Gravitational wave search for Long-transient signals

Joe Popp[1] and Marco Serra[2]

[1] *Department of Physics, Saint Joseph's University*
[2] *Istituto Nazionale di Fisica Nucleare*

(Dated: August 18, 2023)

To detect long-transient wave signals produced by newly born magnetars in the time-frequency domain, we use PyTorch to train a convolutional neural network that is able to detect these signals. A simulation is made to project randomly generated signals onto a square matrix of Gaussian noise. This simulated data is then used to train and test a convolutional neural network. Training is repeated with variable model sizes and training loop inputs until the model can be successfully trained using data with a low signal amplitude relative to the background noise. With a Gaussian noise centered at an amplitude of zero and a standard deviation of 0.5, we were able to find a model and a proper set of training loop inputs to get an accuracy of 74.7% at a signal amplitude of one.

I. INTRODUCTION

With the first detections of gravitational waves from compact binary systems occurring in the last decade, a whole new kind of physics has been revealed. So far our detectors have only been sensitive enough to measure mergers of binary black hole and neutron star systems. Many other types of gravitational waves are theorized however, coming from many different sources that we hope to detect in the future [1]. One of these sources that can produce strong long-transient gravitational waves is newly born magnetars. These stars are of particular interest since they are able to produce a distinct, strong signal that can vary with respect to the neutron star's properties [2].

Magnetars have an extremely large magnetic field inside their core relative to other neutron stars. This strong field leads to a great deal of instability within the star, resulting in a lower level of spherical symmetry, or higher ellipticity. The ellipticity is highest when these stars are first born, when they are spinning at their highest speeds and are most unstable in their core as a result [3][2]. A higher rotational frequency and ellipticity will produce the strongest gravitational waves with which our detectors are be most likely to measure. These stars also have a high rate of spindown early in their lives as a result of the massive release in energy through these gravitational waves. The signature of their signal through time can be used to teach us more about neutron stars as a whole [2].

Since our detectors are not yet sensitive enough to find these long transient-wave signals underneath the noise, we must look for ways to bring out these signals using computational methods. In this project, we use machine learning techniques with a convolutional neural network to find the data that contains our desired signals. Time-frequency maps, modeled after ones that would be generated by LIGO detectors, will be simulated containing our desired signal to train our neural network. This process will be described in Sec. II.

The goal of our machine learning is to allow the computer to "see" the signals generated by these magnetars, matching those described in theory, beyond the strong background noise. The relationship between time and frequency in our magnetar system is non-linear however, so finding the proper parameters for a convolutional filter is not a trivial task [4]. To do this, many rounds of training is required to find the optimal network size and training loop parameters for our machine learning model.

This method of searching for these particular signals is especially useful because finding one in our time-frequency maps confirms it is most likely what we are looking for due to its unique signature. With ordinary, slow spinning, low magnetic field neutron stars, there is negligible change in frequency over the same time scales resulting in a horizontal line in a time-frequency map. This can be much more easily found through machine learning, but it is impossible to differentiate with the many sources of noise that would generate the same type of signal. Using the same method would not be as practical in these scenarios.

Our machine learning model will be created in python using the PyTorch framework[1] as described in Sec. III. Multiple rounds of training will be done to find the best parameters of our network so the computer will be able to find the signal as it is generated at amplitudes near the amplitudes of the background noise. This process is described in Sec. IV with the results of this training illustrated in Sec. V.

II. SIMULATING A SIGNAL

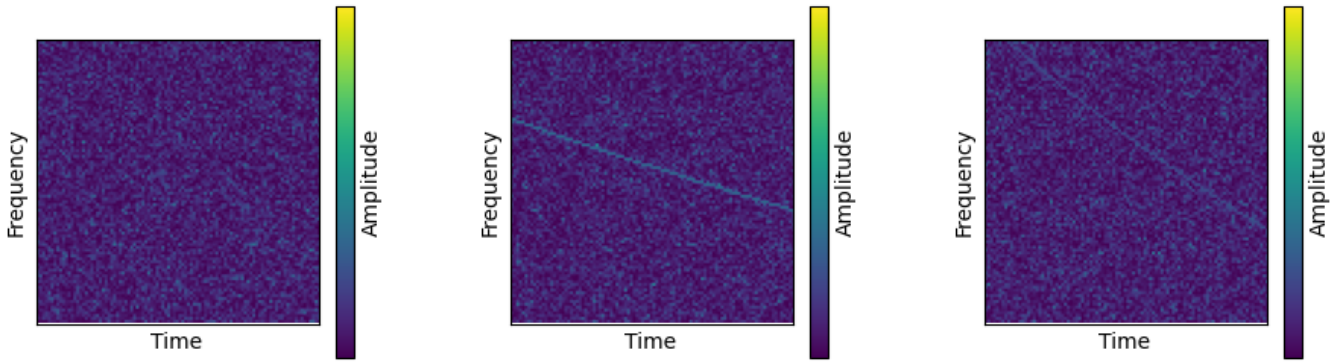
In order to create a neural network that can determine the presence of this specific long-transient gravitational wave signal, simulated data is required for training. To properly train our network, the simulated data needs to fill two requirements: it must resemble real data that would come from the LIGO detectors, and each piece of data must have corresponding metadata telling us whether or not there is a signal to easily calculate a value for the loss. The time-frequency maps from our simulation are square matrices of variable size with the absolute value of Gaussian noise arbitrarily around an amplitude value of zero and standard deviation of 0.5 as shown in Fig. 1a. The magnitudes of the amplitudes here are arbitrary since only the relationship between the signal amplitude and the noise amplitudes is relevant to create a properly trained model. Each map will cover a time of 1200 seconds and a frequency range of 150 hertz. After creating a map of noise, we insert a random signal (with certain constraints) onto a random part of the map.

In this study, we focus on magnetars that rotate around one of their principle axes with the following equation for the spindown [4]

$$\dot{f} = -k_{GW} f^5 \qquad k_{GW} = \frac{32}{5}\frac{GI\epsilon^2\pi^4}{c^5} \tag{1}$$

We use the the inertia of a typical neutron star with a value of $I = 10^{38}$ Kg m^2. The ellipticity (ϵ) represents the measure of asymmetry in the spherical shape of the star and is dependant on the star's properties. For this simulation, we use a range of ϵ values from $3e^{-4}$ to $3e^{-4}$, big enough to get a noticeable change in frequency within our 1200 second time frame. This equation assumes that gravitational waves are the dominant contribution to the energy loss of the star rather than electromagnetic forms of energy [4]. Being a stronger source for the long-transient gravitational waves we are searching for, LIGO is more likely to detect these signals from these magnetars rather than any other neutron star source, making them the best model for our simulation.

[1] Reference book used for this project can be found here[5]



(a) (b) (c)

FIG. 1: Time-frequency maps created by the simulation as described in Sec. II. (a), empty map of Gaussian noise centered around amplitude of 0 with standard deviation of 0.5. (b), map of noise with one possible signal of amplitude 1.5. (c), map of noise with one possible signal of amplitude 1.

To get an equation of frequency with respect to time, we must solve the differential equation in Eq. (1) to get the following equation

$$f = f_0 \left(1 + \frac{t}{\tau}\right)^{-\frac{1}{4}} \qquad \tau = \frac{1}{4 f_0^4 k_{GW}} \tag{2}$$

The initial frequency in our simulation will be anywhere in the range $f_0 = 1250$ to 2000. We take output from this function with randomly chosen parameters and map it out onto the empty noise matrix. The starting point of the function (the point in the signal with highest frequency) is placed at random points along the left or top axes. Examples of this are shown in Fig. 1b and in Fig. 1c. This random origin placement is done to mimic the time-frequency maps that would be created from LIGO data.

A specified number of matrices are created along with a specified percentage of empty matrices when the simulation is run. Since LIGO uses HDF5 files to store its data, each matrix is saved as an HDF5 file, with the entire data set being saved in its own HDF5 group. The program also saves a Boolean value as metadata attached to each matrix, labeling it as containing a signal or not. After running the simulation, we are able to create and store a data set that resembles real data from LIGO observatories with each piece of data labeled as having a signal or not. Next, we must create the model that will use this data to learn to find long-transient signals within the noise.

III. CREATING A MODEL

The first thing required in any machine learning project is to convert the raw data into data that can be used as an input to our neural network. In PyTorch, we are able to create a custom dataset class that acts as the vehicle to bring data from the HDF5 files that we provide directly to the program in a usable format. Later, when we start to train our model, PyTorch's dataloader function will use this custom class to extract the exact data we need in the format we need it to be in.

Next, need to build the model that our data will go through. It is going to start off as a simple convolutional neural network. The exact structure of the model can be found in Lst. 1. Later on, we will adjust its size and each layers parameters to make it more complex, allowing for better results. Our new model takes an individual time-frequency map as an input and outputs a list of two numbers within the range -1 to 1. A higher value at index 0 indicates there is no signal, and a higher value at index 1 indicates the presence of a signal.

After creating a way to access our data and a model to accept it, the information we get from running the data through it is useless since the parameters of the model (the weights and biases) are set at random. We need a way to train the parameters of our model so that it may accurately tell us what time-frequency maps contain a signal. To prepare for this, we must create a training loop.

The training loop is meant to do two things. First of all, it must train the model to calculate the parameters that will generate a meaningful output. Second, it also must give us information on how we might change the inputs of the training loop to improve the results if the desired parameters were not found. In order to train the model, our loop requires an input of the model we want to train, the data set we will use to train in the form of a PyTorch dataloader object, an optimizer, and a loss function. The data is put into the model initially in batches of size ten (arbitrarily) and the average loss is calculated using the loss function. The loss represents a numerical value to describe how much error a model has in predicting the contents of our data. This information is given to the optimizer which goes back through the

Listing 1: Initial Model Class

```
import torch
from torch import nn
import torch.nn.functional as F

class Network(nn.Module):
    def __init__(self, MatDim):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=7, padding=3)
        self.conv2 = nn.Conv2d(8, 4, kernel_size=7, padding=3)
        self.lin1 = nn.Linear(4*int((MatDim/4)**2), 10)
        self.lin2 = nn.Linear(10, 2)
        self.MatDim = MatDim                    #allows for variable size maps

    def forward(self, x):
        x = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        x = F.max_pool2d(torch.relu(self.conv2(x)), 2)
        x = x.view(-1, 4*int((self.MatDim/4)**2))
        x = torch.relu(self.lin1(x))
        x = self.lin2(x)
        return x
```

model, known as back propagation, to adjust its parameters in a way that is expected to lower the loss. Once the entire data set has been through the model, one epoch is completed. After this happens for a specified number of epochs, the training is completed and we have a fully trained model. In order to be able to determine if the training was successful, we also need to input a second set of data that will not be used to train, but will act as a verifying set of data to see how the model is performing after each epoch. This data will be put the through the model and a verifying loss will be calculated without any back propagation like there is for the training data loss. Rather, this loss will be compared to the training loss to determine how successful the training was.

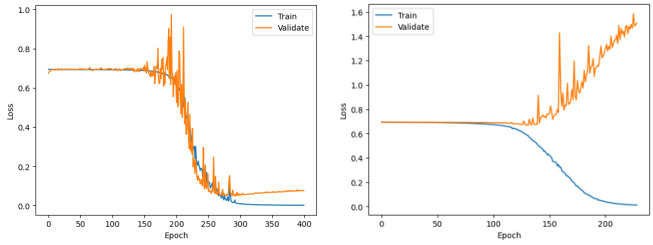
IV. TRAINING OUR MODEL

Having created a training loop, a simple neural network, and a set of data, we now need to begin to train our network. The goal of our training will be to find the weights and biases of our model that will be able to determine the presence of a signal at amplitudes similar to the amplitudes of the noise. To accomplish this, we run several training trials, changing the inputs to our training loop until we get a successfully trained model at a lower signal amplitude than what was previously recorded. Changing one variable at a time will tell us how each impacts the overall training as well as what combination of inputs might be necessary to achieve the best result for our specific project.

To start training, we need to import the data we simulated from our HDF5 file using the custom dataset class we created. We do this by using the dataloader function provided by PyTorch to create training and validating dataloader objects usable by our training loop. We also need to choose an optimizer and loss function from the PyTorch library that we think will function well with our data. We initially chose, somewhat arbitrarily, to use the standard gradient descent (SGD) for our optimizer, and the CrossEntropyLoss function was used as our loss function throughout the project.

Providing the training loop with these inputs as well as the number of epochs we want to train over, we can run the function and get a trained model as our output. With mostly guessing on the inputs for the training loop and the properties of our model, we are likely not going to get a desirable set of weights and biases. This is where our validating loss becomes useful. In Fig. 2, we see two training trials with validating losses and training losses plotted against epoch number. The model is training properly if we see a graph like Fig. 2a where the validating loss is decreasing with the training loss. Our network is learning to accurately identify the data outside the set being used to train it, indicating it is able to find signals in data that the model has never seen before. In Fig. 2b, we see the validating loss increasing as the training loss decreases. This is known as over-fitting. The network is learning to recognize the specific time-frequency maps in the training data rather than learning a general pattern to be able to find a signal in any data. You can begin to see this happen in Fig. 2a after the verifying loss reaches a minimum and the training loss stays close to zero as well. It is okay here however, since the model was able to generalize before over-fitting rather than over-fitting right away.

Along with the loss graph helping us see how successful the training was, we calculate the accuracy of the model at the epoch with the lowest validating loss. A large data set is created that is separate from both the



(a) (b)

FIG. 2: Loss plots of two training attempts. (a), a successful training attempt; over-training occurs after a desirable model has been found. (b), a failed training attempt; over-training occurs right away, before a desirable model is reached.

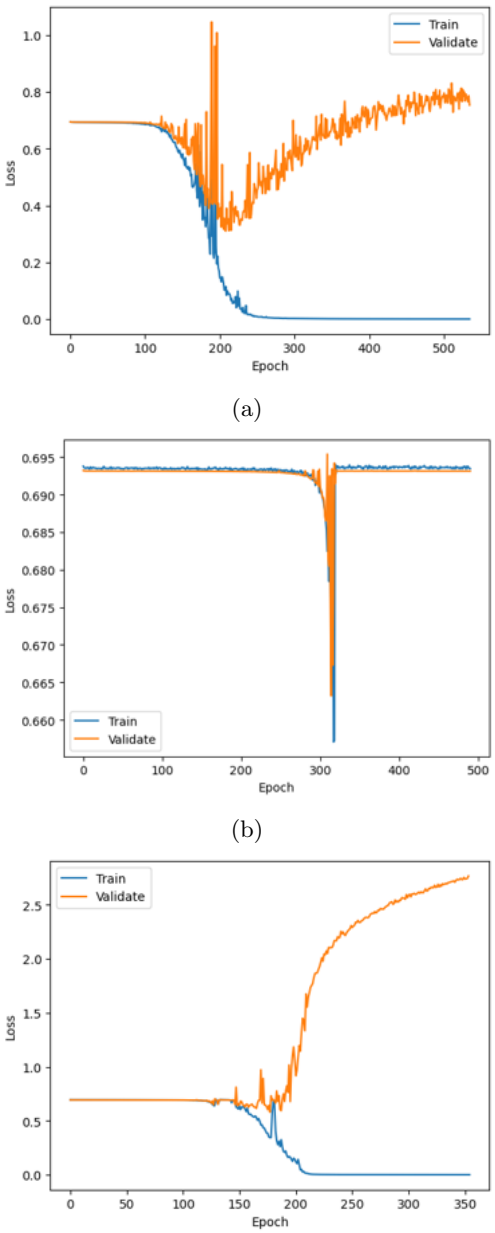
training and validating sets, fed into the model with the parameters that provided the lowest validating loss, and the percentage that the model gets correct is calculated.

Having everything set up to train our model with the data we need to train, we can now start the process of training and refining the inputs to our training loop. We begin by simulating the strength of the signal to a large enough amplitude relative to the noise so as to create a successfully trained model, but not too large leaving no room for improvement. An accuracy around 80 percent works well here. Each run through the loop afterwards will have one thing changed in either the model, optimizer, or data sets, and the new accuracy will be recorded. If the accuracy is found to be better than a previous best on any given trial, the amplitude of the signal will be reduced until the accuracy of the model is again around 80 percent, and the process is repeated. It is repeated until the model is able to be successfully trained at a desired signal amplitude relative to the noise.

V. RESULTS

To begin our training and optimizing of our parameters, we started by running our simulation with a signal of amplitude 1.5 relative to the Gaussian noise centered at an amplitude of 0 with a standard deviation of 0.5. After creating the data sets and running them through our initial model shown in Lst. 1, we calculated an accuracy of 85%. Slowly lowering the signal amplitude over several trials showed us that we could have an amplitude of 1.225 and still get an accuracy of 88.15%. The loss graph for this trial can be seen in Fig. 3a. Here, it is easy to tell when the model begins to over-fit to the training data and the verifying loss starts to increase. The lowest value for the verifying loss was calculated to be at epoch 209; the parameters of the model at this point were used to calculate the accuracy of the entire trial. Any lower signal frequency produced a model with an accuracy well below 80%.

This new data with signal amplitude 1.225 was then



(a)

(b)

(c)

FIG. 3: Loss graphs at each trial where data with a lower amplitude signal was successfully used to train our model. (a), signal amplitude of 1.225; model trained to accuracy of 88.15%. (b), signal amplitude of 1.15; model trained to accuracy of 70.6%. (c), signal amplitude of 1; model trained to accuracy of 74.7%.

used in the following trials. Properties of the other inputs of the training loop were then altered one at a time to see what would improve our accuracy to above 88.15%. The first thing we measured that made an improvement to the accuracy was changing the number of convolutional layers in our model. Increasing it to four layers, as shown in Lst. 2, produced the best accuracy of 90.6%

Listing 2: Final Model Class

```
import torch
from torch import nn
import torch.nn.functional as F

class Network(nn.Module):
    def __init__(self, MatDim):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=7, padding=3)
        self.conv2 = nn.Conv2d(8, 8, kernel_size=7, padding=3)
        self.conv3 = nn.Conv2d(8, 8, kernel_size=7, padding=3)
        self.conv4 = nn.Conv2d(8, 4, kernel_size=7, padding=3)
        self.lin = nn.Linear(4*int((MatDim/4)**2), 2)
        self.MatDim = MatDim                    #allows for variable size maps

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(F.relu(self.conv2(x)),2)
        x = F.relu(self.conv3(x))
        x = F.max_pool2d(F.relu(self.conv4(x)),2)
        x = x.view(-1, 4*int((self.MatDim/4)**2))
        x = self.lin(x)
        return x
```

when the momentum of the SGD optimizer was increased to 0.3. With this new model, we were able to reach a signal strength of 1.15 at an accuracy of 70.6%. The loss graph of this training trial can be found in Fig. 3b. The usual curve of decreasing losses followed by over-fitting is not seen here. We believe this is because the optimiser was not well calibrated for this specific model and data. At the moment the loss returns back to around its original value and plateaus, the optimizer likely adjusts the model's parameters by too large an amount and passes the local minimum in the loss the optimizer was gravitating towards. With a better, or better defined, optimizer that was not found in this study, it is likely much better results can be achieved here.

The next and final variation we were able to find that improved our accuracy from this point was the number of linear layers. Decreasing this number to one, as shown in Lst. 2 gave us an accuracy of 74.1% at a signal amplitude of 1.15. With this new model as our final model, we were able to get the amplitude of our signal down to 1 with an accuracy of 74.7%. The loss graph can be seen to return to an expected shape in Fig. 3c. The validating loss dips down very slightly before the model begins to over-train, but this is enough to get a satisfactory accuracy at a lower amplitude than anything that was previously recorded.

Everything else that was changed throughout the many training trials either prevented training from occurring at any capacity, produced much lower accuracies, or produced no major difference in accuracies. The things we found to prevent training from happening were increasing the stride in the convolutional layers of the model, using tanh or sigmoid activation functions instead of relu, increasing the size of the time-frequency maps, and changing the optimiser from SGD to Adam. Decreasing the batch size from 10, as well as changing the kernel size from 7 in any way lowered our accuracy. Increasing our batch size appeared to make no difference, so starting off with the numbers we used for this and the previous parameters seem to be fairly lucky guesses. Using a leaky relu activation function also seemed to make no difference in the training. Changing the padding in each convolutional layer was quite inconsistent when looking at the accuracies of each trial. With a padding of 0, 1, and 2, respectively, the accuracy was about the same, the model did not train at all, and the accuracy was decreased. Although we found only three variables we could change to improve our results, this could very well be shown to be true for some of the other variables we tested if more trials were completed.

VI. CONCLUSION

In the continuing search for different types of gravitational waves, physicist are looking for ways of spotting long-transient waves from newly born magnetars in LIGO data. In this study, we were able to build a machine learning model and train it with simulated data to find these types of signals down to an amplitude of 1 in time-frequency maps of Gaussian noise centered around an amplitude of 0 and a standard deviation of 0.5 at an accuracy of 74.7%.

This, however, has a long way to go before we are able to achieve accurate results on real LIGO data. The noise coming from the detectors is not a simple Gaussian for example. Our data's noise would have to be simulated to

reflect the capabilities of the detectors more accurately. Adding weights to the noise levels based off the experimental strain values at each frequency of the LIGO detectors is one way to do this. Also, the signal would have to be simulated as having a varying amplitude, since it is most likely not going to be detected at a constant value. After improvements to the simulation are made, many more training trials will have to be done in order to lower the signal amplitude to the levels expected in LIGO observatories. Many trials with constant input parameters should be done, especially for the changes that seemed to make little to no impact on the accuracy as described in Sec. V. Since the parameters of the model are randomly initialized, each trial is different. Getting averages of accuracies rather than individual ones would be imperative to improve our model as much as possible.

VII. ACKNOWLEDGEMENTS

I would like to thank Paul Fulda, Peter Wass, Kathryn McGill, and Nathaniel Strauss at the University of Florida for support. I would also like to thank Marco Serra, Pia Astone, and Christiano Palomba from the Istituto Nazionale di Fisica Nucleare for support as well as Sapienza University of Rome for hosting this research. This project was funded by the National Science Foundation fellowship number PHY-1950830.

[1] S. Dall'Osso, B. Giacomazzo, R. Perna, and L. Stella, Gravitational waves from massive magnetars formed in binary neutron star mergers, Astrophys. J. **798**, 25 (2015), arXiv:1408.0013 [astro-ph.HE].

[2] S. Dall'Osso, L. Stella, and C. Palomba, Neutron star bulk viscosity, 'spin-flip' and GW emission of newly born magnetars, Mon. Not. Roy. Astron. Soc. **480**, 1353 (2018), arXiv:1806.11164 [astro-ph.HE].

[3] C. Thompson and R. C. Duncan, The Soft gamma repeaters as very strongly magnetized neutron stars - 1. Radiative mechanism for outbursts, Mon. Not. Roy. Astron. Soc. **275**, 255 (1995).

[4] M. Maggiore, *Gravitational Waves. Vol. 1: Theory and Experiments* (Oxford University Press, 2007).

[5] E. Stevens, L. Antiga, and T. Viehmann, *Deep Learning with PyTorch* (Manning Publications Co., 2020).