# Recurrent Neural Networks and Synthetic Gravitational Waves

Edvard B. Rørnes* and Isak O. Rukan†
*Institute of Physics, University of Oslo,*
*0371 Oslo, Norway*
(Dated: December 18, 2024)

Gravitational waves, which are ripples in spacetime produced by cataclysmic astrophysical events such as black hole and neutron star mergers, present challenges for detection due to their faint signals often being buried by noise. The Laser Interferometer Gravitational-Wave Observatory (LIGO) has pioneered the detection of these waves, leveraging laser interferometry to measure tiny spacetime distortions. However, traditional detection methods face limitations in distinguishing weak GW signals from background noise.

In this work, we explore the potential of neural networks (NNs) for GW detection, focusing on Recurrent Neural Networks (RNNs) to analyze untreated GW data obtained from publicly available datasets from [**?** ]. Due to the low signal-to-noise ratio (SNR) of this data and the complexity of noise removal techniques, we instead generate synthetic GW data with controllable SNR for training and testing purposes. This approach allows us to evaluate the performance of our custom RNN implementation against the RNN framework provided by `tensorflow.keras`. While detection on untreated data remains challenging, the synthetic dataset allows for systematic evaluation and refinement of NN-based detection methods. Additionally, we provide a utility program[a] which automatically downloads and labels GW data when the signal duration is known, which may be useful for future work aimed at improving detection on real-world datasets.

## 1. INTRODUCTION

Gravitational Waves (GWs) are the product of some of the most extreme events that occur in the universe. While in theory, just about any accelerating object produces GWs, they are so weak that they can only be detected from the most energetic and cataclysmic events [**?** ]. The only sources of detectable GWs with current technology are the mergers of black holes and neutron stars [**?** ], where the immense masses and high velocities involved generate powerful ripples in spacetime. These ripples propagate outward and cause minute distortions in spacetime itself, which can be measured by highly sensitive instruments.

To date, the most advanced experiment to detect these waves is the Laser Interferometer Gravitational-Wave Observatory (LIGO). LIGO uses laser interferometry to measure the incredibly small displacements caused by passing gravitational waves. However, the signals from gravitational waves are often faint and easily overwhelmed by noise, making detection a difficult task.

Neural Networks (NNs) has become a powerful tool in this context. By training a NN on large datasets of both gravitational wave signals and background noise, these models can learn to distinguish between genuine gravitational wave signals and random noise. This, in theory, may allow for more accurate detection and classification of GW events, even in the presence of significant interference, and may be the future of GW detection [**?** **?** ]. NNs can be trained to recognize patterns in the data that correspond to the characteristic signatures of GWs, improving both the efficiency and reliability of detection algorithms. With the growing amount of data from observatories like LIGO, NNs are playing an increasingly important role in identifying new events and advancing our understanding of the universe.

In this work we attempt to detect GWs on untreated data from [**?** ] by building our own Recurrent Neural Network (RNN), along with using `tensorflow.keras`' RNN to test against our own. To do this we created a program which automatically labels GW files granted that the user knows the duration of the GW signal. The performance on the untreated data from [**?** ] is quite poor due to the untreated data having a signal to noise ratio (SNR) which is far too low for our programs. The process of removing noise from GW is quite advanced, and requires state of the art techniques. Due to time constraints and lack of expertise in this field, we instead created a simple program which generates synthetic GW data. This allowed us to control the SNR and focus on training neural networks on processed data, which simplifies the task vastly and allows us to evaluate our RNN's performance. This however still led to poor performance for both the RNN's, only yielding acceptable results is near trivial tasks where the SNR $\sim 100$. After attempting many different parameter combinations, we eventually ended up using keras' Convolutional Neural Network (CNN) instead.

This project is organized as follows: In Section 2, we introduce the essential theoretical background and the key techniques needed to understand our subsequent analyses. Section 3 then details the implementation of the various neural network models, along with the supporting tools and data-processing strategies employed. We address the intrinsic challenges of gravitational wave de-

* e.b.rornes@fys.uio.no
† icrukan@uio.no
[a] GitHub

tection and their implications for machine learning approaches in Section 4. Finally, in Section 5, we present and discuss our findings, evaluating the performance of the implemented models under a range of conditions.

## 2. THEORY

### 2.1. Gravitational Waves

We will simply give a quick introduction to GWs. For a more detailed analysis, please see any graduate level textbook on general relativity, e.g. [**?** ] or [**?** ].

As mentioned, GWs are ripples in spacetime caused by the acceleration of massive objects, such as merging black holes or neutron stars. For sufficiently small ripples, or for an observer sufficiently far away from e.g. a black hole merger, the GWs are accurately described by first order perturbations of a flat spacetime described by the Minkowski metric, $\eta_{\mu\nu}$, with a small perturbation:

$$g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu}, \quad |h_{\mu\nu}| \ll 1, \tag{1}$$

where the metric signature is $(-,+,+,+)$. The Einstein field equations (EFE) are given by

$$G_{\mu\nu} = 8\pi G T_{\mu\nu}, \tag{2}$$

where $G_{\mu\nu}$ is the Einstein tensor which depends solely on the metric $g_{\mu\nu}$, and $T_{\mu\nu}$ is the stress-energy tensor. Applying the perturbed metric to (2) one can show that to first order this gives the linearized EFE [**?** ]:

$$\begin{aligned} -16\pi T_{\mu\nu} = &\Box h_{\mu\nu} - \partial_\mu \partial^\rho h_{\nu\rho} - \partial_\nu \partial^\rho h_{\mu\rho} \\ &+ \eta_{\mu\nu} \partial^\rho \partial^\sigma h_{\rho\sigma} + \partial_\mu \partial_\nu h - \eta_{\mu\nu} \Box h, \end{aligned} \tag{3}$$

where $h \equiv h^\mu{}_\mu = \eta^{\mu\nu} h_{\mu\nu}$ is the first order trace of $h$ and $\Box$ is the flat space d'Alembertian. Note that we do not perturb $T_{\mu\nu}$ since it is in fact a perturbation itself. Now this equation here is quite complicated, however due to a symmetry in the original Lagrangian one has a so-called *gauge freedom*, i.e. a set of transformations which leave the Lagrangian invariant. This can be used to vastly simplify the equations heavily. A common choice is the Lorenz gauge: $\partial^\mu h_{\mu\nu} = 0$. Using this we can reduce (3) to the much simpler differential equation

$$\Box h_{\mu\nu} = -16\pi T_{\mu\nu}. \tag{4}$$

Specializing to the case corresponding to a vacuum ($T_{\mu\nu} = 0$) then we simply have a plane wave equation

$$\Box h_{\mu\nu} = 0, \implies h_{\mu\nu} = C_{\mu\nu} e^{ikx}, \quad k_\mu k^\mu = 0, \tag{5}$$

where

$$C_{\mu\nu} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & h_+ & h_\times & 0 \\ 0 & h_\times & -h_+ & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \tag{6}$$

for a GW traveling along the $z$-axis. The details for how this is derived and why it must take on this form is spelled out in [**?** ]. $h_+$ and $h_\times$ correspond to the "plus" and "cross" polarization respectively, which determine how the gravitational waves stretch and compress the spacetime it passes through.

The stretching and compression from these polarizations are what is measured by observatories such as LIGO and Virgo. These displacements are often given as the dimensionless quantity **strain**, which is simply given by $\Delta L/L$ where $L$ is the effective length of the detector, and $\Delta L$ is the measured change in $L$ as the GW passes by. Note that in the case of detection, $L$ is not simply the length of the detectors themselves, as they use mirrors to cause the beams of light to bounce back and forth multiple times, effectively increasing the length of the detector.

### 2.2. Recurrent Neural Networks

The Recurrent Neural Networks (RNNs) are a class of neural networks specifically designed to handle sequential data or data with temporal dependencies. Unlike traditional Feed Forward Neural Networks, RNNs are capable of "remembering" information from previous time steps. This is done through the so called 'hidden state', which acts as a form of memory by retaining information about prior computations. The hidden state is essentially an array of data that is updated at each time step based on the input data and the previous hidden state. Although this enables RNN to access the temporal dependencies of the data at hand, it greatly increases the computation time compared to that of the Feed Forward Neural Network (FFNN). The standard RNN consists of only one hidden layer, but it is certainly possible to have more than one hidden layer. In fact, this is commonly referred to as the stacked RNN (SRNN), and we will arrive at this neural network further down. However, firstly, we present the structure and general algorithm for the RNN.

#### 2.2.1. Structure

The RNN processes input sequentially, with information flowing step-by-step from the input to the output through a hidden state $h_t$, where the subscript $t$ represents time. Let the output data $y$ be of shape $(N, p_{\text{out}})$ and input data $X$ of shape $(N, p_{\text{in}})$, where $N$ corresponds to the total amount of time points, and $p_{\text{out}}$, $p_{\text{in}}$ the dimension of the output and input, respectively. The network's operations can be expressed by the following two equations [**?** ]:

$$h_t = \sigma^{(h)} \left( W_{hx} X_t + W_{hh} h_{t-1} + b_h \right), \tag{7a}$$

$$\tilde{y}_t = \sigma^{(\text{out})} \left( W_{yh} h_t + b_y \right). \tag{7b}$$

Here, $\sigma_h$ and $\sigma_{\text{out}}$ is the activation function for the hidden layer and the output layer respectively. $W_{xh}$ is the

weight from input to hidden layer, $W_{hh}$ the hidden layer, $W_{yh}$ the output layer and $\tilde{y}$ the output of the RNN. Let now $t$ be divided into a discrete set of times $(t_i)_{i\in N}$. Substituting (7a) into itself recursively leads to a formula for computing $h_{t_n}$:

$$h_{t_n} = \sigma^{(h)}\Big(W_{hx}X_{t_n} + W_{hh}\sigma_h\Big(W_{hx}X_{t_{n-1}} + W_{hh}\sigma_h\left(\cdots + b_h\right) + b_h\Big) + b_h\Big) \tag{8}$$

This shows that the hidden state at time $t_n$ is dependent on the input $X_t$ for $t \in [0, t_n]$, i.e. the current and all previous time steps.

Generally, $X$ could correspond to a large sampling frequency in time, making the computation of the hidden state $h_t$ in (8) computationally demanding. One typical way of dealing with this is to split the data into 'windows' of size $N_W$ in time. These windows should generally overlap, such that no temporal dependencies across windows are left out.

Splitting the data into windows, we define the hidden state for window $n$ as:

$$\begin{aligned} h_n &= \sigma^{(h)}\left(W_{hx}X_n + W_{hh}h_{n-1} + b_h\right) \\ &\equiv \sigma^{(h)}(z_n) \end{aligned} \tag{9}$$

where $X_n$ is the $n$-th window.

#### 2.2.2. General Algorithm

The error between $y$ and the predicted output $\tilde{y}$, is given by some chosen loss function $L(y, \tilde{y})$,

$$L(y, \tilde{y}) = \frac{1}{N}\sum_{n=1}^{N} l(y_n, \tilde{y}_n), \tag{10}$$

where $l$ is some error-metric. For some learning rate $\eta$, the standard update rule for the weights and biases is given by:

$$W \leftarrow W - \eta\frac{\partial L}{\partial W}, \quad b \leftarrow b - \eta\frac{\partial L}{\partial b}. \tag{11}$$

This transformation may be extended using optimization methods aimed at handling exploding gradient, faster convergence, avoiding local minimas, etc. We covered three of these optimization methods in [**?** ]; the root mean squared propagation (RMSprop), the adaptive gradient (AdaGrad) and the adaptive moment estimation (Adam).

Compared to the FFNN, computing the gradient of $L$ with respect to the weights leads to a somewhat more complicated expression due to the temporal dependencies between the hiden states. This can be seen by writing out the partial derivative of the loss function with respect to the weight $W$, being either $W_{hx}$ or $W_{hh}$ (cf. [**?** ]):

$$\frac{\partial L}{\partial W} = \sum_{n=1}^{N}\frac{\partial L}{\partial \tilde{y}_n}\frac{\partial \tilde{y}_n}{\partial W} \tag{12}$$

$$= \sum_{n=1}^{N}\frac{\partial L}{\partial \tilde{y}_n}\frac{\partial \tilde{y}_n}{\partial h_n}\frac{\partial h_n}{\partial W}, \tag{13}$$

where we can use backpropagation through time (BPTT) to write [**?** ]:

$$\frac{\partial L}{\partial W} = \sum_{n=1}^{N}\sum_{k=1}^{n}\frac{\partial L}{\partial \tilde{y}_n}\frac{\partial \tilde{y}_n}{\partial h_n}\frac{\partial h_n}{\partial h_k}\frac{\partial h_k}{\partial W}. \tag{14}$$

Applying the chain rule in succession we obtain:

$$\begin{aligned} \frac{\partial h_n}{\partial h_k} &= \frac{\partial h_n}{\partial h_{n-1}}\frac{\partial h_{n-1}}{\partial h_{n-1}}\cdots\frac{\partial h_{k+1}}{\partial h_k} \\ &= \left[W_{hh}^T\cdot\sigma_h'(z_n)\right]\left[W_{hh}^T\cdot\sigma_h'(z_{n-1})\right]\ldots\left[W_{hh}^T\cdot\sigma_h'(z_{k+1})\right] \\ &= \prod_{j=k+1}^{n}\left[W_{hh}^T\cdot\sigma_h'(z_j)\right], \end{aligned} \tag{15}$$

meaning that

$$\frac{\partial L}{\partial W} = \sum_{n=1}^{N}\sum_{k=1}^{n}\frac{\partial L}{\partial \tilde{y}_n}\frac{\partial \tilde{y}_n}{\partial h_n}\prod_{j=k+1}^{n}\left[W_{hh}^T\cdot\sigma_h'(z_j)\right]\frac{\partial h_k}{\partial W}. \tag{16}$$

Up until now we have considered only one hidden layer. Implementing the stacked RNN is done by essentially creating a hidden state for each 'stack' of RNN. The output of the stacked RNN is computed by feeding the hidden states to each other in succession, starting from the first hidden layer. The $l$-th hidden states in some time window $n$ are given by (cf. [**?** ])

$$h_n^l = \begin{cases} \sigma^{(h)}\left(W_{hx}^1X_n + W_{hh}^1h_{n-1}^1 + b_h^1\right), & l = 1, \\ \sigma^{(h)}\left(W_{hx}^lh_n^{l-1} + W_{hh}^lh_{n-1}^l + b_h^l\right), & l \geq 2, \end{cases} \tag{17}$$

and the output of the stacked RNN in time window $n$ as

$$\tilde{y}_n = \sigma^{(\text{out})}\left(W_{yh}h_n^L + b_y\right), \tag{18}$$

with $L$ being the total amount of hidden layers. Here, the dimensions are $W_{hx}^l \in \mathbb{R}^{d_l\times d_{l-1}}$, $W_{hh}^l \in \mathbb{R}^{d_l\times d_l}$, with $d_l$ being the dimension of the $l$-th hidden state, $l_0$ the dimension of the input and $l^L$ the dimension of the output. The BTT algorithm for a stacked RNN takes on the same form, except that we now have $L$ hidden states.

To update the weights, we define the errors:

$$\delta_{hh}^{k,l} \equiv \frac{\partial h_k^l}{\partial W_{hh}^l}, \ \delta_{hx}^{k,l} \equiv \frac{\partial h_k^l}{\partial W_{hx}^l}. \tag{19}$$

We can then summarize how the hidden-layer weights and biases are updated (cf. [**?** ]):

$$\frac{\partial L}{\partial W^l_{hh}} = \sum_{n=1}^{N}\sum_{k=1}^{n}\frac{\partial L}{\partial h^l_n}\left[\prod_{j=k+1}^{n}\sigma'_h(z^l_j)W^l_{hh}\right]\delta^{k,l}_{hh}, \quad (20a)$$

$$\frac{\partial L}{\partial W^l_{hx}} = \sum_{n=1}^{N}\sum_{k=1}^{n}\frac{\partial L}{\partial h^l_n}\left[\prod_{j=k+1}^{n}\sigma'_h(z^l_j)W^l_{hh}\right]\delta^{k,l}_{hx}, \quad (20b)$$

$$\frac{\partial L}{\partial b^l_h} = \sum_{n=1}^{N}\sum_{k=1}^{n}\frac{\partial L}{\partial h^l_n}\left[\prod_{j=k+1}^{n}\sigma'_h(z^l_j)W^l_{hh}\right]\sigma'_h(z^l_k), \quad (20c)$$

since $\partial h^l_n/\partial b^l_h = \sigma'_h(z^l_k)$. For the output layer, $\partial h_n/\partial W_{yh}$ is only non zero of $n = N$, hence

$$\frac{\partial L}{\partial W_{yh}} = \frac{\partial L}{\partial \tilde{y}_n}\frac{\partial z_N}{\partial h_n}\sigma'_{\text{out}}h_N, \quad (21a)$$

$$\frac{\partial L}{\partial b_y} = \frac{\partial L}{\partial \tilde{y}_n}\frac{\partial z_N}{\partial h_n}\sigma'_{\text{out}}. \quad (21b)$$

The $\delta$-errors can be computed recursively through time. This can be seen by writing them out explicitly. For $\delta^{k,l}_{hh}$ we have:

$$\begin{aligned}
\delta^{1,l}_{hh} &= 0,\\
\delta^{2,l}_{hh} &= \sigma'_h(z^l_2)h^l_1,\\
\delta^{3,l}_{hh} &= \sigma'_h(z^l_3)\left(h^l_2 + W_{hh}\sigma'_h(z^l_2)h^l_1\right), \quad (22)\\
&\vdots\\
\delta^{k,l}_{hh} &= \sigma'_h(z^l_k)\left(h^l_{k-1} + W_{hh}\delta^{k-1,l}_{hh}\right). \quad (23)
\end{aligned}$$

and for $\delta^{k,l}_{hx}$,

$$\begin{aligned}
\delta^{k,1}_{hx} &= \sigma'(z^1_k)X_k,\\
\delta^{k,2}_{hx} &= \sigma'_h(z^2_k)\left(h^1_k + W^2_{hh}\sigma'(z^2_{k-1})X_k\right),\\
&\vdots\\
\delta^{k,l}_{hx} &= \sigma'_h(z^l_k)\left(h^{l-1}_k + W^l_{hh}\delta^{k-1,l}_{hx}\right). \quad (24)
\end{aligned}$$

The dependency for each error term $\delta^{k,l}$ on 'past' error terms, together with the general temporal dependency seen already in (14), leads to a much greater computation time, compared to that of FFNN. Every gradient computation needs an additional propagation through all time-windows. This can lead to gradients blowing up due to only (relatively) minor errors. However, there are multiple ways of resolving this issue. Perhaps the most obvious one is to simply truncate the amount of terms in the algorithm, commonly referred to as 'truncated backpropagation through time' (see e.g. [**?** ]). Apart from that it is an actual simplification, it has the immediate consequence of ignoring long-term dependencies of the data, which in some cases is just the type of information you do not want your model to train on.

### 2.3. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are specialized deep learning architectures used to process data with a spatial or temporal structure, such as images, time series, or signals. The mathematical operation of convolution is the main feature of CNNs, which is inspired by signal processing and serves as a powerful tool for feature extraction.

In its most basic form, convolution for a 1D input signal $f[i]$ and a filter kernel $g[n]$ is defined as:

$$(f * g)[i] \equiv \sum_n f[n]g[i-n]. \quad (25)$$

In the case of 2D data, the convolution is:

$$(f * g)[i,j] \equiv \sum_m\sum_n f[i-m, j-n]g[m,n], \quad (26)$$

where in the case of e.g. images, $f[i,j]$ represents the input pixel intensity at position $(i,j)$, and $g[m,n]$ is the filter kernel. In the context of NNs, convolution can be thought of as a sliding window operation where the filter $g$ is moved across the input $f$. At each position, the filter and the corresponding portion of the input are element-wise multiplied and summed, producing a new value in the output matrix. This process effectively detects patterns encoded in the filter, such as edges, textures, or other localized features.

Convolutions are closely related to concepts in functional analysis and signal processing, where they are used to study how one function modifies another function. In the frequency domain, convolution corresponds to multiplication, which allows the filter to emphasize or suppress specific frequency components of the input. Take an image as an example: high-frequency components correspond to sharp changes like edges, while low-frequency components correspond to smoother regions. By designing filters that target specific frequencies, CNNs can effectively isolate desired features while reducing noise.

Noise often manifests as high-frequency components or irregular patterns in data. The convolution operation is inherently local, meaning it focuses on nearby relationships between data points. Filters can be trained or designed to ignore small-scale irregularities while preserving dominant features, such as edges or repeating structures. In a CNN, the ability to learn filters automatically during training allows the network to focus on the most relevant features of the data, adapting to the specifics of the given task.

The layers in a CNN consists of the following types:

1. Input Layer: This is common to al NNs.
2. Convolutional Layers: These apply learned filters to detect patterns in the input. Early layers capture simple structures (edges, textures), while deeper layers capture complex and abstract features. The outputs are then passed through a given activation function.

3. Pooling Layers: Pooling, such as max-pooling, reduces dimensionality by collapsing local regions:

$$\text{MaxPooling}(x) = \max(x_{\text{window}}),$$

where $x_{\text{window}}$ is a small subset of the input.

4. Flatten Layer: Converts the multidimensional output of convolutional and pooling layers into a one-dimensional vector. This is necessary to connect the extracted features to the fully connected layers for final decision-making.

5. Dense Layers: These process the high-level features extracted by the convolutional and pooling layers to make predictions, such as classifying an image or detecting a signal.

In the context of gravitational wave (GW) detection, CNNs are particularly effective for signal de-noising and feature extraction:

- Convolutional layers can isolate localized patterns in the time-frequency representation of GW data, such as the characteristic "chirp" signal of a binary merger.
- By stacking multiple layers, the network can progressively refine its ability to distinguish noise from signal, leveraging learned patterns specific to GWs.
- Pooling layers enhance robustness by ignoring small-scale noise variations while retaining the overall structure of the signal.

Through this hierarchical process, CNNs filter out most of the background noise, leaving the gravitational wave signal predominantly intact. This capability, coupled with the network's adaptability, makes CNNs a cornerstone for tasks involving noisy, high-dimensional data.

## 2.4. Additional Techniques

### 2.4.1. Regularization & Loss Function

We used $L^2$ regularization for all our results. We refer to [**?** ] for an introduction. The loss function considered for all our results was the weighted binary-cross entropy with this additional $L^2$ regularization:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[S_{\text{CW}}\, y_i \ln(\hat{y}_i) + (1-y_i)\ln(1-\hat{y}_i)\right] + \lambda\sum_{i=1}^{N} w_i^2 \tag{27}$$

where $S_{\text{CM}}$ is the signal class weight and the noise class weight has been normalized to unity.

### 2.4.2. Gradient Clipping

A common method for dealing with exploding gradients, is the method of gradient clipping (see e.g. [**?** ]). This method prevents checks whether the magnitude of the gradient is moving past a certain threshold. If this is true it truncates the current gradient. This can be summarized as:

$$\nabla L \to \frac{\epsilon}{||\nabla L||}\nabla L \quad \text{if} \quad ||\nabla L|| > \epsilon. \tag{28}$$

### 2.4.3. Early Stopping

We opted into using early stopping for the majority of our parameter scans. Given that our program did not improve in a certain number of epochs (between 15% and 40% depending inversely on the number of epochs), and the loss was sufficiently high, we ended the learning process early. If the loss was sufficiently good, we however still kept on going, in the hopes that we would be able to improve the model further. This is mostly done to not waste time on parameter combinations which perform poorly.

### 2.4.4. Dynamic Class Weights

To handle the large class imbalance between the noise and gravitational events, we introduced a dynamic class weight adjustment based on a hyperparameter $\phi$. Given the label array $\mathbf{l}$, if we define $l_i = 1$ when there is a signal, and $l_i = 0$ when there is only noise and normalize the noise class weight to 1, then a simple way to compute a linear dynamical weights for the signal class is:

$$S_{\text{CW}} = \left[\phi - (\phi-1)\frac{\mathcal{E}}{E}\right] \times \left[\frac{\text{len}(\mathbf{l})}{\sum_i l_i} - 1\right] \tag{29}$$

where $S_{\text{CW}}$ refers to the signal class weight (GWs in our case), $\mathcal{E}$ is the current epoch and $E$ is the total number of epochs. The right bracket is just the factor which would give you balanced class weights, i.e. the program is equally punished for only predicting either of the classes. The factor in the left bracket is then just a linearly decreasing factor going from $\phi$ to 1 as $\mathcal{E} \to E$. This essentially gives an early boost to the signal class, and gradually converges to a balanced class weight system where the NN would perform equally poorly from predicting all in either of the classes. A simple example is shown in Fig. 1. Note that the noise-signal label ratio is *not* SNR, one refers to the time axis in the case of GW (i.e. the percentage of time there is a signal), whilst the latter refers to the strain-to-noise ratio (i.e. roughly how visible the signal is at any given time). Dynamical class labels are important in our case since GW events are so rare. Due to this the program may simply decide that it is best not to predict the signal class at all, as it is much more likely

to be punished for ever attempting to 'guess' GWs. Thus dynamical weights are a way to reduce the effects of this particular problem.

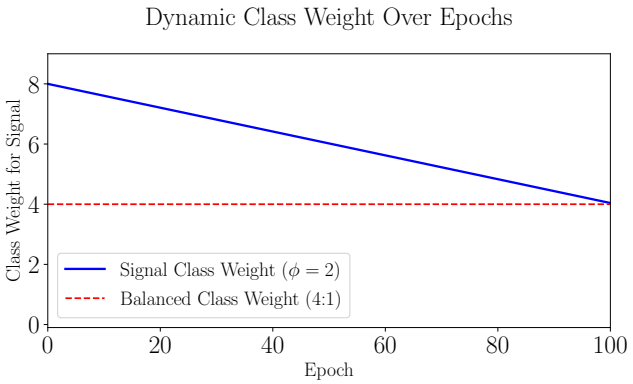


Fig. 1: Dynamic class weight shown for a simple example with $\phi = 2$ where we have a 4:1 noise-signal label ratio over 100 epochs.

#### 2.4.5. Wavelet Transforms

Wavelet transforms are mathematical tools used to analyze signals by decomposing them into time-frequency components. A wavelet transform applies scaled and shifted versions of a so-called *mother wavelet* function, $\psi(t)$, to a signal. The continuous wavelet transform (CWT) of a signal $f(t)$ is defined as:

$$W(a,b) = \int_{-\infty}^{\infty} f(t)\psi_{a,b}(t)\,dt,$$

where $\psi_{a,b}(t)$ is the scaled and shifted wavelet:

$$\psi_{i,j}(t) = \frac{1}{\sqrt{2^j}}\psi\left(\frac{t-k2^j}{2^j}\right),$$

with $a > 0$ representing the frequency and $b$ the time shift.

In the discrete wavelet transform (DWT), the scale and translation are discretized. The DWT is computed as:

$$W[j,k] = \sum_n f[n]\psi_{j,k}[n],$$

making it computationally efficient for practical applications like feature extraction or de-noising.

Unlike traditional Fourier transforms, which decompose a signal into sine and cosine waves of different frequencies, wavelets allow for both time and frequency localization. This makes them particularly useful for analyzing non-stationary signals, such as gravitational waves or audio signals, where the frequency content changes over time. In the context of neural networks, wavelet transforms are often used as a pre-processing step to reduce noise, extract meaningful features, or compress data, thereby potentially improving the performance. This is what we will be using for our CNN.

## 3. IMPLEMENTATION

We implemented a program which downloads the data from [**?** ], and when given the duration of the GW signal of interest, automatically labels the data accordingly. Our definition for 'duration' of an event is rather arbitrary. What we did is simply look at a relevant research paper (found for each event at [**?** ]) which contains plots of the treated data and made an educated guess for where it would be reasonable to detect it. Further we made a class `GWSignalGenerator`, which adds a synthetic GW signal to any arbitrary sequence, and adds corresponding labels to the dataset. This class has a lot of options, such as spin factors and amplitude factors for how much the signal should spike up, generating random events, etc. and was meant to simulate what GW event strain data looks like. Originally this program was meant to be our test runs before working with the real deal. However due to the complicated nature of detecting GWs, this ended up being what the NNs worked with instead. We explain the reasoning for this in the next section.

### 3.1. The Neural Networks

We have implemented three classes, `RNN`, `KerasRNN` and `KerasCNN`, all of which inherits from a parent class called `NeuralNetwork`, containing methods handling saving data (`save_data`), scaling and splitting data (`store_train_test_from_data`), and dividing the data into sequences (`prepare_sequences_RNN`). The class `RNN` is our own implementation of the stacked RNN architecture described in section 2 2.2, using the equations in (20) and (21) when training. On the other hand, the class `KerasRNN`, uses `keras`'s RNN [**?** ]. Common to both RNN classes are their public methods—`train`, `predict`, `cross_validate` and `calculate_loss`—which function as their names suggest. Both classes are initialized with key parameters, including a hidden-layer activation function, an output-layer activation function, a scaler type, and an optimizer, which can be set to plain gradient descent, Adagrad, RMSprop, or Adam. During training, the models require the input data, the number of epochs, batch size, window size, and optionally a clip value for gradient clipping, as implemented in (28).

The `KerasCNN` class is less generalized than the two RNN classes, as it defaults to five layers in the order: convolution, max-pooling, convolution, flatten and a dense layer. When training on multiple data sets, we have used the `train_multiple_datas` method of `KerasCNN`, and we calculate the loss and accuracy metrics using Keras's

build in methods. For details on the specific file structure, we refer to the README file at project 3.

### 3.2. Default Parameters and Parameter Scans

There are many parameters contribute to the training of a neural network. There are some parameters which we defaulted consistently to certain values throughout, while for others we performed a parameter scan.

For our RNNs, we set the window size as ('//' denoting floor division) $N_W = N//100$, batch size as $B = 128$, where $N$ is the total time steps. We consistently chose the hidden layers to be of shape 5, 10, 2, in that order.

For the CNN, we used 16 filters for the convolutional layers and a pool size of $(2, 2)$ for the max-pooling layer. Across all networks, we selected the tanh function as the activation for the hidden layers due to its advantageous properties, such as centering data around zero to mitigate vanishing gradient issues. For the output layer, we applied the sigmoid activation function, which is particularly suitable for binary classification problems as it maps inputs to the range $[0, 1]$, effectively representing class probabilities.

Although we have implemented the possibility of using different optimizers, our results further down is created using exclusively Adam as our optimizer. This is partly because we did not have time to properly investigate the other optimizers in this project, but also because Adam showed more promise early on.

While these settings were fixed, certain key hyperparameters were systematically varied. Specifically, we chose to vary the following parameters:

- $E \in \{10, 25, 50, 100\}$
- $\phi \in \{1.0, 1.1, ..., 1.5\}$
- $\lambda \in \{10^{-10}, 10^{-9}, ..., 10^{-1}\}$
- $\eta \in \{10^{-4}, 5 \times 10^{-4}, ..., 10^{-1}, 5 \times 10^{-1}\}$

where $E$ is the number of epochs, $\phi$ the initial boost (29), $\lambda$ the $L^2$-regularization parameter (27) and $\eta$ the learning rate (11). Due to the program being incredibly slow, we only ended up using the Adam optimizer, and only managed to finish half the parameter scan with $N = 100$. Similarly, due to spending much time implementing our custom RNN, we only had time to perform a few parameter combinations. The performance of these were however so poor – even with an extremely high SNR on the synthetic signal – that they are not worth mentioning. This is likely due to mistakes in the implementation which we did not manage to fix.

Due to a massive lack of performance of the RNN's, we decided to include an implementation of keras' CNN's as well. This was done by performing a wavelet transform on the synthetic signals using the library `pywt`. The coefficients of these wavelet transforms were then fed into the CNN. The CNN consists of

- Input layer size 2.
- 2D convolutional layer with filter size $(3, 3)$ and tanh activation function.
- MaxPooling layer with pool size $(2, 2)$.
- 2D convolutional layer with filter size $(3, 3)$ and tanh activation function.
- Flattening layer
- Dense output layer of size 1 with sigmoid as its activation function.

We then ran a small parameter scan due to time constraints where we included the following parameters:

- $E \in \{10, 25, 50\}$
- $\phi \in \{1.0, 1.1, ..., 1.5\}$
- $\lambda \in \{10^{-10}, ..., 10^{-6}\}$
- $\eta \in \{5 \times 10^{-3}, 10^{-3}, 5 \times 10^{-2}, 10^{-2}\}$

with batch size 128, SNR = 5, $n_{\text{filters}} = 16$ and 5000 timesteps. As before, we did not do this with the actual GW data, and instead opted into using our own signal generator class.

## 4. CHALLENGES

### 4.1. Time Constraints

One of the primary challenges we faced during this project was the significant time required to run the models. The parameter scans for the Keras-based RNN models with $N = 100$ were particularly time-consuming, and, as a result, we were only able to complete approximately half of the planned configurations. This limitation constrained our ability to fully explore the hyperparameter space and systematically assess model performance.

The situation was even more pronounced for our custom implementation of the (stacked) RNN, which naturally demanded far greater computational resources and runtime. Given the inherently more complex structure and optimization challenges associated with our custom RNN, the execution time increased substantially. Consequently, we were only able to test a limited number of parameter combinations for the custom model.

### 4.2. Difficulties with Gravitational Data

As we have mentioned prior, we did not end up using either of the RNN's nor the CNN on actual GW data, or at least not long enough to gather any meaningful results. The reason for this is simple; gravitational wave detection is very difficult!

Our initial motivation for working with GW data stemmed from the belief that modified GW datasets with higher signal-to-noise ratios (SNR) would be available. Additionally, we assumed that, at the very least, we could compare how different models performed relative to one another, even if the results were poor. That being said, we do not exclude the possibility that we were simply a bit too optimistic.

As a result, we spent a considerable amount of time evaluating the performance of our NNs on datasets that were far too complex for effective training, as we saw close to no convergent behavior. In the following, we aim to highlight the inherent challenges of training NNs on GW data to provide context for our efforts and results.

One of the fundamental difficulties lies in the nature of the GW signal itself. GW events are incredibly short in duration, often lasting only a fraction of a second, and are buried deep within overwhelming noise. This makes it extremely difficult to isolate the actual signal, let alone train a NN to identify it. Consider for example Fig. 2 which shows the raw strain data measured by the H1 LIGO detector surrounding the event GW170104 acquired from [**?** ]. The small partition in red is where the GW signal was detected by both LIGO detectors and Virgo, whilst the rest is simply noise. This event has a roughly average peak SNR $\sim 13$ (after treatment) and a rather common duration of roughly 0.04 s, so this would be around what would be expected of a NN to be able to detect. Whilst the strain here is of the order $10^{-18}$, after cleaning the data, [**?** ] shows that the actual strain stemming from the GW event peaks around $10^{-21}$. Thus the unfiltered SNR of the raw data is $\lesssim 10^{-3}$. Clearly it is unreasonable to give this to a NN and expect it to learn anything, and this is even with data which is over a relatively short timespan of under 20 seconds. Note that even if we picked the GW with highest (treated) SNR, this is only $\sim 3$ times higher than this particular example. Thus even in this most ideal case, we still have a raw data SNR which is at best $\sim 10^{-2}$.

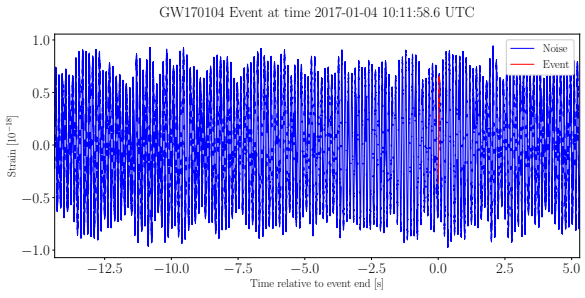


Fig. 2: Raw strain data for GW170104 v2 from the H1 LIGO detector [**?** ]. Blue corresponds to noise whilst red is where the GW event has been detected.

What academics do is they remove all the noise from known sources which occur at this time, e.g. seismic activity, local disturbances and afterwards perform a Bayesian analysis of the data to determine the properties of the source [**?** ]. Finally, they reconstruct the signal using the found properties and check against the original data. For this particular event, they arrived at the signal shown in Fig. 3 which has a background noise $\sim 10^4$ times smaller. This is clearly much easier to work with than the raw data, and a much more realistic task for a NN to be able to detect. However at this point all the work has already been done, and requires far more work than we can manage given our lack of expertise and time for this project.

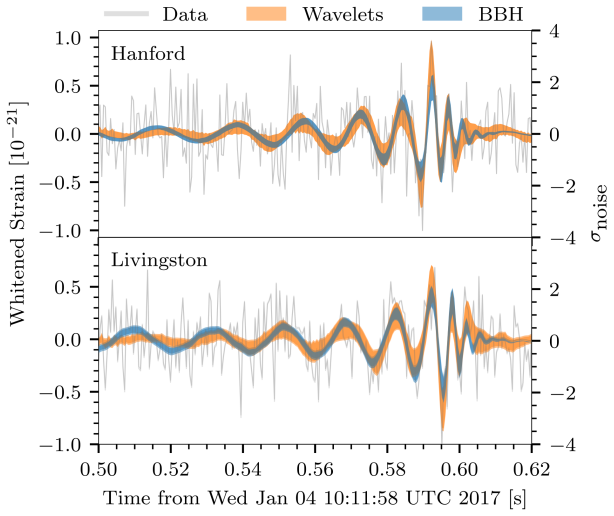


Fig. 3: Treated data and reconstructed signal for GW170104. This figure has been taken from [**?** ].

## 5. RESULTS & DISCUSSION

We begin by considering `tensorflow.keras`'s `SimpleRNN` on a set of synthetic GW signals for various combinations of the learning rate $\eta$ and $L^2$ regularization parameter $\lambda$ as shown in Fig. 4. Here we have used 50 epochs for both the figures, where the upper plot uses a dynamical class weights with $\phi = 1.4$ and the lower plot uses static class weights, i.e. $\phi = 1$. The difference for high $\lambda$ and suboptimal $\eta$ can be seen to be relatively small between the two. In both the cases, large $\lambda$ corresponds to underfitting the model with the high suppression for the weights, and too high (low) value for the initial learning rate causes the model to overshoot (undershoot) the desired minima. For the combinations of $\eta$ and $\lambda$ where the model performs well, we see that there is a clear distinction between the two figures. The model performs much better when it initially starts with higher class weights for the signal class.

The particular example with $\eta = 0.01$, $\lambda = 10^{-8}$, $\phi = 1.4$ with 50 epochs for SNR $\sim 100$ is given in Fig. 5. Here we have shaded the region in red where the RNN predicted a signal in two neighboring points. The blue line corresponds to the signal itself, whilst the green line

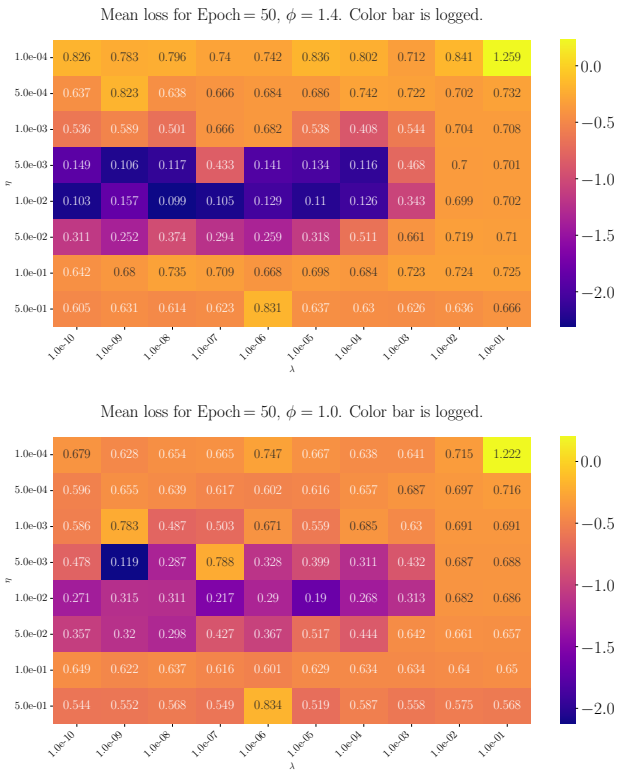


Fig. 4: Average test loss for the 5-fold cross validation using `tensorflow.keras`' `SimpleRNN` for various parameter combinations with 50 epochs. These plots have been chosen to show the effect of an ideal initial boost, $\phi = 1.4$, compared to the simpler case $\phi = 1$ which corresponds to static, but still balanced, class weights. Note that we have taken the logarithm of the colorbar to get more sensitivity at lower loss.

is the labels. Here we have much more than just ideal conditions with a rather large SNR, so obviously the task is not nearly as difficult as actual GW events.

We then tried to up the difficulty, lowering the SNR to $\sim$ 15, which is roughly the range of a treated GW event. This however lead to results so poor that guessing randomly would give a better loss.

For the CNN however we could lower the SNR to as low values as $X$ whilst still having $\alpha$

## 6. CONCLUSION

As mentioned prior, actual gravitational wave detection is very difficult, and requires advanced techniques such as matched filtering and adaptive noise subtraction. With our limited time and resources we were not able to properly clean the data to perform such an analysis. Thus we simply opted into creating our own signal in a very ideal situation.

The RNN's both performed appreciably worse than the

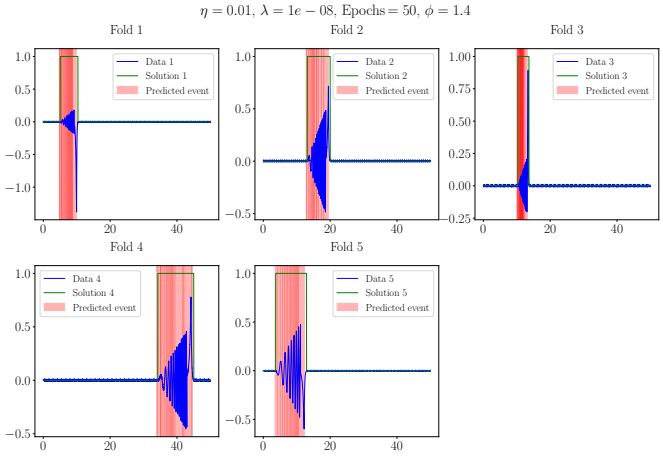


Fig. 5: Keras RNN with $\eta = 0.01$, $\lambda = 10^{-8}$, $\phi = 1.4$ with 50 epochs for SNR $\sim$ 100. The blue line corresponds to the generated data, the green line is the labels whilst the shaded red regions are where the RNN predicted a signal. The $x$- and $y$-axes correspond to the time and strain respectively.

CNN, even if we spent much more time making the prior work. Whilst our custom RNN is likely a fault of our own implementation, keras' CNN clearly outperformed its own RNN across all metrics, even achieving similar performance with an SNR which is an order of magnitude lower.

In retrospect, GW detection was not ideal for this project as it was simply too difficult given our limited time and expertise on this topic. Once we saw the lack of performance from the RNN's we should have immediately attempted different methods. Whilst we did this in the end, it was however too late to do a proper analysis to find optimal parameters for the CNN.