Detection of Gravitational Wave signals using Machine Learning methods and Generative Pre-trained Transformers

Author: Abel Dana Ruiz
*Facultat de Física, Universitat de Barcelona, Diagonal 645, 08028 Barcelona, Spain.**

Advisor: Tomás Andrade Weber and Co supervisor: Roberto Emparan García de Salazar

Abstract: We use Machine Learning methods based on Convolutional Neural Networks to search for gravitational waves signals above the background noise distribution for a data set of simulated gravitational waves and real noise signals from three detectors (LIGO Hanford, LIGO Livingston, and Virgo). A training data set is used to train the ML method to classify data streams in two groups: gravitational wave plus noise (label 1) or only noise (label 0). Later, the method predicts if data streams from a testing data set belong to one or an other category. To generate the code that implements the CNN algorithm we use Generative Pre-trained Transformers, specifically ChatGPT based on GPT-3 and compare them to a human-made CNN. The ML methods are capable to detect gravitational waves if we give ChatGPT freedom to create a CNN without specifying the parameters or the architecture, but are not satisfactory if we try to direct ChatGPT to a specific type of code.

I. INTRODUCTION

The finding of gravitational waves in 2015 by both LIGO detectors [1] was an inflection point in the General Relativity field, confirming one of the conjectures Einstein had based on his theory. The gravitational waves that LIGO detected came from the merging of two black holes belonging to a binary system. Nevertheless, the amount of data that needed to be processed was really big, and the problem of how to handle this sum of data opened the door to a whole range of different data processing techniques.

Two of the most popular data processing methods are Matched Filtering and Machine Learning (ML) methods. The first (see [2]) are based in the application of a linear filter to the data, and turn out to be simple and optimal methods, but they have problems if the noise distribution is non-Gaussian or time evolving, making them slow methods and very data demanding to perform properly. The second ones are not as limited, given that the type of architecture chosen and the amount of data used can make them really strong and reliable methods. Because of that, there has been a growing interest in ML type methods.

The objective of this *Treball Fi de Grau* (TFG) is to use ML techniques to detect gravitational waves from the background noise. The core of the ML method will be a Convolutional Neural Network (CNN), that will be generated using Large-scale Language Model (LLM). Firstly, we will do a short explanation of how CNN work; secondly, we will discuss all the technical aspects of the method, such as the input data, the code and where the LLM will be used; finally, we will compare different LLM-generated CNN's with a human-made one, to see if an open source program can yield better results.

*Electronic address: adanarui7@alumnes.ub.edu

II. CONVOLUTIONAL NEURAL NETWORKS

Essentially, the ML method we will use is a binary classifier: for a data stream, it must return a number that express if it has or not a gravitational wave. To make the method learn we use a huge amount of labeled data (where, in addition to the noise distribution and maybe the gravitational wave above, there is a label determining if there is or there is not injected gravitational wave) and we must use a convenient CNN to make the best prediction.

In our case, the CNN will work with images (see **section III b** for the detail), that can be treated mathematically as an order 3 tensor of size $H \times W \times 3$, where H are the numer of rows, W the number of columns and 3 represents the R, G, B color representation. It is important to say that for our case it was enough to work with a grey scale representation, so this tensor (image) has a matrix representation. So the input is, in general, an order 3 tensor that will go into a sequential series of processes. We can represent the input tensor as $\mathbf{x}^1$, the L different layers of the CNN as $\mathbf{w}^1, \mathbf{w}^2, ..., \mathbf{w}^L$ (where $\mathbf{w}^i$ are the parameters for the ith-layer) and the loss function as z. This loss function gives an idea of the discrepancy between the CNN prediction (that, in this notation, after going through L layers will be labeled $\mathbf{x}^L$) and the real result.

In order to have a good CNN is important to estimate the parameters of each layer, and this is what the CNN does when we say that we make it learn, specifically, it is convenient to minimize the loss function z. The most commond way to do this estimation is with the Stochastic gradient descent, but this is beyond this TFG (see [3] for mathematical details). From now on, we will do a brief explanation of the mathematical bases of the most important layers of CNN: the convolutional layers. However, there are other layers involved in the CNN, but they are out of scope of this report

Taking a general order 3 tensor ($H \times W \times D$ in

the input), for the l-th layer the tensor will have size $H^l \times W^l \times D^l$. We define a convolutional kernel as an order 3 tensor with size $H \times W \times D^l$. On every spatial location, we overlap the convolutional kernel above the input l-th tensor and calculate the products of corresponding elements in the D^l channels and sum the HWD^l products. If we move the kernel over the full image we perform the complet convolution. In precise mathematics, and if we consider than for the l-th layer we need a set of index (i^l, j^l, d^l) and define $\mathbf{y} = \mathbf{x}^{l+1}$, it is possible to see than the convolution operation, in it's simplest representation, can be defined as:

$$y_{i^{l+1},j^{l+1},d} = \sum_{i=0}^{H} \sum_{j=0}^{W} \sum_{d^l=0}^{D^l} f_{i,j,d^l,d} \times x^l_{i^{l+1}+i,j^{l+1}+j,d^l} \quad (1)$$

Where $\mathbf{f}$ are the set of convolutional kernels, and this operation is the one that must be done for every spacial location and for all d's.

Convolutional layers are important because they are the ones capable to recognise patterns if they are correctly applied. So, combined with other layers can recognise some characteristics of the imput image during the learning process and be able to use this patterns previously learned (this is, the parameters $\mathbf{w}^1, \mathbf{w}^2, ..., \mathbf{w}^L$ of the CNN are determined during the training process) to determine the nature of new images.

Having seen all this, the procedure performed during the whole TFG has been considering the CNN as a black box plenty of parameters (now we are refering to the parameters we use during the programming, not the $\mathbf{w}^i$ parameters of the CNN) that can be modified in order to get a correct prediction. To determine this parameters and layer organisation is where we will be using a LLM.

III. MACHINE LEARNING METHOD USING CNN'S

A. Input data and evaluation

During the whole development we have used a data set from the Kaggle competition "G2Net Gravitational Wave Detection" (see [4]). This competition was created to use ML methods for gravitational waves detection. All data streams provided consist in an audio-type file, separated into three time series, each one corresponding to one gravitational waves interferometer (LIGO Hanford, LIGO Livingston, and Virgo). For each time series, there is real noise corresponding to the respective interferometer, and in some of them there is a simulated gravitational wave injected above the noise. The lenght of each time series is $2s$ and the sampling frequency is $2.048Hz$. An example of this time series can be seen in Figure 1, where in the first graphic there is a gravitational wave and in the second there is only noise.

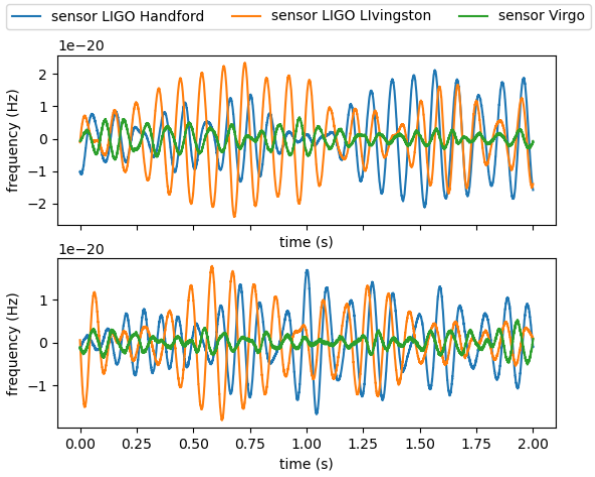

FIG. 1: Gravitational waves signal detected (upper) and noise signal (lower) signal detected by three interferometers.

The data is divided into two sets: the training set and the testing set. The first one consists in data streams, with a label 0 if there is not injected gravitational wave or label 1 if there is injected gravitational wave. The second set consists only in data streams, and are the ones that the CNN must predict.

This ML method returns the name of all the data streams from the testing set in the first column and a real number between 0 and 1 for each data streams in the second. If this number is lower than 0.5, the prediction is no gravitational wave, and if it is greater than 0.5 there is supposed to be a (simulated) gravitational wave injected in the correspondent file.

B. Code construction

The code used is dividided in three parts:

1. Data pre-processing
2. Convolutional Neural Network
3. Use of the CNN to do the predictions

In the first part of the code we use some techniques to pre-process data. All train and test data sets are $77GB$, and we are running them directly on a laptop, so it is necessary to optimize the process.

Firstly, instead of working with the raw audio-type files, one can do a transformation of these ones, and work with a 3D spectrogram [5], with image format. An example of this image format can be seen in Figure 2 , where we have represented the same data as in Figure 1, that is, in the first case there is a gravitational wave and in the second one there is only noise.

Once all data is in image format, one could introduce it to the CNN, but firstly is advisable to do something to optimize the process. In our case, instead of working

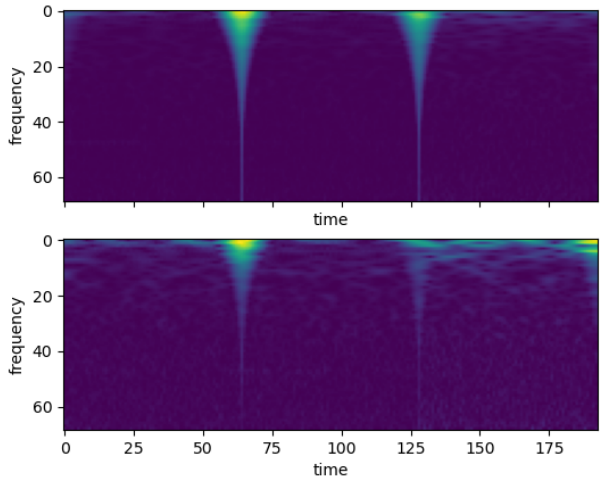

FIG. 2: Spectrogram corresponding to the gravitational waves signal (upper) and to noise signal (lower) detected by three interferometers.

with the raw image-type data set, we divide it into non dependent closed packages of data. We do this with all the data set (train and test). This process makes the compiling time, the training time and the testing time much smaller.

The second part of the code is the core of the ML method. In this part we use the Keras library of Python, which allows us to construct a Neural Network using pre-built layers (such as convolution layers, some neural layers and other layers with specific functions). It is important to say that in this TFG we have only used sequential type architectures (all layers are one after an other), no parallel or more complicated organizations. After defining the CNN, we train it using the testing data set. We also use a validation data set, that serves to evaluate the losses and any statistics during the training process (it is important to say that the validation data set is not used to train).

Finally, the last part of the code is dedicated to see the performance the CNN with the testing data set and returning the results file with the predictions of the ML method.

C. Use of LLM during the code

In the second part of the code (the part where the CNN is implemented) we used a LLM to generate the sequentially organised layers that should be the core of the ML method. The software used for this is ChatGPT, a LLM created by openAI based on the autoregressive language model Generative Pre-trained Transformer 3 (GPT-3).

As we brievly said in **section II**, in this TFG we considered CNN's as a black box plenty of parameters and with a performance very sensible to modifications in them. Because of this, the usage of LLM in this part of the code could give a new perspective to this work: seeing if a LLM was capable to generate a code that implements CNN better than the one that a person with knowledge of the subject was able to create.

IV. RESULTS

A. Performance evaluation

To evaluate the results of each CNN we used a tool given by the Kaggle competition [4], where is possible to submit the final file of results, and it assigns a number between 0.5 and 1 using the area under the Receiving Operating Characteristic (ROC) curve between the predicted probability and the observed target.This type of methods are important in binary classification models.

Defining the true positive rate (TPR) as the rate between positive predictions (in our case predictions greather than 0.5, there should be an injected gravitational wave) and the true positive results, and defining the false positive rate (FPR) as the rate between positive predictions and the true negative results (lower than 0.5, there is only noise), one can calculate the ROC curve in a graphic between the FPR and the TPR. The area under this curve will give a number that can give a mark of the method (see [6]). If this number is from 0.5 to 0.75 the model will not be satisfactory, and if the model is above 0.75 it can be considered a good method (obviously, higher the number, better the method).

B. Pseudo-code LLM generated CNN

Firstly, using a reference CNN from a Kaggle competitor (this competitor or the organisation erased the web page where his code was, so it is impossible to properly quote him), we tried to obtain a similar architecture with similar parameters. This procedure can be seen in Appendix 1, and it wasn't successful: chatGPT [7] wasn't able to replicate a consistent CNN and the results weren't as expected. Basically, the number returned by the ML method was always the same (for all data streams in the testing data set), so the correct predictions where due to a coincidence (for example, if the number was always above 0.5 and casually a file didn't have a Gravitational Wave, the prediction was correct, but not by merit of the ML method).

C. Free LLM generated CNN

Instead of trying to replicate an existing code (as we did in the previous section), in this section we didn't specify all the layers and parameters that the code that implements the CNN should have, we only told it how the input data was, that we wanted a classifier and that the architecture structure should be sequential [8].

After testing the code output, we tried to improve the code asking chatGPT to apply some modifications to the previous code (this modifications were suggested by chatGPT itself).

Finally, chatGPT proposed to use other architecures, such as "Inception" architecture. The CNN generated had a training time of, approximately, 65 hours, so it wasn't viable to use. After asking chatGPT to do a cropped version of the "Inception" code, it returned model_inception.

All this CNN's and the interaction with chatGPT can be seen in Appendix 2.

D. Final results

Using the names for the diferent CNN of Appendix 1 and Appendix 2 and also the evaluation form seen previous in this section, we can write a summary table with the most notable CNN's found.

TABLE I: Training time and accuracy of the most significant CNN's tested.

Model name	Training time (hours)	Mark
model_base	4.60	0.85641
model_pseudo	4.98	0.55892
model_free	1.08	0.84642
model_free+_1	1.26	0.76585
model_free+_2	1.03	0.84433
model_inception	1.04	0.84126

As it can be seen in Table I, the freely generated CNN has almost as good results as the human made one (model_base), but the trainig time is considerably lower. In fact, the first free model that chatGPT produced (model_free) was the one with best performance, and all the other CNN's had worse results, even thougt all of them where supposed to be a better version of model_free.

This results suggest that if chatGPT has less restrictions, it obtains better results. The more restrictions it has, the worse the result is. For model_pseudo, the most restrictive one, it was unable to generate usable CNN's.

V. CONCLUSIONS

In this TFG we have seen that Machine Learning methods based on Convolutional Neural Networks can be a good way to detect gravitational waves, they can provide strong and computationally undemanding models.

Also we have used a Large-scale Language Model to generate Convolutional Neural Networks, with really good results, and we have seen that this specific LLM (ChatGPT based on GPT-3) provides better results, in this particular area, if the demands that we ask are less restrictive. In particular, trying to lead the AI to a particular type of code did not give satisfactory results, since the code that the AI generated didn't predict if there was a gravitational wave signal above the noise. On the other hand, for the (almost) freely generated code the results were almost as good as the ones generated by a human made CNN, but the compling time was up to four times faster, making it even more computationally undemanding.

Acknowledgments

I would like to thank my tutor Tomás Andrade for his invaluable help, my co supervisor Roberto Emparan for finding time to help us with his advice and to all the participants of the Numerical Relativity group.

I would also like to thank my parents, my brother, Marina and all my family for their support.

[1] B.P. Abbott et al. (LIGO Scientific Collaboration and Virgo Collaboration), *Observation of Gravitational Waves from a Binary Black Hole Merger* Phys. Rev. Lett. **116**, 061102 (2016)

[2] Jingkai Yan, Mariam Avagyan, Robert E. Colgan, Doğa Veske, Imre Bartos, John Wright, Zsuzsa Márka, and Szabolcs Márka, *Generalized approach to matched filtering using neural networks* Phys. Rev. D **105**, 043006 (2022)

[3] Jianxin Wu (National Key Lab for Novel Software Technology), *Introduction to Convolutional Neural Networks*, Nanjing University, China, 2017.

[4] European Gravitational Observatory, 2021, *G2Net Gravitational Wave Detection*, Kaggle, <https://kaggle.com/competitions/g2net-gravitational-wave-detection>.

[5] nnAudio, 2019, *nnAudio.Spectrogram.CQT1992v2*, <https://kinwaicheuk.github.io/nnAudio/v0.1.5/_autosummary/nnAudio.Spectrogram.CQT1992v2.html>. (Accessed May 7, 2023).

[6] MathWorks, n.d., *ROC Curve*, <https://www.mathworks.com/discovery/roc-curve.html>. (Accessed May 15, 2023).

[7] OpenAI, *ChatGPT-3: Large-scale Language Model.* <https://openai.com >. (Accessed May 1, 2023).

[8] OpenAI, *ChatGPT-3: Large-scale Language Model.* <https://openai.com >. (Accessed May 7, 2023).

Appendix

In this appendix one can find all the CNN's codes used. I've avoided to write comments in the code because they hindered the code presentation. Also, every code line begins with a " >" to differentiate them to new lines that are continuation of the previous code line.

A. Appendix 1

model_base. This model was made by a Kaggle user, and it was the reference code we took. Actually the web page is not available, so I can't put it in the references, but the help of that web site was huge. The code is:

```
>import keras.layers as L

>entrada = L.InputLayer(input_shape=(69, 193,
1))

>conv = L.Conv2D(3, 3, activation='relu',
padding='same')
>efn = efn.EfficientNetB0(include_top=False,
input_shape=(), weights='imagenet')
>pool = L.GlobalAveragePooling2D()

>densa1 = L.Dense(32, activation='relu')
>densa2 = L.Dense(1, activation='sigmoid')

>model = tf.keras.Sequential([entrada, conv,
efn, pool, densa1, densa2])

>model.summary()

>model.compile(optimizer=
keras.optimizers.Adam(learning_rate=0.001),loss=
'binary_crossentropy', metrics=
[keras.metrics.AUC()])
```

model_pseudo. In this model we wanted to replicate model_base. To do this we asked ChatGPT to write a code of a CNN with:

- input: images on grey scale of 69 x 193 pixels in one channel
- convolutional layer with 3 filters of 3x3 with RELU and padding
- intermediate layer with EfficientNetB0
- global average pooling
- 1 dense layer with 32 neurons, with activation layer relu
- 1 output layer with sigmoid activation function
- sequential model construction
- compilation using Adam optimizer with learning rate 0.001

The resulting code was:

```
>from tensorflow.keras.models import
Sequential
>from tensorflow.keras.layers import Input,
Conv2D, GlobalAveragePooling2D, Dense, Dropout
>from tensorflow.keras.applications import
EfficientNetB0
>from tensorflow.keras.preprocessing.image
import ImageDataGenerator

>input_shape = (69, 193, 1)

>model = Sequential()

>model.add(Input(shape=input_shape))
>model.add(Conv2D(filters=3,
kernel_size=(1,1), activation='relu'))

>efficientnet = EfficientNetB0(weights='imagenet',
include_top=False, input_shape=(69, 193, 3))
>model.add(efficientnet)
>model.add(GlobalAveragePooling2D())
>model.add(Dense(64, activation='relu'))
>model.add(Dropout(0.5))
>model.add(Dense(1, activation='sigmoid'))

>optimizer = tf.keras.optimizers.Adam(lr=0.0001)

>model.compile(optimizer=optimizer,
loss='binary_crossentropy',
metrics=['accuracy'])
```

B. Appendix 2

model_free. This was the first free model that ChatGPT created after we asked it to create a model with:

- input: images on grey scale of 69 x 193 pixels in one channel
- objective: classifier
- sequential model construction

With all the freedom it created:

```
>from tensorflow.keras.models import
Sequential
>from tensorflow.keras.layers import Conv2D,
MaxPooling2D, Flatten, Dense

>model = Sequential()

>model.add(Conv2D(32, (3, 3),
activation='relu', input_shape=(69, 193, 1)))
```

```
>model.add(MaxPooling2D((2, 2)))
>model.add(Conv2D(64, (3, 3),
activation='relu'))
>model.add(MaxPooling2D((2, 2)))
>model.add(Conv2D(128, (3, 3),
activation='relu'))
>model.add(MaxPooling2D((2, 2)))

>model.add(Flatten())

>model.add(Dense(128, activation='relu'))
>model.add(Dense(1, activation='sigmoid'))

>model.compile(optimizer='adam',
loss='binary_crossentropy',
metrics=['accuracy'])

>model.summary()
```

model_free+_1. In this code we wanted to improve the previous code performance. From a long list of suggestions ChatGPT did we asked it to implement (the explanation was generated by ChatGPT):

- Regularization: Add regularization techniques such as dropout or L2 regularization to reduce overfitting. – Dropout randomly sets a fraction of input units to 0 during training, while L2 regularization adds a penalty term to the loss function to encourage smaller weights.
- Deeper Architecture: Experiment with adding more convolutional layers or increasing the number of filters in each layer. This can help the model learn more complex features from the images, but be cautious of adding too many layers, as it may lead to overfitting.
- Batch Normalization: Consider adding batch normalization layers after convolutional layers. Batch normalization can help stabilize and accelerate the training process by normalizing the outputs of previous layers.

Trying to construct a code with this improvements the result was:

```
>from tensorflow.keras.models import
Sequential
>from tensorflow.keras.layers import Conv2D,
MaxPooling2D, Flatten, Dense, Dropout,
BatchNormalization

>model = Sequential()

>model.add(Conv2D(32, (3, 3),
activation='relu', input_shape=(69, 193, 1)))
>model.add(BatchNormalization())
>model.add(MaxPooling2D((2, 2)))

>model.add(Conv2D(64, (3, 3),
activation='relu'))
>model.add(BatchNormalization())
>model.add(MaxPooling2D((2, 2)))

>model.add(Conv2D(128, (3, 3),
activation='relu'))
>model.add(BatchNormalization())
>model.add(MaxPooling2D((2, 2)))

>model.add(Flatten())

>model.add(Dense(128, activation='relu'))
>model.add(Dropout(0.5))
>model.add(Dense(1, activation='sigmoid'))

>model.compile(optimizer='adam',
loss='binary_crossentropy',
metrics=['accuracy'])

>model.summary()
```

model_free+_2. Seeing that the results of the previous model wasn't as good as we expected we only asked to do the changes in architecture (deeper architecture), and the result was:

```
>from tensorflow.keras.models import
Sequential
>from tensorflow.keras.layers import Conv2D,
MaxPooling2D, Flatten, Dense

>model = Sequential()

>model.add(Conv2D(32, (3, 3),
activation='relu', input_shape=(69, 193, 1)))
>model.add(MaxPooling2D((2, 2)))

>model.add(Conv2D(64, (3, 3),
activation='relu'))
>model.add(MaxPooling2D((2, 2)))

>model.add(Conv2D(128, (3, 3),
activation='relu'))
>model.add(MaxPooling2D((2, 2)))

>model.add(Conv2D(256, (3, 3),
activation='relu'))
>model.add(MaxPooling2D((2, 2)))

>model.add(Flatten())

>model.add(Dense(128, activation='relu'))
>model.add(Dense(1, activation='sigmoid'))

>model.compile(optimizer='adam',
loss='binary_crossentropy',
```

```
metrics=['accuracy'])

>model.summary()
```

model_inception. In the explanation of this last code, ChatGPT suggested to use other architecture organisations, and the inception architecture was one of these. Asking to write the code with this architecture the result was too demanding, so I asked for a cropped version:

- input: images on grey scale of 69 x 193 pixels in one channel
- objective: classifier
- sequential model construction
- use Inception architecture
- not to computationally demanding

And the result was:

```
>from tensorflow.keras.models import
Sequential
>from tensorflow.keras.layers import Conv2D,
MaxPooling2D, Flatten, Dense, Dropout

>model = Sequential()

>model.add(Conv2D(32, (3, 3),
activation='relu', input_shape=(69, 193, 1)))
>model.add(MaxPooling2D(pool_size=(2, 2)))

>model.add(Conv2D(64, (3, 3),
activation='relu'))
>model.add(MaxPooling2D(pool_size=(2, 2)))

>model.add(Conv2D(128, (3, 3),
activation='relu'))
>model.add(MaxPooling2D(pool_size=(2, 2)))

>model.add(Flatten())

>model.add(Dense(128, activation='relu'))
>model.add(Dropout(0.5))
>model.add(Dense(1, activation='sigmoid'))

>model.compile(optimizer='adam',
loss='binary_crossentropy',
metrics=['accuracy'])

>model.summary()
```