Leibniz Universität Hannover

Binary Merger Observations & Numerical Relativity

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

August 14, 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

List of Figures

List of Tables

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

2 Gravitational-Wave signals from binary neutron star mergers

Explain the use for this section.
Gravitational waves from two inspiraling neutron stars are among the most interesting signals gravitational wave detectors can detect. They convey information about the highly relativistic regimes of gravity, about the structure of the component stars and about the formation channels of black holes or heavy neutron stars. [Citations] They are however also very hard to detect, as binary neutron star (BNS) systems are very light systems, when compared to inspiraling binary black holes (BBH).
Part 1 of this section will discuss how gravitational waves (GW) are formed and what influences the structure of the resulting waveforms. Part 2 will go over the current method of detecting GW and discuss the advantages and drawbacks.

2.1 The waveform

Explain how the waveform looks like, what it depends on, maybe give the concept how it works in the context of linearized theory (quote bachelor thesis), cite important papers regarding the waveform theory.
Gravitational waves are a solution to the Einstein-equation

$$\mathcal{G}_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}, \tag{2.1}$$

where $\mathcal{G}_{\mu\nu}$ is the Einstein-tensor, $T_{\mu\nu}$ is the energy-momentum-tensor, G is the gravitational constant and c is the speed of light in vacuum. They can be derived, in their linear form, by setting the energy-momentum-tensor to 0 and assuming the metric to be a linear correction of the flat metric $\eta_{\mu\nu}$

$$g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu}. \tag{2.2}$$

With these approximations the Einstein-equation (2.1) simplifies to

$$\mathcal{G}_{\mu\nu} = \frac{1}{2}(\partial_{\alpha\mu} h^{\alpha}_{\nu} + \partial^{\alpha}_{\nu} h_{\mu\alpha} - \partial_{\mu\nu} h - \Box h_{\mu\nu} - \eta_{\mu\nu}\Box h) = 0, \tag{2.3}$$

where $h \coloneqq \eta^{\mu\nu} h_{\mu\nu}$ and $\Box \coloneqq \eta^{\mu\nu}\partial_{\mu\nu}$.
This equation has, due to the symmetry of $h_{\mu\nu}$, 10 independent components, of which only 2 are physical. To see this one can for instance choose the DeDonder-gauge condition

$$\partial^{\alpha}\bar{h}_{\alpha\mu} = 0, \tag{2.4}$$

where $\bar{h}_{\mu\nu} \coloneqq h_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu} h$. One can further restrict the gauge to also satisfy $\bar{h} = -h = 0$ and $\bar{h}_{0\mu} = 0 = \bar{h}_{3\mu}$. The gauge is named transverse-traceless-gauge (TT) and results in the metric to be of the form

$$h^{\mathrm{TT}}_{\mu\nu} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & h_+ & h_\times & 0 \\ 0 & -h_\times & h_+ & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \tag{2.5}$$

Furthermore, (2.3) simplifies to

$$\Box h^{\mathrm{TT}}_{\mu\nu} = 0 \tag{2.6}$$

and thus the components h_+ and $h_\times$ satisfy a wave equation.

2.2 Matched filtering

Explain what matched filtering is, why it works and how it is applied currently.

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 NNs 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 NNs 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 NNs 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 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 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

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

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). 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

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

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.

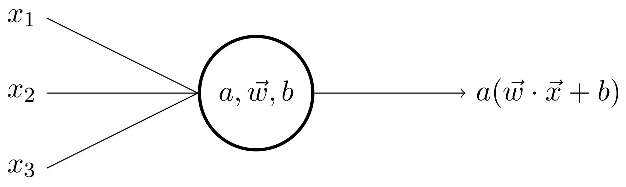


Figure 3.1: Depiction of a neuron with inputs $\vec{x} = (x_1, x_2, x_3)^T$, weights $\vec{w}$, bias b and activation function a.

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

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

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.

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 than called neural networks, as they are a network of neurons. The example of the full-adder demonstrates the principle of a NN 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 can be structured into multiple layers. Each layer consists of one or multiple neurons and each neuron has inputs only from the previous layer. Formally we write

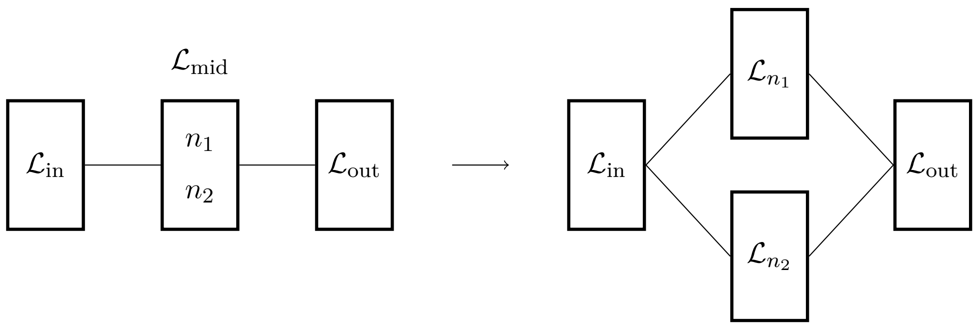
$$\mathcal{L} : \mathbb{R}^{k\times l} \times \mathbb{R}^{l} \times \mathbb{R}^{k} \to \mathbb{R}^{l};\ (W, \vec{b}, \vec{x}) \mapsto \mathcal{L}(W, \vec{b}, \vec{x}) := \begin{pmatrix} n_1\Big((W_1)^T, b_1, \vec{x}\Big) \\ \vdots \\ n_l\Big((W_l)^T, b_l, \vec{x}\Big) \end{pmatrix}, \tag{3.4}$$

where n_i is neuron i on the 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 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 $\mathcal{L}_{\text{in}}$, $\mathcal{L}_{\text{mid}}$, $\mathcal{L}_{\text{out}}$ can be written



$\mathcal{L}_{\text{mid}}$
$\mathcal{L}_{\text{in}}$
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 (n_1 and n_2) can be split into two separate layers ($\mathcal{L}_{n_1}$ and $\mathcal{L}_{n_2}$).

as

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

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 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 the input to the network is propagated from front to back, layer by layer and each layer gets invoked only once. There are also other architectures called *recurrent neural networks* (RNN), which allow for loops 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 can in general be grouped into 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

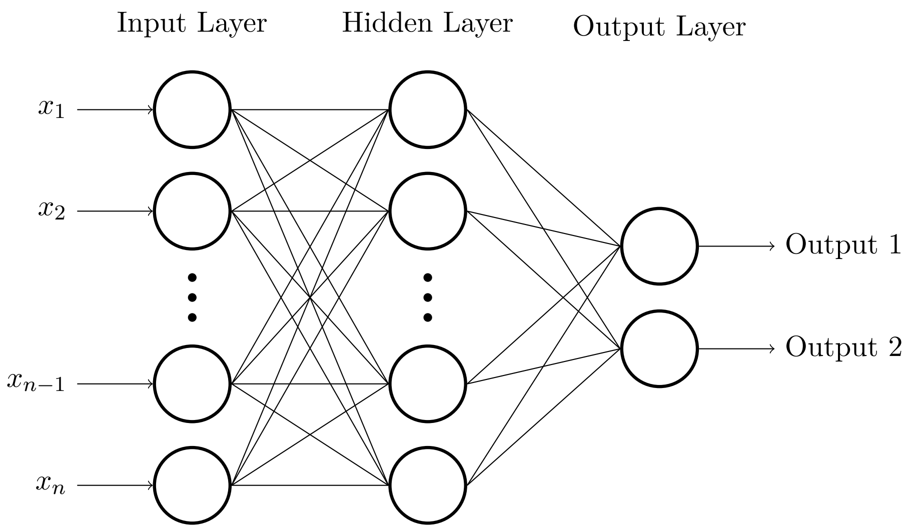


Figure 3.3: A depiction of a simple network with a single input-, hidden- and output-layer. The input-data is a n-dimensional vector $(x_1, \ldots, x_n)^T$ and the output is a 2-dimensional vector. In this picture it looks like the hidden layer has the same number of neurons as the input layer. This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

the data or output. Therefore when designing a network architecture, one designs the shape and functionality of the hidden layers. How well it performs is mainly governed by theses layers. Two networks with different hidden layers are also called different architectures. The architecture of a neural network hence describes how all layers of the network behave and are connected. [Can I find citation for these last statements?]
A NN is called *deep*, if it has multiple hidden layers. The depth of a network analogously is number of layers at the longest path to an output layer. [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 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 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

[1]There 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])

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*. 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

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

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

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

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 $\theta = (W^1, b^1, W^2, b^2, \cdots)$. Gradient decent is than given by

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

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 than made using only the samples from one mini-batch. This alteration of gradient decent goes by the name of *stochastic gradient decent* (SGD). The larger the mini-batch, the more accurate the estimate of the gradient and therefore the 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 $\mathcal{L}^1, \cdots, \mathcal{L}^n$ with weights $W^1, \cdots, W^n$, biases $\vec{b}^1, \cdots, \vec{b}^n$ and activation functions $a_1, \cdots, a_n$. 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 $\nabla_{W^1} L, \cdots, \nabla_{W^n} L$ and $\nabla_{\vec{b}^1} L, \cdots, \nabla_{\vec{b}^n} L$, where

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

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

$$\nabla_{\vec{b}^i} L := \begin{pmatrix} \partial_{b^i_1} L \\ \vdots \\ \partial_{b^i_k} L \end{pmatrix}, \tag{3.11}$$

for $\vec{b}^i \in \mathbb{R}^k$.
To calculate $\partial_{W^i_{jk}} L$ and $\partial_{b^i_j} L$, define

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

such that

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

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

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

To start off derive this loss by the parameter θ_j

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

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

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

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

$$\partial_{\theta_j} a_n(z_n) = \Big(\partial_{\theta_j} z_n\Big) \odot \partial_z a_n(z)\big|_{z=z_n}, \qquad (3.17)$$

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.

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

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

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

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)

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

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

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

With this definition equation (3.19) can be written as

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

and the recursion can be solved to yield

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

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

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

From this the analytic form can be computed to be

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

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. This is done to some extend in practice. An entire pass of the training set is called an *epoch*. In theory, 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 the model 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 $m \geq n$ as a model 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 $m + 1$ free parameters. As $m \geq n$ 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 the fitted model is evaluated on a point, that is not part of the set of points, that was used to generate the fit. 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 than 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 most easy 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 loss value of the validation set, 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 training[2], come from a third independent set. This set is than called the test set.

A general approach to increase the performance of a network is to increase its depth. This is due to two reasons. First of all it has been shown, that a deep network can separate the underlying function it is trying to learn into piecewise linear regions. The number of these regions is than exponential in the depth of the network. Secondly each layer can be viewed as a filter, that looks for certain features in the data. Having multiple stacked layers will enable the network to learn low level features on early layers and combine them into more difficult features on lower layers. (Section 6.4.1 [2]) This idea will be expanded upon in the following subsection 3.4. The depth is not the only parameter of a simple network, that can be scaled to increase performance. A systematic study can be found for instance in [3].
A common problem, that arises when training very deep NNs is the vanishing gradient problem; the gradient calculated by (3.23) or (3.25) is close to zero on early layers, which results in these layers not changing their weights enough. The reason for this behavior can also be understood from (3.23) and (3.25). If the products satisfy $|\partial_z a_i(z) \cdot W^i| < 1$ the gradient is exponentially diminished by the depth of the network.
The opposite can also happen. If $|\partial_z a_i(z) \cdot W^i| > 1$ for most of the layers, the gradient will grow exponentially. This behavior is therefore called the exploding gradient problem. (Chapter 5 [1])
To overcome these problems, one can simply train for longer periods in the case of the vanishing gradient problem (Chapter 5 [1]), use multiple points at which the loss is calculated [4] or adjust the initialization of the weights. (Any references? Also introduce residual connections here? Need to speak about what an epoch is.)

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. (See Figure 3.5) These fully connected layers are called *dense* layers. 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.

3.4.1 Convolution layer

What are the advantages of convolution layers and why do we use them? Disadvantages?
Dense layers have been the starting point for deep NNs and are used to derive a lot of the theory. In subsection 3.3 it was stated, that deeper networks usually perform better. This is however a problem for NNs that consist only of dense layers, as the number of trainable parameters grows exponentially [Citation, can't find one], if the layer size is kept constant between two layers. This causes computational limits that limit the depth. Another problem of fully connected layers is that they are rather static. To understand

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

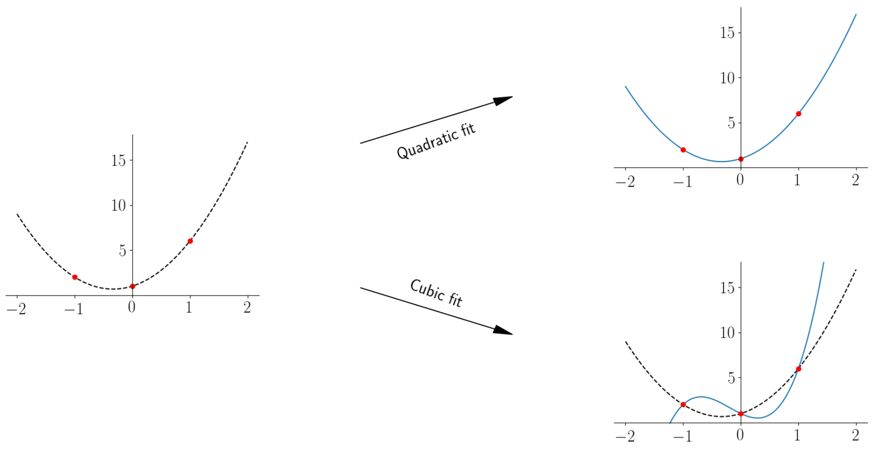


Figure 3.4: Depiction of overfitting in the classical regression. On the left the originally sampled function $f(x) \coloneqq 3x^2 + 2x + 1$ is shown in dashed and black. The red dots are the samples that are being used for the regression on the right. The top right shows the regression, where $g(x) = ax^2 + bx + c$ was used as a basis and recovered the correct parameters $a = 3$, $b = 2$ and $c = 1$. All free parameters are fixed by the data. The lower right plot shows a case of overfitting. The same three points are now used to fit the four free 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 and the MSE would be non-zero.

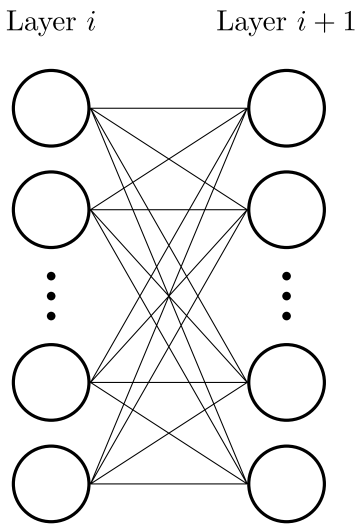


Figure 3.5: Insert description! Maybe inprove this graphic.

this, consider a network that learns to distinguish between cats and dogs. Say, that for the training set all animals are in the lower left hand corner of the image. The validation set than might have the animals not in the lower left, but the upper right corner. A NN consisting purely of dense layers might not be able to adapt to this new position. Even if there are animals in the top right corner for the training set, the network might need a lot of layers and trainable parameters to learn all possible positions.

These restrictions led to the invention of the convolution layer in 1989. [5] Though it was originally conceived in its 2 dimensional variant, only the 1 dimensional convolution layer will be explained here.[3] Contrary to dense layers, the neurons of convolution layers are only connected to a few neurons on the previous layer. Furthermore these connections all share the same weight. Thus one can view a convolution layer as a filter of learnable weights that slide across the input, in a way convolving the the filter with the input data. (see Figure 3.6)

The size of the filter, i.e. how many entries it spans, is called the *kernel size*. If multiple convolution layers with the same kernel size are stacked, the number of trainable parameters increases only linearly with the depth, which is a huge improvement over dense layers.

The capacity of a convolution layer however is not only governed by the kernel size, but also by how many filters are run over the same data in parallel. If a convolution layer runs only a single filter over the data, it might learn to detect a single feature, like a vertical line. Therefore multiple filters, that have different weights, are usually run over the same input data. Each of these different filters will than be able to detect different features. The output these layers produce are called *feature maps*.

Having multiple filters however changes the shape of the output from being 1 dimensional, with just a single filter, to being 2 dimensional with multiple filters. The data each filter outputs is still 1 dimensional and called a *channel* of the final output. To be able to stack convolution layers, they take in a 2 dimensional input and are specified by the number of filters and the kernel size of all these filters[4]. Each filter, that is convolved with the data, spans all channels. The kernel size only specifies how many entries in each channel are used. (see Figure 3.7) This paragraph is hard to read and understand.

As an example say we specify a convolution layer by having 32 filters and a kernel size of 3. Now we use this convolution layer on two different inputs. Input 1 has a shape of 4096×1 and input 2 has a shape of 4096×2. Notice, that input 1 in principle is still 1 dimensional, as it only has one channel. The data still has to be reshaped though to work with the general concept. For input 1, the filter would be of shape 3×1 and the output shape of the convolution layer would be 4094×32. Therefore the convolution layer would have $3 \cdot 1 \cdot 32 = 96$ trainable parameters. For input 2, the filter would need to span both channels and thus has the shape 3×2, the output shape however is still 4094×32. The number of trainable parameters however also doubles to $3 \cdot 2 \cdot 32 = 192$. All of the above disregarded possible bias-values.

Another advantage of the convolution layer are the shared weights. Shared weights

[3]The core concepts are the same, thus the concept can easily be adapted to any number of dimensions.

[4]Usually all filters have the same kernel size.

means, that the value of two output neurons in the same channel only depends on the different input values, as the weights of the filter are the same for both of them. This being an advantage becomes clear, when considering the example from above, where a NN tried to distinguish between cats and dogs. For a convolution layer the position of the animals is not of importance. If it developed a filter that can recognize cats or dogs, it will be able to find them regardless of where in the image they are positioned.
This behavior of the convolution layer is of special importance to our work, as it gives us time invariance. If the network learns to categorize the signals correctly, it does not really matter where in the data that signal is.
In principle convolution networks can even work on data without a predefined length, as the filters are simply shifted across the data. This behavior is however lost, when dense layers are introduced into a convolution network.

Having sparse connections in the convolution layers also leads to stacked convolution layers having a *receptive field*. The receptive field of one output of a convolution layer is the number neurons on the input layer that have, through some path, an influence on its value. (see Figure 3.8)

Though convolution layers are quite different to dense layers, their training can still be easily described by the formalism developed in subsection 3.2. The operations for a single filter can be expressed by using a sparse matrix and multiplying it by the input. For multiple filters, i.e. more output channels, this formalism just has to be extended to tensors.
A single filter F of size n has weights $\vec{w} = (w_1, \dots, w_n)^T$. When applied to an input $\vec{x}$ of length $m > n$, the output has the length $m - n + 1$. Denote the convolution operation by $*$. The output is thus given by

$$[\vec{x} * F]_j = a\left(\left(\sum_{k=0}^{n-1} w_{k+1} \cdot x_{j+k}\right) + b\right), \tag{3.26}$$

where b is the bias and a is the activation function of the layer. This can be rewritten as a matrix product

$$\sum_{j=1}^{m} [\vec{x} * F]_j \cdot \vec{e}_j = a\Big(W \cdot \vec{x} + \vec{b}\Big), \tag{3.27}$$

where $\vec{e}_j$ is the j-th standard basis vector and the filter $(m - n + 1) \times m$-matrix W is given by

$$W = \begin{pmatrix} w_1 & \dots & w_n & & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & & w_1 & \dots & w_n \end{pmatrix}. \tag{3.28}$$

The backpropagation algorithm than only needs to know about the gradient of W with respect to the weights $\vec{w}$.

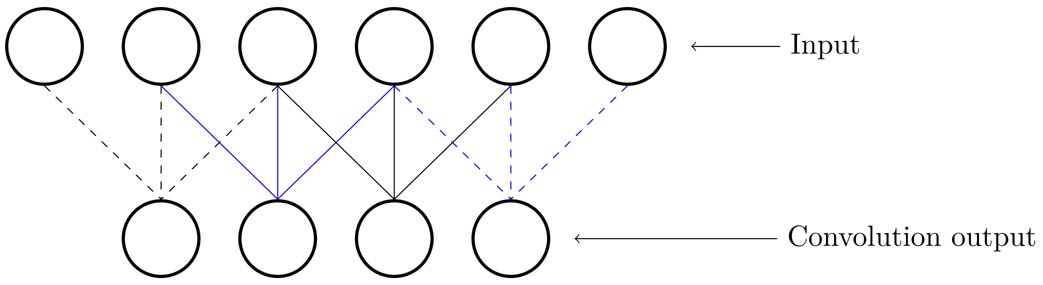


Figure 3.6: Example of a convolution layer with a kernel size of 3. It highlights the sparse connectivity of the different neurons. Each of the output neurons is now only connected to three of the previous neurons. Each of the weights associated with the lines in the picture above is shared, i.e. the leftmost line (independant of its style and color) always represents the same weight. The same is true for the middle and right line in each of the groups. The size of the output is reduced due to the filter having a kernel size > 1.

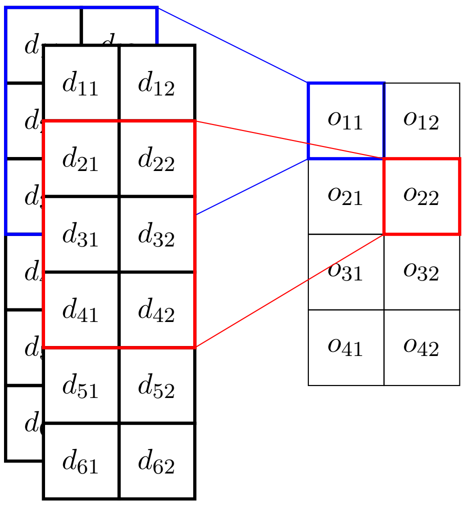


Figure 3.7: Depiction of a convolution layer with two filters, that have a kernel size of 3. The input data d_{11} to d_{62} has two channels and is the same for both the blue and the red filter. The first channel of the output (o_{11} to o_{41}) is produced by sliding the blue filter over the data, the second channel (o_{12} to o_{42}) is produced by sliding the red filter over the data. Notice that the filters span all channels of the input data and only slide in 1 dimension.

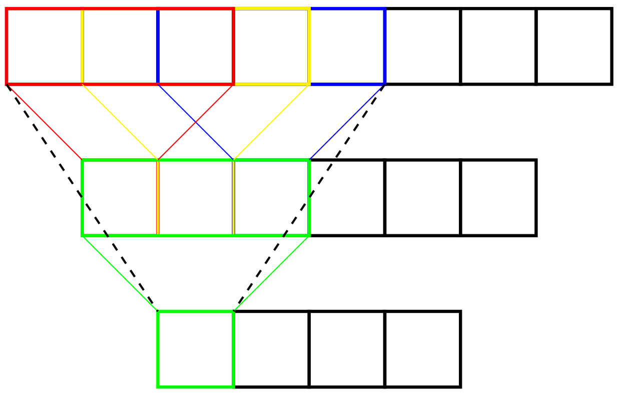


Figure 3.8: Three stacked convolution layers. Although all layers have a kernel size of 3, the final layer is influenced by 5 of the input values. Therefore the receptive field of the final layer is 5. Maybe change the colors and some presentation of this graphic. I don't like how it looks.

3.4.2 Pooling layers

Explain what max pooling does and why it is useful, even when it is counter intuitive.
Pooling layers are another special kind of layers, often used to increase performance of CNNs. Though there are many variations of the specific implementation, the core concept is grouping multiple activations of a single feature map into one activation. The most common pooling layer is the maximum pooling layer, as it puts greater emphasis on strong activations. [6] It works by grouping a certain number of input activations of the previous layer and assigning this group the maximum values of all the grouped neurons. (see Figure 3.9)
Though it does seems counter intuitive, that throwing away information helps the networks performance, the reasons are manifold. First of all pooling in general downsamples the data[5]. The lower number of datapoints results in fewer calculations per forward pass, fewer trainable parameters being used and thus less overfitting. Secondly, maximum pooling increases the impact of strong activations. These strong activations usually come from the parts of the data that resonate strongly with a convolution filter. If this resonance only applies for a small region in the data, there will only be few values on the feature map, that correspond to this resonance. Therefore pooling (in general) leads to greater spatial invariance. The downside of pooling is the loss of positional information. As a rule of thumb, pooling is useful for a decision "is a feature present", but falls short if the question "where in the data is the feature present" is also relevant.
Due to its improvement in spatial invariance, pooling also leads to the next layer having a greater receptive field.

All the actions described above act only on a single feature map, channel by channel. A similar approach can however also be taken for the channels themselves. Such a procedure is called *dimensional reduction* and usually done through a convolution layer with

[5]There have been studies suggesting, that pooling works better than simply sub sampling the data. [7]

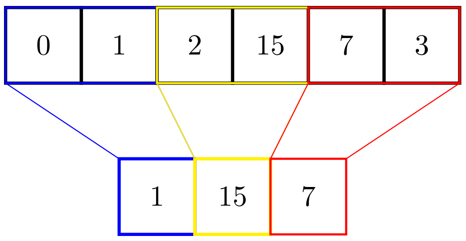


Figure 3.9: An example of a Max pooling layer. It groups together two entries of its input and returns the maximum, thus halving the number of samples per feature map. This process is applied for all channels.

a kernel size of 1. This way, all channels are being connected through a weighted sum, where the weights are learned.[8] Additionally a activation function can be used to introduce non-linear combinations of the channels. In this way, dimensional reduction can not only be seen as a reduction in learnable parameters, but also as means to combine features of different channels. [9] As the different feature maps are added together, this operation combines low level features into higher level ones.
The number of outgoing channels is given by the number of filters used in the convolution layer with kernel size 1.

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, 10, 11] 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. Accordingly the module showcased in this section was developed for image classification and introduced in [4].
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. [4] Another way to increase the capactity 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 capabiltiy to detect larger features within a certain part of the image. If they are too large however, the filter might be close to zero for a lot of the learnable parameters, 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 outweight by the computational overhead created. Therefore sparse matrix operations are not feasible at the moment. [4]
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 than be performed efficiently on the sparse matrices, by utilizing the efficient dense operations on the dense submatrices. This is the approach, the inception modules tries 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 ILSVRC[6] 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. [4, 11]
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. [12, 13]

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 3.10.
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 a 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 subsubsection 3.4.2 is added as a fourth path. 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.

[6]The 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.

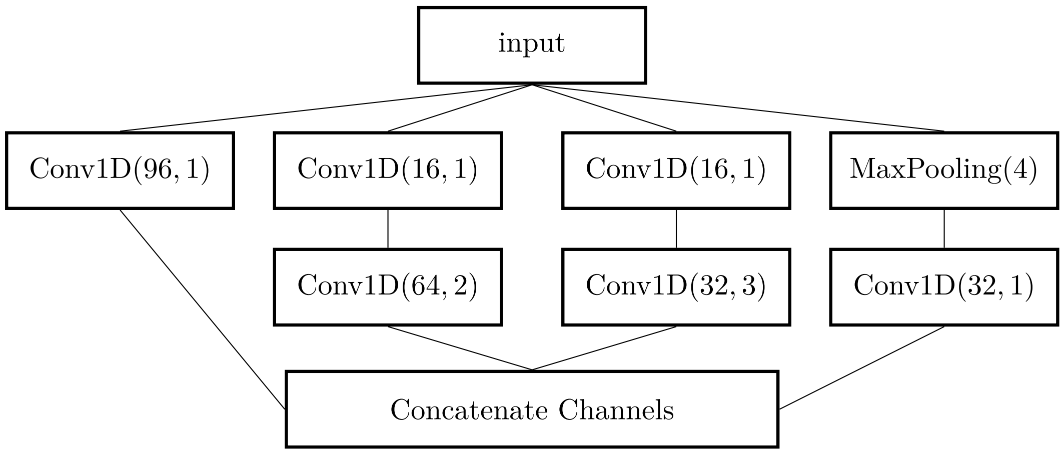
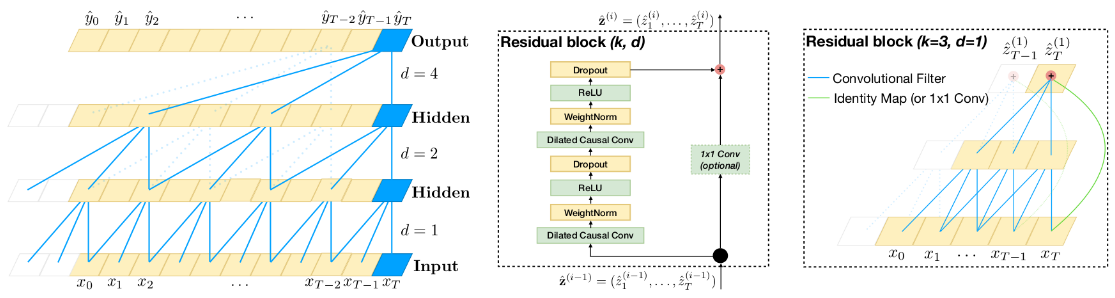


Figure 3.10: Shown are the contents and connections of the inception module as used in this work. (If the filter numbers and values change for the final architecture, change them here too.) The layer Conv1D(x, y) is a 1 dimensional convolution layer with x filters and a kernel size of y. Most of the 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 output of each of these paths is than 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. Temporal convolutional networks (TCN), as used in this work, were proposed by [14]. Their research suggests that this specialized CNN-architecture outperforms RNNs, which were previously the norm for analyzing and processing sequence data.
A TCN basically consists of multiple stacked convolution layers, that are slightly adapted in two different ways. For once, the filters are diluted, meaning, that the weighted sum of the convolution is not taken over successive input points. Instead the weighted sum uses points, skipping a set number of inputs in between. Secondly the connections are causal. This means, that output y_i of the filter depends at most only on points $x_i, \dots, x_1$. Finally the input of each such layer is padded with zeros such, that the output matches the size of the input. (see (a) of Figure 3.11)
The advantage of the dilated convolution layers is, that the receptive field of the network grows exponentially with the depth, whereas without the dilution this growth is only linear. [Citation] The goal of the TCN is to have a receptive field, that spans the entire input length. This still requires a decently deep network for inputs of considerable length. To combat the problem of the vanishing gradient, residual connections are also



(a) (b) (c)

Figure 3.11: Architectural elements in a TCN. Graphic taken from [14]. (a) Exponential increase of the receptive field in diluted convolution layers. Here the dilution factor d scales as 2^i and the kernel size k is set to 3. The causal structure propagates through the layers, as no output is connected to an earlier input. (b) Multiple different layers are utilized for an entire module of the TCN. Also a residual connection is used, to help earlier levels learn. Multiple of these units are stacked to form a TCN. (c) An example of how the residual block from (b) could look like.

introduced. The dimensional reduction layer that is part of the residual connection is simply used to adjust the number of channels to be able to add the input and the output of the residual block together. The full structure of the residual block is shown in (b) of Figure 3.11. The only adaption to the implementation that this work makes is the the replacement of the WeightNorm layer with a traditional BatchNormalization layer, as it is described in subsubsection 3.5.1. This is done for convenience, as there is a pre-implemented version of batch normalization in the software library used. This replacement is valid, as weight normalization is described by the authors to be largely a fast approximation to full batch normalization. [15]

3.5 Regularization

As [2] put it: "Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.". There are many ways, like reducing the number of trainable parameters or adjusting the loss function to prefer specific weights, to achieve this goal. This section however will only introduce two specialized methods, that are introduced as a layer into the network.

3.5.1 Batch Normalization layer

Explain how batch normalization works and why it is useful. (cite according paper)
Batch normalization was introduced in [16] and is used to normalize the inputs of each layer. This helps the network learn faster and generalize more easily.
The normalization tries to fix the distribution of the inputs between different samples, as the layers would need to adapt their weights otherwise for different input distributions. Specifically the goal is to transform the data in a way, that the mean is 0 and the variance is 1. Normalizing data in such a way is in principle not problematic and a standard procedure only for the input layer. [Citation] The problem of normalizing the

input of each individual layer is the backporpagation step, as it can lead to exploding biases. [16]
To solve this issue, gradients of the normalization with respect to multiple inputs need to be computed. Batch normalization uses the samples of each mini-batch to compute the mean, variance and gradients. To reduce computational cost, the normalization is computed for each individual activation and not over all activations. Finally a linear transformation

$$y_i = \gamma \hat{x}_i + \beta \tag{3.29}$$

is applied to the normalized data

$$\hat{x}_i = \frac{x_i - \mu_B}{{\sigma_B}^2 + \varepsilon}. \tag{3.30}$$

Here x_i is the i-th sample of the mini-batch, μ_B is the mean and ${\sigma_B}^2$ is the variance of the activation calculated over the mini-batch. The factors β and γ are learned parameters and ϵ is a constant added for numerical stability.
The linear transformation is applied so that the batch normalization layer can learn to be the identity transformation. Otherwise the normalization could loose the ability to represent some function it previously could have represented.

3.5.2 Dropout layer

Explain what a dropout layer is, what it does, why it is useful.
Dropout layers were introduced in 2014 by [17] and showed great improvements to lowering the generalization error. It works by dropping a random percentage of neurons from the network during training. Though this approach sounds counter intuitive, it has multiple benefits.
One viewpoint is using the dropout layer as a noise source for the network. By dropping some activations during training, the network can't be too strongly dependent on a single connection and has to learn multiple ways of detecting some feature. Therefore the network becomes less sensitive to small alterations of the input. Furthermore the dropout layer can be used as a first layer in a network and act as data augmentation, where it introduces further noise to the data, as the same sample may experience different dropped connections. Therefore the effective number of samples the network sees during training is enlarged.
Another viewpoint is, that training a network with dropout layers does not only tryout the full architecture, but also all sub-networks, that can be created from the full architecture by dropping some connections. The number of sub-networks grows exponentially with the number of dropout layers. This viewpoint is the main selling point promoted by [2] and [17], as it allows to efficiently sample many networks and combine them.
During the evaluation process, the original paper [17] suggests reweighing the weights by the dropout probability, to get an averaging effect. This step is however not done in the software library Keras used in this work. [Citation]

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 will be defined here. 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 neuron equivalent to 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.

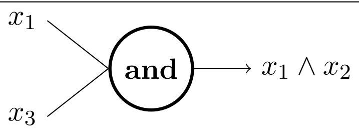
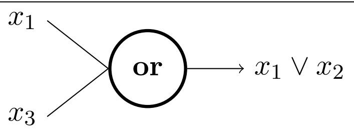
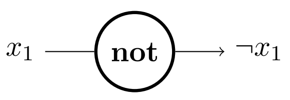
”and”-neuron			”or”-neuron			”not”-neuron	
							
$\vec{w} = (1, 1)$ $b = -1.5$			$\vec{w} = (1, 1)$ $b = -0.5$			$w = -1$ $b = 0.5$	
x_1	x_2	$a(x_1 + x_2 - 1.5)$	x_1	x_2	$a(x_1 + x_2 - 0.5)$	x_1	$a(-x_1 + 0.5)$
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

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.

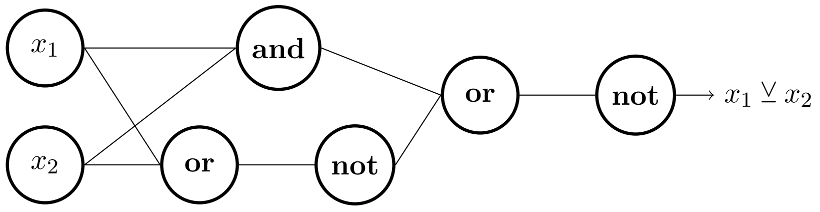


Figure A.1: The definition of a network that is equivalent to an "XOR"-gate.

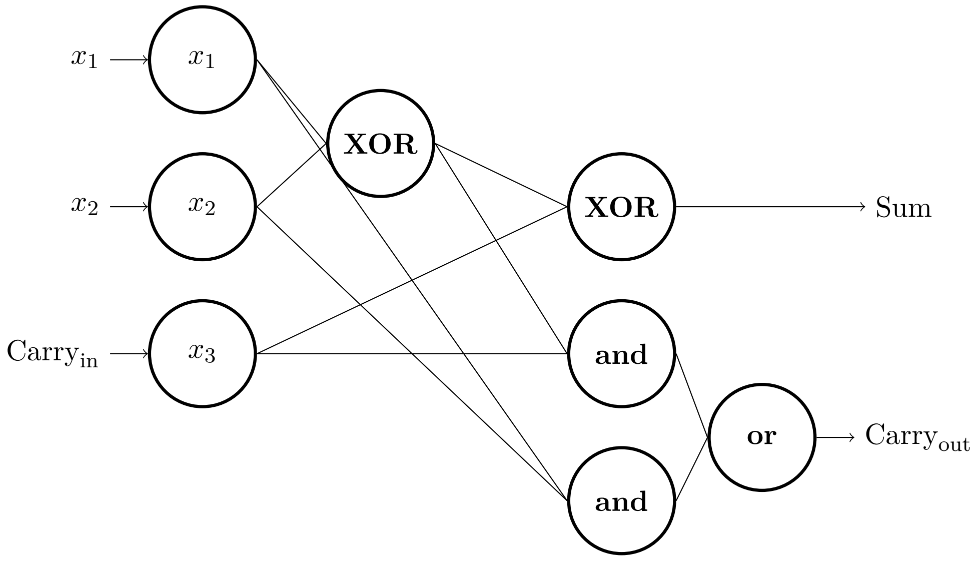


Figure A.2: A network replicating the behavior of a binary full adder.

B Indication that the network does not learn

Sometimes during training only a local minimum is found. The most notable of these minima 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. 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. Furthermore it is assumed, 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

$$f(x) := \frac{1}{b-a}\int_a^b \mathrm{d}y\ (x-y)^2 = \frac{1}{3(b-a)}\Big((b-x)^3 - (a-x)^3\Big). \tag{B.1}$$

To minimize this one could in principle solve $\frac{\partial f}{\partial x} \overset{!}{=} 0$ 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 the training finds a local minimum that leads to the network always returning a fixed value, this value should be

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

In this work another common case is the network having to choose a SNR-value from some continues 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. (The following 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

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

where the step in $(*)$ is not trivial 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

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

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 the total number of data-points.

C Deriving custom loss

For this work the binary decision of "signal" vs. "no-signal" is more important for the performance of the network, than how accurate the predicted SNR-value is. Therefore we would like the network to have a bias towards underestimating the SNR-values of pure noise examples and overestimate those of GW-signals. To achieve this behavior, we tried to use a new loss function, that exponentially penalizes overestimating pure noise samples and underestimating GW-signals. For backpropagation to work properly, the loss will need to be differentiable and its derivative needs to be continuous everywhere. As a starting point we will use the pure noise case first and adapt it to the full loss later on. We start at a distribution of values we want to achieve and later turn it into an error function. This distribution should exponentially decay for values larger than some fixed value and decay like $1/x$ for values smaller than this fixed value. The exponential part was inspired by the solution of the hydrogen atom, though the decay for this case goes like x^2. For this reason, we matched

$$f_1(x) = x^2 e^{-x} \tag{C.1}$$

$$f_2(x) = \frac{1}{a - b \cdot x} \tag{C.2}$$

for $x = 1$, which gave

$$f(x) := \begin{cases} x^2 e^{-x}, & x \geq 1 \\ \frac{1}{e}\frac{1}{2-x}, & x < 1 \end{cases}. \tag{C.3}$$

This distribution has its maximum value $\frac{4}{e^2}$ at $x = 2$. For convenience, we will use

$$\text{dist}(x) := f(x+2) \tag{C.4}$$

from here on out, as the maximum is now centered at $x = 0$. To get an error function from this distribution, that grows exponentially for $x > 0$ and has a value of 0 for $x = 0$, define

$$\text{Err}(x) := \frac{4}{e^2 \text{dist}(x)} - 1. \tag{C.5}$$

To define a loss, that can behave differently for different label values, the error needs to transform based on some measure. Specifically it will need to have some transition between exponential growth for large values of x and exponential growth for small values of x. To achieve this behavior, we rotate the error-function Err around the y-axis and project it onto the x-y-plane afterwards. Therefore we get

$$\text{Err}_{\text{rotate}}(x, \varphi) := \text{Err}(x/\cos(\varphi)). \tag{C.6}$$

For $\varphi \in [0, \pi]$ this function is defined everywhere but $\varphi = \frac{\pi}{2}$.
To get a loss as defined in (3.7) from (C.6), it needs to depend not only on the value the network returns but also on the label. Furthermore, the exponential behavior should be governed by the label value, as we want exponential growth for positive differences,

when the label is small, and for negative differences, when the label is large. To achieve this, the rotation angle will be dictated by the label value.
Therefore we want a function g, that is 0 for all label values smaller than some minimum a and π for all label values larger than some maximum value b. In between a and b, the function needs to be a smooth. The parts will be matched in a way to assure $g \in C^1(\mathbb{R})$. With this one can find

$$g(x,a,b) := \begin{cases} 0, & x < a \\ \pi, & x > b \\ \pi p\left(\frac{x-a}{b-a}\right) & \end{cases}, \tag{C.7}$$

with

$$p(x) := 3x^2 - 2x^3. \tag{C.8}$$

In principle the loss for given values a and b could than be written as

$$L_{\text{exp}}(y_{\text{net}}, y_{\text{label}}) = \text{Err}_{\text{rotate}}(z, g(y_{\text{label}}, a, b)), \tag{C.9}$$

with $z = y_{\text{net}} - y_{\text{label}}$. There are however problems with this definition. First of all, the exponential part is smaller than the linear part for small values of z. This is especially true for values $|z| < 2$. This is a problem, as the SNR-value assigned to pure noise and the smallest signal SNR-value are about 4 apart. Therefore there would be an overlapping region for pure noise and small SNR signals, that is favored by the loss.
To solve this issue one can simply introduce a squish factor s, which the input to the error is multiplied by

$$L_{\text{squish}}(y_{\text{net}}, y_{\text{label}}) := \text{Err}_{\text{rotate}}(s \cdot z, g(y_{\text{label}}, a, b)). \tag{C.10}$$

This however introduces a new problem. With even a relatively small squish factor of $s = 3$, the exponential grows very fast, which causes exploding gradients. To keep the gradients at bay, a cutoff is introduced to the function. To still have a non zero gradient, this cutoff is not flat, but is a linear function with a slope of $\pm s\frac{4}{e}$, which is the same slope as the linear part of (C.10). To keep the entire loss of class C^1, the exponential part and the cutoff are connected by a spline polynomial. For this, we chose to start the spline polynomial at some value $k-1$ from the origin and have it connect to the linear part in a distance of 1.
With

$$u = \max\left[\text{Err}_{\text{rotate}}(s \cdot z + k, g(y_{\text{label}}, a, b)),\ \text{Err}_{\text{rotate}}(s \cdot z + k - 1, g(y_{\text{label}}, a, b))\right]$$

$$l = \min\left[\text{Err}_{\text{rotate}}(s \cdot z + k, g(y_{\text{label}}, a, b)),\ \text{Err}_{\text{rotate}}(s \cdot z + k - 1, g(y_{\text{label}}, a, b))\right]$$

$$\Delta_1 = \begin{cases} \frac{-4s^2 k}{2 - s \cdot k^3} e^{-s \cdot k}, & y_{\text{label}} > \frac{a+b}{2} \\ \frac{4s^2(k-1)}{2 + s \cdot (k-1)^3} e^{s \cdot (k-1)}, & y_{\text{label}} < \frac{a+b}{2} \end{cases}$$

$$\Delta_2 = \begin{cases} -s\frac{4}{e}, & y_{\text{label}} > \frac{a+b}{2} \\ s\frac{4}{e}, & y_{\text{label}} < \frac{a+b}{2} \end{cases} \tag{C.11}$$

$$a_1 = \Delta_2 - \Delta_1 - 2(u - l) \tag{C.12}$$

$$a_2 = 3(u - l) - 2\Delta_1 - \Delta_2 \tag{C.13}$$

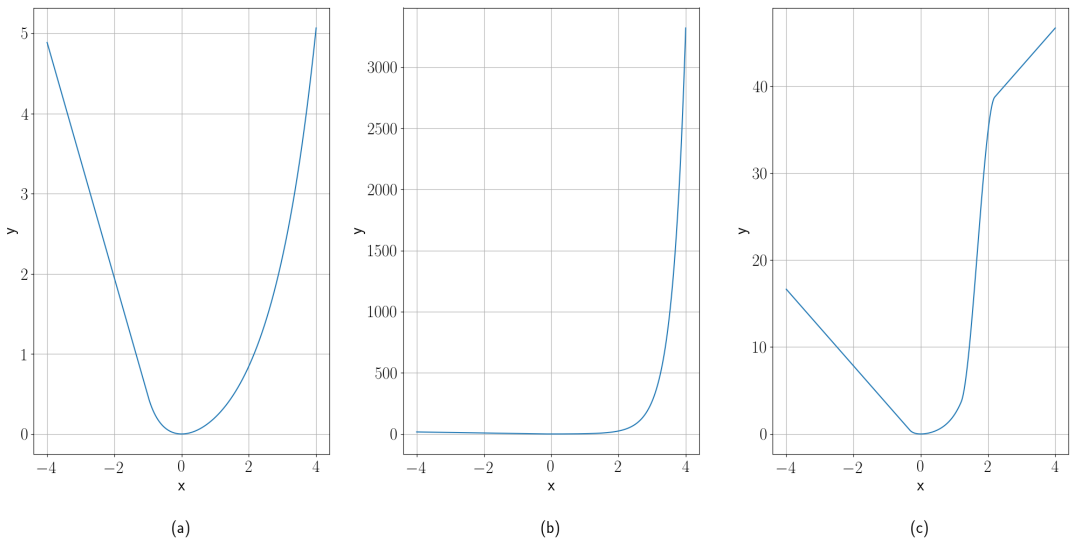


Figure C.1: Three different stages of the custom loss function. (a): The loss as defined in (C.9). For $|x| < 2$ the eyponential part is actually smaller than the linear part. This is not desirable, as for most of our testing the label for pure noise is about 4 away from the smallest label for a signal. (b): To fix the issue of a too small exponential for some purposes, one can introduce a squish factor, that simply multiplies the input by some fixed value. In this case a squish factor of 3 was used. The values in general are a lot larger. (c): Having a large squish factor as in (b) introduces the problem of too large gradients. For this purpose, a cutoff can be introduced. This cutoff grows linearly with the same slope as the linear part of (C.10). The transition between the exponential and linear part however needs to be of class C^1, so that the backpropagation algorithm can optimize. Therefore a spline polynomial connects the two parts.

and

$$p_2(z) \coloneqq a_1(z-k)^3 + a_2(z-k)^2 + \Delta_1(z-k) + l \tag{C.14}$$

one gets

$$L_{\text{full}}(y_{\text{net}}, y_{\text{label}}) \coloneqq \Big\{ \text{Err}_{\text{rotate}}(s \cdot z, g(y_{\text{label}}, a, b)), \quad z > 1 \land y_{\text{label}} > \tfrac{a+b}{2} \tag{C.15}$$

Fix this equation

Glossary

BBH Binary black hole.

BNS Binary neutron star.

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.

RNNs recurrent neural networks.

SGD Stochastic gradient descent.

SNR signal to noise ratio.

TCN Temporal convolutional network.

TT transversal-traceless gauge.

References

[1] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/index.html (cit. on pp. 4, 8, 14).

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org (cit. on pp. 4, 6, 7, 8, 9, 12, 13, 14, 20, 23, 24).

[3] Mingxing Tan and Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *CoRR* abs/1905.11946 (2019). arXiv: 1905.11946. URL: http://arxiv.org/abs/1905.11946 (cit. on p. 14).

[4] 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, 20, 21).

[5] Yann LeCun et al. "Generalization and network design strategies". In: *Connectionism in perspective*. Vol. 19. Citeseer, 1989. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.476.479&rep=rep1&type=pdf (cit. on p. 16).

[6] Zhou and Chellappa. "Computation of optical flow using a neural network". In: *IEEE 1988 International Conference on Neural Networks*. July 1988, 71–78 vol.2. DOI: 10.1109/ICNN.1988.23914 (cit. on p. 19).

[7] Dominik Scherer, Andreas Müller, and Sven Behnke. "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition". In: *Artificial Neural Networks – ICANN 2010*. Ed. by Konstantinos Diamantaras, Wlodek Duch, and Lazaros S. Iliadis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–101. ISBN: 978-3-642-15825-4. URL: https://link.springer.com/chapter/10.1007/978-3-642-15825-4_10 (cit. on p. 19).

[8] Min Lin, Qiang Chen, and Shuicheng Yan. "Network in network". In: *arXiv preprint arXiv:1312.4400* (2013). arXiv: 1312.4400 [cs]. URL: https://arxiv.org/abs/1312.4400 (cit. on p. 20).

[9] Yuan Gao et al. "NDDR-CNN: Layerwise Feature Fusing in Multi-Task CNNs by Neural Discriminative Dimensionality Reduction". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019, pp. 3205–3214. URL: http://openaccess.thecvf.com/content_CVPR_2019/html/Gao_NDDR-CNN_Layerwise_Feature_Fusing_in_Multi-Task_CNNs_by_Neural_Discriminative_CVPR_2019_paper.html (cit. on p. 20).

[10] 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. 20).

[11] 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 (cit. on pp. 20, 21).

[12] 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. 21).

[13] 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. 21).

[14] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling". In: *CoRR* abs/1803.01271 (2018). arXiv: 1803.01271. URL: http://arxiv.org/abs/1803.01271 (cit. on pp. 22, 23).

[15] Tim Salimans and Durk P Kingma. "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks". In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 901–909. URL: http://papers.nips.cc/paper/6114-weight-normalization-a-simple-reparameterization-to-accelerate-training-of-deep-neural-networks.pdf (cit. on p. 23).

[16] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167 (cit. on pp. 23, 24).

[17] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958. URL: http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf (cit. on p. 24).