Master in High Performance Computing

Multioutput regression of noisy time series using convolutional neural networks, with applications to gravitational waves

Supervisors:
Luca Heltai,
Enrico Barausse

Candidate:
Costantino Pacilio

5th edition
2018–2019

Abstract

In this thesis I implement a deep learning algorithm to perform a multioutput regression. The dataset is a collection of one dimensional time series arrays, corresponding to simulated gravitational waveforms emitted by a black hole binary, and labelled by the masses of the two black holes. In addition, white Gaussian noise is added to the arrays, to simulate a signal detection in the presence of noise. A convolutional neural network is trained to infer the output labels in the presence of noise, and the resulting model generalizes over many order of magnitudes in the noise level. From the results I argue that the hidden layers of the model succesfully denoise the signals before the inference step. The entire code is implemeted in the form of a Python module, and the neural network is written in PyTorch. The training of the network is speeded up using a single GPU, and I report about efforts to improve the scaling of the training time with respect to the size of the training sample.

Contents

Docker container

To make your work shareable and reproducible is essential in scientific research. Docker is a software tool that generates computational environments called containers. The advantage in using Docker is that everything runs inside the container: you do not have to worry about the compatibility between your software and the libraries needed to run the code. This makes Docker an ideal tool to share a scientific code.

The main computational body of this thesis consists into a Python module `GWorch.py` and a related notebook `PyTorch_Regression.ipynb`, where all the computations and the plots documented hereafter are explicitely presented. I wrapped everything into a Docker container, which also contains the dataset `TD_PhenomD.hdf5` used to train and test the machine learning algorithms. The container can be downloaded from the DockerHub public repository `cpacilio/gworch` as

```
docker pull cpacilio/gworch:latest
```

The main folder contains a script `./jupyter.sh` to launch a Jupyter environment and start to play around with the code. It is highly recommended to run the notebooks with a CUDA compatible GPU: they are provided with a link to a Colab version, so that you can use the Google Colab GPU.

Chapter 1

Introduction to the problem

1.1 Deep learning

Machine learning is a class of algorithms which learn to perform specific tasks by induction, after being exposed to proper train samples. The task is usually in the form of complex pattern recognition and/or reconstruction of nonlinear correlations between the data. The popularity of machine learning lies in its generalization, i.e., the ability of the machine to replicate the task on new samples to which it was never exposed before. Another factor of popularity is the diffusion of GPU computing: indeed, large train sets are necessary for an efficient training, and speeding up the learning process is essential to make it feasible.

Deep learning [1] is an increasingly popular form of machine learning, in which the input data are processed through a series of nonlinear transformations. By combining enough nonlinear transformations, large classes of functions can be approximated with a high degree of accuracy. The basic algorithms for deep learning are the so called feed-forward neural networks (NN). A famous result, known as the universal approximation theorem [2], states that a feed-forward neural network with a single hidden layer ca approximate any function with compact support in $\mathbb{R}^n$, with very mild assumptions on the kind of nonlinear transformations.

The power of NN is their ability to combine efficient nonlinear representations of the input: this allows them to discover complex patterns with little preprocessing of the raw data. The basic unit of a NN is the neuron: a neuron applies a linear combination to the input, followed by a nonlinear transformation (activation function). The most common activation function is the rectified linear unit (ReLU), defined as $f(z) = max(0, z)$. The coefficients of the linear combination, or *weights*, are the parameters of the NN. The simplest example of NN is a fully connected neural network (FCNN), in which neurons are organized in successive layers; each neuron is connected to all the neurons of both the previous and the next layers, whence the name.

A more elaborated form of NN is a convolutional neural network (CNN). A CNN is a sequence of convolutional layers followed by ordinary fully connected layers. They are based on the concept of weight sharing: a convolutional layer consists in a group of weights (filters) shared by the whole input. Filters represent abstract features of the input. To better extract the relevant features, each filter undergoes a pooling; the common forms of pooling are "average pooling" (sub-groups of weights in a filter are averaged) or "max pooling" (only the max value is selected in each sub-group).

Even more sophisticated forms of NN have been developed during the last decades. They include recurrent neural networks (RNN), autoencoders (AE) and Bayesian neural networks (BNN). Their application lies beyond the scope of this thesis, but in the final discussion I will highlight the role that they can play in future developments of the work.

1.2 Description of the problem

In this thesis I implement a deep learning algorithm to perform a multioutput regression. The dataset is a collection of one dimensional time series arrays, corresponding to simulated gravitational waveforms (GW) emitted by a black hole binary, and labelled by the masses of the two black holes. The task of the algorithm is to infer the output labels with the best accuracy.

The problem is complicated by the fact that GW signals are noisy, because the pure signal is superimposed to the detector noise. The detector noise is not regular: strictly speaking, it is time dependent and not Gaussian, but it is well approximated by a Gaussian noise with a colored (non-flat) power spectrum distribution [3]. In the following I will restrict to the simple case of white Gaussian noise: this is motivated by the fact that, given a colored Gaussian noise, one can always move close to the case of a white noise by whitening the signal; moreover, as shown in [4], an algorithm performing well on white Gaussian noise is also trainable over whitened signals with a colored noise. Notice that the relative loudness of the noise w.r.t. the signal is not fixed, but it is allowed to vary, and one can also attempt to infer it from the model: indeed, in Ch.2.3.2, I expand the algorithm to a three dimensional output regression, including also the noise level between the targets.

The motivation for the problem comes from recent developments in GW astronomy, initiated with the first landmark detection of a GW from the Ligo-Virgo collaboration [5]. It has become clear that, in the upcoming future, astronomy will face the need to process large amounts of data from multiple observational channels, with high precision and in a relatively small time (ideally, in real time or even faster). Moreover, there is a growing consensus on the fact that these objectives can be optimally realized by exploiting the recent advances in high performance computing, such as deep learning and GPU computing [6, 7, 8].

The need for a fast and accurate parameter estimation comes from the purposes of multi-messenger astronomy, in which observations are carried over by a network of multiple detectors looking at distinct observational channels. The rapidity of communication between the nodes of the network is crucial to synchronize the observations on the same source, which must be accurately located and characterized. Current techniques [9] for searching and parameter estimations are based on template based matches (matched-filtering [10, 11]), and on Bayesian inference via stochastic exploration of a large parameter space [12]. These procedures are intensively costing, because of the dimensionality of the parameter space and of the template banks [13]. In contrast, the advantage of using a pre-trained NN is that the computational burden is relegated to the offline training, while the online inference takes only a small time.

Recent years have seen a spreading in the applications of NN to GWs: [14, 4, 15, 16, 17] used CNN for searching, point regression and sky localization; [18, 19] used AE and RNN for denoising; [20, 21, 22] used BNN for Bayesian inference. See also [23, 24]. This list does not pretend to be exhaustive. The work of this thesis follows the spirit of [4, 15]: I will reproduce the basic results and implement some minor improvements. The computational aspects of the project do not require a deep knowledge of GW physics; therefore I will introduce physical concepts only when strictly needed. In the following sections I give a basic understanding of the problem; in particular, Sec.1.2.1 describes the simulation of pure GWs, while Sec.1.2.2 covers noise addition.

1.2.1 Generating the dataset

I need two datasets of GWs, a train_set and a test_set, to train the NN and evaluate its performances. I used the public software PyCBC [25, 26, 27], which was explicitly designed to simulate and analyze GW signals, including real signals from the Ligo-Virgo gravitational interferometers. PyCBC allows to generate simulated GW signals from various approximants, based on interpolations of numerical relativity and exact analytical techniques. Its usage is nicely documented. For example, at `http://pycbc.org/pycbc/latest/html/waveform.html` you find a tutorial on simulated waveforms generation, while `http://pycbc.org/pycbc/latest/html/pycbc.waveform.html` documents the `waveform` package. I used the function `waveform.get_td_waveform` to generate simulates GW signals in the time domain.

`get_td_waveform` takes several arguments to set the approximant, the intrinsic parameters (masses, spins) and the extrinsic ones (geometry of the orbit, space-time location), and the space-time coordinates relative to the detector frame. However I just set five arguments, leaving the rest to their default:

— `approximant`: the numerical method used to approximate the GW waveforms; I used the IMRPhenomD approximant [28, 29];

— `mass1, mass2`: the masses of the components of the binary, in units of $M_\odot$ (solar mass);

— `f_lower`: the starting frequency of the waves (in Hz); I choose `f_lower` = 40 Hz;

— `delta_t`: the time resolution of the signal (in Hz), i.e., the number of bins per second in the time series; I choose `delta_t` = 1/8192 Hz.

Not fixing the other parameters is equivalent to assume that the binary components of the source are non-spinning, inspiraling in a quasi-circular orbit at a fixed distance of 1 Mpc from the detector, and with optimal face-on orientation relative to the detector frame. If you keep fixed all the other parameters, the distance from the source controls the amplitude of the measured signal: more precisely, the amplitude scales as the inverse of the distance. However, as we shall see at the beginning of Ch.2, signals are preprocessed in the inference pipeline, in such a way that the original information about the amplitude does not play any role. Therefore, fixing the distance to 1 Mpc is just a convention without any impact on the final results.

The test_set and the train_set are generated by varying `mass1` and `mass2` in the range $M_1, M_2 \in [5, 80] M_\odot$ and $M_1, M_2 \in [5.5, 80.5] M_\odot$ respectively, with a mass spacing $\Delta M = 1 M_\odot$. Moreover, since the problem is symmetric w.r.t. the exchange of the masses, I restricted to the lower half plane $M_2 > M_1$ without loss of generality (Fig. 1.1). By simple combinatorics, we see that both datasets contain n_samples = 2926 waveforms.

Only the last 1 second of each waveform is selected: this is because CNNs require fixed size inputs, and 1 second is the maximum time extent of IMPRPhenomD simulation for high component masses. The waveforms $h(t)$ are normalized in such a way that $abs(h)|_{\max} = 1$.[1] Fig. 1.2 shows a plot of one waveform from the test_set, corresponding to n_sample = 1000.

The two datasets are collectively stored into a `.hdf5` file — `TD_PhenomD.hdf5`, where TD stands for "time domain" — which is organized according to the following hierarchical structure:

[1] As I already explained, an overall normalization is irrelevant to the algorithm. I normalized to 1 just to store more handy numbers in my arrays, since the typical amplitude of the un-normalized samples is of order 10^{-19} or less.

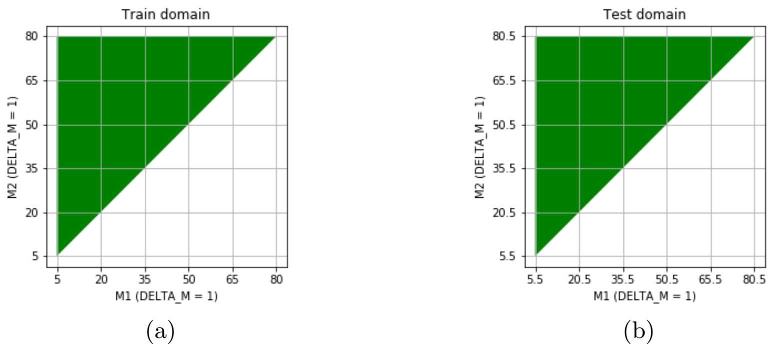


Figure 1.1: A visual representation of the grid domain for train_set (a) and test_set (b).

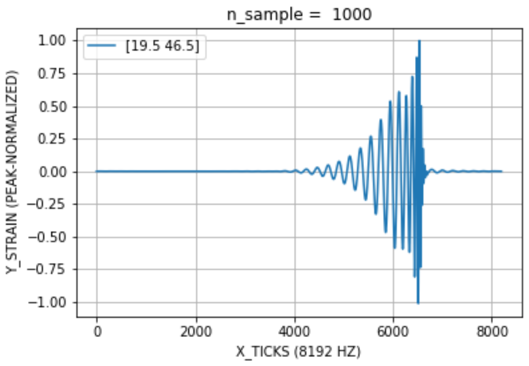
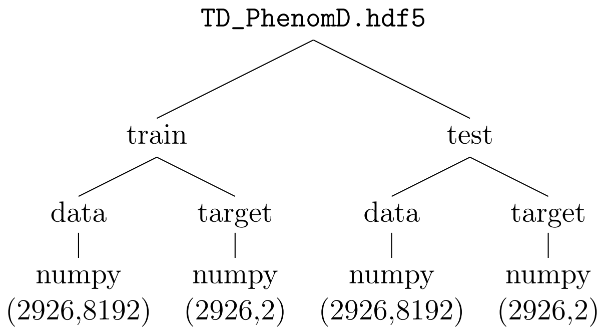


Figure 1.2: Example of a time domain waveform from the test_set, corresponding to $n\_sample = 1000$ or equivalently to $M_1, M_2 = (19.5, 46.5)$.



The ground levels of the tree indicate that the data and target of each group are numpy arrays of shapes, respectively, (n_samples, n_features) = $(2926, 8192)$ and (n_samples, n_targets) = $(2926, 2)$.

1.2.2 Adding the noise

In order to add white Gaussian noise to a numpy array, I used the function `numpy.random.normal(mean,sigma,bins)`, which returns a numpy array of length `bins`, whose points are randomly selected from a Gaussian distribution with a given `mean` and standard deviation `sigma`. The white noise corresponds to `mean` $= 0$, and in our case `bins` $= 8192$. Therefore, `sigma` is the parameter controlling the relative loudness of the noise w.r.t. the signal.

As a measure of the relative loudness, I used the peak signal-to-noise ratio (pSNR), defined as the ratio between the absolute peak of the signal and the standard deviation of the noise. In math: let $s(t) = h(t) + n(t)$ be the total signal in the time domain, decomposed as the sum of the pure signal $h(t)$ and the noise $n(t)$; then

$$\text{pSNR} \ = \frac{max|h(t)|}{\texttt{sigma}} \quad \text{with} \quad n(t) \in \text{Gauss}(0, \texttt{sigma})\,. \tag{1.1}$$

Fig.1.3 gives a visual representation of the relation between the noisy signals (blue) and the pure signals (orange), at two different noise levels pSNR $= 3$ and pSNR $= 1$. We see that, as the pSNR decreases, it becomes more challenging to distinguish the noisy signals from pure noise.

It is obvious that any algorithm will become less accurate as the pSNR decreases. It is a matter of experimental design to decide how much low the pSNR is allowed to be without losing in inference accuracy. The web page `https://www.gw-openscience.org/catalog/GWTC-1-confident/html/` contains an updated catalog of the Ligo-Virgo confident detections: you can visualize the noise level in terms of the matched filtering SNR (MF-SNR), which is defined as the square root of the signal over noise power integrated over frequencies[2] [13, 30]. The MF-SNR for the confi-

[2]In math: $(\text{MF-SNR})^2 = 4\int_{f_{min}}^{\infty} df\, |\tilde{h}(f)|^2/S_n(f)$, where $\tilde{h}(f)$ is the frequency domain representation of the GW strain and $S_n(f)$ is the power spectral density of the noise.

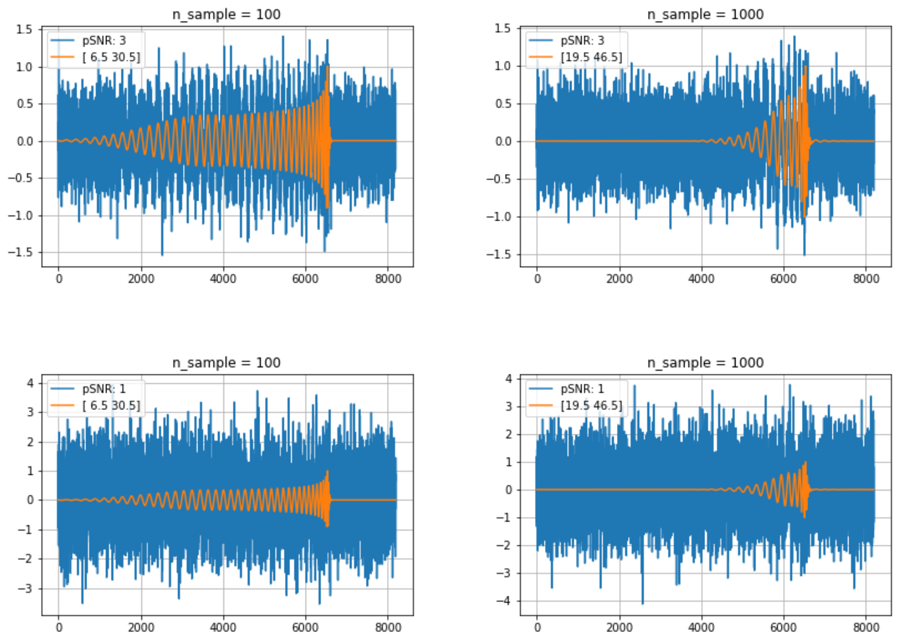


Figure 1.3: Orange: pure signal; Blue: noisy signal. Upper panel: addition of white Gaussian noise at pSNR $= 3$ to n_sample $= 100$ (left) and n_sample $= 1000$ (right) from the test_set. Lower panel: same but pSNR $= 1$.

dent detection oscillates in the range $\approx 10 \div 15$. Moreover, as illustrated in Fig. 4 of [5], the statistical significance of a detection is less than $\approx 2\sigma$ when MF-SNR $\lesssim 10$. Considering that the relation between MF-SNR and pSNR is approximately MF-SNR $\approx 13 \times$ pSNR [19], we can safely restrict to noise levels pSNR $\gtrsim 0.8$.

Chapter 2

Regression algorithms

In this Chapter I describe the implemented NN architecture, I discuss non trivial aspects in designing and optimizing the training, and I document the resulting performances.

— In Sec.2.1 I discuss regression over pure signals, without noise addition. Despite being an over-simplification, never realized in any actual experiment, it already offers important clues for an efficient training. In particular, I will show how a proper redefinition of the output labels improves the inference accuracy.

— In Sec.2.2 I discuss regression over noisy signal, which is the central result of the thesis. The main challenge is train a NN which generalizes efficiently over a broad range of pSNR levels, ideally up to pSNR $\to \infty$, without overfitting around a specific value of the pSNR .

— Finally, in Sec.2.3 I discuss some improvements: in particular, in Sec.2.3.1 I enforce resilience to time translations of the waveforms, while in Sec.2.3.2 I expand the output layer of the NN to estimate the pSNR .

Before going on, let me explain how I preprocessed the data and how I evaluated the performances.

Preprocessing It is a good practice in machine learning to to standardize the input data [31]. Standardization of an array consists into an affine rescaling of the elements, in such a way that the new elements have zero mean and unit standard deviation. This is useful because it helps to reduce the variances in the distribution of the data.

Now, if you have a dataset of shape (n_samples, n_features), you can standardize the samples w.r.t. the features (standardization by rows) or the features w.r.t. the samples (standardization by columns). For example, the `sklearn.StandardScaler` standardizes by columns. In the case of GW waveforms, it is more natural to standardize by rows, because there is no correlation between the corresponding features of distinct samples. Now it is clear why, as claimed in Sec.1.2.1, an overall normalization of the signals is irrelevant.

Performance metrics I evaluate the inference accuracy using the following two metrics:

— the "R squared" (R2) score, defined as (1 minus) the ratio between the square of the inference errors and the variance of the target labels (see https://scikit-learn.org/stable/modules/model_evaluation.html#r2-score):

$$\mathrm{R2} = 1 - \frac{SS|_{\mathrm{res}}}{SS|_{\mathrm{tot}}}\,, \tag{2.1}$$

where

$$SS|_{\mathrm{res}} = \sum_{i=1}^{\mathrm{n\_samples}} (y_i - \hat{y}_i)^2\,, \tag{2.2a}$$

$$SS|_{\mathrm{tot}} = \sum_{i=1}^{\mathrm{n\_samples}} (y_i - \bar{y})^2\,; \tag{2.2b}$$

— the mean relative error (MRE):

$$\mathrm{MRE} = \frac{1}{\mathrm{n\_samples}} \sum_{i}^{\mathrm{n\_samples}} \left| \frac{y_i - \hat{y}_i}{y_i} \right|\,. \tag{2.3}$$

In the above definitions y_i is the true label for the i-th sample, $\hat{y}_i$ is the inferred value from the model, and $\bar{y}$ is the average of y_i over all samples.

The R2 score ranges from $-\infty$ (arbitrarily bad inferences) to 1 (exact inferences). It is a very common metric used by the scientific community to evaluate regressions, and it corresponds to the built in `score` method implemented by the `sklearn` regression classes. The MRE ranges from ∞ to 0. R2 and MRE offer complementary information: for example, you can have an R2 close to 1 while still a MRE above 20%, which you might not consider satisfactory if you are very demanding.

2.1 Regression over clean signals

2.1.1 Ridge regressor

The Python library `sklearn` already offers several regression algorithms: see https://scikit-learn.org/stable/supervised_learning.html for a complete list. It is useful to investigate the problem with an algorithm from a well tested library: this allows to check that the dataset does not suffer from intrinsic problems, and it offers a benchmark to evaluate the convenience of using a NN.

I chose `sklearn.Ridge` because it is quite fast and moderately accurate for the problem at and. The official documentation can be found at https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html#sklearn.linear_model.Ridge: it is a linear regression algorithm with an L2 regularization. The most relevant arguments are `alpha` (the coefficient of the regularization) and `tol` (the precision of the solution). I found that the best choices for my problem are `alpha` $= 100$ and `tol` $= 10^{-3}$.

Fig.2.1a shows the test accuracies of the resulting trained model. The scatter plots show the predicted labels (y_pred) versus true labels (y_true); the orange straight line shows the ideal inference y_pred = y_true. The results are not much satisfactory: the combined R2 score is ≈ 0.82 and the combined MRE is $\approx 20\%$; moreover, it is disturbing to see a great difference in the accuracy of M_1 and M_2, because the physics is symmetric w.r.t. to their exchange.

The last observation suggests that a proper change of variables may improve the results. Let us consider the following redefinition of the labels:

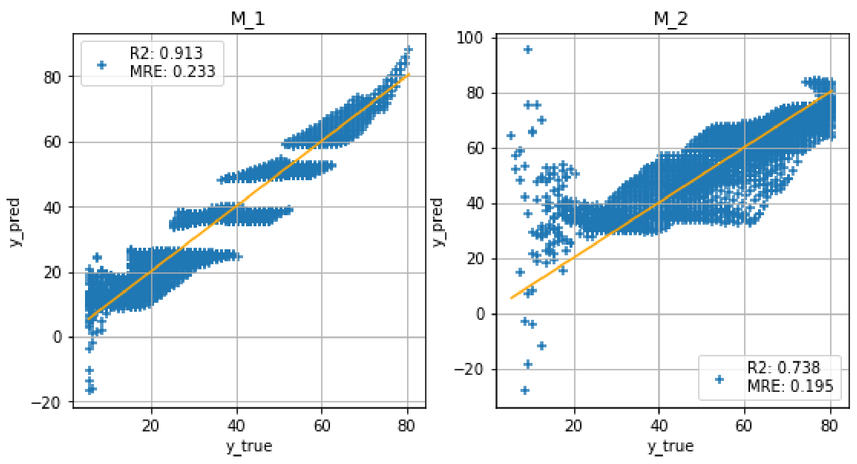
$$M_{\text{tot}} = M_1 + M_2 \qquad \text{(total mass)}\,, \tag{2.4a}$$

$$M_{\text{chirp}} = \frac{(M_1 \times M_2)^{3/5}}{(M_1 + M_2)^{1/5}} \qquad \text{(chirp mass)}\,. \tag{2.4b}$$

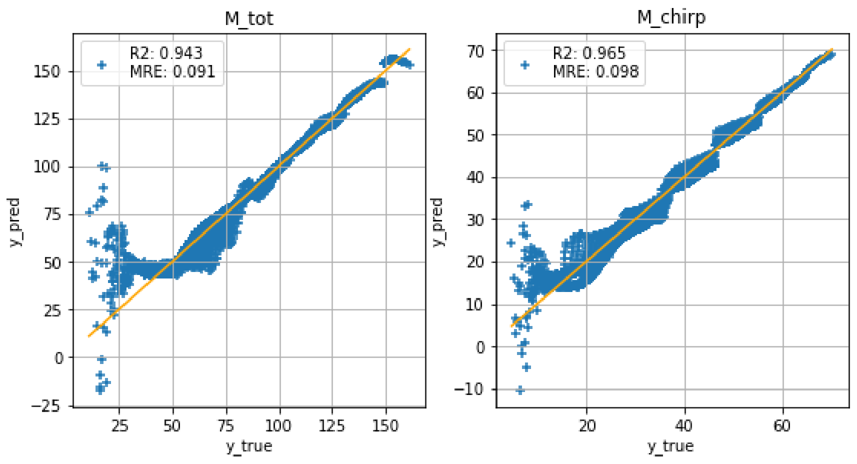
The use of the total mass is motivated by the switching symmetry of the problem, and by the fact that GR is a scale-free theory and M_{tot} approximates the total energy of the system. The use of the chirp mass is less obvious, and it is motivated by the dynamics of the problem. As it is explained in [32], it follows from Einstein's equations that the rate of change of the frequency f of a GW signal is described, at first approximation and for low frequencies, by

$$\frac{df}{dt} \approx \frac{96\,\pi^{8/3}}{5} \left(\frac{G M_{\text{chirp}}}{c^3} \right)^{5/3} f^{11/3}\,. \tag{2.5}$$

Intuitively, if we redefine the labels as they appear in the analytical treatment, we are partially unfolding the nonlinearities of the problem. Therefore, since `sklearn.Ridge` is a linear regressor, the redefinition will improve the resulting accuracy. The results, shown in Fig.2.1b, confirm this intuition. In the new variables, the (combined) R2 score is ≈ 0.95 and the MRE is $\approx 9.5\%$. We see that, already from the analysis of pure signals, we can learn about the importance of redefining the output labels. I will now show that a CNN drastically outperforms the above results.



(a)



(b)

Figure 2.1: Test results of the Ridge regression over the pure signals. (a): regression over the original labels (M_1, M_2). (b): regression over the redefined labels $(M_{\rm tot}, M_{\rm chirp})$.

`Input`	`vector` (`shape`:1×8192)		
`Convolution` `+Max_Pooling` `+ReLU`	`n_kernel`	`kernel_size`	`pool_size`
	16	32	8
	32	32	8
	64	32	8
`Flatten`	`vector` (`shape`:1×704)		
	`input_size`	`output_size`	`Fully connected layers`
`Linear` `+ReLU`	704	64	
	64	64	
`Linear`	64	2	
`Output`	`vector` (`shape`:1×2)		

Figure 2.2: The convolutional neural network designed to perform regression over the pure and noisy signals. Notice that you can modify the last linear layer by increasing the `output_shape` to estimate more targets.

2.1.2 Convolutional Neural Network

Architecture For a deep learning approach to the above regression problem I propose the CNN architecture illustrated in Fig.2.2. The network was built using the PyTorch library: see `https://pytorch.org/docs/stable/index.html` for the official documentation. It has 3 hidden convolutional layers (`torch.nn.Conv1D`), 2 hidden linear layers (`torch.nn.Linear`) and 1 output linear layer. A nonlinear ReLU activation function (`torch.nn.functional.relu`) is applied at the end of all the hidden layers. All the three convolutional layers have the same `kernel_size` = 32 and increasing number of kernel (resp., 16, 32 and 64). After each convolution a max-pooling filter of size 8 is applied. The `stride` is 1 (resp. 8) for the convolutional (resp. pooling) filters.

Training The training depends on several hyperparameters:

— `epochs`: The number of training epochs, which I fixed to 300. Notice that this is just a large number ensuring that the network converges. One can optimize the training time using a validation set. I will describe the benefit of a validation set in Sec.2.2, when dealing with the noisy signals. However, for pure signals, the benefits of a validation set are small.

— `batch_size`: The number of training samples after which the weights are updated. I obtained the best results for `batch_size` = 32.

— `optimizer`: The optimization algorithm. I used the `torch.optim.Adam` with learning rate `lr` = 10^{-3}.

— `criterion`: The loss function to be minimized during the training. I used the mean squared error (`torch.nn.MSELoss`) between y_pred and y_true.

To reduce overfitting, the train_set is shuffled before starting the training, and the batches are shuffled after each epoch. Moreover, I found that the training does not converge if the labels are not mapped into a compact support. Therefore, I normalize the labels during the training, and I save the normalization factors for consistent future inference.

Results I trained the network using the free GPU available on Google Drive (a single-core NVIDIA TESLA K80). Fig.2.3 shows the performances of the resulting model, evaluated on the test_set. We can see that the use of a CNN dramatically improves the results: the combined R2 score is larger than 0.99 and the (combined) MRE does not exceed the 2% level. Moreover, while it is still true that the regression over (M_1, M_2) is worse than the one over $(M_{\text{tot}}, M_{\text{chirp}})$, now the difference is very small and we can undoubtedly say that both regressions are excellent: one possible explanation is that, due to the universal approximation theorem, the CNN is able to handle much better the nonlinearities of the (M_1, M_2) representation.

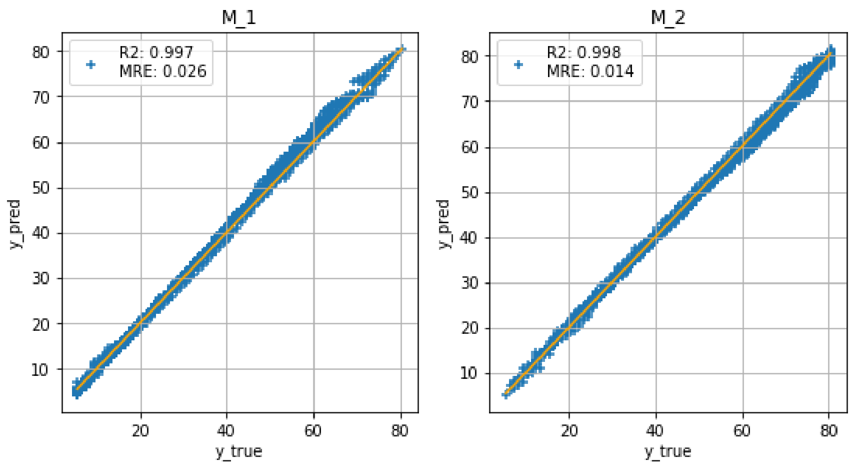
2.2 Regression over noisy signals

To train the previous CNN over noisy signals is more challenging, as it should be already obvious from Fig.1.3: not only the presence of the noise makes the signal less definite, but we also want the network to be accurate over a broad range of noise levels. This problem has been already addressed in [4, 15, 19, 21, 16], and a successful strategy dubbed "curriculum learning" emerged. Here I describe the idea of curriculum learning (CL) as it was implemented in more recent papers [21, 16], along with a variant (CL0) implemented in the initial papers [4, 15, 19][1]:

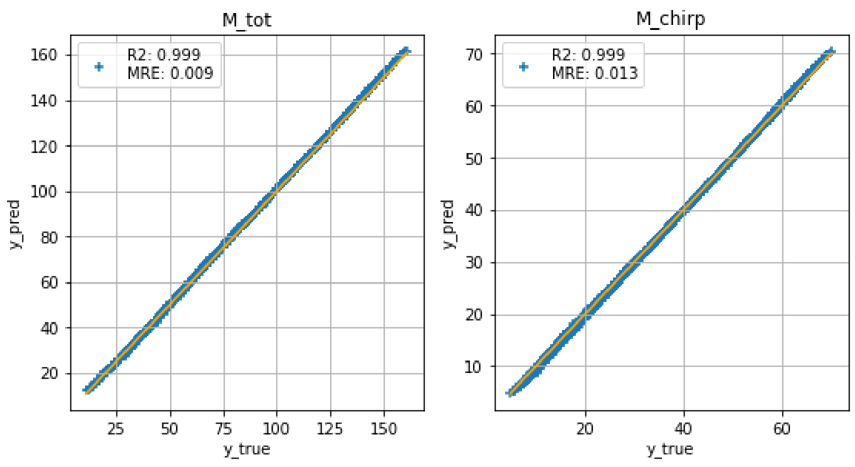
— **Curriculum learning** (CL): The training consists of n iterations; at each iteration, noise with SNR *randomly chosen* in the interval $[N_i, N_{max}]$ is added to the signals, for $i = 1, \ldots, n$. As N_i progressively decreases, the network is first trained on louder signals, which are easier to learn, and then it is gradually exposed to increasing noise levels. Be careful not to confuse the number of iterations with the number of epochs: the latter is re-initialized after each iteration!

— **Curriculum learning 0** (CL0): As before, but at each iteration the added noise has a *fixed* SNR equal to N_i.

[1]One can also switch between the two strategies during the training: for example, in [4, 15] CL0 was the main strategy, but in the last stages of the training CL was preferred.

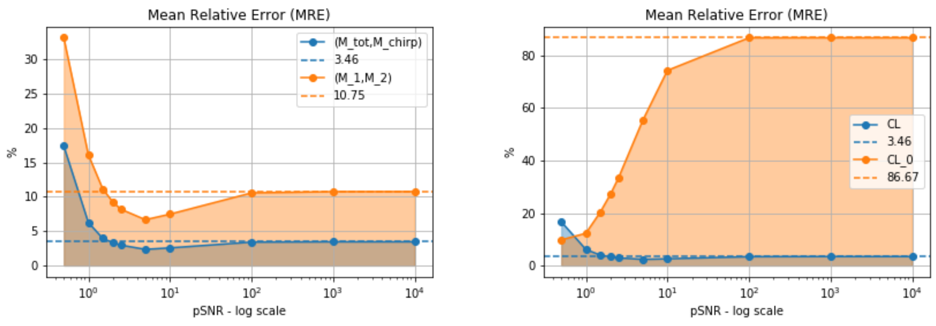


(a)



(b)

Figure 2.3: Test results of the CNN regression over the clean signals. (a): regression over the original labels (M_1, M_2). (b): regression over the redefined labels $(M_{\text{tot}}, M_{\text{chirp}})$.



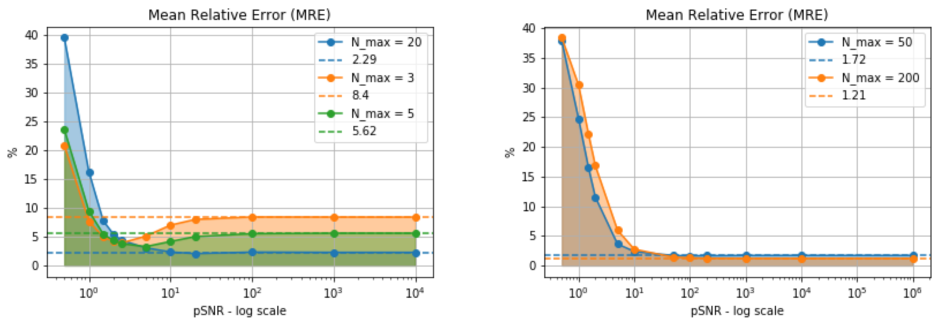
(a) (b)

Figure 2.4: (a): MRE accuracy for training over (M_1, M_2) (orange) and $(M_{\rm tot}, M_{\rm chirp})$ (blue). (b) MRE accuracy for CL versus CL0 training.

It follows non only that the number of samples to which the network is exposed during the training is $n \times$ n_samples, but also that the training epochs and the training time are amplified by a factor n. For example, the training time for the regression over pure signals was $\sim$ 3 mins; for a typical number of iterations, say $n = 10$, and assuming a perfect scaling, the training time will increase to $\sim$ 30 mins. While it is true that 30 mins is not a big deal, there is a compelling reason to push down the training time: our project is just a toy model with only 2 degrees of freedom, but the typical number of parameters of a GW signal is $9 \div 16$. The shape of a realistic dataset (which goes beyond the scope of this thesis) can easily become of order $O(10^{10})$. Therefore, in realistic situations, it is critical to optimize the training time.

To this aim, I implemented an early stopping validation callback: you take aside a 10% random portion of the train_set, call it the validation_set, and use it to early stop the training when the validation loss becomes flat. Flatness is considered achieved when the difference between the validation loss at epochs i and $i + 1$ is less than 10^{-4} for 10 consecutive epochs.

Results I applied the CL strategy with $N_{max} = 5$ and N_i decreasing from 4.5 to 0.1 in steps of 0.5, which corresponds to a total of $n = 10$ iterations. Fig.2.4a shows the test results for training over $(M_{\rm tot}, M_{\rm chirp})$ and over (M_1, M_2): the figure plots the percentage MRE versus the log-scale pSNR . The pSNR ranges from 0.5 to 10^4, the latter corresponding to the limit of pure signal. First, you see that there is a clear benefit in using the $(M_{\rm tot}, M_{\rm chirp})$ representation. Second, there is a minimum in the MRE curve in correspondence of pSNR $= 5$: this is expected because the network was maximally exposed to a pSNR close to $N_{max} = 5$. Perhaps the most important observation is that the curves approach a low MRE plateau in the



(a) (b)

Figure 2.5: MRE vs. pSNR for several choices of N_{max}.

limit of pure signal: this is a spotlight that the hidden layers are actually denoising the signals; indeed, the network was never exposed to such a small noise and the only explanation for such a good accuracy is the occurrence of a denoising.

In contrast, Fig.2.4b shows that naively applying a CL0 strategy results in a very poor final accuracy. This is because the value of pSNR decreases during the training and the network overfits at the local pSNR , without retaining any memory of the previous iterations. The CL strategy fixes this problem by letting pSNR span a range of progressively decreasing lower limit but fixed upper limit.

Fig.2.6 offers a visualization of the model performance via the scatter plots for pSNR $= 2, 1, 0.5$ and 10^6.

How do the results change if you vary N_{max} in the CL strategy? I repeated the training for $N_{max} = 3, 20, 50, 200$, fixing the of iterations to $n = 10$ and decreasing N_i down to 0.1 in uniformly spaced steps. Fig.2.5 shows the MRE when $N_{max} = 3, 5, 20$ (2.5a) and when $N_{max} = 50, 200$ (2.5b). What we see is that, as N_{max} increases, the performances at high pSNR improve but the ones at low pSNR deteriorate; conversely, when N_{max} decreases, the accuracy at high pSNR diminishes while the one at low pSNR does not vary much. Therefore, I conclude that pSNR $= 5$ is a good compromise. (By the way, Fig.2.5b also shows that the MRE has no global minimum for sufficiently high N_{max}, but it falls smoothly to its asymptotic value.)

I also benchmarked the training times as you vary the `epochs`. I considered three `epochs` policies: `epochs` $=$ 100 with no validation_set; `epochs` $=$ 300 with no validation_set; and `epochs` determined by the early stop validation callback, with the only constraint that `max_epochs` $=$ 200. Moreover, in order to measure how the training time scales with n_samples, I repeated the training for a smaller subset (n_samples=500) of the origi-

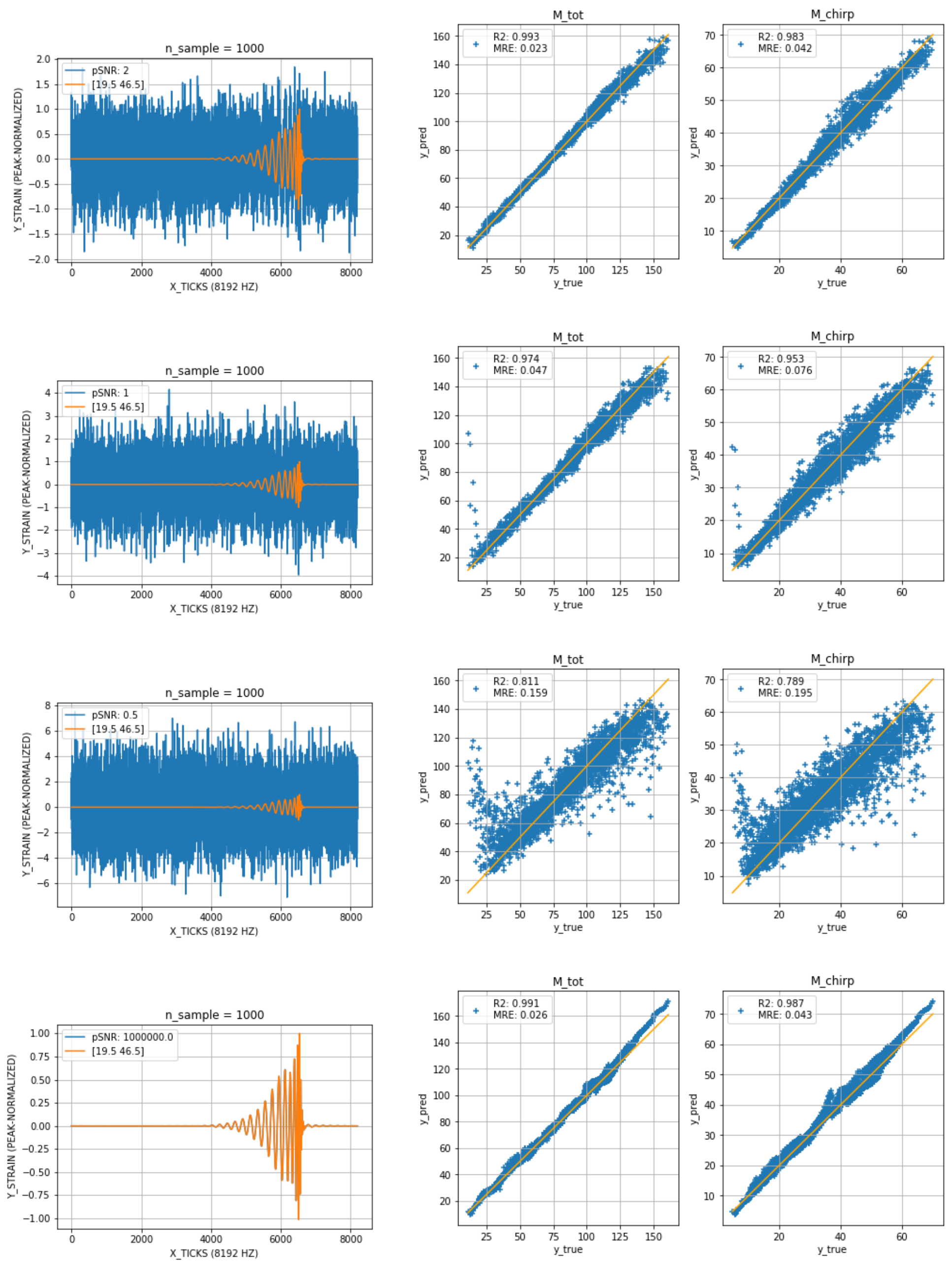
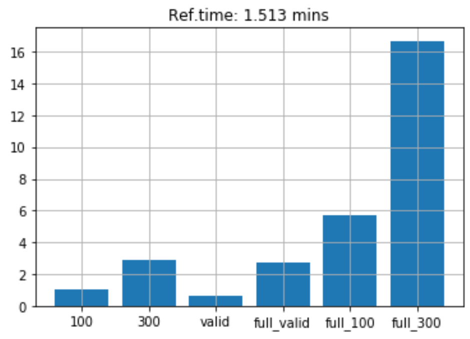
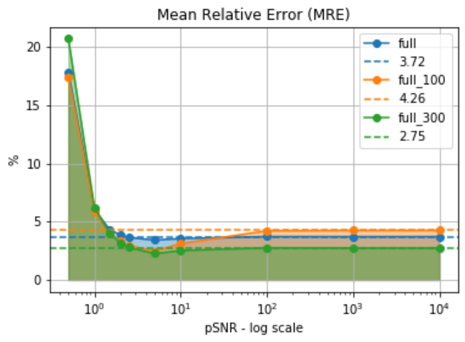


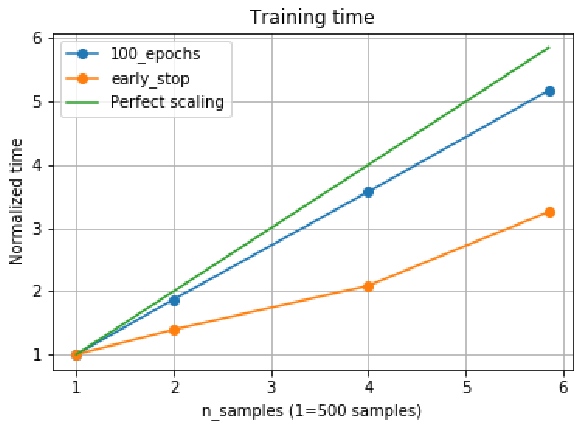
Figure 2.6: Scatter plots for pSNR $= 2, 1, 0.5$ and 10^6. Each scatter plot is accompanied by one example of noisy signal in the test_set (blue time series) and the corresponding clean signal (orange time series), to better visualize the difficulty of the inference. In the last example the noise is so low that the two signals are perfectly superimposed.



(a)



(b)



(c)

Figure 2.7: Benchmarks of the training time and test accuracy for different `epochs` policies. (a): training times for `epochs` $= 100, 300$ and `max_epochs` $= 200$ with early stop validation; the first three bars refer to train_samples $= 500$, while the "full" bars refer train_samples $= 2926$. (b): The corresponding MRE curves for train_samples $= 2926$. (c): Scaling of the training with n_samples, for the case `epochs`=100 and `max_epochs`=200 with early stop validation; the training was measured on a single GPU.

nal train_set. The results are shown in Fig.2.7a. I normalized the times w.r.t. `epochs` = 100 and n_samples = 500. As it is expected, when `epochs` = $100, 300$, the training time scales linearly with both n_samples and `epochs`. On the other hand, when validation callback is implemented, the training time is ~ 2 smaller w.r.t. `epochs` = 100, and I managed to train the model in only ~ 3 mins!. Early stop validation gives also a benefit in terms of sample scaling: as shown in Fig.2.7c, when the model is trained on a single GPU, the training time scales almost linearly for `epochs` = 100, but less than linearly in the presence of validation stop. This is important in perspective, because it allows to scale efficiently the training when a much larger parameter space is explored.

At the same time[2], it is important to check that reduction of the training time does not also results in performance degradation: Fig.2.7b shows the MRE curves for the three `epochs` policies and training over the full train_sample- We see that the difference in prediction accuracy is of the percent order, and therefore the validation callback reduces the training time without sensibly affecting the final performances.

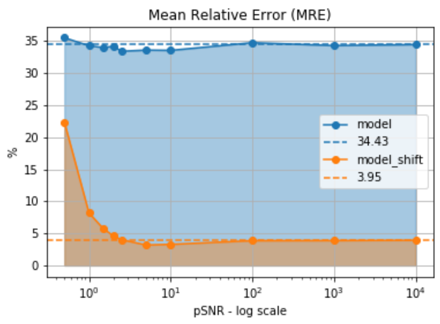
2.3 Refinements of the algorithm

2.3.1 Time translational invariance

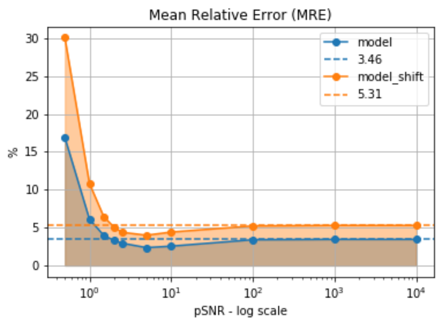
The datasets used in the previous sections suffer from a drawback: the position of the peak of the signals is always at ~ 0.8 seconds. It might be that this information, which in principle should be irrelevant, is percolating in the training and the network is overfitting over it. In order to check the behaviour of the network under peak shifts, I applied a random `numpy.roll` shift to the each signal, with a random left/right rolling between 0% and 5% of the signal length. When necessary, I applied zero padding at the waveform tail. The results are shown in the blue curve of Fig.2.8a: the MRE is roughly constant, with small oscillations around the 35% level, which means that the CNN was overfitting over the position of the signal peak.

This issue can be fixed if the signal peak is randomly shifted during the training. To this aim, I run a modified CL training with random 0-to-5% peak shift. Using the early stop validation policy, the training takes ~ 6 times more (~ 9 mins). The orange curve in Fig.2.8a is the resulting MRE: you see that now the network has become resilient to the position of the peak, and the performances are as good as in the non-shifted case. For consistency, in Fig.2.8b I plotted the MRE corresponding to the un-shifted test_set: the main difference of the new model is a performance degradation at low pSNR , but in both cases the MRE remains within 10% for pSNR $\gtrsim 1$.

[2]I apologize for the different uses of the word "time" in this paragraph!



(a)

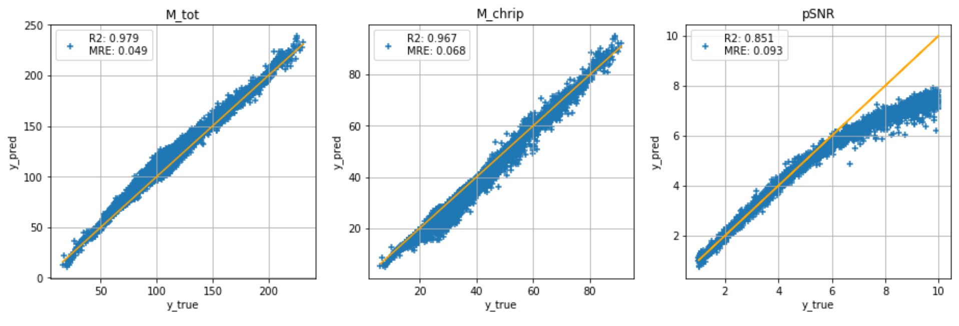


(b)

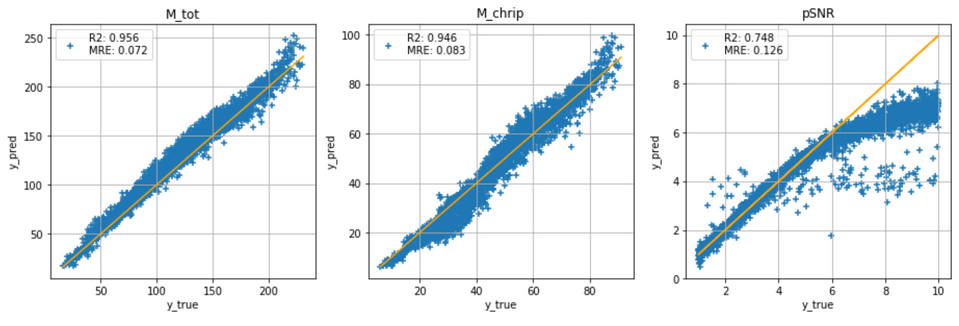
Figure 2.8: (a): Comparison of the test MRE over randomly shifted signals within 5%, for the original model (blue curve) and a new model trained on randomly shifted waveforms (orange curve). (b): MRE of the two models over the original non-shifted test_set.

2.3.2 Estimate the pSNR

The extension of the network to estimate the pSNR is straightforward: from the point of view of the architecture, you just have to add an output node (from 2 to 3) to the final layer; from the point of view of the data preparation, you just have to add a further column to the labels with the corresponding pSNR . During the training, I compactified the pSNR column rescaling it by a factor of 10. The results are shown in Fig.2.9: the scatter plots are relative to the test_set with random noise per sample in the range $[1, 10]$. Fig.s2.9a and 2.9b correspond, respectively, to the cases without and with peak shift. In both cases, the prediction of the pSNR is only accurate for pSNR $\leq N_{max} = 5$: this is expected, because the network was never exposed to larger pSNR . On the other hand, $(M_{\text{tot}}, M_{\text{chirp}})$ are correctly estimated within 10% also for pSNR $\geq N_{max}$.



(a)



(b)

Figure 2.9: Scatter plots for random noise distribution in the range pSNR $\in [1, 10]$. (a): train and test without peak shift. (b) train and test include random 0-to-5 % peak shift.

Chapter 3

Discussion

In this thesis I discussed a deep learning approach to parameter estimation, with a focus on regression over noisy time series. The time signals represent simulated gravitational waves produced by binary black holes orbiting around each other. The main motivation in using deep learning comes from the ability of neural networks to recognize and codify fully nonlinear correlation patterns. I found that the main benefits in terms of performance are: (i) low latency inference; (ii) information compression; (iii) generalization.

— **Low latency inference**
The computational burden of a NN is concentrated on the *offline* training: once the model is trained, inferences for individual signals can be drawed in much less than one second. To be more quantitative, I benchmarked the prediction time on a single CPU: it takes only ≈ 900 μs to make a single inference. Considering that the time window of a signal is 1 sec., we see that deep learning can allow very low latency parameter estimation.

— **Information compression**
NNs occupy a relatively small amount of memory. Indeed, the `.pkl` file containing the network parameters is only 530 kB in size. In contrast, the `.hdf5` dataset I used for training occupies 386 MB; moreover, I generated noisy signals on the run during the training, which resulted in an effective training dataset 10 times larger. This means that the network compresses the information contained in ≈ 4 GB into $\approx 5 \times 10^{-4}$ GB!

— **Generalization**
As I showed in Fig.s 2.4 and 2.6, the network can generalize outside its domain of training, if a proper training strategy is employed. Indeed, thanks to curriculum learning, the hidden layers learn to denoise, and the the NN is able to draw accurate inferences even from pure signals without noise.

Scope for improvements

This work has of course many improvements directions. The final goal, which is beyond the scope of the thesis, is to build a pipeline to process and make inferences from real data. Let me briefly list what I consider the main limitations of the thesis in this sense, and describe what are their possible solutions:

- the network was trained over a 2-dimensional parameter space (M_1, M_2), neglecting the spins and the spatial orientation of the binary;
- I only considered white Gaussian noise;
- I considered samples of fixed time extension, limiting the analysis to only the last 1 second of each GW signal;
- the NN performs point estimate, but it does not output a measure of the inference uncertainty.

The first two are limitations on the size of training template bank. The dataset can be easily extended by varying the other simulation parameters and by adding more realistic noise realizations. In particular, when the orientation w.r.t. the detector is allowed to vary, one can simulate a network of multiple detectors and project the signals over the individual detectors: it would be interesting to study how the accuracy improves by combining the inferences from each detector.

The second limitation comes from the type of NN architecture we are using, namely a CNN. CNNs are standard feed forward NNs admitting only fixed size inputs. However, GW data are recorded as long continuous streams of inputs; moreover, different GWs can cover very different time extents within the input. This calls for an architecture with a more flexible input size. A possible solution is offered by recurrent neural networks (RNN), which are specifically designed to process a series of inputs. A RNN scans a long input of data in smaller steps and, besides forwarding the input through the network, it also saves part of the information into an internal memory. In this way, the information about the early steps is not lost, but in can be used to make more informative operations in later steps.

Finally, uncertainty estimates can be obtained using Bayesian neural networks (BNN). In a BNN the weights are not single numbers, but rather distributions: the goal of the training is to optimize the Bayesian posteriors of the the weights, given the observed training samples. Therefore you can obtain a statistical sample of inferences by picking up weights from their distribution multiple times.

It is worth noting that all these directions have been already explored in the literature with very encouraging results. See, in particular: [15] for addition of more realistic noise; [17] for inference of the sky-location of the signal; [18] for the use of RNN to denoise the signals; [20, 21] for uncertainty estimates using BNN and [22] for an alternative Bayesian approach using variational autoencoders.

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[2] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2 (5):359–366, 1989.

[3] D Shoemaker. Advanced ligo anticipated sensitivity curves tech. *Rep. LIGO-T0900288-v3*, 2010. URL https://dcc.ligo.org/public/0002/T0900288/002/AdvLIGO%20noise%20curves.pdf.

[4] Daniel George and EA Huerta. Deep neural networks to enable real-time multimessenger astrophysics. *Physical Review D*, 97(4):044039, 2018. URL https://arxiv.org/abs/1701.00008.

[5] Benjamin P Abbott, Richard Abbott, TD Abbott, MR Abernathy, Fausto Acernese, Kendall Ackley, Carl Adams, Thomas Adams, Paolo Addesso, RX Adhikari, et al. Observation of gravitational waves from a binary black hole merger. *Physical review letters*, 116(6):061102, 2016. URL https://dcc.ligo.org/public/0122/P150914/014/LIGO-P150914_Detection_of_GW150914.pdf.

[6] Gabrielle Allen, Warren Anderson, Erik Blaufuss, Joshua S Bloom, Patrick Brady, Sarah Burke-Spolaor, S Bradley Cenko, Andrew Connolly, Peter Couvares, Derek Fox, et al. Multi-messenger astrophysics: Harnessing the data revolution. *arXiv preprint arXiv:1807.04780*, 2018. URL https://arxiv.org/abs/1807.04780.

[7] Gabrielle Allen, Igor Andreoni, Etienne Bachelet, G Bruce Berriman, Federica B Bianco, Rahul Biswas, Matias Carrasco Kind, Kyle Chard, Minsik Cho, Philip S Cowperthwaite, et al. Deep learning for multi-messenger astrophysics: A gateway for discovery in the big data era. *arXiv preprint arXiv:1902.00522*, 2019. URL https://arxiv.org/abs/1902.00522.

[8] Eliu Antonio Huerta, Gabrielle Allen, Igor Andreoni, Javier M Antelis, Etienne Bachelet, G Bruce Berriman, Federica B Bianco, Rahul Biswas,

Matias Carrasco Kind, Kyle Chard, et al. Enabling real-time multi-messenger astrophysics discoveries with deep learning. *Nature Reviews Physics*, 1(10):600–608, 2019. URL https://arxiv.org/abs/1911.11779.

[9] Jolien DE Creighton and Warren G Anderson. *Gravitational-wave physics and astronomy: An introduction to theory, experiment and data analysis*. John Wiley & Sons, 2012.

[10] Bruce Allen, Warren G Anderson, Patrick R Brady, Duncan A Brown, and Jolien DE Creighton. Findchirp: An algorithm for detection of gravitational waves from inspiraling compact binaries. *Physical Review D*, 85(12):122006, 2012. URL https://arxiv.org/abs/gr-qc/0509116.

[11] S Babak, R Biswas, PR Brady, Duncan A Brown, K Cannon, Collin D Capano, Jessica H Clayton, T Cokelaer, Jolien DE Creighton, T Dent, et al. Searching for gravitational waves from binary coalescence. *Physical Review D*, 87(2):024033, 2013. URL https://arxiv.org/abs/1208.3491.

[12] John Veitch, Vivien Raymond, Benjamin Farr, W Farr, Philip Graff, Salvatore Vitale, Ben Aylott, Kent Blackburn, Nelson Christensen, Michael Coughlin, et al. Parameter estimation for compact binaries with ground-based gravitational-wave observations using the lalinference software library. *Physical Review D*, 91(4):042003, 2015. URL https://arxiv.org/abs/1409.7215.

[13] Benjamin J Owen and Bangalore Suryanarayana Sathyaprakash. Matched filtering of gravitational waves from inspiraling compact binaries: Computational cost and template placement. *Physical Review D*, 60(2):022002, 1999. URL https://arxiv.org/abs/gr-qc/9808076.

[14] Hunter Gabbard, Michael Williams, Fergus Hayes, and Chris Messenger. Matching matched filtering with deep networks for gravitational-wave astronomy. *Physical review letters*, 120(14):141103, 2018. URL https://arxiv.org/abs/1712.06041.

[15] Daniel George and EA Huerta. Deep learning for real-time gravitational wave detection and parameter estimation: Results with advanced ligo data. *Physics Letters B*, 778:64–70, 2018. URL https://arxiv.org/abs/1711.03121.

[16] Plamen G Krastev. Real-time detection of gravitational waves from binary neutron stars using artificial neural networks. *arXiv preprint arXiv:1908.03151*, 2019. URL https://arxiv.org/abs/1908.03151.

[17] Chayan Chatterjee, Linqing Wen, Kevin Vinsen, Manoj Kovalam, and Amitava Datta. Using deep learning to localize gravitational wave sources. *Physical Review D*, 100(10):103025, 2019. URL https://arxiv.org/abs/1909.06367.

[18] Hongyu Shen, Daniel George, EA Huerta, and Zhizhen Zhao. Denoising gravitational waves using deep learning with recurrent denoising autoencoders. *arXiv preprint arXiv:1711.09919*, 2017. URL https://arxiv.org/abs/1711.09919.

[19] Hongyu Shen, Daniel George, Eliu A Huerta, and Zhizhen Zhao. Denoising gravitational waves with enhanced deep recurrent denoising autoencoders. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3237–3241. IEEE, 2019. URL https://arxiv.org/abs/1903.03105.

[20] Alvin JK Chua and Michele Vallisneri. Learning bayes' theorem with a neural network for gravitational-wave inference. *arXiv preprint arXiv:1909.05966*, 2019. URL https://arxiv.org/abs/1909.05966.

[21] Zhizhen Zhao Elise Jennings Himanshu Sharma Hongyu Shen, E. A. Huerta. Deterministic and bayesian neural networks for low-latency gravitational wave parameter estimation of binary black hole mergers. *arXiv preprint arXiv:1903.01998*, 2019. URL https://arxiv.org/abs/1903.01998.

[22] Ik Siong Heng Francesco Tonolini Roderick Murray-Smith Hunter Gabbard, Chris Messenger. Bayesian parameter estimation using conditional variational autoencoders for gravitational-wave astronomy. *arXiv preprint arXiv:1909.06296*, 2019. URL https://arxiv.org/abs/1909.06296.

[23] He Wang, Zhoujian Cao, Xiaolin Liu, Shichao Wu, and Jian-Yang Zhu. Gravitational wave signal recognition of o1 data by deep learning. *arXiv preprint arXiv:1909.13442*, 2019. URL https://arxiv.org/abs/1909.13442.

[24] Christopher Bresten and Jae-Hun Jung. Detection of gravitational waves using topological data analysis and convolutional neural network: An improved approach. *arXiv preprint arXiv:1910.08245*, 2019. URL https://arxiv.org/abs/1910.08245.

[25] Tito Dal Canton et al. Implementing a search for aligned-spin neutron star-black hole systems with advanced ground based gravitational wave detectors. *Phys. Rev.*, D90(8):082004, 2014. doi: 10.1103/PhysRevD.90.082004.

[26] Samantha A. Usman et al. The PyCBC search for gravitational waves from compact binary coalescence. *Class. Quant. Grav.*, 33(21):215004, 2016. doi: 10.1088/0264-9381/33/21/215004.

[27] PyCBC. v1.14.4. doi: 10.5281/zenodo.3546372. URL https://doi.org/10.5281/zenodo.3546372.

[28] Sascha Husa, Sebastian Khan, Mark Hannam, Michael Pürrer, Frank Ohme, Xisco Jiménez Forteza, and Alejandro Bohé. Frequency-domain gravitational waves from nonprecessing black-hole binaries. i. new numerical waveforms and anatomy of the signal. *Physical Review D*, 93 (4):044006, 2016. URL https://arxiv.org/abs/1508.07250.

[29] Sebastian Khan, Sascha Husa, Mark Hannam, Frank Ohme, Michael Pürrer, Xisco Jiménez Forteza, and Alejandro Bohé. Frequency-domain gravitational waves from nonprecessing black-hole binaries. ii. a phenomenological model for the advanced detector era. *Physical Review D*, 93(4):044007, 2016. URL https://arxiv.org/abs/1508.07253.

[30] Benjamin P Abbott, R Abbott, TD Abbott, MR Abernathy, F Acernese, K Ackley, C Adams, T Adams, P Addesso, RX Adhikari, et al. Gw150914: First results from the search for binary black hole coalescence with advanced ligo. *Physical Review D*, 93(12):122003, 2016. URL https://arxiv.org/abs/1602.03839.

[31] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_2. URL https://doi.org/10.1007/3-540-49430-8_2.

[32] LIGO Scientific, VIRGO collaborations, BP Abbott, R Abbott, TD Abbott, MR Abernathy, F Acernese, K Ackley, C Adams, T Adams, P Addesso, et al. The basic physics of the binary black hole merger gw150914. *Annalen der Physik*, 529(1-2):1600209, 2017. URL https://arxiv.org/abs/1608.01940.