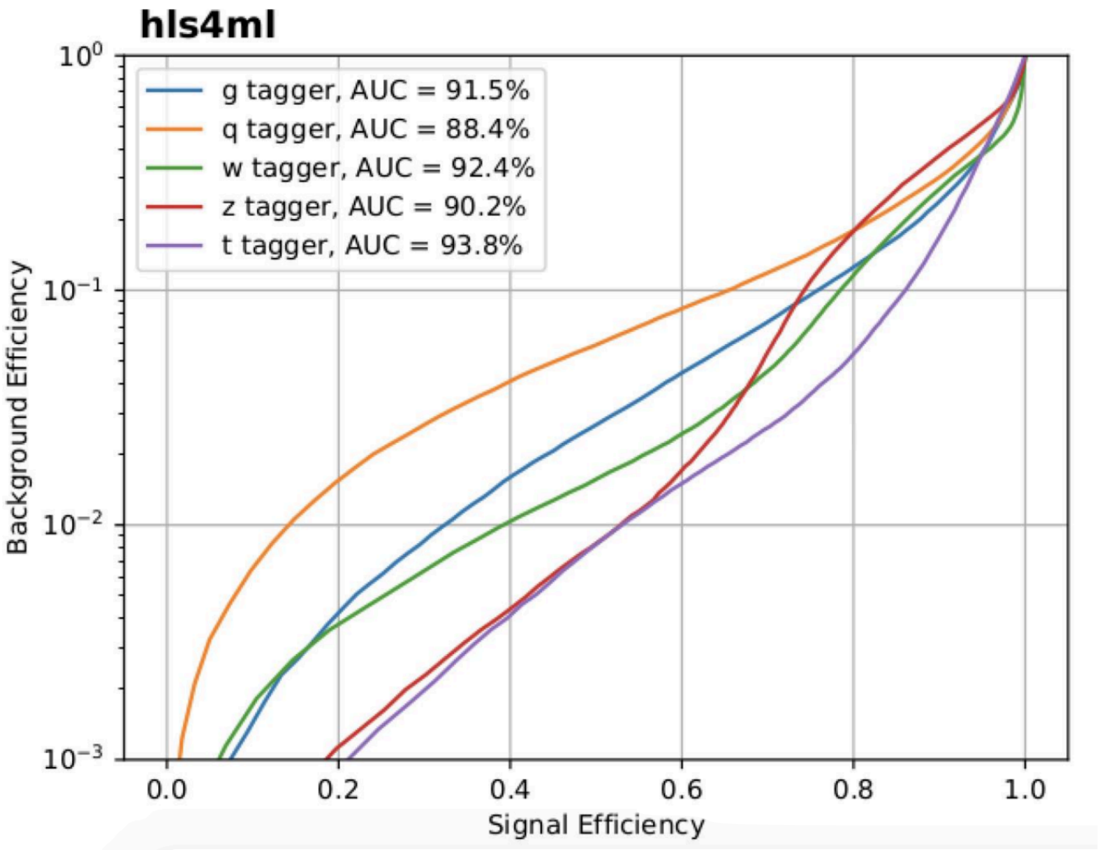


BNN - Jet Classification

- Design an architecture to perform the same jet classification task but now with binary weights and activations
- Performed hyperparameter optimization to find most performant model within some constraints
- Performance is a little bit worse, but not a lot



Original: 16-bit weights
Average accuracy: 0.75



Binarized: 1-bit weights
Average accuracy: 0.72

BNN - Dense Layer

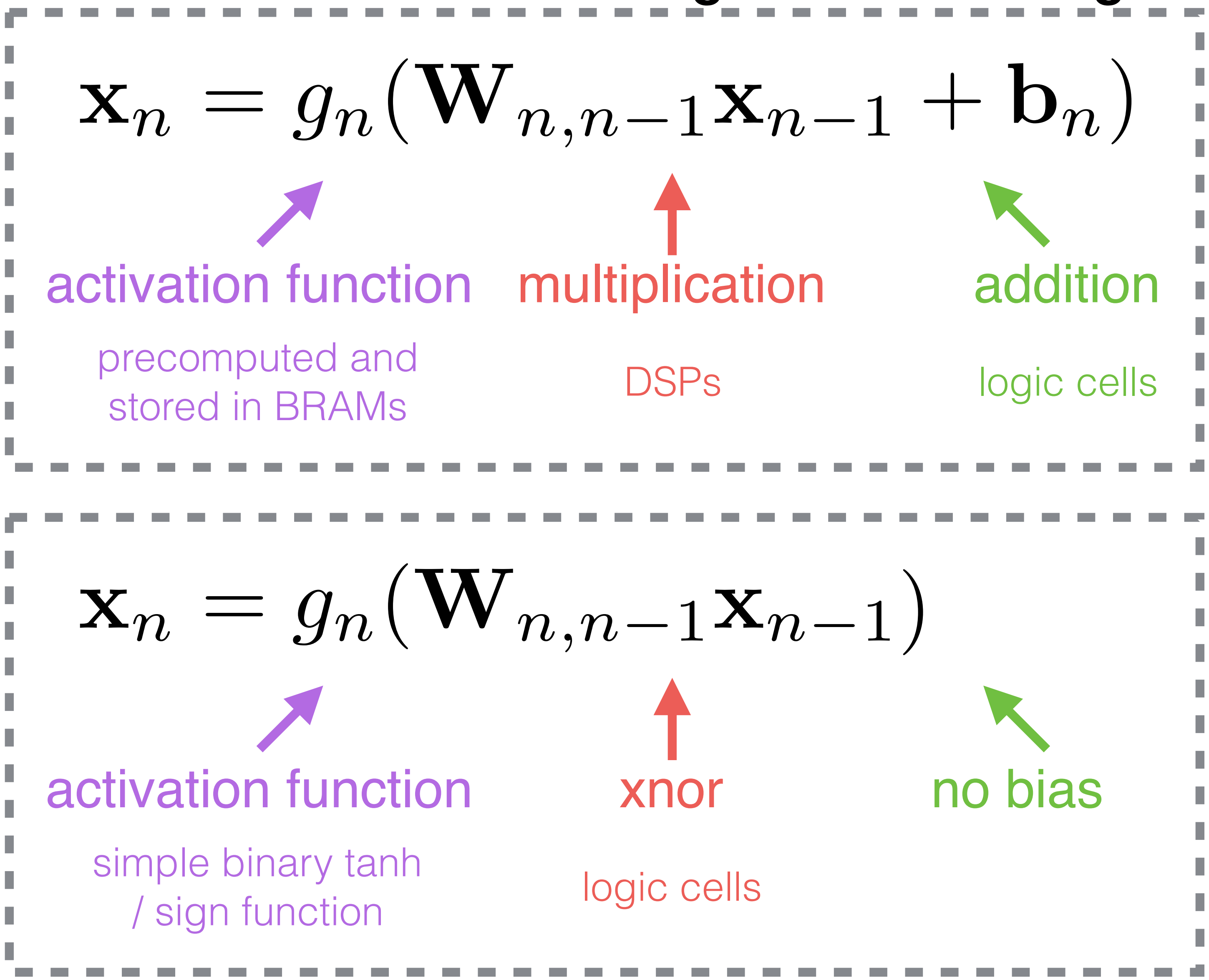
- DSPs often limiting FPGA resource
- Encode ‘-1’ as ‘0’
- Multiplication become XNOR, sum becomes bitcount

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

BNN - Jet Classification

- Results targeting Xilinx VU9P FPGA at 200 MHz

Model	Accuracy	Latency (µs)	DSP (%)	LUT (%)	FF (%)
Original model	0.75	0.06	60	7	1
Original model (70% compressed)	0.75	0.09	15	1.7	0.7
Optimized BNN (16x448x224x224x5)	0.72	0.21	-	15	7
BNN w/ReLu (16x128x64x64x5)	0.70	0.140	4	6	1
Optimized TNN (16x128x64x64x64x5)	0.72	0.11	-	6	1
TNN w/ReLu (16x64x32x32x5)	0.68	0.06	2	2	0.2

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
- I've shown the hls4ml package for running ML inference in sub-microsecond latency on FPGAs
    - For more: fastmachinelearning.org/hls4ml