



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*

July 10, 2019

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

I hereby assure that the thesis at hand has been constituted independently and without the use of any other than the cited sources. I furthermore assure, that all passages taken textually or analogously from other sources are marked as such.

This thesis, in its current or a similar form, has not been submitted to any other examination office.

---

Hiermit versichere ich, dass die vorliegende Arbeit selbständig und ohne Verwendung anderer Quellen, als den angegebenen, verfasst wurde. Zudem versichere ich, dass alle Stellen, die wörtlich oder sinngemäß aus anderen Quellen entnommen wurden, als solche gekennzeichnet sind.

Diese Arbeit hat so oder in einer ähnlichen Form noch keiner anderen Prüfungsbehörde vorgelegen.

---

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

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Gravitational-Wave signals from binary neutron star mergers</b>	<b>2</b>
2.1	The waveform . . . . .	2
2.2	Matched filtering . . . . .	2
<b>3</b>	<b>Neural networks</b>	<b>3</b>
3.1	Neurons, layers and networks . . . . .	3
3.2	Backpropagation . . . . .	7
3.3	Specific layers . . . . .	10
3.3.1	Dense layer . . . . .	10
3.3.2	Convolution layer . . . . .	10
3.3.3	Inception layer . . . . .	10
3.3.4	Batch Normalization layer . . . . .	10
3.3.5	Dropout layer . . . . .	10
3.3.6	Max Pooling layer . . . . .	11
<b>4</b>	<b>Our network</b>	<b>4</b>
4.1	Training data . . . . .	4
4.2	Training and final performance . . . . .	4
4.3	Comparison to matched filtering . . . . .	4
<b>5</b>	<b>Conclusion</b>	<b>5</b>
<b>6</b>	<b>Acknowledgments</b>	<b>6</b>
<b>A</b>	<b>Full adder as network</b>	<b>7</b>
<b>B</b>	<b>Indication that the network does not learn</b>	<b>8</b>
	<b>Glossary</b>	<b>11</b>
	<b>References</b>	<b>13</b>

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 backpropagation 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 general there are two different kinds of inputs to the neuron. Those that are specific to the neuron itself and those that the neuron receives as an outside stimulus. We write the neuron as

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

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). The weights and biases are what is tweaked to control the behavior of the neuron, whereas the outside stimulus is not controllable in that sense. A usual depiction of a neuron and its structure is shown in Figure 3.1.

The activation function is a usually nonlinear scalar function

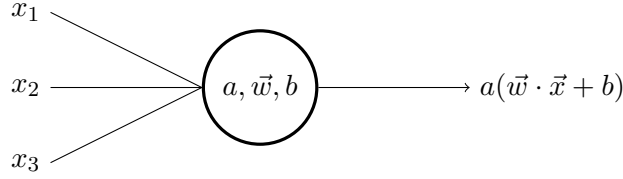
$$a : \mathbb{R} \rightarrow \mathbb{R} \quad (3.2)$$

determining the scale of the output of the neuron. The importance of this activation function and its nonlinearity will be touched upon a little later.

To understand the role of each part of the neuron, consider the following activation

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

function:

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

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 to connect them into more complicated structures, like a full-adder (see Appendix A). These structures are then 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.

In other words, a network aims to calculate some general function by connecting multiple easier functions together. This highlights the importance of the activation function, as it introduces nonlinearities into the network. Without these a neural network would not be able to approximate a nonlinear function such as the XOR-Gate used in Appendix A (section 6.1 in [1]), which caused the loss of interest in NNs around 1940 (section 6.6 in [1]).

Since NNs are the main subject of subsection 3.2 and since it 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

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

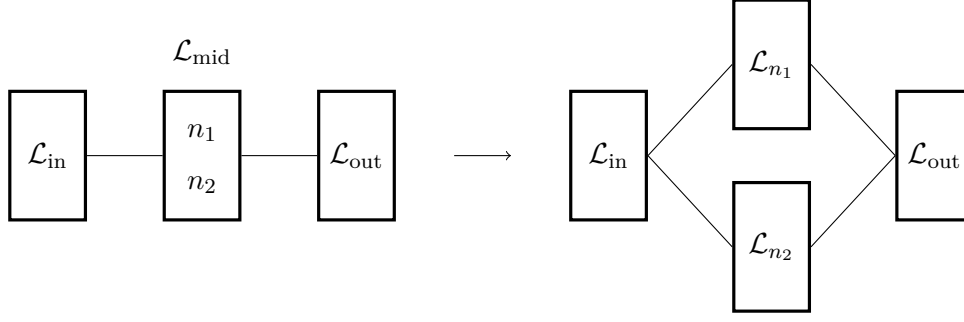
where  $n_i$  is neuron  $i$  on that layer and  $W_i$  is the  $i$ -th row of a  $k \times l$ -matrix. In principle this definition can be extended to tensors of arbitrary dimensions. This would however only complicate the upcoming sections notationally and the principle should be clear from this minimal case, as dot products, sums and other operations have their according counterparts in tensor calculus. As a further 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 then be merged afterwards (see Figure 3.2).

With this simplification one can write a layer simply as

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

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  $\mathcal{L}_{\text{in}}, \mathcal{L}_{\text{mid}}, \mathcal{L}_{\text{out}}$  can be written



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

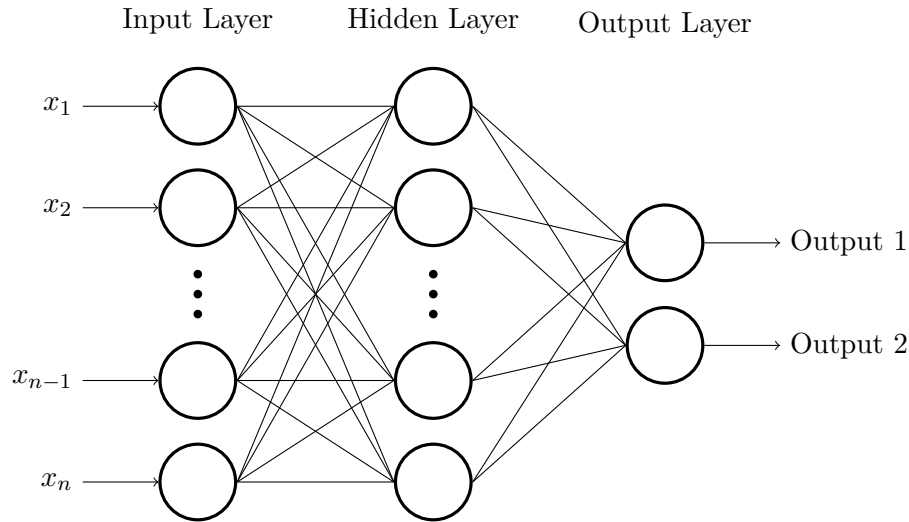
as

$$\begin{aligned}
\mathcal{N}(W^{\text{in}}, \vec{b}^{\text{in}}, W^{\text{mid}}, \vec{b}^{\text{mid}}, W^{\text{out}}, \vec{b}^{\text{out}}, \vec{x}) \\
&:= \mathcal{L}_{\text{out}}(W^{\text{out}}, \vec{b}^{\text{out}}, \mathcal{L}_{\text{mid}}(W^{\text{mid}}, \vec{b}^{\text{mid}}, \mathcal{L}_{\text{in}}(W^{\text{in}}, \vec{b}^{\text{in}}, \vec{x}))) \\
&= a_{\text{out}}(\vec{b}^{\text{out}} + W^{\text{out}} \cdot a^{\text{mid}}(\vec{b}^{\text{mid}} + W_{\text{mid}} \cdot a_{\text{in}}(\vec{b}^{\text{in}} + W_{\text{in}} \cdot \vec{x}))). \quad (3.6)
\end{aligned}$$

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 layers. Especially no loops are allowed, i.e. getting input from some subsequent layer is not permitted. 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* (RNN), 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. Therefore their shape is determined by the data the network is being fed and the expected return. The hidden-layers on the contrary are called "hidden", as their shape and size is not defined by the data. Furthermore the hidden layers do not see the input or labels directly, which means, that the network itself has to "decide" on how to use them (page 165 in [1]). 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



**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.

shape and functionality of the hidden layers. How well it performs is mainly governed by these layers. [\[Can I find citation for this last statement?\]](#)

A NN is called *deep*, if it has multiple hidden layers. [\[Citation\]](#)

## 3.2 Backpropagation

[The beginning of this section feels very wordy and repetitive. Break it down!](#)

In subsection 3.1 the basics of a NN were 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 proves that a sufficiently complicated network can - in principle - calculate any function a computer can, as a 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 approximate some function optimally. For the binary full-adder the weights and biases were chosen by hand, as the problem the network was trying to solve was rather simple. A more general approach however would be beneficial, as not all problems are this simple. Therefore the goal is to design some network and let it learn/optimize the weights and biases such that the error between the actual function and the estimate of the network is minimal.

To do this, some known and labeled data is necessary, in order for the network being able to 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 time series.

The output of this analysis will be some scalar number; the SNR. Therefore the network receives some data as input, of which the true SNR-value is known. This true value will be called *label* from here on out. 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 that maps data  $\rightarrow$  SNR. This process of optimizing the weights and biases in the way described below is enabled by a process called backpropagation, as the error propagates from the last to the first layer. The meaning of this will become clearer in the upcoming paragraphs.

The data used for training is usually split into three distinct parts, each used for different purposes. The first one is the training set. This set is used directly by the network to train on. It is the only dataset the network uses to directly improve its weights and biases. The second one is the validation set. So far only the abstract term "error" was used. This error, in machine learning language, is called the *loss function* and in general is defined by

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

where  $l$  is the number of training samples used to estimate the error and  $k$  is the dimension of the network output.

When doing a regressive fit, one of the standard error functions is the *mean squared error* (MSE), which is the loss function mainly used in this work and that is defined by

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

A more thorough discussion and justification for using MSE as loss can be found in section 5.5 and 6.2.1.1 of [1].

To minimize this loss, the weights and biases of the different layers are changed, usually using an algorithm called *gradient decent*. It works by calculating the gradient of some layer with respect to its weights and biases and taking a step in the opposite direction. For notational simplicity we'll denote the weights and biases of a network by  $\theta$  and call them collectively parameters. It is understood that  $\theta = (W^1, b^1, W^2, b^2, \dots)$ . Gradient decent is then given by

$$\theta' = \theta - \epsilon \nabla_{\theta} L(y_{\text{net}}(\theta), y_{\text{label}}), \quad (3.9)$$

where  $\epsilon$  is called learning rate and controls how large of a step is taken on each iteration. This formula assumes, that all samples from the training set are used to calculate the gradient. In practice this would be too computationally costly. Therefore the training set is split into multiple parts, called mini-batches. A step of the gradient decent is then made using only the samples from one mini-batch. This alteration of gradient decent goes by the name of *stochastic gradient decent*. The larger the mini-batch, the more accurate the estimate of the gradient and therefore fewer steps are needed to get to lower values of the loss. Each step however takes longer to calculate. This means one has to balance the benefits and drawbacks of the mini-batch size.

The real work of training a network now lies in calculating the gradient  $\nabla_{\theta} L(y_{\text{net}}(\theta), y_{\text{label}})$ ,

which is a challenge, as  $\theta$  usually consists of at least a few hundred thousand weights and biases. The algorithm, that is used to calculate this gradient, is called backpropagation or simply backprop and is mostly a iterative application of the chain rule.

For simplicity assume we have a network  $\mathcal{N}(\theta, \vec{x})$  consisting of  $n$  consecutive layers  $\mathcal{L}^1, \dots, \mathcal{L}^n$  with weights  $W^1, \dots, W^n$ , biases  $\vec{b}^1, \dots, \vec{b}^n$  and activation functions  $a_1, \dots, a_n$ . The network will be trained by minimizing the loss given in (3.8). Calculating the gradient  $\nabla_{\theta} L(y_{\text{net}}(\theta), y_{\text{label}})$  requires to calculate  $\nabla_{W^1} L, \dots, \nabla_{W^n} L$  and  $\nabla_{\vec{b}^1}, \dots, \nabla_{\vec{b}^n}$ , where

$$\nabla_{W^i} L := \begin{pmatrix} \partial_{W_{11}^i} L & \cdots & \partial_{W_{1l}^i} L \\ \vdots & \ddots & \vdots \\ \partial_{W_{k1}^i} L & \cdots & \partial_{W_{kl}^i} L \end{pmatrix}, \quad (3.10)$$

for  $W^i \in \mathbb{R}^{k \times l}$  and

$$\nabla_{\vec{b}^i} L := \begin{pmatrix} \partial_{b_1^i} L \\ \vdots \\ \partial_{b_k^i} L \end{pmatrix}, \quad (3.11)$$

for  $\vec{b}^i \in \mathbb{R}^k$ .

To calculate  $\partial_{W_{jk}^i} L$  and  $\partial_{b_j^i} L$ , define

$$\begin{aligned} z^n &:= \vec{b}^n + W^n \cdot a_{n-1}(z^{n-1}) \\ z^1 &:= \vec{b}^1 + W^1 \cdot \vec{x}, \end{aligned} \quad (3.12)$$

such that

$$\mathcal{N}(\theta, \vec{x}) = a_n(z_n). \quad (3.13)$$

To save another index, we will assume a mini-batch size of 1. For a larger mini-batch size one simply has to average over the individual gradients, as sums and derivatives commute.

With this in mind, the loss is given by

$$L(y_{\text{net}}, y_{\text{label}}) = L(\mathcal{N}(\theta, \vec{x}), y_{\text{label}}) = L(a_n(z_n), y_{\text{label}}) = (a_n(z_n) - \vec{y}_{\text{label}})^2. \quad (3.14)$$

To start off derive this loss by some scalar that only  $z^n$  depends on.

$$\partial_{\theta_j} (a_n(z_n) - \vec{y}_{\text{label}})^2 = \left( \partial_{\theta_j} a_n(z_n) \right) (2(a_n(z_n) - \vec{y}_{\text{label}})) \quad (3.15)$$

From there calculate  $\partial_{\theta_j} a_n(z_n)$ , remembering, that  $a_n$  and  $z_n$  are both vectors.

$$\begin{aligned} \partial_{\theta_j} a_n(z_n) &= \partial_{\theta_j} \sum_i a_n^i(z_{n,1}(\theta_j), \dots, z_{n,k}(\theta_j)) \vec{e}_i \\ &= \sum_i \sum_{m=1}^k \left( \partial_{\theta_j} z_{n,m}(\theta_j) \right) \left( \partial_{z_{n,m}} a_n^i(z_{n,1}(\theta_j), \dots, z_{n,k}(\theta_j)) \right) \vec{e}_i \\ &= \sum_i \left( \left( \partial_{\theta_j} z_n \right) \cdot \left( \nabla_{z_n} a_n^i \right) \right) \vec{e}_i \end{aligned} \quad (3.16)$$

Since all activation functions  $a_n^i$  on a layer are the same, the gradient  $(\nabla_{z_n} a_n^i)$  simplifies to  $\partial_z a(z)|_{z=z_{n,i}}$ . With this one gets

$$\partial_{\theta_j} a_n(z_n) = \left( \partial_{\theta_j} z_n \right) \odot \partial_z a_n(z)|_{z=z_n}, \quad (3.17)$$

where  $\odot$  denotes the Hadamard product. The final step to understanding backpropagation is to evaluate  $\partial_{\theta_j} z_n$ . For now assume that  $\theta_j$  is some weight on a layer that is not the last layer.

$$\begin{aligned} \partial_{\theta_j} z_n &= \partial_{\theta_j} \left( \vec{b}^n + W^n \cdot a_{n-1}(z_{n-1}) \right) \\ &= \partial_{\theta_j} W^n \cdot a_{n-1}(z_{n-1}) \\ &= W^n \cdot \partial_{\theta_j} a_{n-1}(z_{n-1}) \end{aligned} \quad (3.18)$$

Inserting (3.18) into (3.17) yields the recursive relation

$$\partial_{\theta_j} a_n(z_n) = W^n \cdot \partial_{\theta_j} a_{n-1}(z_{n-1}) \odot \partial_z a_n(z)|_{z=z_n}. \quad (3.19)$$

The recursion stops, when the layer the recursion is at, is the layer the weight  $\theta_j$  is located on. Assuming, the layer weight  $\theta_j$  is located on is layer  $k$ , one gets

$$\partial_{\theta_j} a_n(z_n) = \left( \prod_{l=0}^{n-k} W^{n-l} \right) \cdot \left( \partial_{\theta_j} W^k \right) a_{k-1}(z_{k-1}) \odot \left( \bigodot_{l=0}^{n-k} \partial_z a_l(z)|_{z=z_l} \right). \quad (3.20)$$

### 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.

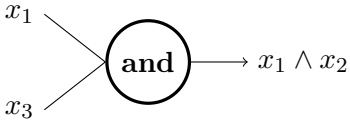
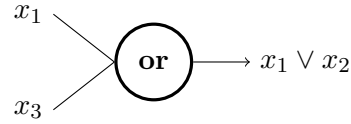



### **3.3.6 Max Pooling layer**

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																																				
																																						
$\vec{w} = (1, 1) \quad b = -1.5$	$\vec{w} = (1, 1) \quad b = -0.5$	$w = -1 \quad b = 0.5$																																				
<table> <tr> <th><math>x_1</math></th> <th><math>x_2</math></th> <th><math>a(x_1 + x_2 - 1.5)</math></th> </tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	$x_1$	$x_2$	$a(x_1 + x_2 - 1.5)$	0	0	0	0	1	0	1	0	0	1	1	1	<table> <tr> <th><math>x_1</math></th> <th><math>x_2</math></th> <th><math>a(x_1 + x_2 - 0.5)</math></th> </tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	$x_1$	$x_2$	$a(x_1 + x_2 - 0.5)$	0	0	0	0	1	1	1	0	1	1	1	1	<table> <tr> <th><math>x_1</math></th> <th><math>a(-x_1 + 0.5)</math></th> </tr> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td></tr> </table>	$x_1$	$a(-x_1 + 0.5)$	0	1	1	0
$x_1$	$x_2$	$a(x_1 + x_2 - 1.5)$																																				
0	0	0																																				
0	1	0																																				
1	0	0																																				
1	1	1																																				
$x_1$	$x_2$	$a(x_1 + x_2 - 0.5)$																																				
0	0	0																																				
0	1	1																																				
1	0	1																																				
1	1	1																																				
$x_1$	$a(-x_1 + 0.5)$																																					
0	1																																					
1	0																																					

**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).

$x_1$	$x_2$	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	0

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

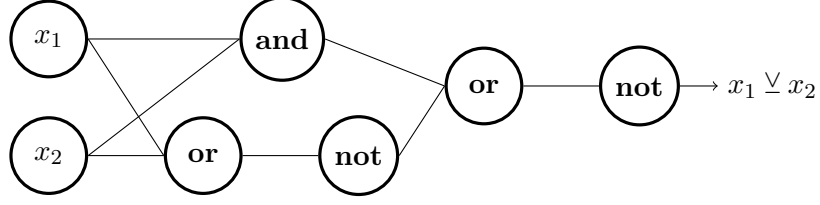
It can be constructed from the three basic logic operations "and", "or" and "not"

$$x_1 \vee x_2 = \neg((x_1 \wedge x_2) \vee \neg(x_1 \vee x_2)). \quad (\text{A.1})$$

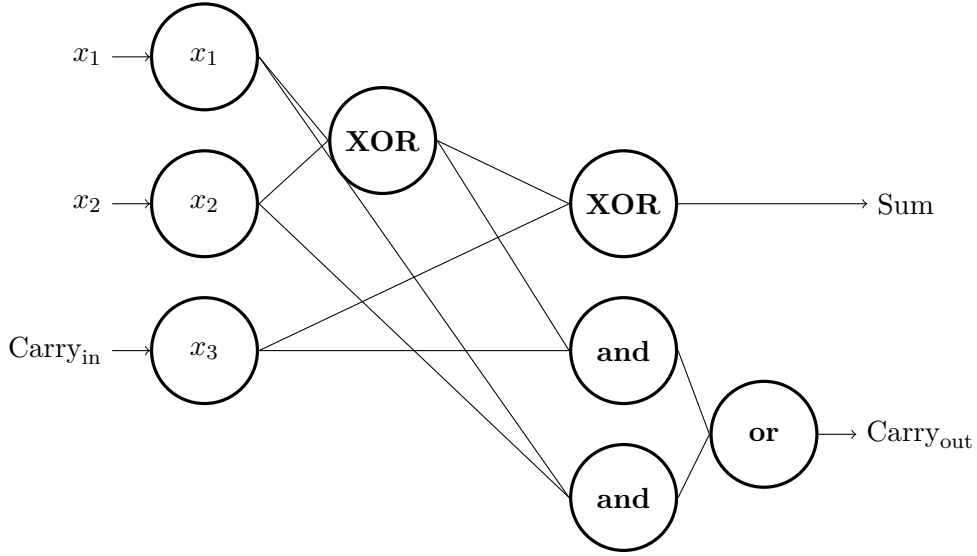
Therefore the basic neurons from Table A.1 can be combined to create a "XOR"-network (see Figure A.1).

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.

## B Indication that the network does not learn

Sometimes during training only a local minimum is found. The most notable of these is when the network just picks a fixed value in the interval and appoints it to any input. It is therefore useful to spot this behavior during training and consider restarting the training again. Therefore the value for a constant output will be calculated in this appendix.

To start off the loss function will be assumed to be the mean squared error and that the SNR-values lie within the interval  $[a, b]$ . The mean squared error of a chosen point  $x$  to every point in this interval is given by

$$f(x) := \frac{1}{b-a} \int_a^b dy (x-y)^2 = \frac{1}{3(b-a)} ((b-x)^3 - (a-x)^3). \quad (\text{B.1})$$

To minimize this one could in principle solve  $\frac{\partial f}{\partial x} \stackrel{!}{=} 0$  and it would yield the correct result. However it should at least intuitively be obvious that the point that minimizes the mean squared error to every point in the interval is the mean value of the interval  $x = \frac{a+b}{2}$ . Therefore if the network has to choose values out of a given interval  $[a, b]$ , minimize the mean squared error and arrives in a local minimum that leads to the network always returning a fixed value, this value should be

$$f\left(\frac{a+b}{2}\right) = \frac{1}{3(b-a)} \left( \left(\frac{b-a}{2}\right)^3 - \left(\frac{a-b}{2}\right)^3 \right) = \frac{1}{12}(b-a)^2. \quad (\text{B.2})$$

In this work another common case is the network having to choose a SNR-value from some continuous interval  $[a, b]$  or pick a discrete value  $c$ . The interval corresponds to the data containing some GW-signal and the fixed point with value  $c$  is the SNR-value assigned to pure noise during training. (This is by no means a strictly mathematical derivation but simply a quick way to calculate the expected value.) The mean squared error is hence given by

$$\begin{aligned} g(x) &:= \lim_{n \rightarrow \infty} \left( \frac{1}{n} \sum_{i=1}^n (x - y_i)^2 \right) \text{ with } \begin{cases} y_i = a_i \in [a, b], \text{ probability } p \\ y_i = c, \text{ probability } (1-p) \end{cases} \\ &= \lim_{n \rightarrow \infty} \left( \frac{1}{n} \sum_{i=1}^{p \cdot n} (x - a_i)^2 \right) + \lim_{n \rightarrow \infty} \left( \frac{1}{n} \sum_{i=1}^{(1-p) \cdot n} (x - c)^2 \right) \\ &= \lim_{n \rightarrow \infty} \left( \frac{1}{n} \sum_{i=1}^{p \cdot n} (x - a_i)^2 \right) + \lim_{n \rightarrow \infty} \left( \frac{(1-p) \cdot n}{n} (x - c)^2 \right) \\ &\stackrel{(*)}{=} \lim_{n \rightarrow \infty} \left( \frac{p}{n} \sum_{i=1}^n (x - a_i)^2 \right) + (1-p)(x - c)^2 \\ &= p \underbrace{\lim_{n \rightarrow \infty} \left( \frac{1}{n} \sum_{i=1}^n (x - a_i)^2 \right)}_{=f(x)} + (1-p)(x - c)^2 \\ &= pf(x) + (1-p)(x - c)^2, \end{aligned} \quad (\text{B.3})$$

where the step in  $(*)$  is not clear and would need a mathematical proof, but intuitively should be clear. **(If there is time, find a proof.)** For large  $n$  all  $a_i$  should contribute equally to the mean value and hence  $p$  is just a proportionality factor.

With  $\partial_x f(x) = 2x - a - b$  one gets

$$\partial_x g(x) \stackrel{!}{=} 0 \Leftrightarrow x = \frac{p}{2}(a+b) + (1-p)c \quad (\text{B.4})$$

as expected. The value of  $g$  at this point will be the expectation value of the mean squared error if the network predicts a single value and optimizes this value. In this work  $p$  is the probability of looking at data containing a GW, i.e. the fraction of data-points containing a GW over total number of data-points.



## Glossary

**FFN** feed forward (neural) network.

**GW** gravitational wave.

**MSE** mean squared error.

**NN** Neural network.

**NNs** Neural networks.

**RNN** recurrent neural network.

**SNR** signal to noise ratio.



## References

- [1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016 (cit. on pp. 5, 6, 8).