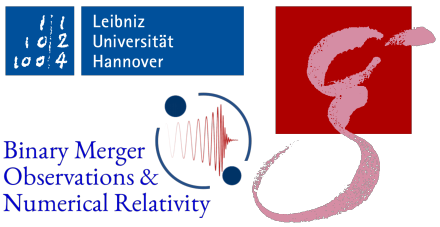


Leibniz Universität Hannover
and
Max Planck Institute for Gravitational
Physics (Albert Einstein Institute)

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

September 8, 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.

1 Introduction

With the first direct detection of a gravitational wave (GW) on September the 14th 2015 [1] the age of gravitational wave astronomy began. It opened up the possibilities to test Einstein's theory of gravity in highly relativistic systems [2], sample the population of compact binary systems consisting of objects like neutron stars or black holes [3], define new astronomical standard candles [4] and many more. The first and second observation run of the advanced LIGO and Virgo detectors [5, 6] led to 11 detections of GWs from different systems [7]. The third observation run, which is currently ongoing, promises to greatly expand this catalog and has already found multiple GW candidates.
The most promising source of GWs are binary systems consisting of two black holes, neutron stars or a mix of these two. So far all confirmed detections of GWs were caused by such compact binary systems. Most of them were generated by two coalescing black holes, the only exception being GW170817 [7]. The latter, instead, was emitted by a binary neutron star (BNS) system [8]. As such it was one of the most interesting signals detected so far. Not only is it the lone signal of its kind observed yet, but we were also able to detect an electromagnetic (EM) counterpart. This allowed to localize the source very precisely and get a detailed frequency evolution of the EM radiation emitted, thus helping to understand the inert dynamics of neutron stars.
To detect an EM counterpart astronomers need to be alerted quickly when the detector registers a possible BNS signal. To put the time scales involved into perspective, the γ-ray burst detected by Fermi-GBM arrived only 1.7 s after the GW [8]. To reduce latency as much as possible and maximize observation time the detection pipeline needs to generate reliable triggers in as close to real time as possible. Alongside the trigger itself an estimate of the sky position needs to be provided as well.
The current pipeline uses the concept of matched filtering where a fixed number of precalculated GW templates are used to search for similar patterns in the detector data. [Citation] These templates cover some area of the parameter space of binary systems we expect to detect. The sensitivity, however, is directly connected to the spacing of templates in this high dimensional parameter space. As our knowledge about binary systems and their dynamics improves and as we develop ever more accurate waveform models, this template bank will grow in size. This is especially true when detectors with greater sensitivity are considered. The downside of an increased size of the template bank is the computational cost associated with it. The CPU time scales directly with the number of waveforms that need to be compared with the data. Therefore, in the future it might not be feasible to use matched filtering under consideration of the full template bank as the only trigger generator. (Are there other trigger generators in use already?)
One of the possible contenders to aid matched filtering in the first data analysis stage is machine learning, a field of computer science that aims to create computer programs which adapt to a problem without direct human interference, i.e. learning from a set of experiences. Most of today's state of the art machine learning algorithms are implementations of neural networks (NNs). They found application in many fields, like computer

vision [9], sound generation or natural language processing [10]. The advantages of NNs are manifold, one of them being computational efficiency once the network is optimized. This and their general success in almost any area makes NNs interesting for GW data-analysis.
Daniel George and E.A. Huerta were the first to apply a deep NN to whitened time series strain data to try and recover GW signals. They were able to reach performances comparable to those of matched filtering at a fraction of the computational cost [11]. Their network, however, was only optimized for signals from binary black hole BBH mergers, thus not covering the cases where quick notifications are most valuable.
This thesis builds on the work of [11] and tries to expand it to BNS signals. Detecting GWs from two coalescing neutron stars using a NN is more challenging as these signals tend to be weaker, contain higher frequencies and are within the sensitive frequency-region of the detectors for longer time periods. Thus we introduce a novel approach to handle longer time series by using multiple sample rates. To get a first hold of the problem we will be ignoring spins and tidal deformabilities of the neutron stars. Our algorithm will be able to take a continuous stretch of strain amplitude time series data and generate from it two output time series; a signal-to-noise ratio (SNR) time series and p-score[1] time-series. To both of these a threshold at fixed false-alarm rate will be applied to generate triggers.

This thesis is structured as follows. Section 2 and section 3 will give a brief summary of the required background knowledge. Basic knowledge of general relativity and linear algebra are assumed. Section 4 will give a deeper motivation to the problem we are trying to analyze and puts this thesis into greater context of related works. Section 5 contains our research and will go into detail about the design decisions that went into our final network. It will furthermore outline the design process and evaluate the resulting architecture on long stretches of data to compare this novel approach with existing methods.

To design and train our networks we use version 2.2.4 of the software library Keras [12]. The former is a wrapper for the deep learning library Tensorflow [13], of which we use the GPU optimized version 1.13.1 for training. To evaluate our networks we use version 1.14.0 of the CPU based implementation of Tensorflow. To generate fake data we use version 1.13.5 of the software package PyCBC [14]. (Remove later quotation of the same). All code related to this thesis is open source and can be found at `https://github.com/MarlinSchaefer/master_project`.

[1]The p-score must not be confused with a p-value. Both have in common that they are normalized to 1. The p-score however does not fulfill any other requirements and is thus not a probability.

2 Gravitational-Wave Signals from Binary Neutron Star Mergers

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, 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. Need to specify, that I use Einstein sum convention in this section and that latin indices are spacial indices, whereas greek indices are over all four components. NEED TO CHANGE MOST CITATIONS FROM BACHELOR THESIS TO ORIGINAL SOURCES! Mention bachelor thesis only as a way to look up detailed calculations. Need to specify which convention is used for $\eta^{\mu\nu}$. Look for "energy-momentum-tensor" and replace by "energy-momentum tensor". Search for "chirp-mass" and replace by "chirp mass". Through entire work replace "decent" with "descent" if some value goes down. (The other just means it was okay)

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.

2.1.1 Linearized Gravity

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 assuming the metric to be a linear correction to the flat metric $\eta_{\mu\nu}$

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

With this approximation the Einstein-equation (2.1) simplifies to

$$\mathcal{G}_{\mu\nu} = \frac{1}{2}(\partial_{\alpha\mu} h^{\alpha}_{\nu} + \partial^{\alpha}_{\nu} h_{\mu\alpha} - \partial_{\mu\nu} h - \Box h_{\mu\nu} - \eta_{\mu\nu}\Box h) = \frac{8\pi G}{c^4} T_{\mu\nu}, \tag{2.3}$$

where $h \coloneqq \eta^{\mu\nu} h_{\mu\nu}$ and $\Box \coloneqq \eta^{\mu\nu}\partial_{\mu\nu}$.
This equation has 10 independent components of which only 2 are physical. To reduce

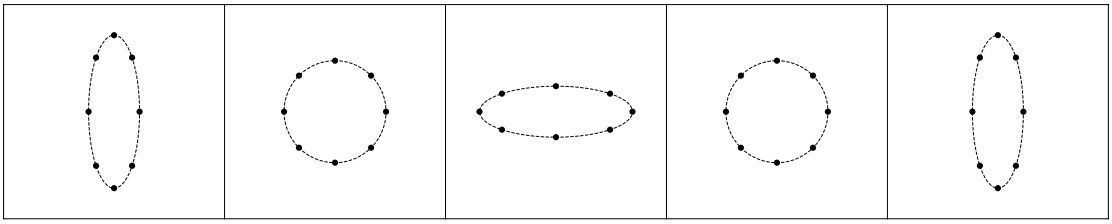


Figure 2.1: This image is taken from [15]. It shows the effect of a GW passing orthogonally through a ring of test masses.

the number of independent components, one can choose gauge conditions through the coordinate transformation $x'^{\mu} = x^{\mu} + \xi^{\mu}$, which leaves the Einstein equation invariant. One of these gauge conditions is the DeDonder gauge

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

where $\bar{h}_{\mu\nu} \coloneqq h_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu}h$. It can be realized by choosing $\Box\xi_{\mu} = \partial^{\alpha}\bar{h}_{\alpha\mu}$. In this gauge the linearized Einstein equation (2.3) reduces to

$$\Box\bar{h}_{\mu\nu} = -\frac{16\pi G}{c^4} T_{\mu\nu}. \tag{2.5}$$

This gauge however doesn't fix $h_{\mu\nu}$ completely, as another transformation $x'^{\mu} = x^{\mu} + \xi^{\mu}$ could be applied when $\Box\xi_{\mu} = 0$. This can be used in a way that $\bar{h} = -h = 0$ and $\bar{h}_{0\mu} = 0 = \bar{h}_{3\mu}$ are also satisfied. The gauge is named transverse-traceless-gauge (TT) and results in the metric to be of the form

$$h^{\mathrm{TT}}_{\mu\nu} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & h_+ & h_\times & 0 \\ 0 & h_\times & -h_+ & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \tag{2.6}$$

(2.6) now has only the two independent components h_+ and $h_\times$ left, which are called the "plus-" and "cross-polarization" of a GW.

Evaluating (2.5) in vacuum reveals the wave-like character of $h_{\mu\nu}$, as

$$\Box\bar{h}_{\mu\nu} = 0 \tag{2.7}$$

is a wave equation. Its solutions travel at the speed of light. Therefore, gravitational waves travel through space-time at the speed of light. The effect that a solution of this equation has on a ring of resting test masses is shown in Figure 2.1. (chapter 3 [15])

(2.7) shows that GW exist and can travel through space. It does, however, not specify how these waves are produced. To do so the energy-momentum-tensor cannot be set to 0. Instead the full equation (2.5) needs to be solved. The solution is known to be

$$\bar{h}(t,\vec{x}) = \frac{4G}{c^4}\int \mathrm{d}^3x' \, \frac{T_{\mu\nu}\left(t - \frac{||\vec{x}-\vec{x}'||}{c}, \vec{x}'\right)}{|\vec{x}-\vec{x}'|}. \tag{2.8}$$

For simplification, it is assumed that the observer is far away from the source when compared to the size of the support of $T_{\mu\nu}$, such that $|\vec{x} - \vec{x}'| \approx r := |\vec{x}|$. Therefore we need to solve

$$\bar{h}(t, \vec{x}) = \frac{4G}{c^4} \frac{1}{r} \int \mathrm{d}^3 x' \; T_{\mu\nu}\left(t - \frac{r}{c}, \vec{x}'\right). \tag{2.9}$$

This equation can be solved to yield

$$h_{ab}^{\mathrm{TT}}(t, \vec{x}) = \frac{2G}{c^4} \frac{1}{r} \ddot{I}_{ab}^{\mathrm{TT}}(t - r/c), \tag{2.10}$$

where $\ddot{I}_{ab}^{\mathrm{TT}}$ is the transverse-traceless-projection of the second time derivative of the second mass moment

$$\ddot{I}^{ab} = c^2 \partial_0^2 \int \mathrm{d}^3 x' \; x'^a x'^b T^{00} = 2 \int \mathrm{d}^3 x' \; T^{ab}. \tag{2.11}$$

As the quadrupole moment Q_{ab} is simply the traceless second mass moment and we project it to its traceless part anyways, (2.11) can be rewritten as

$$\boxed{h_{ab}^{TT}(t, \vec{x}) = \frac{2G}{c^4} \frac{1}{r} \ddot{Q}_{ab}^{TT}(t - r/c)} \tag{2.12}$$

with $Q_{ab} := I_{ab} - \frac{1}{3}\delta_{ab} I_c^c$. This is the famous quadrupole formula. (chapter 5.2 in [15])

To calculate the GW a binary system emits, I_{ab} or Q_{ab} needs to be specified. Furthermore, the transverse-traceless-projection needs to be calculated. The projection turns out to be ((3.64) in [16])

$$\ddot{I}_{ab}^{\mathrm{TT}} = \begin{pmatrix} \left(\ddot{I}_{11} - \ddot{I}_{22}\right)/2 & \ddot{I}_{12} & 0 \\ \ddot{I}_{21} & -\left(\ddot{I}_{11} - \ddot{I}_{22}\right)/2 & 0 \\ 0 & 0 & 0 \end{pmatrix}_{ab}. \tag{2.13}$$

Therefore, the waveforms are given by

$$h_+ = \frac{1}{r} \frac{G}{c^4} \left(\ddot{I}_{11} - \ddot{I}_{22}\right) \tag{2.14}$$

$$h_\times = \frac{2}{r} \frac{G}{c^4} \ddot{I}_{12}. \tag{2.15}$$

The approximations that led to (2.3) restrict the validity of the results above to cases where there are only slight perturbations to flat space-time. (2.2) actually assumes the background to be flat. Therefore, the dynamics of the two bodies orbiting each other are dictated by Newtonian gravity. With this in mind, the binary system we are trying to model is a system of two point-particles with masses m_1, m_2. For simplicity[2] assume

[2]It turns out that this simplification is very accurate. This is due to two reasone. First of all a possibile ellipticity is radiated away before the GW reaches currently detectable frequencies (4.1.3 in [16]). Secondly the rate of change of the orbital radius is small in the regime, where the approximation of linear gravity is meaningful. (4.1.1 in [16])

circular motion. In Newtonian mechanics, this problem reduces to an effective one body problem with the reduced mass $\mu = \frac{m_1 m_2}{m_1+m_2}$. The motion in these relative coordinates is given by

$$\vec{r}(t) = R \cdot \begin{pmatrix} -\sin(\omega_s t) \\ \cos(\omega_s t) \\ 0 \end{pmatrix}, \tag{2.16}$$

where R is the orbital separation of the two point masses and ω_s the orbital frequency. As a result one gets

$$\left[I^{ab}\right] = \mu R^2 \begin{pmatrix} \sin^2(\omega_s t) & -\frac{1}{2}\sin(2\omega_s t) & 0 \\ -\frac{1}{2}\sin(2\omega_s t) & \cos^2(\omega_s t) & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{2.17}$$

and subsequently

$$\left[\ddot{I}^{ab}\right] = 2\mu R^2 \omega_s^2 \begin{pmatrix} \cos(2\omega_s t) & \sin(2\omega_s t) & 0 \\ \sin(2\omega_s t) & -\cos(2\omega_s t) & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{2.18}$$

Therefore, the amplitudes are given by

$$\begin{aligned} h_+ &= \frac{4}{r}\frac{G}{c^4}\mu R^2 \omega_s^2 \cos(2\omega_s t) \\ h_\times &= \frac{4}{r}\frac{G}{c^4}\mu R^2 \omega_s^2 \sin(2\omega_s t). \end{aligned} \tag{2.19}$$

Interestingly, the frequency of the GW is twice the frequency of the orbital period.
The equations (2.19) are written in the source frame, i.e. they are the GW-polarizations emitted in the z-direction, where the z-axis is the one orthogonal to the orbital plane and at the center of mass. When measuring these waves we are not assured that the system emits face-on to our detectors. Therefore, we need to change coordinates to get the emission in a general direction $\hat{n}$. To do so, one simply has to transform the second mass moment into the new frame. These calculations can be found on p.111 in [16] which finally yield

$$\begin{aligned} h_+ &= \frac{4}{r}\frac{G}{c^4}\mu R^2 \omega_s^2 \left(\frac{1+\cos^2(\iota)}{2}\right)\cos(2\omega_s t + 2\Phi) \\ h_\times &= \frac{4}{r}\frac{G}{c^4}\mu R^2 \omega_s^2 \cos(\iota)\sin(2\omega_s t + 2\Phi), \end{aligned} \tag{2.20}$$

where ι is the inclination and Φ is the phase of the wave at $t = 0$.

To be measured these waves need to hit a detector. The most common and currently only operational GW-detectors are advanced Michelson interferometers with an angle of $\pi/2$ between the two arms. If the GW hits such a detector, it will cause a deviation δl in arm lengths given by the detector response functions

$$\delta l = F_+(\theta,\varphi)(\cos(2\psi)h_+ - \sin(2\psi)h_\times) + F_\times(\theta,\varphi)(\sin(2\psi)h_+ + \cos(2\psi)h_\times), \tag{2.21}$$

with h_+ and $h_\times$ as given in (2.20) and

$$F_+(\theta,\varphi) \coloneqq \frac{1}{2}\Big(1+\cos^2(\theta)\Big)\cos(2\varphi)$$
$$F_\times(\theta,\varphi) \coloneqq \cos(\theta)\sin(2\varphi). \tag{2.22}$$

The angle θ is taken between the propagation direction of the GW to the (outwards facing) normal of the detector. φ is the angle between one arm of the detector[3] to the projection of the propagation direction of the GW into the detector-plane. Therefore, the angles θ and φ, or rather their projection onto a global coordinate system, are the declination and right ascension respectively. The angle ψ is known as the polarization angle and is not detectable for a single detector. This is due to the reason that rotating the wave around its propagation axis has the same effect.

All the calculations above disregarded the energy carried away by the GW. To include it one needs to calculate the luminosity of a GW-source, which in turn requires the computation of an effective energy-momentum tensor of the GW itself.
To get this energy-momentum tensor, second order corrections in h of $R_{\mu\nu}$ need to be computed and averaged over time. The result is [Citation]

$$t_{\mu\nu} = \frac{c^4}{32\pi G}\langle \partial_\mu h^{\sigma\alpha}\partial_\nu h_{\sigma\alpha}\rangle. \tag{2.23}$$

The luminosity is the energy flux at spatial infinity and thus given by

$$L_{\mathrm{GW}} = \lim_{r\to\infty}\int_{S^2(r)} \mathrm{d}\vec{n}\ \vec{S}, \tag{2.24}$$

where $S^i = -c\cdot t^{0i}$ and $S^2(r)$ denotes the spherical shell of radius r. When solving this integral and using (2.12) one gets

$$\boxed{L_{\mathrm{GW}} = \frac{G}{5c^5}\langle \dddot{Q}^{ab}\dddot{Q}_{ab}\rangle.} \tag{2.25}$$

This equation can now be applied to the binary system specified by (2.17). To simplify notation and to give measurable parameters, notice that the dynamics of the system under consideration are governed by Newtonian physics and thus Kepler's laws apply. Especially Kepler's third law

$$\omega_s^2 = \frac{GM}{R^3} \tag{2.26}$$

will be of use, where $M = m_1 + m_2$ is the total mass of the system. Using (2.26) to eliminate R in (2.17) and inserting this equation into (2.25) yields

$$L_{\mathrm{GW}} = \frac{32}{5}\frac{c^5}{G}\left(\frac{G\omega_s M_c}{c^3}\right)^{10/3}, \tag{2.27}$$

[3] If the arms were labeled with x and y in such a way that they form a right handed coordinate system with the outwards facing normal vector, the arm the angle φ is taken to is the one labeled x.

where

$$M_c \coloneqq \mu^{3/5} M^{2/5} = \frac{(m_1 m_2)^{3/5}}{(m_1 + m_2)^{1/5}}. \tag{2.28}$$

M_c is called the chirp mass and is the only mass-combination a GW depends on in linearized theory.

According to (2.27), a binary system looses energy when emitting GW. The energy that is carried away is taken from the orbital energy E_{orbit} of the binary system. Therefore, disregarding any other effects[4] that might cause E_{orbit} to vary, we get

$$-\frac{dE_{\text{orbit}}}{dt} = -\frac{1}{2}\frac{G m_1 m_2 \dot{R}}{R^2} \overset{!}{=} L_{\text{GW}}. \tag{2.29}$$

One can again utilize (2.26) to eliminate R and $\dot{R}$ in favor of ω_s and $\dot{\omega}_s$. Furthermore, $\omega_s = \pi f_{\text{GW}}$, and thus

$$\dot{f}_{\text{GW}} = \frac{96}{5}\pi^{8/3}\left(\frac{G M_c}{c^3}\right)^{5/3} f_{\text{GW}}^{11/3}. \tag{2.30}$$

This equation describes a runaway process, as for a positive value f_{GW} the change in frequency is positive, leading to a larger value of f_{GW} and so on. This in turn by (2.26) means that the two masses will come closer and closer together until they touch. The point in time at which the waveform shuts off, will be denoted by t_{coal}. With this, one can define the time until coalescence $\tau = t_{\text{coal}} - t$ and solve the differential equation (2.30). [Cite p.170 [16]]

$$f_{\text{GW}}(\tau) = \frac{1}{\pi}\left(\frac{5}{256}\frac{1}{\tau}\right)^{3/8}\left(\frac{G M_c}{c^3}\right)^{-5/8} \tag{2.31}$$

Now that the frequency evolution of a GW is known, the amplitudes h_+ and $h_\times$ can also be modeled. To do so, revisit the initial assumption (2.16). In this equation R will now be time dependent and $\omega_s t$ will be replaced by $\Phi(t)$, where

$$\Phi(t) = 2\pi \int_{t_0}^{t} \mathrm{d}t' \; f_{\text{GW}}(t'). \tag{2.32}$$

In principle, all time derivatives in (2.18) would need to be redone, taking into account the time dependence of ω_s and R. However, the approximations that have led to these waveforms are quite strong. The rates of change $\dot{\omega}_s$ and $\dot{R}$ will only have non-negligible contributions when frequencies are pretty high and the orbital separation R is small. In these regimes the linear approximation (2.2) will be invalid. Therefore, we can neglect the contributions of $\dot{\omega}_s$ and $\dot{R}$ and still get a qualitative look into the dynamics of the system. Hence, replace ω_s in the prefactor of (2.20) by $\pi f_{\text{GW}}(t)$, $2\omega_s t + 2\Phi$ by $\Phi(t)$ and

[4] These effects could for instance be tidal deformation, mass acquisition or other sources of gravity in the proximity of the binary system.

R by (2.26).
With (2.31), equation (2.32) can be solved to yield

$$\Phi(\tau) = -2\left(\frac{5GM_c}{c^3}\right)^{-5/8}\tau^{5/8} + \Phi_0, \tag{2.33}$$

where Φ_0 is the phase at $\tau = 0$, i.e. at coalescence. Therefore, this value is called the coalescence phase. Combining these results, one gets the time dependent waveforms

$$\begin{aligned} h_+(t) &= \frac{1}{r}\left(\frac{GM_c}{c^2}\right)^{5/4}\left(\frac{5}{c\tau}\right)^{1/4}\left(\frac{1+\cos^2(\iota)}{2}\right)\cos(\Phi(\tau)) \\ h_\times(t) &= \frac{1}{r}\left(\frac{GM_c}{c^2}\right)^{5/4}\left(\frac{5}{c\tau}\right)^{1/4}\cos(\iota)\cos(\Phi(\tau)). \end{aligned} \tag{2.34}$$

Inserting (2.34) and (2.21) shows that in linearized theory the output of a detector depends on 8 parameters. These are the luminosity distance r, the chirp mass M_c, the coalescence time t_{coal}, the coalescence phase Φ_0, the inclination ι, the declination θ, the right ascension φ and the polarization angle ψ. The first four parameters are source intrinsic parameters, where M_c is a combination of the component masses m_1 and m_2. In that sense the parameter space can be extended to be 9-dimensional.
Figure 2.2 shows an example of the time evolution of a waveform described by (2.34). To obtain the spin effects in linearized gravity, one could write down the lagrangian of two spinning particles orbiting each other, solving the lagrange equation for the trajectories of the particles and insert these trajectories into the definition of the quadrupole tensor. (At least that's how I think it could be done. If there is time, maybe do these calculations.)
Alongside energy, GW also carry away angular momentum from the source. Using the quadrupole radiation (2.12), the change in angular momentum evaluates to ((3.97) in [16])

$$\frac{dJ^i}{dt} = \frac{2G}{c^5}\varepsilon^{ikl}\langle\ddot{Q}_{ka}\dddot{Q}_{la}\rangle. \tag{2.35}$$

The angular momentum that is carried away comes from the total angular momentum of the source. This in turn is comprised of the orbital angular momentum as well as the individual spins of the two component masses of a binary system. Therefore it is at least qualitatively understandable that the spins of the two bodies has an effect on the evolution of the waveform. Thus the 9 parameters of a waveform can be extended to 15, if both objects are allowed to rotate. The 6 additional parameters are the individual angular momenta of the two masses. Two further parameters influence the waveform if the objects are not rigid and allowed to deform. This is true for binary neutron stars, as neutron stars are not singularities.
Even though this work deals with BNS-signals, we neglect spin effects and tidal deformability and will thus not go into further detail here. For more information on spin- and tidal effects see [Citations].

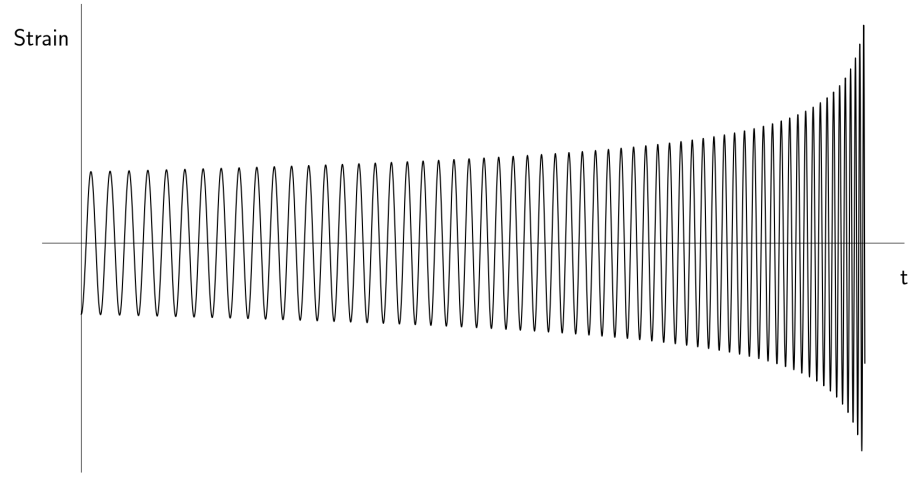


Figure 2.2: This figure is heavily based on Figure 4.1 in [16]. Shown is an example of a waveform as it could be observed by a detector if the linear waveforms describe the source accuratly. Note the frequency and amplitude evolution, they both rise simultaniously. This behavior is called "chirping".

2.1.2 Post-Newtonian Expansion

This part closely follows chapter 5 in [16], mainly stating results and concepts.
As we will see in subsection 2.2, accurate models for the waveforms are necessary to detect GW using traditional methods. This does not change for an approach utilizing machine learning algorithms, as they can only detect waveforms from distributions they have sampled before. Citation? Though the waveform derived in (2.34) gives a good qualitative overview of the rough amplitude evolution of inspiraling binary systems, it is hardly an accurate model. The main drawback of this model are the dynamics used to describe the motion of the system. It assumed (2.2) which in turn means that GWs and source-dynamics can be separated. Therefore, in linearized gravity the motion of a binary system is dictated by Newtonian dynamics, while the GWs are a relativistic effect.
This issue is overcome, by taking an approach called post-Newtonian expansion (PN-expansion). To approximate the solution of the full Einstein equation (2.1), $g_{\mu\nu}$ is expanded in powers of the small parameter $\epsilon \sim v/c \sim (R_s/d)^{1/2}$, instead of using $g_{\mu\nu} \approx h_{\mu\nu} + \eta_{\mu\nu}$. Here v is the typical speed inside the source, R_s is the Schwarzschild radius attributed to the system's total mass and d is the diameter of a world-tube containing the support of the energy-momentum-tensor of the source. Therefore, ϵ is small if the source is not too compact and speeds are low compared to the speed of light.

Specifically the metric expands to (Mention why they start at different orders?)

$$\begin{aligned} g_{00} &= -1 + g^{(2)}{}_{00} + g^{(4)}{}_{00} + g^{(6)}{}_{00} + \dots \\ g_{0j} &= + g^{(3)}{}_{0j} + g^{(5)}{}_{0j} + \dots \\ g_{ij} &= \delta_{ij} + g^{(2)}{}_{ij} + g^{(4)}{}_{ij} + \dots, \end{aligned} \tag{2.36}$$

where $g^{(n)}$ denotes a term $\sim \epsilon^n$. The energy-momentum-tensor can be expanded in an equivalent way

$$\begin{aligned} T^{00} &= T^{(0)\,00} + T^{(2)\,00} + \cdots \\ T^{0i} &= T^{(1)\,0i} + T^{(3)\,0i} + \cdots \\ T^{ij} &= T^{(4)\,ij} + T^{(4)\,ij} + \cdots . \end{aligned} \tag{2.37}$$

These expressions than need to be inserted into (2.1) and terms of the same order in ϵ need to be equated. To get the equations of motion to the n-th PN-order terms up to order ϵ^{2n} need to be kept and computed. Therefore it is possible to have the correction of some quantity to PN-order 2.5.
To get the 1-PN order corrections to the metric, one can once again impose a gauge condition. Gauge condition can be required for any order. The gauge condition most commonly used is still called deDonder gauge, which in the PN-case reads

$$\partial_\mu(\sqrt{-g}g^{\mu\nu}) = 0, \tag{2.38}$$

where g is the determinant of $g_{\mu\nu}$. In this gauge the Einstein equation yields (to 1-PN order)

$$\begin{aligned} \Delta g^{(2)}{}_{00} &= -\frac{8\pi G}{c^4} T^{(0)\,00} \\ \Delta g^{(2)}{}_{ij} &= -\frac{8\pi G}{c^4} \delta_{ij} T^{(0)\,00} \\ \Delta g^{(3)}{}_{0i} &= \frac{16\pi G}{c^4} T^{(1)\,0i} \\ \Delta g^{(4)}{}_{00} &= \partial_0^2 g^{(2)}{}_{00} + g^{(2)}{}_{ij}\partial_i\partial_j g^{(2)}{}_{00} - \partial_i g^{(i)}{}_{00}\partial_j g^{(2)}{}_{00} \\ &\quad - \frac{8\pi G}{c^4}\left\{T^{(2)\,00} + T^{(2)\,ii} - 2g^{(2)}{}_{00}T^{(0)}00\right\}, \end{aligned} \tag{2.39}$$

with $\Delta = \delta^{ij}\partial_i\partial_j$. Observe that higher order terms in (2.39) depend on the lower order terms of the expansion. Therefore the PN-expansion is an iterative process.
The equations (2.39) do in principle have many solutions. A particular solution, however, is specified by the boundary condition. A typical boundary condition is the "no incoming radiation" condition, where it is required that the metric approaches the flat space time metric η as one goes to spatial infinity. The PN-expansion however is only valid in the near region of the source, as it approximates the retarded solutions by a series of instantaneous potentials. To use a correct boundary condition one approximates

the far-field solution and matches it with the near-field PN-solution.
The approximation for the far-field is called Post-Minkowskian-expansion (PM-expansion).
For simplified notation the Einstein equation will be recast into their relaxed form

$$\Box k^{\mu\nu} = \frac{16\pi G}{c^4}\tau^{\mu\nu}. \tag{2.40}$$

To get this result, the deDonder gauge

$$\partial_\nu k^{\mu\nu} = 0 \tag{2.41}$$

was again required. The metric k is given by

$$k^{\mu\nu} := \sqrt{-g}g^{\mu\nu} - \eta^{\mu\nu}. \tag{2.42}$$

Furthermore

$$\tau^{\mu\nu} = (-g)T^{\mu\nu} + \frac{c^4}{16\pi G}\Lambda^{\mu\nu}, \tag{2.43}$$

with $\Lambda^{\mu\nu}$ being a tensor that depends highly nonlinearly on k and g. The expression is given in (5.74) of [16]. To simplify the relaxed Einstein equation (2.40) in the far field, we denote that the energy-momentum-tensor of matter $T^{\mu\nu}$ vanishes outside the source. Therefore the task is solving

$$\Box k^{\mu\nu} = \Lambda^{\mu\nu}. \tag{2.44}$$

To do so, we expand k in powers of R_s/r, which is equivalent to expanding in powers of G. We also expand $\Lambda^{\mu\nu}$ in powers of k. Therefore

$$k^{\mu\nu} = \sum_{n=1}^{\infty} G^n k_n^{\mu\nu} \tag{2.45}$$

$$\Lambda^{\mu\nu} = N^{\mu\nu}[k,k] + M^{\mu\nu}[k,k,k] + \dots, \tag{2.46}$$

where $N^{\mu\nu}[k,k]$ denotes a tensor of quadratic order in G. Equating terms of the same order in G and iteratively using the results for k yields

$$\Box k_1^{\mu\nu} = 0 \tag{2.47}$$

$$\Box k_2^{\mu\nu} = N^{\mu\nu}[k_1,k_1] \tag{2.48}$$

$$\Box k_3^{\mu\nu} = M^{\mu\nu}[k_1,k_1,k_1] + N^{\mu\nu}[k_1,k_2] + N^{\mu\nu}[k_2,k_1] \tag{2.49}$$

$$\vdots$$

or in short

$$\Box k_n^{\mu\nu} = \Lambda_n^{\mu\nu}[k_1,\dots,k_{n-1}]. \tag{2.50}$$

The most general solution to k_1 can be written in terms of retarded multipolar waves. The solution under consideration of the deDonder gauge can be found in equation (5.95) and following of [16]. They consist of multiple retarded potentials and form a multipole expansion of k_1. To find the solution to any order n equation (2.50) needs to be solved,

inserting all previous solution $k_1, \ldots, k_{n-1}$. Therefore, a general solution to (2.50) would be convenient.
Traditionally such a solution is known and given by the retarded Green's function. The problem, however, is that the solution to (2.50) is only valid outside the source but solving it by the retarded Green's function requires knowledge over the entire region. To get around this issue, we observe the fact that we only want the solution of k to some finite order in G. This has the benefit that only a finite number of multipole terms of $\Lambda_n^{\mu\nu}$ have to be used. The finite number of terms enables us to find some constant B, such that $r^B \Lambda_n^{\mu\nu}$ is defined for all r and thus its solution is given by the retarded Green's function. For $B \to 0$ the original divergence is recovered and the multipole expansion has poles. Therefore, near $B = 0$ the solution $I_n^{\mu\nu}(B) = \Box_{\text{ret}}^{-1}\left(r^B \Lambda_n^{\mu\nu}\right)$ can be expanded in a Laurent-series. Taking only the zeroth order term yields a particular solution $u_n^{\mu\nu}$ with

$$\Box u_n^{\mu\nu} = \Lambda_n^{\mu\nu}. \tag{2.51}$$

From this particular solution the general solution can be constructed by adding the homogeneous solution.

As a final step, the PN-expansion can be recast in the form of k, where we expand

$$k_{\mu\nu} = \sum_{n=2}^{\infty} \frac{1}{c^n} \frac{d^n}{du^n} k_{\mu\nu}(u) \tag{2.52}$$

$$\tau^{\mu\nu} = \sum_{n=2}^{\infty} \frac{1}{c^n} \frac{d^n}{du^n} \tau^{\mu\nu}(u), \tag{2.53}$$

with $u = t - r/c$ and τ given in (2.43). Doing so and inserting it into the relaxed Einstein equations results in the recursive relation

$$\Delta \left[\frac{d^n}{du^n} k^{\mu\nu} \right] = 16\pi G \frac{d^{n-4}}{du^{n-4}} \tau^{\mu\nu} + \partial_t^2 \left[\frac{d^{n-2}}{du^{n-2}} k^{\mu\nu} \right]. \tag{2.54}$$

Taking the solution for the 1PN case discussed above as a starting point the (2.54) can be solved in a similar fashion to (2.50) using only a finite number of terms in a multipole expansion of the potentials.

With this setup we mention once more that the regions of validity for the PN- and PM expansion overlap but neither are completely solved. To obtain the full solution the PN-equations need a boundary condition, whereas the PM-equations need a source of some form. Therefore the PM-equations can be viewed as the limiting case of the PN-case and thus provide a boundary condition. The PN-equations on the other hand have fixed multipole potentials that depend on the energy-momentum tensor of matter. These can in turn be used to fix the multipole potentials in the PM-equations and one obtains a full solution.
At this point we will not go further into further details of the formalism itself but rather look at its influence on the waveforms. For further reference on the PN-formalism see

[Citations]. For information about the multipole expansion see [Citations].

Surprisingly, to 1PN order the results are identical to the ones obtained using linearized gravity in subsubsection 2.1.1. For convenience one defines the dimensionless quantity

$$x \coloneqq \left(\frac{GM\omega_s}{c^3}\right)^{2/3}, \tag{2.55}$$

where M is the total mass and ω_s the orbital frequency. Note that $x \sim \frac{v^2}{c^2}$ and thus the PN-expansion can be given in terms of powers in x. For further notational simplicity define the symmetric mass ratio

$$\nu \coloneqq \frac{m_1 m_2}{(m_1+m_2)^2}, \tag{2.56}$$

the post-Newtonian parameter

$$\gamma \coloneqq \frac{GM}{rc^2} \tag{2.57}$$

and the dimensionless time parameter Maybe remove θ as it is not used.

$$\Theta \coloneqq \frac{\nu c^3}{5GM}(t_{\text{coal}} - t). \tag{2.58}$$

Using the metrics acquired from the PN-expansion one can solve the equations of motion for the two inspiraling bodies and obtain corrections to ω_s, γ, the energy E and the radiated power L_{GW}. Combining these results one can than find the phase evolution and emitted waveforms. The computations are extremely long. For this reason we only state the results for the energy and luminosity at 3.5PN order here. (μ in this case is the reduced mass) (equation (5.256) in [16])

$$\begin{aligned} E = &-\frac{\mu c^2 x}{2}\left\{1+\left(-\frac{3}{4}-\frac{1}{12}\nu\right)x+\left(-\frac{27}{8}+\frac{19}{8}\nu-\frac{1}{24}\nu^2\right)x^2\right. \\ &\left.+\left[-\frac{675}{64}+\left(\frac{34445}{576}-\frac{205}{96}\pi^2\right)\nu-\frac{155}{96}\nu^2-\frac{35}{5184}\nu^3\right]x^3\right\}+\mathcal{O}\left(\frac{1}{c^8}\right) \end{aligned} \tag{2.59}$$

In the equation below C is the Euler-Mascheroni constant. (equation (5.257) in [16])

$$\begin{aligned} L_{\text{GW}} = &\frac{32c^5}{5G}\nu^2 x^5\left\{1+\left(-\frac{1247}{336}-\frac{35}{12}\nu\right)x+4\pi x^{3/2}\right. \\ &+\left(-\frac{44711}{9072}+\frac{9271}{504}\nu+\frac{65}{18}\nu^2\right)x^2 \\ &+\left(-\frac{8191}{672}-\frac{583}{24}\nu\right)\pi x^{5/2} \\ &+\left[\frac{6643739519}{69854400}+\frac{16}{3}\pi^2-\frac{1712}{105}C-\frac{856}{105}\log(16x)\right. \\ &\left.+\left(-\frac{134543}{7776}+\frac{41}{48}\pi^2\right)\nu-\frac{94403}{3024}\nu^2-\frac{775}{324}\nu^3\right]x^3 \\ &\left.+\left(-\frac{16258}{504}+\frac{214745}{1728}\nu+\frac{193385}{3024}\nu^2\right)\pi x^{7/2}+\mathcal{O}\left(\frac{1}{c^8}\right)\right\} \end{aligned} \tag{2.60}$$

2.1.3 TaylorF2

Even though the PN-formalism allows us to model the waveforms rather accurately for a long part of the signal it still assumes circular motion and velocities to be small. As the two stars spiral together they will, however, reach a point, where circular motion is not possible anymore. This orbit is called the "innermost stable circular orbit" (ISCO). From here the stars will plunge towards each other, velocities are high and fields are not weak anymore. Therefore, the validity of the formalism breaks down at this point. Finally, after the two bodies have merged, the remaining body will radiate off some more energy. Therefore, the signal consists of three parts: inspiral, merger and ringdown. The inspiral phase is well modeled by the PN-formalism outlined in subsubsection 2.1.2. The ringdown can be described by What can it be described by and where can one find an analysis of this? The merger however is very difficult to model and usually involves numerical solutions of the Einstein equations. These, however, are very costly from a computational perspective, which is a great problem when one searches for GWs. Transition into roughly describing the PN-approximation and TaylorF2 here. Include backreaction in first order as 4.1.1 [16] does, until equation (4.32). Mention that elliptic effects can be mostly disregarded, cite [16]. Go over into PN territory and mention what changes.

2.2 Searching for Gravitational Waves

Explain what matched filtering is, why it works and how it is applied currently. Also need to mention PSD and what it is.

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 [17, 18].

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

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

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.

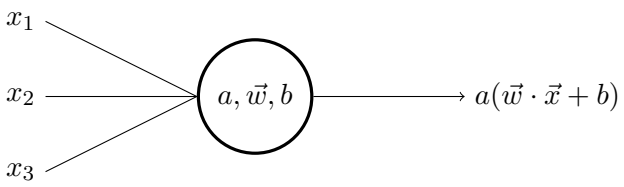


Figure 3.1: Depiction of a neuron with inputs $\vec{x} = (x_1, x_2, x_3)^T$, weights $\vec{w}$, bias b and activation function a.

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. Its 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 [18]), which caused the loss of interest in NNs around 1940 (section 6.6 in [18]).

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 previous layers. Formally we write

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

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

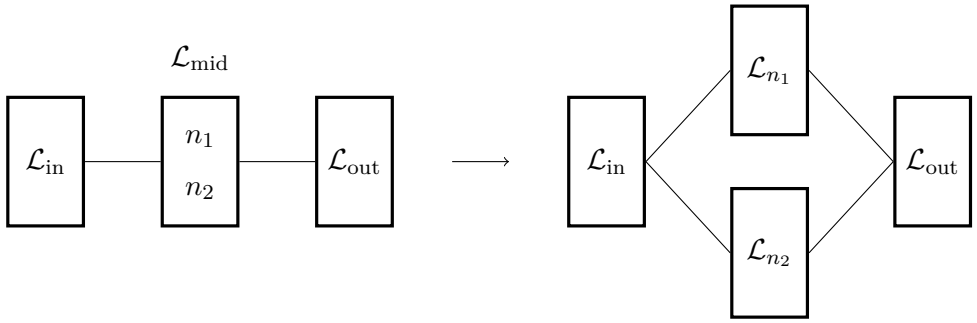


Figure 3.2: Depiction of how a layer ($\mathcal{L}_{\text{mid}}$) consisting of neurons with different activation functions (n_1 and n_2) can be split into two separate layers ($\mathcal{L}_{n_1}$ and $\mathcal{L}_{n_2}$).

as

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

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 [18]). 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

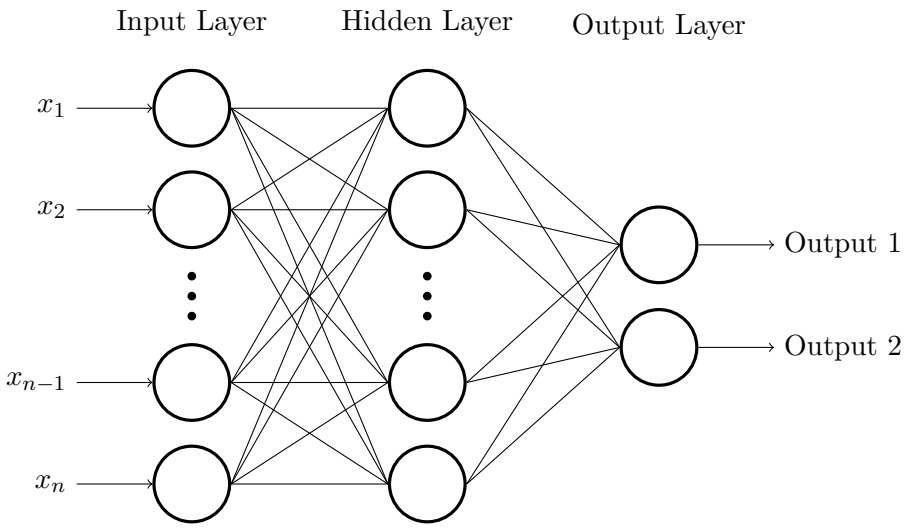


Figure 3.3: A depiction of a simple network with a single input-, hidden- and output-layer. The input-data is a n-dimensional vector $(x_1, \ldots, x_n)^T$ and the output is a 2-dimensional vector. In this picture it looks like the hidden layer has the same number of neurons as the input layer. This 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 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.[5][17]
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

[5]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 [18])

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*[6] 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 [18].
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 cost intensive from a computational perspective. Therefore, the training set is split into multiple parts, called mini-batches. A step of the

[6] For regression problems this value is often also called target value. We will however stick to calling it the label for our training data.

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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)

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

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

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

With this definition equation (3.19) can be written as

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

and the recursion can be solved to yield

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

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 [18])
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 [18].)

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 in detail here, is data augmentation[7]. (See section 7.4 of [18])
To tune out the generalization error, which is the loss value of the validation set, one

[7]Data augmentation is the process of applying transformations to the input samples, to artificially create more data. The transformation have to act in such a way that the resulting data is similar in its properties to the original data.

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[8], 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 [18]) 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 [19].
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 $\left|\partial_z a_i(z) \cdot W^i\right| < 1$ the gradient is exponentially diminished by the depth of the network.
The opposite can also happen. If $\left|\partial_z a_i(z) \cdot W^i\right| > 1$ for most of the layers, the gradient will grow exponentially. This behavior is therefore called the exploding gradient problem. (Chapter 5 [17])
To overcome these problems, one can simply train for longer periods in the case of the vanishing gradient problem (Chapter 5 [17]), use multiple points at which the loss is calculated [20] or adjust the initialization of the weights. Another solution introduced by [21] are residual connections. These are connections of a layers input to its output. Specifically, the input of the layer is added back onto its output. The idea behind this connection is to make it easier for the network to learn an identity mapping by simply setting the weights of the layer to zero. This showed great improvements and the ability to have very deep networks[21].

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.

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

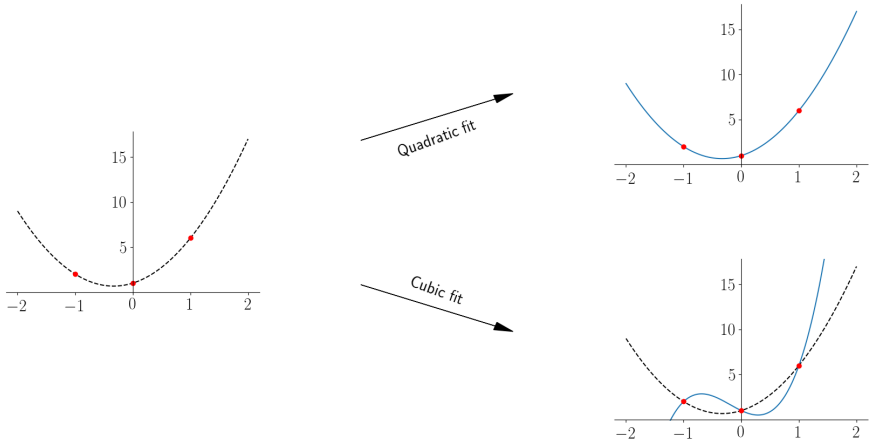


Figure 3.4: Depiction of overfitting in the classical regression. On the left the originally sampled function $f(x) \coloneqq 3x^2 + 2x + 1$ is shown in dashed and black. The red dots are the samples that are being used for the regression on the right. The top right shows the regression, where $g(x) = ax^2 + bx + c$ 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.

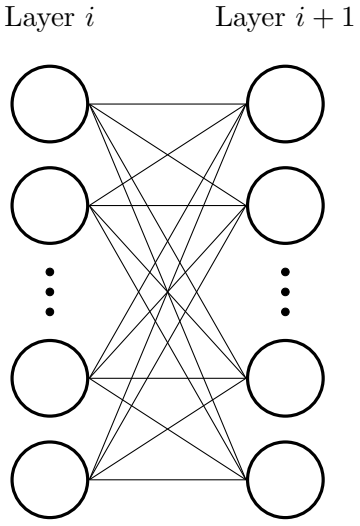


Figure 3.5: Insert description! Maybe inprove this graphic.

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. Being static means, that it is hard for the network to adapt to slight changes in the data. To understand 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. [22] Though it was originally conceived in its 2 dimensional variant, only the 1 dimensional convolution layer will be explained here.[9] 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[10]. 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.

[9] The core concepts are the same, thus the concept can easily be adapted to any number of dimensions.
[10] Usually all filters have the same kernel size.

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 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

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

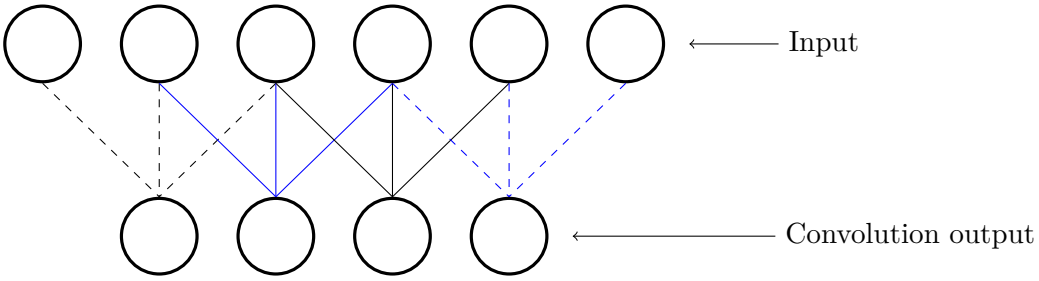


Figure 3.6: Example of a convolution layer with a kernel size of 3. It highlights the sparse connectivity of the different neurons. Each of the output neurons is now only connected to three of the previous neurons. Each of the weights associated with the lines in the picture above is shared, i.e. the 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.

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 $\vec{w}$.

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. [23] 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[11]. 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

[11] There have been studies suggesting, that pooling works better than simply sub sampling the data. [24]

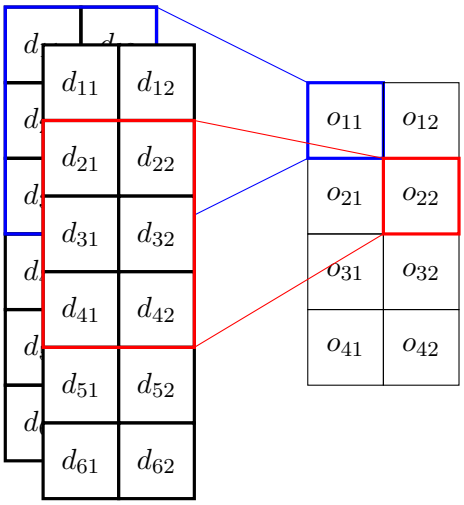


Figure 3.7: Depiction of a convolution layer with two filters, that have a kernel size of 3. The input data d_{11} to d_{62} has two channels and is the same for both the blue and the red filter. The first channel of the output (o_{11} to o_{41}) is produced by sliding the blue filter over the data,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

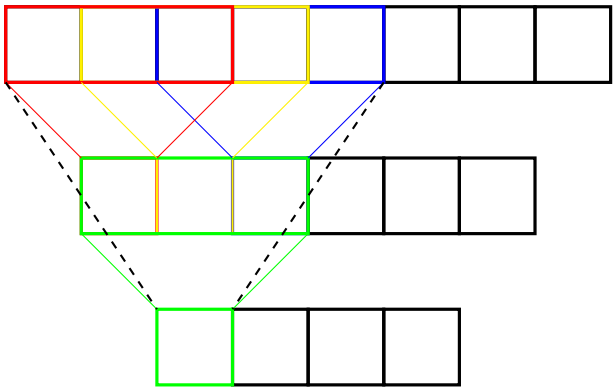


Figure 3.8: Three stacked convolution layers. Although all layers have a kernel size of 3, the final layer is influenced by 5 of the input values. Therefore the receptive field of the final layer is 5. Maybe change the colors and some presentation of this graphic. I don't like how it looks.

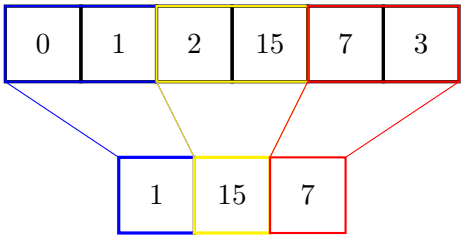


Figure 3.9: An example of a Max pooling layer. It groups together two entries of its input and returns the maximum, thus halving the number of samples per feature map. This process is applied for all channels.

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 a kernel size of 1. This way, all channels are being connected through a weighted sum, where the weights are learned.[25] 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. [26] 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. [18, 27, 9] 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 [20].
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. [20] 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. [20]
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[12] 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. [20, 9]
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. [28, 29]

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

[12]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.

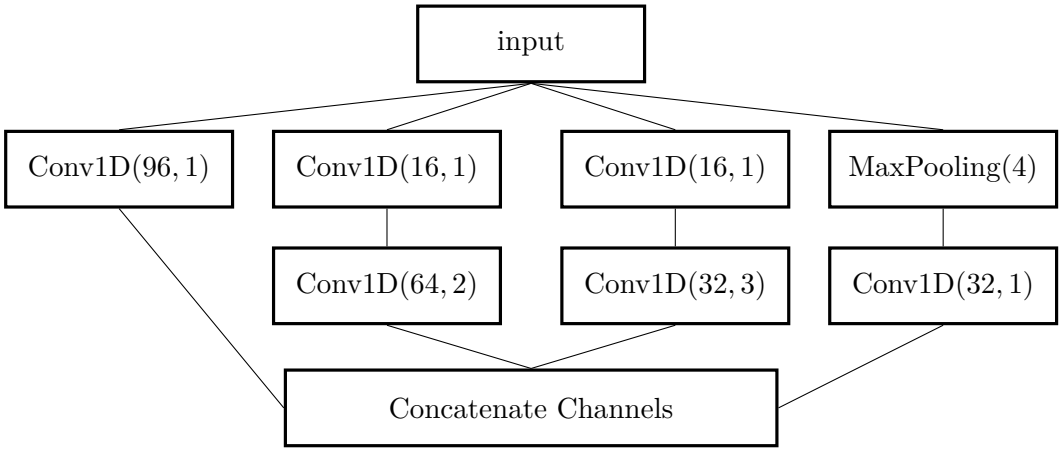
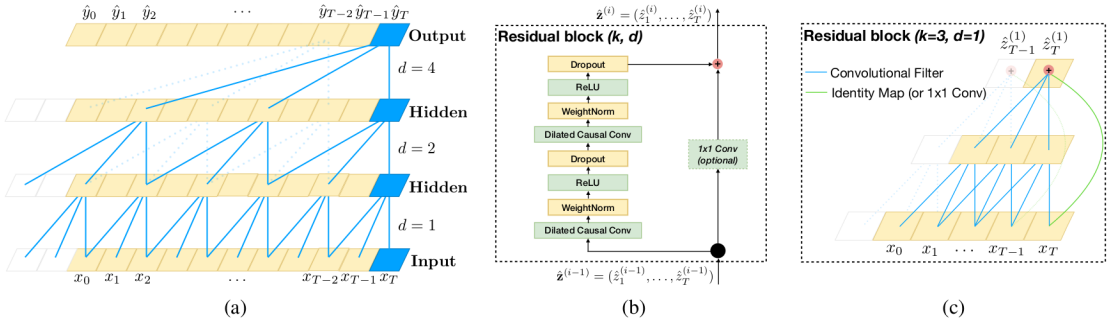


Figure 3.10: Shown are the contents and connections of the inception module as used in this work. (If the filter numbers and values change for the final architecture, change them here too.) The layer Conv1D(x, y) is a 1 dimensional convolution layer with x filters and a kernel size of y. Most of the 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.

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.
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 [30]. 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



(a) (b) (c)

Figure 3.11: Architectural elements in a TCN. Graphic taken from [30]. (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.

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 $x_i, \ldots, x_1$. 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 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. [31]

3.5 Regularization

As [18] 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 [32] 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. [32]
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 only over one dimension of the input. In this work, the mean and variance will be calculated for every channel and thus applied to each channel individually. Finally a linear transformation

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

is applied to the normalized data

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

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 activations 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.

The implementation in Keras differs from this approach in the sense, that the mean μ_B and variance ${\sigma_B}^2$ are only calculated for each individual batch during training. When the network is used to evaluate some data, it will use a fixed mean and variance, that was approximated over all batches during training. They call this the moving average μ_{mov} and moving variance σ^2_{mov} respectively and adjust them after every batch by

$$\begin{aligned} {\mu'}_{\text{mov}} &= m \cdot \mu_{\text{mov}} + (1-m)\mu_B \\ {\sigma'}_{\text{mov}}^{\,2} &= m \cdot {\sigma_{\text{mov}}}^2 + (1-m){\sigma_B}^2, \end{aligned} \tag{3.31}$$

where m is the momentum used and usually set to a high value around 0.99. Using this has the advantage, that only a finite number of samples have a non negligible effect on the mean- and variance value used during inference. Therefore drifts in the input distribution, which might occur during training, can be counteracted.

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 [33] 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 [18] and [33], as it allows to efficiently sample many networks and combine them.
During the evaluation process, the original paper [33] 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]

4 Searching for Gravitational Waves using Neural Networks

Put motivation here

Motivate why the problem we are looking at is interesting and why it might be beneficial to train for SNR instead of simply classifying. Why is BNS more difficult? (Care, I think I wrote a little bit about this in the Data Generating Process section) Maybe drop this if the motivation is clear from the introduction and related works section.

With the rise in popularity of NNs, machine learning techniques were applied to a wide range of problems. One of these is the GW data-analysis where it has found multiple applications.

The use off deep neural networks as a filter to detect GW-signals in noisy detector data was pioneered by Daniel George and E.A. Huerta [11]. They used a CNN to classify time series data into the two categories "noise + signal" and "pure noise" as well estimate some source parameters for BBH signals. The network was able to closely reproduce the results a classical matched filter search could achieve and even showed potential to adapt to eccentric signals which were not used during the training stage. The followup paper [34] used real instead of simulated detector noise, demonstrating that CNNs can be used for real time classification and parameter estimation.

A similar concept has been applied to the search for continuous GW in [35]. The authors show that for a relatively short observation time the performance of their network on low frequency data rivals that of matched filtering, while for higher frequencies or longer observation times their approach falls off. The key advantage of this search is the computational efficiency. Using CNNs reduces the pure search time by several orders of magnitude. Even including the time spent during training and for a necessary followup search seems to reduce the total computational cost. However the authors don't get more specific in that regard.

The recently published paper [36] claims to be able to differentiate between the three classes "noise", "BBH-signal" and "BNS-signal", by using a whitened time series of 2 s duration. We question their results in multiple aspects based on out own research. First of all, their validation and testing set contains only 5000 samples, with a split of only 1/3 being pure noise samples. From so few samples it is difficult to get a good estimate of the false alarm rate, due to the lack of noise realizations. This is one of the possibly many factors that lead to their quoted 0% false alarm probability for BNS and 1% for BBH signals. Furthermore, they shift their signals around the data by ± 0.1 s. With 1667 noise samples this leads to a false alarm rate of $\sim 15\,500\ \frac{\text{samples}}{\text{month}}$ for their loudest noise instance[13]. Secondly, they only calculate their results in terms of peak signal-to-noise ratio and quote a matched filter signal-to-noise ratio of 13 times the peak signal-to-noise ratio. However, using our own data, we found that a matched filter SNR of ~ 15 corresponds to a peak SNR of 0.38. Testing other SNR values we found a consistent conversion factor

[13] We calculated this number as explained below equation (5.2)

of $\mathrm{SNR}_{\mathrm{MF}} = 39.225 \cdot \mathrm{SNR}_{\mathrm{peak}}$[14]. Using this conversion factor for peak SNR to matched filter SNR, their network has a sensitivity of about 70% for BNS signals at matched filter SNR 15 and unknown false alarm rate. Considering the results of this thesis and the architecture they used, these results sound consistent to our findings. (compare the start of subsection 5.2) Finally, their work uses only a single detector. Assuming that both detectors see a signal equally strong means that results they find at SNR x correspond to signals we find at SNR $\sim \sqrt{2}x$.

[37] aims to reduce the confusion that occurs when mixing terms of computer science with those of gravitational data analysis. It criticizes the way statistical significance is claimed by different machine learning approaches. It especially claims that no statistical meaningful false alarm rate can be derived from NNs using only training, validation and testing set, if these sets contained individual samples of pure noise and signals. They base their criticism on the fact that a sliding window approach will not always contain the signals in the way the training set suggests. The waveform will not be positioned in just the way it was during the training stage but at some uncontrollable point. Due to this limitation they say that NNs can only be used as quick and reliable trigger generators that need a followup matched filter search. We try to address most of their points by generating all results from a long continuous time series. All triggers that are generated on this test set will use a fixed threshold that was determined on the validation set. Further measures and precautions will be explained in subsubsection 5.3.2.

A major contribution relating the architecture came from [38], who compared the performance of multiple different general architectures in the field of gravitational wave data analysis. Their findings indicated that a TCN was the best algorithm. They were also able to compare current feed forward neural networks against recurrent neural networks in the scope of GW data analysis and concluded that they in some cases outperform CNN architectures but can't match their TCN.

[39] takes a similar approach to their network architecture as [38] does but uses it to denoise the input data, i.e. recover the waveform from the noisy detector data. They show that the network is able to recover BBH signals to incredible accuracy and thus can learn the characteristics of the noise background. We use their idea as part of our final network.

[14] We report these values based on the following calculation: We define the peak SNR, following [36], as $\max(\text{waveform})/\sqrt{\mathrm{Var}(\text{noise})}$. The matched filter SNR is calculated as described in subsection 5.1 using a single detector. We than divide the matched filter SNR by the peak SNR to get the conversion factor. We use a sample rate of 4096 Hz to generate both noise and signal. Further tests show, that this is only beneficial for the claims of [36] as the PSD increases strongly for large frequencies, thus increasing the conversion factor. At a sample rate of 16 kHz we found a conversion factor of ~ 75.

5 Network Topologies

Need to introduce the optimizer we are using. Probably should do so in the beginning of the evolution section.
This section summarizes the results of our research, trying to find a deep learning algorithm that succeeds in detecting and classifying BNS-signals in noisy data. The goal was to not only classify the signals into the two categories "pure noise" and "signal" but also give an estimate of the signals SNR.
All software written for this work used the PyCBC software package [14] for any code involving gravitational wave data and the deep learning library Keras [12] to rapidly develop and test different network architectures. We used Tensorflow [13] as the backend for Keras. Should I cite the libraries here again? I did so in the introduction.

5.1 The Data Generating Process

Training a NN to detect and classify BNS signals requires a large set of mock data. Using the data without some processing, however, is not possible. This is due to the fact that BNS signals are more difficult to train for than BBH signals, as they are a lot weaker and last for a longer duration[15]. Training a NN on data of such duration sampled at a frequency high enough to resolve the final cycles of the binary system is not feasible, as the required network would need too many trainable parameters. However, to retain most of the SNR in the data, it is not possible to crop it to only contain a small portion of the waveform. To overcome this issue note that even though BNS signals spend a long time in the sensitive region of the detectors, their frequency evolves rather slowly and starts at low values.
For the early inspiral phase of the signal, frequencies are around 30 Hz to 100 Hz. To resolve these frequencies, a sample rate of 60 Hz to 200 Hz is necessary. The higher sample rates are only required for the final few seconds. For these reasons we propose to represent the data using a new multi-rate approach. Using this method, the network does not receive the data at a fixed sample rate, but rather multiple inputs, each sampling parts of the signal at different rates.
We choose the largest window to encompass 64 s of data, where a signal is aligned such, that the high frequency cutoff is roughly 0.5 s from the end. Afterwards we chop the data into parts of duration 2^i s and re-sample each of these parts to have a sample rate of 2^{12-i} Hz, with $i = 0, \dots, 6$. This way each sample rate contains exactly 4096 samples. The 7 re-sampled data segments, however, have overlaps. This is due to the fact, that the lower sample rates contain the data of the higher sample rates, e.g. the 2 s interval sampled at 2048 Hz also includes the 1 s interval sampled at 4096 Hz. For this reason and to reduce the number of input samples even further, we only use the first 2048 samples for each rate, except the highest one. To keep things simple however, the highest sample rate is split into two parts, each containing 2048 samples. Therefore each 64 s

[15] BBH signals spend about 1 s within the sensitive frequency range of our detectors, whereas BNS signals can be visible in the whitened detector data for multiple tens of seconds [8] and when generated even last multiple hundred seconds.

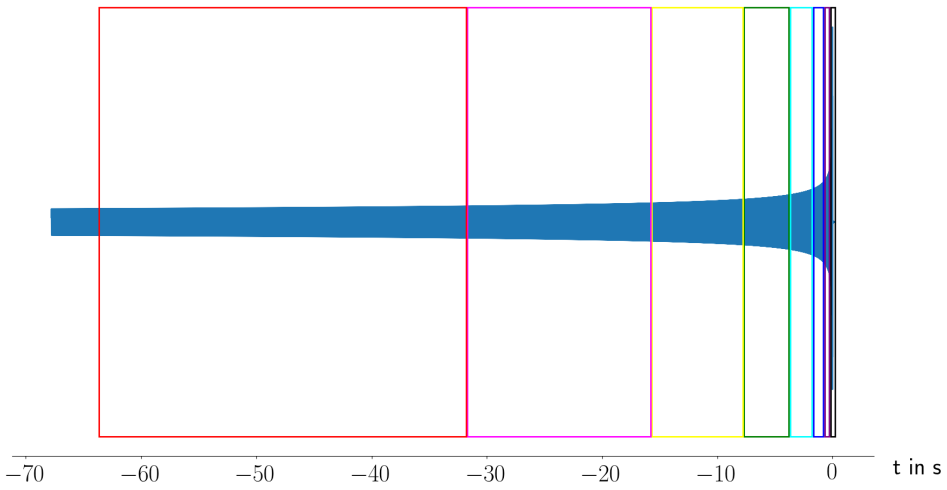


Figure 5.1: The plot shows how each BNS signal is sampled at multiple rates. Only the last 68 s of the entire waveform are shown. Each sample rate and interval has its own color attributed. Specifically they are given by: black=(0.5 s, 4096 Hz), purple=(0.5 s, 4096 Hz), blue=(1 s, 2048 Hz), cyan=(2 s, 4096 Hz), green=(4 s, 1024 Hz), yellow=(8 s, 512 Hz), magenta=(16 s, 256 Hz) and red=(32 s, 64 Hz), where each tuple gives (segment duration, sample rate).

input interval is split and re-sampled to yield 8 inputs, each containing 2048 samples. A depiction of this multi-rate sampling can be found in Figure 5.1.

A common drawback of training deep NNs is the need for large training sets. Luckily we are in a position where we can simulate our training samples and thus can generate an arbitrary amount. To do so we use the PyCBC software package [14]. The final training set contained 56250 different signals and 161250 different noise realizations. All of the noise samples were simulated using the analytic PSD `aLIGOZeroDetHighPower` provided by PyCBC. Therefore all results obtained using this data only hold for stationary gaussian noise. All waveforms were generated using the approximant "TaylorF2", as implemented by PyCBC. (Should this be Lal?) Out of the 17 parameters that could have been varied, we fixed the spins to 0 and neglected tidal effects for simplicity. Furthermore the coalescence time t_{coal} is set to 0 as well. The remaining 8 parameters were chosen to represent a realistic distribution in order to estimate the potential of our approach in a real search. As such, both component masses m_1 and m_2 are uniformly distributed in the range $1.2\,\mathrm{M}_\odot$ to $1.6\,\mathrm{M}_\odot$. Specifically we do not explicitly require $m_1 \geq m_2$ when generating the waveform. The coalescence phase Φ_0 and the polarization angle ψ are uniformly distributed on the interval $[0, 2\pi]$ and the inclination ι is distributed like $\arccos(\text{uniform}(-1, 1))$. Finally the sky-position is isotropic, i.e. θ is distributed like $\arccos(\text{uniform}(-1, 1))$ and φ uniform in $[-\pi, \pi]$.
The luminsoity distance r is not chosen indirectly by fixing the SNR to some value. This

is valid, as the SNR scales inversely with the distance. (Is this true, or is for instance $SNR \approx 1/r^2$?)) In this work, the SNR is uniformly distributed on the interval $[8, 15]$. One has to avoid one major pitfall when fixing or calculating the SNR and comparing it to other results. If one compares the SNR of two signals with the same parameters, the value will depend on the length of the segment used. According to Reference to matched filtering section, cutting off the waveform early might result in a lower or at least inaccurate value of the SNR. We therefore specify very precisely how we calculated the SNR. The waveforms are generated with a lower frequency cutoff of 20 Hz, which results in waveforms, with a duration of about 500 s. Afterwards the waveforms are projected onto the two detectors Livingston and Hanford and cropped in such a way, that they span 96 s and the shutoff lies within the last second. We vary the exact position of the highest amplitude between 0.25 s to 0.75 s from the end of the data stream and choose the signal that arrives at the latest point in time as reference. Only after the waveforms are cropped, we calculate the SNR of the pure signal (while assuming the PSD of the detector) using the waveform itself as a template. Since we are using multiple detectors, the SNR ρ_i is calculated for each detector. The total SNR in the absence of noise is given by

$$\rho_{\text{total}} = \sqrt{\sum_i \rho_i^2}. \tag{5.1}$$

Each waveform is than rescaled by multiplying with the factor $\text{SNR}/\rho_{\text{total}}$, where SNR is the target value. Each noise sample is labeled with SNR 4 by convention. This value can in principle be picked freely but we chose it to resemble the average output of a matched filter search on pure noise. As a last step, before re-sampling the data as described above, the samples are whitened. To do so, we divide each of them by the amplitude spectral density given by $\sqrt{\text{PSD}}$. The PSD again is given by the analytic version `aLIGOZeroDetHighPower` provided by PyCBC. Ideally one would use an estimate of the PSD to whiten the data. This is, however, not possible in our case, as we are storing signals and noise separately and only add them together at run time. Estimating the PSD would have the advantage of being more robust in a real search, as it would counteract drifts in the PSD.

The reason for storing noise and signals separately are resource constraints. To cover the entire parameter-space densely enough and avoid overfitting, a large number of samples is necessary. Initially we generated and stored the sum of signal and noise, instead of storing each category separately. This has multiple disadvantages, but also one key advantage; we can use the pure signal as a filter for the matched filtering (Insert equation number here), thus giving us an upper limit on the quality of the SNR a matched filter search could return. In that sense, we could monitor the performance and compare it to matched filtering directly during training. The disadvantages however at some point outweighed this advantage. The core one being the restricted number of samples. A file containing $500,000$ samples has a size on the order of 200 GB. To train the networks on the data, we completely load it into system memory and need some overhead for formatting. To reduce these costs, we decided to split the signal- and noise-samples and only at runtime add together one instance of each category on the first layer of the net-

work. The second advantage of this approach is less obvious. It enables us to easily feed the network the same signal submerged in multiple different noise realizations, which resulted in performance improvements for tasks similar to ours (Christoph Dreißigacker, personal communication, June 2019).
The split between training and validation set is treated with great care, assuring that not a single noise or signal sample from the training set is used during validation. Therefore the reported loss and accuracy values are representative of a real search. Though they are not the final statistic we report, they are tightly linked to those and give clues about the network and its efficiency.
The data used for training and validating the final network contained $75,000$ different GW-signals and $215,000$ noise realizations[16]. We than generate a set number of unique index pairs (s_i, n_i), where s_i corresponds to a signal and n_i to a noise sample. For the training set these indices may be selected from $s_i \in [0, 3/4 s_t)$ and $n_i \in [0, 3/4 n_t)$, where s_t and n_t are the total number of signals and noise samples respectively. If $3/4 s_t \notin \mathcal{N}$ or $3/4 n_t \notin \mathcal{N}$, the upper index is rounded to the nearest natural number. The total number of pairs generated is equal to the number of usable noise samples $3/4 n_t$. These index pairs represent all samples of the training/validation set that contain a GW. In order to also supply pure noise samples to the learning algorithm, all noise realizations are also used during training. This is achieved by appending all index pairs $(-1, 0), (-1, 1), \ldots, (-1, 3/4 n_t)$ to the list of index pairs generated before. Afterwards this list is shuffled. The NN finally is fed with these $2 \cdot 3/4 n_t$ samples through a function[17] that interprets the indices and reshapes the data. Overall the training set therefore consists of $322,500$ samples with a $1:1$-split between noise and signals. The validation set does has a $3:1$ split in favor of signals and contains the remaining $n_t/4$ noise samples. Therefore, the validation set consists of $215,000$ samples.
The shape of the data depends on the network in use. Our final network expects a list of 16 arrays, as we have 8 different sample rates and a signal and noise input for each of them. The arrays are of shape (mini-batch size, 2048, 2). The last axis is the number of detectors used, whereas the second axis is the number of samples in the time series strain data.

Above we have only discussed the training and validation set in detail. The final results, however, are evaluated on the testing set. To eliminate most possible error sources, the testing set is generated completely independently from training and validation set, sharing only the distribution of parameters as discussed above. The testing set, furthermore, does not consist of individual samples, where each sample either contains a GW aligned correctly or not, but is a set of continuous time series data. These large chunks are then handed to a generator function that chops each time series into overlapping blocks, i.e. sliding a window across the data. Each of these windows has a length of 96 s. The resulting chunk is whitened by dividing out the amplitude spectral density associated

[16]The numbers stated above were the number of samples in the training set. Here we are stating the numbers for the training and validation set combined, as they are stored in the same file. The numbers for the training set are a results of the split described below.
[17]Keras calls this function a generator.

to the analytic PSD `aLIGOZeroDetHighPower` and re-sampled to match the criteria of the network input. The window is than shifted by 0.25 s and the process repeated. The temporal resolution of 0.25 s was chosen because the waveforms are shifted around in the training set by ± 0.25 s. With the step chosen this way, we are guaranteed to have the maximum amplitude of the signal in the sensitive window of the network at some point. The result of sliding the network across the input data in the way described results in a SNR and p-score time series with time resolution 0.25 s. From this triggers can be generated, by choosing some threshold based on the findings on the validation set.
The testing data was generated by Dr. Alexander Harvey Nitz using PyCBC but utilizing different functions.

5.2 Evolution of the Architecture

This section gives an overview of the steps that were taken to arrive at the final architecture. It will chronologically highlight the pivotal points along the way and showcase some ideas that did not work out.
As a starting point we tried to use an easy case, where the SNR was uniformly distributed between 10 and 50, with all other parameters fixed. The neutron stars were modeled with $m_1 = m_2 = 1.4\,\mathrm{M}_\odot$. The data was stored and loaded as the sum of signal and noise and contained data for the two detectors Hanford and Livingston. In general, the data contained samples of signals and pure noise realizations. Until stated otherwise, all of the following networks use this data.
To rate the performance of a network we didn't use the sensitivity of the network yet. Instead we used the variance and mean squared error of the recovered SNR-values compared to the label values in order to estimate performance, hoping that these simple statistics correlate strongly with the actual sensitivity.

The first architecture we used was very close in nature to that of [34], halving, however, the number of filters in each convolution layer and using batch normalization in between the convolution and its activation. Therefore, we used a network of 3 stacked convolution layers, each followed by a batch normalization, ReLU activation (Never introduced the ReLU activation) and maximum pooling layer. The number of filters was doubled after each convolution layer. Since the input data to our network is sampled at multiple rates, this network has 14 input channels instead of 2. (7 input channels per detector, where each of the 7 channels corresponds to a single sample rate)
Though initially trained without pure noise samples and with only the SNR as training goal, we soon changed to use the data as mentioned in the beginning of this section. Furthermore, we added a second output. This second output gives a number between 0 and 1, where 1 corresponds to the network classifying the data as signal and 0 for classifying it as pure noise. If the output is neither 0 nor 1 one can use a threshold to determine if the output should correspond to a signal or pure noise. By default the threshold value is set to 0.5. We will refer to the results this output gives as "p-score" throughout this work, even though it is not really a probability. As this second output tries to categorize the results into two different classes, it is inefficient to use mean squared error as a loss

for this output. Instead we use a loss called categorical crossentropy, which is designed to optimize classification problems. As we are using two different losses now, the network will optimize the sum of the mean squared error from the SNR-output and the categorical crossentropy from the p-score output. This furthermore requires us to split the last layer into two.
Due to the limit statistics calculated at the time, a lot of the statements below are solely qualitative rather than quantitative.
The network was able to recover the SNR of signals rather well but has a large spread for the recovered SNR values of pure noise samples. The sensitivity can be eyeballed to reach 100% only above SNR ~ 25 and dropping close to 0% below SNR ~ 20. The latter is caused by the high SNR values assigned to certain noise samples, some reaching values of up to ~ 20. Furthermore, evaluating the second output on 3000 signals from the validation set gives an accuracy of about 95% and a false positive rate of about 6%. These numbers don't sound terrible but are in context. First of all, signals are expected to be in the SNR-range of 5-15 [Citation]. At these values the false positive rate seems to be a lot higher than the 6% over the entire validation set. Secondly, a false positive rate of 6% equates to a false alarm rate of about $6 \times 10^5 \, \frac{\text{samples}}{\text{month}}$[18]. Ideally the false alarm rate should not exceed $1 \, \frac{\text{sample}}{\text{month}}$ above an SNR of about 8. A month for this work is defined to be 30 days.

From this point the first major iteration was the introduction of inception modules [20]. They replaced the simple convolution layers of the architecture used previously. Furthermore, guided by [20], the inception modules were preceeded by 2 convolution layers. The implementation of the inception module was as a first step a direct adaptation of the original work [20]. It used kernel sizes of $(1, 3, 5)$. With these kernel sizes, results did not improve but rather got worse. Increasing the filter sizes to $(4, 8, 16)$, however, proved to be a useful change. The performance in mean squared error, variance and false positive rate improved significantly, with the false positive rate reaching $\sim 0.015\%$. By eye, the sensitivity also made an improvement as the loudest false positive was estimated to have SNR ~ 15. With this, the sensitive went up to 100% around SNR 20 and only dropped to 0% below SNR ~ 12. Though this is still not an impressive performance, it is a considerable improvement over the simple convolution approach.

Having found the new inception architecture, we conducted a test to figure out which of the 7 sample rates benefit the network the most. We found, that each sample rate, except for the 64 Hz one, benefits the results. Using only the sample rates (2048 Hz, 512 Hz, 128 Hz) gave comparable results to using the channels (2048 Hz, 1024 Hz, 512 Hz, 256 Hz, 128 Hz). Furthermore, shallower networks seemed to yield similar or improved performance, when compared to deeper ones. Shallower and deeper in this case refers to the number of stacked inception modules.

Having found that using only the sample rates (2048 Hz, 512 Hz, 128 Hz) is at least a good approximation to using all sample rates, we tested a new architecture. Previously

[18] This crude calculation uses 0.5 as a threshold value. Therefore, one gets 6×10^5 samples with the output being larger than 0.5 per month.

all sample rates were fed to the network in terms of channels of the same convolution layer. The new architecture assigned a stack of inception modules for each sample rate. This way the channels only represent the different detectors. The result of each stack are than concatenated and fed to some final layers. In the beginning the stacks for different sample rates had different depth. We found however that the network functioned best when each stack had the same depth. Furthermore, results improved when the stacks were not too deep. We settled on an architecture that had a depth of 3 inception modules in each stack, deployed another 2 inception modules after concatenation and condensed them down to the ouput size by the use of 2 dense layers per output.
As a last step, we tested the performance of using the three sample rates mentioned above against using all sample rates. Here using all sample rates improved the results significantly. Therefore, the performance quoted below is derived from the network using all sample rates. From here on out, we will call a network that uses individual stacks of layers for each sample rate a collection network. Therefore we call the kind of network described above a "collect-inception network".
The final iteration of this architecture broke the previous records of mean squared error and variance against the label values. With a false positive rate of $\sim 0.3\%$, it did perform worse than the previous record holder. This was, however, not the metric we judged the performance by at that point in time. Therefore, this network was thought of as the best one. By eye, one can also estimate that the sensitivity didn't drop to 0% even at SNR 10 and reaching 100% at SNR ~ 18. In this aspect the network improved.

The performance of this newest iteration of the network was good enough for us to move on to a more difficult data set. For this one we only changed the SNR range. Instead of varying it between 10 and 50, we went down to varying it between 8 and 15. Furthermore, we introduced sensitivity and false alarm rate as new metrics to gauge performance. We calculate both of these statistics for each of the two outputs and in the following way.
The false alarm rate is a measure that tells us how many outputs above a given value are to be expected in a month of data, when the network evaluates it. To estimate this function we need to sample

$$f(x) = \text{number of noise samples estimated louder than } x \text{ per month.} \tag{5.2}$$

To explain how we estimate this function, we will talk only about SNR. The process for the p-score, however, is equivalent.
The values for x we can sample are the predictions of the network over all pure noise samples. We will denote the number of pure noise samples by m. Each of these noise samples n_i is evaluated by the network and assigned a SNR value of x_i. To estimate $f_i \coloneqq f(x_i)$, we define $f_i' \coloneqq |\{x_j > x_i | j \in [1, m]\}|$. f_i' therefore is a number of samples. To convert it to samples per month, we calculate the observation time, i.e. the time the total number of samples m corresponds to. To do so, we observe that for each signal the waveform is allowed to move around in the noise background by $\pm 0.25\,\text{s}$. Therefore, when we evaluate a continuous time series, we need to shift the network across the data with a step-size of $0.25\,\text{s}$. With this the observation time the m samples cover

is $m \cdot 0.25\,\mathrm{s}$. Therefore, $\frac{f_i'}{m \cdot 0.25}$ is in units "samples per second". From this we find $f_i = \frac{f_i'}{m \cdot 0.25} \cdot 2,592,000 \frac{\text{samples}}{\text{month}}$ as an estimate for f. We usually plot these results in a semi-log-scale plot, as the false alarm rate drops exponentially when going to higher SNRs.
To explain the sensitivity, we will again only go into detail about how we do this in terms of SNR, as the process for the p-score is equivalent.
The sensitivity is a measure of how large the percentage of samples in a certain SNR bin are that get assigned a value larger than some threshold. We choose this threshold as the largest value that was assigned to some noise sample in our validation set, as we want to keep false alarms as low as possible. Therefore the first thing we do is find this maximal value. In a second step we look at all samples from the validation set, that contain a signal. For each of these signals the SNR value was chosen when injecting it into noise. For this reason, we can bin the samples based on this known value. Therefore, the x-values are the SNR values. (This is also true when calculating the sensitivity from the p-score.)
The corresponding y-value is the number of samples in a bin assigned a higher value than our threshold over the total number of samples in a bin. In the case that a bin is empty, we assign it the value 0.
For the first few iterations the sensitivity is calculated slightly wrong, as the p-score output was used in the case of the SNR-sensitivity to judge whether or not we were looking at a false positive. In that sense a false positive was a noise sample, where the p-score was greater than 0.5, thus impacting the threshold used. As the outputs for SNR and p-score tend to be highly correlated however, the differences was not significant and most of the values are close to the real sensitivities. The false alarm rates were also not calculated correctly for the first few runs, as they were always scaled by the same factor, regardless of how many samples the validation set actually contained. We try to not report these bad false alarm rates.

After using the previously best known architecture and taking a quick look at the sensitivity it was able to reach, we were inspired by [38] to try a new kind of network, called temporal convolutional network (TCN). Their results suggested, that a stack of dilated convolution layers outperforms normal convolution layers. Their idea was backed up by findings of [30], who showed that TCN are better suited for sequence data than other architectures. We could however not directly test their implementation as they did not reveal their entire architecture. (I don't have access to the quoted paper in its present state, so maybe they did reveal it. Can I get access?)
As we are training for SNR and p-score, one immediate problem of a pure TCN would be the output shape. This kind of network was originally designed to generate and alter sequence data and thus returns a tensor of the same length per channel as its input had. This would be great, if we could generate a SNR time series for all inputs as training goal. We would then simply get an estimate of the SNR time series as output from our network (compare [37]). It is, however, not trivial to do so and for our purposes

impractical[19]. The first intuitive solution to the problem of reshaping would simply be to use dense layers to scale down the output to the desired dimensions. We did try this approach but had no success. Instead, inspired by [39], we tried to use the TCN as a denoising stage in front of every input stack of the collect-inception-network. To do so, we fed each input to a TCN and added an auxiliary output afterwards. This output used a mean squared error loss to fit the pure waveform in the Livingston detector. The output of the TCN is than added back onto the input and fed into a stack of 3 inception modules. The result of each such stack are finally concatenated and fed into dense layers. Each TCN consisted of 12 diluted convolution layers. The number of convolution layers was chosen such that the receptive field of the TCN covers the entire length of the input, i.e. 4096 samples. The idea behind adding the TCN-output back onto the input was to amplify signals, whilst not throwing away information, if the TCN was not able to recover a signal.
Using the TCN as a denoiser required us to change the way the samples are fed to the network, as we needed the pure GW-signal for the training goal. Thus, instead of feeding the network the sum of signal and noise, it receives the two parts separately and adds them together on the first layer. When evaluating real samples the network will be fed the sum of signal and noise instead. This is not a problem though, as one of the two inputs can simply receive zeros as input. This is a change to the data generation we decided to keep. For a detailed reasoning as to why we decided to change to this new system see subsection 5.1.
The performance of this network exceeded those of any previous network by a considerable margin. Now judging by sensitivity rather than mean squared error or variance, it dropped below 80% sensitivity only for SNRs smaller 12 and didn't considerably drop below 20% at all. The sensitivity improved especially at low SNRs. Using $37,500$ noise samples the false alarm rate can be resolved up to about SNR 9, as that is the value of the loudest sample. Below SNR 9 the false alarm rate is rather high with a lowest value of about $300\,\frac{\text{samples}}{\text{month}}$. Using more noise samples would enable us to finer resolve the false alarm rate. As it is not feasible during development, to evaluate each individual network on a large set of data, we will only use the sensitivity to get a rough estimate of the performance and compare different algorithms in that regard. For the final network a large testing set will be used in order to more finely resolve the false alarm rate and to make stronger statements on the sensitivity.
From here on out we will refer to a network of the kind described in this part as TCN-collect-inception network or TCIN. Though it improves almost all statistics we are using, when compared to a standard collect-inception network, it comes at a cost. This cost is the memory the network needs in order to be trained. In the form used for these results, it used 30 GB of video memory on a NVIDIA GV100 while already reducing the mini-batch size to only 24. This graphics card is currently the only one that has enough memory to support such an architecture. The availability of this hardware is

[19]Defining a SNR time series for data that is sampled at multiple rates is the technical difficulty. The impracticality does only come in, when we store noise and signals separately, as we would need to generate the SNR time series on the fly, which is computationally expensive.

very limited and thus does not allow for rapid development. A smaller alternative would therefore be beneficial.
We also conducted a test, trying to determine if the TCN is able to reconstruct the original waveform from a noisy time series. These tests showed that it is not consistently able to do so. Using mean average percentage error as a loss function and setting the noise free signal as the training goal for a pure TCN, the loss did not fall below $70,000$. Trying to train the TCIN without the use of an auxiliary loss function, however, proved to deteriorate performance. Therefore, other TCINs use the auxiliary outputs as well.

After the success of TCINs, we searched for simpler architectures to reduce memory constrains. For that reason we tried using an old record holder, a simple inception network that was very memory efficient. To improve its preformance, residual connections were added to every inception layers where it was possible to do so without needing to use dimensional reduction layers. This choice was motivated by a conversation with Christoph Dreißigacker and the results of [21]. The residual connections are meant to help the network learn as it is more easily able to recover the identity mapping between the input of an inception module and its output. The total network consisted of 8 stacked inception modules with one intermediate pooling layer and 6 residual connections. Networks using mainly inception modules and combining them with residual connections will be called inception-res networks from here on out. The inception modules were preceded by two convolution layers. The different sample rates were fed to the network as different channels of the same input layer. It also only used the three sample rates (2048 Hz, 512 Hz, 128 Hz). By accident, the kernel sizes in the inception modules were set to $(1, 2, 3)$. This did, however, prove to be beneficial to the network.
The results this network produced were incredible and outperformed anything we had seen so far. Sensitivities stayed close to or above 60% for all SNR bins and the loudest noise sample was estimated at an SNR of ~ 7.8. One thing that seemed strange though, was the plot of recovered SNRs against the label values. For some specific SNR values, the predictions were scattered but lower by a significant margin. This can be seen as streaks when plotting the recovered against the label values, as done in Figure 5.2.

The observed behavior and the incredible performance was due to a bug in how the network was fed with data. The generator combines a pure waveform with one noise realization to create a signal sample and is supposed to combine an array of zeros with a noise realization for a pure noise sample. A missing "if"-statement led to the array of zeros always being replaced with the waveform that was the last in the list of available waveforms. Therefore, instead of pure noise, the network always saw noise plus one specific waveform. The same error with the exact same waveform was present in both the training and the validation set which made this error hard to catch. In fact the error was only discovered about two weeks after it first occurred. Finding the bug this late led to the optimization of the architecture for this flawed data. The best performing network had a sensitivity of more than 95% over the entire SNR range for this flawed data, even once we used more difficult data, i.e. varying more parameters than just the SNR.
Though this mistake might at first seem like a waste of time, it actually helped in two

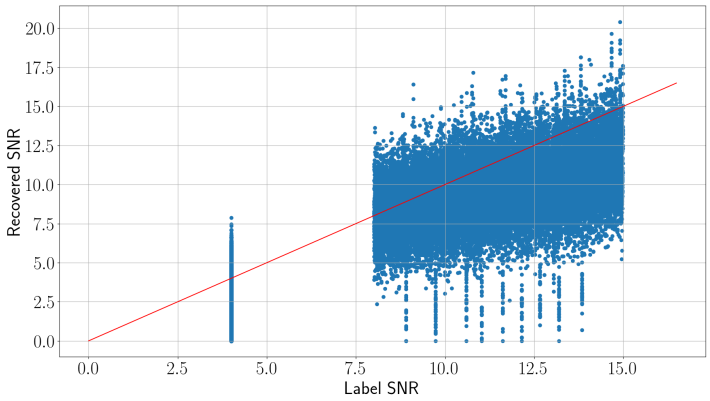


Figure 5.2: Shown are the recovered SNR values on the y-axis and the label values on the x-axis. Each SNR label has multiple y-values assigned to it, as the same waveform is submerged in multiple different noise realizations. The red line indicates where the dots should lie if the network recovered them without error. The dots roughly follow this line with the exception of a few streaks going down to 0. These are the waveforms that resonate strongly with the signal that was mistakenly added to every pure noise sample. Later iterations of the architecture used to produce this plot reduced the number of streaks down to a single one.

aspects. Firstly, it showed the possibility for a network to consistently recover a single sample from different noise realizations and being able to model it well enough. The hurdle for a general search isn't trivial from this point, as in an optimal case we would expect the network to generalize the structure of the waveforms it was trained on and interpolate a template bank, but it is a starting point. Secondly, the optimization led to architectures with improved performance over for instance the TCINs, when evaluated on non-flawed data. For this reason, we will list a few of the notable improvement here.

The first observation was that the mistake of using filter sizes $(1, 2, 3)$ was actually an improvement over using $(4, 8, 16)$, which was previously used and found to be beneficial. Another feature introduced was the reduced number of input samples. Before the different sample rates had some overlap in the time domain, i.e. the last second of the data was sampled by all 7 rates, the last 2 seconds were sampled by all but the highest sample rate and so on. As the higher sample rates also contain all the information the lower sample rates do, this overlap was thrown away. For a detailed description of how this non overlapping multi-rate sampling works see subsection 5.1. A study, testing the two different approaches to sampling the data and reducing the number of input samples by simply using less sample rates, showed that using all sample rates is beneficial, while cropping the overlapping part of the data has no considerable negative effect.

With the performance these architectures suggested, we moved to more difficult data, altering not only the SNR but also component masses m_1, m_2, coalescence phase Φ_0, sky-position θ, φ and inclination ι. As expected, the performance decreased a bit using more difficult data. Trying to recover the old performance led to one of the final improvements to the architecture. Instead of using a single inception stack, we introduced

an architecture that is a mixture of the collect-inception networks and the pure inception networks. It uses the same inputs as the collect-inception networks, but cascades down concatenating two stacks after two inception layers, reducing the initial 8 stacks to 4. These 4 stacks each are fed through two further inception layers before being concatenated again. This procedure is repeated a third time, which results in a single stack of inception layers which is than fed to dense layers, in order to get the outputs. See Figure 5.3 for an overview of the architecture. We will still refer to networks with a similar structure as collect-inception-res networks.
It again improved results a by a little. The sensitivity now didn't drop below 97% for any of the bins, with the loudest noise sample having an SNR ~ 8. This is the last and best performing iteration of the network we found before the bug in the generating process was discovered and fixed. It therefore is still one of the core structures of the final architecture.

Using the corrected data generator with the previously well performing inception-res networks on data, where all previously mentioned parameters were being varied, showed how big an impact the error made. The sensitivity dropped from 95% in every SNR bin to below 20% in the loudest bin. The collect-inception-res network held up a bit better, dropping to 40% sensitivity in the loudest bin. From this point we tried to recover some of the lost performance by developing new features and implementing them into the collect-inception-res network.

One of the first tests was training the network to just optimize the p-score rather than both SNR and p-score, as this second output usually performed a little better than the SNR output. This did however not work and decreased the performance of the network significantly, reaching only $\sim 12\%$ at the loudest point. Using the original architecture with both outputs but adding in one more inception module per stack also resulted in worse performance overall. This decrease, however, was not quite as significant, dropping only to 30% in the loudest bin.

One of the prominent problems the network has is that estimating pure noise with a large SNR value significantly decreases sensitivity. An overestimated noise sample is a lot worse than an underestimated signal, as the former has an effect on all SNR bins. Therefore we tried to implement a loss function that mimics this behavior, by growing exponentially for overestimating noise and only linearly for underestimating it. For signals this behavior is mirrored, i.e. the loss grows exponentially for underestimating signals and linearly for overestimating them. The details on how this loss looks like and how it was derived can be found in Appendix B. It turned out that this approach also didn't help but rather decreased the sensitivity of the network. It even seemed to push the two categories closer together. Further research in this area might prove useful though.

Another approach we took was close in nature to that of SincNet [40]. SincNet convolves the input data with a parameterized filter, that is used in standard signal processing. This has two main advantages. Firstly, the filters the network applies become more

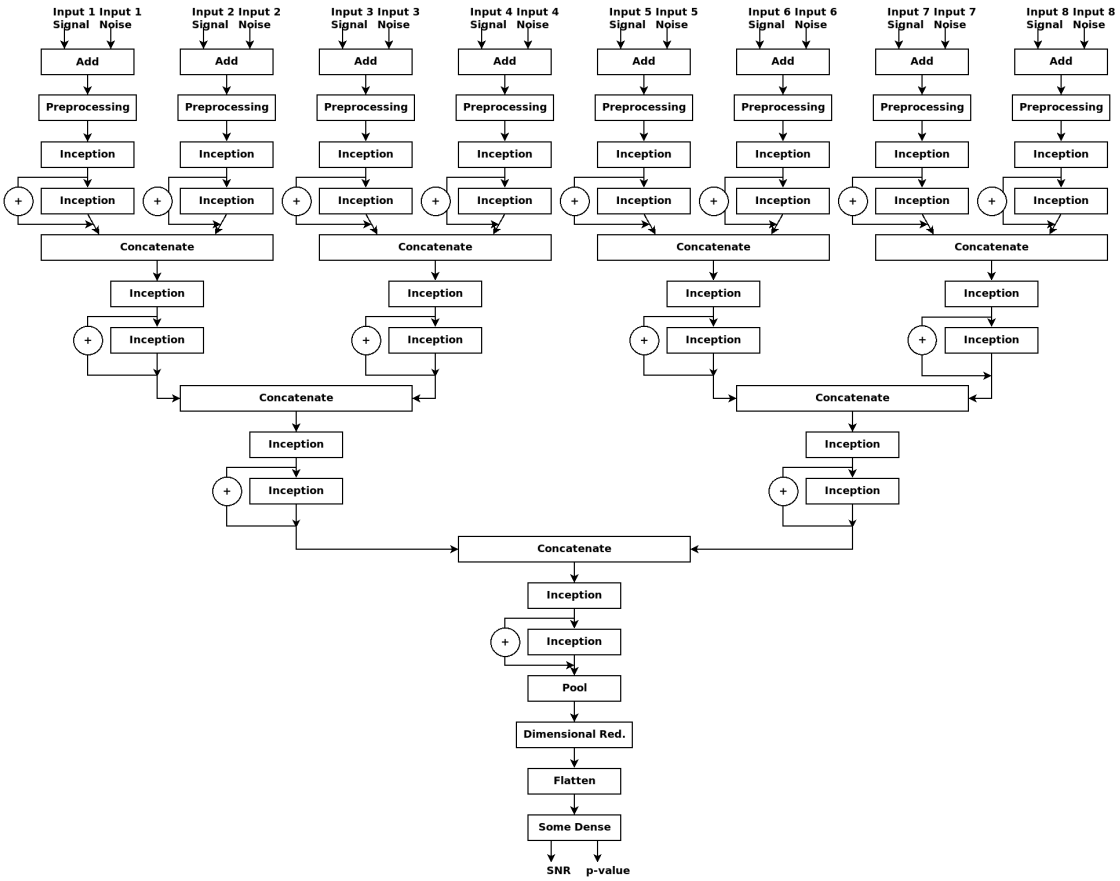


Figure 5.3: The rough architecture of a cascading collect-inception-res network is shown. The preprocessing layers contain 2 convolution layers as well as some dropout and batch normalization. The block labeled "Some Dense" actually consists of two separate stacks of dense layers, that reduce the output of the last inception layer down to the appropriate size. Work on this graphic and make it a bit larger.

interpretable and thus reveal more insight into what the network is actually trying to accomplish. Secondly, the number of trainable parameters is reduced as we don't need to train a large convolution kernel but rather some parameters of a known function. Their implementation used the parametrized layer only as the first layer of the architecture and showed improvements over other purely convolutional approaches when trying to recognize speakers [41]. Based on this work we used our adaptation only as a first layer, too. Instead of using Sinc-functions, we tried to use sums of sine waves. The convolution kernel is thus given by $\sum_{i=1}^{n} A_i \cdot \sin(f_i t + \varphi_i)$, where A_i, f_i and φ_i are learned parameters and t is used to sample the function. The general idea behind this approach is to enable the network to construct a crude template of a general waveform by combining multiple sine waves. To restrict the available frequencies, the layer is given a low and high frequency cutoff. This restriction is necessary to determine the shape of the convolution kernel. To sample a sine wave of frequency f, according to Nyquist, one needs to sample at least with a rate of $2f$. Therefore the sample rate is given by $\mathrm{d}t = \frac{1}{2f_{\text{high}}}$. As we want to fit at least one complete cycle of the lowest frequency into our kernel, the number of samples in the kernel is given by $N = \frac{f_{\text{low}}}{2f_{\text{high}}}$. By default the convolution kernel will always be filled with samples, e.g. a sine wave of frequency $f = 2f_{\text{low}}$ will manage two full cycles in the convolution kernel. We called this layer "Wave-Convolution" or "WConv1D" in short and will from here on out refer to it by this name.
To get a quick estimate of how well this new approach does we devised a simple network consisting of only two convolution-type layers and two dense layers to get the correct output. We than compare two iterations of this network, one trained with the new Wave-Convolution layer and one using a usual convolution layer. All other parameters are kept constant. For the convolution layer, we choose the kernel size in such a way that the number of trainable parameters is on a comparable scale.
First tests of this new layer did apply some false windowing and used a skewed methodology. The results produced by these early tests lost out to pure convolution quite strongly and thus this approach was not further pursued. Revisiting the idea close to finishing this project and fixing the errors previous runs made paints a different but not clear picture. Now the WConv1D layer significantly outperforms the traditional convolution layer, almost doubling the sensitivity in every bin. The results are however questionable in multiple ways. Firstly, both networks experienced strong overfitting, thus rendering the training almost meaningless. Therefore, the difference in performance may be down to pure chance. Secondly, the networks were only followed by a single convolution layer before being fed into the final dense layers. The test setup therefore doesn't consider the impact on deep networks. Furthermore, the convolution kernel used for the convolution network was of size 288, which is well beyond the usual kernel sizes of 16 used in CNNs designed for filtering GW data. It is thus questionable if the extra effort required would justify the use of a WConv1D layer. Finally, the layer was not compared in state of the art networks derived above. Therefore, a lot more research would be necessary to determine the use of this new approach.

As no architectural alteration proved to be highly beneficial to the performance of the network for the difficult data where a lot of parameters are varied, we went back to using

the data that only varies the SNR. We than combined the general approach of the TCIN with the new cascading architecture and found that it outperforms the traditional TCIN. This combination is the final architecture we tried and will be explained in further detail in subsubsection 5.3.1. We then used this final architecture and trained it on the data described in subsection 5.1 to finally evaluate it.

We conclude this section with a few statements about general findings of our research. Using dropout layers in the initial layers after normalizing the input generally had a positive effect on the achievable sensitivities, even when the number of input samples was chosen large enough to avoid overfitting. It generally however comes with the disadvantage of the loss being a lot more jumpy throughout the training and trends in the loss being washed out.
We used the SGD variant "Adam" as an optimizer for most of our runs and did not see improvements when trying different ones like "AdaDelta" or standard SGD. We also briefly tried modifying the learning rate but could not improve results, thus staying with the default learning rate of 10^{-3}.
Finally, when training a network and monitoring the sensitivity throughout the training process, we noticed drops to zero sensitivity at some point. These got more frequent as training went on. We hypothesize that this drop is due to the network trying to split noise and signals rather strongly. If one of the noise samples resonates strongly with the network and it "thinks" to have seen a signal, it will push the value for this sample higher and higher. The p-score additionally will saturate at 1. We therefore recommend to stop training the network after a few of these dips to 0 sensitivity occurred.

5.3 Final Network

Talk about how the final network looks, how it performs and what could be imporved.
First final network stored at tcn_collect_inception_res_net_rev_6_248201905643
Trying to optimize a NN to a specific task is a problem that has no specified end. Therefore, the architecture discussed in this section is only the best of our current efforts. We will start off by discussing it and the underlying design decisions in detail and afterwards evaluate the performance on a long set of continuous strain data.

5.3.1 Architecture

Explain the architecture and the reasoning behind it. Talk about the size of the model, where it could be trained and what the drawbacks of the architecture are. (Drawbacks: Large memory size (hence small batch size), very deep → slow training and maybe vanishing gradients, hyper-parameter-optimization is hard, not everywhere are residual connections (fixable by further dimensional reduction))

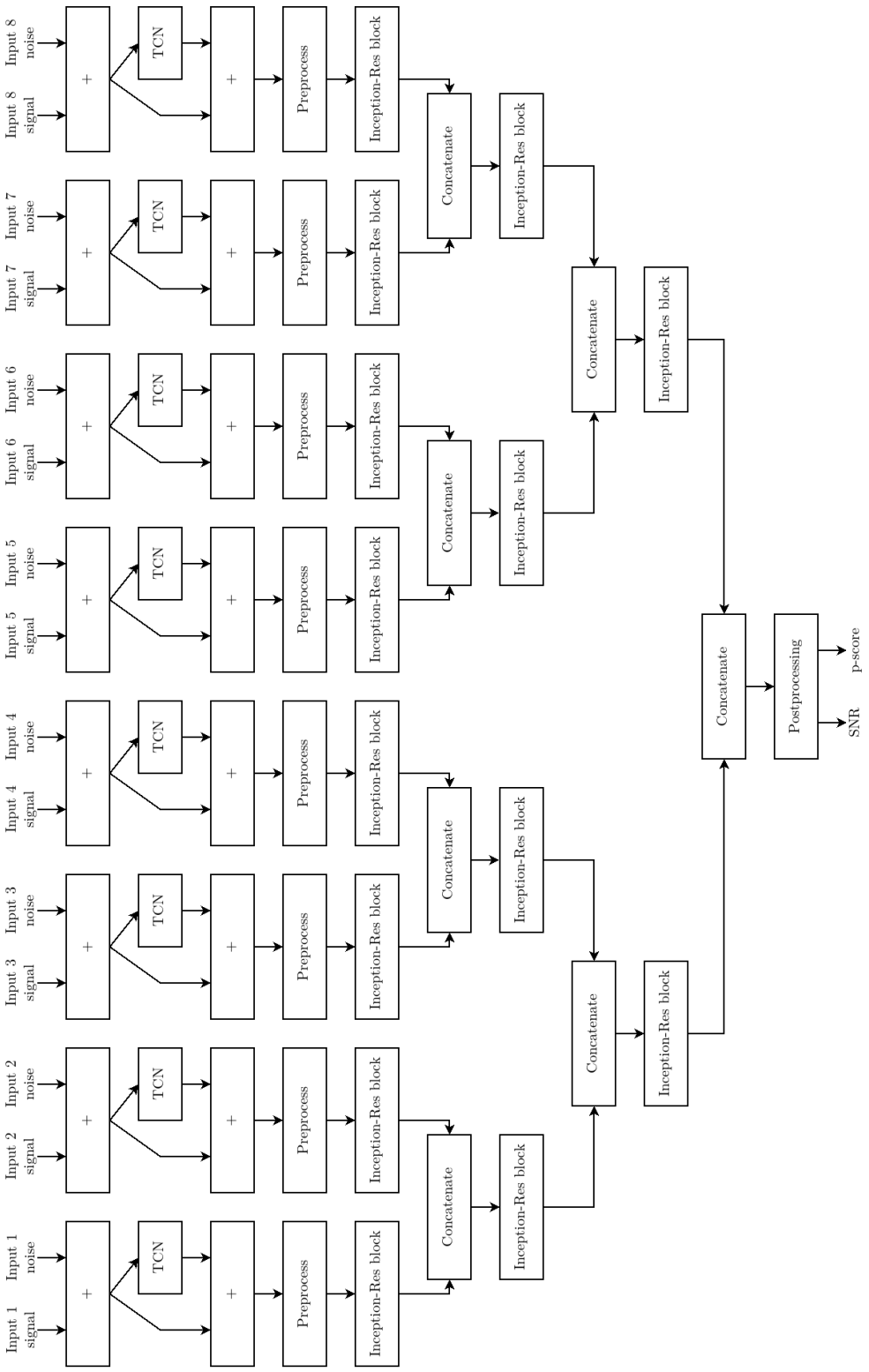


Figure 5.4: Description, need to show auxiliary outputs

5.3.2 Network Performance

Evaluate the performance of the network. Show sensitivity curves, talk about speed advantages, how does it in both cases compare to matched filtering? How does it compare to related works? (Reference the BNS-Net paper, what is different between our approach and theirs? Why does theirs seem to work a lot better? Does it?)

6 Conclusion

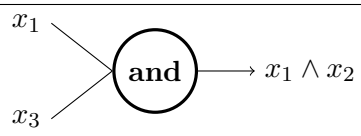
Well duh, give a conclusion and maybe outlook.

7 Acknowledgments

Acknowledge Frank, Alex, Christoph, AEI, spell and grammar checkers.
I thank Dr. Frank Ohme and Dr. Alexander Harvey Nitz for the supervision of this project and their constant support. Their advice and expertise has been crucial for this work.
I am especially thankful to Christoph Dreißigacker, who always offered advise. Constant constructive conversations with him also led to many improvements along the way. He furthermore pointed out several mistakes in drafts of this thesis.
Furthermore, I would like to thank Pascal Auerwald, Lennart Janshen, Maximilian Raimer and Nadine Speer for reading parts of this thesis, pointing out mistakes and smoothing out the flow of the sections.

A Full Adder as Network

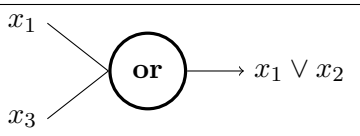
To create a full adder from basic neurons, the corresponding logic gates need to be defined. The equivalent neuron for an "and"-gate was defined in subsection 3.1. There are two more basic neurons which will be defined here. The neuron corresponding to the "or"-gate, which is given by the same activation 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

$\vec{w} = (1, 1) \quad b = -1.5$

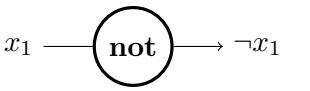
x_1	x_2	$a(x_1 + x_2 - 1.5)$
0	0	0
0	1	0
1	0	0
1	1	1



"or"-neuron

$\vec{w} = (1, 1) \quad b = -0.5$

x_1	x_2	$a(x_1 + x_2 - 0.5)$
0	0	0
0	1	1
1	0	1
1	1	1



"not"-neuron

$w = -1 \quad b = 0.5$

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	$x_1 \veebar x_2$
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.

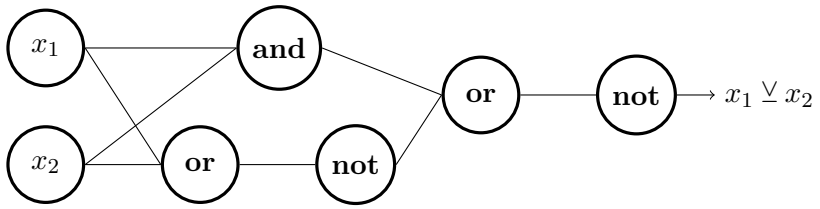


Figure A.1: The definition of a network that is equivalent to an "XOR"-gate.

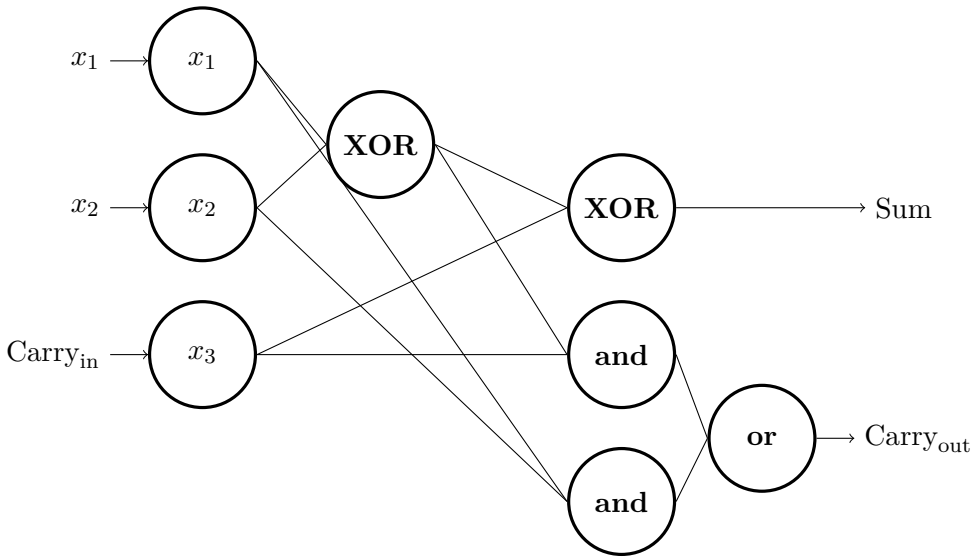


Figure A.2: A network replicating the behavior of a binary full adder.

B 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

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

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

for $x = 1$, which gave

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

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

$$\text{dist}(x) \coloneqq f(x + 2) \tag{B.4}$$

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

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

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

$$\text{Err}_{\text{rotate}}(x, \varphi) \coloneqq \text{Err}(x / \cos(\varphi)). \tag{B.6}$$

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 (B.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

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

with

$$p(x) \coloneqq 3x^2 - 2x^3. \tag{B.8}$$

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

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

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

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

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 (B.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. We restrict $k > 1$.
With

$$\begin{aligned} u &= \text{Err}(s \cdot k) \\ l &= \text{Err}(s \cdot (k-1)) \\ \Delta_1 &= \frac{4s^2(k-1)}{(2 + s \cdot (k-1))^3} e^{s \cdot (k-1)} \\ \Delta_2 &= s\frac{4}{e} \\ a_1 &= \Delta_2 + \Delta_1 - 2(u - l) \\ a_2 &= 3(u - l) - 2\Delta_1 - \Delta_2 \end{aligned} \tag{B.11}$$

and

$$p_2(z) \coloneqq a_1(z - k + 1)^3 + a_2(z - k + 1)^2 + \Delta_1(z - k + 1) + l \tag{B.12}$$

one gets

$$L_{\text{large}}(y_{\text{net}}, y_{\text{label}}) \coloneqq \begin{cases} \text{Err}_{\text{rotate}}(s \cdot z, g(y_{\text{label}}, a, b)), z > -k + 1 \\ p_2(-z), z \in [-k, -k + 1) \\ s\frac{4}{e}|z| + \text{Err}_{\text{rotate}}(-s \cdot k, g(y_{\text{label}}, a, b)) - s \cdot k\frac{4}{e}, z < -k \end{cases} \tag{B.13}$$

and

$$L_{\text{small}}(y_{\text{net}}, y_{\text{label}}) \coloneqq \begin{cases} \text{Err}_{\text{rotate}}(s \cdot z, g(y_{\text{label}}, a, b)), z < k - 1 \\ p_2(z), z \in [k - 1, k) \\ s\frac{4}{e}|z| + \text{Err}_{\text{rotate}}(s \cdot k, g(y_{\text{label}}, a, b)) - s \cdot k\frac{4}{e}, z > k \end{cases} . \tag{B.14}$$

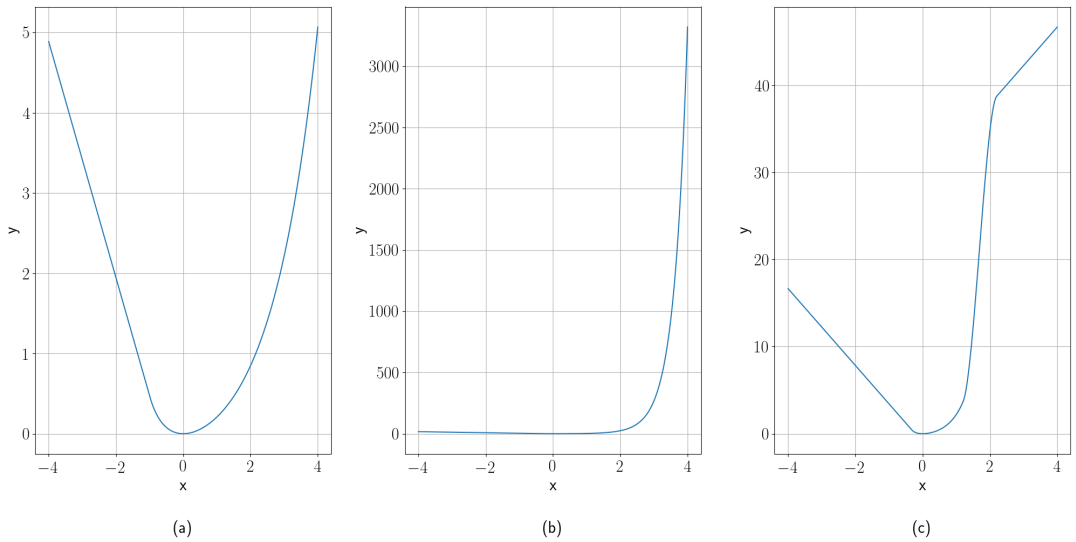


Figure B.1: Three different stages of the custom loss function. (a): The loss as defined in (B.9). For $|x| < 2$ the eyponential part is actually smaller than the linear part. This is not desirable, as for most of our testing the label for pure noise is about 4 away from the smallest label for a signal. (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 (B.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.

The complete loss is than given by

$$L_{\text{full}}(y_{\text{net}}, y_{\text{label}}) = \begin{cases} L_{\text{small}}(y_{\text{net}}, y_{\text{label}}), y_{\text{label}} < \frac{a+b}{2} \\ L_{\text{large}}(y_{\text{net}}, y_{\text{label}}), y_{\text{label}} > \frac{a+b}{2} \end{cases} . \tag{B.15}$$

For this work, we choose values $a = 4, b = 8$ and $k = 2.2$.

Glossary

BBH Binary black hole.

BNS Binary neutron star.

CNN Convolution neural network.

CNNs Convolution neural networks.

EM Electromagnetic.

FFN feed forward (neural) network.

GW gravitational wave.

GWs gravitational waves.

ILSVRC ImageNet Large Scale Visual Recognition Challenge.

ISCO innermost stable circular orbit.

MSE mean squared error.

NN Neural network.

NNs Neural networks.

PM Post-Minkowski'sche Näherung.

PN post Newtonian approximation.

PSD Power spectral density.

RNN recurrent neural network.

RNNs recurrent neural networks.

SGD Stochastic gradient descent.

SNR signal to noise ratio.

TCIN TCN-collect-inception network.

TCN Temporal convolutional network.

TT transversal-traceless gauge.

References

[1] B. P. Abbott et al. "Observation of Gravitational Waves from a Binary Black Hole Merger". In: *Phys. Rev. Lett.* 116 (6 Feb. 2016), p. 061102. DOI: 10.1103/PhysRevLett.116.061102. URL: https://link.aps.org/doi/10.1103/PhysRevLett.116.061102 (cit. on p. 1).

[2] B. P. Abbott et al. "Tests of General Relativity with GW150914". In: *Phys. Rev. Lett.* 116 (22 May 2016), p. 221101. DOI: 10.1103/PhysRevLett.116.221101. URL: https://link.aps.org/doi/10.1103/PhysRevLett.116.221101 (cit. on p. 1).

[3] LIGO Scientific Collaboration, Virgo Collaboration, et al. "Binary Black Hole Population Properties Inferred from the First and Second Observing Runs of Advanced LIGO and Advanced Virgo". In: *arXiv preprint arXiv:1811.12940* (2018). arXiv: 1811.12940. URL: https://arxiv.org/abs/1811.12940 (cit. on p. 1).

[4] Daniel E. Holz and Scott A. Hughes. "Using Gravitational-Wave Standard Sirens". In: *The Astrophysical Journal* 629.1 (Aug. 2005), pp. 15–22. DOI: 10.1086/431341. URL: https://doi.org/10.1086%2F431341 (cit. on p. 1).

[5] J Aasi et al. "Advanced LIGO". In: *Classical and Quantum Gravity* 32.7 (Mar. 2015), p. 074001. DOI: 10.1088/0264-9381/32/7/074001. URL: https://doi.org/10.1088%2F0264-9381%2F32%2F7%2F074001 (cit. on p. 1).

[6] F Acernese et al. "Advanced Virgo: a second-generation interferometric gravitational wave detector". In: *Classical and Quantum Gravity* 32.2 (Dec. 2014), p. 024001. DOI: 10.1088/0264-9381/32/2/024001. URL: https://doi.org/10.1088%2F0264-9381%2F32%2F2%2F024001 (cit. on p. 1).

[7] LIGO Scientific Collaboration, Virgo Collaboration, et al. "GWTC-1: a gravitational-wave transient catalog of compact binary mergers observed by LIGO and Virgo during the first and second observing runs". In: *arXiv preprint arXiv:1811.12907* (2018). arXiv: 1811.12907. URL: https://arxiv.org/abs/1811.12907 (cit. on p. 1).

[8] B. P. et. al Abbott. "GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral". In: *Phys. Rev. Lett.* 119 (16 Oct. 2017), p. 161101. DOI: 10.1103/PhysRevLett.119.161101. URL: https://link.aps.org/doi/10.1103/PhysRevLett.119.161101 (cit. on pp. 1, 40).

[9] 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. 2, 32, 33).

[10] T. Young et al. "Recent Trends in Deep Learning Based Natural Language Processing [Review Article]". In: *IEEE Computational Intelligence Magazine* 13.3 (Aug. 2018), pp. 55–75. ISSN: 1556-603X. DOI: 10.1109/MCI.2018.2840738 (cit. on p. 2).

[11] Daniel George and E. A. Huerta. "Deep neural networks to enable real-time multimessenger astrophysics". In: *Phys. Rev. D* 97 (4 Feb. 2018), p. 044039. DOI: 10.1103/PhysRevD.97.044039. URL: https://link.aps.org/doi/10.1103/PhysRevD.97.044039 (cit. on pp. 2, 38).

[12] François Chollet et al. *Keras*. https://keras.io. 2019 (cit. on pp. 2, 40).

[13] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on pp. 2, 40).

[14] Alex Nitz et al. *gwastro/pycbc: PyCBC Release v1.13.5*. Mar. 2019. DOI: 10.5281/zenodo.2581446. URL: https://doi.org/10.5281/zenodo.2581446 (cit. on pp. 2, 40, 41).

[15] Marlin Schäfer. "Massenbestimmungen kompakter Binärsysteme durch Analyse von Gravitationswellen". Apr. 2018 (cit. on pp. 4, 5).

[16] Michele Maggiore. *Gravitational Waves. Volume 1: Theory and experiments*. Oxford University Press, 2008. ISBN: 978-019-857074-5 (cit. on pp. 5, 6, 8, 9, 10, 12, 14, 15).

[17] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/index.html (cit. on pp. 16, 20, 26).

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org (cit. on pp. 16, 18, 19, 20, 21, 24, 25, 26, 32, 35, 37).

[19] 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. 26).

[20] 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. 26, 32, 33, 45).

[21] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016. URL: http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html (cit. on pp. 26, 49).

[22] 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. 28).

[23] 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. 30).

[24] 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. 30).

[25] 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. 32).

[26] 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. 32).

[27] 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. 32).

[28] 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. 33).

[29] 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. 33).

[30] 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. 34, 35, 47).

[31] 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. 35).

[32] 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 p. 36).

[33] 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. 37).

[34] Daniel George and E.A. Huerta. "Deep Learning for real-time gravitational wave detection and parameter estimation: Results with Advanced LIGO data". In: *Physics Letters B* 778 (2018), pp. 64–70. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2017.12.053. URL: http://www.sciencedirect.com/science/article/pii/S0370269317310390 (cit. on pp. 38, 44).

[35] Christoph Dreissigacker et al. "Deep-learning continuous gravitational waves". In: *Phys. Rev. D* 100 (4 Aug. 2019), p. 044009. DOI: 10.1103/PhysRevD.100.044009. URL: https://link.aps.org/doi/10.1103/PhysRevD.100.044009 (cit. on p. 38).

[36] Plamen G. Krastev. "Real-Time Detection of Gravitational Waves from Binary Neutron Stars using Artificial Neural Networks". In: (2019). arXiv: 1908.03151. URL: https://arxiv.org/abs/1908.03151 (cit. on pp. 38, 39).

[37] Timothy D Gebhard et al. "Convolutional neural networks: a magic bullet for gravitational-wave detection?" In: *arXiv preprint arXiv:1904.08693* (2019). arXiv: 1904.08693. URL: https://arxiv.org/abs/1904.08693 (cit. on pp. 39, 47).

[38] Alexander Schmitt et al. "Investigating Deep Neural Networks for Gravitational Wave Detection in Advanced LIGO Data". In: *Proceedings of the 2Nd International Conference on Computer Science and Software Engineering*. CSSE 2019. Xi'an, China: ACM, 2019, pp. 73–78. ISBN: 978-1-4503-7172-8. DOI: 10.1145/3339363.3339377. URL: http://doi.acm.org/10.1145/3339363.3339377 (cit. on pp. 39, 47).

[39] Wei Wei and E. A. Huerta. "Gravitational Wave Denoising of Binary Black Hole Mergers with Deep Learning". In: (2019). arXiv: 1901.00869. URL: https://arxiv.org/abs/1901.00869 (cit. on pp. 39, 48).

[40] Mirco Ravanelli and Yoshua Bengio. "Interpretable convolutional filters with SincNet". In: *arXiv preprint arXiv:1811.09725* (2018). arXiv: 1811.09725. URL: https://arxiv.org/abs/1811.09725 (cit. on p. 51).

[41] Mirco Ravanelli and Yoshua Bengio. "Speaker recognition from raw waveform with sincnet". In: *2018 IEEE Spoken Language Technology Workshop (SLT)*. IEEE. 2018, pp. 1021–1028. DOI: 10.1109/SLT.2018.8639585 (cit. on p. 53).