Project 3

Edvard B. Rørnes* and Isak O. Rukan†
Institute of Physics, University of Oslo,
0371 Oslo, Norway
(Dated: December 12, 2024)

Abstracting very cool

CONTENTS

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, traveling at the speed of light, and cause minute distortions in spacetime itself, which can be measured by highly sensitive instruments.

One of the most advanced experiments 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 complex task.

Machine learning, particularly neural networks (NNs), has become a powerful tool in this context. By training a neural network 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 [**?** **?**]. Neural networks can be trained to recognize patterns in the data that correspond to the characteristic signatures of gravitational waves, 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. The performance on this untreated data is quite poor due to the untreated data having a signal to noise ratio (SNR) which is very large. 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 allows 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. The code in (cite Github) however does contain a program which automatically labels GW files granted that the user knows the duration of the GW signal, which may be of use for future work.

2. THEORY

2.1. Gravitational Waves

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

2.1.1. Linearized Gravity

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

* e.b.rornes@fys.uio.no
† Insert Email

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

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

Then applying this to the Einstein field equations (EFE),

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

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. 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 travelling along the z-axis. The details for how this is derived is spelled out in [**?**]. h_+ and $h_\times$ correspond to the "plus" and "cross" polarizations which determine how the gravitational waves stretch and compress the spacetime it passes through. The stretching from these are what is measured by observatories such as LIGO and Virgo and the displacements from these are referred to as 'strain'.

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 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. This is done with the introduction of a hidden state h_t, where the subscript denotes at time t. The network can be summarized 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, σ_h and σ_{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}:

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

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. all previous times.

2.2.2. General Algorithm

Consider some general data output y, of shape (N, p_{out}) and some data input X, of shape (N, p_{in}), where N corresponds to the total amount of time points, and p_{out}, p_{in} the dimension of the output and input, respectively. 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.

2.2.3. Backpropagation Through Time

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 η, the standard update rule for the weights and biases is given by:

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

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 FFNN, computing the gradient of L with respect to the weights leads to a somewhat more complicated expression. Consider now 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 [**?**]:

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

Notice here that,

$$\begin{aligned} \frac{\partial h_n}{\partial h_k} &= \frac{\partial h_n}{\partial h_{n-1}}\frac{\partial h_{n-1}}{\partial h_{n-1}}\cdots\frac{\partial h_{k+1}}{\partial h_k} \\ &= \left(\sigma_h'(z_n)W_{hh}\right)\left(\sigma_h'(z_{n-1})W_{hh}\right)\ldots\left(\sigma_h'(z_{k+1})W_{hh}\right) \\ &= \prod_{j=k+1}^{n}\sigma_h'(z_j)W_{hh}. \end{aligned} \tag{15}$$

Define now the errors,

$$\delta_{hh}^k \equiv \frac{\partial h_k}{\partial W_{hh}},\ \delta_{hx}^k \equiv \frac{\partial h_k}{\partial W_{hx}}. \tag{16}$$

Computing these errors leads to the recursive formula,

$$\begin{aligned} \delta_{hh}^1 &= 0, \\ \delta_{hh}^2 &= \sigma_h'(z_2)h_1, \\ \delta_{hh}^3 &= \sigma_h'(z_3)\left(h_2 + W_{hh}\sigma_h'(z_2)h_1\right), \end{aligned} \tag{17}$$

$$\ldots,$$

$$\delta_{hh}^k = \sigma_h'(z_k)\left(h_{k-1} + W_{hh}\delta_{hh}^{k-1}\right), \tag{18}$$

with a similar behavior for δ_{hx}^{N-n}. On the other hand, the last product in the gradient for the (hidden) biases, $\partial h_{N-n}/\partial b_h$, do not lead to a recursive formula (c.f. [**?**]). Hence, for the hidden weights and biases, we have the gradients

$$\frac{\partial L}{\partial W_{hh}} = \sum_{n=1}^{N}\sum_{k=1}^{n}\frac{\partial L}{\partial h_n}\left[\prod_{j=k+1}^{n}\sigma_h'(z_j)W_{hh}\right]\delta_{hh}^k, \tag{19a}$$

$$\frac{\partial L}{\partial W_{hx}} = \sum_{n=1}^{N}\sum_{k=1}^{n}\frac{\partial L}{\partial h_n}\left[\prod_{j=k+1}^{n}\sigma_h'(z_j)W_{hh}\right]\delta_{hx}^k, \tag{19b}$$

$$\frac{\partial L}{\partial b_h} = \sum_{n=1}^{N}\sum_{k=1}^{n}\frac{\partial L}{\partial h_n}\left[\prod_{j=k+1}^{n}\sigma_h'(z_j)W_{hh}\right]\sigma_h'(z_k). \tag{19c}$$

For the output layer, $\partial h_n/\partial W_{yh}$ is only non zero of $n = N$, hence

$$\frac{\partial L}{\partial W_{yh}} = \frac{\partial L}{\partial \tilde{y}_n}\frac{\partial z_N}{\partial h_n}\sigma_{\text{out}}' h_N, \tag{20a}$$

$$\frac{\partial L}{\partial b_y} = \frac{\partial L}{\partial \tilde{y}_n}\frac{\partial z_N}{\partial h_n}\sigma_{\text{out}}'. \tag{20b}$$

$$\frac{\partial L}{\partial W_{yh}} = \frac{\partial L}{\partial h_N}\left[\prod_{j=k+1}^{n}\sigma_h'(z_j)W_{hh}\right]\sigma_{\text{out}}'(z_N)h_N, \tag{21}$$

$$\frac{\partial L}{\partial b_h} = \frac{\partial L}{\partial h_N}\left[\prod_{j=k+1}^{n}\sigma_h'(z_j)W_{hh}\right]\sigma_{\text{out}}'(z_N). \tag{22}$$

The dependency for each error term δ^{N-n} on 'past' error terms 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.

Implementing the stacked RNN is then 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 hidden states in some time window n are given by

$$h_n^l = \begin{cases} \sigma^{(h)}\left(W_{hx}^1 X_n + W_{hh}^1 h_{n-1}^1 + b_h^1\right), & l = 1, \\ \sigma^{(h)}\left(W_{hx}^l h_n^{l-1} + W_{hh}^l h_{n-1}^l + b_h^l\right), & l \geq 2, \end{cases} \tag{23}$$

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

$$\tilde{y}_n = \sigma^{(\text{out})}\left(W_{yh} h_n^L + b_y\right). \tag{24}$$

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.

2.2.4. 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:

$$\nabla L \to \frac{\epsilon}{||\nabla L||} \nabla L \quad \text{if} \quad ||\nabla L|| > \epsilon. \tag{25}$$

2.3. Additional Techniques

2.3.1. 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 on 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.

2.3.2. Dynamic Class Weights

To handle the large class imbalance between the noise and gravitational events, we introduced a dynamic class weight adjustment based on the hyperparameter ϕ. If we define $l_i = 1$ when there is a signal, and $l_i = 0$ when there is only noise and normalize the class weight for the noise to 1. Then a simple way to compute a linear dynamical weights for the signal class is:

$$S_{\text{CW}} = \left[\phi - (\phi - 1)\frac{\text{epoch}}{N}\right] \times \left[\frac{\text{len}(l)}{\sum_i l_i} - 1\right] \tag{26}$$

where S_{CW} refers to the signal class weight (GWs in our case), epoch is the current epoch, N is the total number of epochs and l is the labels. This essentially gives an early boost to the signal class, causing the program to be punished for only guessing noise, and gradually converges to a balanced class weight system, where the NN would perform equally poorly from guessing all in either of the classes. A simple example is show in Fig. 1. This is important in our case since GW events are so rare. Due to this the program may simply decide that it is best not to guess it at all as it is much more likely to be punished for guessing GWs.

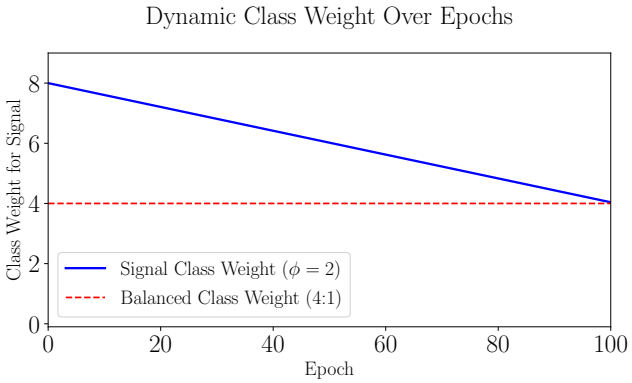

Fig. 1. Dynamic class weight shown for a simple example with $\phi = 2$ where we have a 4:1 noise-signal ratio over 100 epochs.

3. IMPLEMENTATION

We implemented a program which downloads the data from gwosc, and when given the duration of the GW signal of interest, automatically labels the data accordingly. Further we made a class `GWSignalGenerator`, which adds a synthetic GW signal to any arbitrary sequence, and adds corresponding labels to the dataset. This was originally 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 our NNs where working with instead.

The structure of the programs are...

We then performed large parameter sca

4. DISCUSSION

5. CONCLUSION

As mentioned prior, actual gravitational wave detection is very difficult, and requires advanced techniques such as matched filtering and adaptive noise subtraction. In the limited time and resources we were not able to properly clean the data. Thus we opted into...