



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 31, 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

1	Introduction	1
2	Gravitational-Wave signals from binary neutron star mergers	2
2.1	The waveform	2
2.2	Matched filtering	2
3	Neural networks	3
3.1	Neurons, layers and networks	3
3.2	Backpropagation	7
3.3	Training and terminology	11
3.4	Convolution neural networks	12
3.4.1	Convolution layer	14
3.4.2	Max Pooling layer	14
3.4.3	Inception module	14
3.4.4	Temporal Convolutional Networks	16
3.5	Regularization	16
3.5.1	Batch Normalization layer	16
3.5.2	Dropout layer	16
4	Our network	4
4.1	Training data	4
4.2	Training and final performance	4
4.3	Comparison to matched filtering	4
5	Conclusion	5
6	Acknowledgments	6
A	Full adder as network	7
B	Indication that the network does not learn	8
	Glossary	11
	References	13

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.

Large portions of this section are inspired and guided by [1, 2].

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.

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.

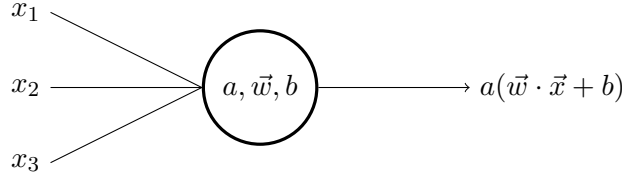


Figure 3.1: Depiction of a neuron with inputs $\vec{x} = (x_1, x_2, x_3)^T$, weights \vec{w} , bias b and activation function a .

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

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

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 net-

works 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 [2]), which caused the loss of interest in NNs around 1940 (section 6.6 in [2]).

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

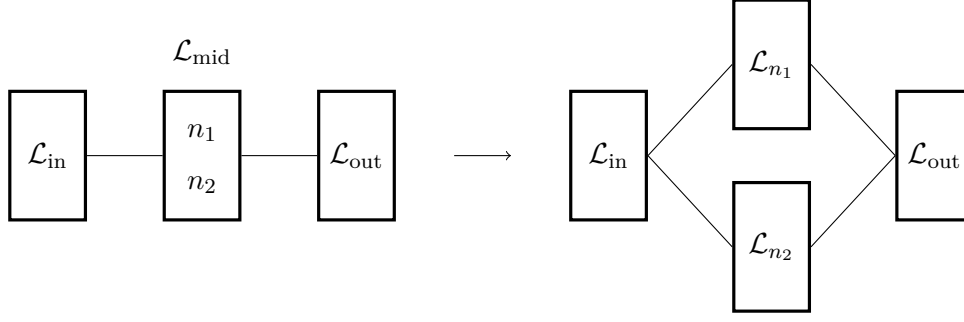


Figure 3.2: Depiction of how a layer (\mathcal{L}_{mid}) consisting of neurons with different activation functions (n_1 and n_2) can be split into two separate layers (\mathcal{L}_{n_1} and \mathcal{L}_{n_2}).

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 [2]). 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

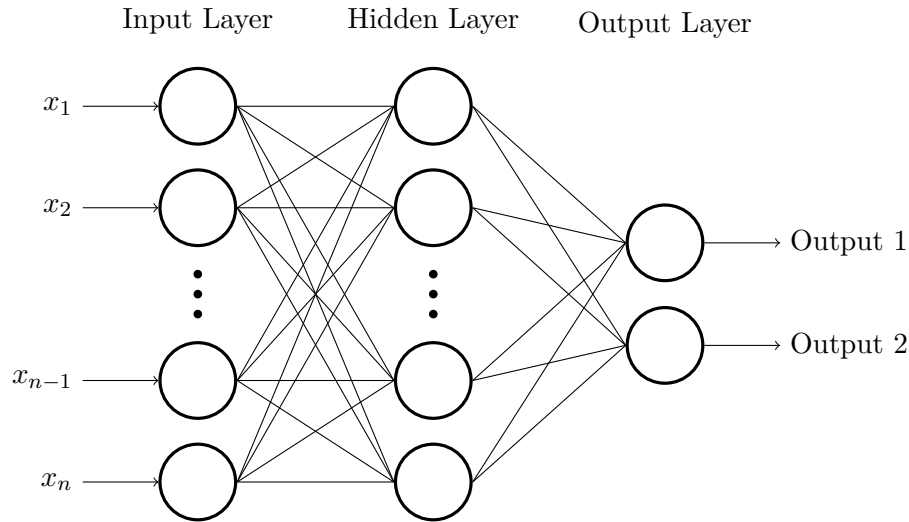


Figure 3.3: A depiction of a simple network with a single input-, hidden- and output-layer. The input-data is a n -dimensional vector $(x_1, \dots, x_n)^T$ and the output is a 2-dimensional vector. In this picture it looks like the hidden layer has the same number of neurons as the input layer. This 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.¹[\[1\]](#) 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

¹There is an even stronger statement called the universal approximation theorem, which states that any Borel measurable function on a finite-dimensional space can be approximated to any degree with a NN with at least one single hidden layer of sufficient size. (p. 194 [\[2\]](#))

minimize some error function. This way of optimizing the weights and biases is called *training*. The data used during training is hence called training data or training set. 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. 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 [2].

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 θ 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 ϵ 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 θ 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

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

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 θ_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

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

The recursion stops, when it reaches the layer the weight θ_j is located on and evaluates to (assuming θ_j is part of layer k)

$$\partial_{\theta_j} a_k(z_k) = \left(\partial_{\theta_j} W^k \right) \cdot a_{k-1}(z_{k-1}). \quad (3.20)$$

The derivative can also be expressed in an analytical form, by utilizing, that the Hadamard product is commutative and can be expressed in terms of matrix multiplications. To do so define

$$[\Sigma(\vec{x})]_{ij} = \begin{cases} x_i, & i = j \\ 0, & \text{otherwise} \end{cases}. \quad (3.21)$$

With this definition equation (3.19) can be written as

$$\partial_{\theta_j} a_n(z_n) = \Sigma\left(\partial_z a_n(z)|_{z=z_n}\right) W^n \cdot \partial_{\theta_j} a_{n-1}(z_{n-1}) \quad (3.22)$$

and the recursion can be solved to yield

$$\partial_{\theta_j} a_n(z_n) = \left[\prod_{l=0}^{n-k+1} \Sigma\left(\partial_z a_{n-l}(z)|_{z=z_{n-l}}\right) \cdot W^{n-l} \right] \cdot \Sigma\left(\partial_z a_k(z)|_{z=z_k}\right) \cdot \left(\partial_{\theta_j} W^k \right) a_{k-1}(z_{k-1}). \quad (3.23)$$

If there is time, check the equations below, as I did not thoroughly recompute them.

The same computation can be done if θ_j is a bias instead of a weight. When this computation is done, equation (3.19) still holds, but the stopping condition (3.20) is simplified to

$$\partial_{\theta_j} a_k(z_k) = \partial_{\theta_j} \vec{b}^k. \quad (3.24)$$

From this the analytic form can be computed to be

$$\partial_{\theta_j} a_n(z_n) = \left[\prod_{l=0}^{n-k+1} \Sigma\left(\partial_z a_{n-l}(z)|_{z=z_{n-l}}\right) \cdot W^{n-l} \right] \cdot \Sigma\left(\partial_z a_k(z)|_{z=z_k}\right) \cdot \partial_{\theta_j} \vec{b}^k. \quad (3.25)$$

Using the recursive formula (3.19) now justifies the term "backpropagation". When a sample is evaluated, it is passed from layer to layer starting at the front. Therefore this is called a *forward pass*. The output the network gives for a single forward pass will probably differ from the label and hence has an error (quantified by the loss function). This error is used to calculate the gradient and adjusts the parameters of the network. The way this is done is given by (3.19). It starts at the last layer and propagates back through the network until it reaches the layer of the weight that should be adjusted.

With these formulae one could in principle calculate the gradient of the loss with respect to all parameters θ and use this gradient to optimize and train a network. In reality this would still be too slow and computationally costly. Instead each layer (or rather each operation) has a backpropagation method associated to it, that returns the gradient based on a derivative to one of its inputs and the gradient from the previous layer.

For clarification, consider an operation that multiplies two matrices A and B and say the gradient calculated by the backpropagation method of the previous layer returned G as its gradient. The backpropagation method for the matrix multiplication now needs to implement the derivative with respect to A and the derivative with respect to B . Thus it will return $G \cdot B^T$ when derived by A and $G \cdot A^T$ when derived by B . (section 6.5.6 [2])

The full backpropagation algorithm than only has to call the backpropagation methods of each layer/operation. (For a more thorough discussion see section 6.5 of [2].)

3.3 Training and terminology

In the previous subsection 3.2 the backpropagation algorithm was introduced as the method used for the network to learn. It used some labeled data to compare its output to and adjust the parameters accordingly. This labeled data was called the training set. In principle the network could be trained over and over again on the same data to further improve the performance of the network. The worst one could fear for is a gradient that vanishes, as the global or a local minimum in the loss is reached.

In practice this is true only partially. At some point the network will start "memorizing" the samples it has seen in the training set. When a network shows this behavior during training it is called *overfitting*. This is a problem not only known in machine learning but also with regressive fits and has the same reason; too many free parameters. Consider a parabola sampled at n points. If a regressive fit is done, the best choice for a basis would be a parabola $f(x) = ax^2 + bx + c$ with the three free parameters a , b and c . If $n \geq 3$ a regressive fit minimizing the MSE would recover the original parabola that was sampled by the n points. However one could also use a polynomial of degree $m > n$ to find a function that runs exactly through all n points and thus minimizes the MSE to the same value of zero too. (see Figure 3.4)

There are however two differences between the two cases. The most obvious one is the number of free parameters. The parabola has three parameters, whereas the polynomial of degree m has m free parameters. As $m > n$ was required, there is at least one parameter that cannot be fixed by the data and is therefore still free. The second difference is the behavior of the MSE when another point is added to the set but the

regression is not redone. For the parabola the MSE will stay zero, for the polynomial of degree m however the MSE will most likely be greater than zero, as the true parabola isn't matched. (Compare lower right of Figure 3.4) The first difference explains why overfitting takes place, there are too many parameters that can be varied, the second difference gives a way to detect when overfitting takes place. If the MSE rises on samples that were not used for the regression, overfitting takes place.

The same concept can then be applied to NNs; if the loss of a network is bigger on different data than that used during training, the network is said to overfit. This second set of samples is called the validation set, as it validates the training. Obviously the data in the validation set must stem from the same underlying procedure, that generated the training set. In the context of this work this means, that the waveforms of the training and validation set must share the same parameter-space.

Contrary to overfitting, there is also a phenomenon called *underfitting*. This occurs, when the number of free, i.e. trainable, parameters of a network is too low. It manifests usually in an occasionally lower loss value of the validation set when compared to the training set. To overcome this issue one can simply increase the number of trainable parameters the network has. Increasing the number of trainable parameters is also called increasing the *capacity* of the network.

Though underfitting is possible, overfitting is usually a lot more common. There are multiple ways to deal with a network, that overfits during training. The first one would be to reduce the capacity of the network. If that is not possible or worsens the results, the second easiest way is to increase the number of samples in the training set. In the realm of this work, this is a possibility, as we use simulated data, that can be generated on demand. For a lot of other applications however this is not feasible and other means are necessary. One way is to use a technique called regularization, which is explained in subsection 3.5 and applied to our networks as well. Another one, which will not be discussed here, is data augmentation. (See section 7.4 of [2])

To tune out the generalization error, which is the difference between training set and validation set loss (Is this true?), one adjusts the architecture. If there are multiple different architectures, the best one is chosen by the performance on the validation set. In this way, the validation set is also used to fit the model, as in the end the human who trains the models selects the best performing network. Therefore all given results, that did not occur during training², come from a third independent set. This set is then called the test set.

3.4 Convolution neural networks

In the previous sections only fully connected layers were used to build networks. These are layers, where each neuron is connected to every neuron on the previous layer. These fully connected layers are called *dense* layers. (See Figure 3.5) In this section a different kind of layer and variants of it will be motivated and introduced. It is the main driving force of modern neural networks and is called convolution layer.

²An example of a result that comes from training the network would be the loss history.

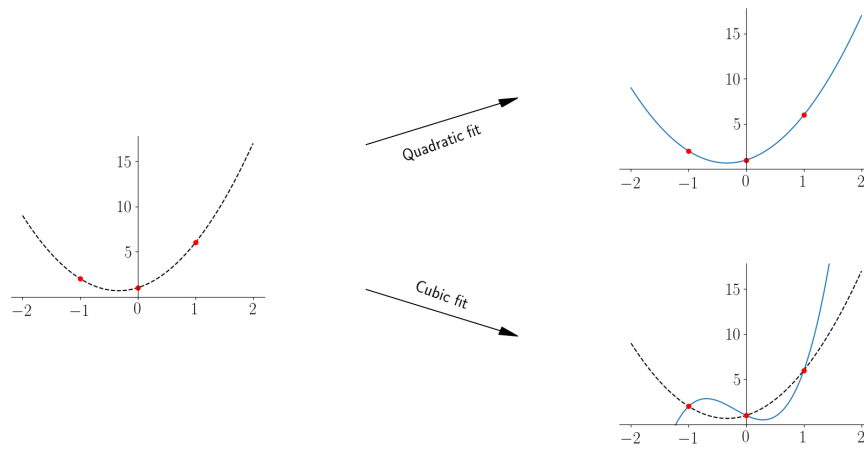


Figure 3.4: Depiction of overfitting in the classical regression. On the left the originally sampled function $f(x) := 3x^2 + 2x + 1$ is shown in dashed and black. The red dots are the samples that are being used for the regression on the right. The top right shows the regression, where $g(x) = ax^2 + bx + c$ was used as a basis and recovered the correct points $a = 3$, $b = 2$ and $c = 1$. All free parameters are fixed by the data. The lower right plot shows a case of overfitting. The same points are now used to fit the parameters a , b , c and d of the function $h(x) = ax^3 + bx^2 + cx + d$. The analytic solution returns $b = 3$, $c = 2 - a$ and $d = 1$ with a being free. Therefore a possible regression could use $a = 5$, which is used in the lower right plot. The points used for regression are all hit, hence the MSE is zero. However if another point on the black dashed line would be used, the fitted model would be off.

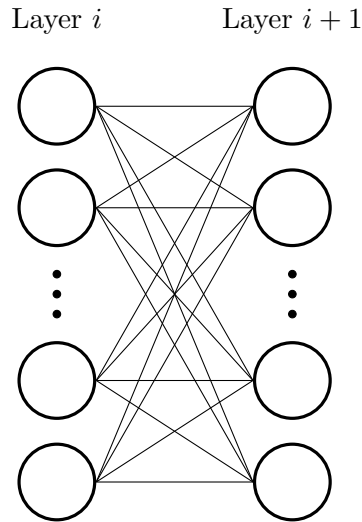


Figure 3.5: Insert description! Maybe improve this graphic.

3.4.1 Convolution layer

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

3.4.2 Max Pooling layer

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

3.4.3 Inception module

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

Networks consisting of stacked convolution layers as introduced in subsubsection 3.4.1 have had great success in image classification. [2, 3, 4] As the field of computer vision is one of the most prominent in machine learning and shows great advances, we use networks successfully applied there as a guideline for new architectures. (Rephrase this part maybe, as the section is structured differently now.)

The advantages of convolution layers over classical dense layers are manifold and discussed in further detail in subsubsection 3.4.1. One of the key advantages however is the comparatively low number of trainable parameters, as the connections are a lot more sparse. The number of these trainable parameters however is still quite large and limits the depth of a Deep-CNN. This becomes especially obvious, when one scales the number of filters used in the convolution layers, throughout the network. If the number of filters of two consecutive convolution layers is scaled by a factor c , the number of trainable parameters increases by a factor of c^2 . (scaling the number of filters is one way to increase the capacity of a network and reduce underfitting) [5] Another way to achieve the same

effect is to scale the convolution-kernel size. **Need to talk about over- and underfitting, model capacity and training data differences in a section before this one! (Probably best after the backprop section)** Larger kernels furthermore provide the capability to detect larger features within a certain part of the image, but might have the problem of being close to zero, which in turn wastes a lot of computational resources. In this situation an approach that utilizes sparse matrices or tensors would be quite beneficial, if the computational infrastructure supports it efficiently. The advantage gained by the lowered number of computations is however mostly outweighed by the computational overhead created. Therefore sparse matrix operations are not feasible at the moment. [5]

A workaround for this problem is grouping multiple sparse operations together into matrices that are mostly sparse but contain dense submatrices. The matrix-operations can then be performed efficiently on the sparse matrices, by utilizing the efficient dense operations on the dense submatrices. This is the approach, that the inception modules try to take. They build a single module, that can be viewed as a layer from the outside. It contains multiple small convolution layers, that build up a larger, sparse filter. Using this new architecture, the GoogLeNet won the 2014 ILSVRC³ image recognition competition in the category "image classification and localization", setting a new record for the top 5 error rate, thus proving the effectiveness of the new module. [5, 4]

As the original work was used to handle 2 dimensional images and thus used 2D-convolutions, the module had to be slightly adjusted to fit the 1 dimensional requirements of the time series data in this work. This was a simple task however, as the difference between the two is simply the shape of the kernel and the way it is moved across the data. With Keras, there are predefined functions to handle 1D and 2D convolutions. The downside of converting the 2 dimensional inception module to a 1 dimensional one however is, that many of the incremental improvements to the module are not applicable, as they rely heavily on the 2D-structure. [6, 7]

The following paragraphs will describe the module used in this work in greater detail. The module consists of 4 independent towers, each consisting of different layers. The full module is depicted in Figure 3.6.

The module consists of three parallel convolution layers, i.e. each of the three layers share the same input. The difference between them is the kernel size. The convolution layers with a larger kernel are preceded by a convolution layer with 16 filters and a kernel size of 1. The purpose of this step is to reduce the number of channels used and is called dimensional reduction. This leads to a fixed input size for the larger kernels, regardless of the depth of the input. In the original architecture filters of size 1×1 , 3×3 and 5×5 were used. Translating them directly to 1 dimensional equivalents, the module should use kernel sizes of 1, 3 and 5. However we empirically found, that the smallest kernel sizes 1, 2 and 3 performed best. **(Did I ever try 1,3,5? If not do so!)**

Finally a pooling layer as introduced in subsection 3.4.2 is added as a fourth path.

³The ILSVRC is a yearly competition for computer vision algorithms. It is widely used as a benchmark to judge how well a network (or any other computer vision software) does. It is always the same set of images, where each image belongs to one of about 1000 classes. The top 5 error rate is the relative number of times, the algorithm in use did not return the correct category within its top 5 choices.

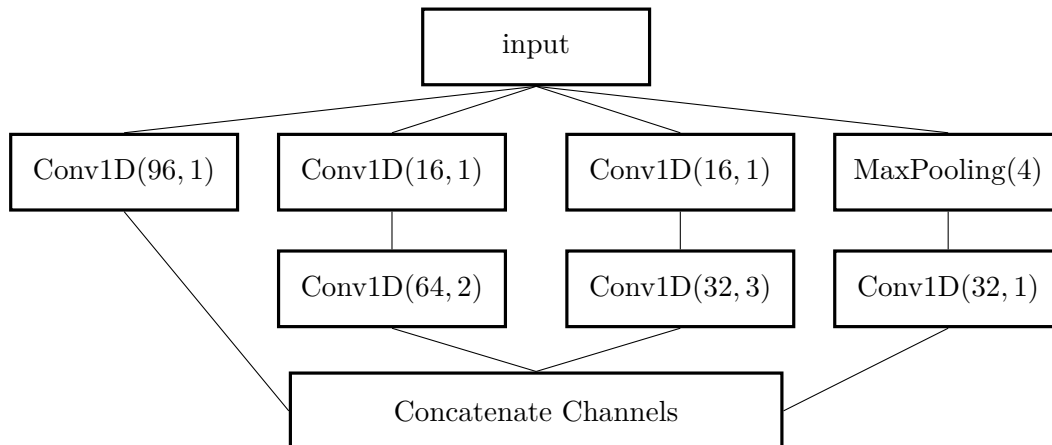


Figure 3.6: Shown are the contents and connections of the inception module as used in this work. (If the filter numbers and values change for the final architecture, change them here too.) The layer $\text{Conv1D}(x, y)$ is a 1 dimensional convolution layer with x filters and a kernel size of y . Most of the convolution layers with a kernel size of 1 are used for dimensional reduction. The only exception is the leftmost one, that consists of 96 filters. The different filter sizes correspond to the ability of detecting features at different scales. The pooling layer is a 1 dimensional pooling layer, that only passes on the maximum value in a bin of size 4. The final layer concatenates the channels of the different towers. This also means, that each tower needs to have the same output-shape, excluding the channels. For this reason all inputs are automatically padded with zeros in such a way, that the output-shapes are correct.

The reasoning behind this step is, that pooling layers have shown great improvements in traditional CNNs and thus the network should be provided with the option to choose this route as well. For this layer the dimensional reduction takes place only after the pooling procedure.

The output of each of these paths is then concatenated along the last axis of the tensor, i.e. along the different channels. For this reason all input to each of the layers is padded with zeros in such a way, that the shape (except for the channels) does not change.

3.4.4 Temporal Convolutional Networks

Explain what they are, what their advantages are and list works that utilized them.

3.5 Regularization

3.5.1 Batch Normalization layer

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

3.5.2 Dropout layer

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

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.

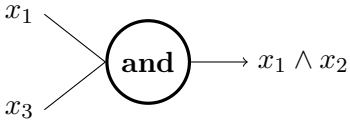
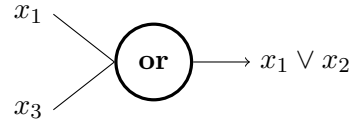
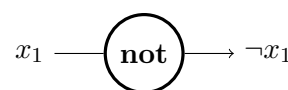
"and"-neuron	"or"-neuron	"not"-neuron																																				
																																						
$\vec{w} = (1, 1) \quad b = -1.5$	$\vec{w} = (1, 1) \quad b = -0.5$	$w = -1 \quad b = 0.5$																																				
<table> <tr> <th>x_1</th> <th>x_2</th> <th>$a(x_1 + x_2 - 1.5)$</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>x_1</th> <th>x_2</th> <th>$a(x_1 + x_2 - 0.5)$</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>x_1</th> <th>$a(-x_1 + 0.5)$</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.

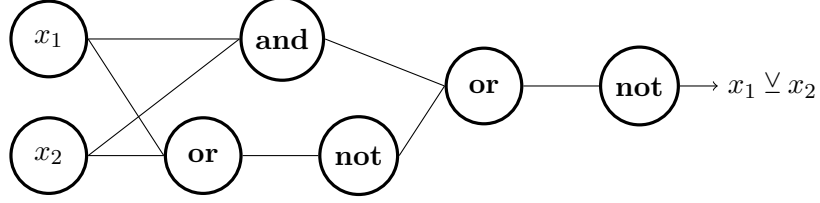


Figure A.1: The definition of a network that is equivalent to an "XOR"-gate.

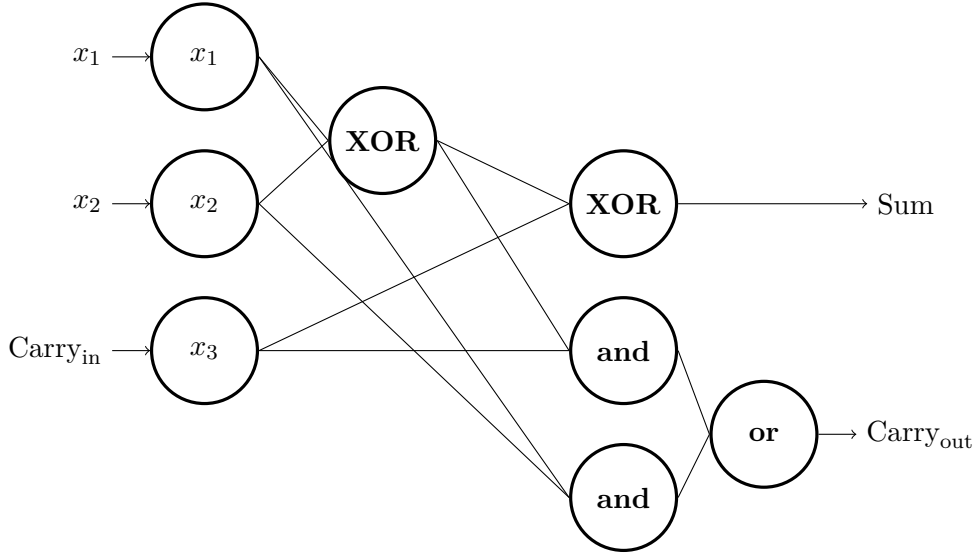


Figure A.2: A network replicating the behavior of a binary full adder.

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

CNN Convolution neural network.

CNNs Convolution neural networks.

FFN feed forward (neural) network.

GW gravitational wave.

ILSVRC ImageNet Large Scale Visual Recognition Challenge.

MSE mean squared error.

NN Neural network.

NNs Neural networks.

RNN recurrent neural network.

SNR signal to noise ratio.

References

- [1] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: <http://neuralnetworksanddeeplearning.com/index.html> (cit. on pp. 3, 7).
- [2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: <http://www.deeplearningbook.org> (cit. on pp. 3, 5, 6, 7, 8, 11, 12, 14).
- [3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf> (cit. on p. 14).
- [4] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: [10.1007/s11263-015-0816-y](https://doi.org/10.1007/s11263-015-0816-y) (cit. on pp. 14, 15).
- [5] Christian Szegedy et al. “Going Deeper With Convolutions”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015. URL: https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf (cit. on pp. 14, 15).
- [6] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016. URL: https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Szegedy_Rethinking_the_Inception_CVPR_2016_paper.html (cit. on p. 15).
- [7] Christian Szegedy et al. “Inception-v4, inception-resnet and the impact of residual connections on learning”. In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017. URL: <https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/viewPaper/14806> (cit. on p. 15).