Tree Tensor Network inference on FPGA

1st FPGA Developers Forum

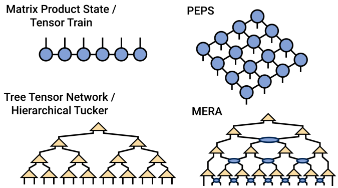
University of Padua

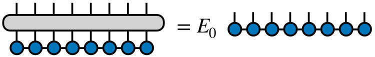
L.Borella, A.Coppi, J.Pazzini, A.Stanco, A.Triossi, M.Zanetti

12 June 2024

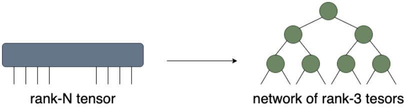
Tree Tensor Networks

Tensor Networks have first been developed to investigate quantum many-body systems on classical computers by efficiently representing quantum wavefunction $|\psi\rangle$ and Hamiltonians H [1]. Typically used for energy minimization or time evolution simulations, they can also be exploited in **Machine Learning** contexts.





They are the result of the factorization of very large tensors into networks of smaller tensors. Several types of decompositions are possible (MPS, MPO etc.): their approximation can be tuned by modifying *bond dimensions*[2].



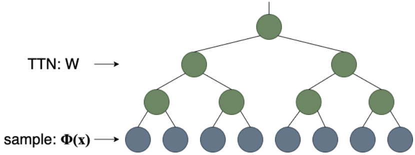
Tree Tensor Networks (TTNs) are a specific type of tensor decomposition that results in a hierarchical tree-like architecture.

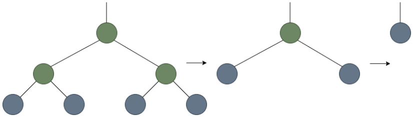
Tree Tensor Networks: Machine Learning

TTNs can be trained as **ML Classifiers** following the decision function:

$$f(x) = W \cdot \Phi(x).$$

The information of an N-features dataset can be encoded in the network W by contracting it with each data sample $\Phi(x)$ and iteratively updating all the inner tensors [3].





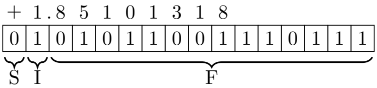
Inference can be performed by contracting the whole TTN with each sample: the resulting vector stores the classification probabilities for each label of the dataset.

In this project:

- **Task**: binary classification, scalar result.
- **Datasets**: Iris[4], Titanic[5] and LHCb[1].
- **Architectures**: 4, 8, 16 input features.
- **Parallelism**: full parallel and partial parallel implementations.
- **Training** in software, **inference** in hardware (FPGA KCU1500).

Input: Numeric representation

All the values in firmware are represented as 16 bit fixed-point numbers, devoting 1 bit for the Sign, 1 bit for the Integer part and 14 bits for the Fractional part, corresponding to [-2,+2] as total representation range, with precision $6.103 \cdot 10^{-5}$.

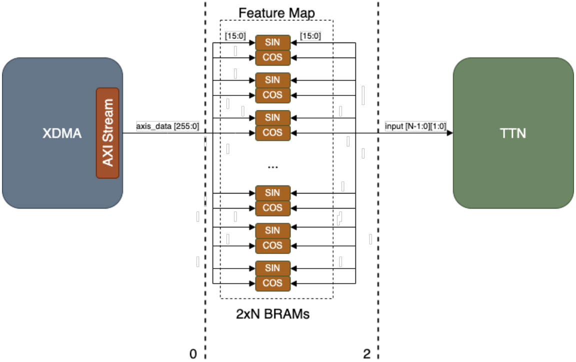


Following the quantum approach, to perform classification each dataset feature x_i is encoded by a local feature map $\Phi(x_i) = [\cos \frac{\pi x_i}{2}, \sin \frac{\pi x_i}{2}]$. In this way, each sample represents a separable state $|\psi\rangle$ resulting from the tensor products of 2-dim vectors. The spinorial mapping encloses input values in the range [-1,+1] and guarantees their representability.

Input data are sent to the FPGA via **AXI-Stream** protocol and the feature map is implemented in hardware. Since the original features live in $\mathbb{R}$, they first need to be rescaled in $[0, \frac{\pi}{2}]$ in software.

Input: Feature map

$\sin(x)$ and $\cos(x)$ functions are implemented in hardware with Vivado IP **Block Memory Generator**. Each BRAM is configured during implementation (`sin.coe, cos.coe`) and fixed in firmware.

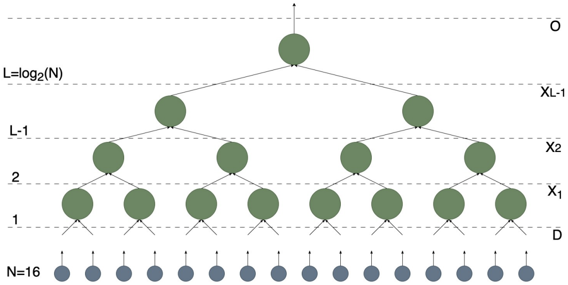


BRAM: lat=2 clk, width=16, depth=65536, corresponding to 131 kB/BRAM. The number of necessary BRAMs is always twice the number of features N.

Weights

TTN architecture fixed in firmware by setting: number of features N, input dimension D, bond dimensions X_i, output dimension O.

Weights loaded from host PC: try different networks and perform quantization tests.

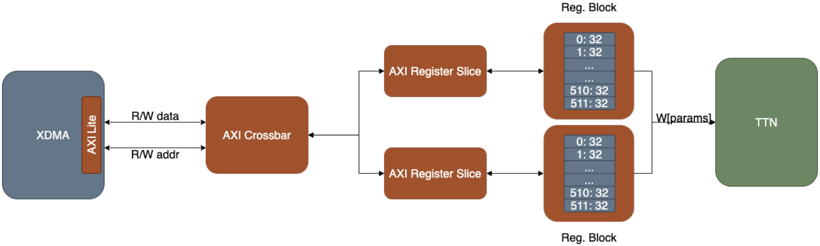


N	D	X_1	X_2	X_3	O	Params	Mem.	Reg.	Blocks
4	2	4	0	0	1	48	96 B	24	1
4	2	8	0	0	1	128	256 B	64	1
8	2	4	4	0	1	208	416 B	104	1
8	2	4	8	0	1	384	768 B	192	1
16	2	4	8	8	1	1728	3 kB	864	2
16	2	4	8	16	1	2944	5 kB	1472	3

The weights are loaded on FPGA via **AXI Lite** protocol: read-write from host PC, read-only from TTN. Blocks of 512x32bit registers generated as custom Vivado IP **AXI Peripheral**.

Weights

Once the architecture is fixed and the FPGA is programmed, the weights registers can be written by host PC and read back for verification. During inference, these values remain static and are only read by the TTN component.

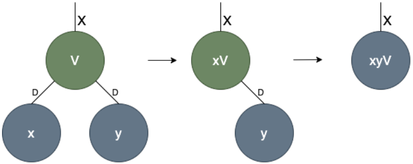


- **Crossbar**: receives AXI Lite information and switches between different register blocks, according to `base_address` value.
- **Register Slice**: slices vectors ([65:0]→2x[31:0]) and registers the values.
- **Reg. Block**: 512x32b registers, 1024x16b weights. Forwards W vector to TTN.
- **Timing**: timing constraints must be relaxed. Area is too big but the values are static.

Node Contraction

Single node contraction is the basic building block operation, which in formulae (Einstein notation) is $z^{\mu} = V^{\mu}_{\nu,\rho} x^{\nu} y^{\rho}$, considering 2 vectors x and y of dimension $[D]$ and one tensor V of dimension $[D, D, X]$.

Vivado IP **DSP Macro** for multiplications, with variable number of registers Δt_{DSP} and intrinsic latency. Two different degrees of parallelization: **Full Parallel** and **Partial Parallel**.



$$
\begin{aligned}
z_0 &= V_{000}\, x^0 y^0 + V_{001}\, x^0 y^1 + V_{010}\, x^1 y^0 + V_{011}\, x^1 y^1 \\
z_1 &= V_{100}\, x^0 y^0 + V_{101}\, x^0 y^1 + V_{110}\, x^1 y^0 + V_{111}\, x^1 y^1 \\
z_2 &= V_{200}\, x^0 y^0 + V_{201}\, x^0 y^1 + V_{210}\, x^1 y^0 + V_{211}\, x^1 y^1 \\
z_3 &= V_{300}\, x^0 y^0 + V_{301}\, x^0 y^1 + V_{310}\, x^1 y^0 + V_{311}\, x^1 y^1
\end{aligned}
$$

Sum
Mult2 Mult1

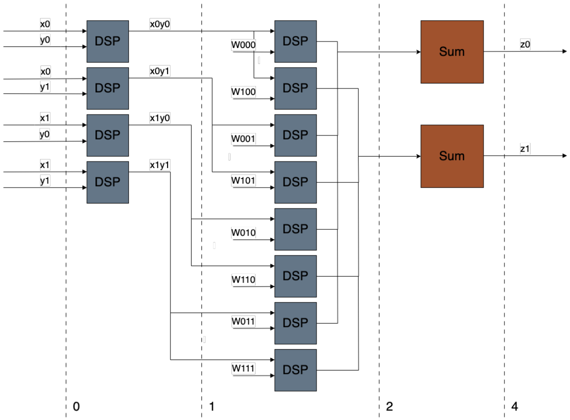
3-factor multiplication, in hardware we split it into the following stages:

Mult1: x and y cartesian product.

Mult2: multiply results of mult1 by corresponding weights V.

Sum: X parallel sums to compute final vector components.

Node Contraction, Full Parallel



Example for $D = 2$ and $X = 2$.

Full Parallel computation: maximize resources and minimize latency.

1 DSP for each multiplication: D^2 at mult1 and XD^2 at mult2.

Adder tree for sum stage.

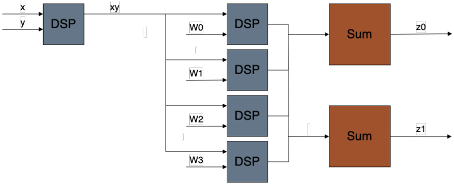
Tree DSPs:

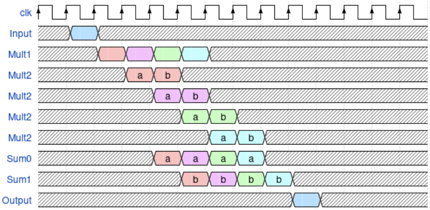
$$\sum_{i=1}^{L} \frac{N}{2^i} X_{i-1}^2 (X_i + 1)$$

Tree latency:

$$\Delta t_{DSP} \sum_{i=1}^{L} 2 + \log_2(X_{i-1}^2)$$

Node Contraction, Partial Parallel





Example for $D = 2$ and $X = 2$.

Partial Parallel computation: reduce resources and increase latency.

1 DSP at mult1, D^2 DSP at mult2 and X serial sums.

Pipelined computation.

Tree DSPs:

$$\sum_{i=1}^{L} \frac{N}{2^i}(X_{i-1}^2 + 1)$$

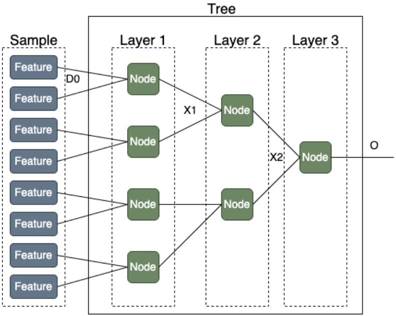
Tree latency:

$$\Delta t_{DSP} \sum_{i=1}^{L} X_{i-1}^2 + X_i + 1$$

Node, Layer, Tree

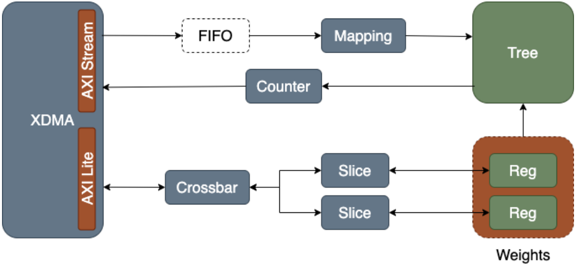
Implementation:

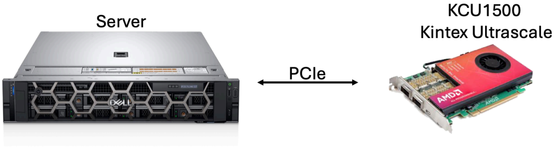
- In library file `tensors.vhd` we fix the parameters N, D_0, X_0, O.
- A VHDL function derives the architecture of the TTN with options:
  `fixed`: $X_i = X_0$
  `minimal`: $X_i = min(X_0, D_0^{2^i})$
  `maximal`: $X_i = D^{2^i}$.
- `layer.vhd` file generates $\frac{N}{2^i}$ nodes for layer i.
- `tree.vhd` file generates $L = \log_2(N)$ layers.



N	D_0	X_0	Impl.	DSP	Latency [clk]
8	2	4	FP	272	64
8	2	4	PP	105	192
16	2	8	FP	2016	104
16	2	8	PP	501	692

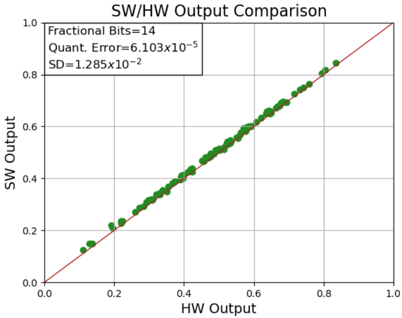
Firmware



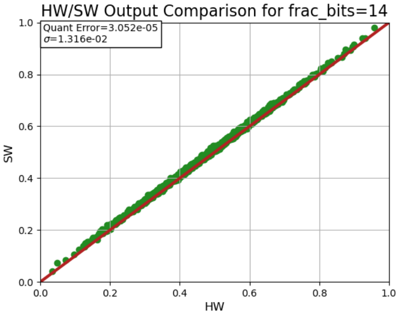


- FIFO only for PP implementation.
- Project developed on KCU 1500 Kintex Ultrascale.
- Board plugged in host PC with PCIe communication.
- Configurable registers for weights: **AXI Lite**.
- TTN input and output values: **AXI Stream**.
- **AXI Stream** clck: 250 MHz. OOC-TTN can reach 500 MHz.

Results: Output Comparison

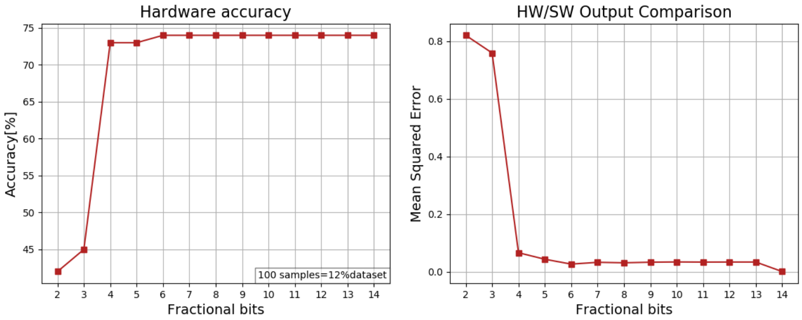


TTN architecture N=8, X_i=[2,4,8,1], 100 samples.

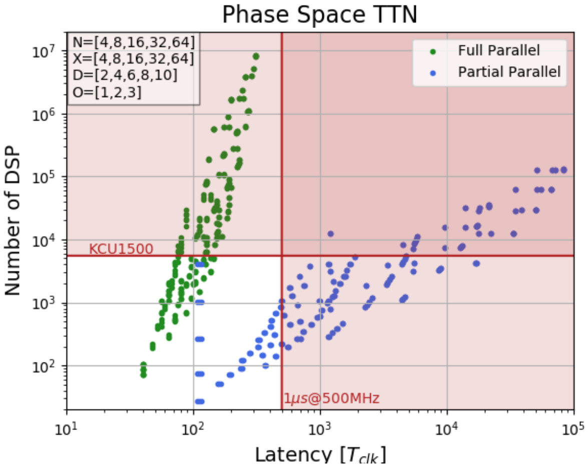


TTN architecture N=16, X_i=[2,4,8,8,1],500 samples.

Results: Quantization



Results: parameter space.



Thank you for your attention.

References I

[1] A. Giannelle D. Zuliani T. Felser D.Lucchesi S.Montangero M. Trenti, L. Sestini.
Quantum-inspired machine learning on high-energy physics data.
2021.

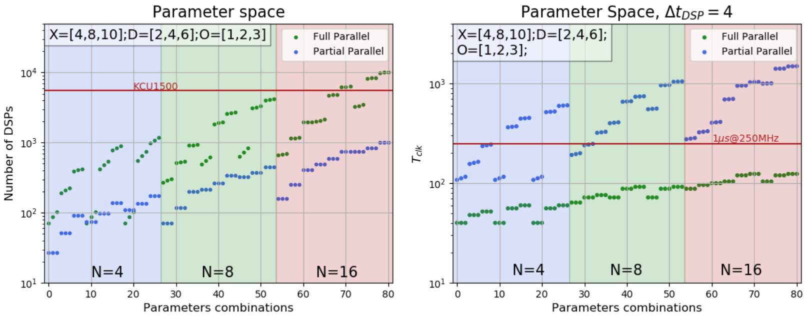
[2] Tensor network.
https://tensornetwork.org.

[3] E. Miles Stoudenmire and David J. Schwab.
Supervised learning with quantum-inspired tensor networks.
2017.

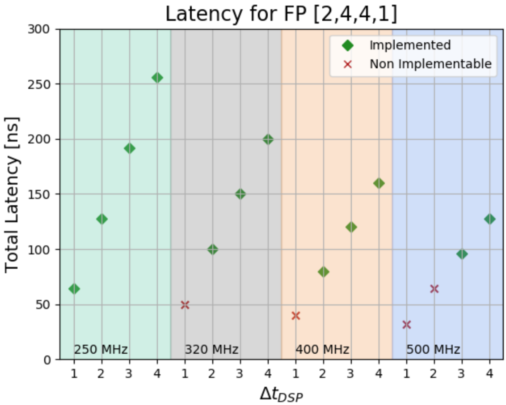
[4] Iris dataset.
https://www.kaggle.com/datasets/uciml/iris.

[5] Titanic dataset.
https://www.kaggle.com/c/titanic/data.

DSP and Latency

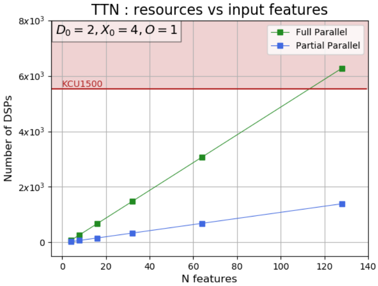


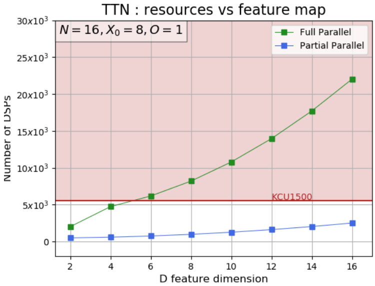
Frequency



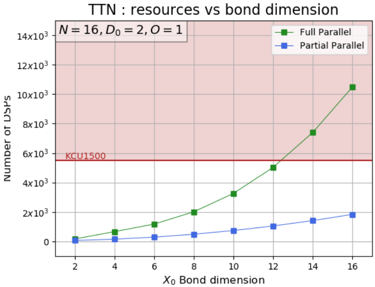
Latency for FP [2,4,4,1]
Implemented
Non Implementable
Total Latency [ns]
300
250
200
150
100
50
0
250 MHz
320 MHz
400 MHz
500 MHz
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Δt_{DSP}

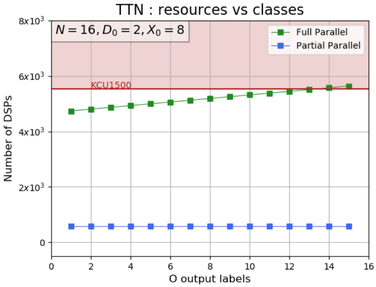
DSP


TTN : resources vs input features
$D_0 = 2, X_0 = 4, O = 1$
Full Parallel
Partial Parallel
KCU1500
Number of DSPs
8x10^3
6x10^3
4x10^3
2x10^3
0
0
20
40
60
80
100
120
140
N features

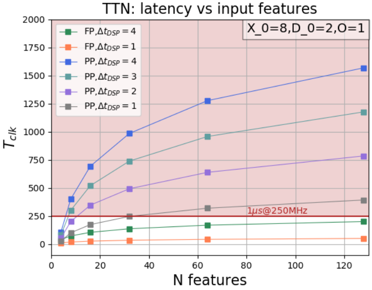

TTN : resources vs feature map
$N = 16, X_0 = 8, O = 1$
Full Parallel
Partial Parallel
KCU1500
30x10^3
25x10^3
20x10^3
15x10^3
10x10^3
5x10^3
0
2
4
6
8
10
12
14
16
D feature dimension

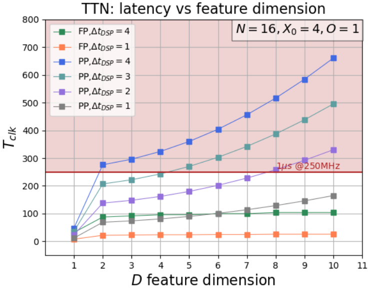
DSP


TTN : resources vs bond dimension
$N = 16, D_0 = 2, O = 1$
Full Parallel
Partial Parallel
KCU1500
Number of DSPs
X_0 Bond dimension
0
2x10^3
4x10^3
6x10^3
8x10^3
10x10^3
12x10^3
14x10^3
2
4
6
8
10
12
14
16

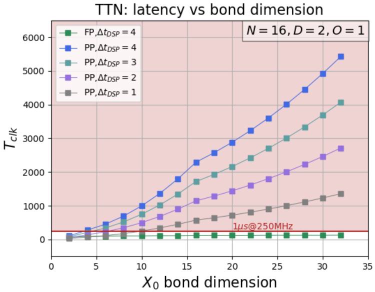

TTN : resources vs classes
$N = 16, D_0 = 2, X_0 = 8$
Full Parallel
Partial Parallel
KCU1500
Number of DSPs
O output labels
0
2x10^3
4x10^3
6x10^3
8x10^3
0
2
4
6
8
10
12
14
16

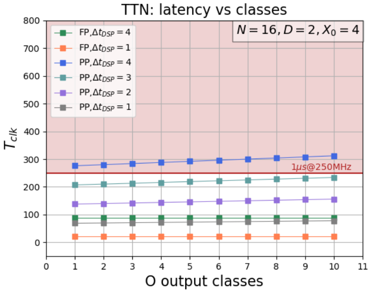
Latency


TTN: latency vs input features
X_0=8,D_0=2,O=1
FP,$\Delta t_{DSP} = 4$
FP,$\Delta t_{DSP} = 1$
PP,$\Delta t_{DSP} = 4$
PP,$\Delta t_{DSP} = 3$
PP,$\Delta t_{DSP} = 2$
PP,$\Delta t_{DSP} = 1$
1μs@250MHz
T_{clk}
N features

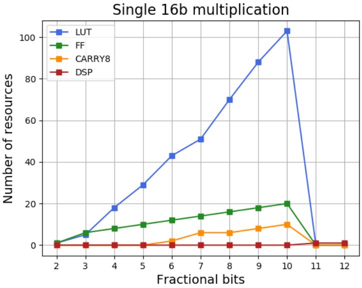

TTN: latency vs feature dimension
$N = 16, X_0 = 4, O = 1$
FP,$\Delta t_{DSP} = 4$
FP,$\Delta t_{DSP} = 1$
PP,$\Delta t_{DSP} = 4$
PP,$\Delta t_{DSP} = 3$
PP,$\Delta t_{DSP} = 2$
PP,$\Delta t_{DSP} = 1$
1μs @250MHz
T_{clk}
D feature dimension

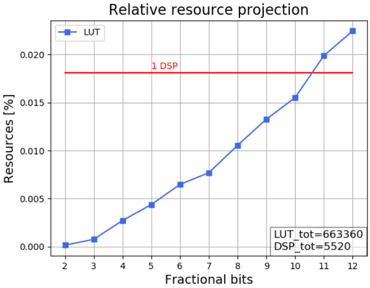
Latency


TTN: latency vs bond dimension
$N=16, D=2, O=1$
FP,$\Delta t_{DSP}=4$
PP,$\Delta t_{DSP}=4$
PP,$\Delta t_{DSP}=3$
PP,$\Delta t_{DSP}=2$
PP,$\Delta t_{DSP}=1$
$1\mu s$@250MHz
T_{clk}
X_0 bond dimension

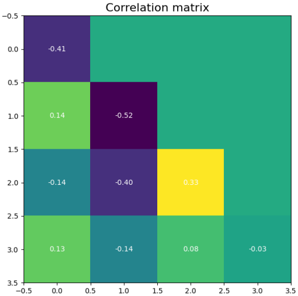

TTN: latency vs classes
$N=16, D=2, X_0=4$
FP,$\Delta t_{DSP}=4$
FP,$\Delta t_{DSP}=1$
PP,$\Delta t_{DSP}=4$
PP,$\Delta t_{DSP}=3$
PP,$\Delta t_{DSP}=2$
PP,$\Delta t_{DSP}=1$
$1\mu s$@250MHz
T_{clk}
O output classes

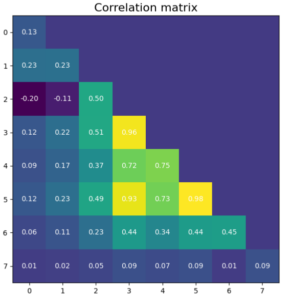
Multiplication Usage



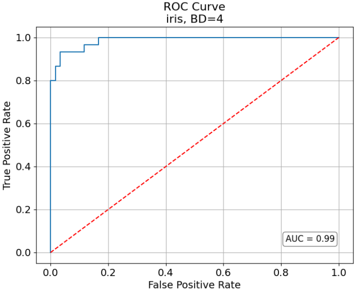


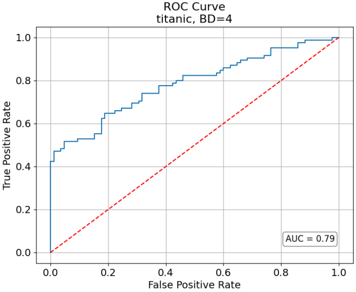
TTN: correlation matrices



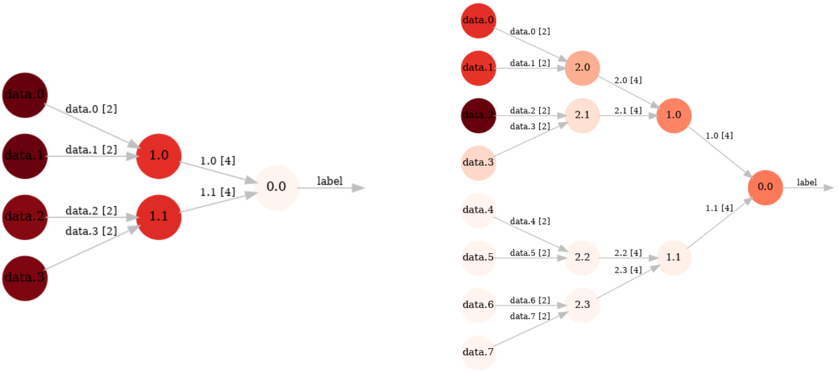


TTN: ROC curves





TTN: entropy


data.0
data.1
data.2
data.3
data.0 [2]
data.1 [2]
data.2 [2]
data.3 [2]
1.0
1.1
1.0 [4]
1.1 [4]
0.0
label
data.4
data.5
data.6
data.7
data.4 [2]
data.5 [2]
data.6 [2]
data.7 [2]
2.0
2.1
2.2
2.3
2.0 [4]
2.1 [4]
2.2 [4]
2.3 [4]