Leibniz Universität Hannover

and

Max Planck Institute for Gravitational Physics (Albert Einstein Institute)

Master thesis

Analysis of Gravitational-Wave Signals from Binary Neutron Star Mergers Using Machine Learning

Marlin Benedikt Schäfer

Supervisors: Dr. Frank Ohme and Dr. Alexander Harvey Nitz

June 4, 2019

This page is intentionally left blank. LÖSCHEN!!! Damit Eigenständigkeitserklärung nicht auf Rückseite gedruckt ist.

Ort, Datum	Marlin Benedikt Schäfer

This page is intentionally left blank. LÖSCHEN!!! Damit Inhaltsverzeichnis nicht auf Rückseite gedruckt ist.

Abstract

Put the abstract here

This page is intentionally left blank. LÖSCHEN!!! Damit Inhaltsverzeichnis nicht auf Rückseite gedruckt ist.

Contents

This page is intentionally left blank. LÖSCHEN!!! Damit erste Seite nicht auf Rückseite gedruckt ist.

3 Neural networks

Explain the use for this section.
Neural networks are machine learning algorithms inspired by research on the structure and inner workings of brains. [Insert quote (Rosenblatt?)] Though in the beginning NN were not used in computer sciences due to computational limitations [Citation] they are now a major source of innovation across multiple disciplines. Their capability of pattern recognition and classification has already been successfully applied to a wide range of problems not only in commercial applications but also many scientific fields. [Quote a few scientific usecases here. Of course using the one for gw but also other disciplines.]
Major use cases in the realm of gravitational wave analysis have been classification of glitches in the strain data of GW-detectors [Citation] and classification of strain data containing a GW versus pure noise [Citation]. A few more notable examples include [list of citations].
In this section the basic principles of NN will be introduced and notation will be set. The concept of backpropagation will be introduced and extended to a special and for this work important kind of NN. (maybe use the term "convolution" here already?) It will be shown that learning in NN is simply a mathematical minimization of errors that can largely be understood analytically.

3.1 Neurons, layers and networks

What is the general concept of a neural network? How does it work? How does back-propagation work? How can one replicate logic gates? (cite online book)
The basic building block of a NN is - as the name suggests - a *neuron*. This neuron is a function mapping inputs to a single output. In the early days this output was always either 1 or 0 [Citation], whereas nowadays it is usually some real number.
In general there are two different kinds of inputs. Those that are specific to the neuron itself and those that the neuron receives as an outside stimulus. We write the neuron as

$$n : \mathbb{R}^k \times \mathbb{R} \times \mathbb{R}^k \to \mathbb{R}; \quad (\vec{w}, b, \vec{x}) \mapsto n(\vec{w}, b, \vec{x}) := a(\vec{w} \cdot \vec{x} + b), \tag{3.1}$$

where $\vec{w}$ are called weights, b is a bias value, $\vec{x}$ is the outside stimulus and a is a function known as the *activation function*(change this to not be emphasized if it is not used for the first time here). A usual depiction of a neuron and its structure is shown in Figure 3.1. The activation function is a scalar function

$$a : \ \mathbb{R} \to \mathbb{R} \tag{3.2}$$

determining the scale of the output of the neuron. To understand the role of each part of the neuron, consider the following activation function:

$$a(y) = \begin{cases} 1, & y > 0 \\ 0, & y \leq 0 \end{cases}. \tag{3.3}$$

x_1	x_2	$a(\vec{w} \cdot \vec{x} + b)$
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.1: Neuron activation with activation function (3.3), weights $\vec{w} = (w_1, w_2)^T = (1, 1)$, bias $b = -1.5$ and inputs $(x_1, x_2) \in \{0, 1\}^2$. Choosing the weights and biases in this way replicates an "and"-gate.

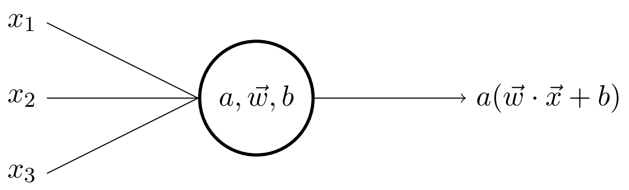


Figure 3.1: Depiction of a neuron with inputs $\vec{x} = (x_1, x_2, x_3)^T$, weights $\vec{w}$, bias b and activation function a.

With this activation function, the neuron will only send out a signal (or "fire") if the input y is greater than 0. Therefore, in order for the neuron to fire, the weighted sum of the inputs $\vec{w} \cdot \vec{x}$ has to be larger than the negative bias b. This means, that the weights and biases control the behavior of the neuron and can be optimized to get a specific output.
The effects of changing the weights makes individual inputs more or less important. The closer a weight w_i is to zero, the less impact the corresponding input value x_i will have. Choosing a negative weight w_i results in the corresponding input x_i being inverted, i.e. the smaller the value of x_i the more likely the neuron is to activate and vice versa.
Changing the bias to a more negative value will result in the neuron having fewer inputs it will fire upon, i.e. the neuron is more difficult to activate. The opposite is true for larger bias values. So increasing it will result in the neuron firing for a larger set of inputs.
As an example consider a neuron with activation function (3.3), weights $\vec{w} = (w_1, w_2)^T = (1, 1)$, bias $b = -1.5$ and inputs $(x_1, x_2) \in \{0, 1\}^2$. Choosing the weights and biases in this way results in the outputs shown in Table 3.1. This goes to show, that neurons can replicate the behavior of an "and"-gate. Other logical gates can be replicated by choosing the weights and biases in a similar fashion (See first section of Appendix A).

Use the introduction of the and-neuron from above to introduce the concept of networks in a familiar way. Having logic gates enables us to build more complex structures, such as full adders and hence we can, in principle, calculate any function a computer can calculate. Only afterwards introduce layers as a way of structuring and formalizing networks.

Since all basic logic gates can be replicated by a neuron, it is a straight forward idea

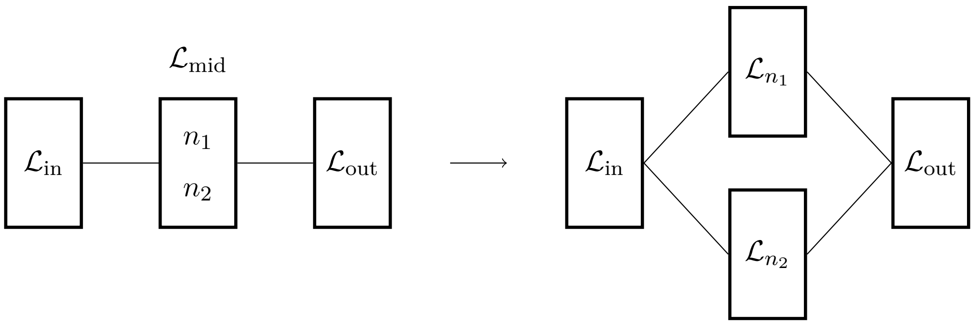

$\mathcal{L}_{\text{mid}}$
$\mathcal{L}_{\text{in}}$
n_1
n_2
$\mathcal{L}_{\text{out}}$
$\mathcal{L}_{\text{in}}$
$\mathcal{L}_{n_1}$
$\mathcal{L}_{n_2}$
$\mathcal{L}_{\text{out}}$

Figure 3.2: Depiction of how a layer ($\mathcal{L}_{\text{mid}}$) consisting of neurons with different activation functions can be split into two separate layers ($\mathcal{L}_{n_1}$ and $\mathcal{L}_{n_2}$).

to connect them into more complicated structures, like a full-adder (see Appendix A). These structures are than called neural networks, as they are a network of neurons. The example of the full-adder demonstrates the principle of a NNs perfectly. It's premise is to connect multiple simple functions, the neurons, to form a network, that can solve tasks the individual building blocks can't.

Since NNs are the main subject of the upcoming part and since this part will be a bit more mathematical, some notation and nomenclature is introduced to structure the networks.
Specifically each network is composed of multiple layers. Each layer consists of one or multiple neurons and each neuron has inputs only from the previous layer. Formally we write

$$\mathcal{L} : \mathbb{R}^{k+l} \times \mathbb{R}^{l} \times \mathbb{R}^{k} \to \mathbb{R}^{l};\ (W, \vec{b}, \vec{x}) \mapsto \mathcal{L}(W, \vec{b}, \vec{x}) := \begin{pmatrix} n_1\Big((W_1)^T, b_1, \vec{x}\Big) \\ \vdots \\ n_l\Big((W_l)^T, b_l, \vec{x}\Big) \end{pmatrix}, \tag{3.4}$$

where n_i is neuron i on that layer and W_i is the i-th row of a $k \times l$-matrix. As a step of formal simplification we will assume that all neurons n_i share the same activation function a. This does not limit the ability of networks that can be written down, since if two neurons have different activation functions, they can be viewed as two different layers connected to the same previous layer. Their output will than be merged afterwards (see Figure 3.2).

With this simplification one can write a layer simply as

$$\mathcal{L}(W, \vec{b}, \vec{x}) = a(W \cdot \vec{x} + \vec{b}), \tag{3.5}$$

where it is understood, that the activation function a acts component wise on the resulting l-dimensional vector.
In this fashion a network consisting of a chain of layers $\mathcal{L}_{\text{in}}$, $\mathcal{L}_{\text{inter}}$, $\mathcal{L}_{\text{out}}$ can be written as

$$a_{\text{out}}\Big(W_{\text{out}} \cdot a_{\text{inter}}\Big(W_{\text{inter}} \cdot a_{\text{in}}\Big(W_{\text{in}} \cdot \vec{x} + \vec{b}_{\text{in}}\Big) + \vec{b}_{\text{inter}}\Big) + \vec{b}_{\text{out}}\Big). \tag{3.6}$$

Hence a network can be understood as a set of nested functions.
An important point with the definitions above is that the layers get their input only from their preceding layer. Especially no loops are allowed, i.e. getting input from some subsequent layer. A network of the first kind is called a *feed forward neural network* (FFN), as for one input each layer gets invoked only once. There are also other architectures called *recurrent neural networks*, which also allow for loops in the networks and work by propagating the activations in discrete time steps. These kinds of networks are in principle closer to the inner workings of the human brain, but in practice show worst performance and are therefore not used or discussed further in this work. [Citations], maybe also mention that RNNs have shown good performance in time series data (which we are working with) but other studies (paper Frank sent around) have shown that TCN also do the job
A FFN in general consists of three different parts called the input-, output- and hidden layer/layers. The role of the input- and output-layers is self explanatory; they are the layers where data is fed into the network or where data is read out. The hidden-layers are called "hidden", as their shape and size, contrary to the other two layers, is not defined by the data. Figure 3.3 shows an example of a simple network with a single hidden layer. In principle there could be any number of hidden layers with different sizes. In this example the input is n-dimensional and the output 2-dimensional. If the input was changed to be (n-1)-dimensional, the same hidden-layer could be used, as its size does not depend on the data or output. Therefore when designing a network architecture, one designs the shape and functionality of the hidden layers.
A NN is called *deep*, if it has multiple hidden layers. [Citation]

3.2 Backpropagation

In subsection 3.1 the basics of a NN where discussed and the example of a network replicating a binary full-adder (see Appendix A) showed the potential of these networks, when the weights and biases are chosen correctly. The example actually proofs that a sufficiently complicated network can - in principle - calculate any function a computer can, as the computer is just a combination of logic gates, especially binary full-adders.
The question therefore is how to choose the weights in a network for it to fit and calculate some function. For the binary full-adder the weights and biases could be chosen quite simply and the network was buildup out of small blocks. A more general approach would however be beneficial. The goal is to design some network and let it learn/optimize the weights and biases to fit some function optimally.
To do this, some known and labeled data is necessary, such that the network can compare its output to some ground truth and adjust its weights and biases to minimize some error function. This way of optimizing the weights and biases is called *training*. To be a bit more specific, the analyzed data in this work is some form of a strain-data time series. The output of this analysis is the SNR of the input strain segment. Therefore the network will receive some strain-data, of which the SNR is known, as input. The network will produce some value from this input data and compare it to what SNR was provided as label. From there it will try to optimize the weights and biases to best fit the function

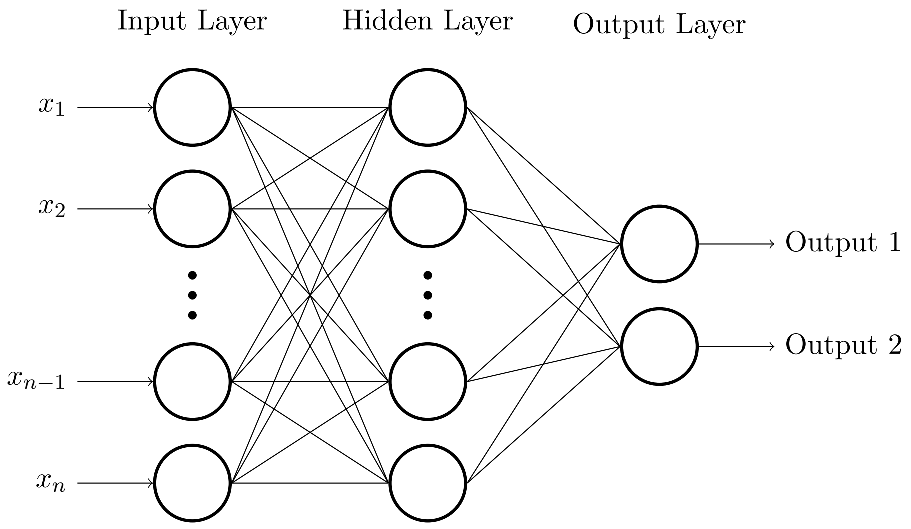


Figure 3.3: A depiction of a simple network with a single input-, hidden- and output-layer. The input-data is a n-dimensional vector $(x_1, \dots, x_n)^T$ and the output is a 2-dimensional vector. In this picture it looks like the hidden layer has the same number of neurons as the input layer. This does not necessarily have to be the case. Lines between two neurons indicate, that the output of the left neuron serves as weighted input for the right one.

that maps strain-data $\rightarrow$ SNR.
This process of optimizing is called backpropagation, as the error propagates from the last to the first layer. The meaning of this will become clearer a little later. Another term, that is used in the context of NNs often, is the *loss function*. The loss function

$$L : \mathbb{R}^l \times \mathbb{R}^l \rightarrow \mathbb{R};\ (y_{\text{net}}, y_{\text{label}}) \mapsto L(y_{\text{net}}, y_{\text{label}}) \tag{3.7}$$

is a function that calculates some form of error between the output of the network and the label value it was given.
When doing a regressive fit, one of the standard error functions is the *mean squared error*, which therefore is the loss function that is mainly used in this work and defined by

$$L : \mathbb{R}^l \times \mathbb{R}^l \rightarrow \mathbb{R};\ (y_{\text{net}}, y_{\text{label}}) \mapsto L(y_{\text{net}}, y_{\text{label}}) := \frac{1}{l} \sum_{i=1}^{l} (y_{\text{net},i} - y_{\text{label},i})^2. \tag{3.8}$$

3.3 Specific layers

Explain that there are not just dense layers.

3.3.1 Dense layer

What is a dense layer, how does it work (can maybe be omitted)

3.3.2 Convolution layer

What are the advantages of convolution layers and why do we use them? Disadvantages?

3.3.3 Inception layer

Explain what it is, how it works. (cite google paper) ONLY IF IT IS REALLY USED IN THE FINAL ARCHITECTURE!

3.3.4 Batch Normalization layer

Explain how batch normalization works and why it is useful. (cite according paper)

3.3.5 Dropout layer

Explain what a dropout layer is, what it does, why it is useful.

3.3.6 Max Pooling layer

Explain what max pooling does and why it is useful, even when it is counter intuitive.

A Full adder as network

To create a full adder from basic neurons, the corresponding logic gates need to be defined. The equivalent neuron for an ”and”-gate was defined in subsection 3.1. There are two more basic neurons which need to be defined. The neuron corresponding to the ”or”-gate, which is given by the same activation function (3.3), weights $\vec{w} = (w_1, w_2)^T = (1, 1)$ and bias $b = -0.5$, and the equivalent neuron for the ”not”-gate, which is given by the activation function (3.3), weight $w = -1$ and bias $b = 0.5$. These definitions are summarized in Table A.1.

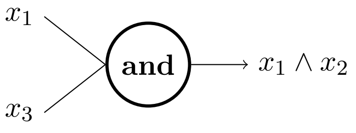
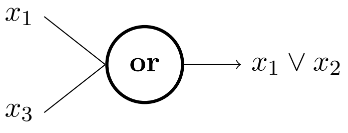
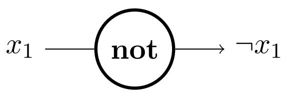
”and”-neuron			”or”-neuron			”not”-neuron	
							
$\vec{w} = (1, 1) \quad b = -1.5$			$\vec{w} = (1, 1) \quad b = -0.5$			$w = -1 \quad b = 0.5$	
x_1	x_2	$a(x_1 + x_2 - 1.5)$	x_1	x_2	$a(x_1 + x_2 - 0.5)$	x_1	$a(-x_1 + 0.5)$
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

Table A.1: A summary and depiction of the main logic gates written as neurons. All of them share the same activation function (3.3).

Using the basic logic gates a more complex structure - the ”XOR”-gate - can be built. A ”XOR”-gate is defined by its truth table (see Table A.2).

x_1	x_2	$x_1 \veebar x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Table A.2: Truth table for the ”XOR”-gate.

It can be constructed from the three basic logic operations ”and”, ”or” and ”not”

$$x_1 \veebar x_2 = \neg((x_1 \wedge x_2) \vee \neg(x_1 \vee x_2)). \tag{A.1}$$

Therefore the basic neurons from Table A.1 can be combined to create a ”XOR”-network (see Figure A.1).

To simplify readability from here on out a neuron called ”XOR” will be used. It is defined by the network of Figure A.1 and has to be replaced by it, whenever it is used.

With this ”XOR”-neuron a network, that behaves like a full-adder, can be defined. A full-adder is a binary adder with carry in and carry out, as seen in Figure A.2.

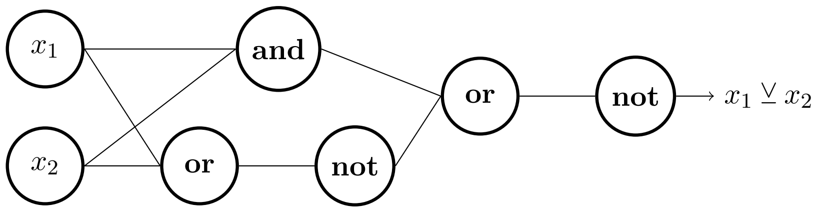


Figure A.1: The definition of a network that is equivalent to an ”XOR”-gate.

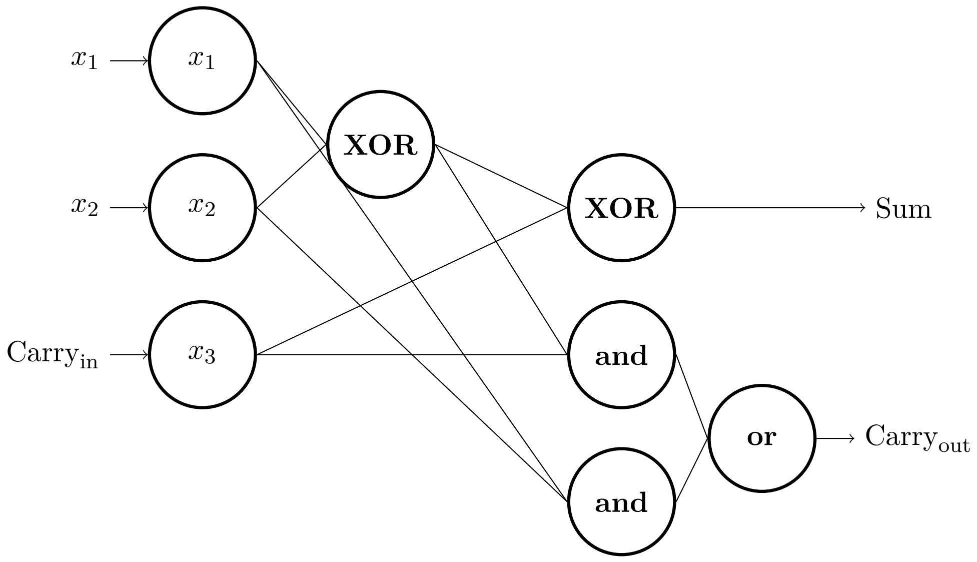


Figure A.2: A network replicating the behavior of a binary full adder.

Glossary

FFN feed forward (neural) network.

GW gravitational wave.

NN Neural network.

NNs Neural networks.

SNR signal to noise ratio.