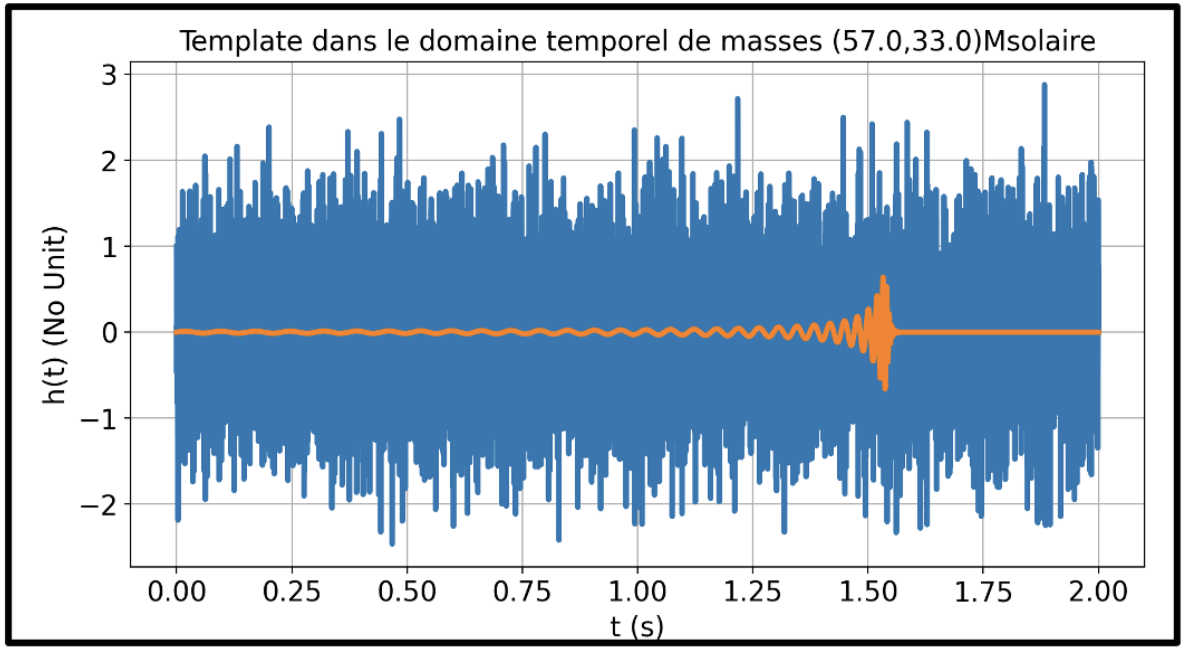
Using Machine Learning to detect Gravitational Waves



Introduction
The network
Performance tests

S.Viret

→ **Goals of the lab:**

⇒ Understand how machine learning can be used for GW detection.

⇒ Understand how to design and train a simple neural network.

⇒ Use a neural network on simulated noisy data in order to find events.

→ **How ?**

⇒ Some explanations (*those slides and me*).

⇒ Hands on sessions: where you work…

→ **What do you need ?**

⇒ A computer and **conda**

→ **STEP 1: install the tool:**

⇒ We will use a small python package (*a very light version of pycbc with machine learning features*):

https://github.com/sviret/MLGWtools/

⇒ First thing you have to do is to install it. Instructions are in the README file:

https://github.com/sviret/MLGWtools/blob/main/README.md

⇒ To check you're done, test this command:

```
python MLGWtools/tests/simple_example.py MLGWtools/generators/samples/noise.csv
```

⇒ If you get a noise plot, it works!

→ First steps, understand how the tool is working:

⇒ In the **`/tests/`** directory of the package you will find a lot of example scripts.

⇒ We will start with the simplest: **`simple_example.py`**. It will show you how to use MLGWtools produce noise samples or templates. Try this:

```
python MLGWtools/tests/buildframe.py MLGWtools/generators/samples/noise.csv
```

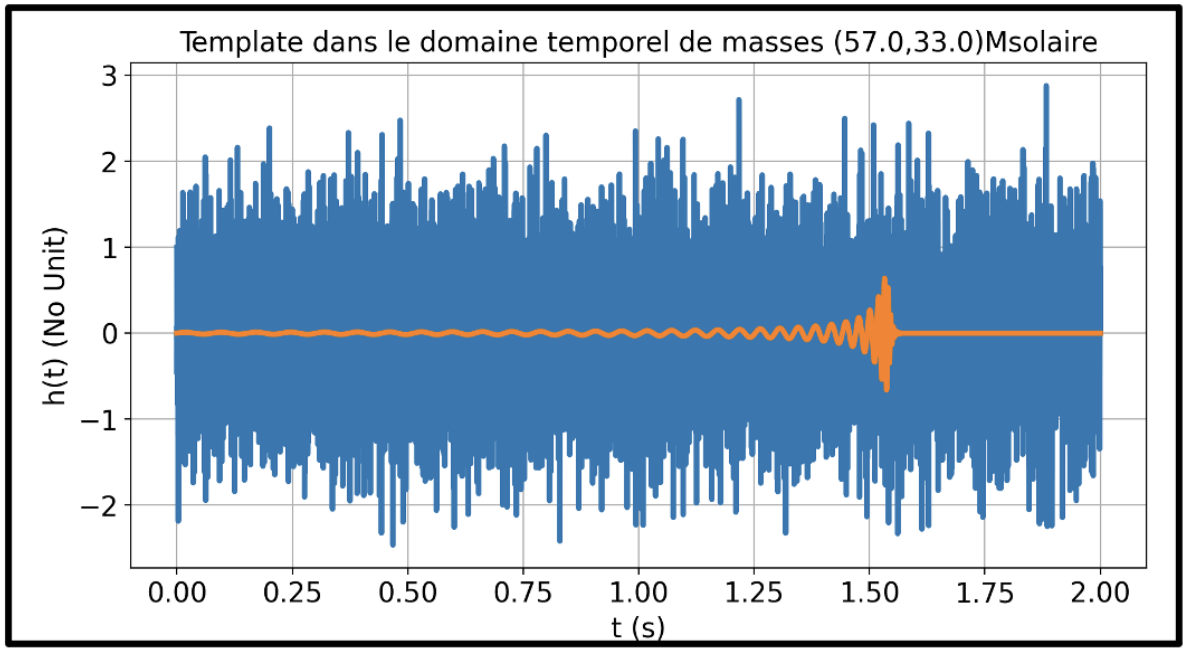
```
python MLGWtools/tests/buildframe.py MLGWtools/generators/samples/template.csv
```

⇒ As you will see, job parameters are stored in **csv option files.** Available options are commented in details. Try it yourself and modify some parameters.

⇒ In particular, compare the signals obtained with different PSD types, plot the PSD,…

→ Hands on 1:

⇒ Try do do this kind of plot, where noise and signal are superimposed and whitened. Do the same with raw data (***unwhitened***), knowing that the method to get raw data is **`signal_raw`**.



⇒ Try to change the signal to noise ratio

→ Detection and machine learning

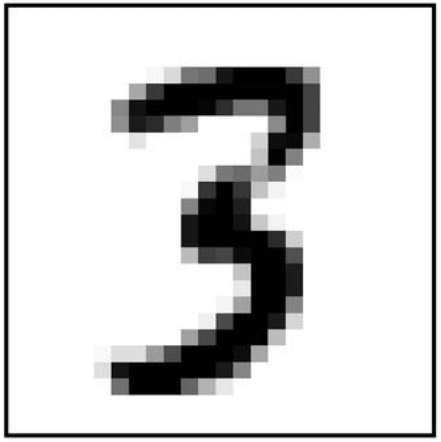
⇒ Until now you have seen how the detection is done with the **match filtering** method

⇒ Detection consists in finding a special shape drowned into a one dimensional strain. **This is actually one very popular topic in machine learning.**

⇒ When it comes to detection of feature, a weapon of choice is the **CNN** (**Convolutional Neural Network**). Was first proposed in the late sixties, and was popularized by LeCun, who was the first to use backpropagation to train them in the nineties.

⇒ Contrary to the perceptron which is a brute force approach, **CNN will sequentially extract features at different scales** (*it's actually inspired by the way our brain handles vision*).

→ CNN working principle: image recognition



⇒ In this example. you want to identify which number is on the image (*this is the famous **MNIST challenge***)

⇒ **Input** = image 28*28=784 pixels. Black and white.

⇒ **Output** = a number from 0 to 9

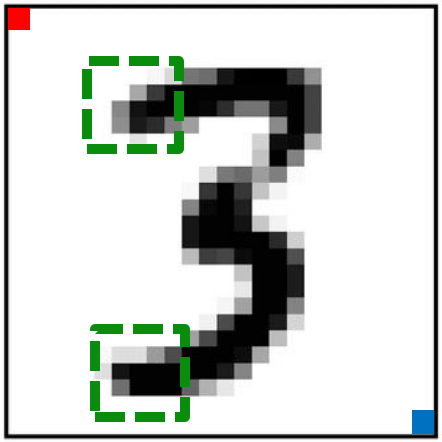
⇒ The challenge is to retrieve the correct number with the best possible accuracy

⇒ An MLP can do the job, but will require a lot of hidden layers and neurons. A nice comparison of MLP vs CNN on this kind of problem is shown here, for example:

https://a-damle.com/project_files/mlp_vs_cnn/mlp_vs_cnn.pdf

→ CNN working principle

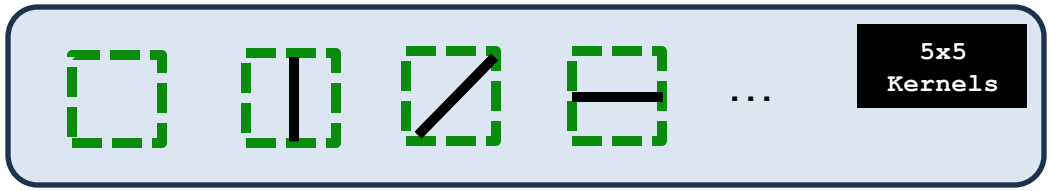
⇒ In an MLP, blue and red pixels will be connected. This is kind of nonsense because they are not sharing any info



⇒ Moreover there will never be a lot of info in a single pixel, it would be wise to look at larger arrays (***eg 5x5 pixels***), which could contain specific shapes.

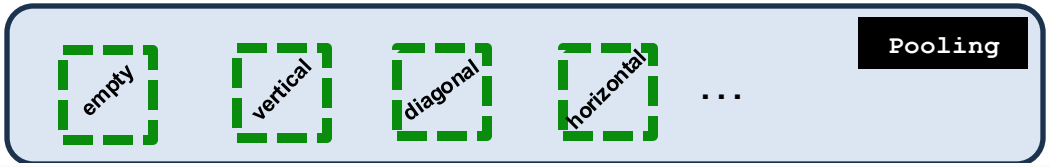
⇒ We will then scan the image with those arrays (*we call them* ***kernels***), and pass the output to the next layer

⇒ Each kernel will look for specific shapes



⇒ The kernels structure will be optimized by the training, you just need to provide the number of kernels you want and their sizes. All the kernels form the **convolution layer**

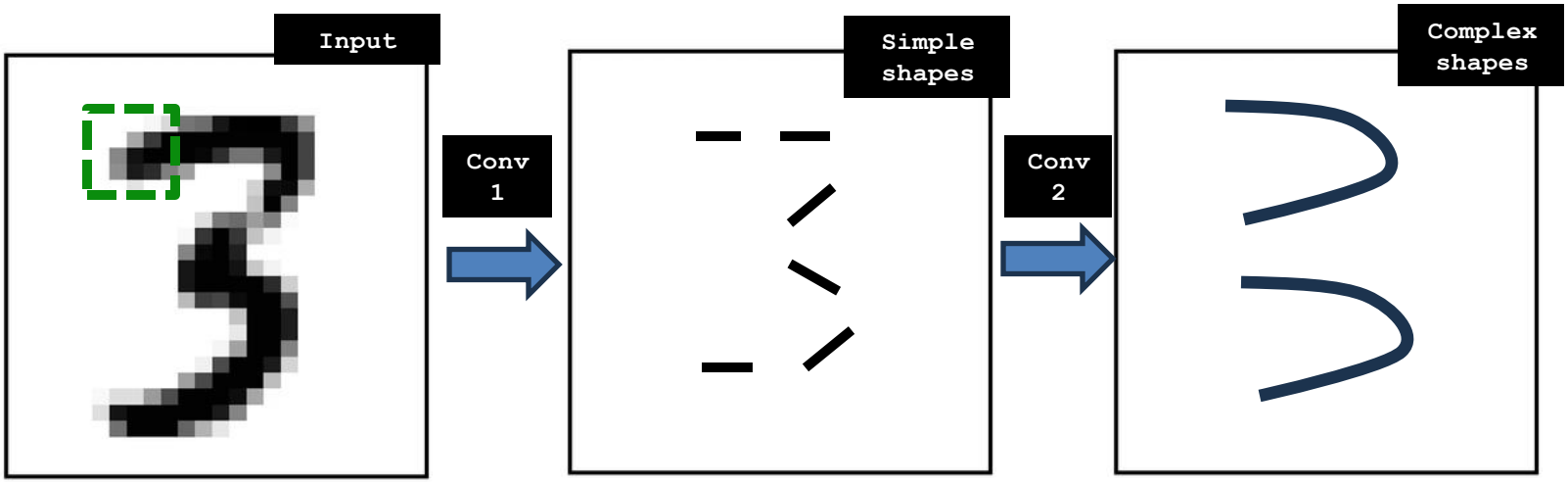
⇒ Convolution if followed by **pooling**, where you basically shrink the kernel output into a single info



⇒ Pooling is usually followed by an activation step. You keep only useful info

→ CNN working principle

⇒ After the pooling layer, you have less neurons, but each of them contains more info regarding its content. Said differently, the image size is smaller, but now it has a depth.



⇒ You can then transmit this info to another convolution layer where you will identify more complex structures, and so on.

⇒ If the images are simple (*like here*), 2/3 layers are sufficient. For larger and more complex images, more layers are needed. **Network design is a case by case thing** (*we will see it later*)

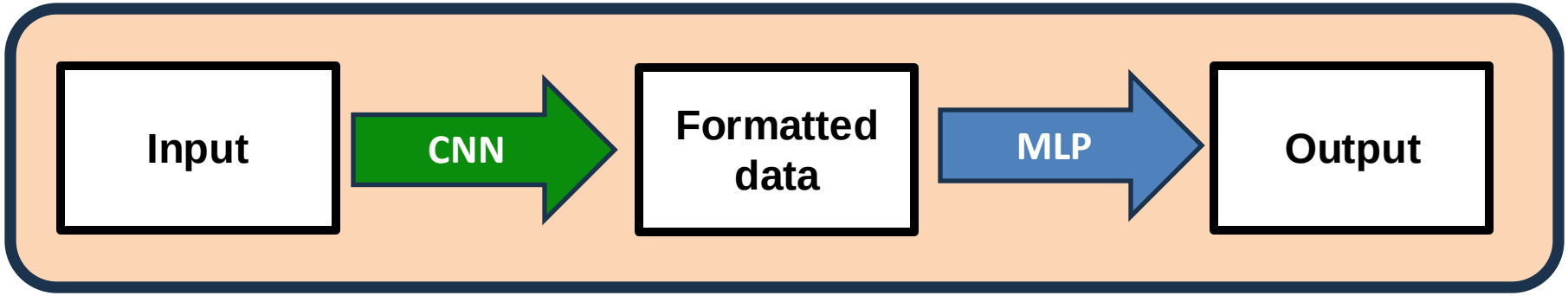
→ Decoding the final result

⇒ When you're satisfied, you have a certain number of neurons with a certain amount of info

⇒ Basically you have projected the initial info into a latent space where the info is much more wisely organized (***looks like some kind of super SVD in a sense***). **We call this part of the job an encoder**

⇒ The encoded data can now pass into a classic MLP; where the info will be processed to give the final categorization. **This is the decoder.** The MLP stage here is much more compact than if you had used a full MLP approach.

⇒ Putting both stages together we obtain what we call an **autoencoder**

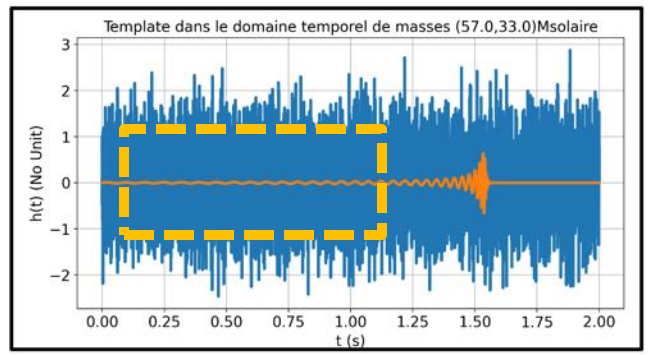


→ CNN-autoencoder for GW detection

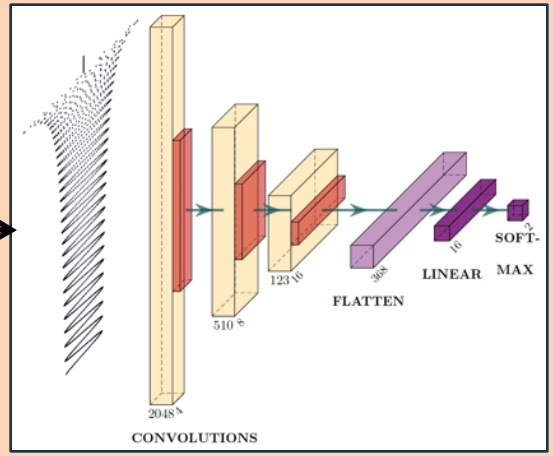
⇒ **2017**: First paper on GW detection with machine learning:

https://arxiv.org/pdf/1701.00008

⇒ The paper itself is relatively simple and clear, have a look! The network proposed takes in input 1s of data, feed it to 3 CNN layers (*encoding*), then pass trough 2 dense layers (*decoding*)



INPUT: we slide the 1s window over h(t), step could be variable, and in particularly much smaller than for match filtering



NETWORK

SIGNAL

NOISE

OUTPUT: two categories at the end: probability to have a signal or noise in the window

→ How to train a network?

⇒ Like for the match filtering, **we will use a bank of templates.**

⇒ But there is a major difference. In our case, we will just need the bank for the training. This is a massive gain in processing time (*don't need to process the whole bank for every new bunch of data*).

⇒ Another difference is that we will need a **bank of noises**. Network needs that to learn the difference between pure noise and signal mixed with noise.

⇒ Also need a **validation sample**, with different set of templates covering the same phase space.

⇒ With MLGWtools, all this is made by macro **`builsamples.py`**. Check how it works and the parameters of the csv options files, and try it yourself. By default the macro will produce the samples of the **`Huerta-George paper`**, but you can try other params (*different mass ranges for example*).

⇒ The take-home message here is that the training sample has to reasonably cover the phase space you want to detect. Tough neural networks can generalize, there is no magic.

→ Network training:

⇒ Now that you have the samples, it's time to train.

⇒ This is where things starts to become empirical…. There are even some papers entirely devoted to that topic: **https://arxiv.org/pdf/2106.03741**

⇒ We will give you some starting point, but you will test a bit by yourself. The training is done in the macro **trainCNN.py**. You launch it with a training scenario. An example is given in **simple_training.csv**.

⇒ For the moment in trainCNN we instantiate the standard encoder (*the simple one of the paper*). The training should take few minutes and work smoothly

⇒ Once the training is completed (*you get also infos during the training*), results are stored in a pickle file named **train_result***.p** You can plot the results using the macro **checkNet.py**.

```
python MLGWtools/tests/checkNet.py train_result***.p
```

→ Hands on 2: training strategies with a given network

⇒ For the moment we won't touch the network, just play with the training

⇒ Play with the different hyperparameters (*Number of epochs, SNR of the trained events, learning rate*) and check the performance of the training

⇒ Find a strategy leading to **overfitting**, another one leading to **underfitting**. Try to understand what is causing this state (*in other words start to understand what is a good or bad strategy*)

⇒ **BONUS:** try to find a better training strategy than the one you started with

→ **Network structure:**

⇒ OK, you know how to train a network, but what about the network itself????

⇒ Here also heuristic approach is often the only rule (*tough things are changing now*), but some common sense can drastically reduce the work needed.

⇒ You will exercise this now

→ Hands on 3: playing with the network structure

⇒ Create a network with another shape, using the `custom_net` method present in trainCNN.py, and train it. You can check its performance with checkNet.

⇒ Create a macro which compares the performance of different network types.

⇒ Do you gain something with more/less CNN/Dense layers, more filters per layers,… ?

⇒ How does it changes the computing time?

⇒ **BONUS:** try to design and train the deeper network of the **Huerta/George paper (Fig.6)**. **Hint**: we are starting from a 2048 vector, not 8192, keep that in mind when defining the kernel sizes

→ Detecting events with a network

⇒ You have now a trained network. You know it's performance from the validation sample.

⇒ We will show how to test it on data. Here we will work on simulated data, but the principle is exactly the same for real data.

⇒ This is a 3-step process

⇒ **Step 1:** produce a data frame

⇒ **Step 2:** pass it trough the network

⇒ **Step 3:** study the output

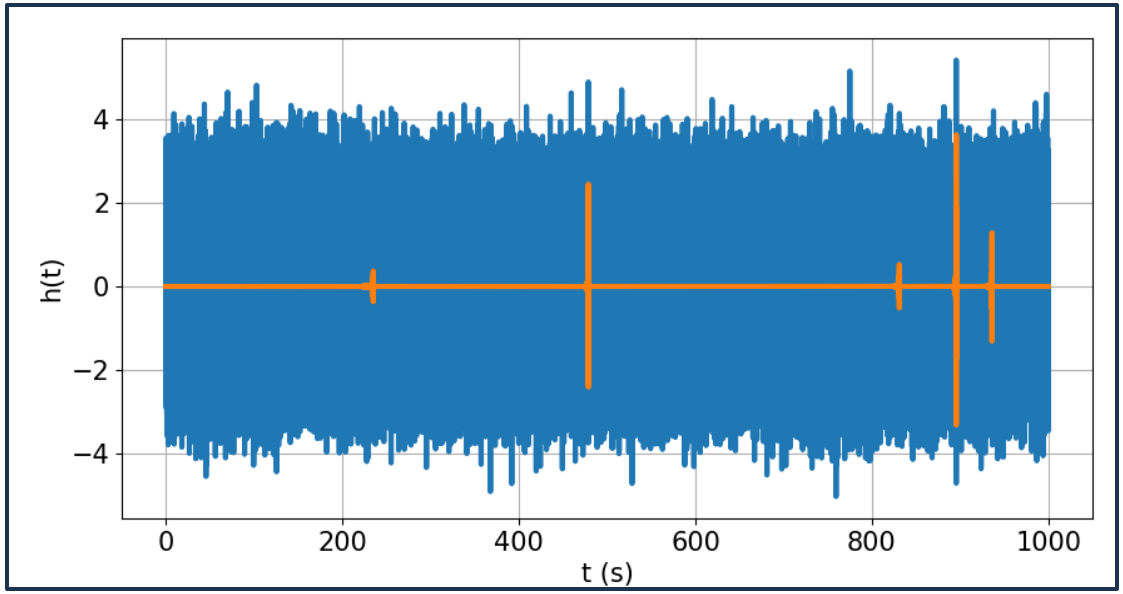
→ Produce a data frame

⇒ After the training, the network parameters are stored in another pickle file (**`network_***_fullnet.p`**). You can display some info about it (*and plot it*), using the macro **`networkinfo.py`**.

⇒ This pickle file is one of the input of the macro **`useCNN.py`** which processes and input frame with the trained network.

⇒ To build a simulated data frame you will use the macro **`buildFrame.py`**, the content of the frame is defined in option file **`Frame.csv`**.

⇒ It will produce a strain of data with some signals randomly injected within it. Information about injected signals is kept, so that you can check the efficiency of your macro afterwards.

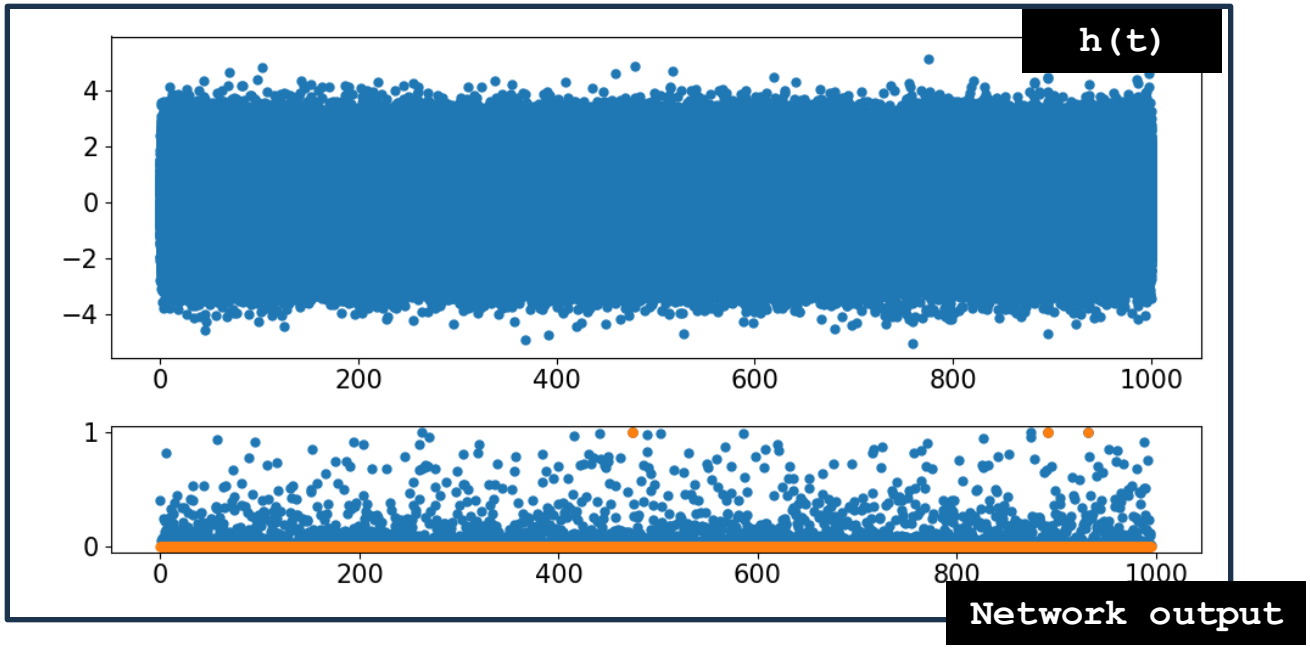


→ Hands on 4: data analysis

```
python useCNN.py -f Frame**.p -n network***.p -s **
```

Last argument is the step size, in s, between 2 inferences.

⇒ After the input stream passes trough the network you end up with an output vector containing the probability of having a GW signal has a function of time



⇒ You will have to develop a macro to analyze this vector in order to find the potential events. You will compare this to the list of injections and compute the efficiency of the network vs the SNR, along with the fake rate

→ Hands on 4: data analysis

⇒ Some hints:

⇒ What is a good signal ? Is a single high output sufficient ? If we require a cluster of consecutive high outputs, which cluster size threshold can we define? And the signal threshold for the hit to belong to a cluster?

⇒ Which criteria can we define to decide whether a signal if matching an injection or not? Which parameters can we compare?

⇒ You have matched clusters and unmatched clusters, how do you evaluate the fake rate?

⇒ How does the cluster size depends on chirp mass, on SNR???

→ **Mini project:**

⇒ You have 1 frame of 1000 seconds, containing 5 events

⇒ 1. Find the possible signal regions with the neural net

⇒ 2. Use match filtering to determine event properties

⇒ 3. Can you propose a modification to the autoencoder in order to get event properties directly from the network?