SAPIENZA
UNIVERSITÀ DI ROMA

Using Deep Convolutional Neural Networks to build a low-latency classifier to trigger the detection of long gravitational-wave transients

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di laurea magistrale in Fisica

15 Dicembre 2017

Candidato:
Federico Muciaccia
matricola 1628218

Relatori:
Pia Astone
Fulvio Ricci

Controrelatore:
Paola Leaci

Anno accademico 2016/2017

Contents

1 Introduction

We propose a method to analyze gravitational-wave data in the time-frequency plane by using an Artificial Neural Network performing classification as an image recognition task.

We have built a Deep Convolutional Neural Network that is able to simultaneously process the data from the three interferometric antennas of LIGO Hanford, LIGO Livingston and Virgo. Our model is optimized to search for long $\mathcal{O}$(days) gravitational-wave transients, but the classifier can also be straightforwardly extended to also classify noise structures, thus providing a future aid on disturbances removal during the online calibration.

The method proposed is characterized by a very fast computation time during the prediction/classification phase, thus enabling the possibility to build a low-latency trigger to allow faster messaging between gravitational-waves detectors and their electromagnetic counterparts: this can widen the possibilities of future multi-messenger astrophysics. The low-latency trigger can also contribute to reduce the computational burden of the current offline analysis pipeline and follow-up.

We trained the model using data from the most recent observational run (O2 C01) of the three-detector network. The classifier is able to reach more than 90% detection efficiency and less than 1% false alarm rate with a signal *time domain* strain much smaller than the equivalent gaussian white noise *time domain* standard deviation $h_S(t) \sim 4 \cdot 10^{-4} \sigma(h_N(t))$. The *frequency domain* signal+noise strain is thus just above the median of the sensitivity curve.

We will also show how there is still room for substantial future improvements.

2 Gravitational waves: ripples in the spacetime metric

Gravitational waves are predicted by Einstein's General Theory of Relativity [47, 48]: perturbations of the gravitational field should propagate as waves. When a mass-energy distribution changes in time, the information about this change should propagate at the speed of light in the form of waves. Gravitational waves are *metric waves* because $g_{\mu\nu}$ is both the metric tensor and the gravitational potential. Thus, when those ripples in the metric of spacetime propagate, the geometry will change in time and, consequently, the proper distance between spacetime points will also change.

Since Einstein's equations are non-linear, arbitrarily strong gravitational waves do not obey linear superposition, making their description difficult. However, for weak fields, a linear approximation can be made, which is accurate enough to describe the exceedingly weak waves that are expected to arrive on Earth from very distant cosmic events. These waves typically result in relative distances increasing and decreasing by 10^{-21} or less. Indeed, current data analysis methods routinely make use of the fact that these linearized waves can be Fourier decomposed.

2.1 Einstein's field equation

Einstein's General Theory of Relativity can be mathematically summarized with the Einstein's field equation [46]

$$R_{\mu\nu} - \frac{1}{2} R\, g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu} \tag{1}$$

where we used Einstein's implicit summation rule and

- $R_{\mu\nu}$ is the Ricci curvature tensor, defined as a contraction of the Riemann curvature tensor $R_{\mu\nu} = R^{\sigma}_{\mu\sigma\nu}$ which is in turn defined as

$$R^{\rho}_{\sigma\mu\nu} = \partial_\mu \Gamma^{\rho}_{\nu\sigma} - \partial_\nu \Gamma^{\rho}_{\mu\sigma} + \Gamma^{\rho}_{\mu\lambda}\Gamma^{\lambda}_{\nu\sigma} - \Gamma^{\rho}_{\nu\lambda}\Gamma^{\lambda}_{\mu\sigma} \tag{2}$$

  where $\partial_\mu \equiv \partial/\partial x^\mu$ and $\Gamma^{\mu}_{\sigma\nu}$ are the Christoffel symbols, also called affine connections, defined as

$$\Gamma^i_{kl} = \frac{1}{2} g^{im} \left(\partial_l g_{mk} + \partial_k g_{ml} - \partial_m g_{kl} \right) \tag{3}$$

where $g_{\mu\nu}$ is the spacetime metric. Thus resulting in the relation

$$R_{\mu\nu} = \partial_\alpha \Gamma^\alpha_{\mu\nu} - \partial_\nu \Gamma^\alpha_{\mu\alpha} + \Gamma^\alpha_{\mu\nu} \Gamma^\beta_{\alpha\beta} - \Gamma^\beta_{\mu\alpha} \Gamma^\alpha_{\nu\beta} \tag{4}$$

- $R = g^{\mu\nu} R_{\mu\nu}$ is the Ricci curvature scalar, which is a contraction of the Ricci tensor
- G is the Newton's gravitational constant
- c is the speed of light in vacuo
- $T_{\mu\nu}$ is the stress-energy-momentum tensor, satisfying the conservation rule

  $$\nabla_\nu T^{\mu\nu} \equiv \partial_\nu T^{\mu\nu} + \Gamma^\mu_{\sigma\nu} T^{\sigma\nu} + \Gamma^\nu_{\sigma\nu} T^{\mu\sigma} = 0$$

  where ∇_ν is called the covariant derivative

2.2 Linearized field equation: pertubative approach

In this section we present the approximate solution of the Einstein equation following a perturbative approach on the flat spacetime, showing that a perturbation of the flat metric propagates as a wave [50].

In the perturbative approach, the metric is written as a perturbation of an exact solution of the Einstein's field equation (equation 1)

$$g^{\text{total}}_{\mu\nu} = g^{\text{exact}}_{\mu\nu} + g^{\text{perturbation}}_{\mu\nu}$$

where we assume a small perturbation

$$|g^{\text{perturbation}}_{\mu\nu}| \ll |g^{\text{exact}}_{\mu\nu}|$$

caused by a source described by a stress-energy tensor $T^{\mu\nu}_{\text{perturbation}}$. The total stress-energy tensor is

$$T^{\text{total}}_{\mu\nu} = T^{\text{background geometry}}_{\mu\nu} + T^{\text{perturbation}}_{\mu\nu}$$

As said before, we will focus our attention on perturbations on the flat spacetime, which is the vacuum exact solution. Let us consider the flat spacetime described by the metric tensor $\eta_{\mu\nu}$ and a small perturbation $h_{\mu\nu}$, such that the resulting metric can be written as

$$g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu} \qquad |h_{\mu\nu}| \ll 1 \tag{5}$$

The affine connection (equation 3) computed on this metric give

$$\Gamma^{\lambda}_{\mu\nu}(g) = \Gamma^{\lambda}_{\mu\nu}(\eta + h) = \cancel{\Gamma^{\lambda}_{\mu\nu}(\eta)} + \Gamma^{\lambda}_{\mu\nu}(h) \tag{6}$$

$$\Gamma^{\lambda}_{\mu\nu}(h) = \frac{1}{2}\eta^{\lambda\rho}\left(\partial_\mu h_{\rho\nu} + \partial_\nu h_{\rho\mu} - \partial_\rho h_{\mu\nu}\right) - \cancel{\frac{1}{2}h^{\lambda\rho}\left(\partial_\mu \eta_{\rho\nu} + \partial_\nu \eta_{\rho\mu} - \partial_\rho \eta_{\mu\nu}\right)} \tag{7}$$

where we have neglected the $\mathcal{O}(h^2)$ terms and all the cancellations are done because $\eta_{\mu\nu}$ is constant and thus its derivative is zero. $\Gamma^{\lambda}_{\mu\nu}(\eta) = 0$ also imply $R_{\mu\nu}(\eta) = 0$ for the Ricci tensor

$$R_{\mu\nu}(g) = R_{\mu\nu}(\eta + h) = \cancel{R_{\mu\nu}(\eta)} + R_{\mu\nu}(h) \tag{8}$$

with

$$R_{\mu\nu}(h) = \partial_\alpha \Gamma^{\alpha}_{\mu\nu}(h) - \partial_\nu \Gamma^{\alpha}_{\mu\alpha}(h) + \cancel{\Gamma^{\alpha}_{\sigma\alpha}(\eta)\Gamma^{\sigma}_{\mu\nu}(h)} + \cancel{\Gamma^{\alpha}_{\sigma\alpha}(h)\Gamma^{\sigma}_{\mu\nu}(\eta)} - \cancel{\Gamma^{\alpha}_{\sigma\nu}(\eta)\Gamma^{\sigma}_{\mu\alpha}(h)} - \cancel{\Gamma^{\alpha}_{\sigma\nu}(h)\Gamma^{\sigma}_{\mu\alpha}(\eta)} \tag{9}$$

and, using equation (6),

$$\partial_\alpha \Gamma^{\alpha}_{\mu\nu} - \partial_\nu \Gamma^{\alpha}_{\mu\alpha} = \frac{1}{2}\left(-\eta^{\alpha\beta}\partial_\alpha\partial_\beta h_{\mu\nu} - \left(\partial_\lambda\partial_\mu h^{\lambda}_{\nu} + \partial_\lambda\partial_\nu h^{\lambda}_{\mu} - \partial_\mu\partial_\nu h^{\lambda}_{\lambda}\right)\right) \tag{10}$$

where we neglected the $\mathcal{O}(h^2)$ terms.

The Einstein's equation is re-written in the form

$$R_{\mu\nu} = \frac{8\pi G}{c^4}\left(T_{\mu\nu} - \frac{1}{2}g_{\mu\nu}T^{\lambda}_{\lambda}\right) \tag{11}$$

using equations (8, 9, 10) we obtain

$$\frac{1}{2}\left(-\eta^{\alpha\beta}\partial_\alpha\partial_\beta h_{\mu\nu} - \left(\partial_\lambda\partial_\mu h^{\lambda}_{\nu} + \partial_\lambda\partial_\nu h^{\lambda}_{\mu} - \partial_\mu\partial_\nu h^{\lambda}_{\lambda}\right)\right) =$$
$$= \frac{8\pi G}{c^4}\left(T^{\text{perturbation}}_{\mu\nu} - \frac{1}{2}g_{\mu\nu}T^{\text{perturbation}\lambda}_{\lambda}\right)$$

being $T^{\text{unperturbed}}_{\mu\nu}$, by definition, related only to $R_{\mu\nu}(\eta)$, and η the unperturbed solution of the field equation.

2.3 Using the harmonic gauge: a wave equation

The form of the Einstein's equation we derived in the latest formula is not uniquely determined: we need to make a gauge fixing. In order to simplify the equation, it is convenient to choose a coordinate system which satisfies the harmonic gauge condition

$$g_{\mu\nu}\Gamma^{\lambda}_{\mu\nu} = 0 \tag{12}$$

which, using equation (7), is here equivalent to

$$\eta^{\lambda\kappa}\left(\partial_\mu h^\mu_\nu - \frac{1}{2}\partial_\kappa h^\nu_\nu\right) = 0$$

Imposing this gauge condition, the Einstein's equation (11) reduces to a wave equation

$$\eta_{\mu\nu}\partial^\mu\partial^\nu h_{\mu\nu} = -\frac{16\pi G}{c^4}\left(T^{\text{perturbation}}_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu}T^{\text{perturbation}\ \lambda}_{\lambda}\right) \tag{13}$$

where

$$\eta_{\mu\nu}\partial^\mu\partial^\nu = -\frac{1}{c^2}\partial_t^2 + \nabla^2 \equiv \Box_{\text{flat}}$$

is the D'Alambertian operator on the flat spacetime.

The wave equation (13) is linear in $h_{\mu\nu}$: it describes the propagation of gravitational waves in the flat background, with the waves obeying the superposition principle. As previously said, this linear approximation holds for the very weak waves we observe on Earth: the oscillation modes are independent and these waves can thus be Fourier decomposed (even if their waveform originated in the highly nonlinear regime, for example in the merging of two black holes).

The equation (13) can be further simplified defining

$$\bar{h}_{\mu\nu} \equiv h_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu}h^\lambda_\lambda$$

so that the equation is written in a more compact form

$$\Box_{\text{flat}}\bar{h}_{\mu\nu} = -\frac{16\pi G}{c^4}T^{\text{perturbation}}_{\mu\nu}$$

which, far from the source (in vacuo) become

$$\Box_{\text{flat}}\bar{h}_{\mu\nu} = 0 \tag{14}$$

with the gauge condition

$$\partial_\mu \bar{h}^\mu_\nu = 0$$

In conclusion, Einstein's theory predicts that a perturbation of a flat spacetime will propagate as a wave traveling at the speed of light.

2.4 The plane gravitational waves

The simplest solution to the wave equation in vacuo (equation 14) is a monochromatic plane wave

$$\bar{h}_{\mu\nu} = \Re\big(A_{\mu\nu} e^{ik_\alpha x^\alpha}\big) \tag{15}$$

where $A_{\mu\nu}$ is the polarization tensor (the wave amplitude) and k_α is the wave vector (in abstract index notation [51]), which must be of null type

$$\eta^{\alpha\beta} k_\alpha k_\beta = 0$$

to fully satisfy the wave equation.

Moreover, the harmonic gauge condition is

$$\partial_\mu \bar{h}^\mu_\nu = \eta^{\mu\alpha} \partial_\mu \bar{h}_{\alpha\nu} = 0$$

and, when applied to equation (15), implies

$$\eta^{\mu\alpha} A_{\alpha\nu} k_\mu = k_\mu A^\mu_\nu = 0$$

that means that, for plane waves, the wave vector is orthogonal to the polarization tensor.

If we call ω the angular frequency of the wave, by convention we have

$$k_\alpha = \left(\frac{\omega}{c}, \mathbf{k}\right)$$

with the relation $\omega^2 = c^2 \mathbf{k}^2$ implied by the null-type of the wave vector k_α.

2.5 The TT-gauge: two wave polarizations

Every massless gauge vector boson in the Standard Model must have only two degrees of freedom: the two states of helicity [52]. Similarly, we expect the graviton, the hypothetical elementary particle propagating the gravitational interaction, to have also just two helicity degrees of freedom so that the resulting macroscopic coherent wave should have just two polarization states.

Let's do some calculation to confirm this statement: let us compute how many of the ten components of the symmetric tensor $h_{\mu\nu}$ are independent. We consider a plane wave propagating in the x-direction. The wave equation (14) thus simplify as

$$\left(-\frac{1}{c^2}\partial_t^2 + \partial_x^2\right)\bar{h}^\mu_\nu = 0$$

where $x^0 \equiv t$ and $x^1 \equiv x$, and the harmonic gauge condition

$$\partial_\mu \bar{h}^\mu_\nu = 0$$

The last equation imposes four independent constraints, reducing the number of degrees of freedom from ten to six. For our x-propagating wave we can impose

$$\bar{h}^t_t = \bar{h}^x_t \qquad \bar{h}^t_y = \bar{h}^x_y$$

$$\bar{h}^t_x = \bar{h}^x_x \qquad \bar{h}^t_z = \bar{h}^x_z$$

We now observe that there isn't a complete gauge fixing yet: alongside the harmonic gauge, we can still have a new wave solution $\bar{h}'_{\mu\nu}$

$$\Box_{\text{flat}}\bar{h}'_{\mu\nu} = 0$$

if we have defined

$$h'_{\mu\nu} = h_{\mu\nu} - \partial_\mu\varepsilon_\nu - \partial_\nu\varepsilon_\mu$$

where ε^μ comes from an infinitesimal coordinate transformation

$$x'^\mu = x^\mu + \varepsilon^\mu$$

and satisfy the wave equation itself

$$\Box_{\text{flat}}\varepsilon^\mu = 0$$

The four functions ε^μ further reduce the number of degrees of freedom from six to two, choosing them appropriately in order to impose

$$\bar{h}^t_x = 0 \qquad \bar{h}^t_y = 0$$

$$\bar{h}^t_z = 0 \qquad \bar{h}^y_y + \bar{h}^z_z = 0$$

The two remaining non-vanishing components are thus $\bar{h}^z_y$ and $\bar{h}^y_y - \bar{h}^z_z$. This is called the TT-gauge (transverse traceless), where the components of the metric tensor h^μ_ν are different from zero only on the plane orthogonal to the direction of the propagation (the transverse plane) and where h^μ_ν is traceless and thus coincide with $\bar{h}^\mu_\nu$. So, at the end, we have

$$h_{\mu\nu} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & h_{yy} & h_{yz} \\ 0 & 0 & h_{yz} & -h_{yy} \end{pmatrix}$$

So we confirmed that a gravitational wave has only two degrees of freedom, corresponding to the two possible polarizations of the wave (named + and $\times$ due to their effect on a ring of small test masses).

3 Gravitational-wave astronomy and multi-messenger astronomy

3.1 GW150914: The dawn of gravitational-wave astronomy

The new era of gravitational-wave astronomy has begun on September 14 2015, when the two detectors of the Advanced Laser Interferometer Gravitational-Wave Observatory (Advanced LIGO) [1] simultaneously recorded a transient gravitational-wave signal, named GW150914 [2]. This was the first direct detection of gravitational waves and the discovery was awarded with the 2017 Nobel prize in Physics [6].

The GW150914 signal, discovered jointly by the LIGO Scientific Collaboration and the Virgo Collaboration, originated from a binary black hole inspiral and merger. Later on, other observations of binary black hole mergers have followed [3, 4, 5], using the data taken jointly by a network of three ground-based interferometers: LIGO Hanford (Washington, USA), LIGO Livingston (Louisiana, USA) and Virgo (Italy).

Gravitational wave detections will gradually become routine in the near future, as the Advanced LIGO and Advanced Virgo detectors will increase their sensitivity and when other interferometers (currently under construction) will join the existing network [38].

Gravitational-wave astronomy will allow to test Einstein's general theory of relativity in the strong-field regime, study the details of the dynamic of black holes and of compact objects like neutron stars, shed some light on the nature of dark matter and deepen our knowledge on the big bang and the origin of the universe. The scientific results will be countless and incredibly precious, opening a new observational window on the universe.

3.2 GW170817: The new era of multi-messenger astronomy

On August 17 2017 the Advanced LIGO/Virgo detectors made their first observation of a binary neutron star inspiral [7]. The event, named GW170817, is the loudest, closest and most precisely localized gravitational-wave signal to date. The coalescence was followed by a short γ-ray burst, detected initially by the Fermi Gamma-ray Space Telescope and by INTEGRAL [8, 9].

This is the first gravitational-wave event observed together with its electromagnetic counterpart. The precise localization of the source, done thanks to the operation of the third gravitational-wave antenna (Advanced Virgo), permitted to follow the source evolution through the whole electromagnetic spectrum, with more than 70 observatories, both on Earth and Space, involved in the search campaign. The electromagnetic follow-up further restricted the localization of the source in the proximity of the galaxy NGC 4993, near the southern end of the Hydra constellation.

The joint observation allowed many scientific results to be inferred from the collected data, providing insight into the astrophysics of compact binary systems and short γ-ray bursts, as well as probing dense matter under extreme conditions, or making independent tests of cosmology and the theory of gravitation itself [8, 9]. GW170817 thus marks the beginning of the new era of *multi-messenger astronomy*.

3.3 The importance of speed in multi-messenger observational campaigns

Online or low-latency pipelines are essential to trigger and coordinate multi-spectrum concurrent observations. The fast exchange of information between different collaborations and detection infrastructures is crucial, as it was demonstrated with the previously-mentioned discovery of the binary neutron star signal.

The faster, the better: many efforts have been done in this direction, with lots of computing resources allocated for online analysis at LIGO/Virgo data centers [39].

To further take advantage of present technologies, the software stack is currently being optimized, ported or rewritten in order to exploit modern computing hardware, such as GPUs or massively distributed heterogeneous (CPU/GPU) infrastructures [10, 22].

3.4 Artificial Intelligence and Computer Vision

Another different approach to address the low-latency problem can come from the field of Artificial Intelligence (AI).

In recent years, thanks to the use of Artificial Neural Networks (ANN) with deep architectures, the field of Deep Learning is born. Starting from 2012 [40], thanks to some noticeable algorithmic and hardware improvements, Deep Neural Networks (DNN) were able to outperform all the other existing Machine Learning techniques in areas such as image recognition or natural language processing.

Artificial intelligence is now widespread, installed in every smartphone of the planet. This revamped interest, originally capitalized by the industrial sector, is now fastly reaching the academic research in applied and fundamental sciences [41].

In the context of gravitational-waves, some algorithms for the analysis of raw data in the time domain are already under testing [11, 12]. Our proposed method does also make use of deep learning: we developed and trained a Convolutional Neural Network (CNN) that simultaneously analyzes the data taken by the three detectors to search for coincident gravitational-wave signals. Convolutional networks have proven to be very effective in the field of computer vision and pattern recognition: that's why our analysis will focus on structures in the time-frequency plane.

3.5 Overview of possible post-merger sources

After the GW170817 binary neutron star merger, a compact remnant is left over: its nature depends primarily on the masses of the inspiralling objects and on the equation of state of nuclear matter. This post-merger remnant could be either a black hole or a neutron star, with the latter being either long-lived or too massive to be stable, so promptly collapsing to a black hole [54].

Thanks to its relatively close proximity to Earth (with 40Mpc^{+8}_{-14} as 90% credible interval [7]), there have been searches for a gravitational-wave signal originated from the remnant, both under the hypothesis of it being a hypermassive neutron star, with a short signal lasting less than a second, or a supermassive one, with an intermediate-duration signal lasting hundreds of seconds. No statistically significant signals have been found up to now [54].

The merger of two neutron stars can have four possible outcomes:

1. the prompt formation of a black hole
2. the formation of a hypermassive neutron star that collapse in a black hole in less than one second
3. the formation of a supermassive neutron star that collapse to a black hole on a timescale from 10 to 1000 seconds
4. the formation of a stable neutron star

The specific outcome depends on the progenitors' masses and the neutron star equation of state. In this event, the total mass of the two merging neutron stars lies between 2.73 and 3.29 solar masses [7]. The search focuses on the last two scenarios.

The prompt formation of a black hole is very unlikely to be detectable by our instruments at the current level of sensitivity in the interested frequency band because the quasinormal-mode ringdown signal from a remnant black hole in the given mass range is expected to be around 6 kHz [55, 56], while current detectors are too disturbed at such high frequencies [54].

The emission from the collapse of an hypermassive neutron star cannot be analyzed with the algorithms we rely on, originally developed for the search of continuous waves and heavily based on the Fourier transform, which is not the best decomposition for very short signals that fastly vary in frequency. A hypermassive neutron star has mass greater than the maximum mass of a uniformly rotating star, but is prevented from collapse through being supported by differential rotation and thermal gradients [57]. It can collapse in less than one second after formation due to rapid cooling through neutrino emission and magnetic breaking of the differential rotation [58, 59]

We therefore focus on intermediate and long-duration gravitational-wave signals from a possible neutron star remnant. The intermediate case can arise if the star is lighter but still supermassive, with a mass larger than the maximum for a non-rotating neutron star: it will spin down through electromagnetic and gravitational-wave emission, eventually collapsing to a black hole on an expected timescale from 10 to less than $5 \cdot 10^4$ seconds after merger [60]. Gravitational-wave emission mechanisms in this scenario include magnetic-field-induced ellipticities [61, 62], unstable bar modes [63], and unstable r-modes [64, 65]. No electromagnetic observations rule out this kind of longer-lived post-merger remnant for GW170817 [54] and so it makes sense to search for the emitted long-lived narrowband gravitational waves [17].

The last scenario is the formation of a stable neutron star. Rapidly rotating neutron stars are the most promising sources of continuous-wave gravitational signals in the LIGO and Virgo frequency band. These stars are expected to emit gravitational radiation through a variety of mechanisms, including elastic deformations, magnetic deformations, unstable r-mode oscillations and free precession [17].

In summary, electromagnetic observations do not provide definitive evidence for or against any of the four possible post-merger outcomes of GW170817, motivating various broad search using data-analysis algorithms which are robust to uncertain waveform morphologies. We will focus only on the isolated stable or metastable neutron star scenario. Our analysis aims to explore a range of duration ($\mathcal{O}$(days)) not already covered by other searches [54] using a general-purpose and robust algorithm.

We stress that our algorithm is a toy model: we do not expect to find any signal because the remnant distance is ~ 40 Mpc, while the maximum distance achievable today for these signals is ~ 20 Mpc, in case of optimal matched filter [15].

Our algorithm will start from data perfectly Doppler corrected. Indeed, the possible remnant have known position in the sky, so that we can perform an exact

Doppler correction to the analyzed data. This will be relaxed in the future, because we believe that artificial neural networks properly trained on uncorrected data can provide a good starting point for all-sky “blind” searches: their posterior output can be used as the prior input for the all-sky pipeline.

3.6 An online trigger

The search for the neutron star remnant will be an offline search. Moreover, knowing the exact time and position of the binary neutron star merger, the search will be constrained to a defined time interval and with a precise Doppler correction, taking also into account the right amplitude modulation due to the antenna pattern. This is a very peculiar case and our aim is to develop a search method more general that can be used also when the intrinsic and extrinsic parameters of the source, as its position in the sky, rotation frequency, spindown, etc are unknown. We look for an algorithm that is general-purpose, robust and straightforwardly usable for future events.

Artificial neural networks have a potential to help in this direction, by adopting a hybrid approach: the fast and highly-nonlinear nature of their algorithm can be exploited to provide a rough prior to the succeeding linear follow-up analysis, diminishing by orders of magnitude its computational load.

Our aim is to build a binary classifier (the simplest one) trained to distinguish between the *noise* and *noise+signal* classes in a given little region of the parameter space and then use this classifier to tile all the relevant part of the parameter space, in order to acquire informations about the possible signal.

This task can be completed in few minutes or even tens of seconds, thanks to the speed of the algorithm and the possibility to seamlessly run it on many GPUs in parallel. This makes this classifier an ideal candidate for an online or low-latency trigger, which can fastly and efficiently spot a signal, roughly extract a possible range of parameters and rapidly provide this information to all the electromagnetic partners as well as the succeeding, more accurate, follow-up analysis.

As said before, the algorithm we developed is a proof of concept: a toy model to start merging artificial intelligence and experimental gravitation.

3.7 Highlight of the method

Signals are searched in the time-frequency plane, where three grayscale spectrograms, one for each detector, are colored and stacked together to obtain a single Red-Green-Blue (RGB) image. According to additive color synthesis, coincident signals will appear as white patterns inside the image. This strategy was chosen to exploit the ability of deep convolutional neural networks in the area of computer vision and pattern recognition, due to their capability to effectively and efficiently decompose the visual space in a hierarchical way [70].

We trained our classifier [77] using injected signals on the real detector noise of the recent O2 data, which includes all the long-lasting disturbances and other nonstationarities.

The prediction phase of the algorithm applied to the entire O2 run can be completed in less than 5 minutes on a single GPU. The run lasted for 9 months and we are referring our analysis to the sub-set in the frequency bandwidth 80-120 Hz, sampled at 256 Hz and using a coherence time of 8192 s.

The method is still under development and can be improved in several aspects: we will discuss later all the assumptions and all the approximations done. We will also discuss the potential extensions and generalizations that we can implement in the future. For instance, we are currently investigating methods to generate cleaner time-frequency images, with an *ab initio* enhancement of our signals. We are also starting to implement a classifier that does not need a full Doppler-correction preprocessing.

In the future, we aim to apply our method directly to raw LIGO/Virgo data, thus enabling true real-time multi-scale signal searches.

4 Signal injection

In this section we will review the whole procedure of injecting a software signal into a noisy background. We will start from the time domain, showing how the injected signal is generated and added to a gaussian white noise time series. We will then explain how the Fourier analysis is carried on and how the time-frequency representation is constructed.

At the end of section (5) we will highlight how our procedure is carried on directly in the frequency domain, with the real detector's noise.

4.1 Time-domain signal generation

As we discussed in section (3.5), we are looking for transient signals that last for $\mathcal{O}(\text{days})$.

The parameters of our signals are:

- starting time t_0
- duration ΔT
- shape in the time-frequency plane $f(t)$
- amplitude as a function of time $A(t)$ in the time-frequency plane

We will restrict our analysis to signals having solely a linear spindown and with constant amplitude.

$$f(t) = f_0 + s\,t \qquad s = \frac{df}{dt} \leq 0 \qquad \frac{dA}{dt} = 0 \tag{16}$$

These signals will thus be modeled as time-truncated sinusoids with linearly-decreasing frequency. We will also restrict our analysis to the cleanest frequency band of the real noise, between 80 and 120 Hz, and to a range of spindowns from -10^{-9} to -10^{-8} Hz/s.

The signals parameters thus become:

- starting GPS time t_0
- duration $\Delta T \simeq 2$ days
- starting frequency f_0 between 80 Hz and 120 Hz
- linear spindown s between -10^{-9} Hz/s and -10^{-8} Hz/s
- constant amplitude A

f_0 and s are randomly chosen in the given interval with a uniform probability distribution. The resolutions in the parameter's grid are $df_0 = d\nu = 1/8192$ and $ds = 0.625\ 10^{-9}$ (16 possible values in the range given above).

To describe our frequency-varying truncated sinusoid we will use the real description instead of the complex one, because the data of the interferometers will be real-valued.

$$h_{\text{signal}}(t) = \Re(e^{i\varphi(t)}) = \cos(\varphi(t), \text{mod} 2\pi) \tag{17}$$

$$\varphi(t) = \int_0^t \omega(\tau) d\tau$$

$$\omega(t) = 2\pi f(t)$$

We have a linear (first order) spindown $s = \frac{df}{dt}$, so

$$f(t) = f_0 + s\ t$$

and hence

$$\varphi(t) = \int_0^t 2\pi(f_0 + s\ \tau) d\tau$$

so that the phase $\varphi(t)$ to put in equation (17) become

$$\varphi(t) = 2\pi \left(f_0\ t + \frac{1}{2} s\ t^2 + \text{constant} \right) \tag{18}$$

Now we have a precise formula for express the time-domain waveform $h_{\text{signal}}(t)$.

The time is obviously discretized during the data acquisition: our data have a sampling rate of 256 Hz. These data are downsampled from another dataset at 4096 Hz, which in turns is obtained from another downsampling from the raw data of the three interferometers, typically sampled at 16 kHz.

The time resolution of our data is thus $1/256 \simeq 0.004$ s. This resolution requires the use of the float64 data type because float32 will otherwise induce dangerous truncations in the signal waveform. Those are the only float64 values in the whole analysis; in all the rest of our pipeline we avoided float64 because our final aim is to run the algorithm fully on GPUs. There are GPUs able to efficiently compute in float64, such as the Nvidia Tesla series, but they are much more expensive than the gaming GPUs we want to exploit in the future, such as the Nvidia GeForce series. GPUs were born as graphics units, mainly for gaming: they were originally designed to render colors and shapes on a screen, which do not require an high-precision calculation. So they are historically optimized to do their best with the float32 data type. Moreover, float32 computation is in general twice as fast as float64 computation, so it's often more time-efficient and cost-effective to avoid float64 every time is possible.

4.2 Time-domain white noise generation

As said before, before discussing the case of the real detector's noise, we want to show how the signal injection is carried on in a simpler stationary case, with gaussian white noise.

The discrete-time gaussian white noise on which we want to inject our signal is, again in the real description,

$$h_{\text{white noise}}(t) = \mathcal{N}(\mu = 0, \sigma)$$

where $\mathcal{N}$ indicates the normal probability distribution. The distribution has zero mean; the value of the standard deviation is instead not important at this level, as will be discussed at the end of this section. The only thing worth noticing is that the standard deviation does not change in time.

In the complex description, the time-domain gaussian white noise distribution has a gaussian shape centered in the origin of the complex plane, with the same standard deviation along the real and imaginary axes. We remind the fact that the Fourier transform of a gaussian distribution is again a gaussian distribution, with $\sigma_f = 1/\sigma_t$ [20].

$$b = \text{FFT}(a) \qquad \text{if } \ a \sim \mathcal{N}(0, \sigma) \ \text{ then } \ b \sim \mathcal{N}(0, 1/\sigma)$$

So the amplitude $\mathcal{A}(f) = \mathrm{FFT}_{\tau_{\text{coherence}}}(h_{\text{white noise}}(t))$ is distributed as a 2D symmetric gaussian: both its real $\Re$ and imaginary $\Im$ part are distributed as gaussians with equal variances. The noise's frequency-domain intensity $\mathcal{I}$ (the squared modulus of its amplitude $\mathcal{A}$) is thus the sum of the square of two gaussians

$$\mathcal{I} = |\mathcal{A}|^2 = \Re(\mathcal{A})^2 + \Im(\mathcal{A})^2$$

So the noise's intensity $\mathcal{I}$ is distributed as a χ^2 probability distribution with two degrees of freedom ($k = 2$)

$$\mathcal{I} \sim \chi^2 = \sum_{i=1}^{k} x_i^2 \qquad x \sim \mathcal{N}$$

$$k = 2 \qquad x_1 = \Re(\mathcal{A}) \qquad x_2 = \Im(\mathcal{A})$$

The probability density can be thus written as

$$\rho_k(\mathcal{I} = I) = \frac{1}{2^{k/2}\Gamma(k/2)} I^{k/2-1} e^{-I/2}$$

where

$$\Gamma\left(\frac{k}{2}\right) = \sqrt{\pi}\frac{(k-2)!!}{2^{(k-1)/2}}$$

is the Euler's gamma function and !! indicates the double factorial, recursively defined as

$$n!! = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ n(n-2)!! & \text{if } n \geq 2 \end{cases}$$

The $\rho_k(I)$ with $k = 2$ is a bell-shaped distribution when plotted in the *semilogx* plane, whose expected value (its mean) is k, while its median can be approximated as $\simeq k\big(1 - \frac{2}{9\,k}\big)^3$ [21]. As always, we will only take into account the median, because it's more robust than the mean. With all the required normalizations (that will be discussed in another section), the white noise standard deviation must be chosen so that the corresponding median matches the spectral level of the frequency-band we are analyzing. Given the fact that the observed spectrum between 80 Hz and 120 Hz is almost flat, we can estimate this spectral level by simply computing the median of the spectrum (or the median of its autoregressive estimator).

However, such fine-tuned calculation is useless here: this white noise procedure is just a proof-of-concept example, because we are ultimately interested in the real detector's noise. We will thus only provide relative values, which are more general: the signal amplitude A_{signal} will be hereafter re-expressed in units of the white noise sigma. The *amplitude ratio* R is thus defined as

$$R = A_{\text{signal}}/\sigma_{\text{white noise}} \tag{19}$$

We will also call this quantity the *scale factor*. We can thus use whatever value for the $\sigma_{\text{white noise}}$, so at the moment we will standardize it to 1. Dealing with the real detector's noise will of course require a different approach.

4.3 Chunking

The data time series is obtained from the sum of the discrete-time signal and gaussian white noise, generated as explained in the previous section

```
data[t] = noise[t] + signal[t]
```

where of course the signal waveform is truncated, so it's zero outside the interval $[t_{\text{start}}, t_{\text{end}}]$, which lasts for $\simeq$ 2 days. The total duration of the generated data is $\simeq$ 6 days, with the injected signal ($\sim$ 2 days) occupying approximately the central third of the time series.

Given the fact that we are dealing with long signals, we have to decompose them in the Fourier basis to carry on the analysis in the frequency domain. This will be done using the Fast Fourier Transform (FFT) algorithm, which is a way to efficiently compute the Discrete Fourier Transform by using symmetries in the calculated terms. However, a fully *coherent analysis* (with coherence time t_{FFT} equal to the observation time $t_{\text{observation}} \simeq$ 6 days) is not possible due to signal characteristics such as the Doppler shift and the spin-down [1], other than the big computational cost. So we will split the time series in chunks and perform and *incoherent analysis*. The temporal length of the various chunks is chosen such that

[1]We need to correct our data to take into account both the spin-down and the Doppler shift, because otherwise the spectral power will be broadened in many bins, diluting our signal.

the corresponding frequency resolution is able to contain the frequency shift due to the Doppler effect [18].

$$\text{coherence time} \leq \sqrt{\frac{1}{\text{maximum frequency variation in the time unit}}}$$

$$\text{frequency resolution} = \frac{1}{\text{coherence time}}$$

Doing so, the Doppler shift is not able to move our signal's spectral peak outside its given frequency bin [13].

Every temporal chunk should contain a number of discrete values that is a power of two: in this way the FFT algorithm is much more efficient, due to the higher internal symmetry of the calculation. The chunk length we have chosen to take into account the Doppler effect is $\tau_{\text{coherence}} = 8192$ seconds. Multiplying this number by the sampling rate of 256 Hz we obtain $8192\ \text{s} \times 256\ \text{Hz} = 2097152$ values contained in each chunk.

Moreover, when the FFT is computed for purely real input (as our time series is), the output is Hermitian-symmetric: the negative frequency terms are just the complex conjugates of the corresponding positive-frequency terms, and the negative-frequency terms are therefore redundant. Therefore, we used the optimized `rFFT` function, which does not compute the negative frequency terms, resulting in a faster algorithm. The result will be a *unilateral* spectrum, from 0 to the Nyquist frequency, with the energy doubled (or multiplied by $\sqrt{2}$ in amplitude) with respect to the output of the regular `FFT` function because the negative-frequency amplitudes are superimposed to the positive-frequency ones. With a sampling rate of 256 Hz, the Nyquist frequency is $256/2 = 128$ Hz.

However, there is a little downside of using the `rFFT` function instead of the regular `FFT`: the length of the output array is not a power of 2 as the input. So care must be taken in the divisions during the following tiling, eventually discarding some data at the border.

At the moment, the FFT calculation of each temporal chunk is done in parallel on CPU. In the upcoming future we will implement GPU parallelization, since this should be more than 10 times faster.

4.4 Windowing

The Fourier decomposition is not very efficient when dealing with discontinuous functions. So chunking our temporal data and simply doing a Fourier transform of every chunk is not the best way to proceed, because chunking is equivalent to applying a rectangular window to the data, which is a discontinuous apodization function.

This problem can be mitigated by a proper choice of the window function to apply to the chunked data [66] just before the application of the Fourier transform.

```
FFT(windowing(chunking(temporal_data)))
```

Our desired window function should be continuous (better if also with continuous derivative), starting and ending with zero, and providing an almost-flat part to deal better with frequencies which vary in time.

A possible solution is to use the Tukey window function [67], also named *flat top cosine edge* window, which has reduced side lobs and good peak gain. It has the following structure:

- an ascending cosine for the first quarter

$$w(n) = \frac{1}{2}\left(1 + \cos\left(\pi\left(\frac{2n}{\alpha(N-1)} - 1\right)\right)\right)$$

- a flat part in the second and third quarter

$$w(n) = 1$$

- a descending cosine for the last quarter

$$w(n) = \frac{1}{2}\left(1 + \cos\left(\pi\left(\frac{2n}{\alpha(N-1)} - \frac{2}{\alpha} + 1\right)\right)\right)$$

where N is the number of values in the chunk, n their index. The value usually chosen for the parameter α is 0.5.

The figure (1) shows the Tukey window and its behavior in the frequency domain. It can be regarded as a cosine lobe of width $\alpha N/2$ convolved with a rectangular window of width $(1 - \alpha/2)N$.

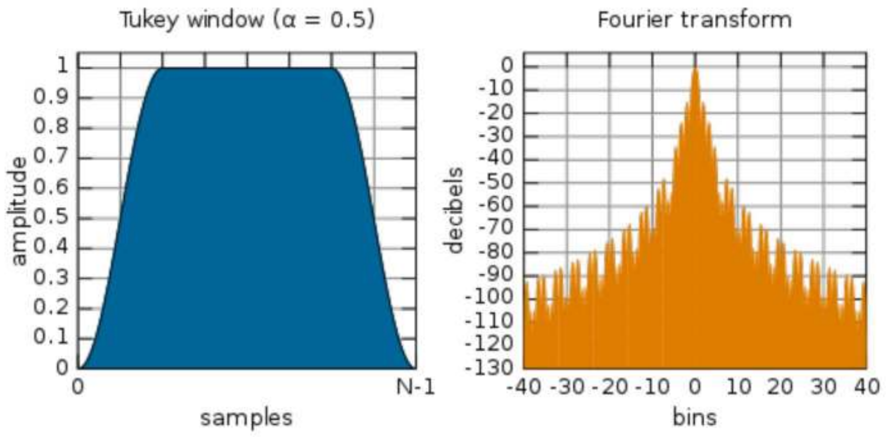


Figure 1: The Tukey window function, with $\alpha = 0.5$ and its Fourier transform.

4.5 Interlaced chunks

To avoid too much power loss at the edge of the window (figure 1), which is equivalent to discarding some data, the chunks are interlaced by half (Welch's method [68]) and a small normalization correction is applied to the computed spectra.

So, with 64 *separated* chunks of 8192 s each we can cover the whole duration of our temporal data, which is $\sim$ 6 days. But, due to the interlacing, we have to double this number, having 128 *interlaced* chunks in total. The Fourier transform of every interlaced chunk, together with its spectrum, will thus be slightly correlated to the one preceding and the one following.

4.6 Building the spectrogram

Once we have computed the Fourier transform of all the windowed interlaced time chunks, we have to build all the corresponding spectra.

The unilateral spectrum is obtained from the unilateral amplitude

$$S(f) = N|A(f)|^2$$

where N is obtained with the following code

```
N = square( sqrt(2) * normalization_factor *
            window_normalization * sqrt(1 - percentage_of_zeros) )
```

where the various quantities are already computed and stored in the .SFDB09 file, as will be discussed in section (5.1).

The frequency resolution of the spectrum is the inverse of the coherence time used in the Fourier transform

$$\delta f = \frac{1}{t_{\mathrm{FFT}}} \tag{20}$$

so it is $1/8192 \simeq 10^{-4}$ Hz.

So we will produce $N_{\mathrm{chunks}} = 128$ spectra and we will stack them to produce a *spectrogram*, which is a time-frequency representation of our data. The algorithm should be

```
spectrogram = stack(normalize(square(abs(rFFT(chunks)))))
```

Given the fact that the chunks of the time series are interlaced by half, we will have twice the number of time bins (as compared to the regular t_{FFT} chunking with flat windowing).

In figure (2) there is an example of a spectrogram constructed from the real O2 data taken by the LIGO Hanford detector. It can be seen as a collection of vertical lines stacked together. Every vertical line is one pixel wide and represents the spectrum of one temporal chunk. So the number of chunks used is the same as the number of pixels along the time axis, which is 128. The original 64 chunks are interlaced as previously described. The frequency axis is here is truncated at a small frequency interval; this frequency interval is represented by 256 frequency pixels. Each frequency pixel represents a frequency interval equal to the frequency resolution in equation (20).

The black vertical lines represent the time chunk whose spectra were discarded, as will be explained in section (5.3). As can be seen, they are a non-negligible fraction of the original number of spectra. There are a lot of temporal holes in the real O2 data, where the data were disturbed or not science-quality of if the detector was off for technical maintenance. The grayscale pixel values (sidebar in

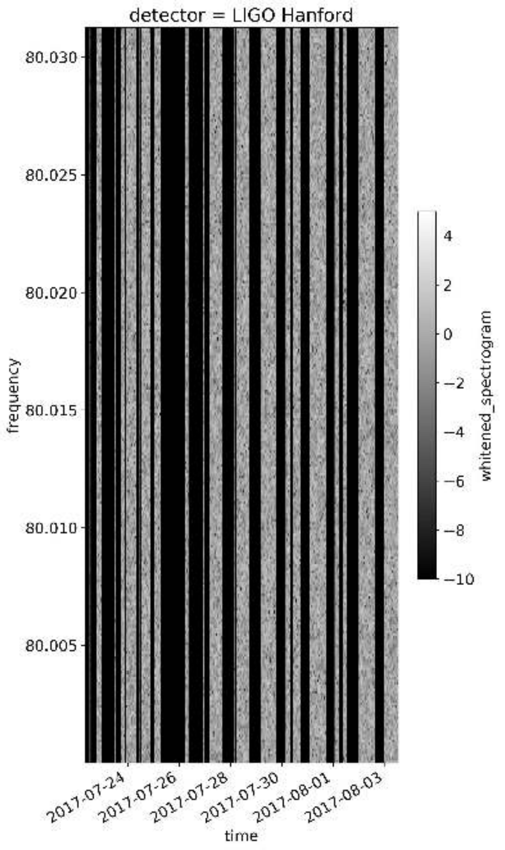


Figure 2: A spectrogram constructed from the real O2 data taken by the LIGO Hanford detector. The total time is $\sim$ 6 days and the total frequency range is $\sim$ 0.03 Hz. The gray values represent the logarithm of the whitened power spectrum.

figure (2)) are the logarithm of the whitened spectra, defined in the next section. The logarithm is used for visualization purpose only.

4.7 Whitening

A whitened spectrum is a spectrum which is transformed to become flat in frequency: white. This transformation consists in the division of the spectrum by an average estimation of itself, so that the resulting expected value is 1 for every frequency. When every unilateral spectrum relative to every temporal chunk of data is whitened we obtain a whitened spectrogram.

In the following, we will use the autoregressive spectrum as our average estimator: it is a sort of running average that is able to follow the nonstationarities of the noise, excluding the narrow frequency peaks from the running average because they can be potential signals. Its precise algorithmic definition can be found in [13]. We did not compute ourself the autoregressive spectrum of every spectrum, because it can be found in the corresponding .SFDB09 file.

A visual example of an autoregressive spectrum is shown in figure (3). With this whitening procedure, we will eliminate (normalize) all the broad frequency peaks (which are wider than a couple of frequency bins) and we will standardize the noise's base value, leveling the hills in the raw spectrum (figure 4). Only the narrower peaks are left, which are the possible signal candidates.

It is however important to notice that this whitening procedure only "whiten" the expected value of the spectrum and *not* its variance or any other higher-order moments of its distribution. The variance of a nonstationary noise is also nonstationary. There is indeed a relic higher-order effect in the data due to the original nonstationary colored noise. Thus, the real data won't be perfectly distributed as a $\chi^2_{k=2}$ (figure 6), as the gaussian white noise should be. Thus, we think it's worth to investigate other whitening strategies to mitigate this problem.

4.8 Signal visibility

The signal we injected has amplitude ratio (or scale factor) $R = 0.004$. This means that the signal amplitude is much smaller than the noise's standard deviation in the time domain. Figure (5) shows the injected signal (zoomed 50x to make it visible) and the white noise background (not zoomed) to which it will be added.

We have chosen the value of $R = 0.004$ because it is the last one at which the signal can be clearly seen by eye in the spectrogram (figure 7). Having such a small signal in the time domain means that in the frequency domain the signal is just above the median of the noise spectrum (figure 6). In other words, the signal is just above the sensitivity curve of the detector, which is the intrinsic limit at which a signal can be seen. A signal below the sensitivity curve will be literally eaten by the surrounding noise (at least if some kind of ad-hoc preprocessing isn't

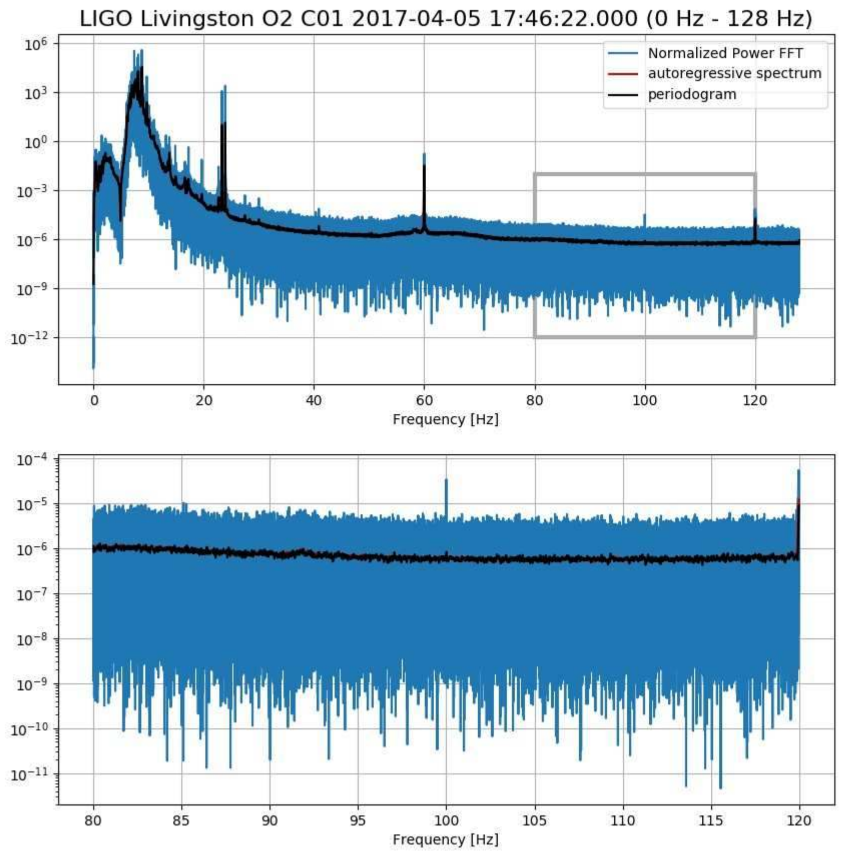


Figure 3: An example of autoregressive spectrum, here perfectly coincident with the periodogram. The peak at 120 Hz is not narrow enough to be excluded from the autoregressive spectrum (as the peak at 100 Hz is) and it will thus be leveled away in the whitened procedure.

used, such as matched filters or other denoising procedures).

In figure (6) we have plotted the logarithmic histogram of the *real* noise of the three detectors in the 80-120 frequency band (almost flat sensitivity curve). A signal starts to become invisible when it is getting closer (from the right) to the median value of the histogram.

To give a feeling of this mechanism, we will show the spectrograms at different

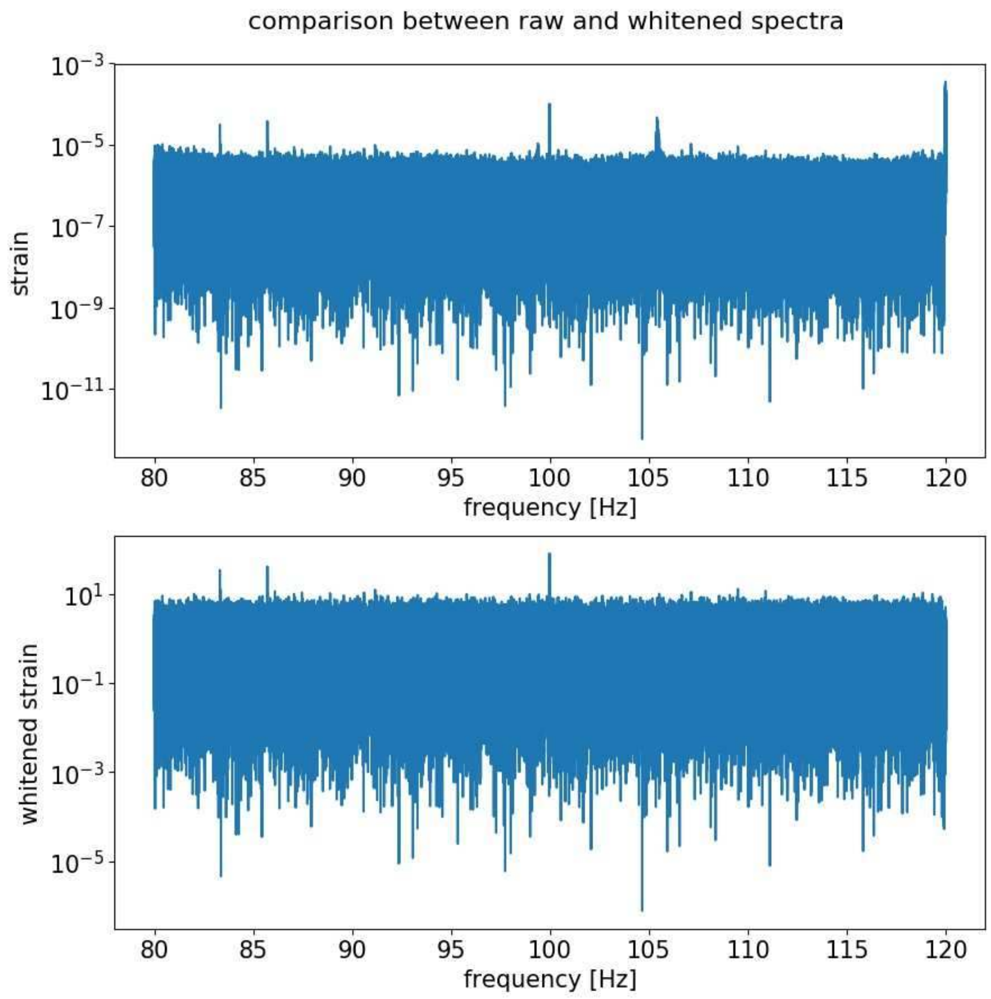


Figure 4: The effect of the whitening procedure. The broad peaks around 105 Hz and 120 Hz disappeared.

values of R, to understand when the signal can be clearly seen in the data (figure 7). At R = 0.005 the injected signal is still well visible. We believe that R = 0.004 is the acceptable limit of visibility: the signal is starting to vanish but can still be clearly distinguished from the noise structures. At R = 0.003 we can barely see something and it's not easy to distinguish the signal from accidental linear structures of the noise (which are very common with the case of real detector's noise). At R = 0.002 the injected signal is completely invisible, surpassed by the surrounding noise.

To summarize, R = 0.004 is the last level at which a signal can be safely recognized by eye in a white noise background and thus it will also be close to the ultimate level achievable with an artificial neural network. Anyway, this is only a useful reference, since our algorithm will simultaneously analyze data from the three detectors[2], with all the structures and nonstationarites that can be found in the real noise (see section 5.4).

[2]In the future, we will repeat these numerical trials with a 3-channel RGB white noise, to see if the ultimate visibility level can be significantly lowered due to the exploitation of the coincidences.

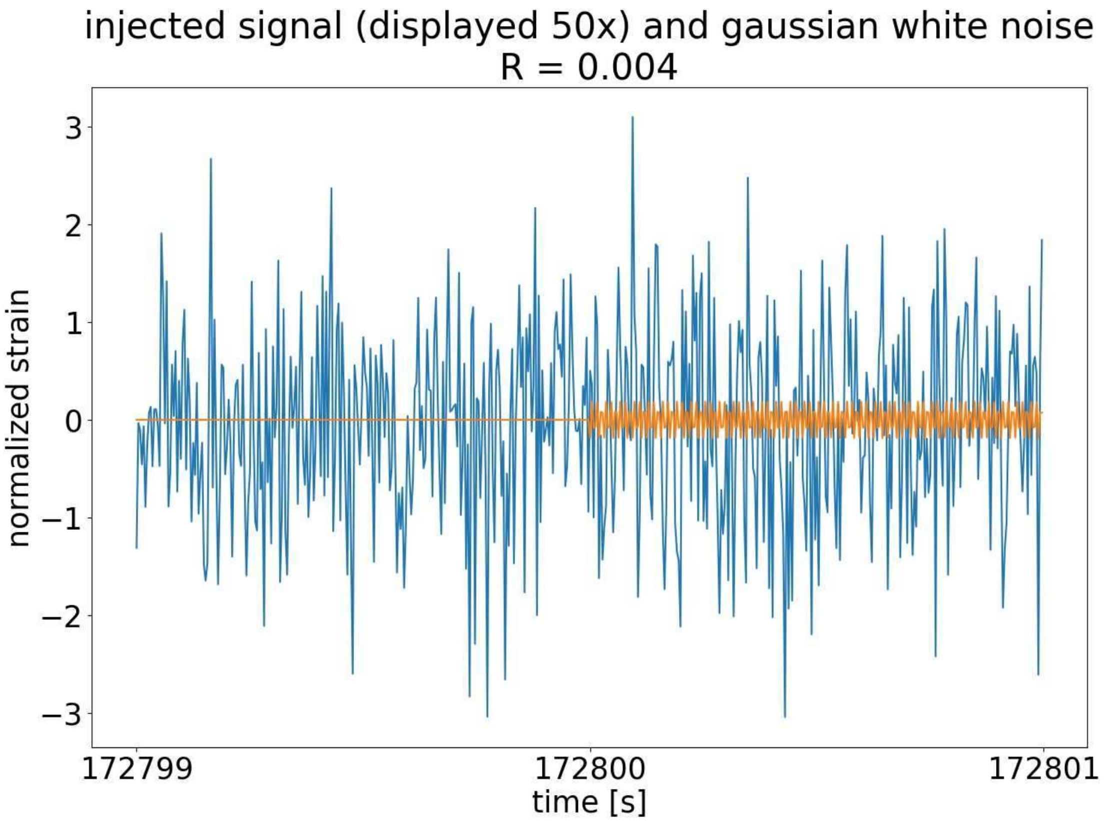


Figure 5: Injected signal (displayed 50x) and gaussian white noise (not zoomed). The plot shows only the two seconds around the starting of the signal (at GPS time 172800). The whole signal lasts for ~ 2 days, so it's not completely shown in the picture. The signal amplitude (here zoomed 50x for visualization purposes) is 0.004 time smaller than the white noise standard deviation, here normalized to 1. The white noise strain $h_N(t)$ is gaussianely distributed around 0.

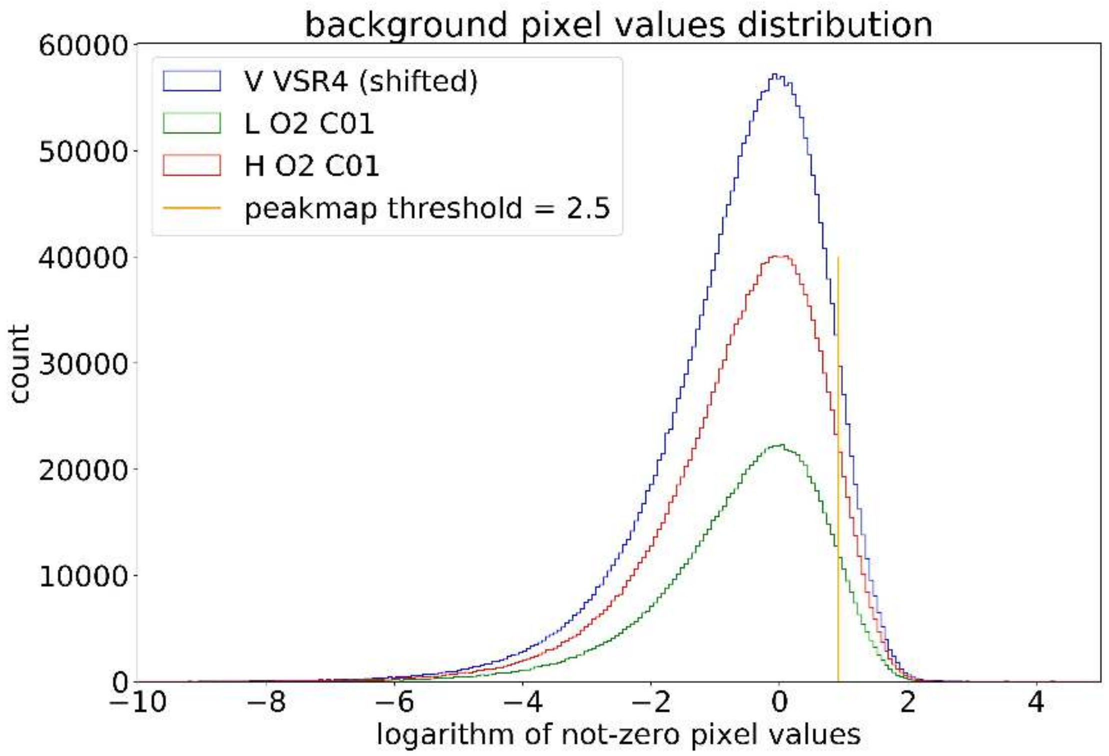
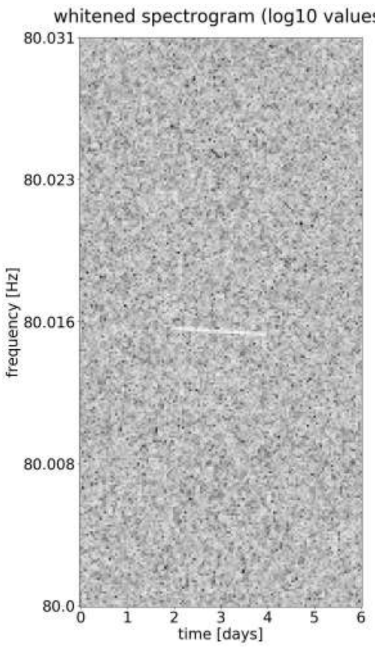
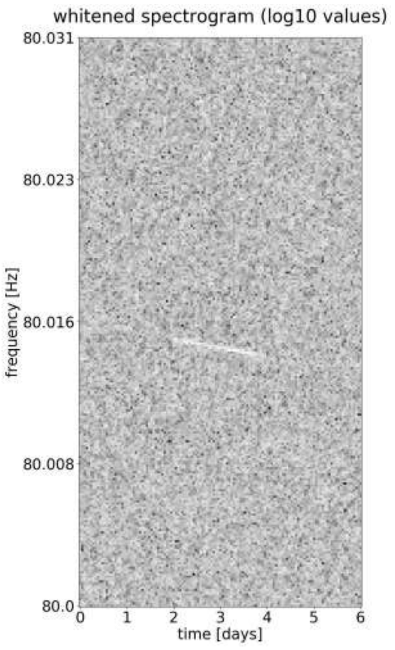


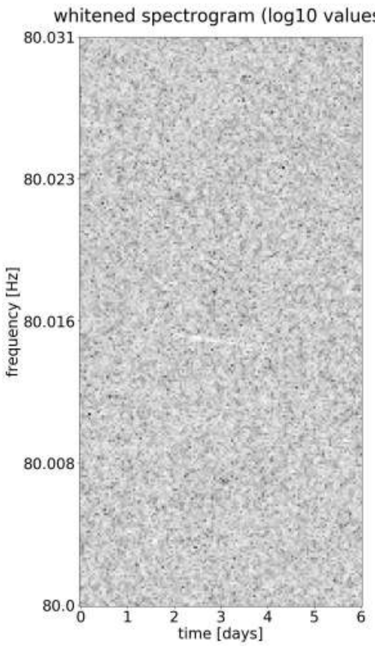
Figure 6: Logarithmic histograms of the whitened real noise of the three detectors in the 80-120 frequency band. The LIGO Livingston is higher than LIGO Hanford because it has fewer holes in its data (better duty cycle). Each whitened spectrum has median 1, so their logarithm has median 0.



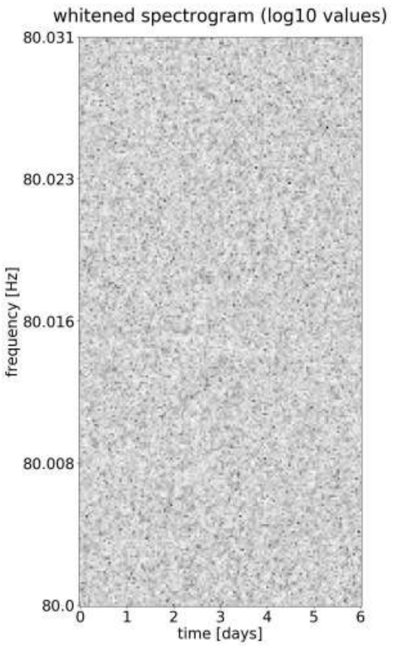
(a) $R = 0.005$



(b) $R = 0.004$



(c) $R = 0.003$



(d) $R = 0.002$

Figure 7: Different spectrograms with signals of different decreasing amplitude ratios. At R = 0.002 the signal is completely eaten by the surrounding white noise, meaning that it has passed below the median level of the noise (figure 6), corresponding to the sensitivity curve of the detector, so it is no more visible.

5 Data preprocessing

5.1 Conversion from .SFDB09 to .mat

Our work will not start from raw data, but instead from a collection of Fourier transforms of predefined coherence time. Those pre-elaborated data are saved in a .SFDB09 file.

The SFDB09 file format (Short FFT DataBase, 2009 specification) was developed inside the *Virgo Rome* group and the functions required to handle this type of file are part of the Snag package [26]. Snag is a C/Matlab data analysis toolbox oriented to gravitational-wave antenna data; its actual version is v2, released on 12 May 2017.

We want to use Python as the programming language of our analysis. Since the SFDB is a custom file format with no Python bindings, we decided to convert all the data to the Matlab mat format, which is a common and rather standard file format with bindings available also for Python. We later used the `scipy` module to import the data inside Pyhton, but at the moment this module cannot load the most recent mat v7.3 format, which is based on the hdf5 specifications [27]. So we used the less recent mat v5.0 format.

Given the complexity of the SFDB data format, reading a whole SFDB file is not a straightforward operation: we had to partially rewrite an already-existing Matlab script, named `read_SFDB.m`, in order to iteratively call the predefined function `read_block_09` which, as its name suggests, reads only a single block inside the file. We cleaned the script code to make it ready for the official future insertion in the Snag package. The new script is able to automatically convert all the files in a given data folder (and its subfolders), but the conversion is still sequential, so not made in parallel.

Every .mat file contains the following variables, extracted from the header and the body of the .SFDB09 file:

- `detector` Could be `Nautilus`, `Virgo`, `LIGO_Hanford` or `LIGO_Livingston`.
- `gps_time` Starting time of the FFT chunk, up to nanosecond precision. The

float32 data type is not sufficient to represent the GPS time with the required precision, because the last two digits are truncated, introducing an error of tens of seconds. So we will use float64 here. When needed, this GPS time is converted to standard human-readable UTC time via the `gps2utc` function of the Snag package. The ISO 8601 compliant date-time format is YYYY-MM-DD HH:MM:SS.sss

- `fft_length` The coherence time used in the analysis. The suggested value is 8192 seconds.
- `starting_fft_sample_index` If the FFT do not starts from frequency 0, it indicates the first frequency index. The index refers to the number of samples, not to frequency (as opposed to `starting_fft_frequency`).
- `unilateral_number_of_samples` Number of samples in half of the frequency band. The unilateral spectrum only covers positive frequencies.
- `reduction_factor` Expresses how much the `autoregressive_spectrum` is subsampled with respect to the FFT. The suggested value is 128, so that the FFT value is averaged for 128 time intervals
- `fft_interlaced` Is `True` if the FFT are interlaced, which means that they are also computed between half intervals, to make the spectrogram smoother.
- `number_of_flags` Number of data labeled with some kind of warning flag (eg: non-science flag)
- `scaling_factor` Define a scaling factor to apply to all data, in order to deal with bigger numbers and gain more numerical stability. The scaling factor used in our analysis was arbitrarily defined to be 10^{-20}. That's why our strain values will differ from the ones usually seen in the literature.
- `mjd_time` The starting time of the FFT chunk, expressed in the Modified Julian Date format.
- `fft_index` The frequency index, not to be confused with the frequency value.

- `window_type` Window type used when computing the FFT. The suggested value is `flat_top_cosine_edge` (figure 1), because the use of `Hamming` is not recommended here due to the presence of the spindown.
- `normalization_factor` Normalization factor for the power spectrum estimated from the square modulus of the FFT due to the data quantity `sqrt(dt/n_fft)`.
- `window_normalization` Corrective factor due to power loss caused by the edges of the Tukey FFT window (figure 1).
- `starting_fft_frequency` The suggested value is 0 Hz.
- `subsampling_time` Sampling time used to obtain a given frequency band, subsampling the data. The Fourier transform of complex values results in a frequency band which also covers negative frequencies, so the physical frequency band is half that frequency band.
- `frequency_resolution` The frequency resolution is the inverse of the coherence time $\Delta\nu = 1/t_{FFT}$
- `velocity` The 3 components of the detector's velocity (in equatorial cartesian coordinates) evaluated at half the FFT time chunk.
- `position` The 3 components of the detector's position (in equatorial cartesian coordinates) evaluated at half the FFT time chunk.
- `length_of_averaged_time_spectrum` Length of the FFT divided in pieces by the `reduction_factor`.
- `number_of_zeros` Number of artificial zeros, used to fill every time hole in the corrupted FFT data (eg: non-science data).
- `endianess` Specify the sequential order in which bytes are arranged into larger numerical values when stored in memory.
- `spare1` Unused variable, defined to make the SFDB09 format future-proof.
- `spare2` Same as `spare1`.

- `spare3` Same as the other `spare*` variables.
- `percentage_of_zeros` Percentage of artificial zeros in every FFT chunk.
- `spare5` Same as the other `spare*` variables.
- `spare6` Same as the other `spare*` variables.
- `scientific_segment` Discontinued variable. Not used anymore.
- `spare9` Same as the other `spare*` variables.
- `periodogram` The periodogram is just a smoothed version of the original spectrum. It can be used in the succeeding whitening phase.
- `autoregressive_spectrum` The autoregressive spectrum is an estimator of the spectrum, constructed with a particular running average algorithm. The procedure is able to follow the nonstationary level of the noise and is tuned to exclude narrow frequency peaks from the running average, because they can be potential signals. Further details on the algorithm can be found in the reference paper [**?**].
- `fft_data` The unilateral amplitudes resulting from the computation of the Fourier transform on the real-valued time series. Being the unilateral amplitude a complex number, its data type can be interpreted as `float32 + i*float32 = complex64`.

Given the fact that the final outputs of the data preprocessing will be images to be processed on a GPU, we have chosen to save data directly in single precision (float32) wherever possible.

Our conversion script finds all .SFDB09 files in the data directory and its subdirectories and converts all those data in .mat format. From now on, the whole data analysis will be carried out using Python.

5.2 Conversion from .mat to .netCDF4

The choice of the file format used for the analysis is crucial because it must fulfill specific needs. The right file format can simplify the managing of the data and their processing.

We will now list all the requirement our file format must satisfy:

- easy to be read from and written by Python. As we said, our analysis will entirely be in Python.
- standardized and widely used. In this way we will have access to many well-tested pre-defined routines, avoiding bugs and saving time for the coding process.
- able to handle a huge amount of data (*big data*).
- binary (not textual), to occupy less space on disc, and preferably compressed, with the compression/decompression operation handled transparently.
- able to seamlessly handle data from multiple little files, merging them properly. This is for two reasons: small files are easier to process with low-memory computers and also because data corruption can always happens (for example due to disk failures or remote copy failures), so it's better to waste a small binary file rather than a huge one.
- *parallel*, because we really need to process data in parallel. This is one of the strongest requirements.
- *out-of-core*, meaning that we should be able to perform operations on datasets bigger than the computer's RAM. In this way we can pass from a *smaller-than-memory* limitation for our dataset to a much wider *smaller-than-disk* limitation. This is another key requirement.
- abstract, meaning that most of the operations must be carried out automatically by the file format or the high-level libraries used to handle it. This will simplify the code, saving time and avoiding bugs.

All these requirement fits with the netCDF4 file format, which is a dialect of the hdf5 file format especially designed for parallelism and out-of-core computation: netCDF4 is widely used in the context of geophysics and meteorology, where the processing of huge amount of data is required. Moreover, when used together with `dask` and `xarray` python libraries, netCDF4 have two other useful features:

- it is symbolic, meaning that all the operations are done in *lazy* mode (instead of the *greedy* mode often used in calculus libraries) and written to a computational graph (direct acyclic graph) which will be evaluated only at the end, after an automatic optimization process which takes into account which operations are called and in which order and how the data are physically arranged on the disk. This is the key of its parallelism and out-of-core nature. A similar approach can be found in the TensorFlow library [80], which we already use in the learning part of our pipeline.
- it allows the definition of a database of multidimensional structures, with all the dimensions cross-linked between objects, allowing us to perform operations on the same axis of different objects and on different axes on the same object. For example, a "spectrogram" object will have three dimensions (or axes): frequency, time and intensity value. We can have an array of many spectrograms to analyze and every spectrogram can come from a different detector. All those dimensions can be cross-linked with the dimensions of other object, such as the "non-science flags", which also have the time and detector dimensions.

  ```
  intensity = spectrograms[index, frequency, time, detector]
  science_mode = flags[index, time, detector]
  ```

  All the objects have to change coherently when an operation is performed on the dimensions of one of them. For example, if we flag a given time-chunk of a given detector as "non-science", the corresponding values of the spectrogram must be automatically excluded, in a coherent way.

The conversion we made from mat format to netCDF4 one is done with a python script. We didn't do this operation in the precedent Matlab script because we wanted to fastly get rid of the sequential nature of the SFDB09 reading function: it's much better to sequentially convert the data, store them on disk and later efficiently process them in parallel (perhaps multiple times, at the developing stage) rather than sequentially read them and sequentially process them. The conversion will be rewritten to work in parallel in the upcoming future.

We will now provide an example of the multidimensional structures stored in the netCDF4 files. With the following single command we can load all the data of the whole O2 run of the LIGO detectors. The data are immediately loaded in lazy mode and out-of-memory, transparently concatenating all the various little files along the "time" axis.

```
>>> xarray.open_mfdataset('./O2/C01/128Hz/LIGO*.netCDF4')
<xarray.Dataset>
Dimensions:                (time: 11304, detector: 2, frequency: 327680)
Coordinates:
  * time                   (time) datetime64[ns] 2016-11-30_15:17:03 ...
  * frequency              (frequency) float32 80.0001 80.0002 80.0003 ...
  * detector               (detector) string 'LIGO Hanford' 'LIGO Livingston'
Data variables:
    spectrogram            (frequency, time, detector) float32 0.0 0.0 ...
    whitened_spectrogram   (frequency, time, detector) float32 0.0 0.0 ...
    locally_science_ready  (time, detector) bool False False False ...
    globally_science_ready  (time) bool False False False False ...
Attributes:
    FFT_length:         8192
    Nyquist_frequency:  128
    start_ISO_time:     2016-11-30 00:29:35.000
    calibration:        C01
    observing_run:      O2
```

Our dataset is thus a collection of multidimensional labeled arrays, which share some dimensions among them.

Before closing this section, we want to highlight another advantage of using high-level libraries and complex-structured file formats. The xarray python module allow us to index the tensors inside a netCDF4 file in two ways: via the regular numerical indices or via their relative labels. For example, the following label selection code allows us to select all the data between 80 and 81 Hz, coming from the LIGO Hanford detector and taken from the beginning of the O2 run till the middle of January. A similar example is done for the index selection.

```
# label selection
dataset.sel(frequency = slice(80, 81), # Hz
            detector = 'LIGO Hanford',
            time = slice('2016-11-30', '2017-01-15'))

# index selection
dataset.isel(frequency = slice(0, 100),
             detector = 0),
             time = slice(0,1000))
```

This is very useful and simplifies the code a lot, also avoiding potential errors. We hope that the netCDF4 format and all the relative libraries and modules will gain more attention in the future for the gravitational-wave analysis because we think that huge and complex pipelines do require easy-to-use and well-designed underlying software to keep the codebase easy-to-read and well-maintainable.

5.3 Spectra construction and selection

The most important object in the dataset is the `whitened_spectrogram`, whose construction was already explained in the section about signal injection (section4).

We will now highlight how the spectra are computed from the data stored in the original .SFDB09 file (later converted in .mat and .netCDF4) and how those spectra are preprocessed and selected to discard the corrupted ones. We start calculating the power spectrum from the FFT data (the `unilateral_complex_amplitude`) using the following formula:

```
total_normalization = sqrt(2)*normalization_factor*
                      window_normalization*sqrt(1 - percentage_of_zeros)
```

```
power_spectrum = square(abs(unilateral_complex_amplitude*total_normalization))
```

Where the factor $\sqrt{2}$ is for the passage from bilateral to unilateral spectrum. Resulting in a spectrum like the one depicted in figure (8).

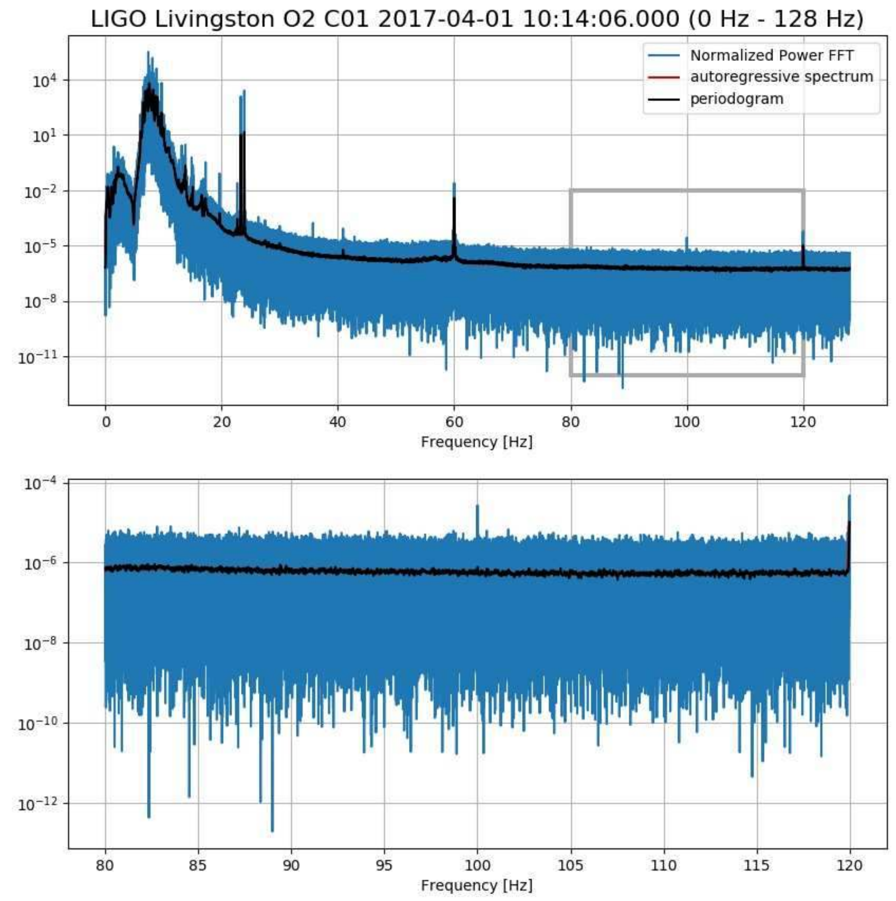
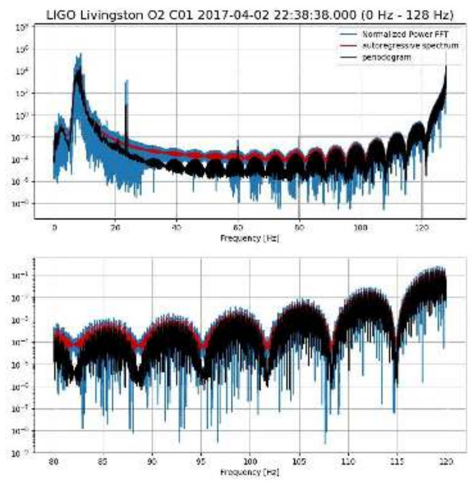


Figure 8: An example of a spectrum calculated for the LIGO Livingston detector, with a zoomed view of the cleanest frequency band. The autoregressive spectrum and the periodogram are perfectly superimposed.

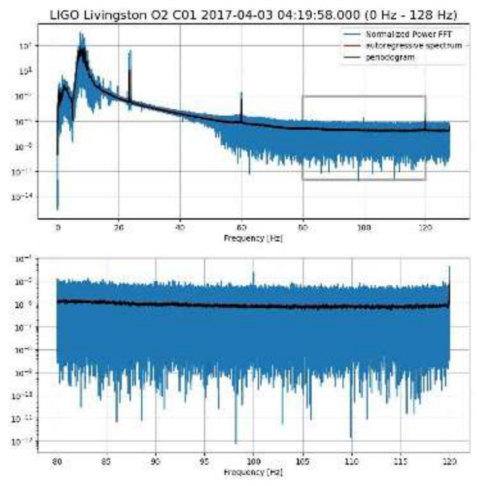
We will restrict our analysis to “science mode” data in the cleanest frequency band, empirically selected roughly in the region from 80 to 120 Hz (figure 8), where the spectrum is almost flat. This will help to have data that are approximately invariant under frequency shifts, which is sometimes desirable, as will be explained later when we will talk about convolutional neural networks.

All our analysis will refer hereafter only to this frequency band between 80 and 120 Hz. In the future we will extend our analysis to the more difficult and disturbed regions of the spectrum.

Some of the spectra were greatly disturbed (figure 9a) and should be discarded.



(a) A wasted spectrum.



(b) A kept spectrum, with local disturbances that do not interest our selected frequency band (80-120 Hz).

Figure 9: Examples of spectra selection

We will not cover in detail the procedure we used to select the spectra since we want to perform this selection with a dedicated neural network in the upcoming future. At this level of development and testing, suffice is to say that the selection was temporarily made with an estimator based mainly on the relative difference between the periodogram and the autoregressive spectrum [13], together with other criteria such as the percentage of holes (missing or discarded data) both in the time-domain and frequency-domain data or the relative difference between the level of strain and the average spectral level of the clean spectra in the whole run (as every circular definition, it requires iteration until convergence).

All those data-driven criteria were applied only in the selected frequency band, to avoid rejections due to disturbances localized in other parts of the spectrum (figure 9b). Even if the median is robust to the presence of outliers, we applied our selection criteria iteratively, because we don't want our estimators (such as the median) to significantly change value due to a not negligible number of corrupted data.

All the discarded spectra were flagged and all the corresponding values were put to zero. When those zeros induced some numerical divergences, for example when computing the logarithm, the results were truncated to the last value available inside the float32 data type. We noticed that a float32 `unilateral_amplitude` is enough to correctly compute the spectrum in the great majority of the cases. However, when the spectrum is corrupted and its values lie outside the usual range by many order of magnitudes, we can encounter the possibility of overflow and truncation errors. Modern software libraries for numerical calculus such as `numpy` can handle these errors seamlessly. However, it's important to exactly know which are all the consequences of GPU computation, with the passage from float64 to float32.

5.4 Noise features

Instead of a white noise, we are using the real data of the whole O2 run. So our injected signals will have a rich background with lots of peculiar features: for example resonances and hardware injections.

The *pulsar3* is a hardware injection placed at 108.857 Hz with linear spindown of -1.46e-17 Hz/s [69], so we should be able to see it in our selected band (80-120 Hz). The injection is present from the start to the end of the run.

The following plot shows the whole O2 run (9 months) for a frequency window of 0.025 Hz: the pulsar3 is clearly visible with its annual and daily Doppler shift. The plot shows the data of the Hanford detector, which is the most stable and clean in this frequency band. Notice the black gaps where the detector was off, out-of-lock or not in science-mode.

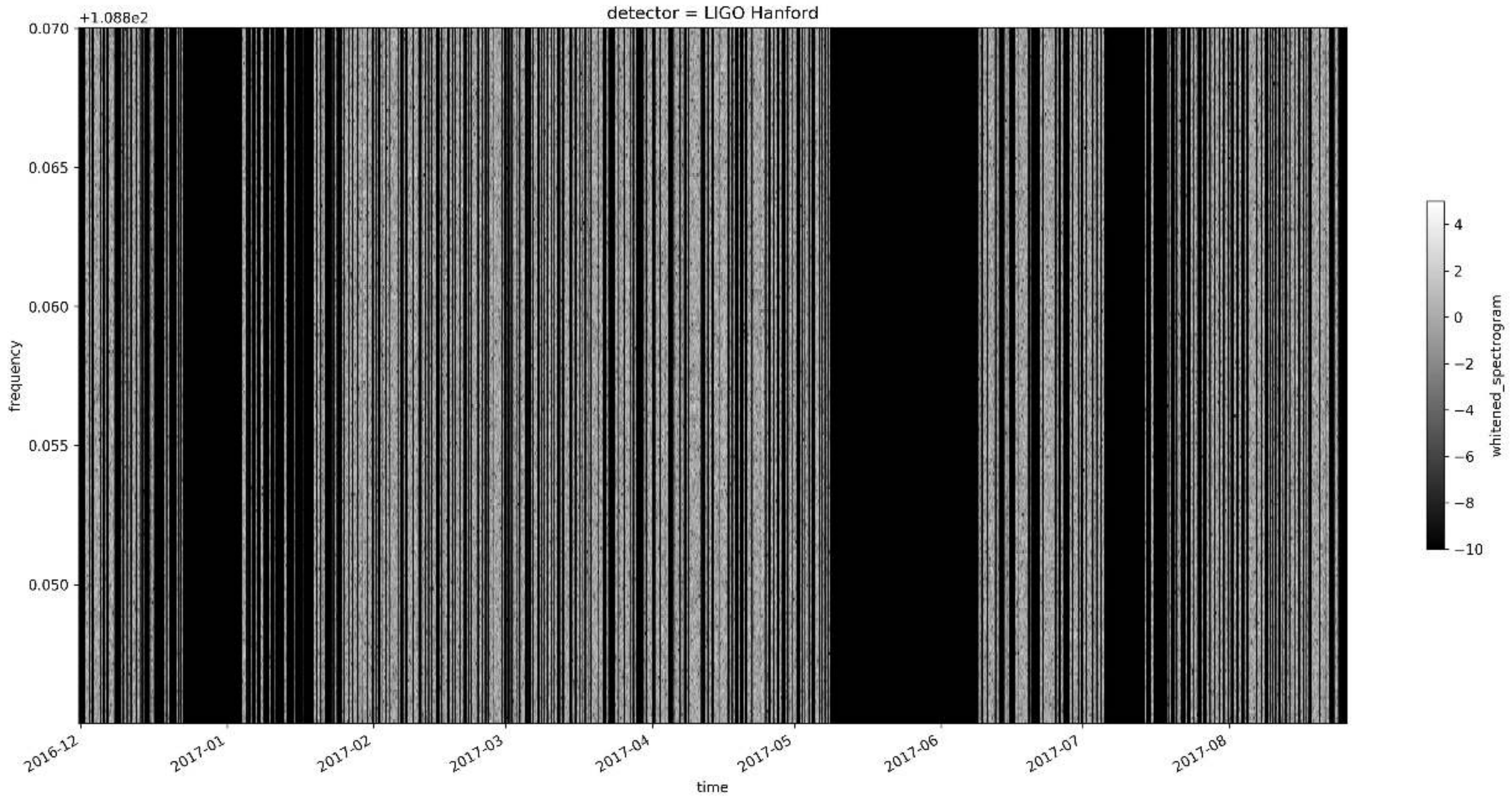


Figure 10: The *pulsar3* hardware injection, pictured over the whole O2 run. The annual Doppler sinusoidal modulation is clearly visible over the 9 month timescale. The underlying daily modulation, which is invisible at this level of zooming, has the effect of broadening the time-frequency pattern, making it blurry and less visible

This is instead how the same hardware injection appears in our images, which are only $\sim$ 6 days long

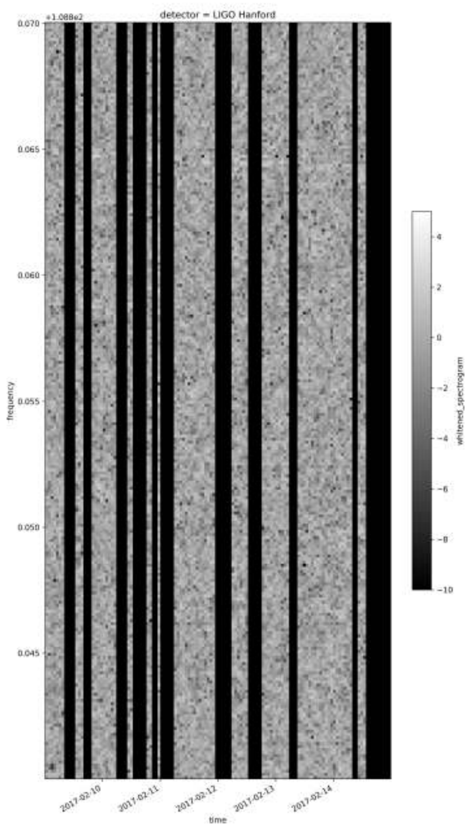


Figure 11: The *pulsar3* hardware injection, framed in only 6 days.

This feature cannot be easily seen (it's it the upper part of the figure) because it is *not* Doppler corrected. This means that the daily modulation is broadening the energy in many bins. That's because the daily Doppler induces a sensible frequency variation inside one FFT length (8192 s) and so the FFT window (flat-top-cosine-edge) will broaden the signal accordingly.

There are also many resonances in the spectrum that are tight enough to survive the whitening phase. See for example the line at 100 Hz in the following plot.

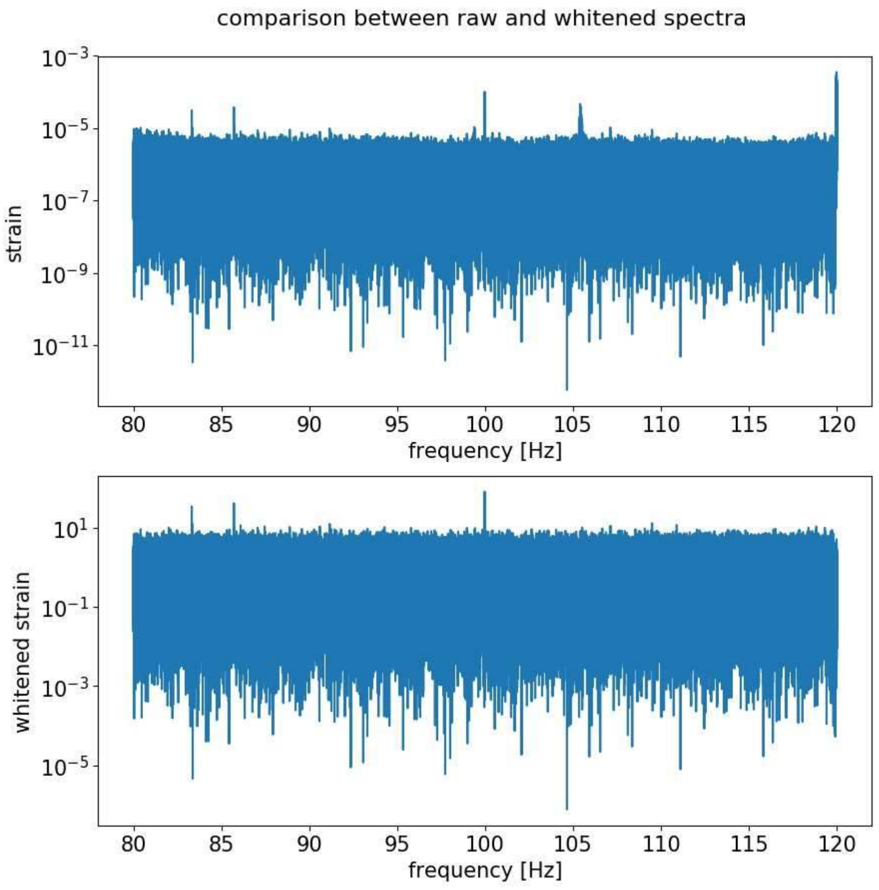


Figure 12: Visual demonstration of some narrow resonances surviving the whitening phase.

This is how this huge line appears in the whitened spectrogram (figure 13). Notice that, because of its high frequency stability, this line does not exhibit frequency-broadening due to the FFT window.

It will be crucial for our artificial neural network to be able to recognize all these various features and don't confuse them with the signals that we will inject.

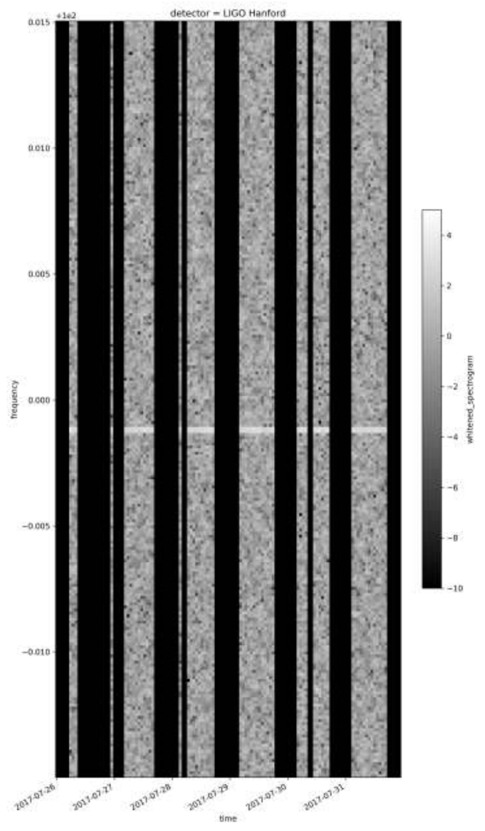


Figure 13: Resonance line present in the data and not canceled by the whitening procedure.

5.5 The dataset loading

Thanks to the netCDF4 file format, we were able to load the dataset in chunks in order to do out-of-memory computation. Each memory chunk is automatically chosen by the software. We loaded the whole 9-month-long O2 dataset for the LIGO Hanford and LIGO Livingston detectors, while we used the old VSR4 dataset for Virgo, because we have not enough data from this detector inside the O2 run. To better resemble the O2 data, we artificially shift the VSR4 starting time and artificially decrease its amplitude, in order to simulate a better sensitivity:

```
fake_O2_spectrogram = VSR4_spectrogram * exp(-4)
```

Moreover, we used the Virgo VSR4 dataset *twice*, in order to have more statistics for the training of our neural network (figure 14).

5.6 Detector's duty cycle

Now we have to quantify the temporal stability of the duty cycle of the three detectors. We thus need to define a temporal scale on which to compute a running average. Given the fact that the images we will analyze cover a period of ~ 6 days, we have chosen to compute the average on a timescale of one week.

For each detector, we have an array called `is_locally_science_ready` which contains the boolean flag associated to every FFT chunk. 0 means that the chunk is corrupted or missing and 1 means that the chunk have passed all the selection criteria. We also have an array called `is_globally_science_ready` which contains the global flags. A global flag is 1 only if *all* the corresponding detector's flags are 1. Those global flags thus indicate when all three detectors were in science mode.

To compute the average on given timescale, we convolved the flag array with a kernel defined as

```
kernel = ones(number_of_time_pixels)
```

where we remind that every time pixel of the spectrogram represents an interlaced chunk of 8192 seconds. The `number_of_time_pixels` was chosen to represent the duration of a week ($\sim$ 150 pixels) and the result of the convolution was finally normalized to 1 by dividing for that number.

We computed this "time stability" indicator for the local flags of the three single detectors and for the global flags of the detector network as a whole. The results are plotted in figure (14): the two technical shutdowns in winter (December-January) and spring (May) can clearly be seen.

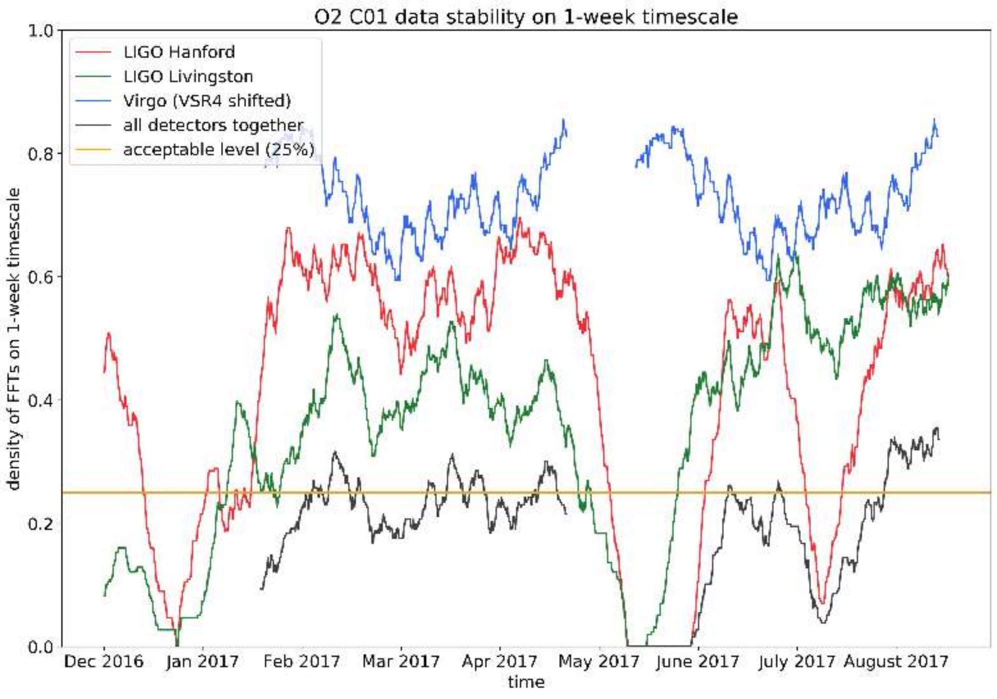


Figure 14: Duty cycle of the three single detectors and of the network as a whole, averaged on a timescale of 1 week. The horizontal line indicates what we believe is the minimum acceptable level of the network's duty cycle to be able to construct our training images without too many gaps.

5.7 The choice of macrochunks

Our aim is to build RGB spectrograms, which will constitute the images fed to our neural network classifier. As will be discussed in the next section, an RGB spectrogram is just a stacking of three colored spectrograms, one for each detector. We are interested in minimizing the presence of holes in the RGB images. We want thus to choose our temporal slices of ~ 6 days where we have the maximum global duty cycle possible. As can be seen in figure (14) following the black line, the best time slices are:

- in the middle of February
- in the middle of March
- in the middle of April

- around the beginning of August

Of course we have carefully fine-tuned the temporal placement of the two VSR4 Virgo datasets in order to emphasize the peaks already present in the LIGO detector network.

The whole O2 run is then divided in temporal macrochunks of $\sim$ 6 days (the time included in one image) and a macrochunk is used in the following analysis only if the average global duty cycle inside it was more than 25% (yellow line in figure (14)). We thus obtained $N_{macrochunk}$ sub-datasets, each one containing the frequencies from 80 Hz to 120 Hz for all the three detectors. Given the frequency resolution of 1/8192 Hz, in each macrochunk we have 3 spectrograms of 128 time pixel $\times$ 327680 frequency pixels.

We have then split those 327680 frequency pixels in sub-bands of 256 pixels each, thus obtaining 2560 images of 256 $\times$ 128 pixels for each detector.

At the end we have $N_{macrochunk} \times 2560 \times 3$ images of 256 $\times$ 128 pixels each, similar to those in figure (15).

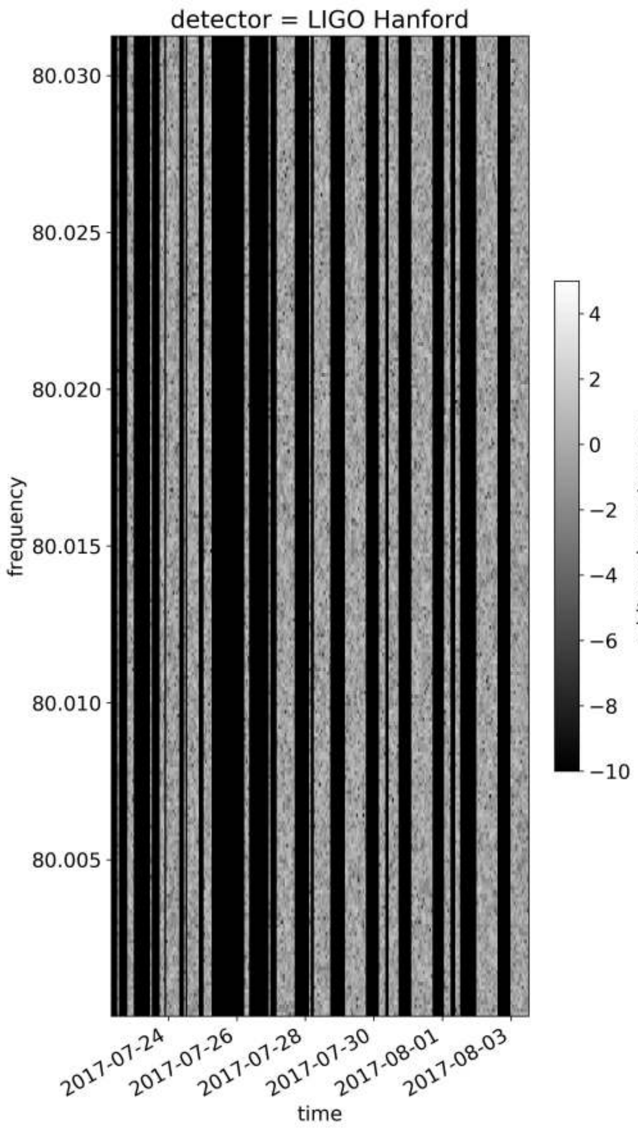


Figure 15: An example of one of the 2560 images in every temporal macrochunk.

5.8 Create the background images: RGB color spectrograms

For each given time macrochunk (128 pixels) and frequency sub-band (256 pixels) we have three different images, one for each detector, each one similar to the one depicted in figure (15).

Given the fact that a gravitational-wave signal must be simultaneously present in at least two detectors to be classified as such, we developed a way to visually enhance the coincidences between the three detectors. The method consists in using a different color scheme for the spectrograms of each detector, using one of the three primary colors: red (R), green (G) and blue (B). Some examples of these single-colored spectrograms are depicted in figure (17).

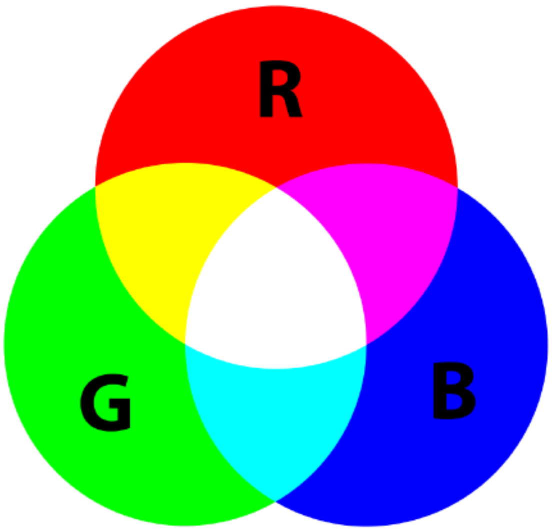
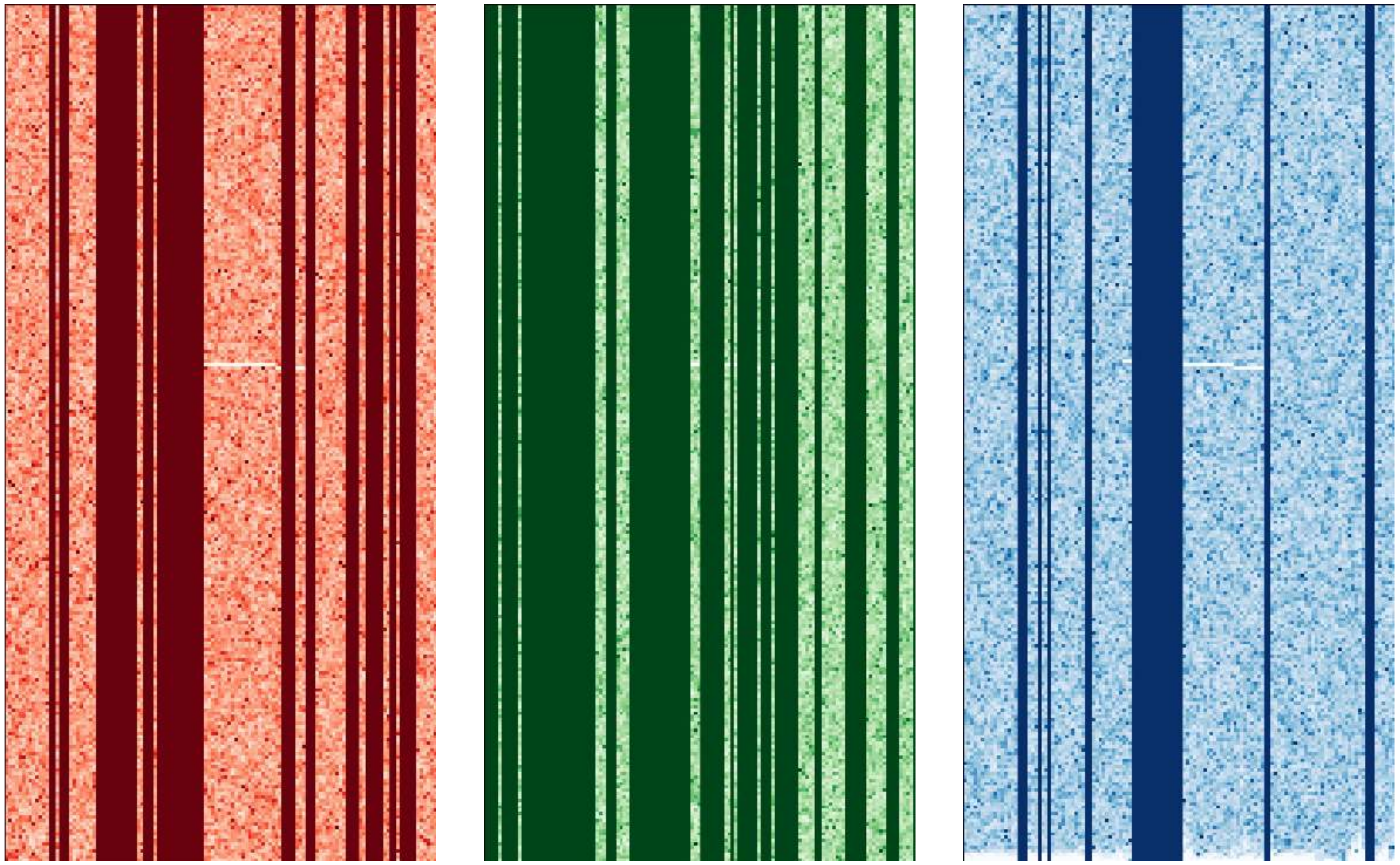


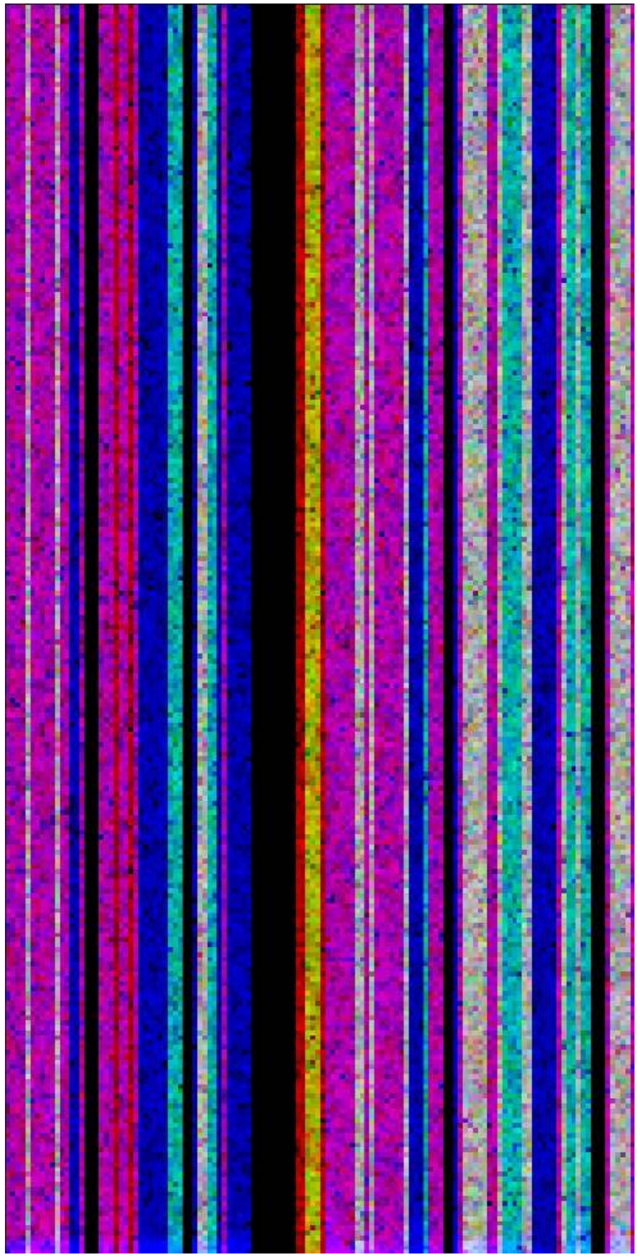
Figure 16: The mechanism of additive color synthesis. LIGO Hanford is assigned to red, LIGO Livingston to green and Virgo to Blue. A complete coincidence will appear in white, while a partial coincidence in yellow, magenta or cyan, depending on which couple of detectors is involved.

Then the three single-colored spectrogram (figure 17) are stacked together to give a single RGB image (figure 18).



(a) LIGO Hanford = Red (b) LIGO Livingston = Green (c) Virgo = Blue

Figure 17: Three colored spectrogram, to be stacked together to obtain a single RGB image.


(a) noise-only RGB spectrogram

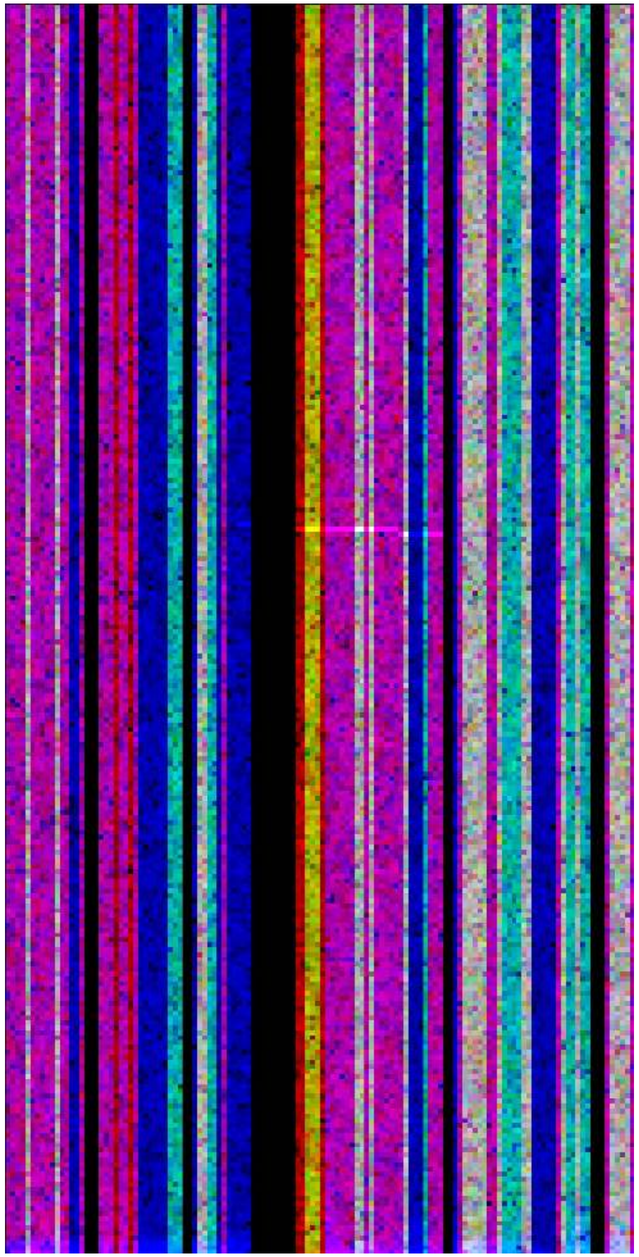

(b) RGB spectrogram with an injected signal

Figure 18: RGB spectrograms obtained stacking together the three single-colored spectrograms in figure (17).

As can be seen in figure (18a), we have different color combination, according to the additive color synthesis (figure 16). The black vertical stripes correspond to the chunks where the data from all the three detectors are missing (because they were excluded as corrupted or because the detectors were temporally off). The stripes of primary colors (red, green, blue) are the temporal chunks where only one detector is present. The stripes colored in yellow, magenta or cyan are the ones that have two detectors simultaneously active. Finally, the white stripes have all the three detectors working together.

As opposed to the three single-colored spectrogram in figure (17), the RGB spectrogram in figure (18b) shows a more consistent display of the injected signal, enabling the viewer to immediately recognize it as a coincident signal. We notice how the line appears globally white, even if it changes color every time it pass through a different vertical stripe. With this trick, we can use coincidences even for small periods of time: if a signal appears as a coincidence in two detectors and than later disappear from one of the two detectors (because of the antenna pattern or a technical shutdown), we can still easily visually follow it and determine its parameters, such as the ending time or the spindown.

With this method we can immediately reject the long disturbances (hereafter called “long glitches”) that mimic the signal in one detector, because they do not have any composite color (yellow, magenta, cyan or white). The RGB spectrogram is a compact representation of the data that aid to cross-correlate between different detectors just by eye. It is thus a perfect way to represent our data for the image recognition task that we want to perform with our convolutional neural network.

5.9 Signal injections in the frequency domain

All the preceding figures (17a, 17b, 17c and 18b) have a signal inside them. In section (4) we described the procedure to inject signals starting from the time domain. But all the signals in these pictures are not injected following that procedure because all the images we analyzed were constructed starting from the .SFDB09 files, which we remind are a collection of spectral amplitudes. Thus we have to inject signals directly in the frequency domain.

The signals we will inject are just straight lines in the spectrogram, with no artifacts of any kind due to the windowing (frequency-domain side lobes due to the time-domain apodization function). There is thus no frequency spreading, even due to a not-perfectly-corrected Doppler shift. Those straight lines are somehow contradictory: even if they do not have the spindown corrected, they have no frequency broadening. In the reality, when a signal is not a pure sinusoid, even with slowly-varying frequency, there is always an unavoidable frequency broadening, much larger than the dispersion due to the shape of the Fourier-transformed window function. Moreover, those signals are perfectly Doppler corrected. This is of course an approximation: at any time, all the energy of the signal is concentrated in only one frequency bin; this maximizes signal visibility. We remind that in our pipeline we are building a toy model, so it does make sense to use simple toy signals.

The straight segments are not directly superimposed on the background images, because the scale of the images is logarithmic and

$$\log(\text{background} + \text{signal}) \neq \log(\text{background}) + \log(\text{signal})$$

We thus have to do the injection and *then* convert the whitened spectrogram to logarithmic values and plot it as an image.

To conclude, we injected a signal by adding to the whitened spectrogram's "nonlogarithmic" pixels a straight line, with constant fixed intensity. We created different datasets with different (decreasing) injected signal intensities: 32, 16, 8, 4, 2, 1. But how those added numbers relate to the noise level? A whitened spectrum, by definition, has median 1. The median is different from the mean, because its definition imply

$$\log(\text{median}(x)) \simeq \text{median}(\log(x))$$

where the $\simeq$ symbol takes into account the little possible variations due to the binning differences between the linear and logarithmic values. Thus, the logarithm of a whitened spectrum has median 0 (figure 19).

The yellow line depicted in figure (19) represents the peakmap threshold. At this level, it doesn't matter what a peakmap is (it will be defined later, when we will talk about the follow-up): suffice is to say that all the data below this threshold

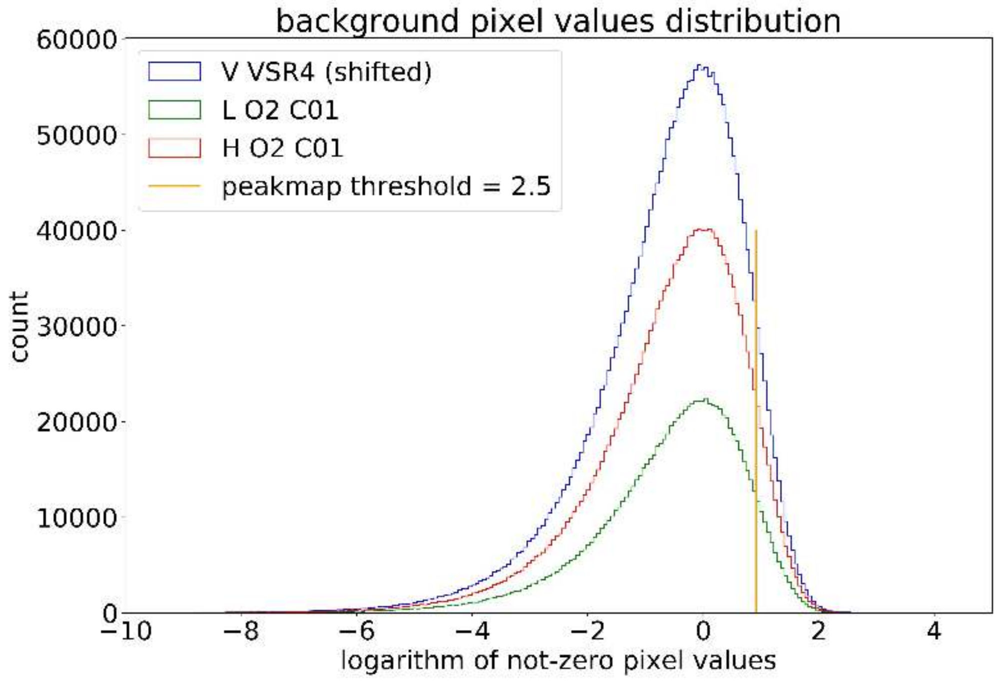


Figure 19: Histogram of the logarithm of the nonzero values of all the pixels inside an RGB spectrogram.

are discarded from the follow-up analysis. Talking about "nonlogarithmic" pixel values, the peakmap threshold is placed at $R = 2.5$ [13], where R is the whitened ratio (which we remind has median 1). This means that an injected signal of intensity 1 will be mostly discarded from the peakmak (and thus from all the succeeding analysis) because

$$1 \text{ (noise median)} + 1 \text{ (signal)} < 2.5 \text{ (peakmap)}$$

If our analysis will be able to detect an injected signal intensity of 1 (as at the end will be), it will be able to potentially recover signals that are otherwise discarded from the current analysis that involve a peakmap threshold of 2.5: those signals are actually undetectable with such pipelines. We remind, however that the peakmap threshold can always be lowered (for example to the value 2), but this operation imply a huge increase in the request of computing power to be able to handle all the resulting candidates.

6 Artificial intelligence

6.1 Deep learning

The quest for the construction of an artificial life, a machine that is able to think, dates back in the history of humankind. In the last century, this quest was addressed by the fields of Cybernetics and Artificial Intelligence, whose development led to the construction of machines that are able to automate routine labor or beat humans at chess. In the last decades many huge progresses have been made: today there are machines which can visually recognize objects, understand natural languages, make translations, propose medical diagnoses, drive cars and even beat humans at the game of Go. The humankind succeeded to build machines that outperform humans themselves in many difficult tasks: this may not be properly called "intelligence", but surely is a first step in the right direction.

However, there's a big conceptual distinction between being able to play chess and being able to recognize a dog. The first task is intellectually difficult for humans but relatively straightforward for computers: chess can be completely described by a list of formal rules and the problem can be resolved by brute-force by what is called an *expert program*, which is a machine with all the required knowledge hard-coded into it. This is called the *knowledge base* approach to artificial intelligence. The second task is indeed easy for people to perform, but hard to be described formally and, consequently, incredibly hard to be done in an automatic fashion by an expert machine. Accurate image recognition implies owning a huge amount of knowledge about the world, which is often subjective, intuitive and difficult to formalize. Computers need to capture this knowledge in order to behave in an intelligent way or at least to emulate an intelligent behavior.

The solution to those problems, that are easy to solve for humans, required a big forward step in the design of the information-processing mechanism inside the machines. Systems relying on hard-coded knowledge were proven to have many difficulties when dealing with intuitive problems such as recognize someone's voice or distinguish a dog from a cat. This suggested that machines need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as *Machine Learning* [72].

Simple machine learning algorithms, such as logistic regression, depends heavily on the *representation* of the data they are given: the algorithm will learn to solve a task by adapting its internal parameters (so modifying its internal knowledge), but always starting from a set of pre-determined *features*, which are the pieces of information that are marked as relevant by the data scientist. Data must be represented in an effective way: for instance, the points in a scatterplot may be represented by their cartesian or polar coordinates, depending on which representation makes the final task easier. The choice of the right representation is crucial to correctly and easily perform a given task. For instance, it's straightforward to do algebra with the arabic representation of numbers, while it's much more difficult to do it with the roman representation of numbers. Thus, feature design and extraction is an essential part of many automatic processes, trying to provide a summary of the dataset as concise and as complete as possible.

However, it is often difficult to know which set of features is the most effective and how those features should be represented in the different situations. For example, the presence of the tail is a useful information when classifying cats versus cars, but a tail can have many different shapes, thicknesses, colors, positions, orientations, etc. Formalize all this knowledge is difficult, especially when dealing with abstract concepts that can have many different variations. So, as with machine learning we were letting the machine to learn the informations (the parameters) of the classifier, with *Representation Learning* we are also letting the machine to learn the proper representation of the data, building its own set of features to use as the input for the classifier (whose parameters are also learned from the data). By gathering knowledge from experience, this approach avoids the need for humans to formally specify all the knowledge the computer needs.

Anyway, the most abstract features are often the most subtle but also the most discriminating. For example, the accent can be used to discriminate between a scottish and an irish speaker, both using the english language. The accent will influence every piece of data we will observe (every sound and every word), but disentangling those informations is very difficult: for instance, if you are beginning to learn the german language, you won't likely be able to recognize if someone has a Berlin accent: that will require a lot of practice and a deep mastering of the language and all its phonetic variations. Since extracting high-level abstract features is in general very hard and computing and data intensive, a smart decomposition of the problem is required.

One solution was thus to let computers learn from experience (raw data) and understand the world in terms of a hierarchy of concepts, where each concept is defined through its relation to simpler concepts. With complex and abstract features, the graph of this hierarchy of concepts turns out to be quite deep: that's why this approach is called *Deep Learning.* In deep learning the feature extraction part and the classification part somehow fuse together, with the classifier using features at different levels and implementing logic gates between different representation levels. The machine thus learns to perform a multi-step and multi-scale algorithm, which often offers an exponential speedup with respect to less effective (swallower) decompositions of the computation [70]. So, for tasks requiring abstract features, the deeper the better.

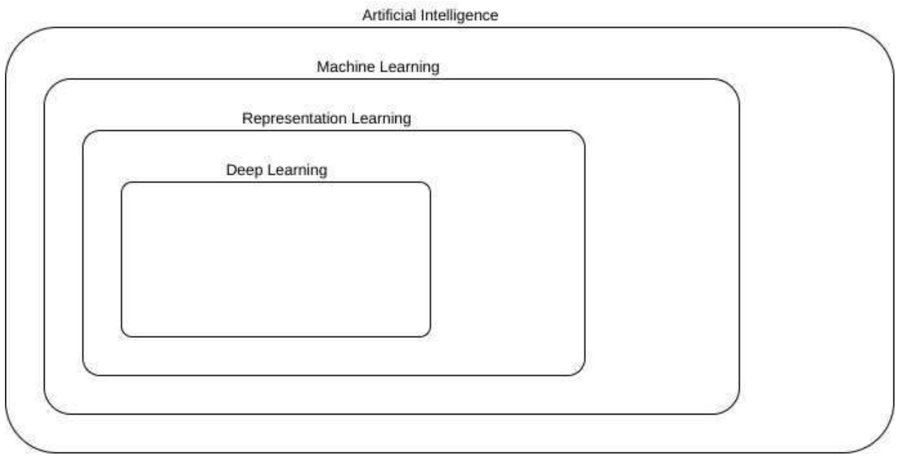


Figure 20: A Venn diagram showing how Deep Learning is a subset of Artificial Intelligence.

6.2 Biologically-inspired artificial neural networks

The idea underlying deep learning aim to mimic the processing of informations that takes place in the human brain. Despite the enormous complexity of the biological brain, we can construct a naive model of it by schematizing it as an ensemble of neurons linked by synaptic connections with a peculiar network topology. Our biological mechanism of vision seems to involve a very deep neuronal structure, starting from the bottom of the eyes and finishing near the back of our head. So there are neurons which are more closely connected to the raw image input and other additional neurons, organized in a hierarchical fashion, which gradually elaborate the available information. This structure was emulated in Deep Artificial Neural Networks (figure 21), where the artificial neurons are organized in layers and the information regarding the input flows from one end to the other (*feed-forward* networks).

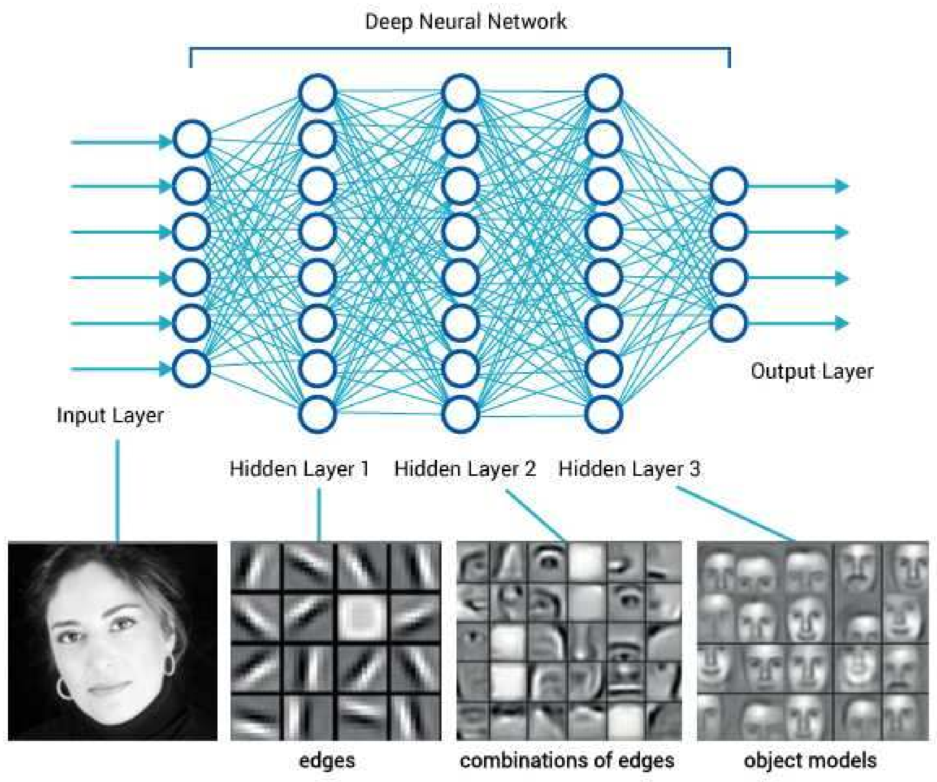


Figure 21: An artificial neural network, with an example of the representation hierarchy of the hidden layers.

This biologically-inspired kind of computation turns out to be very different from the computation we are used in our current software: it is distributed, redundant, sparse, hierarchical and fault-tolerant. In the following section we will review all the basic building blocks that constitute an artificial neural network, with particular emphasis on *convolutional* neural networks. We will discuss how these networks are constructed and how their pieces connect and interact together, giving rise to this peculiar type of computation.

7 Model architecture

An artificial feedforward neural network is organized in consecutive layers: those layers could be of different kinds and each type have a specific usage and is required to achieve a specific goal.

Neural networks are highly nonlinear structures that were proven to be universal approximators [82]. Their high degree of nonlinearity is obtained by stacking many many layers one on top of the other, alternating linear parts with slightly nonlinear parts.

In the following we will focus on Deep Convolutional Neural Networks, which are actually the state-of-the-art in the field of image recognition. The architecture of the neural network we have built (our *model*) is presented in section (7.15), but some of the design choices are discussed directly in these subsections regarding the various building blocks.

7.1 Fully connected

Fully connected or *dense* layers are the building blocks of the more general feedforward neural networks.

The network depicted in figure (22) has five dense layers: two visible (the input and the output one) and three hidden. Every neuron x_i in every layer l is connected with all the neurons of the following layer $l+1$. As shown in the image, the resulting graph between two layers is fully connected: every unit interacts with every unit in the following layer. Every synaptic connection between neurons has a given strength, which can vary in time due to the learning process. These connections are called weights and are mathematically described by a dense matrix $w_{ij}^{(l+1)(l)}$ that links the two vectors of neurons $x_j^{(l)}$ and $x_i^{(l+1)}$ (in abstract index notation

[51]).

$$x_i^{(l+1)} = w_{ij}^{(l+1)(l)} x_j^{(l)}$$

So, for the moment, the values of the neurons for a given layer can be obtained with a chain of simple matrix multiplications starting from the first input layer. We can also introduce some *bias* parameters $b_i^{(l)}$ for every layer of neurons, in order to obtain an affine transformation

$$x_i^{(l+1)} = w_{ij}^{(l+1)(l)} x_j^{(l)} + b_i^{(l+1)}$$

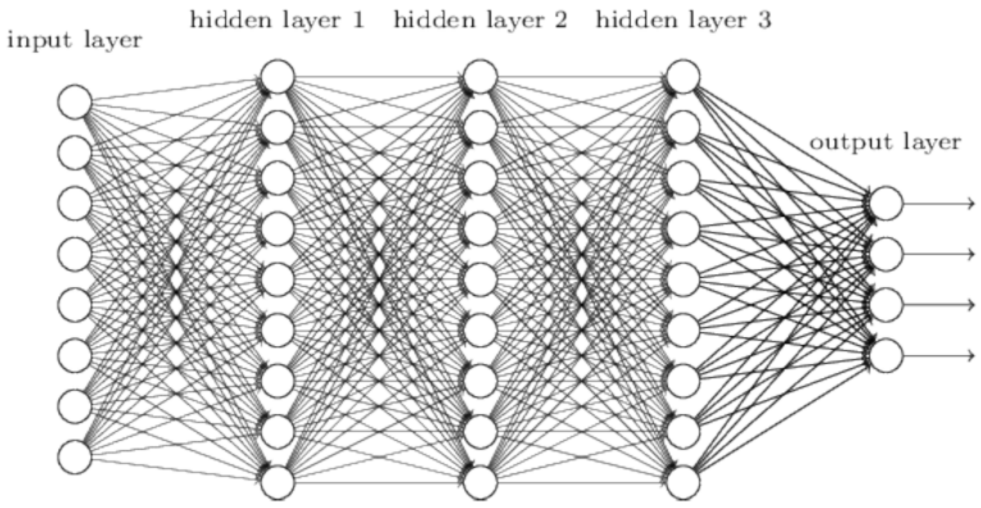


Figure 22: An example of a 1D fully-connected neural network, with three hidden layers with nine neurons each.

The number of neurons in every layer, especially in the hidden ones, is problem-dependent. We will not cover this topic in detail: we will only highlight that if the number of neurons decreases between two layers we obtain a compressed representation of the data (which is often useful), while if the number of neurons increases between two layers we obtain a higher-dimensional representation (which, again, is often useful). In particular, a compressed representation helps to distill the relevant informations in the data, while an higher-dimensional representation helps to implement a simpler classifier, due to the Cover's theorem [81], which states that the probability of two classes being linearly separable increases when we increase the number of dimensions of the space where we nonlinearly map the data.

As we said, artificial neural networks are nonlinear objects, so we have to amend the previous formula

$$y = w \cdot x + b$$

with an element-wise nonlinear activation function at every layer

$$y = \varphi(w \cdot x + b) \tag{21}$$

Activation functions are another element drawn from biology: every biological neuron has an activation potential, that can often be in form of a threshold; when the sum of its inputs pass the threshold the neuron fires, giving rise to an electrical spike which will be transmitted to the neighboring neurons. The activation function used in our neural network will be presented in section (7.3).

So, looking again at figure (22), the values of the output neurons will result from a chain of linear and nonlinear parts. We can write this chain with a simplified notation as

$$y = \varphi(w_4 \cdot \varphi(w_3 \cdot \varphi(w_2 \cdot \varphi(w_1 \cdot x + b_1) + b_2) + b_3) + b_4)$$

The equivalent function f learned by the neural network

$$y = f(x; \theta)$$

will be a highly nonlinear function of the input x, with all the learned weight and bias parameters summarized in the symbol θ, which represents the whole configuration of our neural network.

7.2 Convolutions and cross correlations

The fully-connected network depicted in figure (22) has 3 hidden layers, 270 weights (synaptic connections) and 35 biases. Modern networks require hundreds of hidden layers and so we have to find a way to lower the number of used parameters in order to be able to train them effectively.

This need has lead to modern convolutional networks [34], which are architectures inspired by the structure of the mammalian visual system. Convolutional Neural

Networks [32, 33] are by far the most used networks in Computer Vision. They are specialized for processing data that have a known grid-like topology. Mathematically, the matrix multiplication seen in equation (21) is simply replaced with a convolution (or, as we will see, with a cross-correlation). Conceptually, convolutional layers can be seen as a restriction to the general fully-connected layers: they implement both *local connectivity* and *parameter sharing*.

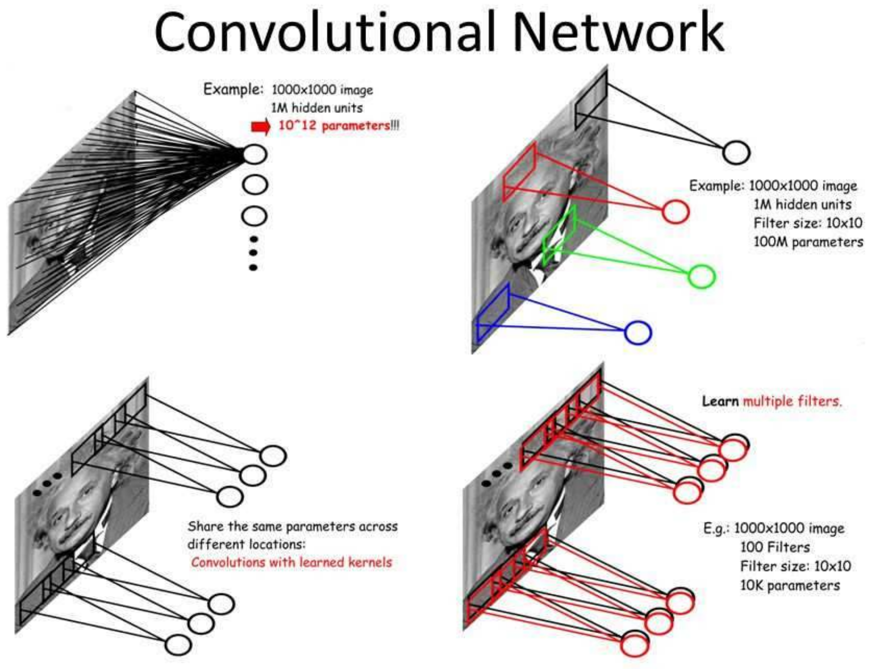


Figure 23: A 2D example of the difference between full connectivity (top left), local connectivity alone (top right), local connectivity with parameter sharing (bottom left) and local connectivity with multiple-filter parameter sharing (bottom right).

Local connectivity means that every neuron is no more connected with all the neurons on the previous layer, but only to the neuron in a given finite region, called its *receptive field*. The graph is no more fully-connected. This biologically-inspired constrain accounts for the fact that neurons in the visual cortex only process local informations, looking for local patterns and then assemble them together in the deeper layers. Local connectivity greatly reduces the number of needed parameters: for example, if we prune all the connections outside a receptive field of three neurons from the network depicted in figure (22), we pass from 270 weights to 93 parameters.

Let's now talk about parameter sharing. Images have many statistical properties that are invariant under translations: for example, a photo of a cat remains a photo of a cat if it is translated by few pixels in whatsoever direction. Convolutional networks take this property into account by sharing parameters across multiple image locations. The same feature (convolutional filter, or *kernel*) is computed over different locations in the input image. In the example above, the 93 local independent parameters become 12 shared weights, with a single kernel of length three. To summarize: we now have local connections that share their parameters nonlocally. Parameter sharing has thus enabled convolutional networks to dramatically lower the number of model parameters and to significantly increase network size without requiring a corresponding increase in training data. Of course having only one convolutional kernel can be not enough, so we can add other convolutional filters to the network, to fulfill our needs.

Let's move from the previous 1D example to a 2D example: let our input be an image 1000×1000 pixel and our single kernel be 5×5 pixels (so a receptive field of 5 for the convolutional neuron). We have to place our kernel in one position of the image, do our computation and then move the very same kernel to another position an do the computation again until we tile the entire image. Even if the image is 1000×1000, the number of parameters for this convolutional layer is always $5 \times 5 = 25$. A similar example is shown in figure (23), with a kernel 10×10.

We now want to describe how exactly the computation is carried on. The mathematical definition of convolution between two real-valued functions is

$$s(t) = (x * w)(t) = \int da\, x(a)w(t-a)$$

where the function x is the input, the function w is the *kernel* and the output s is the *feature map*. The discrete counterpart of the previous formula is the following 1D convolution

$$s[t] = (x * w)[t] = \sum_a x[a]w[t-a]$$

while the convolution for a two-dimensional image I with a two-dimensional kernel

K is

$$S[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n] \tag{22}$$

Many software libraries do implement the *cross-correlation* instead

$$S[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i+m, j+n]K[m,n] \tag{23}$$

which is equivalent to a convolution with the kernel flipped [3]. The flipping of the kernel is irrelevant for the learning: the algorithm will converge to the appropriate kernel, flipped or not. That's probably why the operation in equation (23) is very often (incorrectly) called "convolution" too.

As we were previously saying, in a convolutional layer there can be many convolutional filters. Those different filters will be stacked together to form a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors (notice that this definition is different from the one used in the Theory of Relativity, because it doesn't imply Lorentz covariance). Given the linear nature of the operation, the convolution between tensors can be done in parallel on a GPU hardware, via optimized libraries such as Nvidia CUDNN.

The number of kernels to be used in every convolutional layer of the network (together with their kernel size) is problem dependent and also layer dependent. As before, we will only sketch which are the main lines of reasoning. Of course, more kernels with a bigger receptive field will mean more parameters to be trained and thus more memory requirements. It is a common practice to place few kernels in the lower layers of the network, because their learning tend to converge to few very general and simple features: lines, colors, edges, orientations (figure 24).

Moreover, it's a very common practice to gradually increase the number of kernels and their sizes with the depth of the network. This account for the fact that the deeper kernels are the result of a complicated composition of the earlier ones, resulting in a big variety of more complicated features (figure 25).

[3] If we start the indexing from the center of the kernel matrix, the flipped kernel is defined as $\tilde{K}[m,n] = K[-m,-n]$. Thus, with the substitutions $m \to -m$ and $n \to -n$ in equation (23) we obtain $\sum_m \sum_n I[i-m, j-n]\tilde{K}[m,n]$, which is equivalent to $\sum_m \sum_n I[m,n]\tilde{K}[i-m, j-n]$ (equation 22) because the convolution operation is commutative.

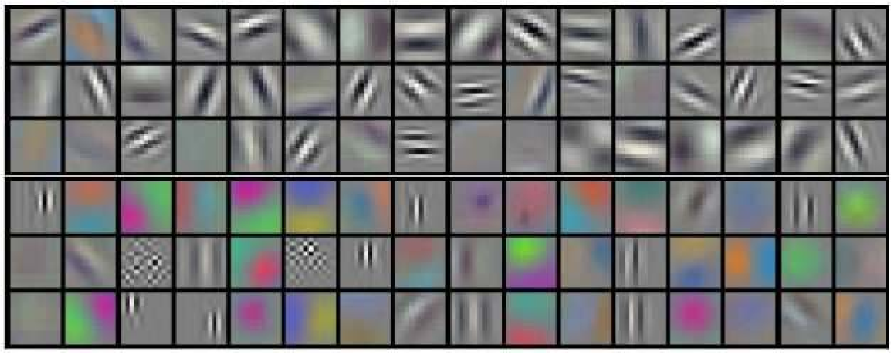


Figure 24: Some simple features, usually learned in the early layers of the network.

These complicated features need to have their proper "space" in memory. For example, if a given convolutional layer has 8 filters, we want the filters of the next layer to embed compositions of couples of these 8 previous filters, so we may want to set a number somewhere between 16 and 64 kernels for that layer. This is done to take into account the growing complexity of the learned features.

Our images do require features that are far less complex than those shown in figure (25), so we decided to keep our network small: we used 6 convolutional layers with only 9 filters for every layer, each one 3×3 pixels wide. However, we did not optimize this number in any way: the optimization of the number and size of the kernels is a long process, often involving the use of *generative* networks to visually check which features the model is effectively learning. This will be done in the upcoming future.

In summary: Convolutional Neural Networks allow us to effectively incorporate domain knowledge into the network architecture. With respect to fully-connected networks, we lose permutation invariance and we gain the notion of topology. This allows us to create networks that are specialized to deal with images of variable size and particularly well suited to the problem of pattern recognition inside them [70]. This is done by implementing local connectivity and by sharing the kernel weights: those are prior assumptions that, when satisfied, effectively lower the number of parameters or make other huge simplifications to the calculation. Of course those assumptions must be satisfied for the process to succeed: for example, the assumption of shared weight immediately crashes if we don't have translational invariance in the image. That's why we selected an almost-flat frequency band of

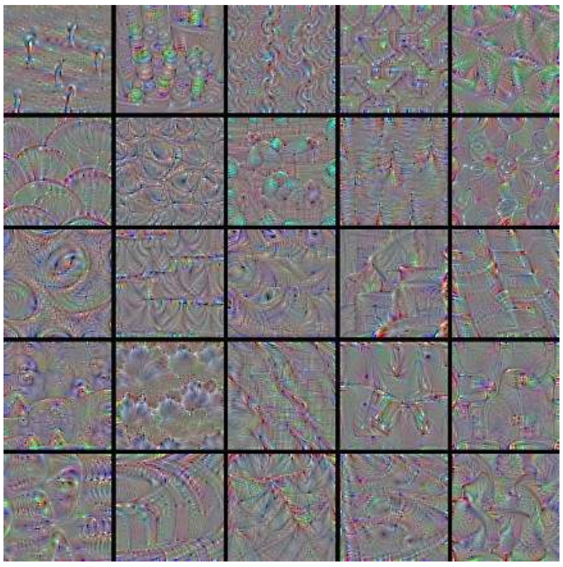


Figure 25: Some complicated texture features, usually learned in the intermediate layers of the network.

the detector's sensitivity curve and used a whitening phase: we want to have approximate frequency translation invariance.

Our convolutional model, discussed in section (7.15), is very simple: it doesn't have any branch, recursion loop or block decomposition (such as the Inception module [37]). We don't want to build a state-of-the-art network: research into convolutional networks proceeds so rapidly that a new state-of-the-art architecture is announced approximatively every month. We only want to build a toy model to begin to probe the possibility of neural networks in the field of gravitational-wave data analysis.

7.3 Rectified Linear Unit (ReLU)

A Rectified Linear Unit (ReLU) is the activation function defined as

$$\mathrm{ReLU}(x) = \max(0, x) \tag{24}$$

It is a piecewise linear function: so it is nonlinear while preserving a lot of useful properties of linear functions.

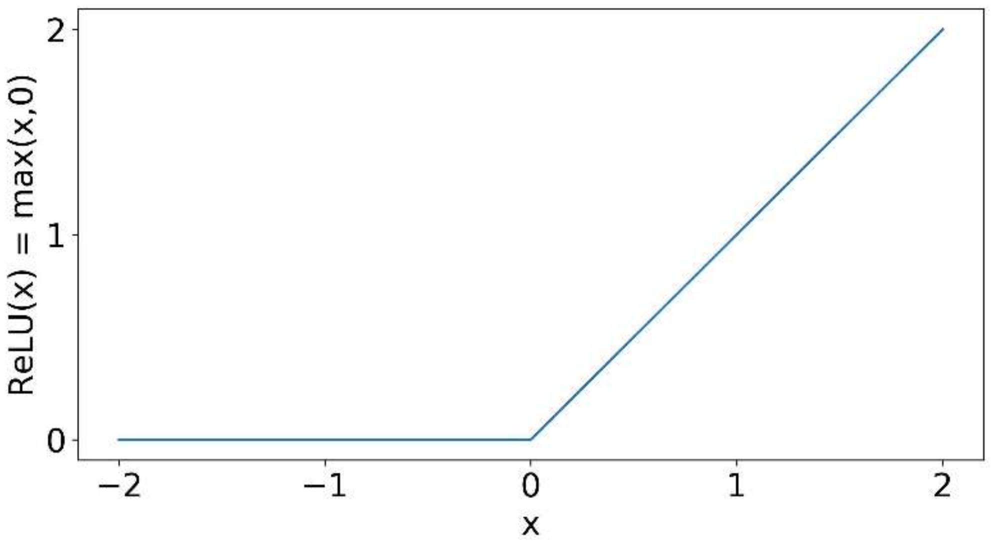


Figure 26: Graph of the ReLU activation function.

Linear models are easy to optimize with gradient-based methods. The derivative of the ReLU is always 1 where the unit is active (that is: where the input is positive) and the second derivative is 0 almost everywhere. This means that the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects [72]. Moreover, ReLU is a convex function, which is a very useful property when searching for points of minimum.

One ReLU drawback is that the learning via gradient methods cannot take places with examples where the activation is zero (that is: where the input is negative). To partially solve this issue, the *bias* elements are often initialized to small positive values. There also exist some ReLU generalizations that specifically take this problem into account [72].

ReLU is not differentiable in $x = 0$. This may seem to invalidate it for use with a gradient-based learning algorithm. We have empirical evidence that, in practice, gradient descent still performs well enough. This is in part due to the fact that Deep Neural Network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly [72].

While not differentiable in $x = 0$, ReLU has well-definite right and left derivative in that point. When the input is zero, instead of reporting that the derivative is undefined or raising an error, current software implementation will randomly return one of the one-sided derivatives during the backpropagation. This may be heuristically justified by the unavoidable presence of numerical errors and truncations: during the training phase, a value of 0 is very unlikely to be exactly zero; more likely, it may come from the float32 truncation of some very small value around zero.

Numerical experiments show us that the behavior of different activation functions can be very counterintuitive: the *softplus*, a smooth version of the ReLU defined as $\zeta(x) = \log(1 + e^x)$, has worst performances compared to ReLU, despite it being differentiable everywhere, and the same empirically holds true for the *sigmoid* or *tanh* activations.

While having a low degree of nonlinearity, the ReLU activations can be combined across many layers to allow the model to learn an arbitrary nonlinear function.

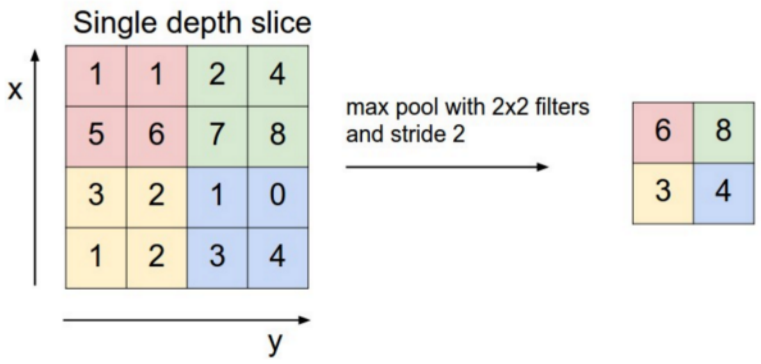
ReLU is thus the default recommended activation function for hidden units in modern feedforward neural architectures: it's simple, very effective and computationally cheap. A big part of the modern development of neural networks can be ascribed to it.

7.4 Max pooling

A pooling function replaces the output of a neuron with a summary statistic of the nearby neurons. The *max pooling* operation [36] returns the maximum output within a square neighborhood.

By the effect of pooling, the representation becomes approximately invariant under small translations of the input. That means that most of the outputs won't change when the inputs are translated by a small amount. Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is. For example, if we want to recognize a face we don't need to know the exact location of the eyes with pixel-perfect accuracy, so we can use a more robust and lighter computation.

Because pooling summarizes the responses over a whole neighborhood, many units are redundant and we can use just a fraction of the output neurons, discarding all the others: just like a downsampling operation. The smallest pooling size is 2x2 and a common choice is to half its linear size, resulting in a feature map reduced by a factor of four, as shown in the figure.



This downsampling operation strongly reduces the parameters required for the next layer, thus making the remaining part of the network to weight less in memory. Given the limited amount of graphical memory in current GPUs, this turns out to be a crucial element in architecture design.

We explained that max-pooling over spatial regions induces invariance to translations, but if we max-pool over different convolutional filters the network can learn which transformation to become invariant to. An example is depicted in the following figure: max-pooling over filters with small rotations will induce approximate rotational invariance. This is very useful because the concept of "two" is indeed invariant under small rotations.

Rotations and other symmetry transformations should be small: this can be immediately understood analyzing the case on the number nine: what happens if there is a rotation of 180 degrees? Will the number be correctly classified?

Apart from the memory benefits and the gain of invariance, there is also another reason why pooling is so important: it induces in the network an explicit *hierarchical structure*, leading to a hierarchical decomposition of the visual space. Different features live at different observing scales: for example, it doesn't make sense to search for an eye if we are looking at a face with a microscope; viceversa, it doesn't make sense to search for a single hair if we are looking at a photograph taken far away from the subject with a wide-angle lens.

Concepts are arranged hierarchically, with a pyramidal interdependence between different scales (at least in feedforward networks, where there are no synaptic loops). For example, in a regular family picture we have the following multi-scale structure:

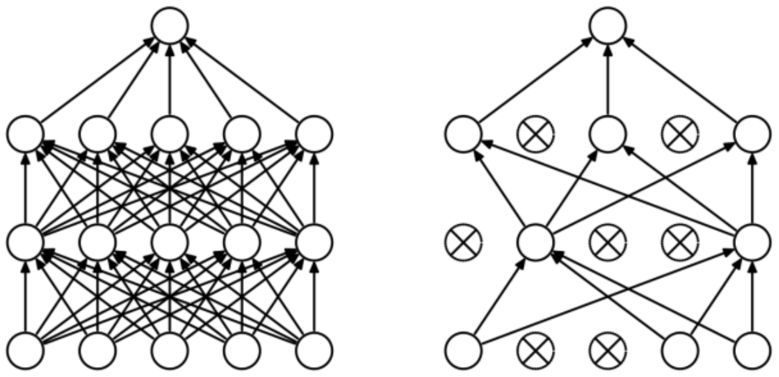
- small scale: *line*, *angle*, *edge*, *color*
- intermediate scale: *eye*, *nose*, *ear*
- large scale: *face*, *body*, *human*
- full-length scale: *brother*, *son*, *family*

Some readers will immediately find many analogies with the *Renormalization Group* in the context of the Ising model. The pooling operation is indeed very similar to the Kadanoff blocking procedure, being the bridge to a higher level of description. The intimate relation between Deep Neural Networks and the Renormalization Group is still unknown. Some theoretical works are beginning to address this incredibly interesting connection between Artificial Intelligence and Statistical Physics [70, 41].

7.5 Dropout

Dropout is a computationally inexpensive method of regularization [75]. The power of the method lays in the fact that it provides an approximation to training and evaluating a bagged ensemble of exponentially many neural networks [72].

Specifically, dropout trains the ensemble consisting of all subnetworks that can be constructed by removing non-output units from the base network. Those subnetworks are not independent and do share parameters between them, so a weight update relative to one subnetwork will affect the convergence of all the other subnetworks. The unit removal is done by multiplying its output value by zero: for every image in the minibatch, a random binary mask is generated and is applied to all the units in the given layer. A dropout layer usually acts on hidden neurons, but can also act directly on the input: in that way it behaves similarly to data augmentation.


(a) Standard Neural Net (b) After applying dropout.

It is important to stress that all the submodels in the ensemble share some hidden units. This means that training with Dropout forces hidden neurons to perform well regardless of which other hidden neurons are active. Units thus must map features that are general-purpose and useful in many different contexts. The random shutdown of neurons can be seen as a form of targeted and adaptive disruption of the information contained in the input image rather than the corruption of its raw pixel values (as it's done for example with the regularization done by the addition of noise).

For example, in the context of face recognition, we can imagine that a hidden unit h_i has learnt to recognize a nose in the image. Dropping h_i means that the face recognition must succeed even if the information about the presence/absence of a nose is discarded. So the model is forced to learn to detect a face from the presence of other features, such as for example lips or eyes, or to redundantly learn other ways to encode the presence of a nose [72]. That's what robustness and fault-tolerance mean. Traditionally, if some noise is added to the input image, the information about the presence of a nose is not erased, unless the noise magnitude is so high that nearly all information in the original image is lost.

Dropout, like every other regularization technique, reduces the effective capacity of the model and slows down the training. That's the price to pay for a better generalization capability.

Dropout is more effective than other regularizers such as weight decay or sparse activity regularization [75]. It also guarantees the fault-tolerance of the network and the distributed nature of its internal computations (the classification output must be correct even if lots of neurons are randomly shutted down).

To summarize, Dropout is computationally inexpensive, conceptually simple, robust and well implemented: that's why it is the most widely-used implicit ensemble method and that's why we've chosen it as the regularizer for our network.

After many trials and fine-tunings, we have placed a dropout layer with probability $P = 0.2$ after every convolutional block. This means that the input of every neuron of the network is randomly erased with 20% probability. A higher probability value, such as the typical $P = 0.5$, turns out to give a too much difficult training. The chosen value $P = 0.2$, albeit small, is enough to guarantee enough noise to stabilize the learning phase (for example lessening the risk to be trapped in some local minima).

7.6 Flatten

The flatten layer simply reshape all the previous neurons in a one-dimensional vector. This operation erases the information about topology, generally marking the boundary between the convolutional and fully-connected part of the model. This can be seen as the point where the correlation length of the input diverges, resulting in a fully-connected graph. The concept of metric thus becomes meaningless (every node of the graph is connected with distance 1 to every other node) and so also the one of topology.

7.7 Softmax

The Softmax function is used to represent a probability distribution over a discrete variable with n possible values. It is thus used as the output of a classifier to represent the probability distribution over n different classes. Its alternative name, *softargmax*, suggests that it indeed is a smooth version of the *argmax* function. A softmax layer is defined as

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\Sigma_j e^{z_j}} \tag{25}$$

where the indices i and j run from 0 to n and $\mathbf{z}$ is the output of the n neurons in the preceding layer.

It is a generalization of the logistic sigmoid function, which is used to represent a probability distribution over a binary variable

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

because $\text{sigmoid}(x) = \text{softmax}(\mathbf{z})_1$ in the case of binary $\{0, 1\}$ classification.

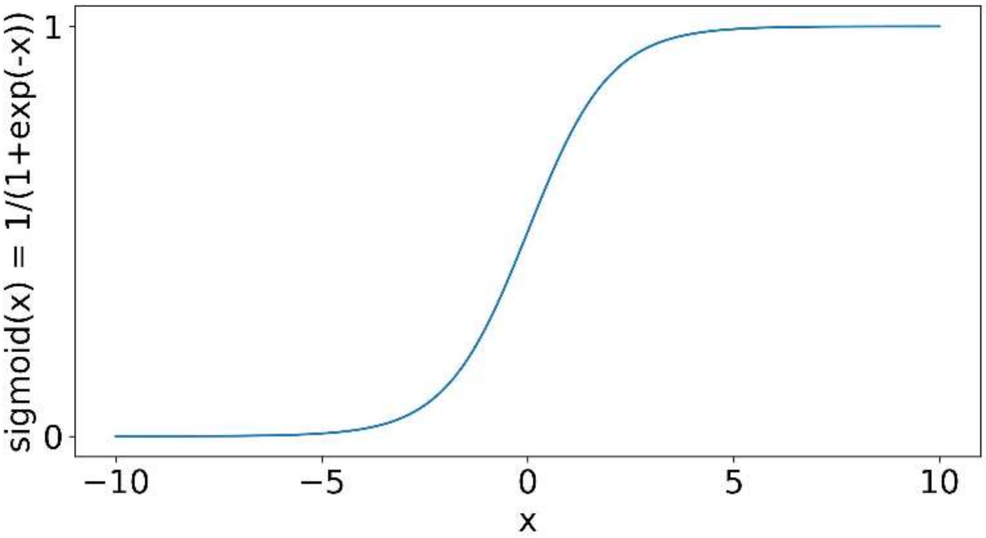


Figure 27: Graph of the sigmoid activation function.

The output y_i of a softmax layer represents the probability distribution $P(i)$ of a discrete variable, because it satisfies the Kolmogorov axioms [35]:

$$\begin{cases} P(i) \geq 0 & \text{non-negativity} \\ \Sigma_i P(i) = 1 & \sigma\text{-additivity and unit measure} \end{cases}$$

with the caveat, regarding the σ-additivity, that the various classes must be mutually exclusive.

Like the sigmoid, the softmax activation can saturate and thus can cause learning difficulties if the loss function is not designed to compensate for this saturation. So objective functions that don't use a logarithm to undo the previous exponentiation will fail to learn when the argument of the `exp` becomes very negative, causing the gradient to vanish [72]. For instance, squared error is not a good loss function for softmax units, as opposed to negative log-likelihood or cross-entropy.

Moreover, to obtain a numerically-stable variant of the softmax, insensible to extreme input values, we can use

$$\text{softmax}(\mathbf{z} - \max_i z_i)$$

having noticed that the softmax output is invariant when adding the same scalar to all its input: $\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} + c)$.

7.8 Objective function: categorical cross-entropy

For a n-class classification problem, the obvious choice for the loss function is the *categorical cross-entropy*, which is a generalization of the binary cross-entropy.

The cross-entropy between two probability distribution P and Q is defined as

$$H(P, Q) = H(P) + D_{KL}(P||Q)$$

where

- $H(\mathrm{x}) = \mathbb{E}_{\mathrm{x} \sim P}[I(x)]$ is the Shannon entropy, also denoted $H(P)$, representing the expected amount of information in an event x drawn from the P distribution

- $I(x) = -\log P(x)$ is the self-information of an event $\mathrm{x} = x$
- $D_{KL}(P||Q) = \mathbb{E}_{\mathrm{x}\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{\mathrm{x}\sim P}\left[\log P(x) - \log Q(x)\right]$ is the Kullback-Leibler divergence, which is a non-negative quantity that quantifies a sort of distance between the two distributions P and Q. Notice that it is not a true distance because it is not symmetric: $D_{KL}(P||Q) \neq D_{KL}(Q||P)$

The expression for the cross-entropy thus become

$$H(P, Q) = -\mathbb{E}_{\mathrm{x}\sim P} \log Q(x)$$

that, in the discrete boolean case, means

$$H(P, Q) = -\sum_{i\in\{0,1\}} P(i) \log Q(i)$$

If $P(1) = y_{\text{predicted}}$ is the predicted probability of being a signal, we will have $P(0) = 1 - y_{\text{predicted}}$. So, the quantity to be minimized is

$$H(P, Q) = -y_{\text{predicted}} \log(y_{\text{true}}) - (1 - y_{\text{predicted}}) \log(1 - y_{\text{true}}) \tag{26}$$

where y_{true} is the true class label used in supervised learning.

Minimizing this quantity with respect to Q is equivalent to minimize the corresponding $D_{KL}(P||Q)$ because they are identical up to a constant. It's important to notice that, given the asymmetrical behavior of the Kullback-Leibler divergence, minimizing $D_{KL}(P||Q)$ or $D_{KL}(Q||P)$, both with respect to Q, does *not* give the same result. The former has the tendency to place high probability anywhere that the true distribution places high probability, while the latter has the tendency to rarely place high probability anywhere that the true distribution places low probability.

So, care must be taken when choosing which of the two we want to minimize: it depends on the specific problem and on what we want to achieve.

7.9 Stochastic gradient descent

Most learning algorithms involve some sort of optimization, via the minimization of the *objective function*, also called *loss function*, *cost function* or *error function*, depending on the field of application. This minimization can be done via *gradient descent* methods, which allow to find local minima and other critical points, such as saddle points of the loss landscape. Second-order methods are generally avoided due to their instabilities and their computational cost.

Ideally, we would like to arrive at a global minimum, but this might not be possible. A comparable-depth local minimum will perform nearly as well as the global one, thus being an acceptable halting point.

Pure optimization is a very difficult task: in Deep Learning we are interested in approximate minimization. Optimization algorithms may fail to find a global minimum when there are multiple minima or plateaux present. We generally accept such solutions even though they are not truly minima, as long as they correspond to significantly low values of the cost function. Moreover it was demonstrated that in the context of Deep Learning (when billions of parameters are involved) local minima are very rare: the wide majority of critical points in the cost landscape are saddle point. Therefore we settle for finding a value of the objective function that is very low but not minimal in any formal sense.

Apart from that, it's important to understand that the learning is an *indirect* optimization: what we really seek to minimize is the *validation accuracy*, but we have no access to it. As a consequence, we try to minimize another function, the *cost function*, hoping that this will lead us in the correct direction [72].

To minimize f, we would like to iteratively find the direction in which f decreases the fastest which, by definition, is the direction pointing opposite to the gradient. That's the *steepest descent* method.

The proposed new point is

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} - \eta \nabla_{\mathbf{x}} f(\mathbf{x})|_{\mathbf{x}_{\text{old}}} \tag{27}$$

where $\nabla_{\mathbf{x}} \equiv \frac{\partial}{\partial x_a}$ (in abstract index notation [51]) and η is a positive scalar determining the side of the step, called the *learning rate.*

We want to minimize the total cost function $L(\mathbf{x}, y; \theta)$, where $\mathbf{x}$ are the training samples, y the training labels and θ all the parameters of the network (from thousands to billions, depending on its size). To exploit parallelism over the various samples, the total loss function is often decomposed as the sum (or average) over training examples of some per-example loss function $L_N = \sum L_1$. For instance, the *mean squared error* objective function falls in this category. The computational cost of one single gradient evaluation is $\mathcal{O}(N)$, with N the number of samples in the dataset. This makes the use of a big dataset practically infeasible, despite using lots of data is the main way to obtain good generalization. Stochastic gradient descent solves exactly this problem.

Stochastic Gradient Descent is an extension of the gradient descent algorithm. Together with its various modern generalizations, such as *Adam*, it is the most used training algorithm in Deep Learning. The key point of the algorithm is the substitution of the exact gradient, computed on the whole dataset, with an expectation of it, computed on a much smaller dataset called the *minibatch*, After the substitution $\nabla L \leftarrow \hat{g}$, the gradient descent method is applied as in equation (27). The cost of one update of the weights does not depend anymore on the training set size.

There were many concerns in the past regarding the use of gradient descent, especially in nonconvex optimization. Today we empirically know that Machine Learning models work very well when trained with gradient descent. The gradient-based optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the cost function quickly enough to be useful [72].

7.10 Adam

There are many variants of the Stochastic Gradient Descent nowadays. For the optimization of our network we used the state-of-the-art *Adam* [29], which belongs to the class of adaptive learning rate algorithms. Its name is an abbreviation of “adaptive moments”. We have chosen it because of its robustness to the choice of hyperparameters and the efficient behavior of its default values. We also noticed that, compared to regular Stochastic Gradient Descent, in general the training can escape faster from difficult flat or saddle regions, avoiding to remain stucked for a lot of time. This turns out in a faster convergence speed.

Given the complexity of the algorithm, we will not provide a detailed explanation, sending the interested reader directly to the original paper [29].

7.11 Minibatch size

The size of our model is constrained by the amount of graphical memory of our GPU (Nvidia Tesla K20 with 5 GB of graphical memory) and by the size of the minibatch we want to use during the training.

As we said, the bigger and deeper the model, the better the classification accuracy (with some important caveats regarding overfitting and training time). Moreover, the bigger the minibatch size, the smoother the training: a bigger minibatch size means higher statistical accuracy in the gradient evaluation of the objective function we want to minimize.

Although a bigger minibatch is always desirable, values higher than 128 should be avoided when the training is carried on a single GPU because stochastic gradient descent algorithms are designed to work in the small-minibatch regime [72].

Given the small number of parameters in our model and the compact size of our images, we were able to use a minibatch of 128 images for every update of the weights (that is: for every *iteration* in the learning phase).

7.12 Supervised and unsupervised learning

The learning algorithms can be divided into two major subclasses: *supervised* and *unsupervised*. In supervised learning there is a teacher (the supervisor) that knows the correct answer, for example the correct class which the image belongs to. This information is usually called the *target* or the desired output. Comparing the desired output with the actual predicted output of the neural network we evaluate what mistake has been made (if any) and propagate this information to the whole network. The exact procedure will be explained in every detail later.

This approach requires a preexisting correct label for every image. But labeling data is in general a human-made procedure that can be very expensive and time-consuming. The great majority of the data available are unstructured: the photographs you shoot do not contain a label of who is portrayed, the music on the radio does not contain the information of which notes are being played, the streams of LIGO/Virgo detectors do not contain a label telling us if a signal is present and which class it belongs to.

Sometimes those informations can be recovered with different strategies, often making some compromises:

- for free: we can use humans themselves, perhaps inducing some social behavior that pushes people to manually tag their photos spontaneously, as happens in many social networks or in the Gravity Spy project [30]. However, this labels can be inaccurate and incomplete.
- almost for free: we can use machines to run montecarlo simulations to obtain the label relative to some fake signals injected into the LIGO/Virgo stream. However, we have no way to previously know the label of the real signals in the data.
- expensive: acquiring a labeled dataset made by someone else. That's also why "data is the new oil".

All of these approaches are difficult and expensive in terms of money or human work.

The unsupervised learning solves exactly this problem: it deals with unstructured data trying to organize them finding common patterns. The clustering algorithms and the principal component analysis are examples of this kind of procedures. The prototypical unsupervised neural architecture is the *autoencoder* [31], where the learning algorithm tries to find a compressed representation of the data as lossless as possible, which is called a *sufficient statistics*. This is done by minimizing the difference between the raw input and output reconstructed after the compression-decompression stage. This operation can also be carried out in a hierarchical fashion, with an architecture called *stacked autoencoders*.

Of course, supervised learning is much more effective than unsupervised learning: for humans, learning Quantum Mechanics from a book requires more or less one year, while discovering it from scratch (with no books) required more or less one century.

Unsupervised learning can thus be used as an aid for supervised learning, for example when most of the data are unstructured but there are still some labels available. This approach is called *semi-supervised.* Another common usage in the past was to use the unsupervised learning to provide a good initialization point for the following supervised stage [32, 33].

In our work we will only make use of supervised learning, obtaining the class labels from numerical simulations. This makes our approach model dependent because the real signals will only be recognized if they will match the shape of the fake injected signals. However, this is not completely true, because we have empirical evidence that modern artificial neural networks are able to generalize fairly well. They are robust by construction and their generalization ability is a major concern during their architectural design, as was explained in the section about the regularization procedures.

Neural networks are good at generalizing, but they cannot do miracles: a tennis player will be easily able to start to play ping-pong in an acceptable way within minutes, but he won't be able to play football in such a good way. Similarly, a neural network trained to recognize linear transients lasting a couple of days will be easily able to also recognize previously-unseen transients with a curved shape in the time-frequency plane, but will obviously be completely unsuitable to recognize short millisecond bursts.

7.13 Backpropagation and weight update

A *supervised learning* procedure consists of iteratively updating the parameters of the network (the *weights* and *biases*) using the informations gained from the evaluation of the data in the *training set* and the comparison with the known desired output.

The training set is divided into chunks: each one is a *minibatch* and every computation on a minibatch is called an *iteration* of the algorithm. In every iteration, we firstly compute the predictions of our model for the data in the minibatch (this is called the *forward-computation step*) and compare them with the true class labels we want to successfully reproduce. The difference between the desired and predicted output is the *error*. The information about the errors made at a given iteration must be sent to all the weights in the network, so that hopefully the same error won't be repeated in the future. This is done by providing some kind of metric for the error that has been done (the *loss function*) and trying to minimize this function following its descending-gradient direction, in order to minimize the future errors. This is called *backward-propagation of errors*, or simply *backpropagation*.

The update will be performed for every synaptic weight in every layer of the network. Instead of "weights and biases", the term "weights" is widely used for the sake of simplicity. The update at iteration t is computed with the gradient of the loss function computed on the weights of the previous iteration $t-1$.

But how is the gradient evaluated? How is the backpropagation implemented, to update the weights of the neural network? Given the difficulty to find the answers on the web, we will provide the detailed derivation of the backpropagation rule. We will do it in the assumption of a feedforward fully-connected network, with sigmoid nonlinearities and mean squared error loss function. So, for every fully-connected layer, we have

$$y = \varphi(w \cdot x + b) \equiv \varphi(s) \tag{28}$$

where y is the neuronal output of the layer, x the neuronal input, w the matrix of synaptic weights, b the scalar bias and φ the sigmoid activation function, defined as

$$\varphi(s) = \frac{1}{1+e^{-s}} \tag{29}$$

where its derivative has the nice property

$$\varphi'(s) = \varphi(s)(1-\varphi(s)) \tag{30}$$

We will start our derivation from the iterative weight update rule (equation 27), already seen in the section about the stochastic gradient descent algorithm (section 7.9)

$$\Delta w[t] = -\eta \nabla_w L(w)|_{w[t-1]} \tag{31}$$

where

- $\Delta w[t] = w_{\text{new}} - w_{\text{old}}$ is the correction to the weights w done at the iteration t. Being the learning algorithm sequential, the index used for the iterations is a discrete-time index.

- $L(w)$ is the loss function relative to that weight configuration. In the following derivation we will use the *mean squared error* as our loss function

$$L = \mathbb{E}[e^2] = \frac{1}{N}\sum_{i=1}^{N}\left(d - f(x_i;w)\right)^2 \tag{32}$$

  where e is the *error*, defined as the difference between the desired output d and the predicted output $f(x_i;w)$, where x_i is a single raw input of the minibatch, w is the weight configuration, f is the function implemented by the neural network and N the number of samples in the minibatch.

We now rewrite equation (31) explicitating all the indices (and suppressing the temporal step)

$$\Delta w_{ad}^{(l)(l-1)} = -\eta \frac{\partial L}{\partial w_{ad}^{(l)(l-1)}}$$

where $w_{ij}^{(l)(l-1)}$ are the $n \times m$ fully-connected weights between the layer l with n neurons and the previous layer $l-1$ with m neurons. Notice that the gradient of the scalar loss function is done with respect to a tensor. To compute it, we must apply the derivative chain rule

$$\frac{\partial L}{\partial w_{ad}^{(l)(l-1)}} = \frac{\partial L}{\partial x_a^{(l)}}\frac{\partial x_a^{(l)}}{\partial s_a^{(l)}}\frac{\partial s_a^{(l)}}{\partial w_{ad}^{(l)(l-1)}} \tag{33}$$

Notice that the expression above is not rigorously correct in Einstein notation: the indices are everywhere the same. It's for the sake of simplicity: in the complete calculation all the terms with different indices will result in a Kronecker δ.

Using the equation 28 we obtain

$$\frac{\partial s_a^{(l)}}{\partial w_{ad}^{(l)(l-1)}} = x_d^{(l-1)}$$

$$\frac{\partial x_a^{(l)}}{\partial s_a^{(l)}} = \varphi'(s_a^{(l)})$$

where φ' can be easily obtained from the property (30), which we repeat is true only for the sigmoid function.

So at this point we only need to compute the derivative of L to close the chain in expression (33). Following the equation (32), we can provide an exact result for only the last layer L of the fully-connected feedforward neural network. It turns out that the other values must be computed iteratively, again using the chain rule

$$\frac{\partial L}{\partial x_a^{(l)}} = \begin{cases} -2(d_a - x_a^{(L)}) = -2e_a & \text{if } l = L \\ \frac{\partial L}{\partial x_e^{(l+1)}} \frac{\partial x_e^{(l+1)}}{\partial s_e^{(l+1)}} \frac{\partial s_e^{(l+1})}{\partial x_a^{(l)}} & \text{if } l < L \end{cases}$$

where, again using equation (28), we can write

$$\frac{\partial x_e^{(l+1)}}{\partial s_e^{(l+1)}} = \varphi'(s_e^{(l+1)})$$

$$\frac{\partial s_e^{(l+1)}}{\partial x_a^{(l)}} = w_{ea}^{(l+1)(l)}$$

We have thus a well defined iterative way to compute the effect of the gradient in every layer of the network, called the *delta rule* method. In implicit tensor notation (one underline for vectors and two underlines for matrices) this rule can be summarized in the following concise way

$$\frac{\partial L}{\partial \underline{\underline{w}}^{(l)(l-1)}} = \underline{\delta}^{(l)} \otimes \underline{x}^{(l-1)}$$

$$\underline{x}^{(l-1)} = \begin{cases} -2\ \underline{e}\ \varphi'(\underline{s}^{(l)}) & \text{if } l = L \\ \underline{\delta}^{(l+1)} \cdot \underline{\underline{w}}^{(l+1)(l)} & \text{if } l < L \end{cases}$$

where the δ is just a symbol used for historical reasons, *not* the Kronecker δ

$$\delta_a^{(l)} \equiv \frac{\partial L}{\partial x_b^{(l)}} \frac{\partial x_b^{(l)}}{\partial s_a^{(l)}}$$

In conclusion, we have found a well defined procedure to perform the weight update via stochastic gradient descent. This *iterative* procedure is called *backpropagation* and must be done for every iteration. Minibatch after minibatch, the full training set will be evaluated. The evaluation of the entire dataset is called an *epoch*. After one epoch is concluded, the training set could be shuffled and the entire procedure repeated again and again. It will require many many epochs for the algorithm to reach convergence. Ideally, we want to reach an absolute minimum of the cost function: this means to find the optimal configuration w^* of the weight such that

$$w^* = \underset{w}{\operatorname{argmin}}\, L(w)$$

so that

$$L(w^*) \leq L(w) \quad \forall w$$

As said before, this is in general practically impossible. So we will be happy with low-enough solutions, that will provide in a reasonable amount of time a good-enough classification performance on the validation set.

7.14 Initialization

Weight initialization is probably one of the most important aspects of the learning. Deep Learning training algorithms are usually iterative and their convergence to acceptable solutions in an acceptable amount of time strongly depends on initialization strategies. The initial point can determine whether the algorithm converges at all. When learning does converge, the initial point can determine how much time is required and how low the final loss will be. Moreover, final points of comparable cost can have wildly varying generalization error, so the initial point will also affect generalization [72].

Neural network optimization is not yet well understood and modern initialization strategies are determined heuristically. Most of current initialization procedures use random values for the weights, trying to break symmetry between different units to avoid useless redundancies. Usually, the values are drawn from a gaussian or uniform distribution. The choice of the distribution does not seem to matter much; however, the scale of the initial distribution has a large effect on both the optimization outcome and the ability of the network to generalize.

Larger initial weights will yield a stronger symmetry-breaking effect and will also avoid losing signals during the prediction and back-propagation phase: larger matrix values will result in larger outputs after the matrix multiplication. However, if the weight values are too big, the exploding output can yield a saturation of the activation function, causing the gradient to vanish. These competing factors determine the ideal initial scale of the weights to a given layer.

Among all the currently available initialization strategies, we decided to use the *normalized initialization* [28]

$$W_{ij} = \text{uniform}\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right) \tag{34}$$

where W_{ij} are the weights of a fully-connected layer of m inputs and n outputs. It was heuristically designed as a compromise between the goal of having the same activation variance and the goal of having the same gradient variance for all layers, assuming a model with no nonlinearities. Numerical evidences show that this strategy performs reasonably well also with modern deep architectures, despite them being highly nonlinear. The interested reader can look at the original article for further details [28].

All the *bias* terms were instead initialized to zero: this choice is compatible with most of the initialization schemes[72].

7.15 Network overview

In the previous sections we reviewed every separate piece of the convolutional architecture, trying to give an insight of what's going on under the hood and briefly discussing hyperparameter tuning. In this section we will sketch the structure of our model, showing how the different pieces are assembled together.

Being a proof-of-concept, we tried to keep the design of our network as simple as possible [77], being minimalistic while still having enough expressive power to perform the given task [71].

A graphical scheme of the network is pictured in figure (28). The model definition, the training phase and the validation one were all carried out using the *TFlearn* [78] and *Keras* frameworks [79], both based on the *TensorFlow* backend [80].

Our input is made by 128 RGB images, each one made of 256x128 pixel. $N = 128$ is the *minibatch* size, as was discussed earlier in section (7.11).

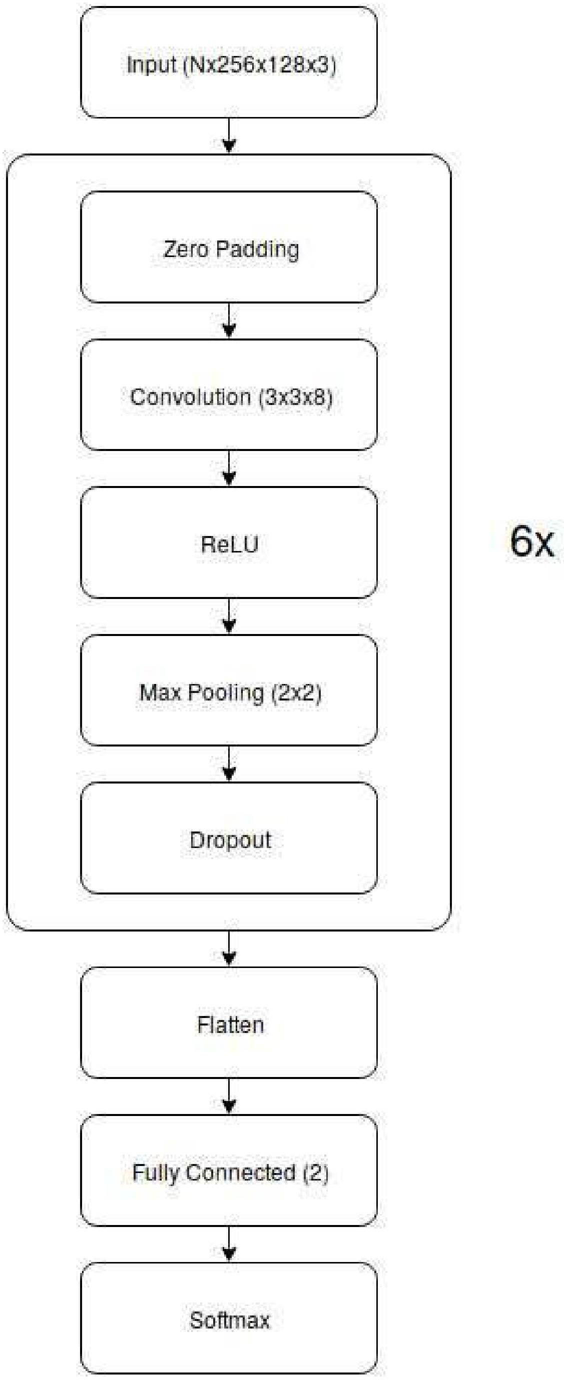


Figure 28: Sketch of the architecture of our model.

Then there is a sequence of 6 simple *convolutional blocks*, each one composed as follows:

- `ZeroPadding2D`
- `Convolutional2D`
- `ReLU`
- `MaxPooling`
- `Dropout`

where the *zero padding* ensures that the size of the following convolutional output is still a power of two. Every *convolutional* layer in the network has the same kernel size (3x3) and number of filters (8). This is not a common practice, because in general the effective kernel size widens and the number of filters increases as the network gets deeper. This choice was discussed in section (7.2). Every convolutianal unit has a *ReLU* activation function. After the nonlinearity, a *max pooling* is performed: the contraction size of 2x2, which is the minimum possible, means that the feature map area is shrunk by a factor of four after the pooling. The final step of every block is a soft *dropout* ($P = 0.8$), to regularize the model and avoid overfitting. We didn't use any other type of regularizer, such as L1 or L2 penalties.

Despite the use of *batch normalization* is strongly encouraged in convolutional networks, we avoided these type of layer at the current stage of model development, because we notice that we are not able to fully control the learning boost they provide. We need more data to control overfit, a better initialization strategy and a bigger minibatch size (which could be achieved using multiple GPUs).

After the *flattening* layer, we placed a *fully connected* layer with only two output neurons, one for every class. Those neurons have a *softmax* activation function, in order to obtain class probabilities as the final output of our classifier.

The loss function we used is the *categorical cross-entropy* and the optimizer was *Adam*, which is a modern variation of stochastic gradient descent.

7.16 Parameter counting

Artificial neural networks have a huge number of internal parameters to adapt during the learning phase. In general, the more parameters involved the more expressive the network is. More expressive power means better ability to perform a given task, but also means longer and more difficult training, as well as more computing resources needed.

With our simple architecture we have a total of 4088 trainable parameters. This number is a tiny fraction of what's actually the state-of-the-art in the field of image recognition: modern ultra-deep networks have millions of parameters and take weeks to be trained on massive GPU clusters.

But this number is enough to represent the knowledge required to successfully perform our classification task. Given the narrowness of our datasets, a substantially higher number of parameters can easily lead to overfitting even if an aggressive regularization strategy is used.

For each 2D convolutional layer the number of parameters is

```
input_filters * kernel_width * kernel_height * output_filters + output_filters
```

while for each fully connected layer the number of parameters is

```
input_neurons * output_neurons + output_neurons
```

where the additions take into account the number of *bias* parameters. All the other layer types do not have any trainable parameter.

The results are summarized in the table (1).

Layer	**Output Shape**	**Parameters**
Input	(minibatch, 256, 128, 3)	0
Convolution2D	(minibatch, 256, 128, 9)	252
ReLU	(minibatch, 256, 128, 9)	0
MaxPooling2D	(minibatch, 128, 64, 9)	0
Dropout	(minibatch, 128, 64, 9)	0
Convolution2D	(minibatch, 128, 64, 9)	738
ReLU	(minibatch, 128, 64, 9)	0
MaxPooling2D	(minibatch, 64, 32, 9)	0
Dropout	(minibatch, 64, 32, 9)	0
Convolution2D	(minibatch, 64, 32, 9)	738
ReLU	(minibatch, 64, 32, 9)	0
MaxPooling2D	(minibatch, 32, 16, 9)	0
Dropout	(minibatch, 32, 16, 9)	0
Convolution2D	(minibatch, 32, 16, 9)	738
ReLU	(minibatch, 32, 16, 9)	0
MaxPooling2D	(minibatch, 16, 8, 9)	0
Dropout	(minibatch, 16, 8, 9)	0
Convolution2D	(minibatch, 16, 8, 9)	738
ReLU	(minibatch, 16, 8, 9)	0
MaxPooling2D	(minibatch, 8, 4, 9)	0
Dropout	(minibatch, 8, 4, 9)	0
Convolution2D	(minibatch, 8, 4, 9)	738
ReLU	(minibatch, 8, 4, 9)	0
MaxPooling2D	(minibatch, 4, 2, 9)	0
Dropout	(minibatch, 4, 2, 9)	0
Flatten	(minibatch, 72)	0
Fully Connected	(minibatch, 2)	146
Softmax	(minibatch, 2)	0

Table 1: Output shape and number of parameters for each layer in our model

8 Training

In this section we will describe the training phase and its optimization, detailing all the strategies we used in order to obtain a satisfactory validation performance.

8.1 Train-test split

The first operation to do before the training is to split the data into two different sets: the *train set* and the *test set*. The model will be trained only using the informations from the train set. The test set will never take part to the gradient descent algorithm: it will instead be periodically evaluated by the model, to gain a feeling of how the classifier is performing on a completely new and independent dataset. The informations about this evaluation will then be immediately discarded, without using them for the training: in this way, every successive evaluation of the test set will be like the first evaluation of a never-seen-before dataset.

To summarize: the train set is used to feed the model, while the test set is used to probe its generalization capabilities. Testing its ability to generalize is essential to be sure that the model is not overfitting, so to check that it is effectively learning general features and not accidental correlations occurring in our finite-sized dataset.

When a model has a good generalization property, its performances on the train and test sets will be roughly the same: it should be able to successfully classify a dataset never seen before. The overfitting regime has a distinctive signature: the train performances are much better than the test performances because in the test set all those accidental correlations are lost. So it is essential to periodically evaluate the test set during the training, in order to understand if the neural network is beginning to overfit. The two curves of train and test loss functions versus time should roughly overlap. The learning process is stochastic, so that the two curves will be noisy and we need to check that they are compatible with respect to their statistical uncertainties.

To do the comparison easier and faster, it is preferable to have nearly the same uncertainty (statistical fluctuations) for the two sets. This requires that they have to contain the same number of samples. The original dataset will be divided into two equal parts: the two sets will be populated randomly, taking care to preserve in each one the original ratio between the two classes (which is here 50:50).

Following this procedure, we are halving the dataset we effectively own, because only the "train" half will be used for the training of the model, while the "test" half will never directly participate in the training. This unavoidable halving of the available data can be a problem if our original dataset is not so large, because deep neural networks need a lot of samples to be successfully trained. There are different approaches to overcome this problem: one of them is called *data augmentation*.

8.2 Data augmentation

Data augmentation is a way to increase the effective size of your dataset: every input image is preprocessed according to some pre-determined transformations, such as horizontal or vertical flip, rotation by some angle, zooming, padding, addition of noise, etc. This helps to prevent overfitting, because the network sees a slightly different image every time.

However, we remind that care must be taken when using data augmentation because it can introduce in the network unwanted invariances. For example, a horizontal up-down flip will make an image labeled as "nine" to appear as a "six" and viceversa, thus merging together the two classes and introducing a dangerous conceptual error. In a similar way, an up-down flip of our RGB spectrograms will transform a spindown in a spinup, thus potentially reducing the noise-rejection power of the classifier. Indeed, in our model, none of the injected signals can have a spinup, while the noise disturbances do.

Unfortunately, we haven't used any data augmentation due to software bugs in the TFlearn library: some software routines were always crashing. We know that TFlearn is a young library and it's still under heavy development, so perhaps those bugs will be fixed in the very near future. However, our dataset is still large enough to allow a successful training of the network, even when halving the data for the train-test split. To be sure to avoid any kind of overfitting, we used many dropout layers in our model, as was discussed in the section regarding the design of the architecture.

8.3 Comparison with humans

When we initially started the training trials, the loss curve was remaining flat for tens of epochs: we weren't able to make the learning starting to converge. This is unusual for such a small network.

Recognizing a tiny horizontal line in a forest of colorful vertical lines (figure 18) may be a task too hard to begin with, in particular if the concept of line is not yet developed. If a problem is too difficult, the learning procedure is unlikely to ever converge. To probe the effective difficulty of the problem, we decided to make a human case study: humans surely have the concept of line, but they are not used to recognize signals inside an RGB spectrogram.

We selected some random students in a library, with the only bias that they should not belong to a scientific faculty (because the artificial neural network hasn't the concepts of noise, signal and spectrogram).

We asked them to do exactly what the network does: see an image, read its boolean label ("yes" or "no"), try to understand the logic behind the label assignation, go to the next image and repeat the procedure. This loop will continue until the inner logic is clear enough to be able to do a confident prediction for the next image. We didn't give the students any other information about the nature of the images: only images and yes-no labels. In addition, we asked them to describe how their thoughts were evolving, what they were thinking.

Despite we have tested only less than five students, the results were clear: all of them can complete the task within 60-70 images. All of them were firstly trying to find a pattern in the colored vertical lines, which were the most evident pattern (figure 18). No one noticed the tiny white horizontal line before the 30th image. After realizing that the vertical patterns were not significant, they switched their attention to other features in the image, fastly finding the small horizontal line. After ~ 70 images they were all able to correctly predict the following image with certainty (the injected signal intensity was the biggest possible).

So we concluded that the task could be successfully done and is not overwhelmingly difficult in principle: we just need to wait longer for the beginning of convergence or we have to make clearer that we are interested in the horizontal lines.

Following this last intuition, we started injecting dummy signal-like shapes with thicker horizontal lines. The network effectively learned to correctly classify them in a limited number of iterations: we needed ~ 30 epochs to begin to significantly decrease the cost function. Then, we decreased the line thickness progressively, tracking the time required to escape from the initial plateau of the cost function. In the end, with one-pixel-wide lines, the cost function was beginning to decrease around epoch 70. This was a direct confirmation of what we thought: the problem is not too hard and the learning process is able to converge, but it will require quite a lot of time.

Now we have to figure out a way to improve the learning phase and how to speed it up, at least in the initial stage. Having in mind this goal, e tested the *transfer learning* approach.

8.4 Transfer learning

Transfer learning is a technique consisting in the initialization of the network's weights with the weight deriving from another network that has previously learned to solve a task which is somehow related to the one we aim to solve. It's pretty intuitive: knowing how to distinguish a square from a circle can be a useful skill if we aim to distinguish a car from a horse, just because the horse has no wheels attached.

We noticed that our network, initialized with the weight used to recognize thicker lines, was fastly able to learn to recognize slimmer lines. The escaping of the initial cost plateau characterizing the early epochs of the training (the "I don't really know what to do" plateau) was ~ 5 times faster with respect to the random initialization case: from ~ 70 to ~ 15 epochs. This speedup is probably caused by the lost predominance of the vertical lines in the initial stages, which were the first cause of confusion, as our human experiments were telling us.

So, for all the succeeding training trials, we used this dummy pretrained weights to re-initialize our network, so to be able to begin the learning much faster when using the real dataset. Thanks to this speedup, we were able to do much more training trials than was previously possible, allowing us to fine-tune better the other parts of the network.

8.5 Curriculum learning

Transfer learning is not the only way to speed up the training: if we want to learn how to perform a difficult task, it's often recommended to begin to learn starting from an easier version of that task. For instance, if we want to be able to distinguish between a dog and a wolf, it's recommended to start with the easier distinction between dogs and cats or dogs and tigers. This is called *curriculum learning* [83]: it is based on the idea of planning a learning process, beginning by learning simple concepts and progress to learning more complex concepts that depend on these simpler concepts. This procedure has also proven to be effective when training animals. Moreover, it tightly reflects how humans teach: teacher start by showing easier and more prototypical examples and then help the learner to refine the decision surface with less obvious cases.

Curriculum learning is a procedure belonging to the class of *continuation methods*, where the training starts with easier cost functions and progressively goes toward the most difficult one. Easier cost functions are obtained by blurring the harder ones, so to avoid the difficulties induced by local nonconvexities in the optimization procedure. The easier cost functions can thus eliminate long-lasting flat regions and decrease variance in gradient estimate, making local updates easier to compute and improving the correspondence between local update directions and the direction towards the global solution [72].

For this reason we decided to train our model using different datasets with decreasing signal intensities: 32, 16, 8, 4, 2, 1. The logic of the algorithm is reported below, written in pseudocode:

```
if it's the first initialization ever
    initialize the weights randomly
otherwise
```

```
loop over all signal intensities in descending order
    if it's the biggest signal intensity
        load the dummy weights from the thick-line training
    otherwise
        load the weights obtained from the previous signal intensity
    initialize the model with these weights
    load the dataset for the given signal intensity
    train the model on this dataset until convergence
    save the resulting fine-tuned weights on a separate file
```

It's important to notice that the curriculum learning appears to be strictly sequential: if a needed skill is missed at a given stage, it is unlikely that it will be recovered (or discovered) at a later stage. That's why it's crucial to correctly classify as much images as possible at a given signal intensity before the passage to a smaller one. That's mainly true at the beginning of the training, with loud signals, where a human is capable of correctly classify all the images without much effort.

That's why we created a dedicated *callback* to stop the training only when the classification accuracy is 100%.

8.6 Perfect accuracy

A callback is an operation that is performed on a regular basis during the training, for example at the end of every epoch. The most commonly-used callbacks are the evaluation of some metrics to gain a feeling of the running performances of the model or the check of some conditions to stop the training or modify its flow.

Since we noticed that our classification task can be straightforwardly done by humans, at least at high signal intensities, we wanted to be sure that our neural network also reaches the 100% accuracy level for that datasets. The accuracy is defined as the fraction of correctly-classified samples

$$\text{accuracy} = \frac{\text{correctly-classified samples}}{\text{all samples to classify}} \tag{35}$$

So we implemented a callback that periodically checks if the training accuracy is 1 and stops the training only if this condition is satisfied (at least at high signal intensities). This will guarantee the proper development of useful skills at the right time during the curriculum learning, as said before.

In principle, it could be worth to continue the training even if the level of 100% accuracy has been reached, because it can contribute to increase the separation between the two classes, making the classifier more reliable and robust. In practice, we noticed that the gradient information is so low that the learning could start to behave like a driftless random walk, with the risk to lose what we have obtained so far. The low level of information available can also cause overfit. To avoid this risk, we stopped our training at the first realization of the accuracy condition.

8.7 Early stopping

Early stopping is a simple and effective regularization technique: the training is stopped when the test classification error stops to decrease, following the train one, and starts to increase again, departing from the train curve. The training is thus stopped when the two curves do not overlap anymore (figure 29). We remind that the classification error is defined as

$$\text{classification error} = \frac{\text{misclassified samples}}{\text{all samples to classify}} = 1 - \text{accuracy} \tag{36}$$

In the overfitting regime the train error is much lower than the test error. So the early stopping prevent the model to enter the overfitting regime: that's why it is a regularization technique. Moreover, it's a form of regularization that can be used without damaging the learning dynamics at all.

After many trials, we didn't find a satisfactory metric that is able to correctly quantify the separation between the two learning curves of train and test, so to place a separation threshold to decide when the training must stop. The choice was done on the base of the experience gained from all the past training trials. Further investigations are needed to be able to fully automatize this important part of the training.

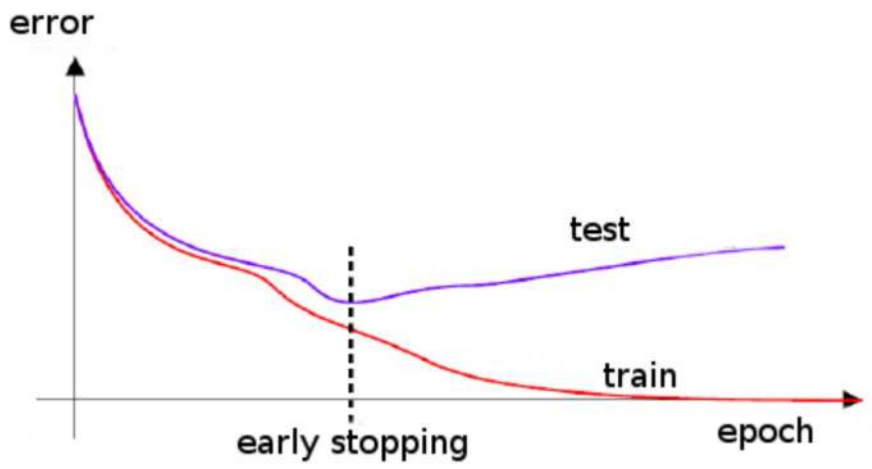


Figure 29: The early stopping technique: the training is stopped when the train and test curves begin to separate.

9 Validation

The model validation is the benchmark of the performances of the neural network on some completely new datasets. In the following, the network configuration used was the one obtained from the training on the dataset with the smallest signal intensity, after all the curriculum learning.

9.1 The threshold choice

The model is used to make predictions on these new datasets and those predictions are compared with the true labels in order to acquire informations on the model performances.

The two true classes of "noise" and "signal+noise" are encoded as 0 and 1, which are discrete numbers. The output of the classifier is instead a real number: the probability of the image to be a signal. Thus, we have to define a threshold to know where to separate the two classes.

This threshold has to be chosen carefully, looking for a trade-off between the false positives and the false negatives. The main constraint is to minimize the signal dismissal, even if this implies to include some noise in the form of false positives.

Those false positives can be later flagged and removed by the follow-up analysis.

In principle, the threshold could depend on the amplitude of the signals we want to investigate. We have used four validation dataset, with decreasing signal intensities 4, 2, 1.5, 1. The definition and interpretation of those values were previously discussed in section (5.9). The following figures (30, 31, 32, 33) show the histograms of the output of the classifier, where the events are divided according to the true class they belong to.

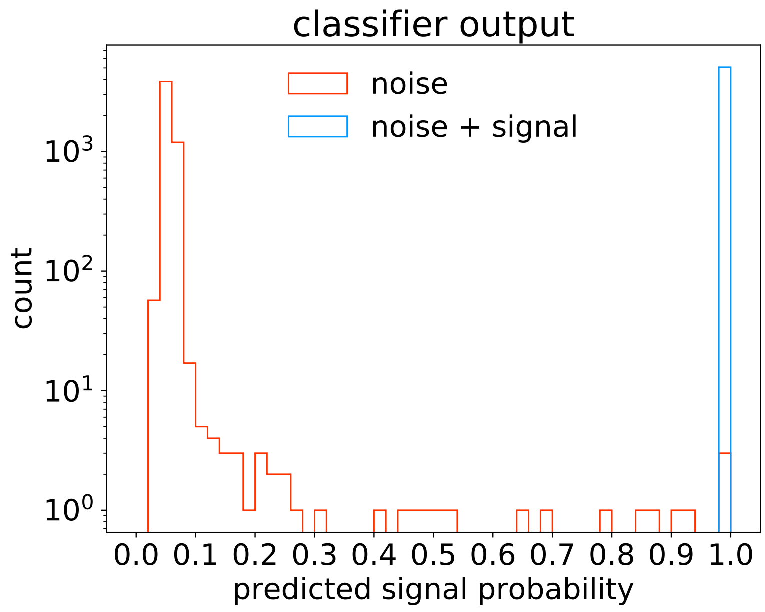


Figure 30: Output of the classifier for the validation dataset with signal intensity 4. The proposed threshold is $P_{\text{signal}} \sim 0.95$, as explained in the text.

From figure (30), at signal intensity 4, we note that the classifier has the absolute certainty about an event being a signal, so we can ideally place the threshold at $P_{\text{signal}} \sim 0.95$. This absolute certainty tells us that this task is too easy to be interesting: we are dealing with huge signals and our aim is to explore the boundary of what can be detected.

In figure (31) and (32), with signal intensity 2 and 1.5 respectively, the threshold can be placed somewhere near $P_{\text{signal}} \sim 0.30$ and $P_{\text{signal}} \sim 0.15$. From the figures we note that the task is more difficult now. In figure (32), despite the rough symmetry of the two classes, we have chosen an asymmetrically low threshold to

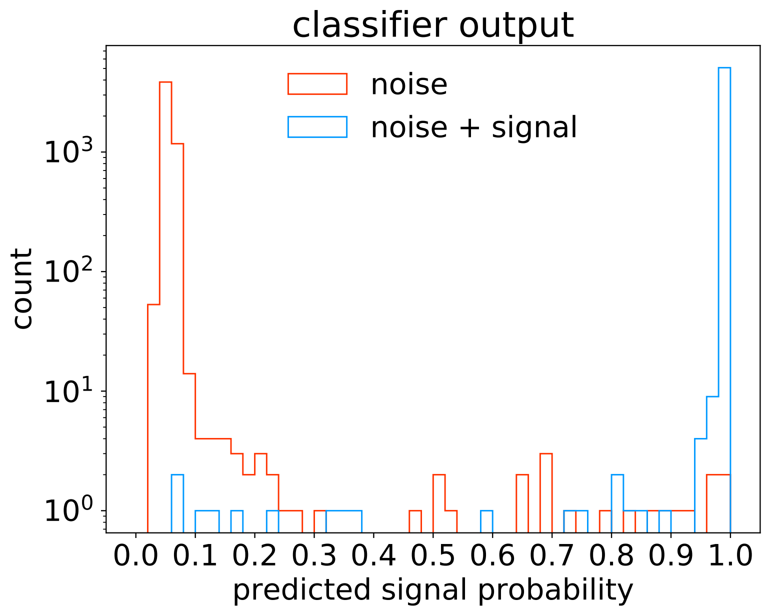


Figure 31: Output of the classifier for the validation dataset with signal intensity 2. The proposed threshold is $P_{\text{signal}} \sim 0.30$.

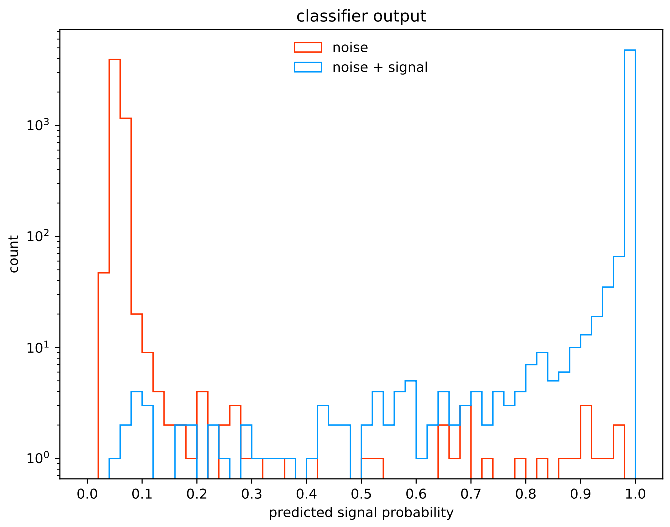


Figure 32: Output of the classifier for the validation dataset with signal intensity 1.5. The proposed threshold is $P_{\text{signal}} \sim 0.15$.

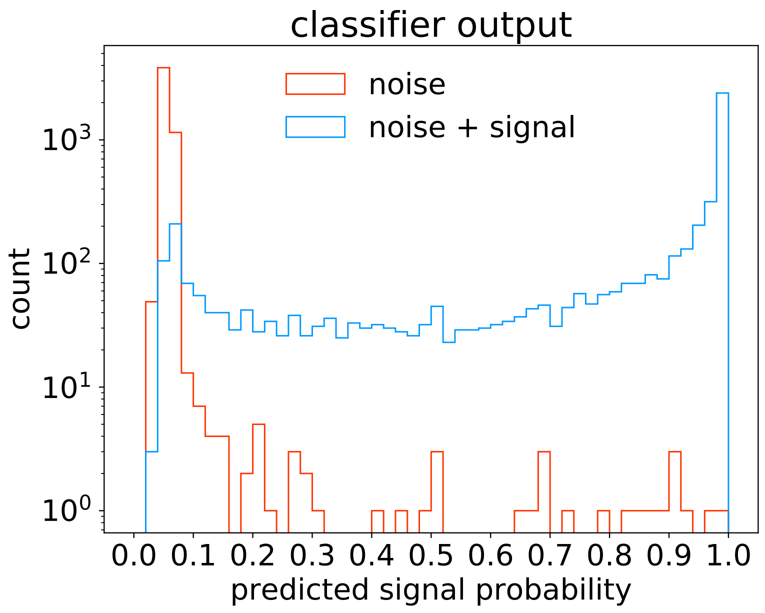


Figure 33: Output of the classifier for the validation dataset with signal intensity 1. The proposed threshold is $P_{\text{signal}} \sim 0.15$.

gather as much signals as we can.

In the case of signal intensity 1, shown in figure (33), the task is really difficult and the concept of noise is less vague than the concept of signal (the exact opposite of what happened in figure (30)). Since the histogram is in logarithmic scale, our threshold needs to be near $P_{\text{signal}} \sim 0.15$ or even just above 0.10: as close as possible to the noise cluster.

So far we have shown that the optimal threshold for our needs is a function of the incoming signal intensity, but we have to marginalize with respect to this quantity, because we don't know the amplitudes of the signal that we will find in our detectors. Since we expect the faint signals to be more frequent than the louder ones, we decided to let them dominate the marginalization, resulting in a threshold $P_{\text{signal}} \sim 0.15$.

We note that the choice of the threshold is itself a sort of clustering and classification problem, so it can in principle be automatically done exploiting another neural network. However, at present, we don't implement such a complicated task.

Now we have a way to round our predicted results to 0 or 1, so to be able to compare them with the true class labels.

9.2 Confusion matrix

The confusion matrix is a way to fully summarize the performances of a binary classifier. It is built sorting the results according to the following table:

	predicted noise	**predicted signal**
real noise	true negative (TN)	false positive (FP)
real signal	false negative (FN)	true positive (TP)

These four matrix elements are the building blocks to define the *metrics* we will use to benchmark our model. From now on, the elements of the matrix will be abbreviated as TN, FP, FN, TP.

9.3 Metrics

Let us define some metrics. We refer to the equation (35), which defines the *accuracy*. This can be rewritten as

$$\text{accuracy} = \frac{TP + TN}{TN + FP + FN + TP} \tag{37}$$

the *classification error* (equation 36) thus becomes

$$\text{classification error} = \frac{FP + FN}{TN + FP + FN + TP} = 1 - \text{accuracy} \tag{38}$$

We define two other quantities: *efficiency* and *purity*, which in Machine Learning jargon are called *recall* and *precision* respectively. The efficiency is the number of events selected as signal among all the events which are real signals. It measures how many desired events we succeeded to capture.

$$\text{efficiency} = \frac{TP}{TP + FN} \tag{39}$$

The purity is the number of real signals among all the events classified as signal. It quantifies how much the selected stream is composed by what we really want.

$$\text{purity} = \frac{TP}{TP + FP} \tag{40}$$

Applying the correct normalization we also have

$$\text{rejected noise} = \frac{TN}{TN + FP} \tag{41}$$

$$\text{false alarms} = \frac{FP}{TP + FP} = 1 \text{ - purity} \tag{42}$$

$$\text{missed signals} = \frac{FN}{TP + FN} \tag{43}$$

$$\text{selected signals} = \frac{TP}{TP + FN} = \text{efficiency} \tag{44}$$

TN+FP are all the real noises and TP+FN are all the real signals. We note that the percentage of selected signals coincides with the efficiency, by definition.

Now we can compute all those metrics for all the validation datasets with different signal intensities, so to show in a quantitative way how the task gets harder lowering the signal intensity.

9.4 Results

The results are summarized in table (2), where we report the performances of the classifier at different signal intensities. Every validation set is composed of 10240 image, so that the statistical uncertainties are negligible ($< 0.05\%$) and will not be reported in the table.

We note that we are able to reach $> 90\%$ efficiency and $< 1\%$ false alarm with signal intensity 1. As explained in section (5.9), this means that we are able to recover signals that are otherwise excluded from the current analysis with peakmap threshold $R = 2.5$. In the following section we will highlight the key points of this pipeline, because our future aim is to merge our algorithm with it, allowing a faster and deeper all-sky hierarchical search of gravitational-wave signals.

signal intensity	**rejected noise** (%)	**false alarms** (%)	**missed signals** (%)	**selected signals** (**efficiency**) (%)	**purity** (%)	**accuracy** (%)
1.0	99.25	0.81	9.66	90.34	99.19	94.78
1.5	99.29	0.73	0.20	99.80	99.27	99.54
2.0	99.32	0.68	0.08	99.92	99.32	99.62
4.0	99.44	0.57	0.00	100.00	99.43	99.72

Table 2: Performances of the classifier, evaluated at different signal intensities. Every validation set utilized here is composed of 10240 images: the statistical uncertainties of the percentages reported in the table will thus be neglected. With the smallest signal intensity we are able to reach $> 90\%$ detection efficiency and $< 1\%$ false alarm.

10 Future developments: hierarchical pipeline and follow-up

The trigger we developed, using our neural network classifier, will be the first step of the future hierarchical pipeline used to search for gravitational-wave transient signals. The data analysis is hierarchical because it consists of subsequent steps that focus on a tighter and tighter portion of the parameter space. As part of this pipeline, the classifier output will be fed to a more accurate follow-up.

In this section we will briefly review some of the pieces of the current hierarchical pipeline used for continuous-wave signals in the *Virgo Rome* group. There are efforts to generalize this pipeline to long transient signals, as well as optimize the legacy code to be able to efficiently run on massively distributed CPU/GPU infrastructures. The integration and cooperation with our neural network trigger is just the first step in this direction: make the analysis future-proof, being able to handle and process the huge amount of data (and signals) we expect from the future observational runs of the LIGO/Virgo detectors.

10.1 Peakmap

A peakmap is a compressed representation of a whitened spectrogram. This is the first step of the hierarchical procedure and it will be the default input for the following frequency-Hough transform. The aim of the peakmap is to enhance the visibility of the signal in the time-frequency plane by distillation of the relevant information, trying to get rid of the effect of the background noise. This representation was originally designed to be an optimized input for the Hough transform, so it must be discrete (binary), sparse and lightweight on disk.

The procedure consists of recording only the local maxima along each temporal slice of the whitened spectrogram [13]. A point is a local maximum if its value is higher than the one preceding and the one following, so the sign of the difference between the element should switch from positive to negative.

From the computational point of view, this search can be parallelized, even on GPU, by using convolutions. We start by computing in parallel the sign of the difference between two adjacent positions in the spectrum using the software routine `sign(diff(spectrum))` and then convolve it with the kernel `[-1,1]`, searching the positions where the result is exactly `2`. These are the positions of local maxima. We note that this is different from just imposing a null first derivative.

As a further step to reduce the incidence of the non-stationary noise, we only keep local maxima that have whitened amplitude ratio greater than $R = 2.5$. This threshold is chosen to increase sparsity and minimize the computational burden of the follow-up, while preserving the largest fraction of all the possible signals [14].

It is important to stress that a potential candidate which is left out from the peakmap, due to an inaccurate or non-optimal construction, will never be recovered by the Hough and the following coherent follow-up. This is the case with signals that fall below the threshold. Efforts are being done to address this important issue:

- accelerate the computation of the hierarchical pipeline [10] so to be able to handle a lower threshold
- use of the Radon transform on the full whitened spectrogram [85]
- develop other complementary (linear or nonlinear) approaches that make use of the full whitened spectrogram. The method developed here, based on an artificial neural network, is one example.

The figure (34) shows an example of a peakmap, where the small injected signal on the gaussian white noise background can clearly be seen.

10.2 The frequency-Hough transform

The Hough transform is a feature extraction technique that is used to detect straight lines inside an image [23].

Suppose we want to automatically recognize all the straight lines in a noiseless picture. Every straight line obeys the equation $y = a\,x + b$, so we want to recover the pair of parameters $(a,\, b)$ that uniquely characterize that line. We can use the

Hough transform to create a mapping from the $(x,\, y)$ plane to the $(a,\, b)$ plane. In the $(a,\, b)$ plane (the parameter space) we have to make a 2D histogram to search for peaks that correspond to the searched parameters. This procedure is depicted in figure (35), where the input image is noiseless and it is clear that there are two lines inside the image. The figure (36) shows a much more complicated case, where the input image is very noisy.

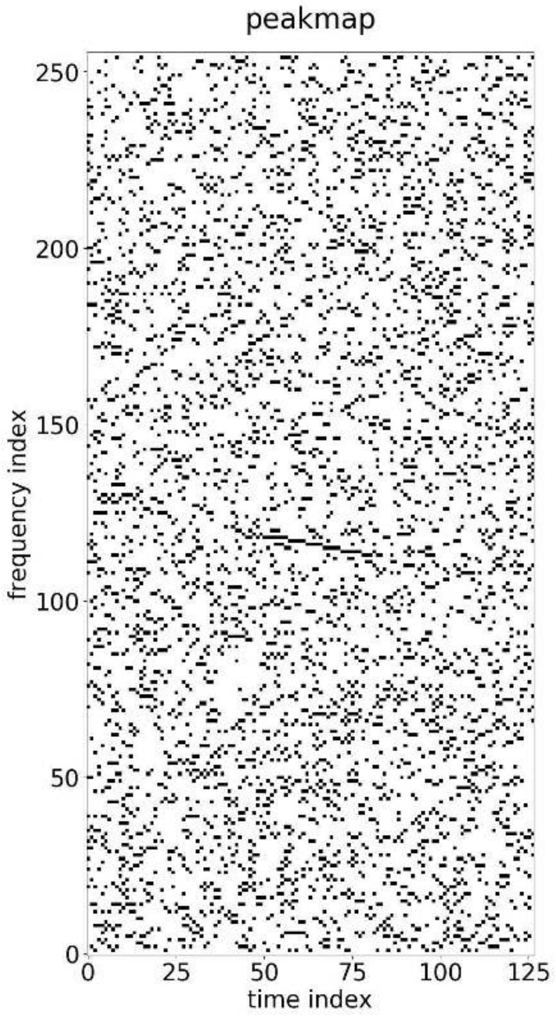
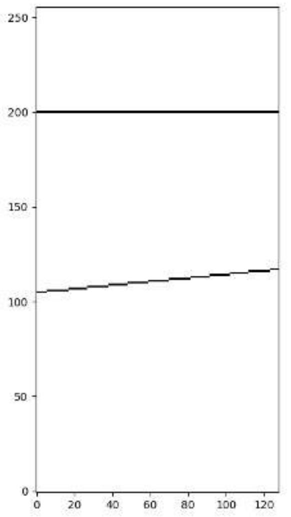
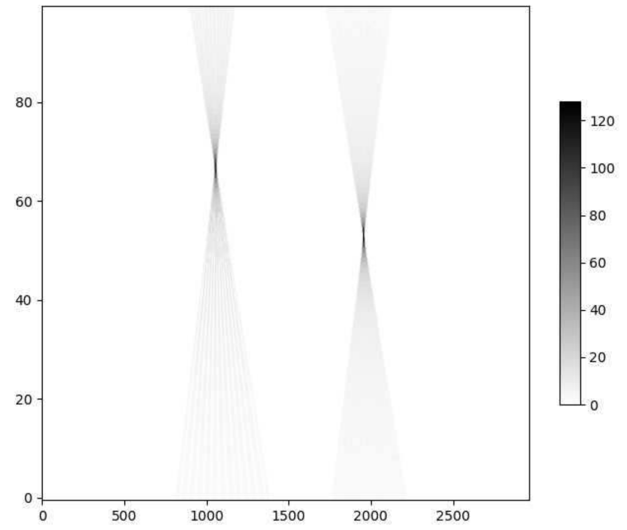


Figure 34: Peakmap of an injected transient signal on a gaussian white noise background. This is a compressed representation of the time-frequency plane.

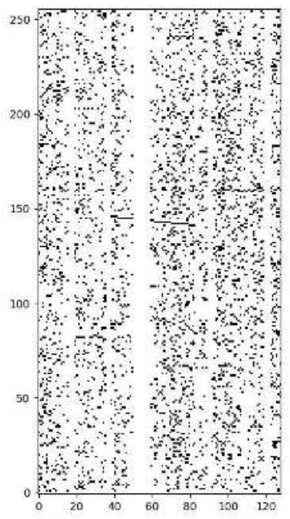


(a) Simple peakmap: two lines in the $(x,\, y)$ plane.

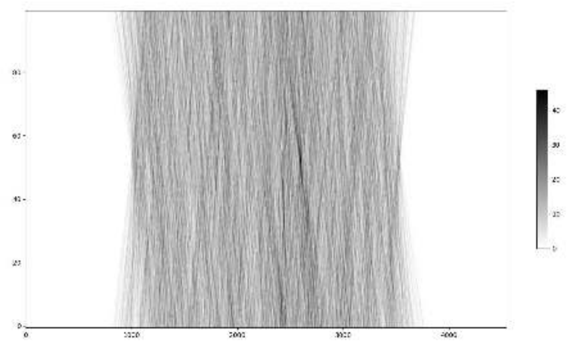


(b) Simple Hough: two clusters in the $(a,\, b)$ plane. We note the degeneration of the reconstructed parameters, marked by the characteristic cones in the parameter space.

Figure 35: A simple example of how the Hough transform works, mapping the $(x,\, y)$ in the $(a,\, b)$ one.



(a) Complicated peakmap resulting from a portion of the O2 data.



(b) Hough map associated to the peakmap on the left. The signal peak is lower than the peak resulting from a faint linear noise structure in the peakmap.

Figure 36: An example of how the frequency-Hough works with the O2 data.

The peaks in figure (35b) are well visible and the voting procedure (also called *selection of candidates*) is straightforward. The voting procedure is crucial when the input image is noisy, as the true peakmaps are (figure 36a). With noisy images, the output of the Hough is very noisy too and roughly-aligned noise points can become a prominent peak in the parameter space, which can also be higher than the peak corresponding to the true line. For instance, in figure (36a), the injected signal is the oblique segment in the middle of the peakmap, but there is also a faint horizontal line due just to the background noise. This background line turns out to be the higher peak in the Hough map (figure 36b).

Given the fact that the number of lines present in the image is not known, we need to define a criterion to know how many peaks we want to recover. As in the above example, the number of selected peaks is crucial to be able to correctly recover the original line parameters, but the more peaks are selected, the more computing-intensive the follow-up will be. Thus, the criterion to choose the number of selected peaks varies from analysis to analysis. In the case of continuous signals, the chosen number is from 2 to 8 candidates for every frequency interval of 1 Hz and for every point in the sky grid.

Currently, there are efforts to implement the Hough transform on GPU, to take full advantage of the parallel nature of the algorithm [10] and thus being able to select even more candidates for the follow-up. Nonetheless, there is an intrinsic limitation to be taken into account: the algorithmic complexity of the Hough transform scales as $\mathcal{O}\left(N^{n-2}\right)$, where N is the size of the image space and n is the number of parameters [84]. Thus, the Hough transform is fast only with $n = 2$, so it can efficiently be used only to detect straight lines (two parameters: a and b) and not, for example, parabolic structures like signals with second-order spindown.

10.3 Critical ratio

Peak selection in the Hough map shall be done with respect to the background noise level. For this reason , before the candidate selection, we prepare the data by performing a procedure that can be seen as a whitening process of the Hough map.

In the frequency-Hough, the two parameters b and a are the initial frequency f_0 of the signal and its linear spindown s. The Hough map $N(f_0, s)$ can be re-expressed as a map of critical ratios, where the critical ratio (CR) of every point $\{f_0, s\}$ is computed as

$$\mathrm{CR}(f_0, s) = \frac{N(f_0, s) - \mu_{\text{noise}}}{\sigma_{\text{noise}}} \tag{45}$$

where μ_{noise} and σ_{noise} are computed on the whole map $N(f_0, s)_{\text{noise}}$, that is the Hough output for the noise-only peakmap. The $N(f_0, s)_{\text{noise}}$ output is different for every noisy peakmap input, so we need to average over all the results to obtain meaningful numbers. Given the fact that the mean μ and the standard deviation σ are not robust in the presence of outliers, we use the median and the centered percentile at 68.27%.

In the search for continuous-waves, the usual choice for the value of the critical ratio at which a weak continuous signal can be found in the data with a significant statistical confidence is CR $\gtrsim 5$.

We are already working on the code to see if this value can be lowered with the usage of our algorithm or if this same value can be achieved in a less computing-intensive way.

11 Conclusions

We propose a method that exploits Artificial Intelligence to recognize gravitational-wave signals in the time-frequency plane. The algorithm makes use of a Deep Convolutional Neural Network, which is a class of architectures often used in the field of image recognition and Computer Vision. We translate the signal detection problem into a classification problem, where an Artificial Neural Network is trained to classify the images of a portion of the time-frequency plane into two mutually exclusive classes: "noise-only" and "noise+signal".

This method is not actually able to do estimations of the signal's parameters: the classifier can only say which is the probability to have a signal in a given time-frequency frame. It could be thus used as a trigger to aid other computing-intensive pipelines to shrink the volume of the parameter space in which to search in. For instance, it could be applied to all the hierarchical all-sky "blind" searches, where lots of the computing resources are allocated to blindly search in the initial coarse-grained grid. Being able to aid these pipelines lowering their computational burden can also mean to be able to perform deeper searches in a much wider parameter space.

We optimized our data preprocessing and the network's training to search for long gravitational-wave transients of durations of $\mathcal{O}$(days). The initial motivation for the search for this type of signals was the possibility that the post-merger remnant of GW170817 could be a stable or metastable neutron star. However, the methodology we propose is quite general, so in principle our classifier can be re-trained to be able to search for different types of signals or even to classify noise glitches in the detector's commissioning phase. We are currently adapting our pipeline to be able to classify bursts from supernova core-collapse.

Our neural network is trained on images constructed from the data of the most recent observational run (O2) of the two LIGO detectors, with the addition of an old clean dataset (VSR4) for the Virgo detector. This in order to prove that our model is able to efficiently recognize signals even in the presence of noise nonstationarities that mimic those signals.

Moreover, thanks to a new type of preprocessing of the spectrograms, our algorithm is able to process the data from the three interferometric antennas simultaneously. Exploiting the parallelism of the classification algorithm, we are able to perform the whole analysis in less than 5 minutes on a single GPU. This fast computing time opens the possibility to build online or low-latency triggers to further improve the possibilities of multimessenger astrophysics.

Nonetheless, our pipeline is not optimal in many aspects and a lot of work should be done to further improve it and integrate it with the other existing algorithms. The first thing to be done is merging this pipeline with the relative already-existent follow-up and then rewriting the preprocessing stage in order to be able to read the raw time-domain output of the three detectors, perhaps even in real-time.

Our algorithm is not meant to represent a state-of-the-art of any kind: our wish was only to experiment and build a toy model that is able to numerically demonstrate the potential of the application of artificial intelligence to the gravitational-wave data analysis, showing that the results can be competitive both in computing time and detection efficiency with respect to other well-established pipelines. This work should be regarded as a first step in this promising direction.

References

[1] Aasi et al. - Advanced LIGO - 2015 - Classical Quantum Gravity 32, 074001 (2015)

[2] LIGO Scientific Collaboration and Virgo Collaboration - Observation of gravitational waves from a binary black hole merger - 2016 - PhysRevLett.116.061102

[3] LIGO Scientific Collaboration and Virgo Collaboration - GW151226: Observation of gravitational waves from a 22-solar-mass binary black hole coalescence - 2016 - Phys. Rev. Lett. **116**, 241103

[4] LIGO Scientific Collaboration and Virgo Collaboration - GW170104: Observation of a 50-solar-mass binary black hole coalescence at redshift 0.2 - 2017 - Phys. Rev. Lett. **118**, 221101

[5] LIGO Scientific Collaboration and Virgo Collaboration - GW170814: A three-detector observation of gravitational waves from a binary black hole coalescence - 2017 - Phys. Rev. Lett. **119**, 141101

[6] The Royal Swedish Academy of Sciences - Press release: the Nobel prize in Physics 2017 `https://www.nobelprize.org/nobel_prizes/physics/laureates/2017/press.pdf`

[7] LIGO Scientific Collaboration and Virgo Collaboration - GW170817: Observation of gravitational waves from a binary neutron star inspiral - 2017 - Physical Review Letters **19**, 161101

[8] LIGO Scientific Collaboration, Virgo Collaboration et al. - Multi-messenger observations of a binary neutron star merger - 2017 - The Astrophysical Journal Letters, 848:L12

[9] Abbott et al. - Gravitational waves and gamma-rays from a binary neutron star merger: GW170817 and GRB 170817A - 2017 - The Astrophysical Journal Letters, 848:L13

[10] La Rosa, Palomba, Astone - Continuous gravitational-wave signal analysis using GPGPU (preprint 2018)

[11] George, Huerta - Deep neural networks to enable real-time multimessenger astrophysics - 2017 - arXiv:1701.00008

[12] Gabbard, Hayes, Messenger, Williams - Matching matched-filtering with deep networks for gravitational wave astronomy - (preprint)

[13] Astone, Frasca, Palomba - The short FFT database and the peak map for the hierarchical search of periodic sources - 2005 - Classical and Quantum Gravity **22** (2005) S1197–S1210

[14] Astone, Colla, D'Antonio, Frasca, Palomba - Method for all-sky searches of continuous gravitational wave signals using the frequency-Hough transform - 2014 - PhysRevD.90.042002

[15] Dall'Osso, Giacomazzo, Perna, Stella - Gravitational waves from massive magnetars formed in binary neutron star mergers - 2015 - The Astrophysical Journal, 798:25

[16] Thrane, Kandhasamy, Ott, Anderson, Christensen, Coughlin, Dorsher, Giampanis, Mandic, Mytidis,Prestegard, Raffai, Whiting - Long gravitational-wave transients and associated detection strategies for a network of terrestrial interferometers - 2011 - PhysRevD.83.083004

[17] LIGO Scientific Collaboration and Virgo Collaboration - White Paper on Gravitational Wave Searches and Astrophysics (2015-2016 edition), §3.11 and §3.15

[18] Astone, Bassan, Bonifazi, Carelli, Coccia, Cosmelli, D'Antonio, Fafone, Frasca, Minenkov, Modena, Modestino, Moleti, Pallottino, Papa, Pizzella, Quintieri, Ronga, Terenzi, Visco - Search for periodic gravitational wave sources with the Exploter detector - 2000 - arXiv:gr-qc/0011072

[19] Frasca - Time domain windows for PSS search - internal report

[20] Papoulis - Probability and Statistics - 1989 - Prentice Hall

[21] Wikipedia page about the χ^2 distribution: https://en.wikipedia.org/wiki/Chi-squared_distribution

[22] LIGO Scientific Collaboration, Virgo Collaboration et al. - First low-frequency Einstein@Home all-sky search for continuous gravitational waves in Advanced LIGO data - 2017 - arXiv:1707.02669

[23] Hough - Machine analysis of bubble chamber pictures - 1959 - Proceedings, International conference on high-energy accelerators and instrumentation

[24] RGBgw code repository https://github.com/FedericoMuciaccia/cnn4gw

[25] cnn4gw code repository https://github.com/FedericoMuciaccia/RGBgw

[26] Snag: a data analysis toolbox oriented to gravitational-wave antenna data.
webpage: http://grwavsf.roma1.infn.it/snag/
installation instructions: http://grwavsf.roma1.infn.it/snag/Snag2_UG.pdf
user guide: http://grwavsf.roma1.infn.it/snag/Snag2_UG.pdf
programming guide: http://grwavsf.roma1.infn.it/snag/Snag2_PG.pdf

[27] HDF5 website: https://support.hdfgroup.org/HDF5/

[28] Glorot, Bengio - Understanding the difficulty of training deep feedforward neural networks - 2010 - Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR 9:249-256, 2010

[29] Kingma, Ba - Adam: A Method for Stochastic Optimization - 2014 - arXiv:1412.6980

[30] Zevin, Coughlin, Bahaadini, Besler, Rohani,Allen, Cabero, Crowston, Katsaggelos, Larson, Kyoung Lee, Lintott, Littenberg, Lundgren, Oesterlund, Smith, Trouille, Kalogera - Gravity Spy: Integrating Advanced LIGO Detector Characterization, Machine Learning, and Citizen Science - 2016 - arXiv:1611.04596

[31] Bengio - Learning Deep Architectures for AI - 2009 - Foundations and Trends in Machine Learning. 2

[32] LeCunn - Generalization and network design strategies - 1989 - Connectionism Perspective

[33] LeCunn - Backpropagation applied to handwritten ZIP code recognition - 1989

[34] LeCun, Bottou, Bengio, Haffner - Gradient-Based Learning Applied to Document Recognition - 1998 - Proceedings of the IEEE, 86(11):2278-2324

[35] Kolmogorov - Grundbegriffe der Wahrscheinlichkeitsrechnung - 1933 - Springer

[36] Zhou, Chellappa - Computation of optical flow using a neural network. In Neural Networks - 1988 - IEEE International Conference, pages 71–78

[37] Szegedy, Liu, Jia, Sermanet, Reed, Anguelov, Erhan, Vanhoucke, Rabinovich - Going Deeper with Convolutions - 2014 - arXiv:1409.4842

[38] Uchiyama et al. - Present status of large-scale cryogenic gravitational wave telescope - 2004 - Class. Quantum Grav., vol. 21, 5, 2004, pp. S1161–S1172

[39] Klimenko, Yakushin, Mercer, Mitselmakher - Coherent method for detection of gravitational wave bursts - 2008 - arXiv:0802.3232

[40] Krizhevsky, Sutskever, Hinton - ImageNet Classification with Deep Convolutional Neural Networks - 2012

[41] Metha, Schwab - An exact mapping between the Variational Renormalization Group and Deep Learning - 2014 - arXiv:1410.3831

[42] Shibata, Duez, Liu, Shapiro, Stephens - Short gamma-ray bursts in the "time-reversal" scenario - 2006 - PhRvL, 96, 031102

[43] Faber, Rasio - Binary neutron star mergers - 2012 - LRR, 15, 8

[44] Baiotti, Rezzolla - Binary neutron star mergers: a review of Einstein's richest laboratory - 2017 - RPPh, 80, 096901

[45] Metzger, Quataert, Thompson - Short-duration gamma-ray bursts with extended emission from protomagnetar spin-down - 2008 - MNRAS, 385, 1455

[46] Einstein - Die Feldgleichungen der Gravitation - 1915 - Sitzungsberichte der Preussischen Akademie der Wissenschaften zu Berlin: 844–847

[47] Einstein - Näherungsweise Integration der Feldgleichungen der Gravitation - 1916 - Sitzungsberichte der Königlich Preussischen Akademie der Wissenschaften Berlin

[48] Einstein - Über Gravitationswellen - 1918 - Sitzungsberichte der Königlich Preussischen Akademie der Wissenschaften Berlin

[49] Ricci - Dispense del corso di Gravitazione Sperimentale - 2017

[50] Ferrari, Gualtieri - Lecture notes on General Relativity, Black Holes and Gravitational Waves - 2017

[51] Wald - General Relativity - 1984 - The University of Chicago Press

[52] Weinberg - The Quantum Theory of Fields - 1995 - Cambridge University Press

[53] Ohanian, Ruffini - Gravitation and spacetime - 2013 - Cambridge University Press

[54] Abbott et al. (LIGO Scientific Collaboration and Virgo Collaboration) - Search for post-merger gravitational waves from the remnant of the binary neutron star merger GW170817 - 2017 - (preprint)

[55] Shibata, Taniguchi - Merger of binary neutron stars to a black hole: disk mass, short gamma-ray bursts, and quasinormal mode ringing - 2006 - PhRvD, 73, 064027

[56] Baiotti, Giacomazzo, Rezzolla - Accurate evolutions of inspiralling neutron-star binaries: prompt and delayed collapse to a black hole - 2008 - PhRvD, 78, 084033

[57] Baumgarte, Shapiro, Shibata - On the maximum mass of differentially rotating neutron stars - 2000 - ApJL, 528, L29

[58] Shapiro - Differential rotation in neutron stars: magnetic braking and viscous damping - 2000 - ApJ, 544, 397

[59] Hotokezaka, Kiuchi, Kyutoku, Muranushi, Sekiguchi, Shibata, Taniguchi - Remnant massive neutron stars of binary neutron star mergers: evolution process and gravitational waveform - 2013 - PhRvD, 88, 044026

[60] Ravi, Lasky - The birth of black holes: neutron star collapse times, gamma-ray bursts and fast radio bursts - 2014 - MNRAS, 441, 2433

[61] Palomba - Gravitational radiation from young magnetars: preliminary results - 2001 - Astronomy and Astrophysics, 367, 525

[62] Cutler - Gravitational waves from neutron stars with large toroidal B fields - 2002 - PhRvD, 66, 084025

[63] Lai, Shapiro - Gravitational radiation from rapidly rotating nascent neutron stars - 1995 - ApJ, 442, 259

[64] Lindblom, Owen, Morsink - Gamma-ray burst afterglow plateaus and gravitational waves: multi-messenger signature of a millisecond magnetar? - 1998 - PhRvL, 80, 4843

[65] Andersson - A new class of unstable modes of rotating relativistic stars - 1998 - ApJ, 502, 708

[66] Harris - On the use of windows for harmonic analysis with the discrete Fourier transform - 1978 - Proceedings of the IEEE. 66 (1): 51–83

[67] Tukey - An introduction to the calculations of numerical spectrum analysis - 1967 - Spectral Analysis of Time Series: 25–46.

[68] Welch - The use of Fast Fourier Transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms - 1967 - IEEE Transactions on Audio and Electroacoustics, AU-15 (2): 70–73

[69] O2 hardware injections database
website: `https://ldas-jobs.ligo.caltech.edu/~keithr/cw/O2_injection_params_O2.html`

[70] Mallat - Understanding Deep Convolutional Networks - 2016 - arXiv:1601.04920

[71] Uncini - Neural Networks (preprint)

[72] Goodfellow, Bengio, Courville - Deep Learning - 2016 - MIT Press
website: `http://www.deeplearningbook.org`

[73] Ioffe, Szegedy - Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift - 2015 - arXiv:1502.03167

[74] Krizhevsky, Sutskever, Hinton - ImageNet Classification with Deep Convolutional Neural Networks - 2012 - Advances in Neural Information Processing Systems 25

[75] Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov - Dropout: a simple way to prevent neural networks from overfitting - 2014 - The Journal of Machine Learning Research

[76] Goodfellow, Pouget-Abadie, Mirza, Xu, Warde-Farley, Ozair, Courville, Bengio - Generative adversarial networks - 2014 - arXiv:1406.2661

[77] RGBgw code repository
website: `https://github.com/FedericoMuciaccia/RGBgw`

[78] TFLearn: Deep learning library featuring a higher-level API for TensorFlow
website: `http://tflearn.org/`

[79] Keras: The Python Deep Learning library
website: `https://keras.io/`

[80] TensorFlow: An open-source software library for Machine Intelligence
website: `https://www.tensorflow.org/`

[81] Cover - Geometrical and Statistical properties of systems of linear inequalities with applications in pattern recognition - 1965 - IEEE Transactions on Electronic Computers. EC-14: 326–334.

[82] Cybenko - Approximations by superpositions of sigmoidal functions - 1989 - Mathematics of Control, Signals and Systems, 2 (4), 303-314

[83] Bengio, Louradour, Collobert, Westom - Curriculum learning - 2009 - International Conference on Machine Learning, ICML

[84] Shapiro, Stockman - Computer vision - 2001 - Prentice-Hall Inc.

[85] Radon - Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten - 1917 - Berichte über die Verhandlungen der Königlich-Sächsischen Akademie der Wissenschaften zu Leipzig, Mathematisch-Physische Klasse, Leipzig: Teubner (69) 262–277