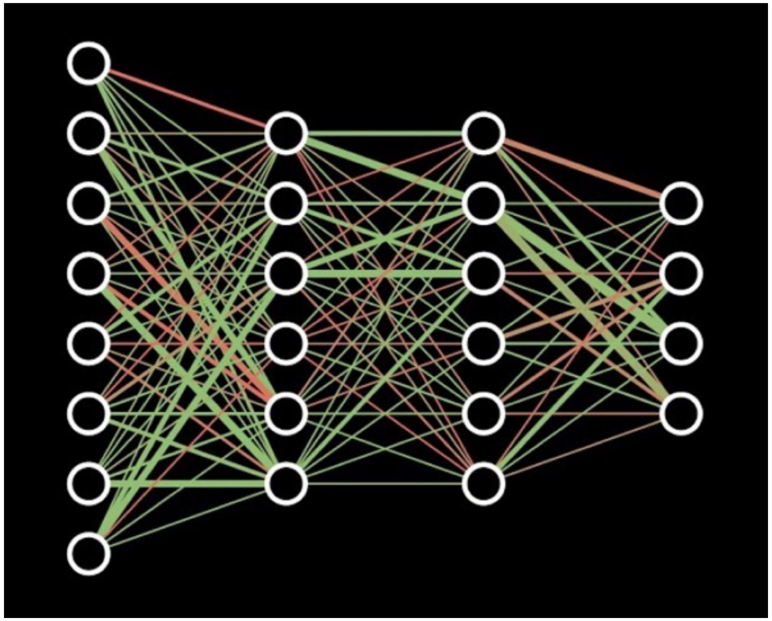
Machine learning for GW detection

A beginner's perspective



Introduction
Results
Conclusions and plans

I.Larbi, ***S.Viret***
IP2I Lyon

→ **Foreword:**

→ We are new to both gravitational waves and machine learning, so there might be evident points in this presentation for most of you.

→ All this is work in progress, we are open to comments, suggestions, discussion.

→ **Context**

→ The canonical GW detection method (***matched filtering***) is theoretically optimal. It's also relatively fast on current CPU based platform.

→ MF-based pipelines have been optimized for years, those algorithms contain a lot of useful informations on GW detection. **This knowledge base is a very important source for any new algorithm**

→ Machine Learning (ML) methods, if properly used, can provide a simplified approach to the detection. The complexity is buried offline in the training phase, the online part just becomes a set of scalar products.

→ Initial studies show promising results, but there are still some limitations (*eg long signals*)

→ What we have done?

→ This presentation is mostly a summary of the work done in 2022 by Idriss LARBI (*master student*). I took this over since October. IL report is available here:

https://gitlab.in2p3.fr/og-ip2i/og-ip2i-au-quotidien/-/raw/master/Internships/larbi_2022.pdf?inline=false

→ The goal of this work was to reproduce the results obtained by ***Huerta and George (*),*** and to develop a simple software framework to develop and test the different training scenarii presented in ***Shafer et al. (**)***.

→ Keep it very basic for the moment: simple template, simple noise, simple training sample structure, only two categories in output (*signal or noise*). We are in learning mode…

(*) https://arxiv.org/pdf/1701.00008.pdf
(**) https://arxiv.org/pdf/2106.03741.pdf

→ The tools

→ All the results presented here were obtained with the Python code available here:

https://github.com/sviret/DeepWave/tree/FFTW/DGWS

→ For the ML tools we use the d2l environment (*), which is pretty complete and heavily documented. We also use PyCBC to generate some signals.

→ I wrote a small tutorial page to install and use DGWS package:

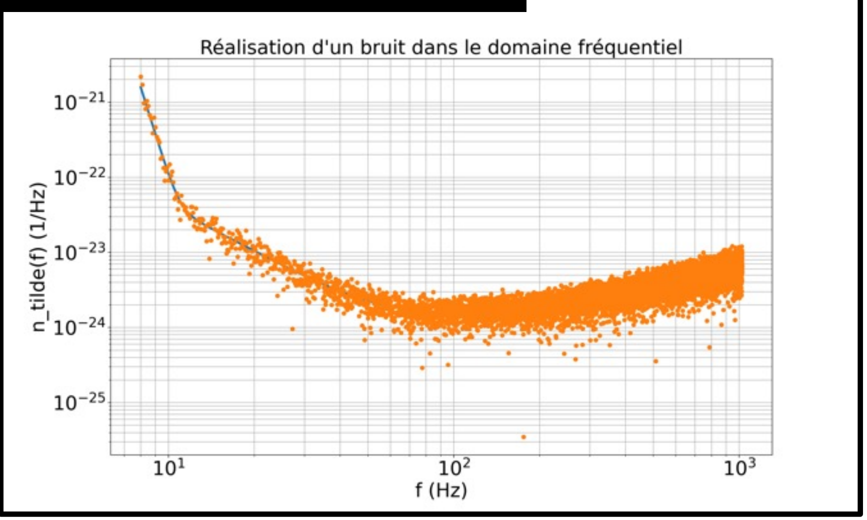
https://sviret.web.cern.ch/sviret/Welcome.php?n=Virgo.ML

→ This is all preliminary (*cf slide 1…*), but relatively simple to use. You should have everything to reproduce the results presented here with few commands.

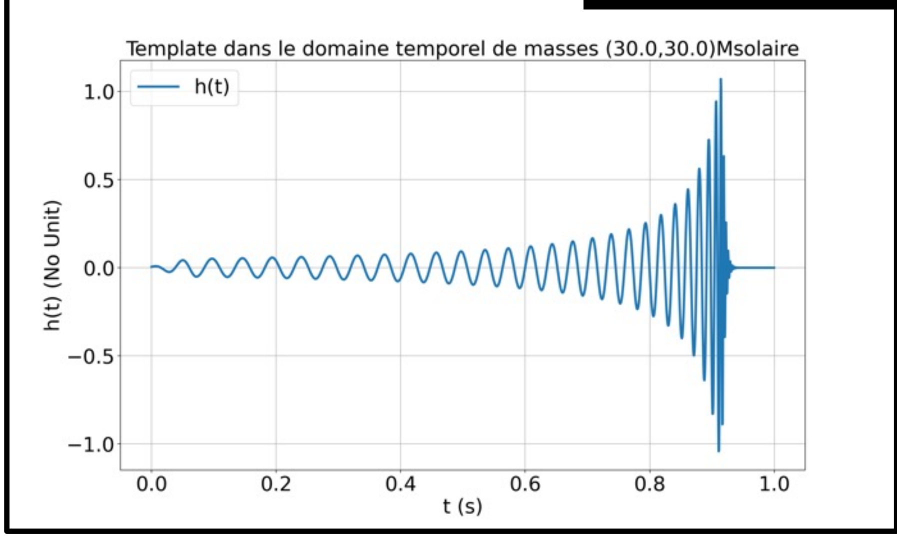
(*) http://d2l.ai/

→ Producing signals (*and noise*)

A noise realisation with colored PSD



A whitened template

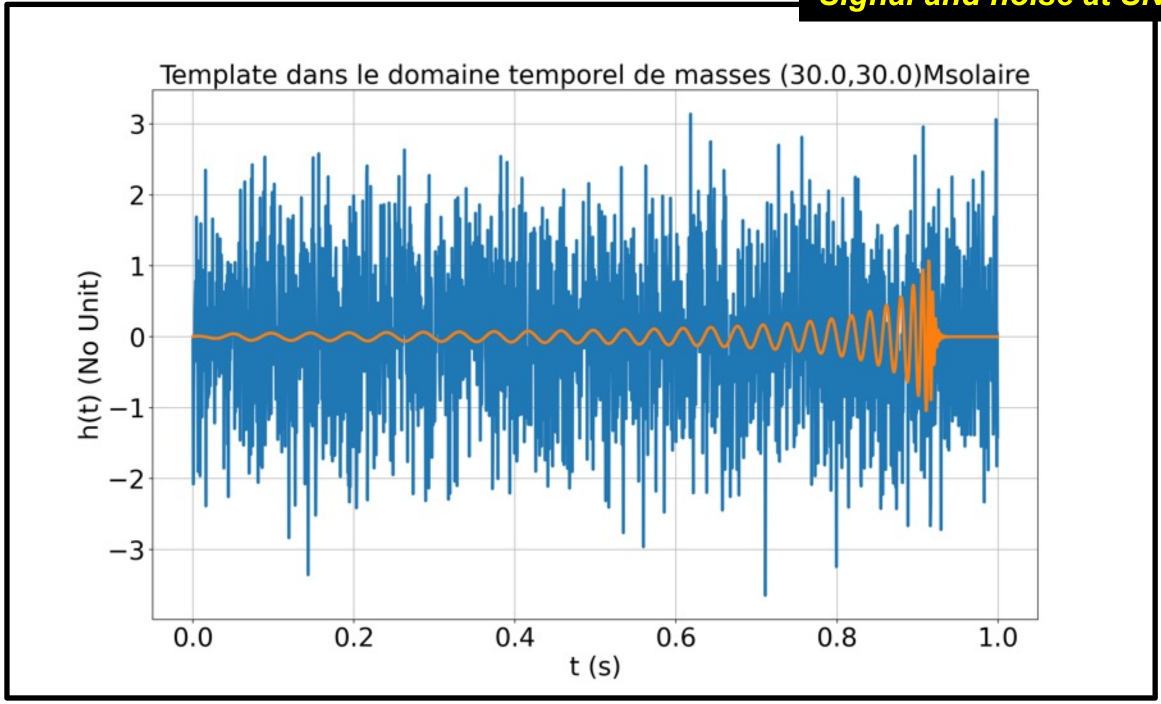


→ Simple macros to produce noise samples (*with flat or colored PSD*), and templates (*simple GEM model or SEOBNRv4 via pyCBC, whitened or not, sampling frequency,…*)

→ Easy to produce a training sample with any given config

→ Playing with SNRs

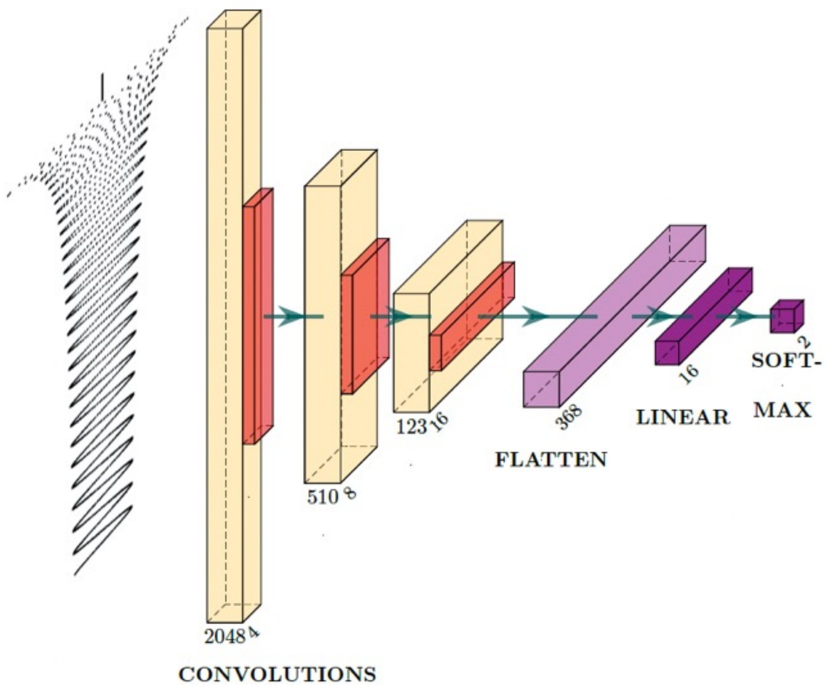
Signal and noise at SNR=10



→ As both noise and template are whitened, getting signal at a given SNR is fairly straightforward. For the training we will use different SNRs, starting from the same initial sample at SNR=1. The framework is taking care of that so it's transparent for the user.

→ The network

The tested CNN



→ Start with the simple convolutional neural network (***CNN***) from ***Huerta & George***:

- 1-dimensional input (*time serie*)
- 3 convolution layers
- 2 output categories

→ We use a lower sampling frequency and a lower mass range than in the initial study (*lower time, we're just using CPU for the moment*).

→ Here also it's pretty easy to modify the network, ML libraries come with a lot of practical tools.

→ Importance of the training parameters

→ **Network training strategy is fundamental, this is where you need to understand what you are trying to do.**

→ They are many knobs to tune:

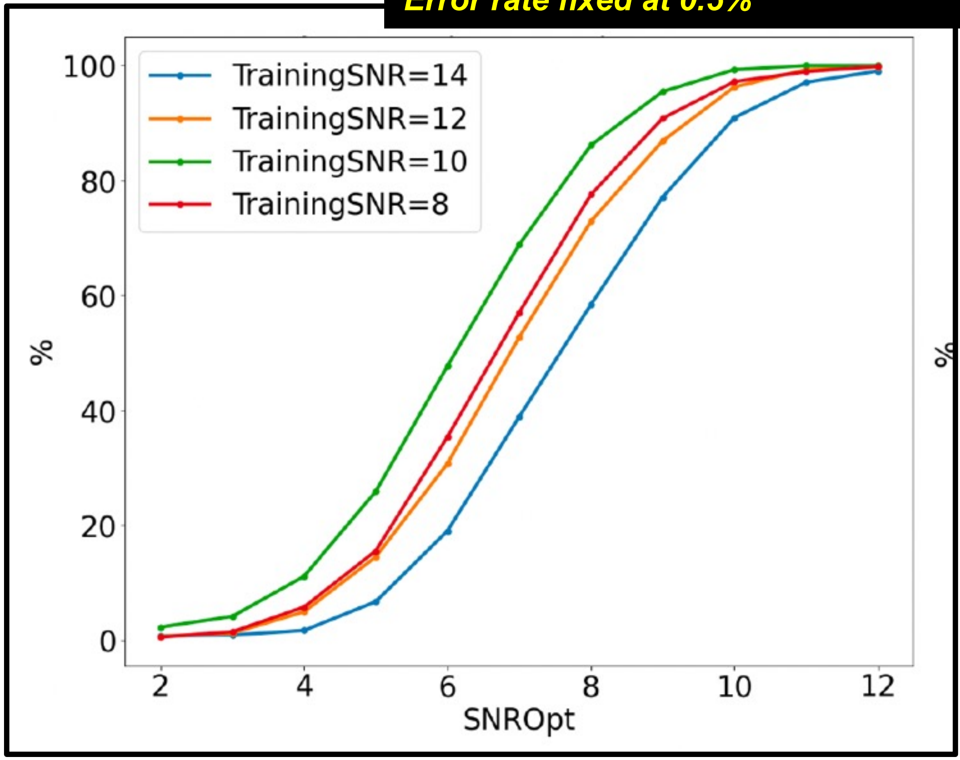
- *SNRs*
- *Learning rate*
- *Batch size*
- *Template bank*
- *Number of training steps*
- *…*

→ Of course most of those parameters are not independent, so finding the right combination can be a pain if you don't make some initial assumptions. The work by ***Shafer et al.*** is a very good illustration of the problem

→ **Deep understanding of the classical algorithms could be very useful here, in particular to define the training sample.**

→ SNR influence (*)

ROC curves for different training SNRs
Error rate fixed at 0.5%



→ If you train the network with a fixed SNR, very low values will not be optimal as convergence will become too complicated. Here SNR10 gives better results than SNR8

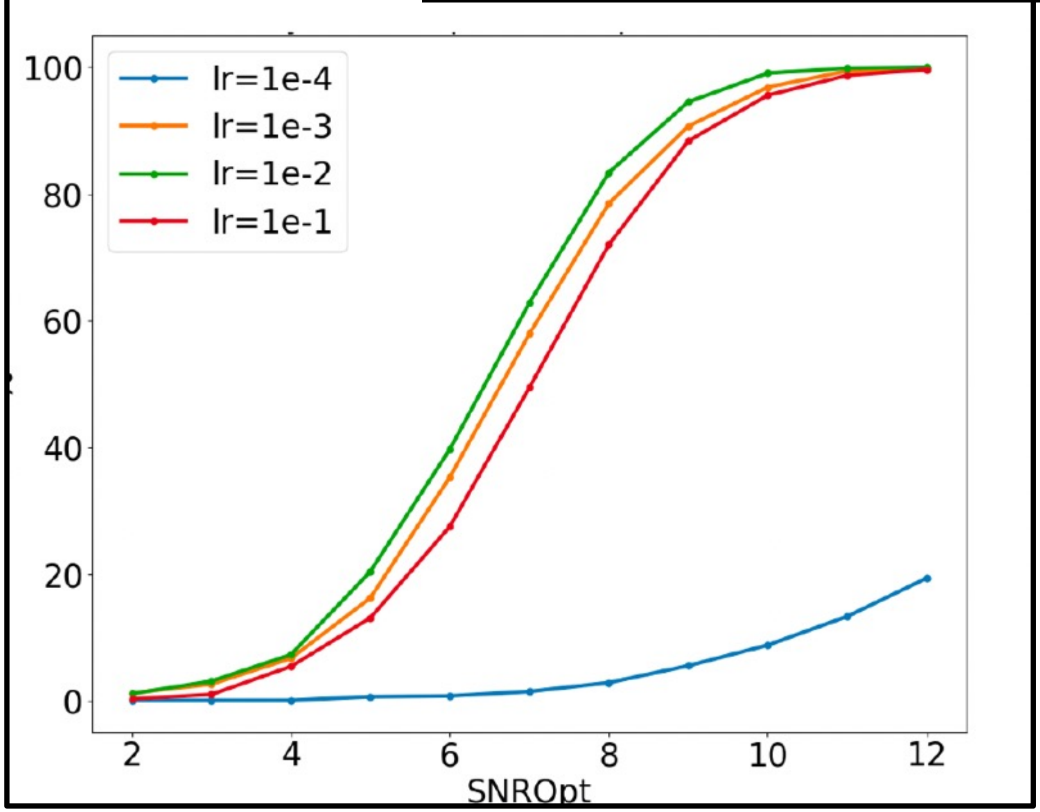
→ But if SNR is too large you loose efficiency on lower SNRs

→ The solution is to train the CNN with decreasing SNR values, from high to low.

(*) <u>Note</u>: noise used for these plots was lower than expected due to a bug in generation. Plateau is a bit too optimistic here

→ Learning rate influence (*)

ROC curves for different learning rates



→ For the learning rate you also have to find a tradeoff

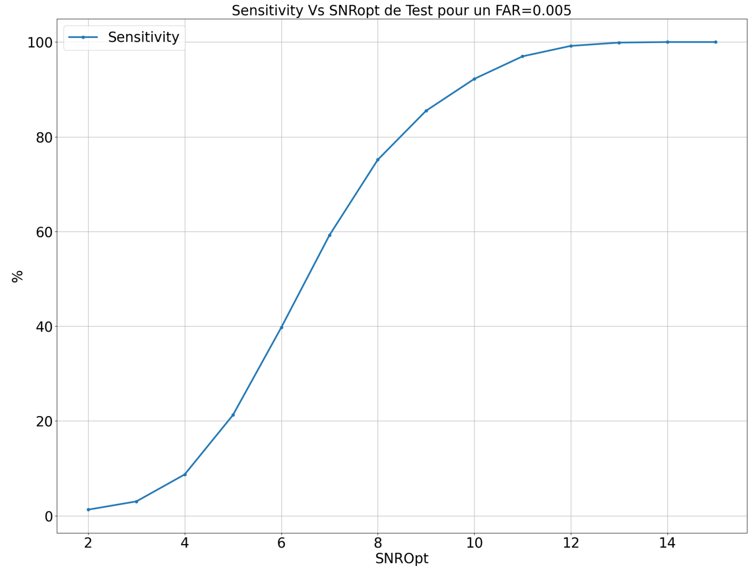
→ And so on for the other parameters, this is vast…

→ IL just had time to explore few parameters during his internship, there is still a lot of work ahead…

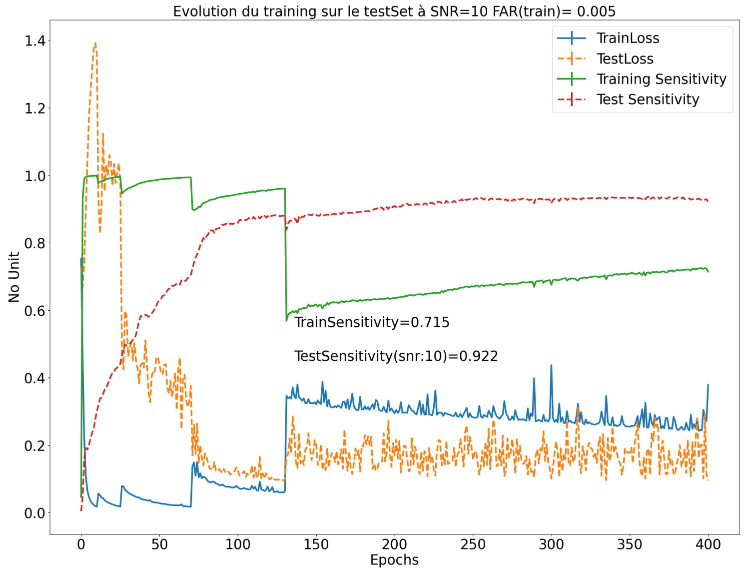
(*) <u>Note</u>: noise used for these plots was lower than expected due to a bug in generation. Plateau is a bit too optimistic here

→ Results with an ‘optimal’ training (*and correct noise*):

‘Optimized’ ROC curve



Training evolution



→ Result obtained with 400 epochs, at different SNR ranges.

→ Slightly worse than Huerta paper (*they reach plateau at 8*). But we see from the learning plot that there is possibly still some margin.

→ This is sufficient to validate our framework.

→ And now?

→ Until now we have more or less reinvented the wheel. But we have understood things, and developed a lightweight testing framework. We now have a toy to play with...

→ Among future tasks are:

- Go for GPU acceleration
- Pursue the work on training
- Test other more complex networks
- **Work on longer signals (*possibly with a 2 CNNs approach*)**
- **Increase the number of output categories**
- ***As said earlier we also are open to suggestions***

→ We also want to use this framework as a comparison platform for hardware-based approach. Indeed FPGA-based approach is not very interesting for the detection itself, **but it might help to drastically reduce the training time** (*clear limiting factor here*). We want to investigate that.