Master in Advanced Physics

Specialty in Astrophysics

Master Thesis

GRAVITATIONAL WAVE DETECTION WITH DEEP LEARNING

Juan Diego Salamanca Cerón

Tutor (1): Roberto Ruiz de Austri PhD
Tutor (2): Alejandro Torres PhD

Academic year: 2021/22

Contents

1 Introduction

On September 14 of 2015, the first gravitational wave (GW), generated by a binary black hole merger, was detected by the Advanced LIGO observatory [1] which confirmed the existence of black holes and marked the beginning of gravitational wave astronomy. This new source of information from the universe, since previously we could only count on electromagnetic radiation, enables us to test general relativity with precision and provides valuable experimental data for multiple areas of physics. Since then, GW detection has become a growing area of research with promising results. We now have three observatories, Livingston and Hanford from LIGO in the United States [2] and Virgo in Italy [3] with multiple gravitational wave event detections and candidates [4], which are expected to cooperate with future GW observatories like a third Advanced LIGO in India [5] and KAGRA in Japan [6] in the coming years.

Currently, the LIGO and Virgo collaboration uses the matched filtering (MF) method to process the data in order to detect the GW from a compact binary coalescence (CBC) in between the experimental noise. The method briefly consists in creating a template bank that describes the waveforms of the merging systems that are expected to be observed and then comparing it the observed signal to find a match [7]. They also take into account the time that the GW takes to get from an observatory to another. The time difference is adjusted and then compared to the other signals. This has proven to be a succesful way of discovering signals from binary systems.

However, MF still has its shortcomings. Since this method requires a complete search in the template bank, the processing speed declines as the banks increase in size. Given that the number of theoretical waveforms is continuously increasing, the processing speed of MF is expected to decline if no suitable adjustments are made [8]. In addition to this, we know that the computational cost of this method increases with the broadness of the detector's sensitivity and the number of observatories, which will most likely increase in the future [9].

This are probably a few of the reasons why, recently, a variety of machine learning methods have been developed in the area of GW detection, [10, 11, 12] with results that will probably keep improving as they have done until now. It is now well known that machine learning has been successful in many different types of tasks which explains why it has become an outstanding candidate for GW detection. Deep learning algorithms, specifically convolutional neural networks (CNNs), have been predominantly researched among the machine learning methods, since they greatly reduce the amount of calculation in the online process [13]. In a nutshell, machine learning consists of feeding data to an algorithm so it can "learn" the behavior of that data and later accurately predict outcomes for similar types of data.

In this work, we focus on comparing the performance to detect GWs of two specific types of CNN architectures. Namely, a U-Net type CNN and a Temporal convolutional Network (TCN). Simply put, the U-net consists on a block of convolutional and downsizing operations applied to the data followed by a block of of slightly different convolutions and operations that increase back the size of the data (not necessarily to its original size). This will be explained in more detail in section 4.6. The reason this architecture was chosen, is because of the success that it has had on other areas like biomedical imaging[14][15], seismic denoising[16] or even for time series segmentation[17], which results of particular interest for the field of GW detection since we also deal with time series. Most CNNs used until now to detect GW limit themselves to have a binary outcome. They receive a signal and predict whether or not there's a GW within the noise[10]. Here, we'll not only determine if there is a GW in the signal, we will also locate it, as done by Gebhard et al. [11], and argue how the U-Net type CNN could succeed to a significant degree at this task. After showing how the U-net works and how it performs with the data, we'll compare it with the TCN.

2 Introduction to Gravitational Wave Astronomy

The detection of GWs is currently being done by the LIGO, Virgo, Kagra (LVK) collaboration. Very basically, they have the design of a Michelson interferometer with arms that measure 3km for Virgo and Kagra [3] [6] and 4km for LIGO [2]. A visualization of the structure of the LIGO detectors can be seen in figure 1. The principle implemented by design on these observatories is based on the way GW interact with matter. Simply put, they stretch objects in one direction and compress them in the perpendicular direction, hence the L shape of the interferometer. The detector then measures the relative length of the arms using the interference pattern created by the laser that goes through the arms. If a GW, whose amplitude (projected onto the detector) is big enough, passes through the detector, it will create a difference in the length of the arms and therefore, a change in the phase of the light that goes through them. The effect of the wave on the detector can be characterized by the following equation [18].

$$\delta L_x - \delta L_y = h(t)L$$

Where L_x, L_y, L are all equal to the designed length of the arms of the interferometer and h is the GW strain amplitude projected onto the detector. The stretching and compression of the GW are both done perpendicularly with respect to the direction in which the wave propagates. That being the case, these effects are maximized when the propagation direction of the wave is perpendicular to plane of the arms of the interferometer and minimized when it is parallel. Consequently, the detectors process the projections of the GW on this plane [1].

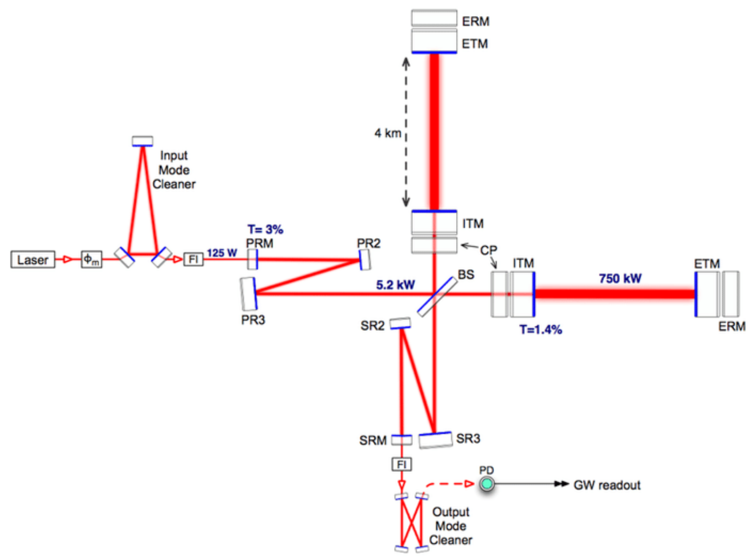

Figure 1: Setup of the LIGO detectors taken from [2]

These detectors have to deal with multiple sources of noise. For example, to mitigate the noise created by the seismic activity of the earth, the detectors have to be well isolated in the horizontal and vertical direction from these perturbations. We also have the gravity gradient (Newtonian) noise. This one is created by seismic surface waves that produce density fluctuations on the Earth's surface. For that reason, some components of the interferometer can experiment changes in the gravitational pull that they feel from the surface of the Earth, causing a disturbance in the signal produced by the interferometry. This problem can be approached by setting an array of seismometers to monitor these effects around the components of the detector that need to be isolated form this noise. With this, a subtraction signal can be developed to reduce the gravity gradient noise. There is also thermal noise, created by dissipation processes, like changes in the refractive properties of the mirrors due to thermal fluctuations or thermal dilation or contraction, in some components of the interferometer [19]. This can be approached by altering the composition and properties of the materials of the pieces that can generate this type of noise. Finally, we also have to take into account quantum noise like photoelectron shot noise or radiation pressure. Shot noise can be mitigated by fine-tuning the power and the wavelenght of the laser and the radiation pressure by increasing the mass of the mirrors or decreasing the power of the laser, at the cost of degrading sensitivity at higher frequencies[20]. Additionally, there may be some isolated cases that generate noise like a tree falling in the woods or some other unpredictable environmental event. The noise from the detectors can be approximated to be stationary and Gaussian while away from transient disturbances, but analyses from the members of the collaboration do account for deviations from this approximation [21]. Nevertheless, a white noise is not a good approximation given the amplitude spectral density of the data, which makes sense with the specific types of noise sources mentioned above. The resarchers have to take all of this into account when searching for the, most likely faint, GW signal.

To have a greater degree of certainty when declaring a detection, the LVK collaboration uses multiple interferometers, similar in design (The two LIGO ones are actually identical). Since GWs are expected to travel at a finite speed, which is the speed of light, there will be a detection delay, in the order of milliseconds, between the interferometers used in the detection. Up until now, signals have been detected using different combinations of the LIGO and VIRGO detectors.

As a consequence, the researchers can have greater confidence establishing a confirmed detection if, after adjusting for this delay, both signals match the same type of gravitational waveform. It also helps to distinguish false positives generated by local events that disturb the signal measured by the detector in ways that look like a GW. This is extremely important because GW signals are very faint which requires the detectors to very sensitive. But this sensitivity also enables them to pick up multiple noise signals from the set-up of the detector itself, or from the environment as described previously [22].

2.1 Sources of GWs

Any acceleration that is not spherically or cylindrically symmetrical produces GWs. This leaves us with four main sources: Continuous, inspiral, burst and stochastic GWs [23]. The continuous GW is produced by systems with fairly constant and well defined frequencies. These can be binary systems of starts or black holes long before the merger or a single star with an irregularity, rotating around its own axis. These are expected to be weak when compared to burst or inspiral GWs. Inspiral GWs are created at the end of a binary system when the two objects merge into one. These objects can be a couple of black holes, neutron stars or a combination of both. As their orbital distance decreases their speed increases, thereby causing the frequency of the GW to increase. Burst GW are the one generated by a short duration unanticipated source. It is hypothesized that this could be caused by a supernova or a gamma-ray burst but little is known about the form of the waves these events would produce. The stochastic GWs would be the gravitational analogous of the cosmic microwave background (CMB). It is possible that in the early stages of the universe a large amount of random and independent events generated GWs creating a cosmic, gravitational wave background. These, like the CMB would be a continuous noise coming from every part of the sky. Unfortunately these are also expected to be very weak and therefore, extremely hard to detect.

For this project we'll use only simulations of binary systems of black holes to train the network. Since these have been detected multiple times and therefore have multiple data on them, it seems a reasonable choice to start training the networks with them. For a more graphic idea of how the wave would look within noise, you may use figure 2 which has been generated with the package ggwd [24] using the effective one body numerical relativity (EOBNR) model (SEOBNRv4) [25] from the LIGO Algorithm Library (LAL).

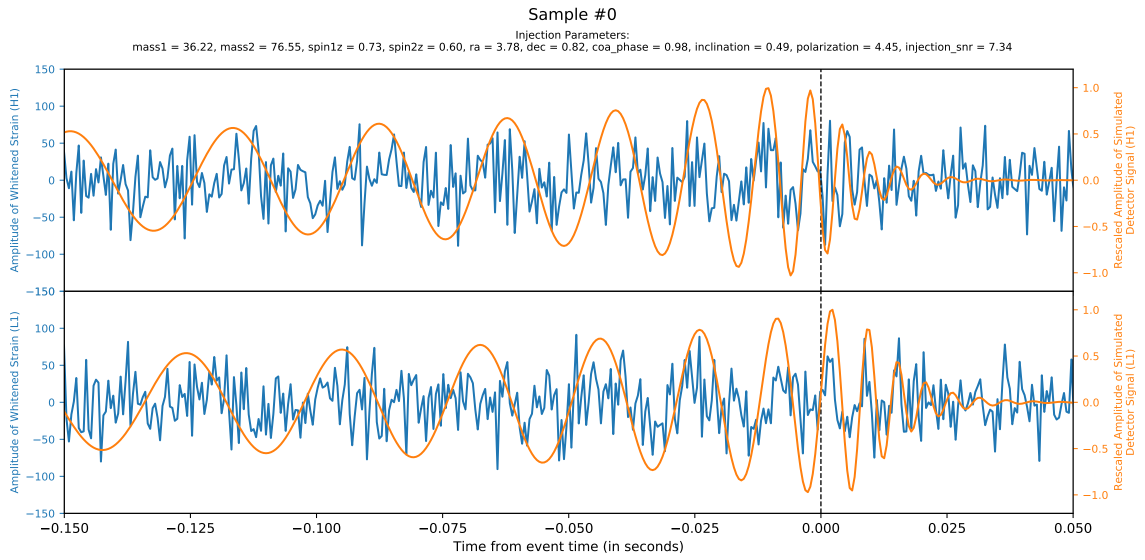

Figure 2: GW simulation generated using the repository from [24]. The strain on the top corresponds to a wave observed by the Hanford detector and the one on the bottom to the Livingston detector. These were simulated with the following parameters: Masses of 36.22 and 76.55 solar masses and z-spin values of 0.73 and 0.60 for a first and second black holes respectively, a right ascension of 3.78, declination of 0.82, coalescence phase of 0.98, inclination of 0.49, polarization of 4.45 and an injection SNR equal to 7.34.

3 Neural Networks (NNs)

In order for the reader to have at least an intuitive idea of how the algorithm works, we'll explain briefly the architecture of a simple NN, a CNN, and finally the U-Net type CNN and the TCN.

But first, it is probably in our best interest to have a basic notion of how a machine learning algorithm learns from the data. There are two types of learning, supervised and unsupervised. Since in this work we only use supervised learning, we'll limit ourselves to explain just that. The algorithm is fed with two types of data, the samples and the labels. For memory saving purposes, we only feed the data to the algorithm divided into smaller subsets called batches. Depending on the memory limitations, one should choose an appropriate batch size to run the code. The samples are the data that the algorithm uses to make a prediction and the labels are the ground truth predictions that the algorithm should ideally produce from said data.

The algorithm then uses the samples to produce an outcome and compares it to the labels. By compare, we mean that it computes a function that describes the error between the prediction and the labels. It then tries to minimize this error function via gradient descent or another method that allows the computer to find the minimum of the function. With this information, the algorithm adjusts the learnable parameters of the network so they serve to minimize the error. As the error decreases, the output of the network becomes more and more similar to the labels. This process will be explained in much more detail in the Backpropagation section.

It is common practice to divide the data in a training, validation and test sets. The training set is the one that contains the data that the algorithm will actually learn from. This means that it will use the data to make a prediction, compute the error with respect to the labels and minimize it by adjusting its parameters. The validation and test sets are used for almost the same thing. The network won't learn from them, which means it wont change its parameters while going through the data. The only difference between these two is that the validation data set is used to keep track of the algorithm's performance. After a certain number of iterations with the training set, the network will go through the validation data set and evaluate its performance. This one tends to be significantly smaller than the training set. The test set is used when the algorithm is done training and is expected to have its highest performance. This one is usually smaller than the training data set but not as much as the validation one. The reason the training set has to be different from the other two is so we can see how the network performs with data it has never seen before. When creating a NN, the objective is build one that performs well enough, whatever that may mean on each specific situation, on real life where new data emerges constantly. Therefore we need to, at least sporadically, be checking how the network performs with data it hasn't trained with to know if it is actually useful.

The validation set is specially helpful to find out when the network is "overfitting" its parameters to the training set. Overfitting happens when the network starts becoming better and better at making predictions from the training set but its performance on unseen data that hasn't trained with, like the validation data, starts to decline. We can interpret this as the network starting to only memorize the training data, therefore becoming extremely efficient at making accurate predictions from it, but when it comes to new data, it only replicates something close to a prediction that would fit the data from the training set. We can see the moment where this starts to happen during the training by looking at the behaviors of the loss function in the training and validation set. The loss function will be explained in more detail in the next sections but for now, it is enough to know that the loss function can be interpreted as the difference between the prediction and the label. Then, the smaller the loss, the more accurate the predictions for the specific data set that is being used. We can see the loss from both data sets, the training and validation one, plotted below.

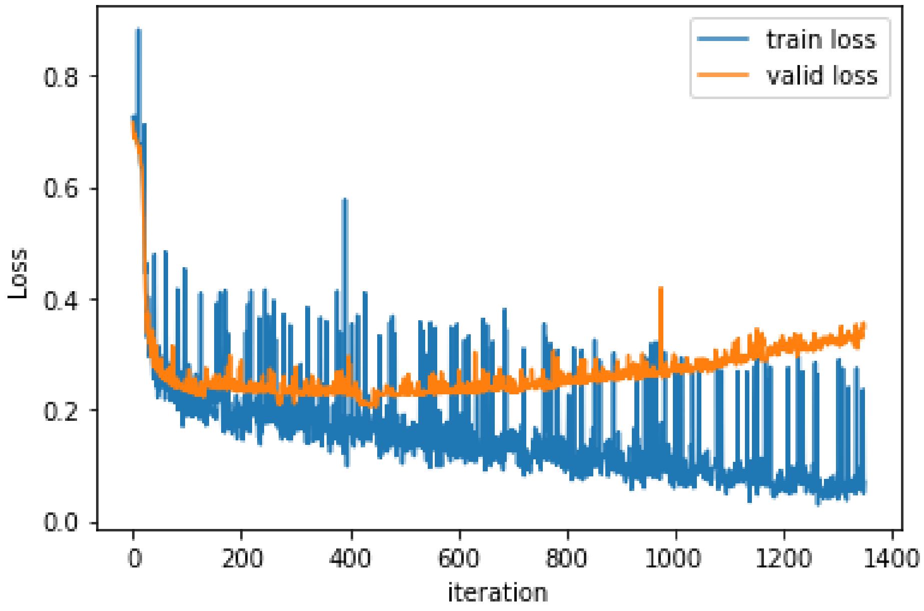

Figure 3: Training (blue curve) and validation loss (yellow curve) obtained using the code of a binary classification network used to detect whether or not there is a GW in a strain of data.[10]

In Fig. 3 you can clearly see how the training loss, on average, keeps dropping while after a certain point the validation loss actually starts to go up. Therefore, from that point forward, the real life performance of the network is starting to decrease while its ability to predict accurately from the training data increases.

3.1 Multi-layer Perceptron

A simple NN consists on a series of layers, each one with a certain number of neurons through which the data is processed. To understand how the network operates lets describe first what a neuron is in this context and how it works.
A neuron receives an input $x = (x_1, x_2, ..., x_n)$ which is multiplied, as a scalar product, by a vector of weights $w = (w_1, w_2, ..., w_n)$. This result is then summed with a scalar bias b and injected to an "activation function" σ [13]. The output of this function is the output of the neuron. There are several activation functions, but the most common ones are the ReLU and Sigmoid function [26] which will be explained later. The weights and the biases are the "learnable" parameters, which means that these are the

ones that change and adjust to the data that is being learned by the algorithm so it can produce accurate results.

As we said before, a NN can have multiple layers, each with multiple neurons as seen in figure 4. For a neuron that belongs to the first layer or the "input layer", the input x will obviously be the input data. If it belongs to one of the next layers, then the input x will be the output of all the neurons in the previous layer. For example, if one of the layers has n neurons, then each of the neurons in the next layer will receive an input of the type $x \in \mathbb{R}^n$, which means that the neuron has to have n weights to process the data. So the number of weights of each neuron in a layer depends on the number of neurons in the previous layer. Usually increasing the number of neurons improves the performance of the network with the cost of more training time required in order to get the network to its peak performance. It is recommendable to start with a low number of neurons and layers and start scaling according to the networks performance through trial and error[27], also considering the available hardware to run the code.

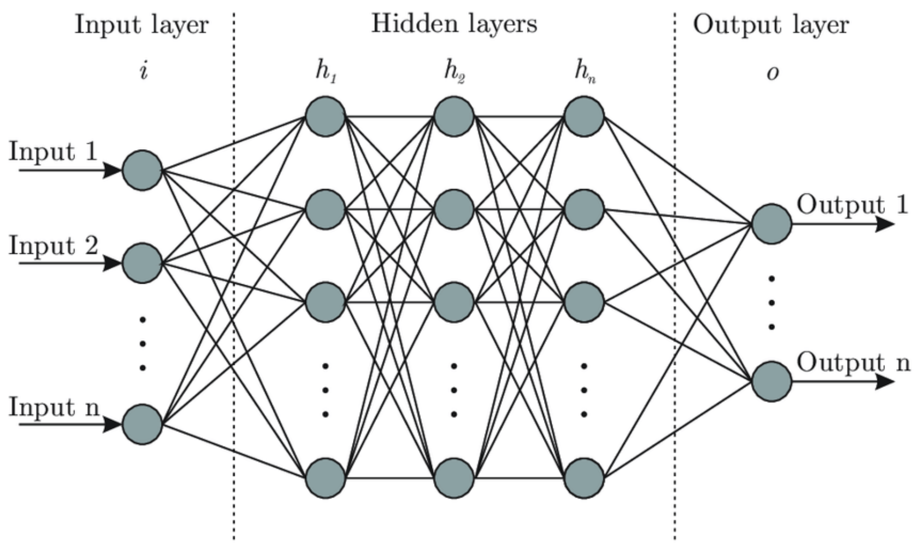

Figure 4: Standard architecture for a deep NN[27]. The term "deep" is used when it contains multiple hidden layers.

The mathematical representation is as follows:
A neuron that receives an input $x = (x_1, ..., x_n)$ will produce an output

$$y = \sigma(wx^T + b). \tag{1}$$

If this neuron belongs to a layer of m neurons, then we can construct a vector $Y = (y_1, ..., y_m)$ with the outputs of all the neurons of the layer. Therefore, Y will be the input of each neuron, with m weights each, in the next layer.

It is common to choose (as we did) the sigmoid function as the activation function on the last layer or "output layer" since it produces a value between zero and one. This is useful when you want a probabilistic result as an output, as one does when trying to detect a GW. We want an output that gives a probability that a GW being present within the noise. The sigmoid function is described by the equation:

$$\sigma = \frac{1}{1+e^x}. \tag{2}$$

Another common activation function is the ReLU function. This one is most commonly used in the input and "hidden" layers, which are the ones between the input and the output layer. The function is described by the following equation:

$$\sigma(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

This was the activation function used in the hidden layers of this work.

3.2 Backpropagation

Backpropagation is the method by which the network actually learns from the data making it an essential concept if the objective is to understand NNs. It is an algorithm that computes the gradients of the error or "cost function" throughout each neuron of the network so we can update their weights and biases in order to minimize the error and consequently, learn from the data [28].

Let us begin by explaining what exactly is this error or, as it was called before, the cost function (which is its common name). It is the function that computes the difference between the ground truth or label and the prediction. There are many examples of cost functions but for a more simple and clear explanation we'll begin with the simplest one, the mean squared error (MSE), described by the equation:

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{3}$$

where y_i is the ground truth and $\hat{y}_i$ is the prediction value.

Imagine for now the smallest network possible, which is one that has only one neuron with only one weight and one bias. Given what we explained in the previous section, the output of the neuron, using a sigmoid activation function, should be:

$$\hat{y} = (1 + exp(z))^{-1} = (1 + exp(w \cdot x + b))^{-1}, \tag{4}$$

where x is the input, w the weight, and b the bias. This leaves with a cost function of the form:

$$(y - (1 + exp(w \cdot x + b))^{-1})^2. \tag{5}$$

In order to numerically approach the minimum of the cost function we would have to update the weight and bias applying the following equations based on the gradient descent method:

$$w' = w - \eta \frac{\partial J}{\partial w}, \tag{6}$$

$$b' = b - \eta \frac{\partial J}{\partial b}. \tag{7}$$

Here, η is the learning rate. It defines how big of a step the parameters take in the direction of the derivative. This should be carefully chosen given that a very small learning rate would make the algorithm too slow but a very big one could make the gradient descent procedure diverge. That is because a very big jump towards the minimum could result in jumping over the minimum and actually getting farther away from it. This is why now a days the learning rate is variable, it updates making itself smaller and smaller as you approach the minimum, thereby decreasing the probability of divergence.

To calculate this derivatives we can use the chain rule of calculus for each variable:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w}, \tag{8}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial b}. \tag{9}$$

Using this same method we can update the parameters across multiple neurons. Lets say we now have a system of two consecutive neurons each with one weight and bias. Naturally, the outputs of the neurons can be described by the equations:

$$\hat{y}_n = (1 + exp(z_n))^{-1} \quad ; \quad z_n = w_n \cdot x_n + b_n. \tag{10}$$

Where $n = 1, 2$ for the first and second neurons respectively. But we know that the input of one neuron is equal to the output of the previous one. Then, we also have the relation:

$$x_2 = \hat{y}_1 \quad \Rightarrow \quad \frac{\partial x_2}{\partial z_1} = \frac{\partial \hat{y}_1}{\partial z_1}. \tag{11}$$

So, in order to update the parameters of the last neuron we only use the equations (6) through (9) but for the parameters of the first neuron we have to consider some extra partial derivatives according to the chain rule that results from equation (11). This would leave us with the expression:

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial z_2}\frac{\partial z_2}{\partial x_2}\frac{\partial x_2}{\partial z_1}\frac{\partial z_1}{\partial w_1}. \tag{12}$$

So now we can update w_1 using the same gradient descent method of equation (6). As you can see, to update the parameters of neurons that are farther away from the final output we just need to keep multiplying the correct partial derivatives that the chain rule dictates in order to find the gradient corresponding to the neuron in question. If we have multiple neurons on multiple layers we would have to take in account the partial derivatives of all the different inputs that go into each neuron.

In practice, the MSE causes some trouble when you're trying to find the global minimum of the function[13]. The details of this matter are outside the scope of this work. Nevertheless we will introduce the function that is most commonly used in deep learning algorithms, the binary cross entropy (BCE) function.

The function is described by the equation:

$$J = -\frac{1}{n}\sum_{i=1}^{n} y_n log(\hat{y}_n) + (1 - y_n) log(1 - \hat{y}_n). \tag{13}$$

As one can see, the function has the desired behavior for outputs between zero and one. If the case is that the target value $y_n = 1$, then the cost will be zero for a prediction of $\hat{y}_n = 1$ and very high for a prediction close to zero since J tends to infinity when $\hat{y}_n$ tends to zero. On the other hand, for a target $y_n = 0$, the cost goes to zero for a prediction $\hat{y}_n = 0$ and tends to infinity for a prediction that tends to one.

4 CNNs

Now that we have a basic understanding of NNs we can explain how CNNs work. CNNs get their name from the convolutional layers that compose them, even though they can have other types of layers as well. Each convolutional layer defines a set amount of filters or kernels that are to be applied to the input data. Since this is an essential topic for this work, we shall explain first how the kernels are constructed and how they work.

4.1 Kernel size and stride

The kernels can be interpreted as the neurons of a CNN. They are the ones that take an input, apply some operations according to their parameters and produce an output that is to be processed by the kernels of the next layer if there is one.

Think of a kernel as a matrix in the case of a $2-D$ image or a vector for the case of the $1-D$ time series. The values of these kernels are initialized randomly and then are adjusted according to the backpropagation algorithm discussed in the corresponding section. What a kernel does, is that it slides on to a certain position of the input and computes the convolution with the overlapping values. For a better understanding consider the example below:

$$\begin{array}{|c|c|c|c|c|}\hline 24 & 15 & 37 & 43 & 71 \\ \hline 13 & 62 & 22 & 87 & 23 \\ \hline 25 & 95 & 46 & 57 & 96 \\ \hline 58 & 95 & 23 & 87 & 56 \\ \hline 75 & 87 & 54 & 39 & 68 \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|}\hline 172 & 106 & 187 \\ \hline 252 & 92 & 231 \\ \hline 277 & 123 & 183 \\ \hline \end{array}$$

The output is obtained by computing the dot product of the kernel (3×3 matrix composed of ones and zeroes) with the overlapping values of the input (5×5 matrix) on each step. By this we mean that on the first step the kernel is overlapped with the top left corner of the input and computes the dot product with the 3×3 submatrix that has a 15 in the top center position and a 46 in the bottom right position. Now the kernel moves to right and repeats the same operation until it reaches the right end of the input, then it moves down and starts all over from left to right. The technical term for these movements of the kernel is "stride". So if the kernel moves two positions on each step we say that the size of the stride is equal to two. The size of the stride is predetermined when initializing the convolutional layer and, as a consequence, the kernels. For the example above, the stride is equal to one which means the mathematical expression for each component of the output can be described by the equation below.

$$O_{i,j} = \sum_{n=0}^{K_h} \sum_{m=0}^{K_w} K_{n,m} \cdot I_{n+i,m+j}$$

Where $O_{i,j}$ are the components of the output, K_h, K_w are, respectively, the height and width of the kernel minus one and $K_{n,m}$ are the components of the kernel. Notice that $i \in [0, I_h - K_h - 1]$ and $j \in [0, I_w - K_w - 1]$ where I_h, I_w are the height and width of the input. For this equation we've taken the indices of the matrices that begin at 0, so if we have a 5×3 matrix, its indices will go from 0 to 4 in the first dimension and from 0 to 2 in the second one. It is very important to clarify that this equation only works on this ideal case where we have a stride value of one and are not concerned with other layer parameters that will be explained in the next sections.

As you may have noticed, the output is a 3×3 matrix. This size is a direct consequence of the kernel size and stride parameters. Since the kernel size is three and the stride is one, the kernel can only move three times to the right before running out of columns in the input matrix. The same applies when moving down, it can only do it three times before running out of rows. On the first step, the kernel would overlap with the first three columns and the first three rows. After one stride to the right, it would overlap with the columns $2, 3, 4$ and then $3, 4, 5$ keeping the rows the same. Thereafter, it would return to the columns $1, 2, 3$ but now one row down, i.e. overlapping with the rows $2, 3, 4$. This is repeated until the kernel goes through every component of the input matrix. As said before, on each of these steps the convolution is computed with the overlapping values of the input.

This is how kernels from convolutional layers are commonly used for 2-D data. For a 1-D time series is even more simple because the strides can only be taken from left to right, as it can be seen in the example below:

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 2 & 4 & 5 & 7 & 3 & 9 & 4 & 6 \\ \hline\end{array} \quad * \quad \begin{array}{|c|c|c|}\hline 0 & 1 & 1 \\ \hline\end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|c|c|c|}\hline 9 & 12 & 10 & 12 & 13 & 10 \\ \hline\end{array}$$

Here we have a 1-D kernel of size 3 and also stride 1 applied to an input of size eight. So, in the first step it will overlap with the first three values of the input vector and take 5 strides to the right computing the convolution on each step. Counting the first one, this gives us 6 convolution computations and thereby resulting in an output vector of size equal to 6.

4.2 Padding

Padding is a tool used in convolutional layers to alter the dimensions of the input before applying the kernel to it. It surrounds the input with some predetermined values before computing the convolutions. Usually, the default values of the padding are only zeroes. This is how a padding of size one filled with zero values would look like in the 2-D example shown previously:

0	0	0	0	0	0	0
0	24	15	37	43	71	0
0	13	63	22	87	23	0
0	25	95	46	57	96	0
0	58	95	23	87	56	0
0	75	87	54	39	68	0
0	0	0	0	0	0	0

This is very useful if you want to alter the dimension of the output without altering the information of the input. For example, if we applied the the same 2-D kernel used before we would get an output of size 5×5. Since the input matrix's dimensions increased by two, so does the output's. When the kernel is set to add padding such that its size is equal to the size of the kernel minus one, we call it "full padding".For a 1-D vector the padding would have the following shape:

0	2	4	5	7	3	9	4	6	0

Analogously, if we applied the same 1-D kernel used before, we would get a size 8 output instead of 6.

Using the padding filled with zeroes allows us to, for example, have an output of the same size as the input while getting no contribution from the padding, which can be desirable depending on the situation.

4.3 Dilation

Dilation can be understood as a way of expanding the kernel so it covers more ground on the input without increasing its size and therefore increasing its parameters, making the network heavier. The size of the dilation is the the space between the input values that the kernel is overlapped on. A dilation value of 2 for a size two kernel in our 2-D input example from before would look like this:

$$\begin{array}{|c|c|c|c|c|}\hline \colorbox{lightgray}{24} & 15 & \colorbox{lightgray}{37} & 43 & 71 \\ \hline 13 & 63 & 22 & 87 & 23 \\ \hline \colorbox{lightgray}{25} & 95 & \colorbox{lightgray}{46} & 57 & 96 \\ \hline 58 & 95 & 23 & 87 & 56 \\ \hline 75 & 87 & 54 & 39 & 68 \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 0 & 1 \\ \hline \end{array}$$

The gray squares in the input matrix would be the values that would be overlapped with the kernel. So the convolution would be computed with the kernel and the 2×2 submatrix composed by the gray squares.

As before the kernel would take strides to the right and then down while holding the same configuration. So you may have concluded that this would return a 3×3 output even though the kernel size is now 2. Because of the dilation, the kernel can only take two strides to the right and down, given that the stride value is still one.

For the 1-D case it would take the form:

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline \colorbox{lightgray}{2} & 4 & \colorbox{lightgray}{5} & 7 & 3 & 9 & 4 & 6 \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|}\hline 0 & 1 \\ \hline \end{array}$$

The convolution would be computed with the 2 and 5 values, then with 4 and 7, 5 and 3, until going through the whole input vector.

Inter-layer Communication:
Here, the filters can be seen as the neurons for the conventional NN. So if one layer has n filters, then each filter on the next layer will have to process the n outputs of the previous layer. If the next layer has m filters then each one of these will take the summed n outputs of the previous layer as input and will produce one output. Since each one of the m filters produces one output, the layer will produce m outputs. This repeats all the way to the output layer. The number of inputs that a layer gets is also called the input channels and the number of kernels or outputs that it produces is called the output channels.

4.4 Pooling layers

CNNs usually contain also pooling layers. These layers reduce the dimension of the data by taking either the average (Average Pooling) or the maximum value (Max Pooling) of the elements of the data that each filter processes at a time. So, depending on the size of the filter it will take in the amount of data according to its size and return only one value that would be the average or the maximum value depending on the type of the pooling. Let us reuse the examples of the previous sections using Max Pooling:

$$\begin{array}{|c|c|c|c|c|}\hline 24 & 15 & 37 & 43 & 71 \\ \hline 13 & 63 & 22 & 87 & 23 \\ \hline 25 & 95 & 46 & 57 & 96 \\ \hline 58 & 95 & 23 & 87 & 56 \\ \hline 75 & 87 & 54 & 39 & 68 \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|}\hline 95 & 95 & 96 \\ \hline 95 & 95 & 96 \\ \hline 95 & 95 & 96 \\ \hline \end{array}$$

For the 1-D example it would be:

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 2 & 4 & 5 & 7 & 3 & 9 & 4 & 6 \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|c|}\hline & & \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|c|c|c|}\hline 5 & 7 & 7 & 9 & 9 & 9 \\ \hline \end{array}$$

For these pooling layers we also assumed a stride value equal to one. It might be worthy to clarify that the $*$ used between the matrices doesn't mean a convolution operation in the case of the pooling layers. Here we just use it to show that the filter is being applied to the input.

4.5 Transposed convolutional layer

These are the layers are not very commonly used in CNNs but they happen to be a very useful tool to increase the size of the data in the same proportions that a conventional convolutional layer decreases it. We will see why this is appropriate for our work in the next section where the U-Net will be explained. For now we'll stick to general transposed convolutions and how they operate. The main difference with the standard convolution is the way that the output is constructed. The way it is done is as follows: First, we multiply each of the values of the kernel by the first (top left) component of the input. This will naturally give us a matrix that has the same size as the kernel. Now we do the same with the next value of the input, that is, the one on the right with respect to the first one. This will give us another matrix with the same size as the kernel, just as before. These matrices need to be overlapped so we can sum the overlapping values. The way we put them together is by shifting the second matrix one element to the right so they overlap on every element except the ones on the far left for the first matrix and the ones the far right for the second matrix. This is better understood with the following example:

$$\begin{array}{|c|c|}\hline 3 & 4 \\ \hline 6 & 8 \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & 1 & 0 \\ \hline 0 & 0 & 1 \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|c|}\hline 0 & 3+0 & 0+4 & 0 \\ \hline 3 & 3+4 & 0+4 & 0 \\ \hline 0 & 0+0 & 3+0 & 4 \\ \hline \end{array}$$

It is very important to state that the example above is NOT the final result of the transpose convolution, it is only the result of the first to steps described in the previous paragraph. Notice how the matrix is made by overlapping the kernel multiplied by 4 shifted one element to right with the matrix that results from multiplying the kernel by 3, analogously to the input where the component whose value is 4 is one element to right with respect to the one whose value is 3.

Now, to obtain the final result we just have to keep applying this logic for the other two components, constructing the output with their respective matrices and corresponding shifts in position. The final result would take the form:

$$\begin{array}{|c|c|}\hline 3 & 4 \\ \hline 6 & 8 \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & 1 & 0 \\ \hline 0 & 0 & 1 \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|c|}\hline 0 & 3+0 & 0+4 & 0 \\ \hline 3 & 3+4+6+0 & 0+4+0+8 & 0+0 \\ \hline 0+6 & 0+0+6+8 & 3+0+0+8 & 4+0 \\ \hline 0 & 0+0 & 6+0 & 8 \\ \hline \end{array}$$

For the one dimensional case we can just consider the logic from the first two steps since we can only move from left to right.

4.6 U-Net

The U-Net is a particular type of CNN. It uses convolutional and pooling layers but, unlike typical CNNs, it also uses 'up-sampling' layers. These, as used in this work, increase the dimensions of the input by a predetermined scale factor. We can consider the U-Net divided in three main blocks. The descent block, the bottom block and the ascent block. The descent block is very similar to an ordinary CNN, it consists of convolutional and pooling layers. On the bottom block there is only convolutional layers. The main difference lies in the ascent block. Here is where we use the up-sampling layers.

The up-sampling layer acts according to a scale factor that determines by how much the input's dimensions will be increased. There are also many types of up-sampling, meaning, many ways to fill in the new components of the input created by the increase in dimensions. For this work it is enough to explain the only one we used which is the default one, "nearest" up-sampling. This one just replicates the values that the original matrix had and adds them to the to the new, surrounding components. For a better understanding of what this layer does, let us use again a graphic example:

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|c|}\hline 1 & 1 & 2 & 2 \\ \hline 1 & 1 & 2 & 2 \\ \hline 3 & 3 & 4 & 4 \\ \hline 3 & 3 & 4 & 4 \\ \hline \end{array}$$

This would be the result of applying an up-sampling layer of scale factor two to a 2×2 input matrix with the values shown above. As expected, the 1-D example is even more simple since it only expands in one dimension. Every value of the 1-D input would just be replicated the number of times indicated by the scale factor.

After increasing the dimension of the data we then concatenate it with the corresponding layer from the descent block as visualized in figure 5.

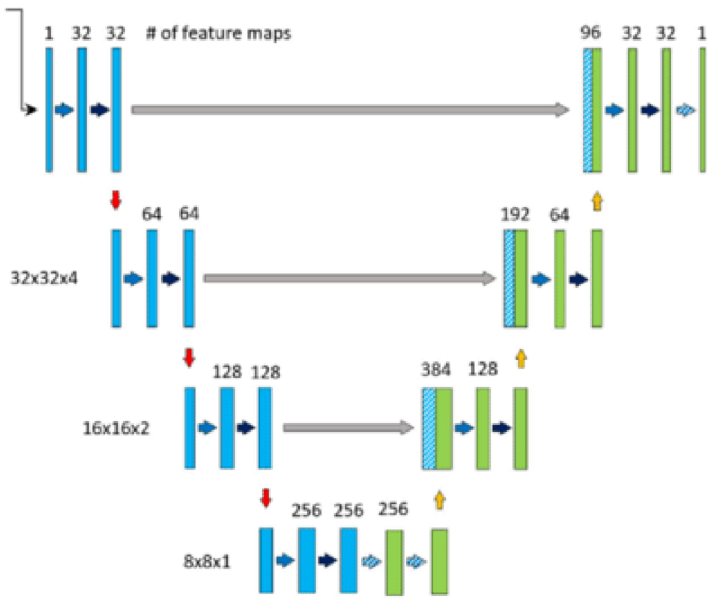

Figure 5: Image representation of the U-net architecture for a 2-D type of input [15]. The numbers on top of each rectangle are the number of feature maps or the number of inputs going into the next layer and the ones on the side represent the size of the data. The horizontal arrows on the descent, for our work, would represent the 1-D convolution layers and the red arrows the pooling layers. On the ascent the horizontal arrows would be the transposed convolution layers and the yellow ones would be the up-sampling layers. Finally the grey arrows would represent the concatenations done after each up-sampling.

To know which is the corresponding layer, you have to picture the layers of the algorithm in a "U" shape, hence the name. The first layer would be the left tip of the "U", which starts the descent block by downsizing the data with the pooling layers. The lowest point of the U would be the bottom block and from there until the right tip we would have the ascent block increasing back the size of the data. Therefore the output from the first convolutional layers, before the pooling one, would be concatenated with the output of the last up-sampling layer, before the last convolutional layers. This way each pooling layer has a corresponding up-sampling layer that somewhat reverses the effect on the size of the data. The concatenation of the data increases the number channels that the next convolutional layer has to take but we program it to output the same number of channels that are present in the corresponding ascent layer.

4.7 Temporal Convolutional Network (TCN)

The TCN is a type of fully convolutional network, which means it only has convolutional layers. This network is constructed with a convolutional layer as input and output layer and a series of causal convolutional blocks (CCBs) as hidden layers. Each CCB has a convolutional layer with a predetermined dilation and an asymmetrical padding calculated from the formula

$$padding = (k_s - 1) * dilation, \tag{14}$$

where k_s is the kernel size. This is followed by a crop of the output where we remove the last elements of the series and this is activated by a Leaky ReLU function [29]. These three operations are repeated once on each CCB, which means that each CCB has two convolutional layers, two crops and two Leaky ReLU functions. The way the leaky ReLU function works is described by the following equation:

$$\sigma(x) = \begin{cases} x, \ x > 0 \\ m \times x, \ x \leq 0 \end{cases}$$

where $m \in (-\infty, 0)$ is a negative slope specified as an argument of the function.

The reason these blocks have the word "causal" in their name is because it avoids taking information from the future to compute a convolution [30]. Since we are dealing with time series as input data, depending on where we are standing on the series we can consider the information on the left as the past and the one on the right as the future. Therefore when we compute a convolution with kernel of size 3 with no dilation for example, the first element of the output would have information of the first, second and third elements of the input. So this first element would be constructed using information from the future i.e. the second and third elements of the input. So you can see how this problem is fixed with the asymmetrical padding described above. The padding is only used on the left side of the series and its value is calculated such that the first element of the output will only be produced with the first one of the input and the values of the padding (set to zero by default). Extending this logic, you can see how each output element will only be constructed by the corresponding input element in the present and some others from the past (two of them in case of the kernel of size 3). You can see an explicit example of how a causal convolution produces its output in figure 6 below:

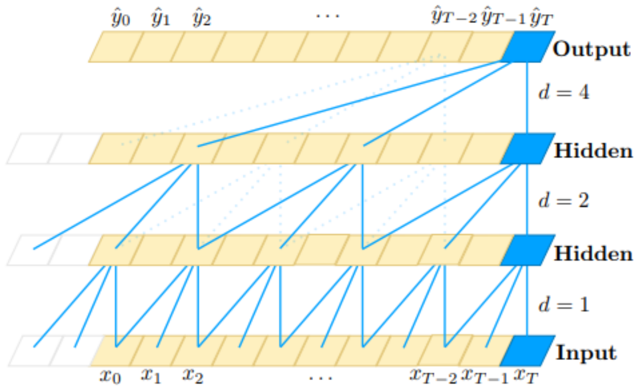
$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$ ... $\hat{y}_{T-2}$ $\hat{y}_{T-1}$ $\hat{y}_T$
Output
$d = 4$
Hidden
$d = 2$
Hidden
$d = 1$
Input
x_0 x_1 x_2 ... x_{T-2} x_{T-1} x_T

Figure 6: Image of the inputs and outputs of a causal convolution with different values of dilation and a kernel size of 3. Image taken from [30].

In summary, the kernel will just start off using only the first element of the series and move on from there. The TCN has many uses and other features that could be explained but that is outside the scope of this work since we only want to provide a very basic technical notion of how these networks work.

5 The U-Net for GW Detection

Now that we've, hopefully, clarified the basic notions of the algorithms that we use in this work we can start explaining how our's works. Starting with the U-NET, we choose to have three levels in total, two ascent and descent blocks and, of course, one bottom block. Each descent block is made up by two convolutional layers, the first one increases the number of channels while the second one keeps them the same, and one MaxPooling layer. On the bottom block we have two convolutional layers and two transpose convolutional layers. These last two, are meant to restore the small downsizing of the data that the ordinary convolutional layers cause. On the ascent block we have one up-sampling and two transpose convolutional layers, the first one receives as input the already concatenated data, and reduces the number of channels so it matches the one on the corresponding ascent layer. The second one decreases again the number of channels for the same purpose. All the layers are followed by a ReLU activation function except for the pooling and up-sampling layers, which are not followed by any activation function, and the last convolutional layer which is followed by the sigmoid function.

Therefore, the output of the code, for each temporal series (the input), is another one with the exact same dimensions but with values $\in (0, 1)$. The purpose of the algorithm is to output a vector that contains zeroes where there is no GW and ones where there is, i.e. a segmented form of the input. By segmented we mean that we separate two distinct pieces of the strain of data, the noise and the GW signal. In order to achieve this, we apply a smoothing of the data and a simple criterion to turn the values of the output into a binary form. Any value less or equal than a half is converted into a zero while the others are converted into ones. It is probably important to clarify that this criterion is not applied when the algorithm is learning. The reason for this is that, if we were to apply it while training the network, an output of 0.51 would then have the same error, or loss, than an output of 0.99 for a label of 1, since both would be turned into a one, which is not desirable. Therefore the criterion is only applied when measuring accuracy and other performance metrics while on validation or testing. All of these post-processing procedures will be discussed in more detail in following sections.

5.1 Layers' Parameters

In this section we'll describe some specific parameters of our code. Characteristics like the number of channels, kernel size, pooling and up-sampling parameters.

Starting with the input layer, we have two input channels which receive one strain of data each. The data is simulated such that we have a group of strains that represent signals measured from the Hanford observatory (H1) and another coming from the one in Livingston (L1), each one taken care of by one input channel. On the other hand, it has a kernel size of 10 and 64 output channels that are followed by another convolutional layer with the same number of input and output channels (which obviously has to be 64 to match the previous layer) but with size 8. After, we have a Maxpooling layer of size and stride value of 2. This cycle of two convolutional layers and on pooling layer is repeated while doubling the number of channels after each of the two pooling layers, leaving us with 256 channels in the bottom, after the last convolutional layer. Now we begin the ascent by "reversing" what was done in the descent. Remember that, because of the size of each convolutional kernel the size of the output is slightly smaller than the input. So, by reversing we mean decreasing the number of channels and increasing back the size of the data. We start by applying two transpose convolutional layers. The first one cuts in half the number of channels and the second one leaves them the same, analogous to what is done in the descent part. Subsequently, an Up-sampling layer increases the size of the data by a factor or two. The output is then concatenated with the corresponding output of the descent block. This is repeated until reaching the output layer which has 64 input channels and one output channel for the prediction time series.

For both this U-net and the TCN we used the Pytorch libraries to build the network and implement a system of checkpoints[11] so the weights and the biases would be saved in case there was any failure that wiped the current state of the network from the server. To run the code, we used Google Colab in connection with Drive to save all the relevant files, protecting them from whenever the session expired, erasing any files in it.

6 The TCN for GW wave detection

As with the U-Net, an explanation of the specific TCN used in this work is necessary. The network can be divided into three basic parts the input layer, the temporal convolutional layer, subdivided in several CCBs, and the output layer, all of them having in common a stride value of one. Since the network receives the same couple of strains from H1 and L1, the input layer has to have two input channels to process them. In here, we start off with 512 kernels, or output channels, of size one. The number of kernels per layer will be maintained throughout the whole network until, of course, the output layer, also of size one, which has to have only one output channel that is followed by a sigmoid function to return the final prediction as a time series of equal size as the input.

Each of the CCBs are composed of two of these: a convolutional layer with a kernel size value of 2 and with a padding that is calculated according to the equation (14) in section 4.7, a cropping layer which only cuts off the part of the output produced by the padding at the right of the time series so only the left one remains and a LeakyReLU activation function. In order to preserve the size of the outputs equal to the one of the input we have to set the number of elements that need to be cut off from the output of the convolution equal to the value of the padding. Overall, this means that on each CCB we have a convolution, a cropping and a LeakyReLU followed by this three same items again in the same order. The only thing that differs one CCB from another is that the next one will have a different dilation value, which will be doubled with respect to the previous one and thereby doubling the padding value and the number of elements removed from the right side of the output as well. The first CCB will begin with a dilation value of 1, which means that this procedure of doubling the dilation that is repeated after each CCB, will reach 1024 after the 11 blocks that the TCN has. So, in summary, we have 11 CCBs, each one with two convolutional layers with kernels of size two, two cropping layers that take out the output produced by the right hand side padding and two LeakyReLU functions, one for the output of each convolutional layer after being cropped. And finally each CCB will have a dilation and padding value that is double the one on the prior CCB.

Given that the TCN has exactly the same type of output than the U-Net, a time series of equal size as the input with values inside the open interval of $(0, 1)$, we can and should apply the same post-processing that we do with the U-Net results. We also put the outputs through the smoothing and the threshold criterion talked about in the previous section in order to assess the network's performance.

7 Input Data

Before we can understand how the network's performance is measured, it is best to lay a foundation, without going into too much detail, of how the data was generated and how the labels are created according to that data in order to know why the performance metrics that we use make sense for this specific type of algorithm.

7.1 Technical features

The strains of data that we feed the algorithm can be divided into two categories, noise and injections. The noise strains are the ones that do not contain any GW signal, while the injection ones are the sum of a noise background and an injection, the GW signal. This signal has a point of maximum amplitude where we consider to be the location of the injection. With this information the label is constructed by taking a symmetric interval around this location, in this case, the interval is of $0.2s$. We take each strain to be of 8 s with a sampling rate of 2048 values per second, which means that each strain has 16384 elements. All the values within the interval will be equal to one and the rest to zero. With the current sampling rate that would mean that we have 409 one values on each strain with injection. As said before, in the data there will also be present strains of pure noise. This is very important since, in real life, the algorithm will probably be dealing mostly with pure noise which needs to learn how to recognize. Also, if we were to only include samples with injections, the algorithm could just learn to solve a very trivial problem where it only has to assign 1 s on a certain spot of the strain regardless of the signal. Naturally, the labels that correspond to these noise samples will be filled with only zeroes.

7.2 Data Generation

The noise background was obtained from real recordings from the first observation run of LIGO (O1). In order for the data to be used to train and test the network it had to comply with the followings requirements[11]. Since we need to be able to train the network with two input channels, only data that was available from H1 and L1 could be used. The data had to be of at least CBC_{CAT3} quality according to the GWOSC[31] [11] and it could not contain any GW signals, either synthetically injected or from real detections. To get these desired noise samples a GPS time is randomly selected inside the observation run data. If a symmetrical interval around it fulfills the criteria discussed above, then it is considered a valid time. Afterwards, the interval is down-sampled from 4096 Hz to 2048 Hz in order to save memory.

In parallel, the parameters of the waveforms to be injected in the noise are randomly sampled. The waveforms are simulated, using the ggwd software, exclusively from binary black hole (BBH) systems with parameters like the masses of the black holes, the z components of their spin, polarization, declination, inclination, right ascension, injection SNR and coalescence phase angle.

The values of these parameters were sampled from the following intervals. The masses ranged between 10 and 80 solar masses, the spins were chosen independently from 0 to 0.998, the polarization was sampled from the interval $[0, 2\pi]$ and the SNR from $[5, 20]$. On the other hand the right ascension and declination were sampled together from a uniform distribution over the sky. The coalescense phase and the inclination are the angles that determine the location in the sky of the observatory as seen from the reference frame of the source of the GW. This reference frame is set in such a way that the z axis is perpendicular to the plane in which the black holes orbit each other. The simulations of the LALSuite return two time series for the sampled parameters, the two polarization modes of the GW. With the help of PyCBC one can calculate the projection onto the antenna pattern, which is a function that describes directional sensitivity of the detector, of H1 and L1 for the given location and corresponding polarization angle. After, the time offset between detectors is corrected according to the relative position of the source.

Consequently, the signals are injected into the background noise such that the injection is centered on the noise interval. The optimal match filtering SNR, which is the maximal SNR possible resulting from using the time inverted series itself as a filter, and the network match filtering SNR are calculated (NOMF-SNR). The second one is used to determine the scale factor λ by which the waveform needs to be multiplied to ensure the injection has the desired SNR. Modifying the SNR by multiplying the signal by this factor λ can be interpreted as moving the source closer of further from the observer. Now that that signal has the desired SNR it is added to the noise background.

Afterwards, the result is then whitened with PyCBC and high-passed at 20 Hz. For the whitening, we first have to Fourier transform the data to take it to the frequency domain. Now, we divide data Fourier coefficients by the amplitude spectral density of the noise to ensure that each frequency bin in the data has equal significance [21]. Finally, the data is inverse Fourier transformed to return to the time domain. The only thing left is to crop the data strain to the correct length, which is 8 seconds. This whole process can be visualized in figure 7 for a more clear understanding.

Finally, we are left with data that is used to train and evaluate the algorithm. For the training data set we have 32768 strains from which 3/4 are injection samples and the rest are noise samples. For the validation and test set, we have 4096 and 16384 respectively. Both of these last data sets have the same proportions in terms of noise and injections as the training data set.

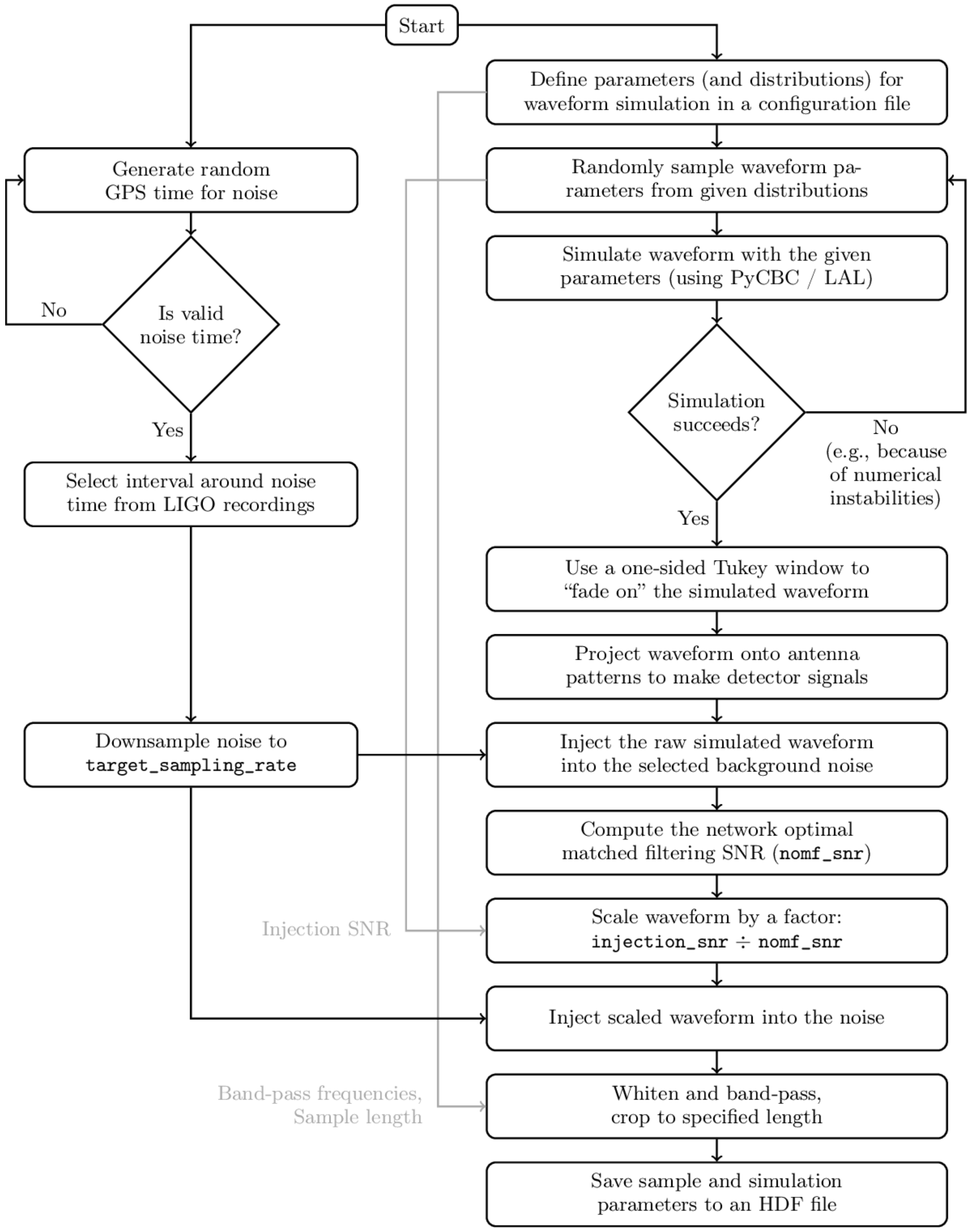
Start
Define parameters (and distributions) for waveform simulation in a configuration file
Generate random GPS time for noise
Randomly sample waveform parameters from given distributions
Is valid noise time?
No
Yes
Simulate waveform with the given parameters (using PyCBC / LAL)
Simulation succeeds?
No (e.g., because of numerical instabilities)
Yes
Select interval around noise time from LIGO recordings
Use a one-sided Tukey window to "fade on" the simulated waveform
Project waveform onto antenna patterns to make detector signals
Downsample noise to target_sampling_rate
Inject the raw simulated waveform into the selected background noise
Compute the network optimal matched filtering SNR (nomf_snr)
Injection SNR
Scale waveform by a factor: injection_snr ÷ nomf_snr
Inject scaled waveform into the noise
Band-pass frequencies, Sample length
Whiten and band-pass, crop to specified length
Save sample and simulation parameters to an HDF file

Figure 7: Data generation flow chart by Gebhard et al[11]

8 Performance Metrics

For the binary classification problem, other authors have used the accuracy and the validation or training loss as performance metrics for the network[10]. By accuracy, we mean the fraction of output elements that match the label over the total amount of elements in the output prediction. On the other hand, the losses are the result of the error function between the label and the prediction that the algorithm calculates in order to update the parameters and decrease the error.

Even though the accuracy increases and the validation loss decreases in the training process, using these metrics can be misleading. The problem with this approach when we are trying to locate the signal within the time series is that, because of the huge amount of values on each series, we can have an apparently very good accuracy (beyond 95%) and very low validation loss (below 0.1) and still have an ineffective network that doesn't recover the signal and/or has a a significant amount of false positives (cases where the network's output had a value of 1 where there was no signal). This is because, the great majority of the values on the label are zeroes, since there is only one GW signal located in a very small time interval within the time series. Hence, the algorithm could very well be only predicting vectors full of zeroes, with or without some sporadic ones regardless of the location of the signal, and still get a very high accuracy. Therefore, we'll be using the detection ratio (DR) and the inverse false positive rate (IFPR) like in [11].

8.1 Detection ratio (DR)

The detection ratio will be calculated by dividing the number of signals recovered by the network by the total number of injections in the dataset. The way we count detections is by taking all the intervals of 1 s in the output, after smoothing and thresholding the time series. The smoothing consists in creating a "window" which is a 1-D array that has the same value on each of its components and applying the convolve operation of numpy with with the network's output. Thresholding refers to the act of applying the criterion discussed in the beginning of section 5 where we turn the output's components in zeroes or ones. With this done, we find the ones interval center t_c and check if it is close enough to the injection center. This will be marked as a candidate for detection if it passes the following test:

$$|t_c - t_g| < \delta t, \tag{15}$$

where t_g is the ground truth injection center (point of maximum amplitude) and δt is an arbitrary interval chosen to evaluate whether or not we count the output as a detection. For this work we use multiple values for δt and compare the network's performance between them. It is important to state that only one candidate can be counted as a detection. If we have multiple candidates, there will be only one detection and the rest of the candidates will be considered as false positives. The smoothing is essential for this test because it lets us have one large peak when there is a small interruption in an interval of ones instead of two smaller peaks. For example, if we only applied the threshold to the output we would often get intervals like this: $[0, ...0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, ..., 0]$ (where the dots are completely filled with zeros). If the center of the injection lies very close to where the central zero is, we would register one detection and one false positive for this particular time series which would negatively affect the performance metrics even though the output accurately predicts where the injection is without really causing reason to think there is a false positive.

This is a much better performance metric because it actually tells us if the network is being able to recognize GW signals mixed with experimental (or simulated) noise. Nevertheless, this metric alone cannot be enough to have a high degree of confidence in the network. The reason is that the algorithm could very well output an interval of ones in the desired location of the time series, but it could also produce more of these intervals in many other locations rendering the correct one meaningless. That is why we also have to include the some information about false positives, which in this case will be the inverse false positive rate.

8.2 Inverse False Positive Rate (IFPR)

False positives are all the intervals of 1 s that are not regarded as a detection after smoothing and thresholding the time series. As mentioned in the previous section, the smoothing minimizes this number by joining two very close peaks but it also removes very small intervals of ones in the middle of large amounts of zeroes. Consequently, intervals like $[0, ..., 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, ..., 0]$ would turn into an array of zeroes only, which works very well for noise samples whose output contains a couple of ones without them being a significantly big interval to be counted as a false positive.

In the figures below, we can see an example of how the output of the algorithm accurately predicts a GW signal with the correct location in an injection sample, and how it returns purely zeros in the noise only sample. The yellow curve belongs to the label and the blue one to the prediction. Notice how, on the strain that contains a GW (figure 8), the very small or thin intervals get turned into zeroes leaving only the peak that clearly stands above all others (figure 9) and, on the noise strain (figure 10), all the peaks are turned to zeroes (figure 11) because they're either too short or too thin to be considered a candidate for detection and consequently become a false positive.

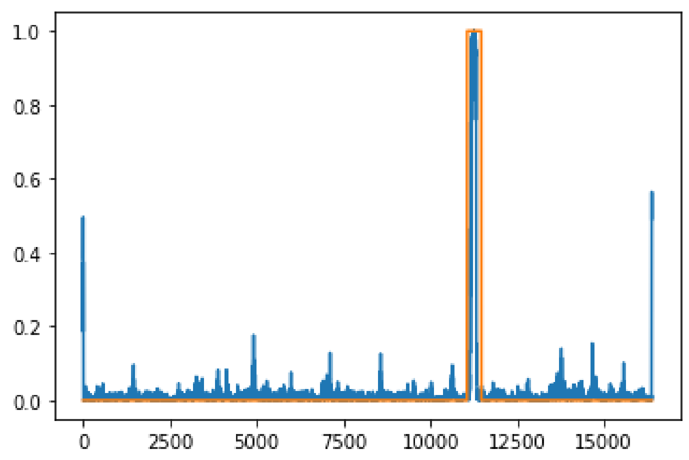

Figure 8: Example of a raw output from the network that results in an accurate detection of the signal.

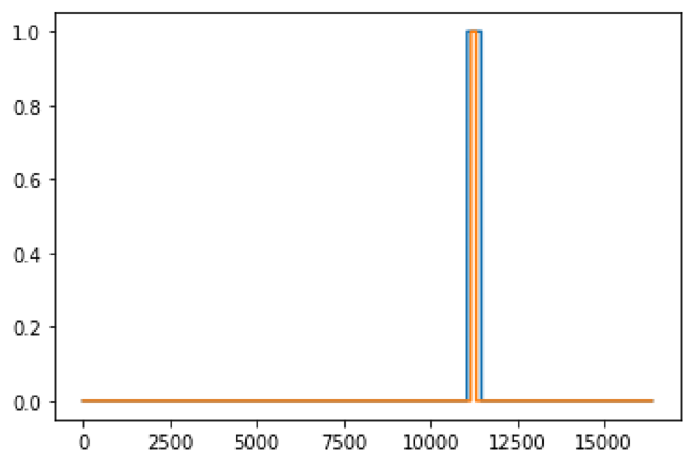

Figure 9: Example of the same previous output after being smoothed and thresholded.

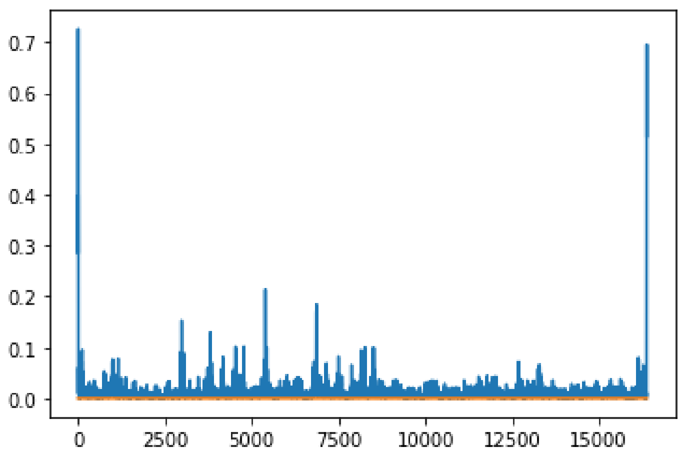

Figure 10: Example of a raw output from the network that accurately predicts that the strain contains only noise.

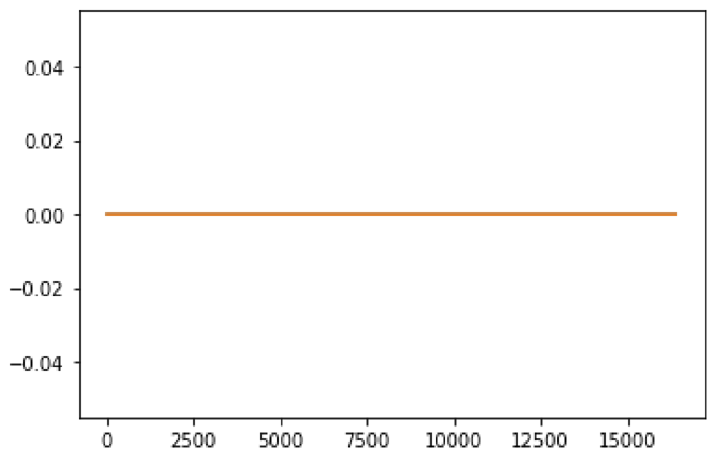

Figure 11: Example of the same noise sample after being smoothed and thresholded. No detetion peak is visualized.

For this metric we first have to calculate the false positive rate. This is the number of false positives that the algorithm produces per unit of measuring time. This means that we count the number of false positives in the whole dataset and divide it by the sampling time that corresponds to the dataset which, in this case, would be 8 s, the time length of each strain, times the number of strains in the dataset. Therefore, the IFPR would be the average time that it takes for the network to produce a false positive, which means that we want this metric to be as big as possible.

There, SNR is more of an auxiliary value for the previously discussed metrics in order to see how sensitive is the network to the GW signals for a given amount of noise.

Like with the δt parameter we use multiples values of SNR and see how effective the algorithm is among all of them [11].

9 Results

9.1 U-Net

During our tests we trained multiple networks and evaluated their performance. We started with one that had only 64 channels in its heaviest layer (bottom one) and only had one input channel, which means that only received information from the H1 strains instead of both H1 and L1. This first one proved unsuccessful so it is not included in the results. After, we expanded the network to have 256 channels in the bottom layer but still with only one input channel and finally we adjusted to get that second input channel. For each one we calculated the DR over SNR as explained in previous sections. The network seemed to never experience overfitting, only a plateau in the increasing accuracy and decreasing validation loss. It is important to state that the accuracy was measured only in the validation set. Therefore, we can say that the network never seems to loose performance on the validation set, it just stops increasing.

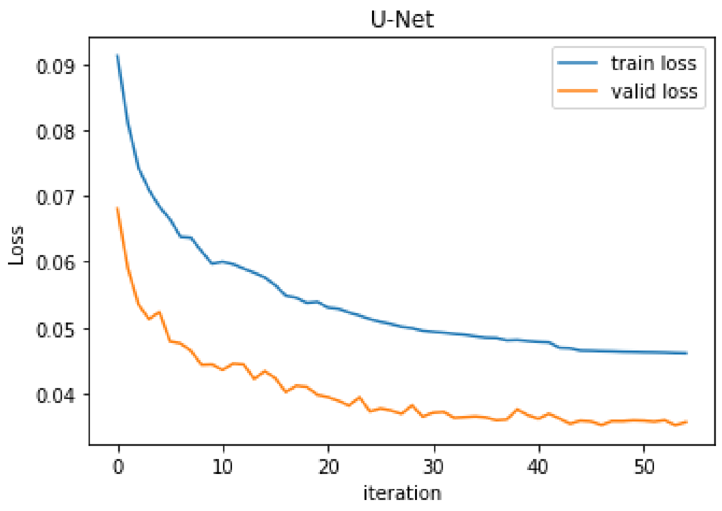

Figure 12: Average training and validation loss for each strain measured on each validation cycle of the U-net model with two input channels and 256 bottom channels.

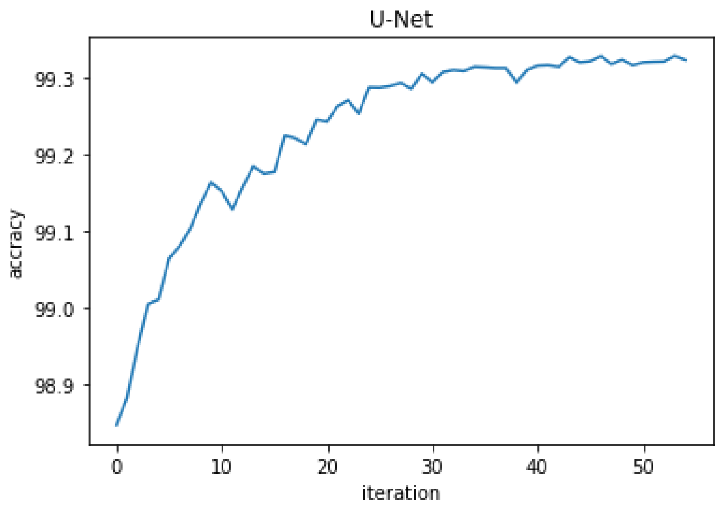

Figure 13: Accuracy measured each validation cycle of the U-net model with two input channels and 256 bottom channels.

As you may have noticed in figures 12 and 13, the validation loss never surpasses the training loss, indicating no overfitting. These behavior was shared by the other versions of the U-Net.

The results from the two versions of the U-Net in the form of DR over SNR plots are presented in figures, 14 and 15.

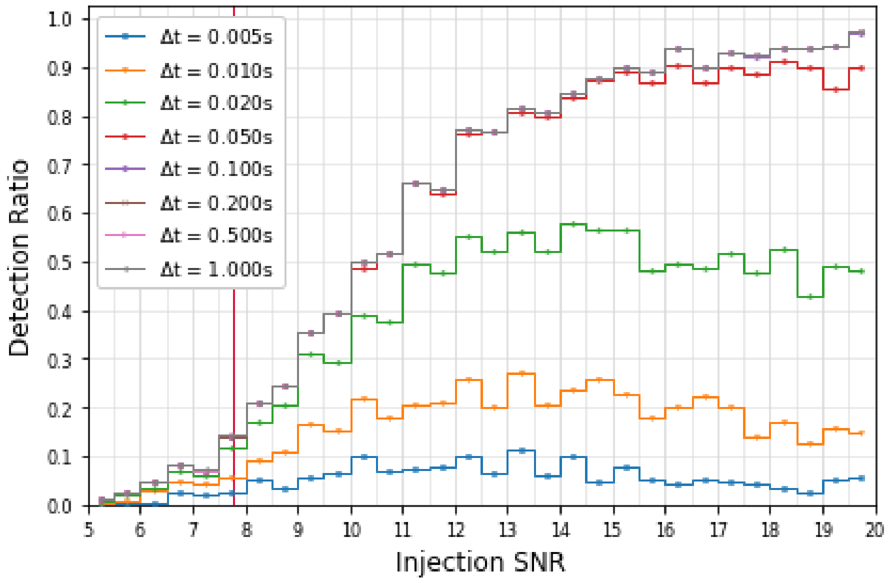

Figure 14: DR over SNR for the the network of 1 input channel and 256 bottom channels.

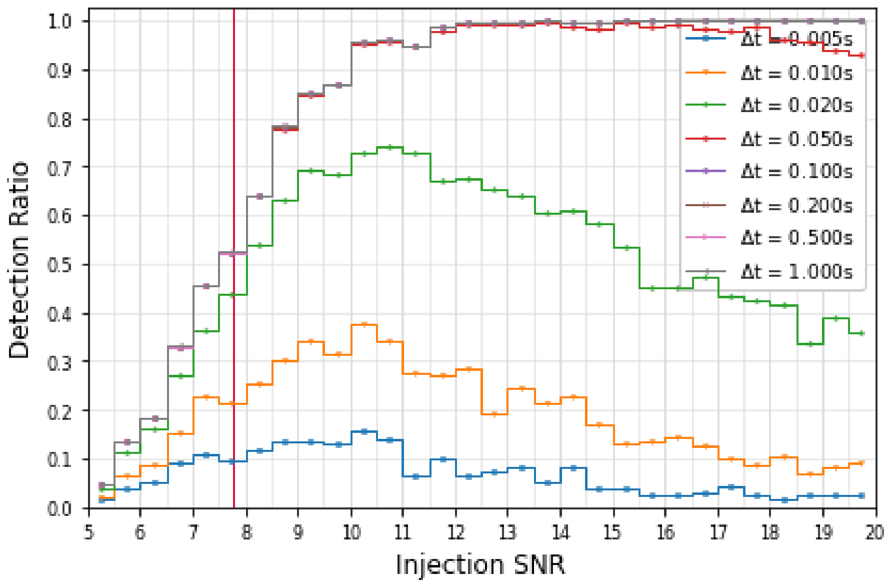

Figure 15: DR over SNR for the the network of 2 input channel and 256 bottom channels.

The Δt value corresponds to the interval around the injection center where the prediction peak center can be found to be counted as a detection. For example, for a value of $\Delta t = 0.5s$ only those predictions that have a peak of ones whose center is no farther from the injection center than 0.5 s will be counted as detections. The distance in seconds is translated to an index in the time series simply by multiplying it by the sampling rate of 2048 Hz.

For the first U-Net (1 input and 256 bottom channels) the three biggest values of Δt manage to reach this 90%. The main difference of the second U-Net (the only one with 2 input channels) lies on the three largest values of Δt. In figure 15 we see how just after a 10 SNR, these three surpass the DR of 90%. Still, we can see other significant differences in the other values of Δt around SNRs of $9 - 14$ where the second U-Net has a larger DR than the second one although, it drops below this one on the largest values of the SNR.

We also computed the DR over the IFPR for each of these networks whose results are shown in figures 16 and 17 where the IFPR metric shows a high degree of improvement when increasing the number of channels.

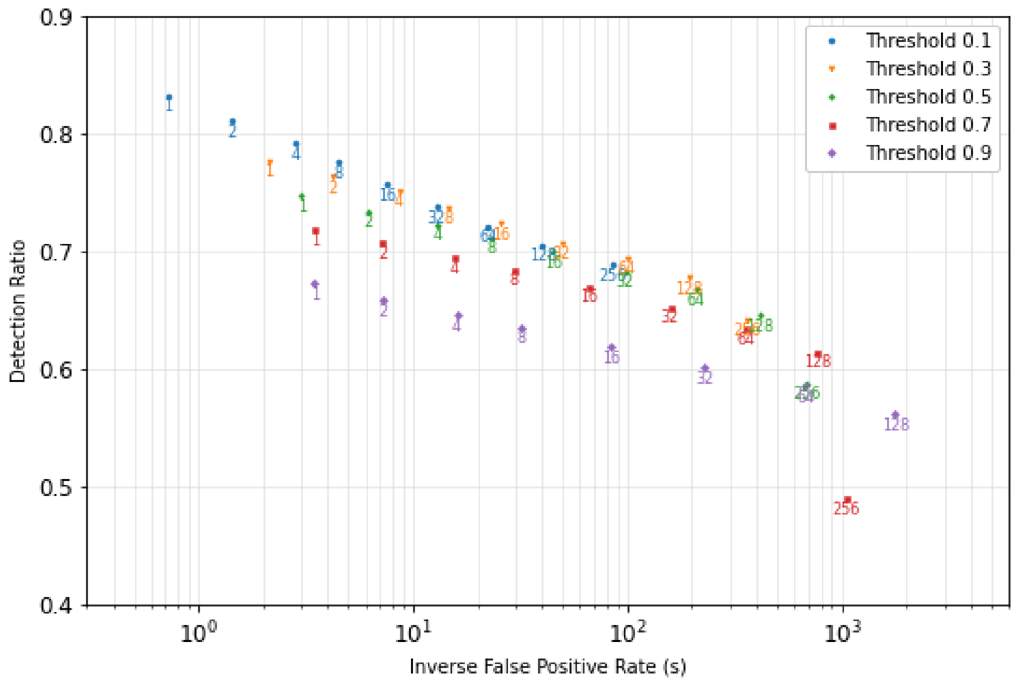

Figure 16: DR over IFPR for the network of 1 input channels and 256 bottom channels.

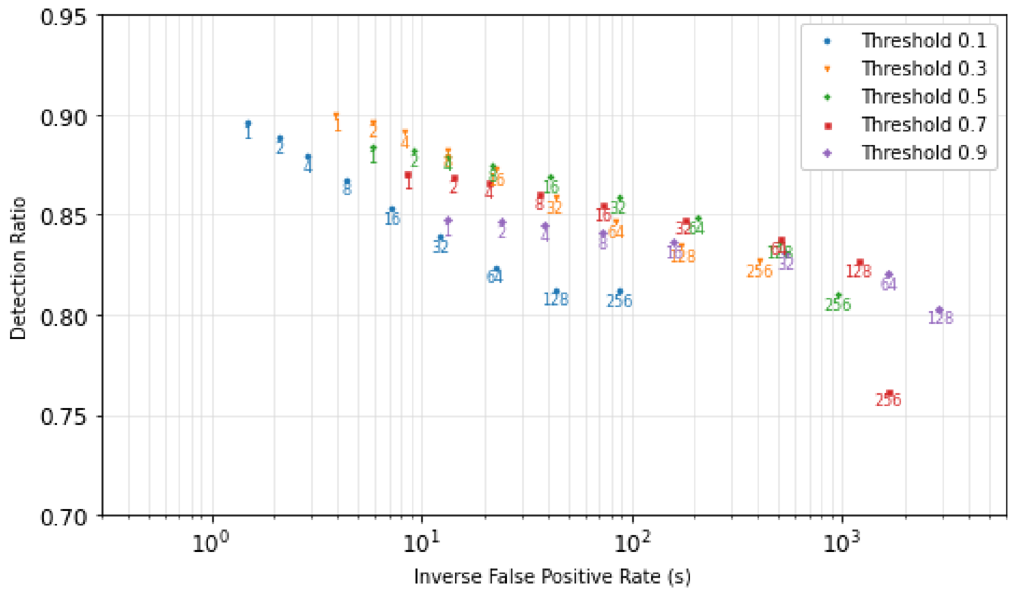

Figure 17: DR over IFPR for the network of 2 input channels and 256 bottom channels.

Not only we see that the detection ratio, in general, goes up throughout all the

IFPR and threshold values, but we also see that the slope of the data flattens as the channels increase. Which means that the algorithm can learn to predict less and less false positives while keeping a good detection ratio.

The threshold values correspond to the one we discussed in section 5 to turn the algorithm's raw output into a one or a zero. So, naturally, the larger the threshold is, we'll have less false positives but also less detections. Correspondingly, a smaller threshold will mean more detections but also more false positives. On the other hand, the numbers below each data point belong to the window size for the smoothing operation.

Therefore, having this plot helps us to fine-tune these parameters in order to have the most accurate network depending on the specific context.

9.2 TCN

The TCN was trained with 128 and 512 channels in the hidden layers. We can see a significant degree of improvement with the substantial increase in channels as expected. The difference is mainly seen for the smaller values for Δt since for the biggest three there is almost no change as you can appreciate from figures 18 and 19.

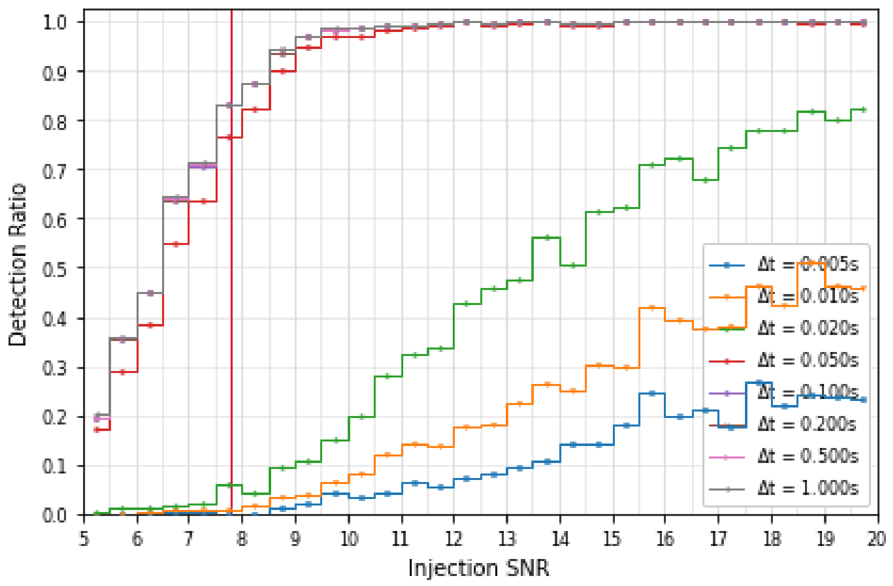

Figure 18: DR over SNR for the TCN network with 128 channels per layer.

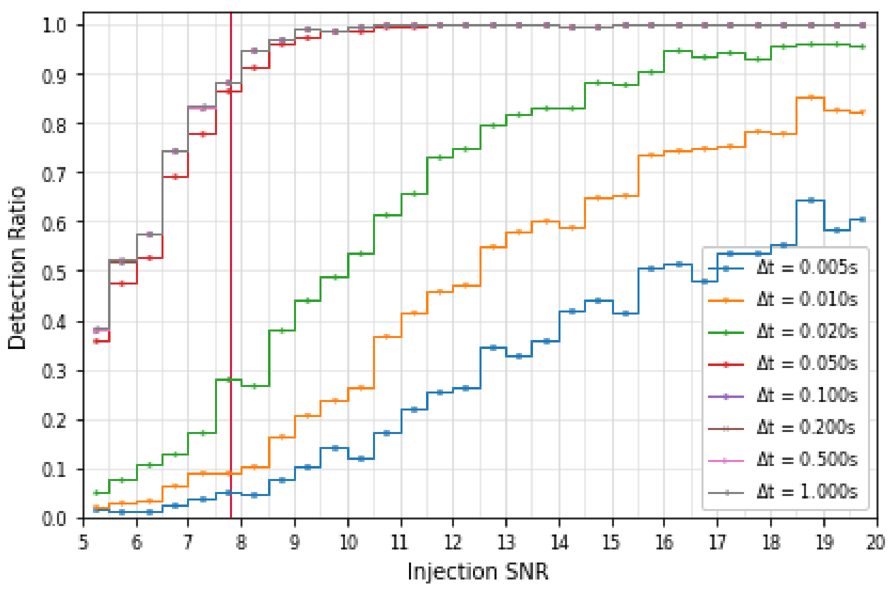

Figure 19: DR over SNR for the TCN network with 512 channels per layer.

Although for the three largest values of Δt the DR crosses the 90% value, it is interesting to look at $\Delta t = 0.020s$. For this value, the DR never crosses 90% on the first TCN while on the second one, it does after a SNR value of 16. It is also remarkable for this Δt that on the first TCN, around an SNR of 13 it doesn't even reaches a DR of 50% while on the second one it already goes over 80%. Very significant improvements were made on the values of $\Delta t = 0.010, 0.005$ but unfortunately not reaching the 90% DR even at the largest SNR considered.

As for the DR over the IFRP plots there is a substantial increase between the small TCN, represented by figure 20, and the big one, seen in figure 21, in both metrics simultaneously. As it can be seen in those figures, in general, the data points present higher values for the two metrics. There is also a remarkable improvement for the smallest threshold value 0.1. The points corresponding to this threshold seem to be higher and far more to the right in the plot of the 512 channel TCN with respect to the 128 channel one. There are some data points, like the yellow 4 and the blue 16 that have a higher detection rate in the smaller TCN but a very high cost in false positives. These points, for the smaller TCN lie extremely close to the zero in the IFPR axis, indicating a very high and/or frequent production of false positives making the detections loose a big portion of their value.

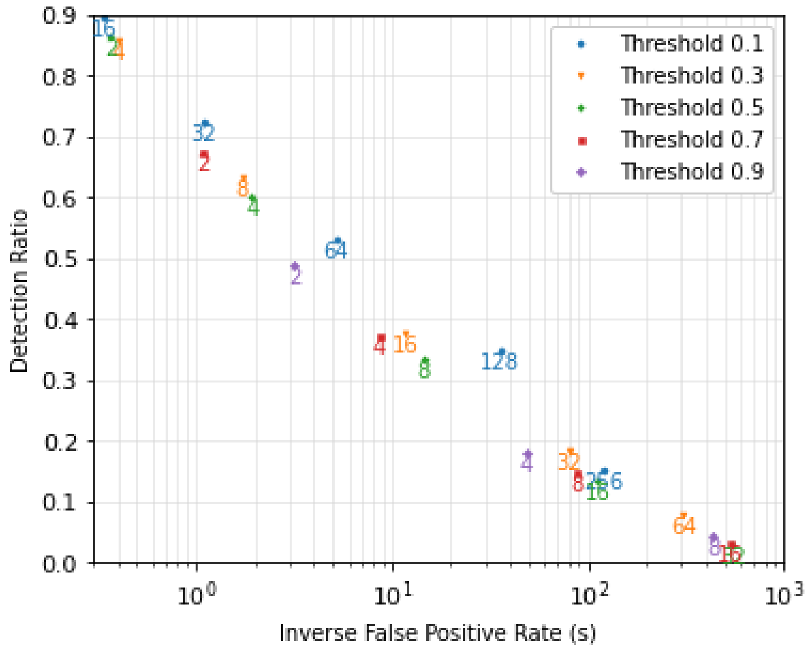

Figure 20: DR over IFPR for the TCN network with 128 channels per layer.

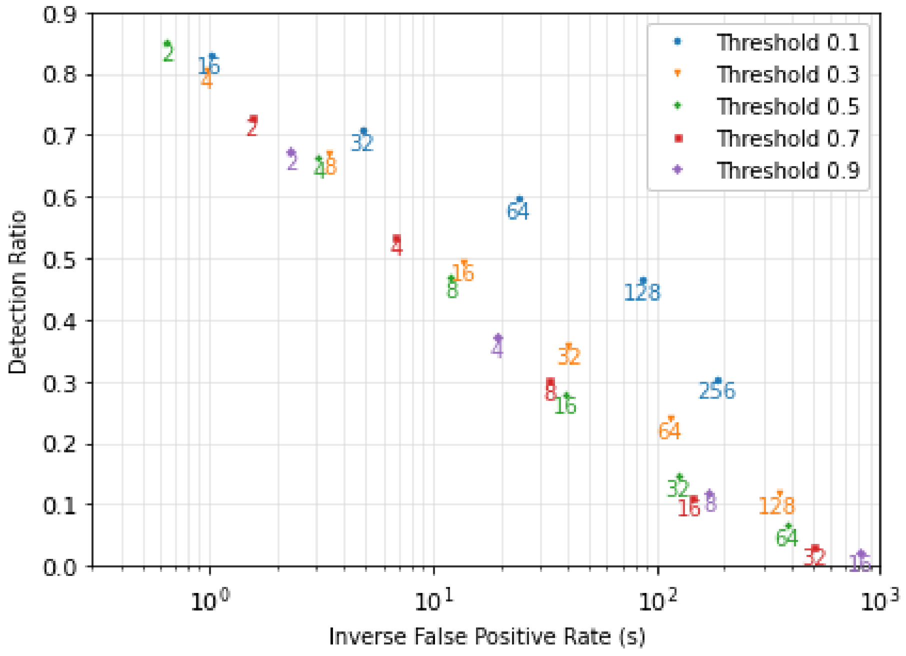

Figure 21: DR over IFPR for the TCN network with 512 channels per layer.

Finally we show in table 1 below the amount of learnable parameters (LP) of each network with different channels (Ch).

Network	U-Net (256, 1 Ch)	U-Net (256, 2 Ch)	TCN (128 Ch)	TCN (512 Ch)
LP	1,804,929	1,805,569	790,017	12,597,249

Table 1: Number of learnable parameters of our networks. The two channel numbers in the parenthesis of the U-Net examples correspond to the number of bottom and input channels respectively.

10 Limitations

As said before, these results were produced by running our codes in the google colab. In this environment we had access to 2.2 GHz CPUs with 25.46 GB of RAM and only one GPU that can be an Nvidia K80 or a T4 with with a very similar RAM storage. This means that we could only run one code on GPU at a time while the others run a lot slower and thereby limiting the amount of tests we could perform and the amount of channels we could use for each network due to time constraints. Also, colab would only allow a certain idle time before disconnecting the session from the servers making us loose some progress on some occasions.

11 Conclusions

In this work we have explored, to the best of our knowledge, the application of two CNN architectures very new to the field of GW detection, the U-Net and the TCN. Even though, the U-Nets we have tried do not match yet the performance of similar algorithms present in the literature, we have to consider that this type of architecture is significantly lighter in terms of learnable parameters compared to other 1D CNNs which do segmentation as well (ie. see for instance Ref. [11]). Therefore they are more easy to train. Given the increase in performance with little increases in channels, it wouldn't be unreasonable to expect the U-Net to tie the other networks' performance while still running significantly faster. These other networks usually have 512 kernels per layer with more than 12 layers. Our's has 4 layers of 64, 128 and 256 kernels each. Another worthy remark is that our U-Net only has three levels unlike other models with up to five levels [14].

Regarding the TCN, the biggest one clearly presents a better performance than the biggest U-Net but at a much higher computational cost that is shown in table 1. As you may have seen on the table, the heaviest TCN has over twelve million parameters that take a very long time to train, even with a GPU it requires very high amounts of RAM even for small values of batch size, given that the data that is fed to the network is also considerably heavy. Although, the small TCN also outperforms the heaviest U-Net on the smaller values of δt with bigger SNRs even though the U-Net gets the better performance (comparing it to the small TCN) for smaller SNRs. Around SNR values of 10, the bigger U-Net clearly gets a better DR than the first TCN, but given the sudden change in slope to a negative value, the U-Net starts to get outperformed approximately after a SNR of 13. Still, we have to remember that this U-Net is still small in levels and kernels per layer relative to the other models cited in this work.

You can see how there is still plenty room for future experiments and improvements in this specific kind of architecture of CNNs, either by increasing the channels or the levels which would increase the number of layers and concatenations. With any of these options we increase the number of learnable parameters of the network which is likely to increase its performance. Additionally, given that in the future we will have more observatories from where we can draw data, we could increase our input channels to incorporate this observatories, potentially increasing performance with a minimum cost of just one input channel as we saw in the results of the U-Net comparing one vs. two input channels. There is also the probability that some kind of *curriculum learning* [32] could be used to better the sensitivity of the networks in the SNRs where they don't perform well. *Curriculum learning* is when one starts by feeding the algorithm "easy" data and then gradually increases the difficulty. This could be done by progressively exposing the network to more and more data with the SNRs where they underperform.

For the future, we could try to train our networks with data from O2 and O3 observations to see if they are able to "detect" real events. Additionally we could attach our networks to a simple 1-D classifier to extend our work to account for Binary Neutron Stars Systems as well. The classifier would determine the type of the source while the networks would detect the signal.

12 Acknowledgments

I would like to thank my advisors for guiding me throughout this work. They were always willing to assist me and point me in the right direction whenever I needed it. Certainly, this work as it is, could not have been achieved without their help and their knowledge to see through the nuance and the complexity of the topics that were dealt with. Roberto Ruiz guided me through the whole area of CNNs, which was completely new for me when we started this work. Without his remarkable help with the production of data for the network or the structure of the TCN, the process for obtaining the results would've had a lot more obstacles. He also motivated me in between different stages of this project, making me understand the importance of it. Alejandro Torres made sure I wasn't lagging behind with the theoretical and experimental side in the field of gravitational wave astronomy. He explained to me some of the subtleties of this topic and called me out whenever I made any unjustified assumptions, afterwards providing

the proper scientific rigor required.

References

[1] 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.

[2] Junaid Aasi, BP Abbott, Richard Abbott, Thomas Abbott, MR Abernathy, Kendall Ackley, Carl Adams, Thomas Adams, Paolo Addesso, RX Adhikari, et al. Advanced ligo. Classical and quantum gravity, 32(7):074001, 2015.

[3] Fet al Acernese, M Agathos, K Agatsuma, D Aisa, N Allemandou, A Allocca, J Amarni, P Astone, G Balestri, G Ballardin, et al. Advanced virgo: a second-generation interferometric gravitational wave detector. Classical and Quantum Gravity, 32(2):024001, 2014.

[4] R Abbott, TD Abbott, F Acernese, K Ackley, C Adams, N Adhikari, RX Adhikari, VB Adya, C Affeldt, D Agarwal, et al. Gwtc-2.1: Deep extended catalog of compact binary coalescences observed by ligo and virgo during the first half of the third observing run. arXiv preprint arXiv:2108.01045, 2021.

[5] CS Unnikrishnan. Indigo and ligo-india: Scope and plans for gravitational wave research and precision metrology in india. International Journal of Modern Physics D, 22(01):1341010, 2013.

[6] Yoichi Aso, Yuta Michimura, Kentaro Somiya, Masaki Ando, Osamu Miyakawa, Takanori Sekiguchi, Daisuke Tatsumi, Hiroaki Yamamoto, Kagra Collaboration, et al. Interferometer design of the kagra gravitational wave detector. Physical Review D, 88(4):043007, 2013.

[7] 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.

[8] Ian Harry, Stephen Privitera, Alejandro Bohé, and Alessandra Buonanno. Searching for gravitational waves from compact binaries with precessing spins. Physical Review D, 94(2):024012, 2016.

[9] Benjamin P Abbott, R Abbott, TD Abbott, S Abraham, F Acernese, K Ackley, C Adams, VB Adya, C Affeldt, M Agathos, et al. Prospects for observing and localizing gravitational-wave transients with advanced ligo, advanced virgo and kagra. Living reviews in relativity, 23(1):1–69, 2020.

[10] Heming Xia, Lijing Shao, Junjie Zhao, and Zhoujian Cao. Improved deep learning techniques in gravitational-wave data analysis. Physical Review D, 103(2):024040, 2021.

[11] Timothy D Gebhard, Niki Kilbertus, Ian Harry, and Bernhard Schölkopf. Convolutional neural networks: A magic bullet for gravitational-wave detection? Physical Review D, 100(6):063015, 2019.

[12] Alvin JK Chua, Chad R Galley, and Michele Vallisneri. Reduced-order modeling with artificial neurons for gravitational-wave inference. Physical review letters, 122(21):211101, 2019.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org.

[14] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention, pages 234–241. Springer, 2015.

[15] Adam Hilbert, Vince I Madai, Ela M Akay, Orhun U Aydin, Jonas Behland, Jan Sobesky, Ivana Galinovic, Ahmed A Khalil, Abdel A Taha, Jens Wuerfel, et al. Brave-net: fully automated arterial brain vessel segmentation in patients with cerebrovascular disease. Frontiers in artificial intelligence, page 78, 2020.

[16] Weiqiang Zhu, S Mostafa Mousavi, and Gregory C Beroza. Seismic signal denoising and decomposition using deep neural networks. IEEE Transactions on Geoscience and Remote Sensing, 57(11):9476–9488, 2019.

[17] Mathias Perslev, Michael Jensen, Sune Darkner, Poul Jørgen Jennum, and Christian Igel. U-time: A fully convolutional network for time series segmentation applied to sleep staging. Advances in Neural Information Processing Systems, 32, 2019.

[18] R. D'Inverno. Introducing Einstein's Relativity. Clarendon Press, 1992.

[19] Janyce Franc, Nazario Morgado, Raffaele Flaminio, Ronny Nawrodt, Iain Martin, Liam Cunningham, Alan Cumming, Sheila Rowan, and James Hough. Mirror thermal noise in laser interferometer gravitational wave detectors operating at room and cryogenic temperature. arXiv preprint arXiv:0912.0107, 2009.

[20] Matthew Pitkin, Stuart Reid, Sheila Rowan, and Jim Hough. Gravitational wave detection by interferometry (ground and space). Living Reviews in Relativity, 14(1):1–75, 2011.

[21] Benjamin P Abbott, Rich Abbott, Thomas D Abbott, Sheelu Abraham, Fausto Acernese, Kendall Ackley, Carl Adams, Vaishali B Adya, Christoph Affeldt, Michalis Agathos, et al. A guide to ligo–virgo detector noise and extraction of transient gravitational-wave signals. Classical and Quantum Gravity, 37(5):055002, 2020.

[22] Benjamin P Abbott, R Abbott, TD Abbott, MR Abernathy, F Acernese, K Ackley, M Adamo, C Adams, T Adams, P Addesso, et al. Characterization of transient noise in advanced ligo relevant to gravitational wave signal gw150914. Classical and Quantum Gravity, 33(13):134001, 2016.

[23] Introduction to ligo gravitational waves. https://www.ligo.org/science.php.

[24] https://github.com/timothygebhard/ggwd.

[25] Alejandro Bohé, Lijing Shao, Andrea Taracchini, Alessandra Buonanno, Stanislav Babak, Ian W Harry, Ian Hinder, Serguei Ossokine, Michael Pürrer, Vivien Raymond, et al. Improved effective-one-body model of spinning, nonprecessing binary black holes for the era of gravitational-wave astrophysics with advanced detectors. Physical Review D, 95(4):044028, 2017.

[26] Akhilesh A Waoo and Brijesh K Soni. Performance analysis of sigmoid and relu activation functions in deep neural network. In Intelligent Systems, pages 39–52. Springer, 2021.

[27] Facundo Bre, Juan M Gimenez, and Víctor D Fachinotti. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. Energy and Buildings, 158:1429–1441, 2018.

[28] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. nature, 323(6088):533–536, 1986.

[29] Stamatis Mastromichalakis. Alrelu: A different approach on leaky relu activation function to improve neural networks performance. arXiv preprint arXiv:2012.07564, 2020.

[30] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv preprint arXiv:1803.01271, 2018.

[31] GWOSC. Technical details for advanced ligo and virgo event data releases.

[32] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In Proceedings of the 26th annual international conference on machine learning, pages 41–48, 2009.