Distinguishing Between Cosmic Strings signal peaks from background noise in Gravitational-Wave Data Using a Neural Network

Marc Duran Gutierrez[1]

[1] *Department of Physics, Utrecht University*

February 24, 2025

Abstract

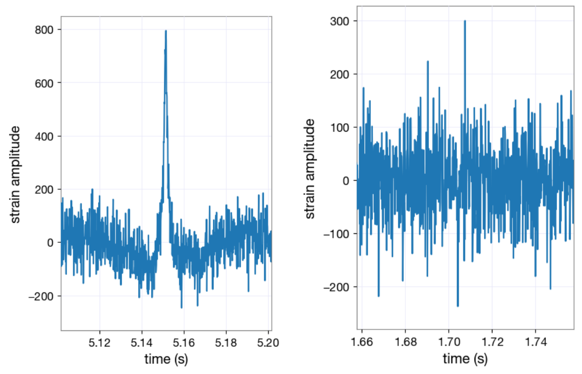
The detection of gravitational waves from cosmic strings is hindered by the sensitivity of current detectors and background noise. Background noise can be similar to a cosmic string signal, so it is crucial to develop a way to distinguish between the two so that future detectors, such as the Einstein telescope, have a shot at detecting cosmic string signals. In this work, we train a convolutional neural network (CNN) for this task, consisting of seven convolutional layers for feature extraction, and three fully connected layers for binary classification. The model achieved an accuracy of 95%. This study therefore demonstrates the potential of neural networks to enhance the detection of gravitational waves.

1 Introduction

The first detected gravitational wave (GW) signal was GW150914[1] in 2015. GW signals can originate from several sources, one of which is cosmic strings. Cosmic strings have not been detected yet, but future telescopes, such as the Einstein telescope will have a higher sensitivity, improving the chances of detecting a cosmic string signal [2]. Since the first observed GW, many technological advancements have been made, yet finding a signal in the background noise can still be difficult. This is because noise can look like a signal, so it is important to develop a way to distinguish between a true signal and noise that looks like a signal. In this work, we focus on the use of machine learning to distinguish between a simulation of a cosmic string signal and background noise, which could be detected by the Einstein Telescope in the future.

1.1 Cosmic strings

Cosmic strings are one-dimensional topological defects and originate from field theories [3]. These defects could have formed from spontaneous symmetry breaking in the early universe [4]. Cosmic strings are of interest since they affect our understanding of quantum field theory and string theory models in the early universe.



(a) The signal produced by the cusp of a cosmic string. (b) Background noise with a peak.

Figure 1: The cosmic string signal and background noise.

Cosmic strings appear at cosmological scales as thin strings with large densities, and their motion is well described by the Nambu-Goto action [5]. Cosmic strings can both be open strings or closed loops, and two cosmic strings can interact with each other.

To detect cosmic strings we first need to understand what a GW signal originating from a cosmic string source looks like. Cosmic strings can produce GW signals in multiple ways. Examples include the formation of cusps and kinks [6]. Here we will focus on the cusps. A cusp is a singularity, where a point traveling along the curve would have to turn around. When this occurs, the physical string snaps into a cusp shape and is instantaneously accelerated to the speed of light at that point. A burst GW is then emitted in the direction of acceleration [3]. Such a signal can be seen figure 1a.

With the knowledge of what the signal looks like, the next step is to search for a signal. The problem with the search for signals is that background noise can also form a peak, as can be seen in figure 1b. Since a GW signal from cosmic strings can be so similar to background noise, it is crucial to develop a method to distinguish between them.

1.2 Machine learning and neural networks

Machine learning (ML) is a subgroup of artificial intelligence that focuses on the study of statistical algorithms that learn from datasets. These algorithms can then apply this knowledge to new, unseen datasets. There are many different kinds of ML algorithms [7] but in this work, we focus on the application of such a neural network (NN) on GW datasets.

A NN in machine learning is based on the neural structure of the brain, consisting of nodes that are connected. These nodes loosely resemble neurons in the brain. Each node receives a "signal" from the connected nodes. This signal is a number. The node then uses the numbers to compute its output by passing them through some nonlinear function. This function is called the activation function. The strength of the signal at a connection is given by the corresponding weight, these weights are constantly being adjusted during the learning process [8].

Each neural network starts with an input layer, which is where the signal enters the NN. It then goes through a collection of hidden layers, before finally arriving at the last layer, which is the output layer. This output can have many forms, but in this work, the output layer is a binary classifier. The NN is trained by passing through a training dataset and checking if the output corresponds to the label given to the data. After each data point passes through the NN parameters are updated accordingly. Then a validation set is used to check if the NN has improved. It is important to note that the NN does not learn from this validation set since it is only used to check the accuracy. This process can be repeated a number of times, the number of repeats is referred to as the number of epochs. One can also choose to stop the learning early, for example when a certain accuracy is achieved. Lastly, the NN is given a test dataset. This dataset is used to check the accuracy of the model after learning [9].

2 Method

2.1 Data

The data used to train the neural network is computer-generated data. There are two datasets; one with signals injected into background noise, and one with just background noise. Each data set has the same structure and has three sets of strain data, one for each arm of the Einstein telescope.

For each noise dataset, Gaussian noise was generated. Each dataset consists of 65536 data points, spread over a time interval of 8 seconds.

For each signal dataset, again Gaussian noise was generated. This noise was then injected with a cusp signal. These cusp waveforms were generated using the `LALSimulation` package [10]. The specifics of this procedure can be found in [3].

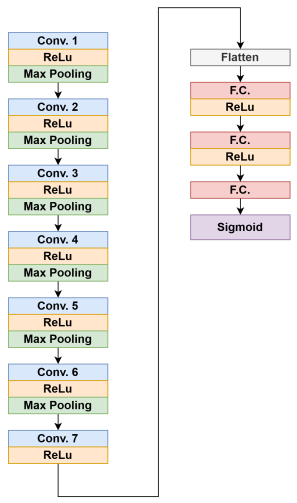


Figure 2: CNN architecture. F.C. stands for fully connected layers. Note that while a Sigmoid activation is shown in the figure, there is no explicit Sigmoid function in the implementation. This is because the chosen loss function, BCEWithLogitsLoss, incorporates the Sigmoid activation internally.

2.2 Neural network architecture

The proposed model used in this work is a convolutional neural network (CNN) designed for a binary classification problem. The architecture, shown in figure 2, extracts features from the input through seven convolution layers, followed by fully connected layers that classify the input as either a signal peak or background noise. The network accepts three initial time series generated as if detected by the three different arms of the Einstein telescope, which are convolved into a feature map. This input data can be represented as a 3D tensor with the shape $[B, D_{in}, W_{in}]$, where B is the chosen batch size, the depth D denotes the number of data channels, which initially is three corresponding to the number of different arms of the telescope time series. The width W is the length of the time series.

Conv. Layer	**In Channels**	**Out Channels**	**Kernel Size**	**Stride**	**Padding**
1	3	16	11	1	3
2	16	32	7	2	2
3	32	64	5	2	1
4	64	128	5	2	1
5	128	256	3	2	1
6	256	256	3	2	1
7	256	256	3	2	1

Table 1: Convolutional Layer Parameters

After passing through a convolutional layer, the data are modified by applying a specific number f of kernels, also called filters, with the shape [k, input channels]. Here, k denotes the time window for each convolution operation. Specifically, kernels slide over the time axis of the input data, performing element-wise multiplications and then adding the elements. Apart from the window size (k), additional parameters such as stride and padding can be adjusted. Stride s corresponds to the step size at which the convolutional filter moves across the input while padding p is the number of zeros added to each side of the input signal. This helps preserve the dimensions of the input during transformations. Each kernel operation produces a channel of feature map which encodes specific patterns or correlations over the window k. The resulting output tensor has the shape

$$\begin{pmatrix} B \\ D_{\text{out}} \\ W_{\text{out}} \end{pmatrix} = \begin{pmatrix} B \\ f \\ \left\lfloor \frac{W_{\text{in}}+p-k}{s} \right\rfloor + 1 \end{pmatrix}. \tag{1}$$

Here, D_{out} and W_{out} are analogous to the input data depth and width but after the convolution.

The first layer applies 16 filters of size $k = 11$ with a stride of $s = 1$ and padding of $p = 3$. Subsequently, layers 2 to 7 progressively increase the number of filters (from 16 to 256), see table 1, while reducing the feature map's spatial size. This increase enhances the network's capacity to capture complex patterns, but it also raises the chances of overfitting.

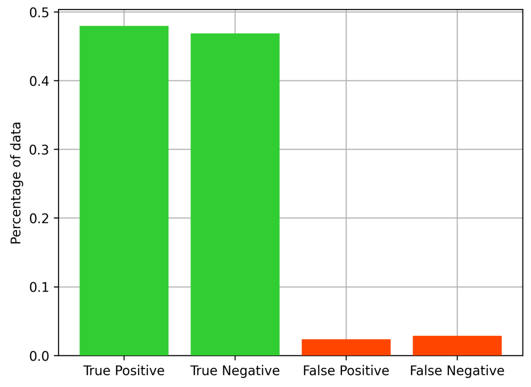
Layer	**Input Shape**	**Output Shape**
Conv. 1	3, 65536	16, 65532
Pool. 1	16, 65532	16, 32766
Conv. 2	16, 32766	32, 16382
Pool. 2	32, 16382	32, 8191
Conv. 3	32, 8191	64, 4095
Pool. 3	64, 4095	64, 2047
Conv. 4	64, 2047	128, 1023
Pool. 4	128, 1023	128, 511
Conv. 5	128, 511	256, 255
Pool. 5	256, 255	256, 127
Conv. 6	256, 127	256, 64
Pool. 6	256, 64	256, 32
Conv. 7	256, 32	256, 16
Flatten	256, 16	4096
F.C. 1	4096	256
F.C. 2	256	64
F.C. 3	64	1

Table 2: Data shape transformations in the Convolutional Neural Network for each batch

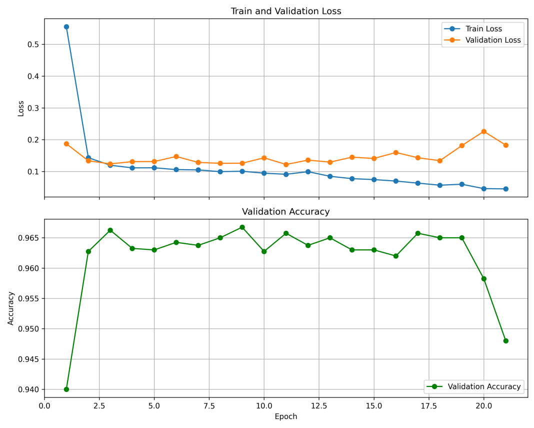
Moreover, after each convolution, an activation function `ReLu` and max pooling are applied. `ReLu` introduces non-linearity by replacing all negative values with zeros. Then, max pooling keeps the maximum value in a specific pooling window. This helps reduce the dimensions of feature maps while retaining important information [11]. After the feature learning layers, the multidimensional output is flattened into a 1D vector. This vector works as the input for the fully connected layers, which make the final decision. In other words, they act as the "decision-making" layers after combining features from convolutions. All these data shape transformations are summarized in table 2.

2.3 Workflow

Before entering the NN model, the data is divided into three subsets: a training batch (70%), a validation batch (20%), and a testing batch (10%). Then, the model architecture is initialized using the `ConvNN` class, which defines the convolutional and fully connected layers for feature extraction and classification. To optimize the model, the function uses the `Adam` optimizer, one of the most popular optimization algorithms in deep learning. It is an extension of stochastic gradient descent (SGD) that incorporates two key concepts: adaptive learning rates for each parameter and momentum-based updates for faster convergence using past gradients [12]. The optimizer is configured with an initial specific learning



(a) Normalized distribution of predicted classifications from the NN compared to the actual data classifications. The simulation includes an equal number of signal and background events (10,000 each). The results show high classification accuracy, with the True Positive bar (correctly classified as signal) and True Negative bar (correctly classified as background) dominating the distribution. Misclassifications account for less than 10%, as indicated by the red bars.



(b) Training process of the NN, illustrating the training loss (blue line) decreasing steadily with each epoch, and the validation loss (orange line), which remains low but increases slightly towards the end. The validation accuracy (green line) also declines slightly at the end, triggering the early stopping algorithm, which halted the training after 21 epochs.

Figure 3: Resulting figures showing the data classification distribution (left) and the learning process (right)

rate of 0.001 and a weight decay equal to 0.0001 for regularization. For the loss function, it is employed `nn.BCEWithLogitsLoss`, which allows for more stability than using a sigmoid followed by a binary cross-entropy loss per separated, that is the reason why there is no sigmoid function after the last fully connected layer [13].

The training process is a loop of the model undergoing multiple epochs. Each epoch consists of two different phases: training and validation. In the training phase, the weights are allowed to be updated with backpropagation. The `trainloader` consists of data batches with signals and labels, and for each batch, the model computes predictions through a forward pass. The loss is calculated based on the predictions and truth labels. The optimizer then updates the model's parameters to minimize the loss. The average training loss for the epoch is accumulated and stored for later analysis. Then, the testing phase conducts an unbiased test, utilizing the weights gained from the previous phases. Using the `validationloader`, the model processes the data in batches to calculate predictions, the subsequent loss, and the accuracy in order to monitor the performance. To prevent overfitting and unnecessary computations, the model incorporates an early stopping mechanism. This feature monitors the validation loss across epochs. If the validation loss does not improve for a number of epochs, training is terminated. This number is specified by the `patience` parameter.

Once the training and validation phases are completed, the model is evaluated with the test dataset. Similar to the validation phase, the predictions and accuracy are computed with the ground-truth labels. This test phase provides an unbiased final assessment of the performance. Finally, histograms are made to visualize the percentage of both false and true positives and negatives. Therefore, this model is a robust pipeline for binary time series classification using a convolutional neural network.

3 Results

To evaluate the performance of the NN, a simulation was conducted using 10.000 cosmic string events and an equal number of background events. The results of this run are illustrated in figure 3a and figure 3b, which depict the classification distribution and training process, respectively.

Figure 3a shows a high classification accuracy as evidenced by the high percentage of True Positives (correctly classified as signal events) and True Negatives (correctly classified as background events). Notably, only 5% of the events were misclassified, aligning with the final validation accuracy of approximately 95% shown in figure 3b. The detailed classification statistics are summarized in table 3.

	True Positive	True Negative	False Positive	False Negative
Events	959	937	47	57

Table 3: Raw classification results of the unbiased testing phase in the NN simulation.

Figure 3b provides insights into the training process of the NN. The training loss decreases with each epoch, reaching a minimum value of approximately 0.03, while the final validation loss reaches ± 0.18. The validation accuracy initially rises rapidly, converging at 96.5% before slightly declining towards the end of the run. Although the simulation was initially planned for 70 epochs, it was terminated early at 21 epochs due to the observed decrease in validation accuracy.

Finally, the implemented efficiency measures led to a simulation time of approximately 7 hours on a 64-core GPU. This setup utilized a batch size of 100 events and incorporated an early stopping mechanism, which halted the simulation after 10 epochs and prevented overfitting.

4 Discussion and Conclusion

This work showed promising results regarding the application of NN on classification between cosmic string signals and background noise, achieving an accuracy of 95%. Despite this success, there are several areas for improvement and further research can be done. Firstly analyzing misclassified events to identify their properties could provide valuable insights into the network's limitations. This can lead to modifications in the convolutional layers, potentially increasing the accuracy. Furthermore, the fine-tuning of initial parameters, such as the batch size and learning rate (perhaps by implementing the latter as a more dynamic parameter), could significantly improve the efficiency of the NN, enabling a faster runtime and broader applicability for larger datasets. Upgrading to more advanced hardware could also reduce computation time.

Future research should be able to classify more challenging data, such as glitches, which closely resemble cosmic string signals. To achieve this, the NN might need to focus more on waveform details, prompting to change the convolutional layers to prioritize the classification of shape over size. Another promising direction involves transforming the input data from strain measurements of gravitational waves into spectrograms, which might provide more information for the NN to learn from.

References

[1] B. A. et al. (LIGO Scientific Collaboration and V. Collaboration), "Observation of gravitational waves from a binary black hole merger," *Phys. Rev. Lett.*, vol. 116, p. 061102, Feb 2016.

[2] Einstein Telescope Collaboration, "Einstein telescope official website," 2025. Accessed: 2025-01-24.

[3] Q. Meijer, M. Lopez, D. Tsuna, and S. Caudill, "Gravitational-wave searches for cosmic string cusps in einstein telescope data using deep learning," *Physical Review D*, vol. 109, Jan. 2024.

[4] K. Schmitz and T. Schröder, "Gravitational waves from cosmic strings for pedestrians," 2024.

[5] M. Sakellariadou, "Cosmic strings and cosmic superstrings," *Nuclear Physics B - Proceedings Supplements*, vol. Volumes 192–193, 2009.

[6] R. e. a. Abbott, "Constraints on cosmic strings using data from the third advanced ligo–virgo observing run," *Phys. Rev. Lett.*, vol. 126, p. 241102, Jun 2021.

[7] IBM, "What is machine learning (ml)?," *IBM*, 2025.

[8] L. Hardesty, "Explained: Neural networks," *MIT News*, 2017.

[9] S. C. M. L. Alessandro Grelli, Marc van der Sluys, "Computational aspects of machine learning," 2024. Lecture notes for CAoML.

[10] LIGO Scientific Collaboration, Virgo Collaboration, and KAGRA Collaboration, "LVK Algorithm Library - LALSuite." Free software (GPL), 2018.

[11] GeeksforGeeks, "Relu activation function in deep learning." https://www.geeksforgeeks.org/relu-activation-function-in-deep-learning/, n.d. Accessed: 2025-01-27.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[13] PyTorch, "torch.nn.bcewithlogitsloss — pytorch documentation." https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html, n.d. Accessed: 2025-01-27.