Machine learning for online applications

Some ideas and an example

A.Boudon, Q.David, G.Galbit, G.Joubert, S.Viret

*IP2I Lyon **Virgo group** and **Edaq service***

Introduction
ML on FPGA
Example

→ **Goal of this presentation:**

→ Introduction: why is machine learning (ML) an interesting field for future interferometers online architectures?

→ Description of an hardware-based approach for online applications requiring low-latency and high data troughput.

→ Illustration with a well-known GW detection neural network.

→ ML and gravitational waves:

→ First paper in 2016, since then the field is following steadily the latest developments. Not a huge community but quite proactive (*more info:* https://iphysresearch.github.io/Survey4GWML/).

2016: CNN
(*eg:* https://arxiv.org/abs/1701.00008v3)

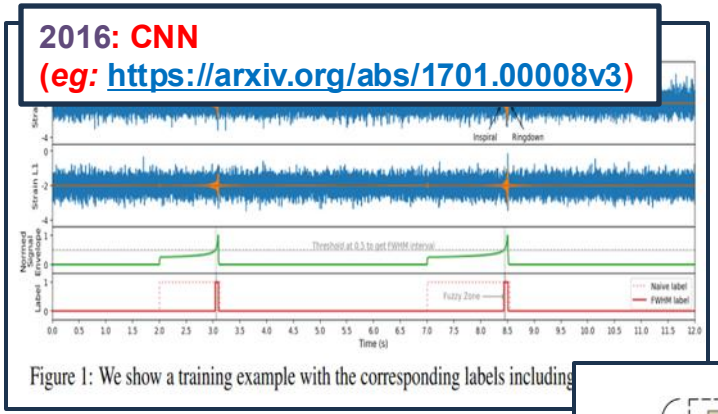
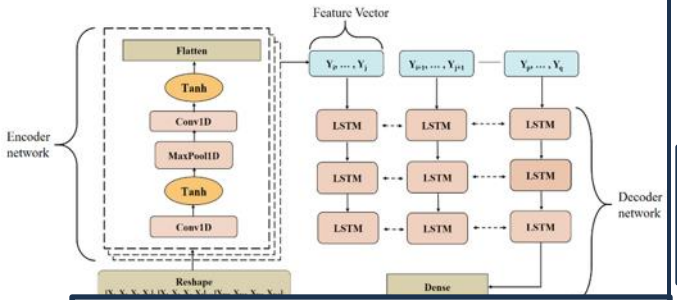
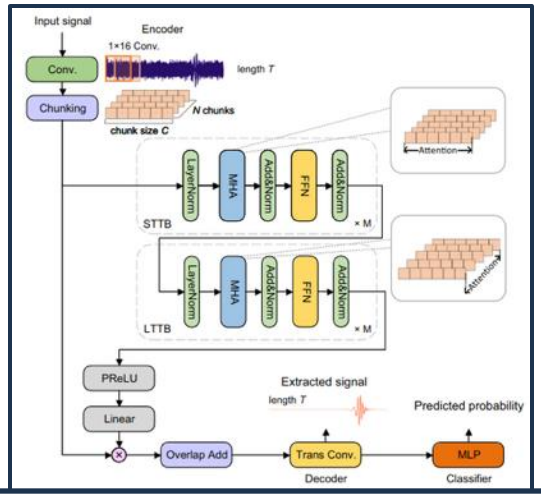


Figure 1: We show a training example with the corresponding labels including



2019: Smart denoising with LSTM
(https://arxiv.org/abs/1909.06367v2)



2024: Denoising and categorization with Transformers
(https://arxiv.org/abs/2406.06324)

2016 **2019** **2024**

→ **Using ML online:**

→ More recently, some concrete effort were made to tackle the online side (*eg.* ***Seismic noise reduction with reinforcement learning in LIGO***, *online detection pipeline (**https://arxiv.org/pdf/2403.18661v1**)).*

→ Online controls and online analysis are very different fields, still both can be helped by ML. Other sectors can benefit from this field, the potential is clear, even for current detectors…

→ There is no magic tough. The apparent simplicity of ML tools shouldn't hide the fact that it's a very complex field.

→ Even more accurate if you want to go online, as there are some specific constraints to keep in mind: **robustness, low latency, simplicity**, etc **You cannot consider ML as a magic black box.**

→ How to go online?

→ Going online means realtime, but **what do we call realtime**?

→ If realtime is millisecond scale and your data troughput is small (*analysis*), GPU are appropriate. But if you're interested in lower latencies and/or much larger I/O rates (*controls, denoising*), **you have to think differently.**

→ It's particularly true for applications related to DAQ. In which case using lower level components, like **FPGAs** (*reprogrammable ICs*) could become more interesting than GPUs

→ Can also work with a **much lower power** budget than a GPU. **If you want a fanless DAQ, it's an important point.**

→ In the following we will focus on FPGAs (*long standing expertise at IP2I*), but few points we will mention are valid also for GPU-based online approaches.

→ **ML and FPGAs: a long story:**

→ Back in the 80s, first neural network inference implementations were done on FPGAs

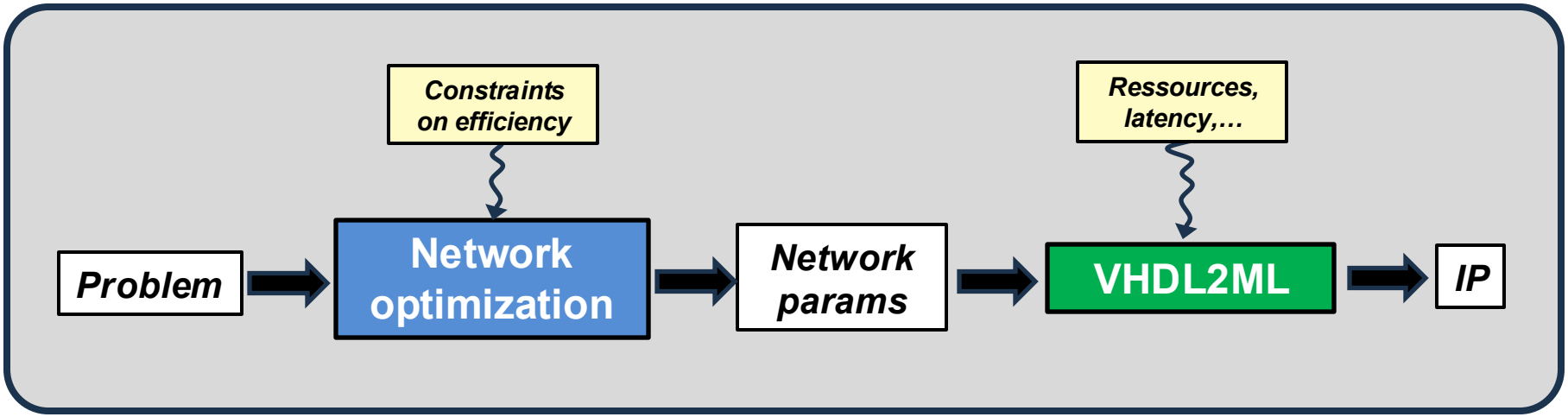
→ GPUs washed away all that, but FPGAs recently reclaimed some popularity, in particular in HEP (*eg* ***HLS4ML***). Already used by some LHC experiments trigger scheme (*see for example* ***https://arxiv.org/html/2503.09428v1***)

→ But it comes to a price:

- Working on FPGA is rather more complex
- You can implement **only simple architectures** unless you have big FPGAs. Certainly not a big issue for LHC L1 trigger applications, but possibly a limitation for others (*in particular if you're looking for low power*)

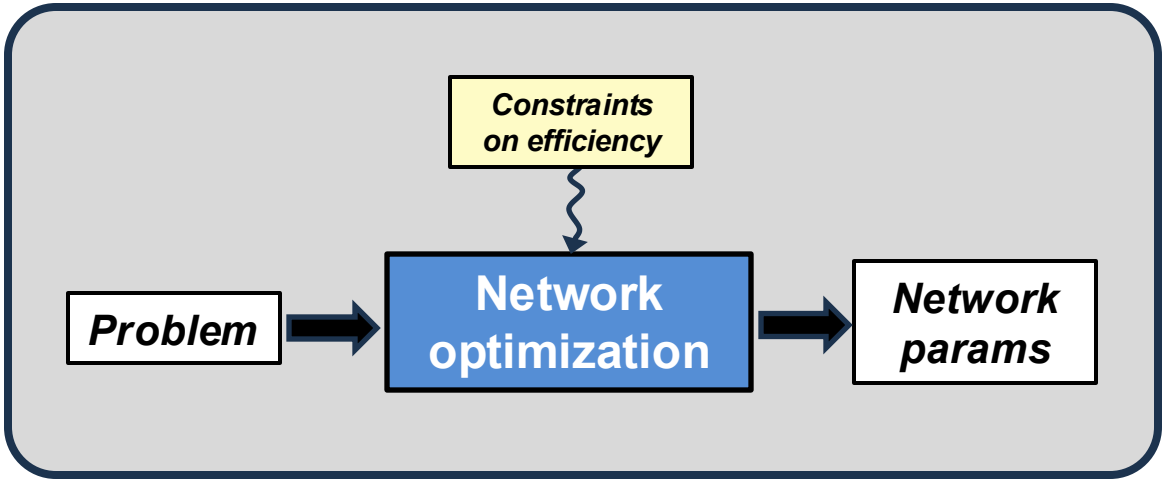
→ **Our strategy:**

→ Try to develop a simple approach to implement complex neural network architecture on small FPGAs. Keep **low latency** and **good efficiency**, but also aim for **less power consumption**.



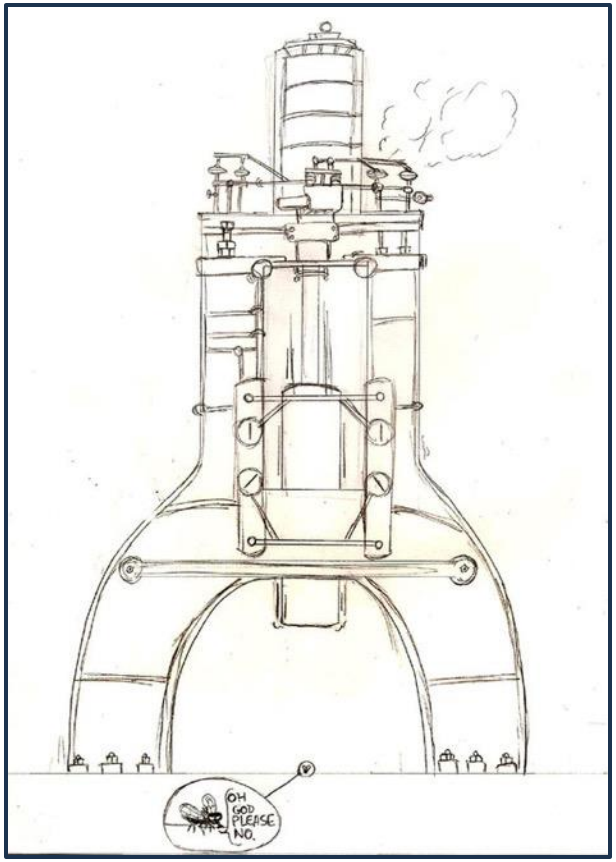
→ The first step is actually also relevant if you use a GPU. **Compact networks are much easier to understand and control.**

→ This work is done within the IN2P3 THINK2 R&T program (*see* ***https://think.in2p3.fr/*** *and* ***https://gitlab.in2p3.fr/think2*** *for more info*)


Constraints on efficiency
Problem
Network optimization
Network params

→ Network optimization

→ **A common mistake in machine learning is to start from a network far more powerful than what you need.**



→ Choosing the good model is a crucial step, often overlooked

→ The key there is to properly define what you have in hand, and what you are looking for.

→ From there you can define the most adapted architecture. There is no magic for this, you need to learn a bit, but there are some nice ressources (*eg:* ***https://d2l.ai/***)

→ **Network optimization**

→ OK you have the right network, but now how do you dimension it?

→ **No magic here too, a**nd actually even less litterature. Heuristic approach is most common strategy (*the hammer, again…*), you test 10's of networks, and take the best one.

→ **How you define best is also a good question**. Are you just interested in efficiency, or also on fake rate minimisation,… For FPGA (*for any online application btw*) number of parameters, operations per inference, etc… are also important. How do you account for them?

→ For simple architecture, a bit of thinking can help a lot

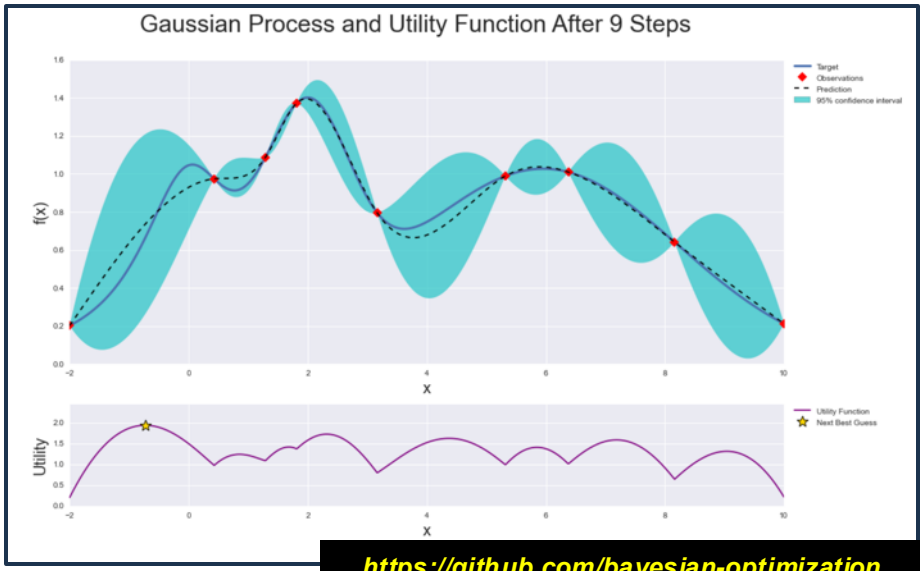
→ But when you start to play with complex neurons and/or large structure, this approach is quickly becoming cumbersome.

→ Simplifying network optimisation with bayesian approach

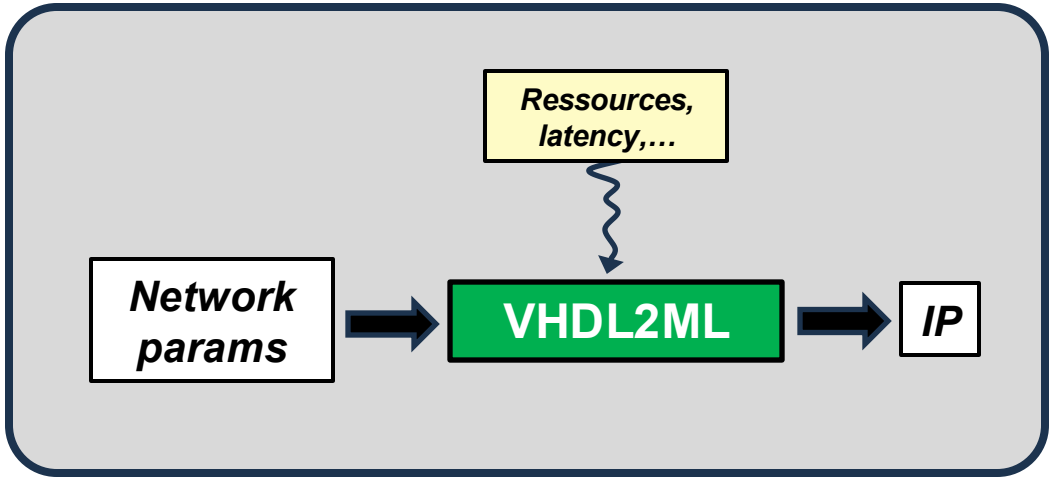
→ **Bayesian optimization** can help you to converge towards an optimal network topology

→ The optimization process tries to maximise a cost (*utility*) function. You can put the network efficiency there, but also sizing info.

→ You end up with a network combining **good efficiency**, but also optimized **inference complexity**

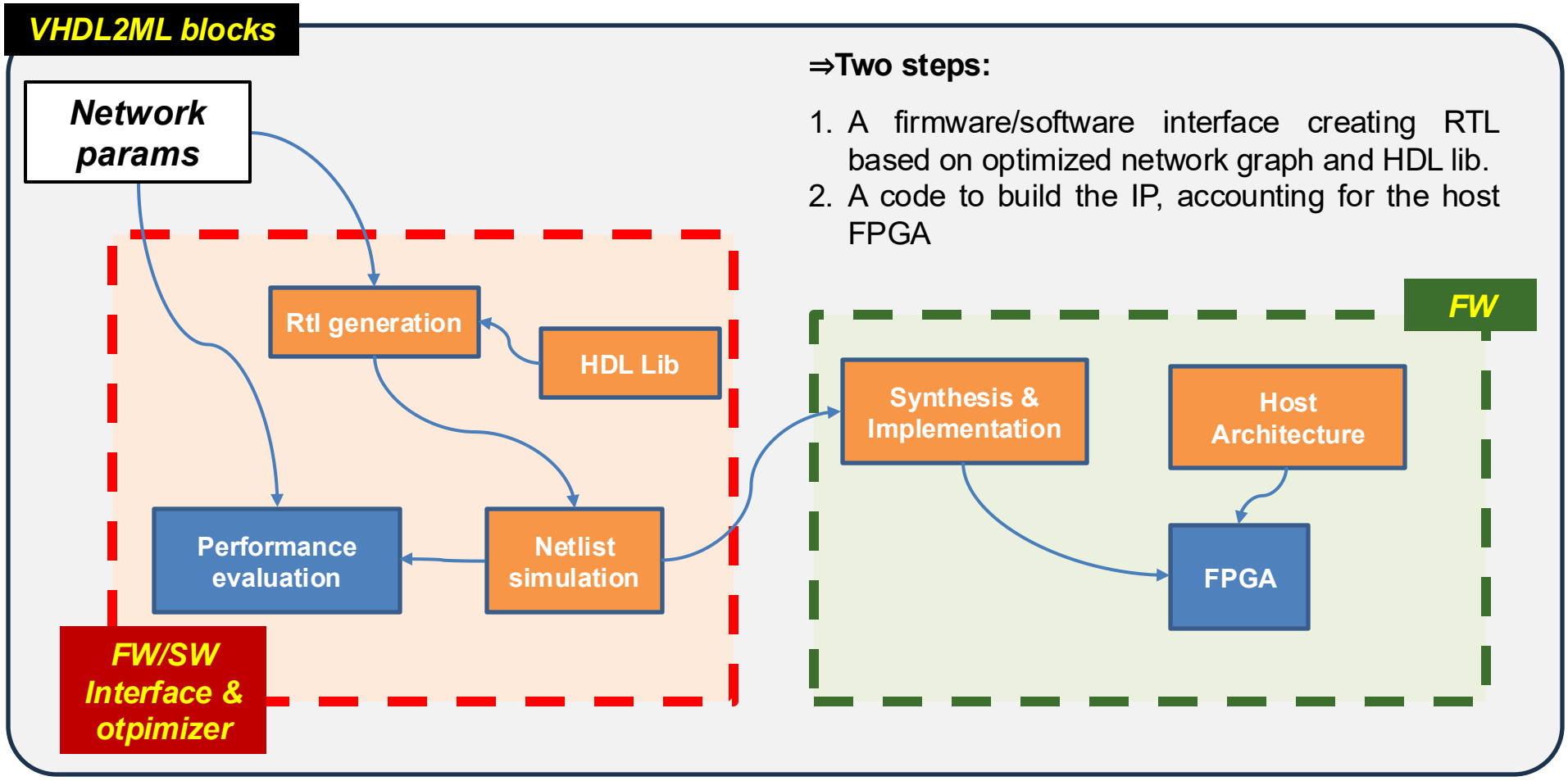


→ Applying this method to any type of network provides you with the **most online-friendly topology.** For an FPGA approach **it's mandatory** to have this done before going to the next stage. For a GPU-based, if you plan to go online, **it's worth the effort.**


Ressources, latency,…
Network params
VHDL2ML
IP

→ VHDL4ML implementation

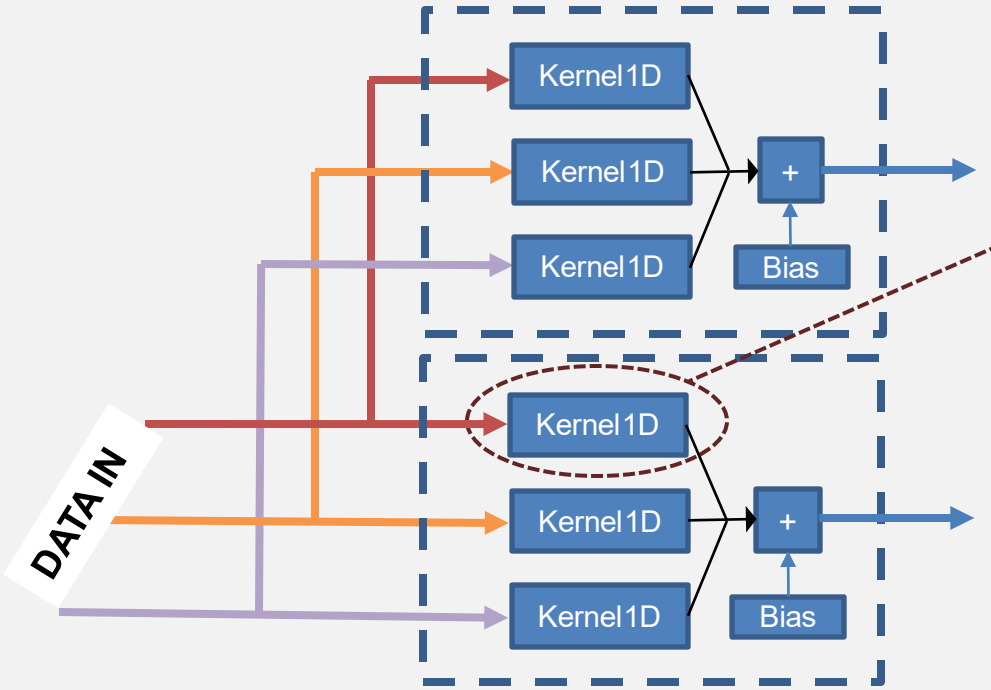
→ The key elements here are the **HDL library** and the **graph optimizer system**.



→ HDL library

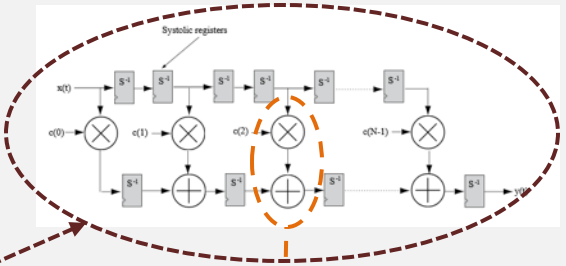
→ The library contains all the basic processing units (**PU**) necessary to build an **RTL version of neurons**

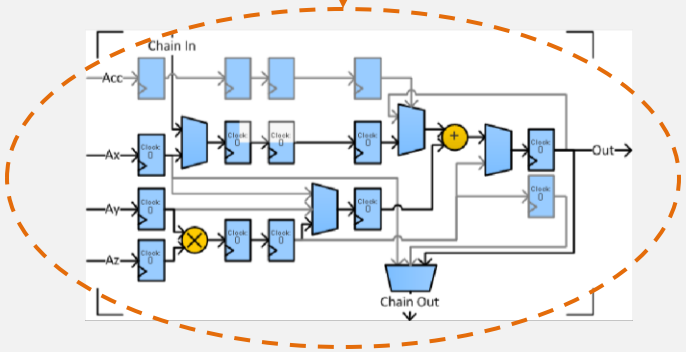
Example: Conv1D layer with 2 kernel of size 3



DATA OUT

Kernel 1D module = FIR filter



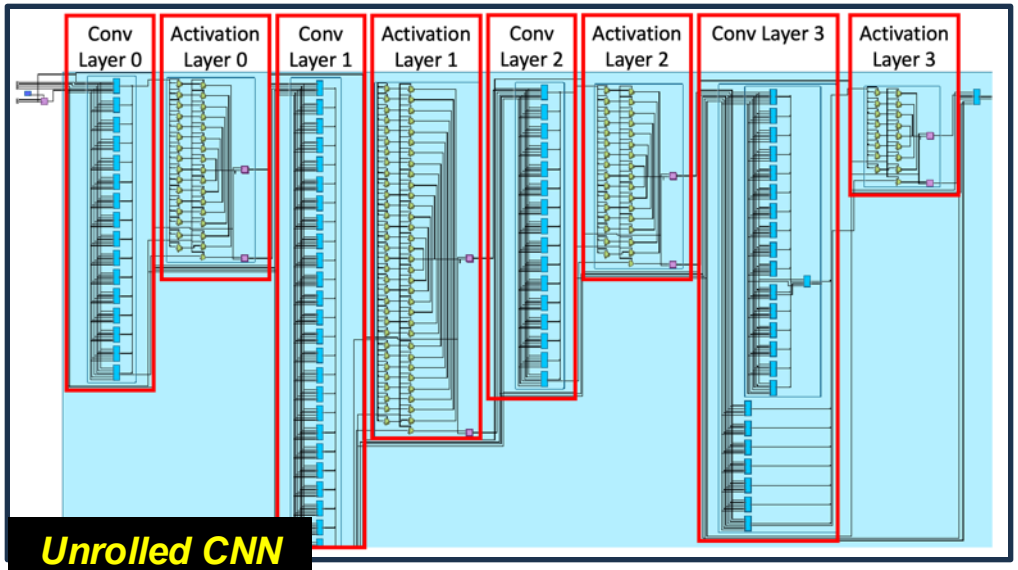


Low level PU implementation

→ **From the HDL lib one knows the latency/ressource budget of each PU**

→ RTL generation

→ For each neuron type (*CNN, MLP, Activation,…*), HDL lib provides a basic RTL model. From there it's easy to create two RTL codes for the network.



Unrolled CNN encoder

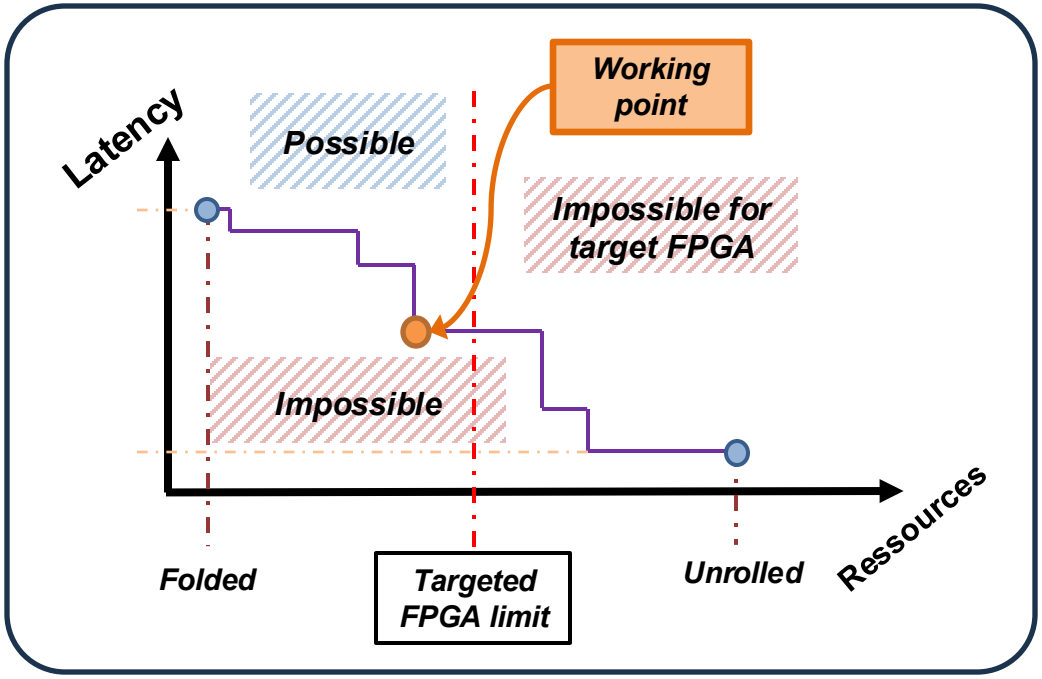
→ **Unrolled version** (*PU are used only once*). You get the best latency, but this is clearly suboptimal in terms of ressource usage.

→ **Folded version** (*only one PU used for all PU-related operations*). Best ressource usage, but poor latency.

→ One has to define an algorithm which, starting from one of those points and based on our latency/ressources constraints, finds an optimal working point

→ Optimizer principle

→ The optimization algorithm starts from one end.



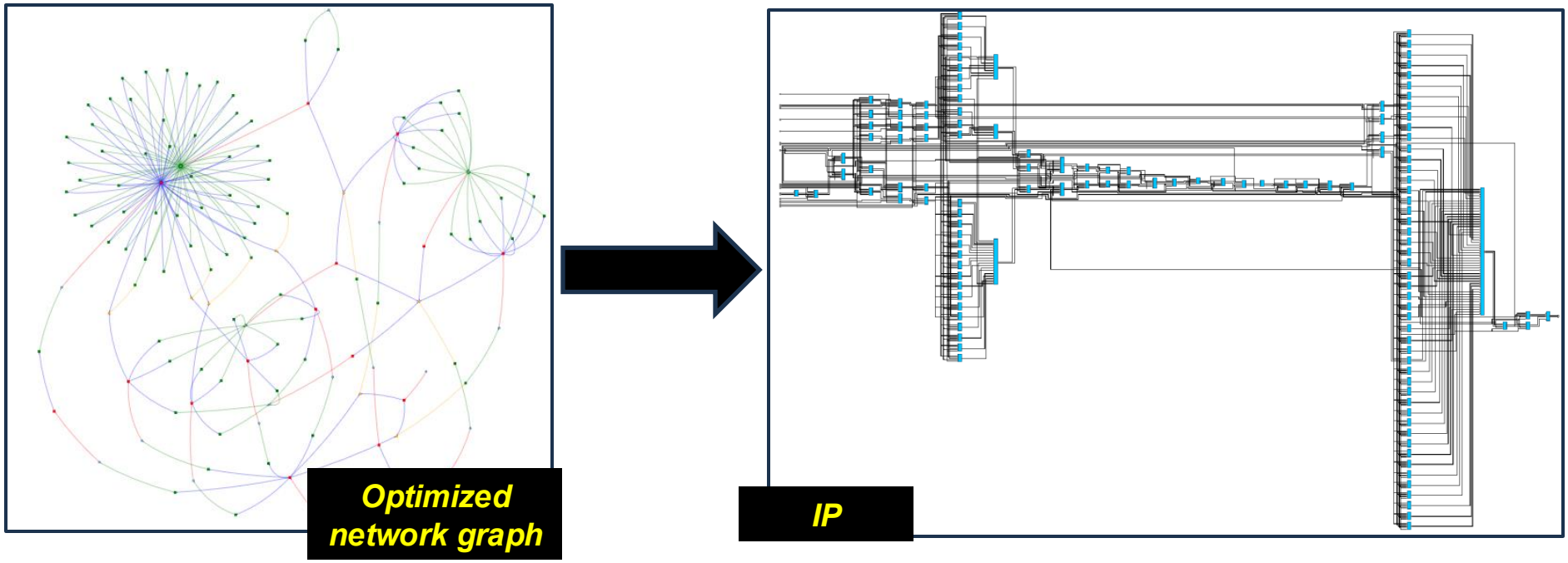
→ **The best working point depends on latency/ressources constraints,** and is found during the process.

→ The algorithm uses the latency/ressource parameters of each processing unit, there is no RTL involved here

→ Folding/unfolding iteration consists in respectively removing/adding processing units to the design.

→ Increasing processing unit reusability (*during folding*) causes the insertion of **memory buffers** and **multiplexers.**

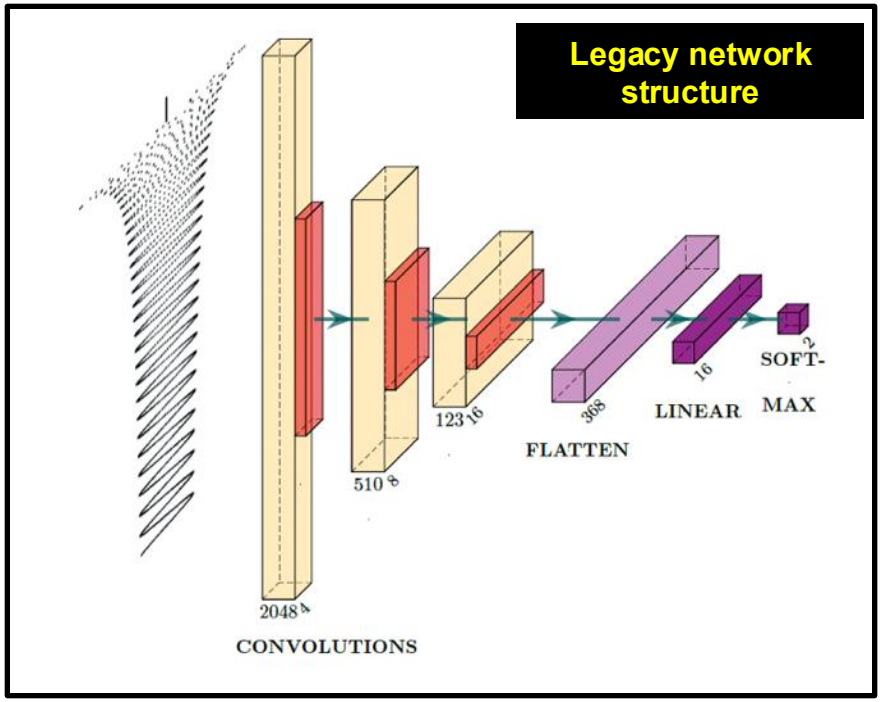
→ **From optimized network to IP:**



→ The best graph (*based on your latency/ressources constraints*) is synthetised and the network IP is created

→ The final scheduling (*given by the optimizer*) is handled by a **control unit** which is independent of the final network IP

→ Illustration on a simple use case:



→ Started from a legacy network (1701.00008v3).

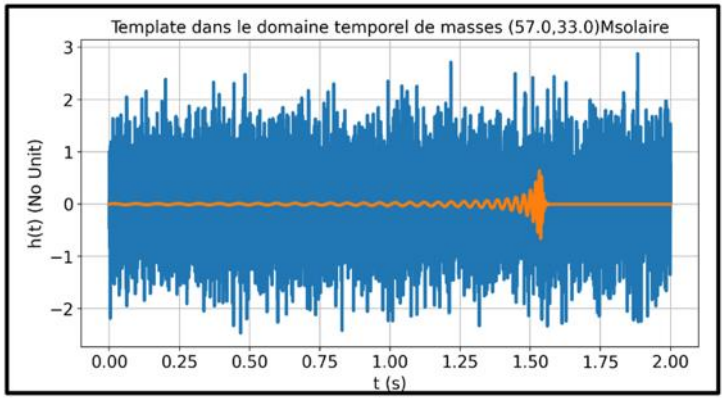
→ Very simple CNN encoder take 1s of the data strain as input (2048 points), a provides 2 values at the output:

- Compatibility of the data with signal
- Compatibility of the data with noise

→ There are much more elaborated architectures today, but this one is simple, efficient, and particularly attractive for FPGA implementation

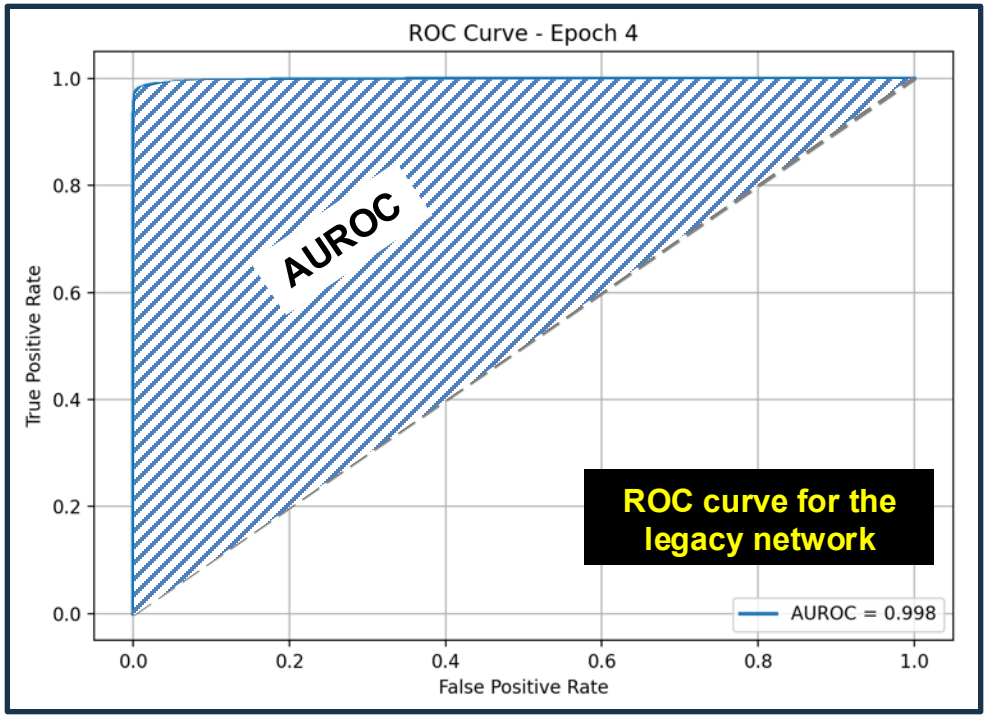
Number of parameters: 37094

Number of operations for 1 inference:
-> N additions : 1187600
-> N multiplications: 1209184



→ Make the network FPGA friendly:

→ Remove some **intensive steps**: batchnorm (*don't needed because data is whitened upstream*), use relu activations, remove the last softmax activation step (*signal cat over threshold is sufficient*)



→ Then try to adjust hyperparameters in order to maximise efficiency at low fake rate, and minimise of the number of operations and parameters (*relatively easy with a simple network*).

→ This work helps to define hyperparams priors, which can be fed to the bayesian optimizer to find optimal architectures.

→ The optimizer gives a reduced set of good results, just have to pick up the best one

→ FPGA-friendly encoder, structure and perf:

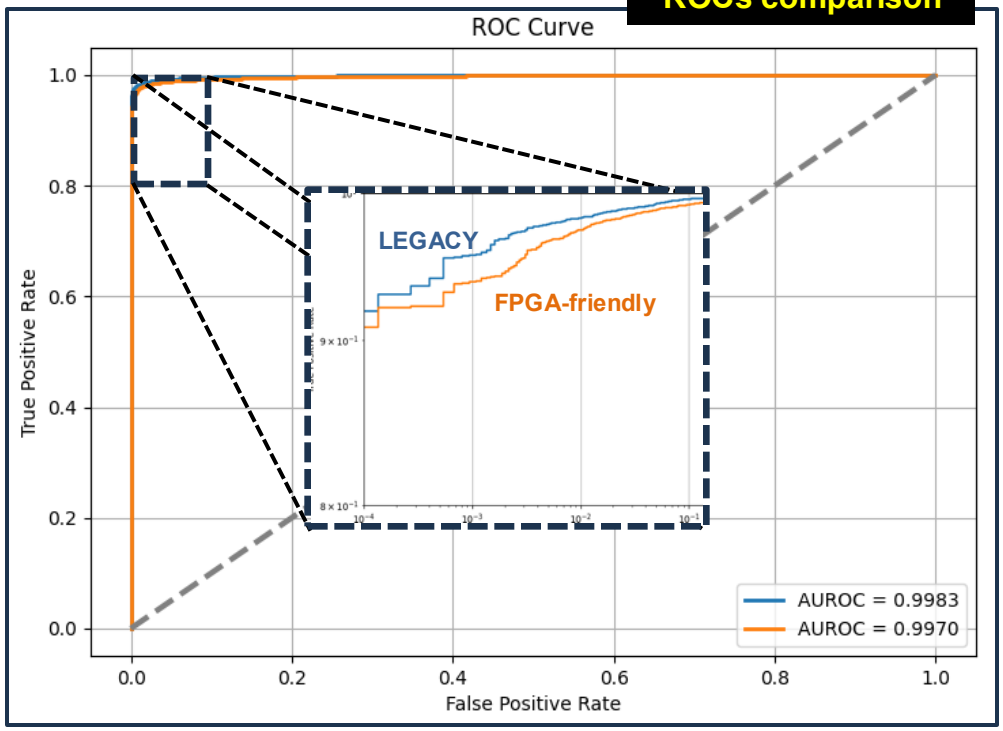
→ Best candidate from the optimizer:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 2048, 1)]	0
conv1d (Conv1D)	(None, 2041, 2)	18
max_pooling1d (MaxPooling1D)	(None, 510, 2)	0
activation (Activation)	(None, 510, 2)	0
conv1d_1 (Conv1D)	(None, 504, 7)	105
max_pooling1d_1 (MaxPooling1D)	(None, 126, 7)	0
activation_1 (Activation)	(None, 126, 7)	0
conv1d_2 (Conv1D)	(None, 122, 9)	324
max_pooling1d_2 (MaxPooling1D)	(None, 30, 9)	0
activation_2 (Activation)	(None, 30, 9)	0
conv1d_3 (Conv1D)	(None, 25, 8)	440
max_pooling1d_3 (MaxPooling1D)	(None, 6, 8)	0
activation_3 (Activation)	(None, 6, 8)	0
flatten (Flatten)	(None, 48)	0
dense (Dense)	(None, 2)	98

Total params: 985
Trainable params: 985
Non-trainable params: 0

Network structure

ROCs comparison



Number of parameters: 985 (37094)

Number of operations for 1 inference:
-> N additions : 131326 (1187600)
-> N multiplications: 131374 (1209184)

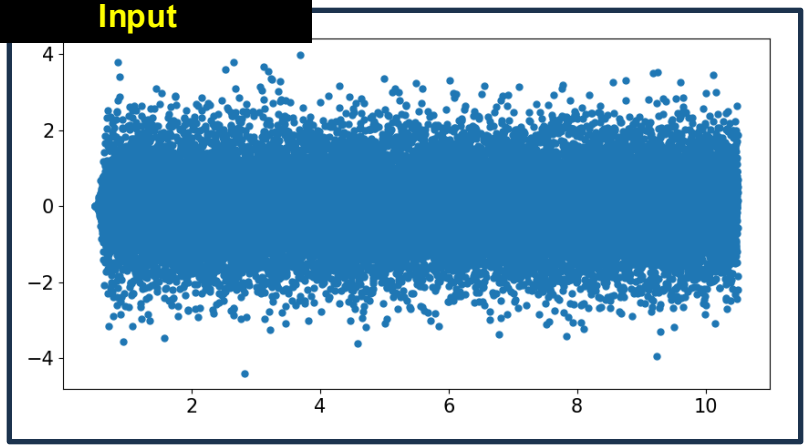
→ The performance loss is relatively negligible w.r.t. the **massive network size reduction**. This is a quick first look, there is room to improvement here.

→ **On the FPGA side this is clearly not the same story**

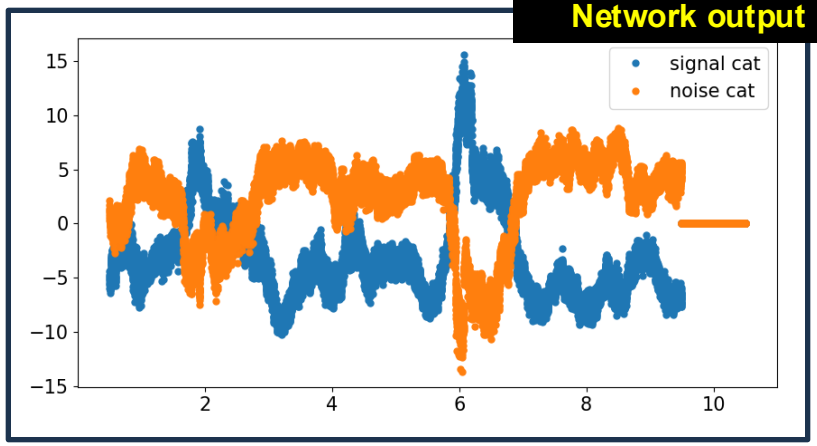
→ RTL implementation and simulation

→ 10 seconds of simulated data incl. one injection, compare the software network output (Tensorflow), with RTL simulation:

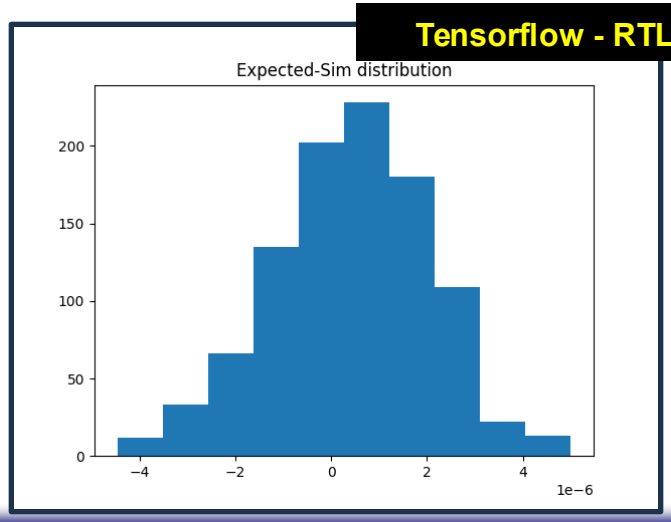
Input



Network output



Tensorflow - RTL



→ The RTL network is built with HDL lib processing units

→ **Output comparison validate the HDL lib.** This is a first important step.

→ Now start to test optimisation stage

→ Conclusion

→ Machine learning can play a significant role in the next generation of interferometers online architecture. But one should keep things compact and robust. **Frugality is the key here.**

→ **A lot can/should be done to optimize the network upstream.** Bayesian optimization is an important ally there, but this will work only if you understand what you're doing (*eg to choose the right network*).

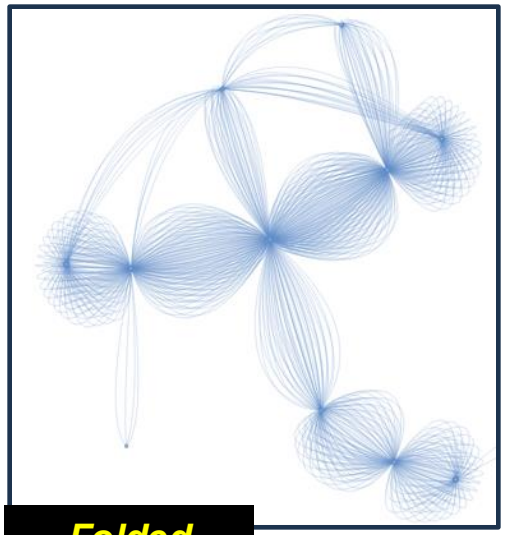
→ Once done, depending on the problematic you want to tackle, you can opt for a classic GPU-based approach, or for a low latency/low power hardware based approach.

→ We are currently developing tools linked to the latter solution, based on FPGAs. Our goal is to provide a library which can be used to implement optimally complex architectures on small FPGAs.

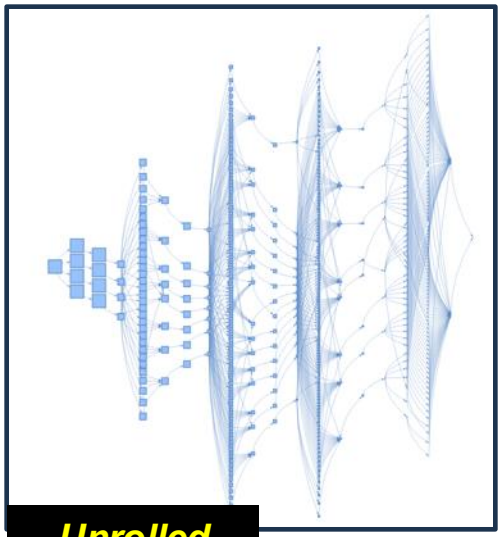
→ Can be the basis of a low-power ML-based firmwares for 3rd GEN frontend algorithm.

→ STEP 1: optimizer working principle

1. Construct the fully folded network graph: each node is a processing unit

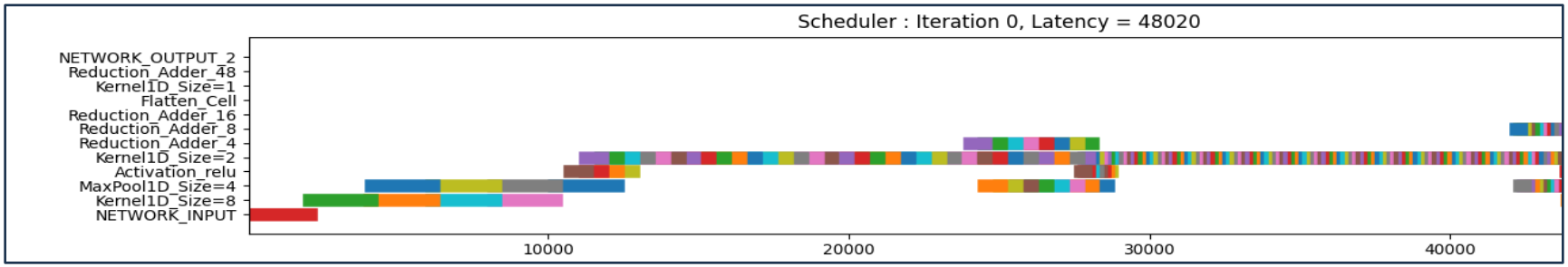


Folded



Unrolled

2. Estimate the total latency

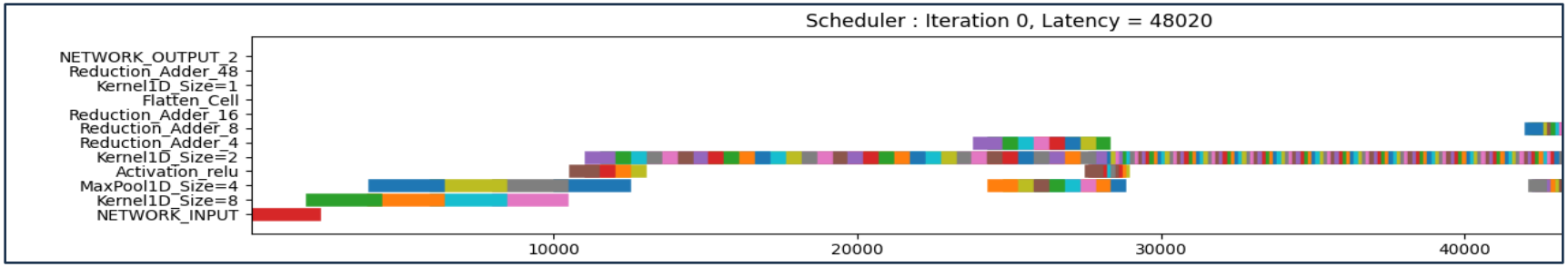


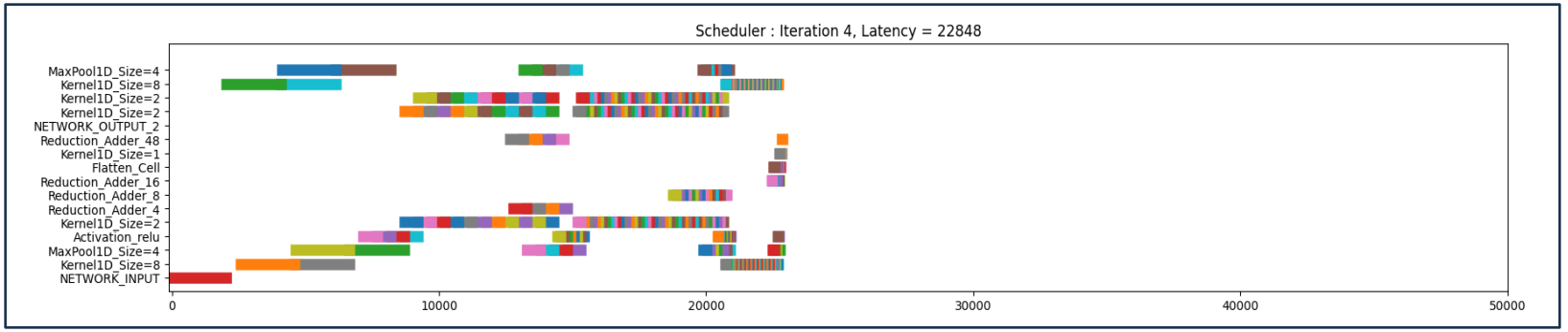
→ STEP 1: adding paralelization:

3. At each iteration, add some blocks (↗ ressources) to paralellize processing and reduce latency

4. Add memory blocks to handle data and network parameters

5. Add multiplexers to further simplify graph

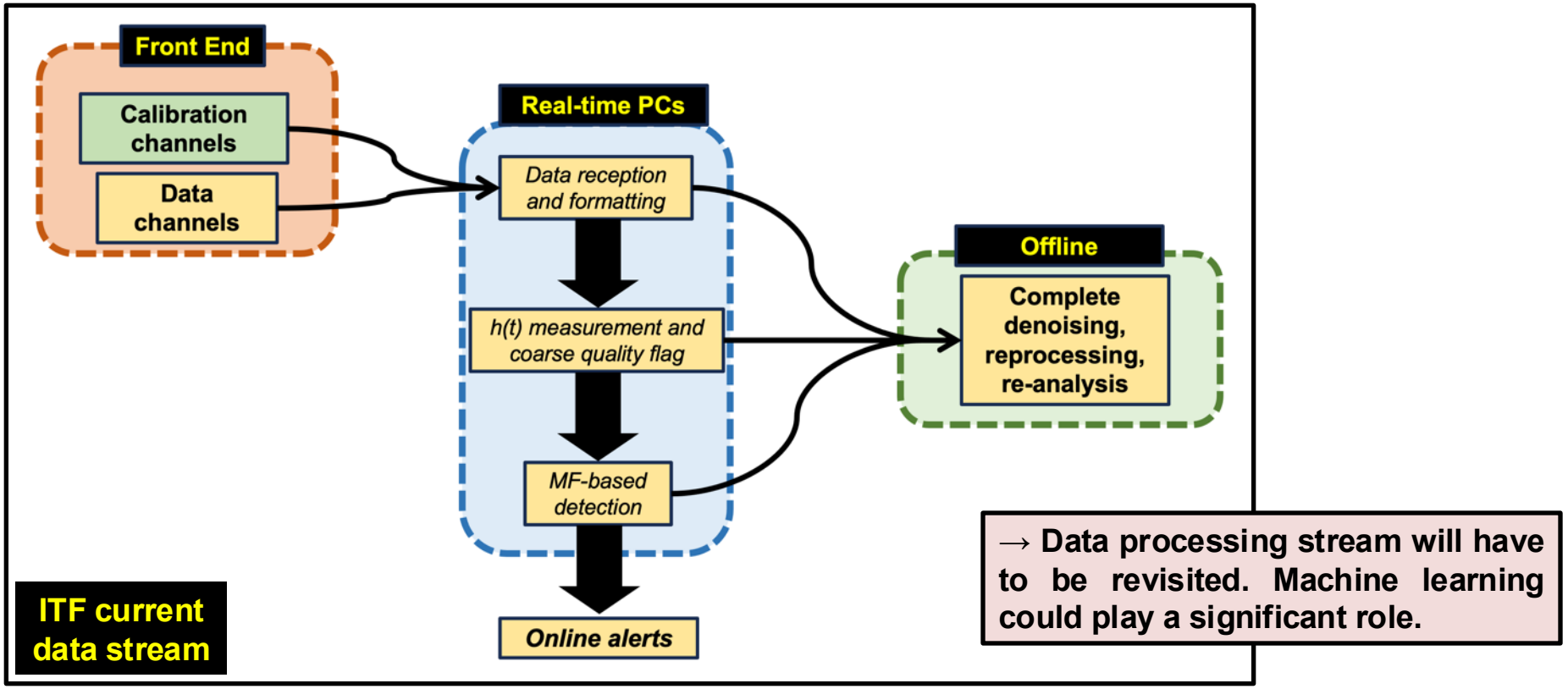




→ Context:

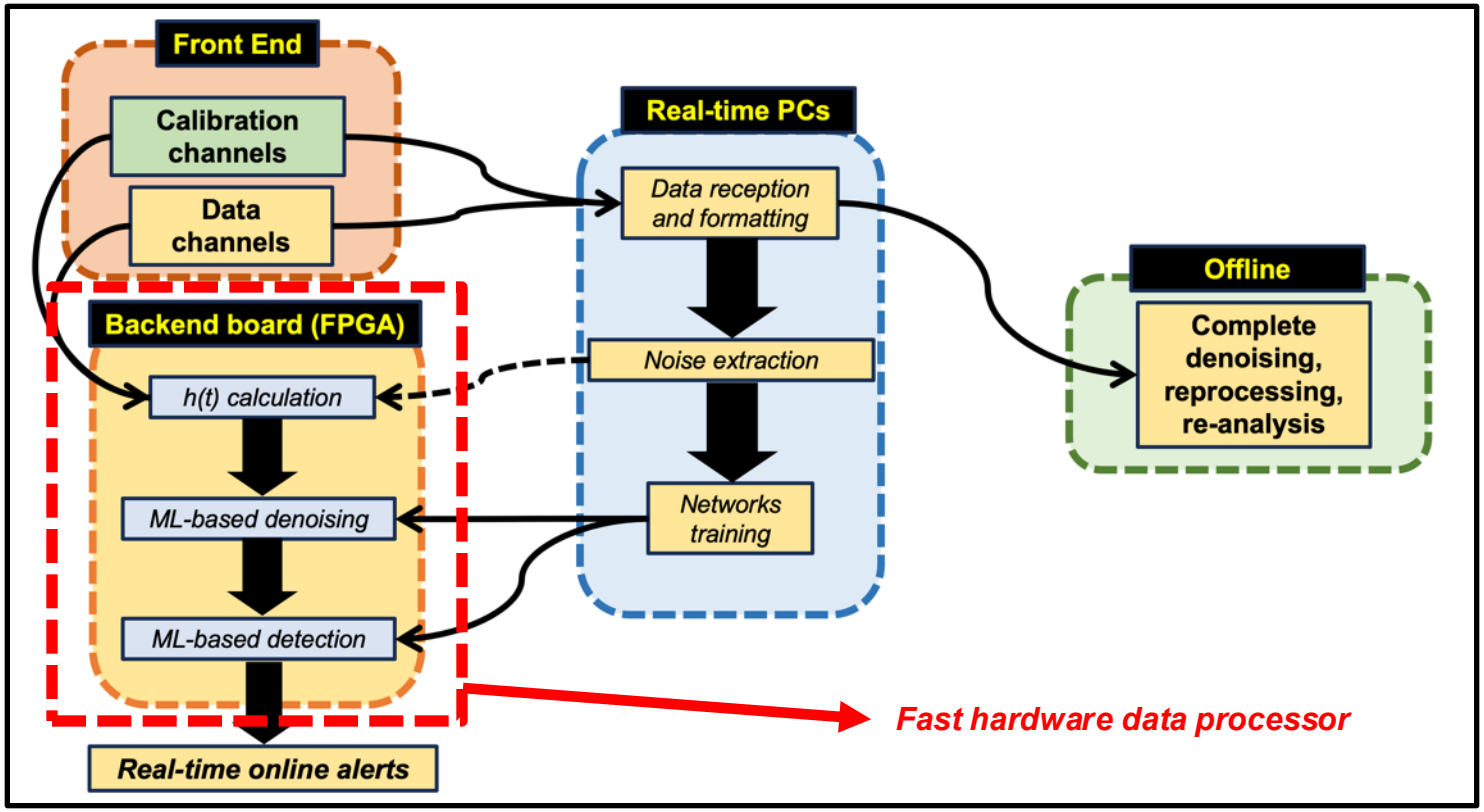
→ The next generation of GW ITFs will have 10x more sensitivity ⇒ 1000 times more events

→ You will start to experience pileup 😀… You will see many more BNS events with EM counterpart (*MM golden events*). You will need a fast and efficient online detection, **a kind of proto GW trigger.**



→ Why going hardware?

→ A fully ML-based DAQ for the next generation of detectors could look like that:



→ Denoising step (*hardware unfriendly*) can be skip in first approach.

→ **Next steps**

→ Add more complex cells to the HDL lib: currently looking at RNN cells and attention layers.

→ Continue to develop and improve the graph optimizer

→ Start to port simple architectures (*like the GW encoder*) on low level FPGAs (*eg CycloneV*)