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 30, 2019

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

Abstract

Gravitational waves are now observed routinely. Therefore, data analysis has to keep up with ever improving detectors. One relatively new tool to search for gravitational wave signals in detector data are machine learning algorithms that utilize deep neural networks. The first successful application was able to differentiate time series strain data that contains a gravitational wave from a binary black hole merger from data that consists purely of noise. This work expands the analysis to signals from binary neutron star mergers, where a rapid detection is most valuable, as electromagnetic counterparts might otherwise be missed or not observed for long enough. We showcase many different architecture, discuss what choices improved the sensitivity of our search and introduce a new multi-rate approach. We find that the final algorithm gives state of the art performance in comparison to other search pipelines that use deep neural networks. On the other hand we also conclude that our analysis is not yet able to achieve sensitivities that are on par with template based searches. We report our results at false alarm rates down to $\sim 30 \frac{\text{samples}}{\text{month}}$ which has not been tested by other neural network algorithms. We hope to provide information about useful architectural choices and improve our algorithm in the future to achieve sensitivities and false alarm rates that rival matched filtering based approaches on.

Contents

List of Figures

List of Tables

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 for the first time. The first and second observation runs 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 [8].

The most promising source of GWs that can be detected 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 is GW170817 [7], which, instead was emitted by a binary neutron star (BNS) system [9]. As such, it is one of the most interesting signals detected so far. It is not only the first and only signal of its kind observed yet, but it was also possible to detect the 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 internal dynamics and structure of neutron stars [10]. As detectors become more sensitive with each technological improvement, BNS signals are expected to be detected more frequently in the future.

In order to detect the associated EM counterparts, 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 [9]. The source was then visible for another 48 h [10]. While it is unlikely that a fast GW detection will enable astronomers to catch the associated γ-ray burst, it will help to maximize observation time in the optical- and x-ray regime. It is therefore vital to reduce latency as much as possible and find detection candidates reliably, as telescope time is expensive. To reduce costs even further an estimate of the sky position needs to be provided as well. Some EM counterparts would also be missed if telescopes are not provided with a rough location as their field of view is often very narrow.

Most of the current pipelines that try to identify GWs within the detector data use the concept of matched filtering where a fixed number of pre-calculated GW templates are used to search for similar patterns in the detector data [11]. These templates cover the area of the parameter space of binary systems that are expected to exist and be detectable. The sensitivity of this search to GWs, however, is directly dependent on the spacing of templates in this high dimensional parameter space. This is due to the fact that if a signal lies between two templates it gets assigned a lower detection statistic. As the knowledge about binary systems and their dynamics improves and as more accurate waveform models are developed, the template bank will grow in size. 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.

One of the possible contenders to aid matched filtering in the first data analysis stage is machine learning. It is a field of computer science with the goal 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 have application in many fields, like computer vision [12], sound generation [13] or natural language processing [14]. The advantages of NNs are manifold, the most important one in the context of this thesis being computational efficiency once the network is optimized. This efficiency and their general success in almost any area make NNs a promising tool 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 to recover GW signals. They were able to reach performances comparable to those of matched filtering at a fraction of the computational cost [15]. Their network, however, was only optimized for signals from binary black hole (BBH) mergers, thus not covering the cases of BNS signals where quick notifications are most valuable.

This thesis builds on the work of [15] 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 have a lower strain amplitude, 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 rates at which the data is sampled. To get a first hold of the problem, we are ignoring spins and tidal deformabilities of the neutron stars. We expect the effects of this approximation to be small, as spins are thought to be close to zero and the impact of deformation due to tidal effects minimal. The algorithm takes a continuous stretch of whitened strain amplitude time series data and generates two output time series from it; a signal-to-noise ratio (SNR) time series and a p-score[1] time-series. To both of these a threshold at fixed false-alarm rate is applied to generate detection candidates.

This thesis is structured as follows: Section 2 and section 3 give a summary of the required background knowledge. Specifically, section 2 gives a brief overview of the theory involved with modeling and detecting GWs, whereas section 3 describes how NNs work and introduces the concepts needed to understand the final algorithm. Section 4 gives a deeper motivation to the problem we are trying to analyze and puts this thesis into greater context of related works. Section 5 contains the results of our research and goes into detail about the design decisions that went into our final network.

To design and train our networks, we use version 2.2.4 of the software library Keras [16]. The former is a wrapper for the deep learning library Tensorflow [17], of which we use the GPU optimized version 1.13.1 for training. To evaluate our networks, we use version 2.2.5 of Keras and version 1.14.0 of the CPU based implementation

---

[1]The p-score must not be confused with a p-value. Both have in common that they are normalized to 1 and as such the p-score gives values in the range $[0, 1]$. It does, however, not fulfill any other requirements and is thus not a probability.

of Tensorflow. To generate fake data, we use version 1.13.5 of the software package PyCBC [18]. All code related to this thesis is open source and can be found at https://github.com/MarlinSchaefer/master_project.

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 [2, 19, 20]. They are however also very hard to detect, as binary neutron star systems are very light, when compared to inspiraling binary black holes.

Part 1 of this section will discuss how gravitational waves are modeled 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.

Notationally, we use the Einstein sum convention, denote 4-dimensional space time indices by greek and spatial indices by latin letters. By convention we choose $\eta^{\mu\nu} = \text{diag}(-, +, +, +)$ for the flat space time metric.

2.1 The Waveform

A gravitational wave generally consists of three stages, an inspiral, a merger and a ringdown phase. In the first phase, the two bodies spiral slowly towards each other following quasi-circular orbits. Once they are close enough though, they will not be able to stay on these circular paths and plunge towards each other. This plunge results in the two bodies merging. Therefore, this phase is commonly referred to as "merger". After they collided the remnant will radiate off some more energy. This is the so-called ringdown phase.

We will see that frequencies of GWs coming from a BNS system are very high due to their low total mass. As such even the inspiral phase will contain frequencies higher than the ones the detectors are sensitive to. For this reason we only model the inspiral phase in the following subsections.

2.1.1 Linearized Gravity

Most calculations of this section are layed out in detail in [21].

Gravitational waves are a prediction of the Einstein-equation

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

if a weak field is assumed. Here $\mathcal{G}_{\mu\nu}$ 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. The weak field limit is given by

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

where $g_{\mu\nu}$ is the metric of curved space time, $\eta_{\mu\nu} = \text{diag}(-, +, +, +)$ is the metric of flat space time and $h_{\mu\nu}$ is a small perturbation, with $|h_{\mu\nu}| \ll 1$ and $|\partial_{\sigma_1} \ldots \partial_{\sigma_n} h_{\mu\nu}| \in$

$\mathcal{O}(|h_{\mu\nu}|) =: \mathcal{O}(h)$. With this approximation the Einstein-equation (2.1) simplifies to

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

where $h := \eta^{\mu\nu}h_{\mu\nu}$ and $\Box := \eta^{\mu\nu}\partial_{\mu\nu}$.

This equation has 10 independent components of which only 2 are physical. To reduce 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

$$\partial^{\alpha}\bar{h}_{\alpha\mu} = 0, \qquad (2.4)$$

where $\bar{h}_{\mu\nu} := h_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu}h$. 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

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

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

$$h_{\mu\nu}^{\text{TT}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & h_{+} & h_{\times} & 0 \\ 0 & h_{\times} & -h_{+} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \qquad (2.6)$$

Equation (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 [22].

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

$$\Box\bar{h}_{\mu\nu} = 0 \qquad (2.7)$$

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.

Equation (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

$$\bar{h}(t,\vec{x}) = \frac{4G}{c^4}\int \mathrm{d}^3x' \frac{T_{\mu\nu}\left(t - \frac{||\vec{x}-\vec{x}'||}{c}, \vec{x}'\right)}{|\vec{x} - \vec{x}'|}. \qquad (2.8)$$
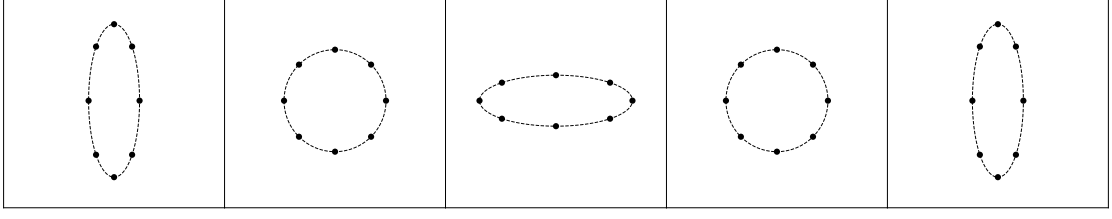
Figure 2.1: This image is taken from [21]. It shows the effect of a GW passing orthogonally through a ring of test masses.

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 $|\vec{x} - \vec{x}'| \approx r \coloneqq |\vec{x}|$. 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 evaluated 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 $Q_{ab} \coloneqq I_{ab} - \frac{1}{3}\delta_{ab}I_c^c$. This is the famous quadrupole formula [23].

To calculate the GW a binary system emits, I_{ab} or Q_{ab} needs to be specified. Furthermore, the TT-projection needs to be calculated. The projection turns out to be ((3.64) in [22])

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

Therefore, the waveforms are given by

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

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

7

The approximations that led to equation (2.3) restrict the validity of the results above to cases where there are only slight perturbations to flat space-time. Equation (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 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

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

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 [22]

---

[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 [22]). 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 [22])

which finally yield

$$h_+ = \frac{4}{r}\frac{G}{c^4}\mu R^2\omega_s^2\left(\frac{1+\cos^2(\iota)}{2}\right)\cos(2\omega_s t + 2\Phi)$$

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

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 LIGO and Virgo 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

$$F_+(\theta, \varphi) := \frac{1}{2}\left(1 + \cos^2(\theta)\right)\cos(2\varphi)$$

$$F_\times(\theta, \varphi) := \cos(\theta)\sin(2\varphi). \tag{2.22}$$

These response functions ignore the relative motion between the earth and the emitting system, as signals from binary systems spend only a couple seconds within the sensitive frequency band. [9]

The angle θ is taken between the propagation direction of the GW to the (outwards facing, relative to earth) 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 as choosing a certain polarization angle.

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 (equation (2.37) in [22])

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

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

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

---

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

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

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

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

where

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

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

According to equation (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

$$-\frac{dE_{\text{orbit}}}{dt} = -\frac{1}{2} \frac{Gm_1 m_2 \dot{R}}{R^2} \stackrel{!}{=} L_{\text{GW}}. \tag{2.29}$$

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

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

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 equation (2.26) means that the two masses will come closer and closer together until they touch. The point in time at which this happens and 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). (p.170 [22])

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

---

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

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

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

In principle, all time derivatives in (2.18) would need to be recomputed, 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 $\pi f_{\mathrm{GW}}(t)$, $2\omega_s t + 2\Phi$ by $\Phi(t)$ and R by (2.26).
With (2.31), equation (2.32) can be solved to yield ((4.30) in [22])

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

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

$$h_+(t) = \frac{1}{r}\left(\frac{GM_c}{c^2}\right)^{5/4}\left(\frac{5}{c\tau}\right)^{1/4}\left(\frac{1+\cos^2(\iota)}{2}\right)\cos(\Phi(\tau))$$

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

Inserting equation (2.34) into equation (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 (section 4.1.1 in [22]). 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).
Alongside energy, GWs also carry away angular momentum from the source. Using the quadrupole radiation (2.12), the change in angular momentum evaluates to ((3.97) in [22])

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

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 the binary system. Therefore it is at least qualitatively understandable that the spins of the two bodies has an effect on the
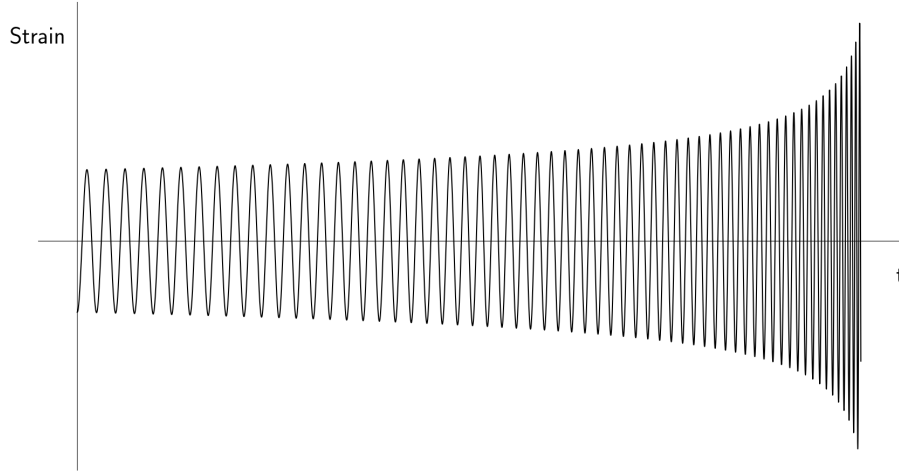
11

Figure 2.2: This figure is derived from Figure 4.1 in [22]. Shown is an example of a waveform as it could be observed by a detector if the linear waveforms describe the source accurately. Note the frequency and amplitude evolution, they both rise simultaneously. This behavior is called "chirping".

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 as they are expected to be small and will thus not go into further detail here.

2.1.2 Post-Newtonian Expansion

This part closely follows chapter 5 in [22], 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. 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 assumes equation (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 $g_{\mu\nu} \approx \eta_{\mu\nu} + h_{\mu\nu}$. 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 ((5.3) in [22])

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

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

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

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. The gauge condition most commonly used is still called deDonder gauge, which in the PN-case reads

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

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

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

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 its 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 [22]. 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

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

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

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

$$\vdots$$

14

or in short

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

The most general solution to k_1 can be written in terms of retarded multipolar waves. The solutions under consideration of the deDonder gauge can be found in (5.95) and the following equations of [22]. 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 equation (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 into further details of the formalism itself but rather look at its influence on the waveforms.

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

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

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

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

and the post-Newtonian parameter

$$\gamma := \frac{GM}{rc^2}. \tag{2.57}$$

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 [22])

$$
\begin{aligned}
E = -\frac{\mu c^2 x}{2} \Bigg\{ 1 &+ \left(-\frac{3}{4} - \frac{1}{12}\nu \right) x + \left(-\frac{27}{8} + \frac{19}{8}\nu - \frac{1}{24}\nu^2 \right) x^2 \\
&+ \left[-\frac{675}{64} + \left(\frac{34445}{576} - \frac{205}{96}\pi^2 \right)\nu - \frac{155}{96}\nu^2 - \frac{35}{5184}\nu^3 \right] x^3 \Bigg\} + \mathcal{O}\left(\frac{1}{c^8} \right)
\end{aligned} \tag{2.58}
$$

In the equation below C is the Euler-Mascheroni constant. (equation (5.257) in [22])

$$
\begin{aligned}
L_{\text{GW}} = & \frac{32c^5}{5G}\nu^2 x^5 \left\{ 1 + \left(-\frac{1247}{336} - \frac{35}{12}\nu\right)x + 4\pi x^{3/2} \right. \\
& + \left(-\frac{44711}{9072} + \frac{9271}{504}\nu + \frac{65}{18}\nu^2\right)x^2 \\
& + \left(-\frac{8191}{672} - \frac{583}{24}\nu\right)\pi x^{5/2} \\
& + \left[\frac{6643739519}{69854400} + \frac{16}{3}\pi^2 - \frac{1712}{105}C - \frac{856}{105}\log\left(16x\right)\right. \\
& + \left.\left(-\frac{134543}{7776} + \frac{41}{48}\pi^2\right)\nu - \frac{94403}{3024}\nu^2 - \frac{775}{324}\nu^3\right]x^3 \\
& + \left.\left(-\frac{16258}{504} + \frac{214745}{1728}\nu + \frac{193385}{3024}\nu^2\right)\pi x^{7/2} + \mathcal{O}\left(\frac{1}{c^8}\right)\right\}
\end{aligned}
\tag{2.59}
$$

2.1.3 TaylorF2

This section closely follows section 2.4.2 and 2.4.4 of [24].

The previous section outlines how the energy and luminosity can be obtained to some finite PN order. To actually get a waveform from these results, we need to solve

$$
\frac{\mathrm{d}E}{\mathrm{d}t} = -L_{\text{GW}} \;\rightarrow\; \frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{L_{\text{GW}}}{\mathrm{d}E/\mathrm{d}v}
\tag{2.60}
$$

and from there obtain the phase using

$$
\Phi = \int \mathrm{d}t\,\omega, \qquad\qquad M\omega = v^3.
\tag{2.61}
$$

There are many ways to approximate solutions to these equations, but we will focus only on the TaylorF2 approximation, as it is the one we use in this work. It starts by inverting equation (2.60) to get $t(v)$. It therefore tries to solve

$$
\frac{\mathrm{d}t}{\mathrm{d}v} = -\frac{\mathrm{d}E/\mathrm{d}v}{L_{\text{GW}}}.
\tag{2.62}
$$

To do so, the right hand side of this equation is re-expanded in terms of $v/c \sim \epsilon$ to 3.5 PN order. This inverted equation can then be analytically integrated and the result can be found in (2.73) of [24].

Similarly Φ can be computed from the equations (2.61), using

$$
\frac{\mathrm{d}\Phi}{\mathrm{d}v} = \frac{v^3}{M}\frac{\mathrm{d}t}{\mathrm{d}v} = -\frac{\mathrm{d}E/\mathrm{d}v}{L_{\text{GW}}}.
\tag{2.63}
$$

The integral can once more be performed analytically and the result can be found in (2.75) of [24].

The time domain phase can then be constructed from equations (2.62) and (2.63) and

is called TaylorT2. In subsection 2.2 we will find, however, that the analysis will be done in the frequency domain. One can construct the frequency representation of the orbital phase and in extension the entire waveform from the results of TaylorT2. These waveforms are hence called TaylorF2.

To do so we have to calculate

$$\tilde{h}(f) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \mathrm{d}t \; h(t) e^{2\pi i f t} = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \mathrm{d}t \; A(t) \cdot e^{-2i\Phi(t)} \cdot e^{2\pi i f t}. \tag{2.64}$$

To analytically solve this equation, we expand $\Phi(t)$ around the time t_f where the derivative of the phase is equal to the Fourier frequency, i.e.

$$2\dot{\Phi}(t_f) \overset{!}{=} 2\pi f. \tag{2.65}$$

Assuming a slowly varying amplitude $\dot{A}/A \ll 2\dot{\Phi}$ and expanding Φ to second order the integral in equation (2.64) can be solved to yield

$$\tilde{h}(f) = A(t_f)\sqrt{\frac{\pi}{\ddot{\Phi}(t_f)}}e^{i\left[2\pi f t_f - 2\Phi(t_f) - \pi/4\right]}. \tag{2.66}$$

In this derivation the spherical harmonic decomposition was disregarded and we only quoted results for the 2-2-mode. An explicit expression for the amplitude can be found in (2.84) of [24].

2.2 Searching for Gravitational Waves

The LIGO Scientific Collaboration and the VIRGO collaboration use a wide range of different online pipelines that aim to rapidly generate triggers if a GW event is present within the data [11]. Most of them are modeled searches and based on the concept of matched filtering, i.e. they need a pre-generated model to find a signal that is similar to this model. As we are using PyCBC to generate our waveforms, backgrounds and results we will be going into detail only about the PyCBC Live pipeline [25]. In the first part of this section we will briefly recapitulate the basics of matched filtering before summarizing the PyCBC Live pipeline in the second part.

2.2.1 Matched Filtering

Section 2.1 discussed how GW are generated. The detectors register them as a differential arm length of their two interferometer arms. However, the amplitude of this change in distance is on the order $\mathcal{O}(10^{-18})\,\mathrm{m}$ and as such is highly contaminated with noise originating from thermal motion of atoms, quantum shot noise and others [24]. In fact the amplitude output of the detector is dominated by noise (7.3 in [22]). For this section we will assume that the data s contains a signal h submerged in noise n. Therefore,

$$s(t) = n(t) + h(t) \tag{2.67}$$

in the time domain. To still be able to detect GWs a filtering technique names matched filtering is applied to the data. It is a modeled filter, thus requiring knowledge of the signal it is trying to find. The basic concept goes as follows. Assume we use the exact waveform that is present within the data as a filter. Then, to decide whether or not the template is contained in the data, we can multiply the template with the data and integrate this product over some time T, averaging the output by dividing by T.

$$\frac{1}{T} \int_0^T \mathrm{d}t \; s(t) \cdot h(t) = \frac{1}{T} \int_0^T \mathrm{d}t \; h^2(t) + \frac{1}{T} \int_0^T \mathrm{d}t \; n(t) \cdot h(t) \tag{2.68}$$

For $T \to \infty$ the first term will approach some finite value h_0^2, where h_0 is the characteristic amplitude of the GW. The second term, though, will approach 0 as $n(t)$ and $h(t)$ are uncorrelated (7.3 in [22]). Therefore, if we have a long signal h and integrate for the entire duration of this signal, the value equation (2.68) will give is very close to 0 if the filter h is not contained in the data and some finite value otherwise. This enables us to test wether or not some template signal is submerged within the noisy background. If we have multiple templates, the template that matches the actual signal best will most likely produce the largest value.

Equation (2.68) can further be optimized to maximize the value it returns when the perfect template is chosen. To do so, we need to introduce the power spectral density (PSD). The PSD characterizes the power contained within the noise at a given frequency bin. As $n(t)$ is real valued the fourier transform $\tilde{n}(f)$ fulfills $\tilde{n}(-f) = \tilde{n}^*(f)$, where $*$ denotes the complex conjugate. The power in each frequency bin is given by

$$\langle \tilde{n}^*(f)\tilde{n}(f') \rangle = \delta(f - f')\frac{1}{2}S_n(f), \tag{2.69}$$

where $S_n(f)$ denotes the one sided PSD. It is called one sided as the factor $1/2$ in equation (2.69) results in

$$\langle n^2(t) \rangle = \int_0^\infty \mathrm{d}f \; S_n(f). \tag{2.70}$$

Without the factor the integral would have boundaries $\pm\infty$.

The PSD can be estimated from data by integration over some period of time T and is given by

$$S_n(f) = \frac{2\langle |\tilde{n}(f)|^2 \rangle}{T} \tag{2.71}$$

and has the unit Hz^{-1}. If we refer to a PSD in this work, it will always be the one sided variant. The amplitude spectral density (ASD) is the square root of the PSD and used to color noise. Especially it is used to whiten data, i.e. divide the data by the ASD associated to the PSD of the underlying noise. Whitening is used to create data where every frequency contains the same power and thus frequencies that are very noisy are suppressed.

The matched filter can be derived using the PSD and is proven to be optimal for gaussian noise. Optimal meaning that the value the process obtains when the template and the

19

waveform within the data are the same is maximized. We use this matched filter to create a statistic called signal-to-noise ratio ρ, which is given by [25]

$$\rho^2 := \frac{||\langle s|h\rangle||^2}{\langle h|h\rangle},\tag{2.72}$$

with

$$\langle a|b\rangle = 4 \int_0^\infty \mathrm{d}f \; \frac{\tilde{a}(f)\tilde{b}^*(f)}{S_n(f)}.\tag{2.73}$$

This SNR informs us of how likely the it is that the template h is contained within the data s. The higher the value, the more likely it is that a GW was found. To use this method to detect GWs, the SNR of multiple templates has to be calculated. Then a threshold can be applied such that detection is only claimed if a minimal value of the SNR is exceeded. In a realistic search some more criteria have to be fulfilled before a detection is reported. For some more details see subsubsection 2.2.2.

2.2.2 Detection Pipeline (PyCBC Live)

This section summarizes the results of [25].
PyCBC Live is the online detection pipeline based on the PyCBC library optimized for low latency trigger generation. It utilizes a template bank of $\mathcal{O}(10^5)$ pre-calculated waveforms in a matched filter search. For multiple detectors this analysis is done individually for all detectors and only afterwards combined into a single statistic. Therefore, the goal is to calculate a SNR time series for every detector and template. To do so the expression

$$\rho^2(t) = \frac{4}{\langle h|h\rangle} \int_0^\infty \mathrm{d}f \; \frac{\tilde{s}(f)\tilde{h}^*(f)}{S_n(f)} e^{2\pi i f t}\tag{2.74}$$

has to be computed for every template h and all detector data s. In order to keep up with the constant data-stream the input is downsampled to $2048\,\mathrm{Hz}$ and high-passed. The latter allows for computations to be executed in single precision. Furthermore, the workload can be easily parallelized over multiple compute notes to speed up the process. This process is repeated every $8\,\mathrm{s}$ on a chunk of data that contains between $32\,\mathrm{s}$ and $256\,\mathrm{s}$ of time series strain data. The PSD $S_n(f)$ is estimated from the data by dividing the minute prior to the analysis segment into overlapping intervals, each of duration $4\,\mathrm{s}$, and computing equation (2.71) from it. The frequency bins are finally averaged over all intervals.
After computing the SNR time series, the peak SNR of each one is thresholded. For the observation run 2 this value was chosen to be ~ 5.5. No peak SNRs below the threshold value are kept and all but the loudest $\mathcal{O}(10^3)$ are discarded. Sometimes these triggers are then discarded altogether. This happens if the PSD estimate shows drifts in detector noise, sensitivity estimates report poor data quality or the SNR values are unusually high. If non of these conditions apply, some more consistency tests are applied and the SNR is reweighed based on a χ^2 signal consistency test. Until now this procedure was done for all detectors individually. The resulting single detector triggers are then

combined to report a detector network SNR. To do so, the templates that generated a trigger need to have generated a trigger in all other detectors. The time difference at which the peak was found in each detector is furthermore not allowed to be greater than the time of flight difference, which is $\sim 10\,$ms. The maximum separation is therefore set to $15\,$ms.

For the two detectors Hanford and Livingston all remaining trigger pairs are combined to a network SNR given by

$$\rho_c^2 = \rho_{\rm H}^2 + \rho_{\rm L}^2 + 2\log\left(p(\theta)\right), \tag{2.75}$$

where $p(\theta)$ is the astrophysical probability of a trigger observed with parameters θ.

To estimate the significance of the trigger, the false alarm rate of the procedure producing the reported trigger is calculated. In order to get enough data that does not contain a physical signal but captures the current detector performance, the data of one detector is shifted in time. The length of time to shift is determined by the time of flight between the two observatories. If a trigger was found in such time shifted data, we would know that it could not be a real trigger as it violates causality. To do this efficiently only the triggers for each detector are shifted in time. This procedure allows to get false alarm rate estimates of down to 1 per 100 years. The threshold of notifying other astronomers of a trigger is chosen to be 1 per 2 month.

The described procedure produces reliable triggers. It is able to keep up with the detector output but introduces on average $16\,$s of latency, i.e. a trigger is reported $16\,$s after the event occurred on average. An estimate of the sky position is provided by a follow up called BAYESTAR [26].

3 Neural Networks

Neural networks are machine learning algorithms inspired by research on the structure and inner workings of brains. [27] Though in the beginning NNs were not used in computer sciences due to early limitations [28] 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. Major use cases in the realm of gravitational wave analysis have been classification of glitches in the strain data of GW-detectors [29] and classification of strain data containing a GW versus pure noise [15, 30].
In this section the basic principles of NNs will be introduced and notation will be set. The concept of backpropagation will be introduced. 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 [28, 31].

3.1 Neurons, Layers and Networks

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

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

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

23

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 given in equation (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.
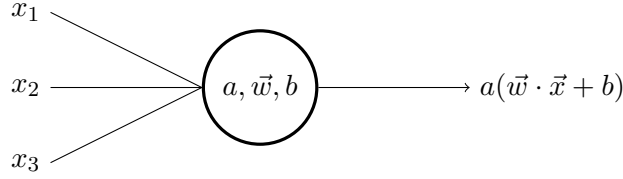

Figure 3.1: Depiction of a neuron with inputs $\vec{x} = (x_1, x_2, x_3)^T$, weights $\vec{w}$, bias b and activation function a.

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.

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 binary full-adder (see Appendix A). These structures are then called neural networks, as they are a network of neurons. The example of the full-adder demonstrates the principle of a 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
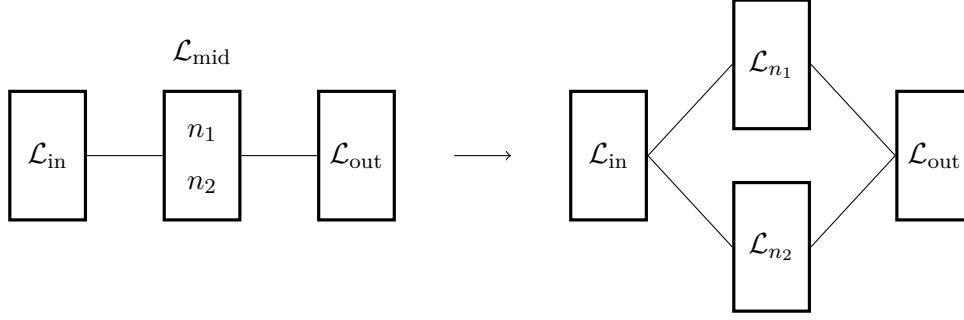
Figure 3.2: Depiction of how a layer ($\mathcal{L}_{\text{mid}}$) consisting of neurons with different activation functions (n_1 and n_2) can be split into two separate layers ($\mathcal{L}_{n_1}$ and $\mathcal{L}_{n_2}$).

(section 6.1 in [28]), which caused the loss of interest in NNs around 1940 (section 6.6 in [28]).

One of the most popular activation functions is the rectified linear unit (ReLU). It is given by

$$a(y) = \begin{cases} y, & y > 0 \\ 0, & y \leq 0 \end{cases}.$$
(3.4)

It has been empirically shown that this simple nonlinear activation improves the performance of NNs [32].

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

$$\mathcal{L} : \mathbb{R}^{k \times l} \times \mathbb{R}^l \times \mathbb{R}^k \to \mathbb{R}^l; \ (W, \vec{b}, \vec{x}) \mapsto \mathcal{L}(W, \vec{b}, \vec{x}) := \begin{pmatrix} n_1\left((W_1)^T, b_1, \vec{x}\right) \\ \vdots \\ n_l\left((W_l)^T, b_l, \vec{x}\right) \end{pmatrix},$$
(3.5)

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

$$\mathcal{L}(W, \vec{b}, \vec{x}) = a(W \cdot \vec{x} + \vec{b}),$$
(3.6)

25

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 as

$$
\begin{aligned}
\mathcal{N}&\left(W^{\text{in}}, \vec{b}^{\text{in}}, W^{\text{mid}}, \vec{b}^{\text{mid}}, W^{\text{out}}, \vec{b}^{\text{out}}, \vec{x}\right) \\
&:= \mathcal{L}_{\text{out}}\left(W^{\text{out}}, \vec{b}^{\text{out}}, \mathcal{L}_{\text{mid}}\left(W^{\text{mid}}, \vec{b}^{\text{mid}}, \mathcal{L}_{\text{in}}\left(W^{\text{in}}, \vec{b}^{\text{in}}, \vec{x}\right)\right)\right) \\
&= a_{\text{out}}\left(\vec{b}^{\text{out}} + W^{\text{out}} \cdot a^{\text{mid}}\left(\vec{b}^{\text{mid}} + W_{\text{mid}} \cdot a_{\text{in}}\left(\vec{b}^{\text{in}} + W_{\text{in}} \cdot \vec{x}\right)\right)\right). \quad (3.7)
\end{aligned}
$$

Hence a network can be understood as a set of nested functions.

An important point with the definitions above is that the layers get their input only from their preceding layers. Especially no loops are allowed, i.e. getting input from some subsequent layer is not permitted. A network of the first kind is called a *feed forward neural network* (FFN), as 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 worse performance and are therefore not used or discussed further in this work.

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 output. 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 [28]). Figure 3.3 shows an example of a simple network with a single hidden layer. In principle there could be any number of hidden layers with different sizes. In this example the input is n-dimensional and the output 2-dimensional. If the input was changed to be (n-1)-dimensional, the same hidden-layer could be used, as its size does not depend on the data or output. Therefore, when designing a network architecture, one designs the shape and functionality of the hidden layers. 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.

A NN is called *deep*, if it has multiple hidden layers. The depth of a network, analogously, is the number of layers used in the longest path from some input to an output layer.

3.2 Backpropagation

In subsection 3.1 the basics of a NN where discussed and the example of a network replicating a binary full-adder showed the potential of these networks, if the weights and biases are chosen correctly. The example actually proofs that a sufficiently complicated
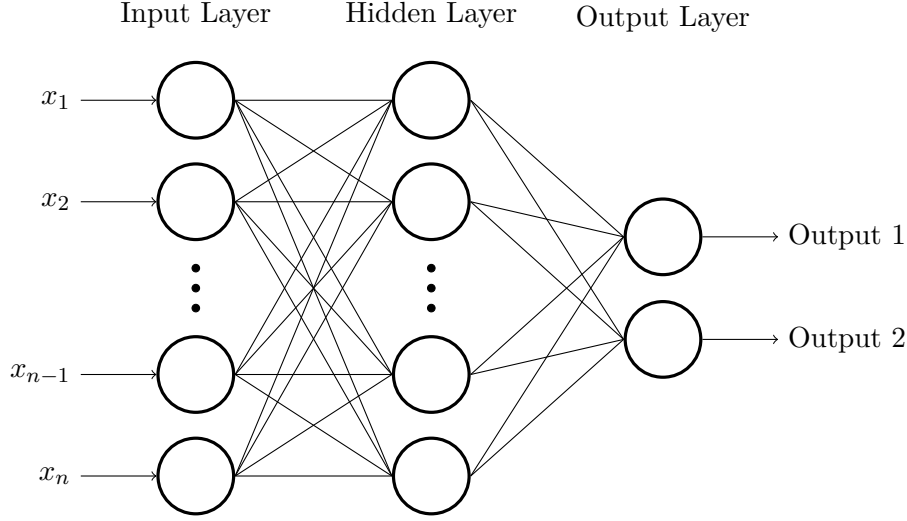
Figure 3.3: A depiction of a simple network with a single input-, hidden- and output-layer. The input-data is a n-dimensional vector $(x_1, \ldots, x_n)^T$ and the output is a 2-dimensional vector. In this picture it looks like the hidden layer has the same number of neurons as the input layer. This does not necessarily have to be the case. Lines between two neurons indicate, that the output of the left neuron serves as input for the right one.

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][31]

Therefore, the question is how to choose the weights in a network for it to approximate some function optimally. For the binary full-adder the weights and biases were chosen by hand, as the problem the network was trying to solve was rather simple. A more general approach, however, would be beneficial, as not all problems are this simple. Therefore, the goal is to design a network which learns/optimizes 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[6], in order for the network to be able to compare its output to some ground truth and adjust its weights and biases to minimize an 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 a scalar number; the SNR [7]. Therefore the network receives some

---

[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 [28])

[6]This statement is strictly only true in the context of supervised learning, which we will be focusing on. There is, however, also a field called unsupervised learning, where the machine learning algorithm picks up on general concepts within the data itself and clusters it accordingly.

[7]In actuality we will also train for a second value called the p-score. For simplicity we will however stick with just the SNR for this section.

data as input, of which the true SNR-value is known. This true value will be called *label*[8] from here on out. The network will produce a 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 efficiently 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. Backpropagation alone is not an optimization algorithm, it is used to calculate gradients and as such is the basis on which the optimization algorithm works.

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

$$L : \mathbb{R}^{l \times k} \times \mathbb{R}^{l \times k} \rightarrow \mathbb{R}; \ (y_{\text{net}}, y_{\text{label}}) \mapsto L(y_{\text{net}}, y_{\text{label}}), \tag{3.8}$$

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

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

$$L : \mathbb{R}^{l \times k} \times \mathbb{R}^{l \times k} \rightarrow \mathbb{R}; \ (y_{\text{net}}, y_{\text{label}}) \mapsto L(y_{\text{net}}, y_{\text{label}}) := \frac{1}{l} \sum_{i=1}^{l} (\vec{y}_{\text{net},i} - \vec{y}_{\text{label},i})^2. \tag{3.9}$$

A more thorough discussion and justification for using MSE as loss can be found in section 5.5 and 6.2.1.1 of [28].

To minimize this loss, the weights and biases of the different layers are changed, usually using an algorithm called *gradient descent*. It works by calculating the gradient of one layer with respect to its weights and biases and changing their values by following the negative gradient. For notational simplicity we will 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 descent is then given by

$$\theta' = \theta - \epsilon \ \nabla_\theta L(y_{\text{net}}(\theta), y_{\text{label}}), \tag{3.10}$$

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 gradient descent is then made using only the samples from one mini-batch. This alteration of gradient descent goes by the name of *stochastic gradient descent* (SGD). The larger the mini-batch, the more accurate the estimate of the gradient becomes 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

---

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

the mini-batch size.

The real work of training a network 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 backpropagation and is based on an iterative application of the chain rule.

For simplicity we assume 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 (3.9). Calculating the gradient $\nabla_\theta L(y_{\text{net}}(\theta), y_{\text{label}})$ requires to calculate $\nabla_{W^1} L, \cdots, \nabla_{W^n} L$ and $\nabla_{\vec{b}^1} L, \cdots, \nabla_{\vec{b}^n} L$, where

$$\nabla_{W^i} L := \begin{pmatrix} \partial_{W_{11}^i} L & \cdots & \partial_{W_{1l}^i} L \\ \vdots & \ddots & \vdots \\ \partial_{W_{k1}^i} L & \cdots & \partial_{W_{kl}^i} L \end{pmatrix}, \tag{3.11}$$

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

$$\nabla_{\vec{b}^i} L := \begin{pmatrix} \partial_{b_1^i} L \\ \vdots \\ \partial_{b_k^i} L \end{pmatrix}, \tag{3.12}$$

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

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

$$z^n := \vec{b}^n + W^n \cdot a_{n-1}\left(z^{n-1}\right)$$
$$z^1 := \vec{b}^1 + W^1 \cdot \vec{x}, \tag{3.13}$$

such that

$$\mathcal{N}(\theta, \vec{x}) = a_n(z_n). \tag{3.14}$$

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

$$L(y_{\text{net}}, y_{\text{label}}) = L(\mathcal{N}(\theta, \vec{x}), y_{\text{label}}) = L(a_n(z_n), y_{\text{label}}) = (a_n(z_n) - \vec{y}_{\text{label}})^2. \tag{3.15}$$

To start off we derive this loss by the parameter θ_j

$$\partial_{\theta_j}(a_n(z_n) - \vec{y}_{\text{label}})^2 = \left(\partial_{\theta_j} a_n(z_n)\right)(2(a_n(z_n) - \vec{y}_{\text{label}})) \tag{3.16}$$

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

$$\partial_{\theta_j} a_n(z_n) = \partial_{\theta_j} \sum_i a_n^i(z_{n,1}(\theta_j), \ldots, z_{n,k}(\theta_j)) \vec{e}_i$$

$$= \sum_i \sum_{m=1}^{k} \left(\partial_{\theta_j} z_{n,m}(\theta_j) \right) \left(\partial_{z_{n,m}} a_n^i(z_{n,1}(\theta_j), \ldots, z_{n,k}(\theta_j)) \right) \vec{e}_i$$

$$= \sum_i \left(\left(\partial_{\theta_j} z_n \right) \cdot \left(\nabla_{z_n} a_n^i \right) \right) \vec{e}_i \tag{3.17}$$

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

$$\partial_{\theta_j} a_n(z_n) = \left(\partial_{\theta_j} z_n \right) \odot \partial_z a_n(z)\big|_{z=z_n}, \tag{3.18}$$

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.

$$\partial_{\theta_j} z_n = \partial_{\theta_j} \left(\vec{b}^n + W^n \cdot a_{n-1}(z_{n-1}) \right)$$

$$= \partial_{\theta_j} (W^n \cdot a_{n-1}(z_{n-1}))$$

$$= W^n \cdot \partial_{\theta_j} a_{n-1}(z_{n-1}) \tag{3.19}$$

Inserting (3.19) into (3.18) yields the recursive relation

$$\partial_{\theta_j} a_n(z_n) = \left(W^n \cdot \partial_{\theta_j} a_{n-1}(z_{n-1}) \right) \odot \partial_z a_n(z)\big|_{z=z_n}. \tag{3.20}$$

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

$$\partial_{\theta_j} a_k(z_k) = \left(\partial_{\theta_j} W^k \right) \cdot a_{k-1}(z_{k-1}). \tag{3.21}$$

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 we define

$$[\Sigma(\vec{x})]_{ij} = \begin{cases} x_i, & i = j \\ 0, & \text{otherwise} \end{cases}. \tag{3.22}$$

With this definition, equation (3.20) can be written as

$$\partial_{\theta_j} a_n(z_n) = \Sigma\left(\partial_z a_n(z)\big|_{z=z_n} \right) \cdot W^n \cdot \partial_{\theta_j} a_{n-1}(z_{n-1}) \tag{3.23}$$

and the recursion can be solved to yield

$$\partial_{\theta_j} a_n(z_n) = \left[\prod_{l=0}^{n-k+1} \Sigma\left(\partial_z a_{n-l}(z)\big|_{z=z_{n-l}} \right) \cdot W^{n-l} \right] \cdot \Sigma\left(\partial_z a_k(z)\big|_{z=z_k} \right) \cdot \left(\partial_{\theta_j} W^k \right) a_{k-1}(z_{k-1}). \tag{3.24}$$

The same computation can be done if θ_j is a bias instead of a weight. When this computation is done, equation (3.20) still holds, but the stopping condition (3.21) is simplified to

$$\partial_{\theta_j} a_k(z_k) = \partial_{\theta_j} \vec{b}^k. \tag{3.25}$$

From this the analytic form can be computed to be

$$\partial_{\theta_j} a_n(z_n) = \left[\prod_{l=0}^{n-k+1} \Sigma\left(\partial_z a_{n-l}(z)\big|_{z=z_{n-l}}\right) \cdot W^{n-l} \right] \cdot \Sigma\left(\partial_z a_k(z)\big|_{z=z_k}\right) \cdot \partial_{\theta_j} \vec{b}^k. \tag{3.26}$$

The recursive formula (3.20) 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 equation (3.20). It starts at the last layer and propagates back through the network until it reaches the layer of the weight that should be adjusted. The parameters are then changed by SGD as given in equation (3.10).

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 asked for the derivative by A and $G \cdot A^T$ when aked for the derivative by B. (section 6.5.6 [28])

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 [28].)

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. The full training set is then used multiple times to train the network. Each entire pass of this set is called one *epoch*. In theory, this should lead to the network continuously improving. The only thing that could stop this improvement would be a vanishing gradient. This could be due to either finding the global minimum of the loss with respect to the network parameters, which would be great, or due to the optimizer finding a local minimum, which is in general not ideal.

In practice training the network for many epochs on the same data set can lead to problems. 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.* It 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. In the case of $n \geq 3$ 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. (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 which were 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 it. If the MSE rises on samples that were not used for the regression, overfitting takes place.

The same concept can then be applied to NNs; if the loss of a network is bigger on different data than that used during training, the network is said to overfit. This second set of samples is called the validation set, as it validates the training. Usually the data in the validation set stems from the same underlying procedure that generated the training set. In the context of this work this means 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 and overall bad performance. 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[9]. (See section 7.4 of [28])

To tune out the generalization error, which is the loss value of the validation set, one adjusts the architecture. If there are multiple different architectures, the best one is chosen by the performance on the validation set. In this way, the validation set is also used to fit the model, as in the end the human who trains the models selects the best performing network. Therefore all given results, that did not occur during training[10], come from a third independent set. This set is then called the test set.

One 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 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 [28]) 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 [33].

A common problem that arises when training very deep NNs is the vanishing gradient problem; the gradient calculated by equations (3.24) or (3.26) 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 equations (3.24) and (3.26). If the factors satisfy $|\partial_z a_i(z) \cdot W^i| < 1$ the gradient is exponentially diminished by the depth of the network.

The opposite can also happen. If $|\partial_z a_i(z) \cdot W^i| > 1$ for most of the layers, the gradient will grow exponentially. This behavior is therefore called the exploding gradient problem. (Chapter 5 [31])

To overcome these problems, one can simply train for longer periods in the case of the vanishing gradient problem (Chapter 5 [31]), use multiple points at which the loss is calculated [34] or adjust the initialization of the weights. Another solution introduced by [35] 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. Residual connections showed improvements in accuracy and enabled the use of even deeper networks [35].

---

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

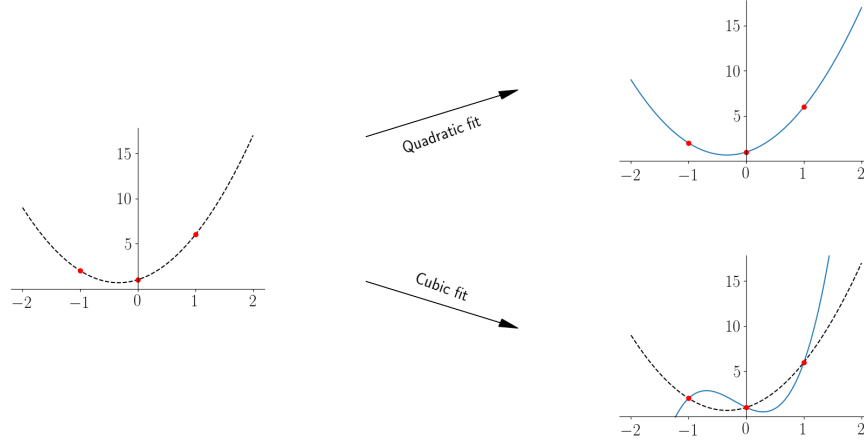[10]An example of a result that comes from training the network would be the loss history.

Figure 3.4: Depiction of overfitting in the classical regression. On the left the originally sampled function $f(x) := 3x^2 + 2x + 1$ 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.

3.4 Convolutional 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 left side of 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 convolutional layer.

3.4.1 Convolutional Layer

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 linearly with the number of neurons on each layer. This causes computational limits if the number of neurons per layer should stay at a reasonable number, thus limiting 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.[11] These restrictions led to the invention of the convolutional layer in 1989. [36] Though it was originally conceived in its 2 dimensional variant, only the 1 dimensional convolutional layer will be explained here.[12] Contrary to dense layers, the neurons of convolutional 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 convolutional layer as a filter of learnable weights that slides across the input, in a way convolving the the filter with the input data. (see right side of Figure 3.5)

The size of the filter, i.e. how many entries it spans, is called the *kernel size*. If multiple convolutional 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 convolutional 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 convolutional 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 convolutional layers, they take in a 2 dimensional input and are specified by the number of filters and the kernel size of all these filters[13]. 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 3.6)

As an example say we specify a convolutional layer by having 32 filters and a kernel size of 3. Now we use this convolutional 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 convolutional layer would be 4094×32. Therefore the convolutional 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

---

[11]This is true only to some degree, as a sufficiently large dense layer could emulate a convolutional layer by setting a lot of weights to 0. Compare equation (3.29).

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

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

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 convolutional 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 convolutional 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 convolutional 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 convolutional 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 convolutional network.

Having sparse connections in the convolutional layers also leads to stacked convolutional layers having a *receptive field*. The receptive field of one output of a convolutional layer is the number neurons on the input layer that have, through some path, an influence on its value. (see Figure 3.7)

Though convolutional 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 $\vec{w} = (w_1, \ldots, w_n)^T$. 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

$$[\vec{x} * F]_j = a\left(\left(\sum_{k=0}^{n-1} w_{k+1} \cdot x_{j+k}\right) + b\right), \tag{3.27}$$

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

$$\sum_{j=1}^{m} [\vec{x} * F]_j \cdot \vec{e}_j = a\left(W \cdot \vec{x} + \vec{b}\right), \tag{3.28}$$

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

$$W = \begin{pmatrix} w_1 & \ldots & w_n & & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & & w_1 & \ldots & w_n \end{pmatrix}. \tag{3.29}$$

The backpropagation algorithm then only needs to know about the gradient of W with respect to the weights $\vec{w}$.
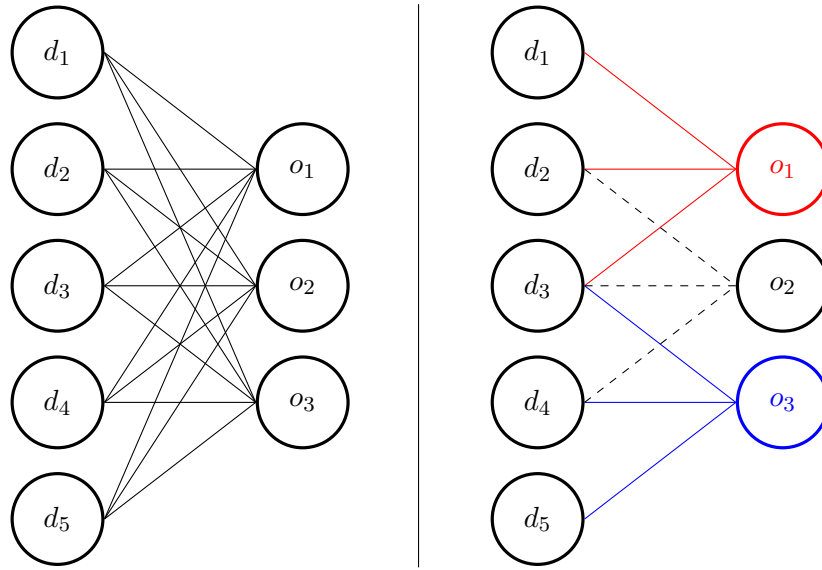
Figure 3.5: Left: Two layers with dense connections. In this case the right layer is the dense layer. Each of the 3 outputs o_1 - o_3 is connected to every input d_1 - d_5. Therefore, the dense layer has 15 trainable weights. For a dense layer the number of output neurons is not fixed but can be freely chosen. We could in principle have used any number of neurons on the right layer. **Right**: An example of a convolutional layer with kernel size 3. The two layers have the same number of neurons as the dense example on the left, but only 3 trainable parameters, as the weights in each of the three colored connections are shared. Contrary to the dense layer the number of output neurons for the convolutional layer is fixed by the kernel size. To get more or less output neurons we would need to adjust the number of neurons we combine to form an output.
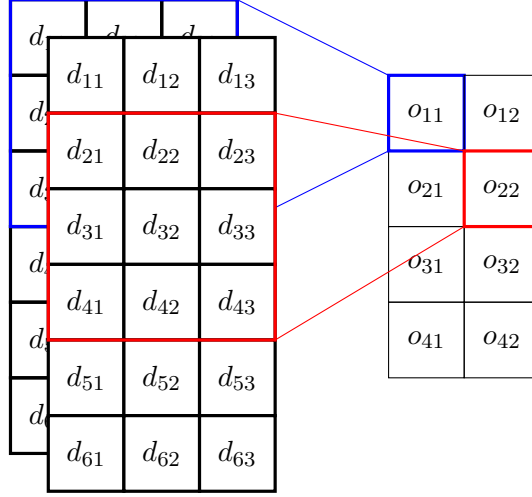
Figure 3.6: Depiction of a convolutional layer with two filters that have a kernel size of 3. The input data d_{11} to d_{63} has three channels and is the same for both the blue and the red filter. The first channel of the output (o_{11} to o_{41}) is produced by sliding the blue filter over the data, 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.
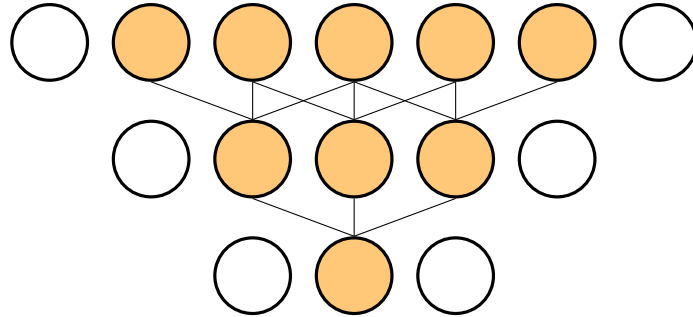

Figure 3.7: Three stacked convolutional layers. Although all layers have a kernel size of 3, the final layer is influenced by 5 of the input values. Therefore the receptive field of the final layer is 5.

3.4.2 Pooling Layers

Pooling layers are another special kind of layers, often used to increase performance of convolutional neutral networks (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 [37]. 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 3.8)
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[14]. 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 convolutional filter. If this resonance only applies for a small region in the data, there will only be few values on the feature map that correspond to this resonance. Therefore, pooling (in general) leads to greater spatial invariance. The downside of pooling is the loss of positional information. As a rule of thumb, pooling is useful for a decision "is a feature present", but falls short if the question "where in the data is the feature present" is also relevant. Due to its improvement in spatial invariance, pooling also leads to the next layer having a greater receptive field.

All the actions described above act only on a single feature map, channel by channel. A similar approach can however also be taken for the channels themselves. Such a procedure is called *dimensional reduction* and usually done through a convolutional layer with a kernel size of 1. This way all channels are being connected through a weighted sum, where the weights are learned [39]. Additionally an 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. [40] 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 convolutional layer with kernel size 1.

3.4.3 Inception Module

Networks consisting of stacked convolutional layers as introduced in subsubsection 3.4.1 have had great success in image classification. [28, 41, 12] 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

---

[14]There have been studies suggesting that pooling works better than simply sub sampling the data [38].
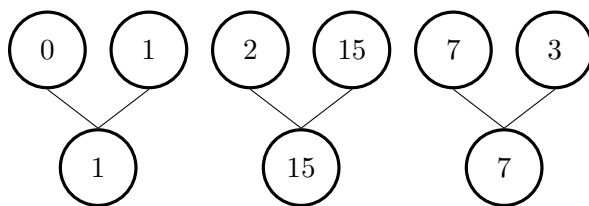
Figure 3.8: An example of a Max pooling layer. It groups together two entries of its input and returns the maximum, thus halving the number of samples per feature map. This process is applied for all channels individually.

in [34].

The advantages of convolutional 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 convolutional layers throughout the network. If the number of filters of two consecutive convolutional 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, if trained optimally [34]. Another way to increase the capactity is to scale the convolution-kernel size. 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. [34]

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 try to take. They build a single module, that can be viewed as a layer from the outside. It contains multiple small convolutional layers, that build up a larger, sparse filter. Using this new architecture, the GoogLeNet won the 2014 ImageNet Large Scale Visual Recognition Competition (ILSVRC)[15] 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. [34, 12]

---

[15]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.

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. [42, 43]

The following paragraphs will describe the module used in this work in greater detail. The module consists of 4 independent parallel lanes, each consisting of different layers. The full module is depicted in Figure 3.9.

The module consists of three parallel convolutional layers, i.e. each of the three layers share the same input. The difference between them is the kernel size. The convolutional layers with a larger kernel are preceded by a convolutional layer with 16 filters and a kernel size of 1. The purpose of this step is to reduce the number of channels used and is called dimensional reduction. This leads to a fixed input size for the larger kernels, regardless of the depth of the input. In the original architecture filters of size 1×1, 3×3 and 5×5 were used. Translating them directly to 1 dimensional equivalents, the module should use kernel sizes of 1, 3 and 5. However, we empirically found that the smallest kernel sizes 1, 2 and 3 performed best.

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

3.4.4 Temporal Convolutional Networks

Temporal convolutional networks (TCN), as used in this work, were proposed by [44]. 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 convolutional layers, that are slightly adapted in two different ways. For once, the filters are dilated, meaning, that the weighted sum of the convolution is not taken over successive input points. Instead the weighted sum uses points, skipping a set number of inputs in between. Secondly the connections are causal. This means that output y_i of the filter depends at most only on points $x_i, \ldots, x_1$. 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 3.10)

The advantage of the dilated convolutional layers is that the receptive field of the network grows exponentially with the depth if the dilation is scaled exponentially, whereas
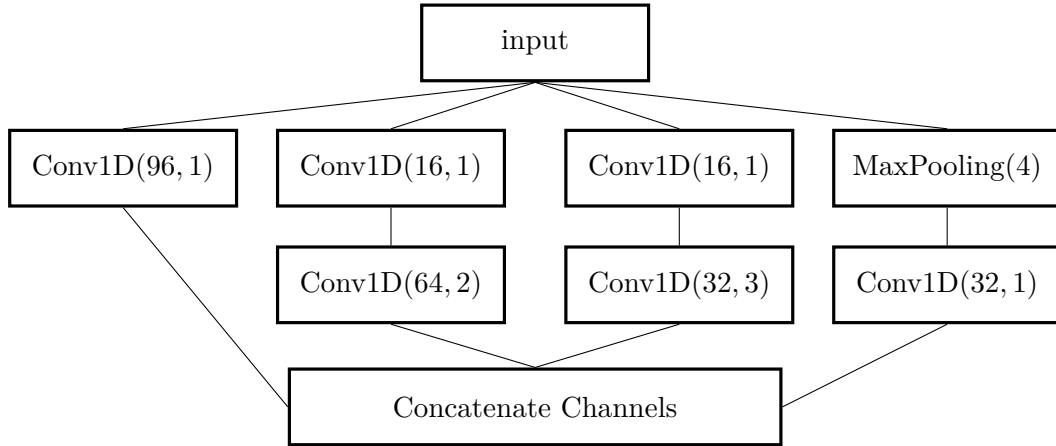
Figure 3.9: Shown are the contents and connections of the inception module as used in this work. The layer Conv1D(x, y) is a 1 dimensional convolutional layer with x filters and a kernel size of y. Most of the convolutional layers with a kernel size of 1 are used for dimensional reduction. The only exception 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.

without the dilation this growth is only linear. 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 3.10. 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. [45]

3.5 Regularization

As [28] 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 to achieve this goal, like reducing the number of trainable parameters, adjusting the loss function to prefer specific weights, using batch normalization or using dropout. This section will only cover the last two methods.
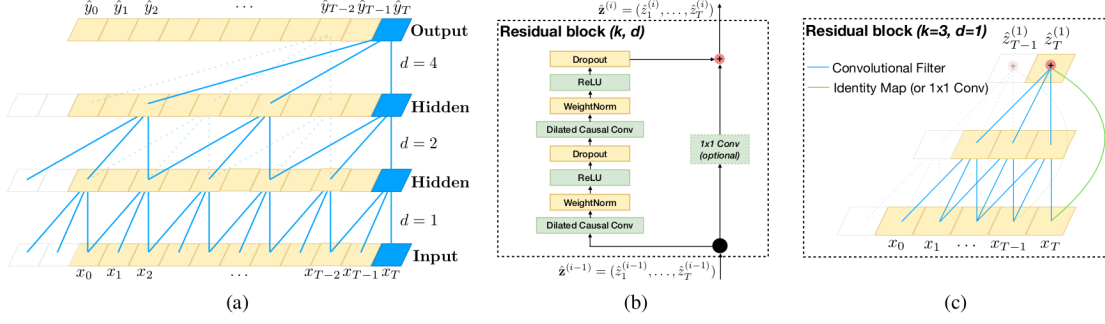
Figure 3.10: Architectural elements in a TCN. Graphic taken from [44]. (a) Exponential increase of the receptive field in dilated convolutional layers. Here the dilation factor d scales as 2^i and the kernel size k is set to 3. The causal structure propagates through the layers, as no output is connected to a later 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.

3.5.1 Batch Normalization Layer

Batch normalization was introduced in [46] and is used to normalize the inputs of each layer. This helps the network learn faster and generalize more easily [46].

The normalization tries to fix the distribution of the inputs between different samples, as the layers would need to adapt their parameters 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. The problem of normalizing the input of each individual layer is the backpropagation step, as it can lead to exploding biases. [46] 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

$$y_i = \gamma \hat{x}_i + \beta \tag{3.30}$$

is applied to the normalized data

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B{}^2 + \varepsilon}}. \tag{3.31}$$

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 network 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

$$
\begin{aligned}
\mu'_{\mathrm{mov}} &= m \cdot \mu_{\mathrm{mov}} + (1-m)\mu_B \\
\sigma'_{\mathrm{mov}}{}^2 &= m \cdot \sigma_{\mathrm{mov}}{}^2 + (1-m)\sigma_B{}^2,
\end{aligned} \tag{3.32}
$$

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

Dropout layers were introduced in 2014 by [47] and showed great improvements to lowering the generalization error. They work 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 that the dropout layer acts 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 [28] and [47], as it allows to efficiently sample many networks and combine them.

Although dropout layers have many advantages, they come at the cost of less stable learning. This is especially evident in the loss that is a lot less consistent. Furthermore, if the dropout rate is chosen too high, the network will stop learning altogether as it can not adapt to too many dropped connections.

During the evaluation process, the original paper [47] 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. Therefore, the dropout layer has no effect when a network that utilizes this form of regularization is used.

4 Searching for Gravitational Waves using Neural Networks

Detecting GWs from noisy detector data is a difficult problem as the potential signals are very faint. Current matched filtering based pipelines, as outlined in subsection 2.2, are very sensitive but also have two major drawbacks. First of all, they are only proven to be optimal for Gaussian noise. The detector data on the other hand contains non-Gaussian noise transients and thus is not entirely Gaussian or stationary [48]. For this reason further measures than just using matched filtering have to be taken. Secondly, even though current detection pipelines are able to keep up with the constant data-stream from the detectors, they introduce latency on the order $\mathcal{O}(10\,\mathrm{s})$. This means that astronomers are alerted of an event about $10\,\mathrm{s}$ after the GW has passed the detector [25]. If the signal came from a binary system containing at least one neutron star an EM counterpart might be detectable. This counterpart contains a lot of information about the neutron star itself and the evolution of the system [10]. Maximizing the observation time is thus very important. Furthermore, the computational cost for matched filter searches grows as detectors get more sensitive since more templates will be needed to cover the entire parameter space. This will either lead to even higher latency or necessitate more powerful hardware.

To reduce the latency and computational cost more efficient searches are required. Very promising contenders are neural networks which are known for their computational efficiency once trained. They are furthermore capable of generalizing well to unseen data. As such they might be able to exceed sensitivities of matched filtering based approaches on non-Gaussian noise.

For these reasons using NNs to analyze data for GWs has grown in popularity and multiple very promising results have been found. This section therefore gives an overview of the state of the art machine learning techniques that aim to classify GW strain data and puts the thesis at hand into context of these works.

The use of deep neural networks as a filter to detect GW-signals in noisy detector data was pioneered by Daniel George and E.A. Huerta [15]. They used a CNN to classify time series data into the two categories "noise + signal" and "pure noise" as well as estimate some source parameters for BBH signals. The network was able to reproduce the sensitivity a classical matched filter search could achieve and even showed potential to adapt to eccentric signals which were not used during the training stage. They did, however, calculate their sensitivity at a false alarm probability of 0.6%, which equates to a false alarm rate of $155\,000\,\frac{\text{samples}}{\text{month}}$. The followup paper [49] used real detector noise, demonstrating that CNNs can be used for real time classification and parameter estimation. The algorithm was furthermore accurate even in the presence of non-Gaussian transients.

A similar concept has been applied to the search for continuous GWs in [50]. 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

45

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 do not get more specific in that regard.

The recently published paper [30] tries to classify time series strain data into the three categories "pure noise", "BBH signal" and "BNS signal". As such it is the first publication that tries to classify BNS signals using a deep NN. In that regard it is very similar to our work. This enables us to compare our results to their findings.

Both networks are able to consistently recover BNS signals with a low false positive probability. We do, however, suggest to use not the false positive probability but rather a false alarm rate. Doing so produces results that are more easily comparable to realistic searches. We therefore estimate the false alarm rate at which the authors of [30] generated their results. To do so we use the fact that the GWs are shifted by $\pm 0.1\,$s within the data. Therefore, the network would need to be slid across the data using a step size of $0.1\,$s. Their testing set contained only 5000 samples, of which ~ 1667 were pure noise samples. With so few samples the false alarm rate can only be estimated very roughly and down to $\sim 15\,500\,\frac{\text{samples}}{\text{month}}$ (for details on how to do so see subsubsection 5.2.2).

Furthermore, the results in [30] are solely given in terms of peak signal-to-noise ratio (pSNR). The authors quote a conversion factor of ~ 13 between the SNR matched filtering would return and pSNR. The source they used calculated this factor only for BBH signals. We found that for BNS signals this factor is roughly[16] 39.225, i.e. $\text{SNR} \approx 39.225\,\text{pSNR}$.

Finally, to compare our results to the findings of [30] we note that our algorithm uses the data of 2 detectors. Therefore, signals that are injected at SNR x in [30] would on average be assigned a label value of SNR $\sim \sqrt{2}x$ for our analysis.

A detailed comparison can be found in subsubsection 5.4.2.

[51] 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 statistically 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 positioned during the training stage but at some uncontrollable point. Due to this limitation they suggest 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 and statistics empirically from a long continuous

---

[16]We report these values based on the following calculation: We define the peak SNR, following [30], as $\max{(\text{waveform})}/\sqrt{\text{Var(noise)}}$. 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 [30] 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.

time series. Further measures and precautions will be explained in subsubsection 5.4.2. A major contribution relating our final architecture came from [52], the authors of which 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 for this task. 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.

[53] takes a similar approach to their network architecture as [52] 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.

5 Network Topologies

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 to give an estimate of the signals SNR.

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 their strain amplitude is a lot lower and the signal lasts for a longer duration[17]. 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 $30\,\mathrm{Hz}$ to $100\,\mathrm{Hz}$. To resolve these frequencies, a sample rate of $60\,\mathrm{Hz}$ to $200\,\mathrm{Hz}$ 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 $64\,\mathrm{s}$ of data, where a signal is aligned such that the high frequency cutoff is roughly $0.5\,\mathrm{s}$ from the end. Afterwards we chop the data into parts of duration $2^i\,\mathrm{s}$ and re-sample each of these parts to have a sample rate of $2^{12-i}\,\mathrm{Hz}$, with $i = 0, \ldots, 6$. 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 $2\,\mathrm{s}$ interval sampled at $2048\,\mathrm{Hz}$ also includes the $1\,\mathrm{s}$ interval sampled at $4096\,\mathrm{Hz}$. 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 $64\,\mathrm{s}$ 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

---

[17]BBH signals spend about $1\,\mathrm{s}$ within the sensitive frequency range of our detectors, whereas BNS signals can be visible in the whitened detector data for multiple tens of seconds [9] and when generated even last multiple hundred seconds.
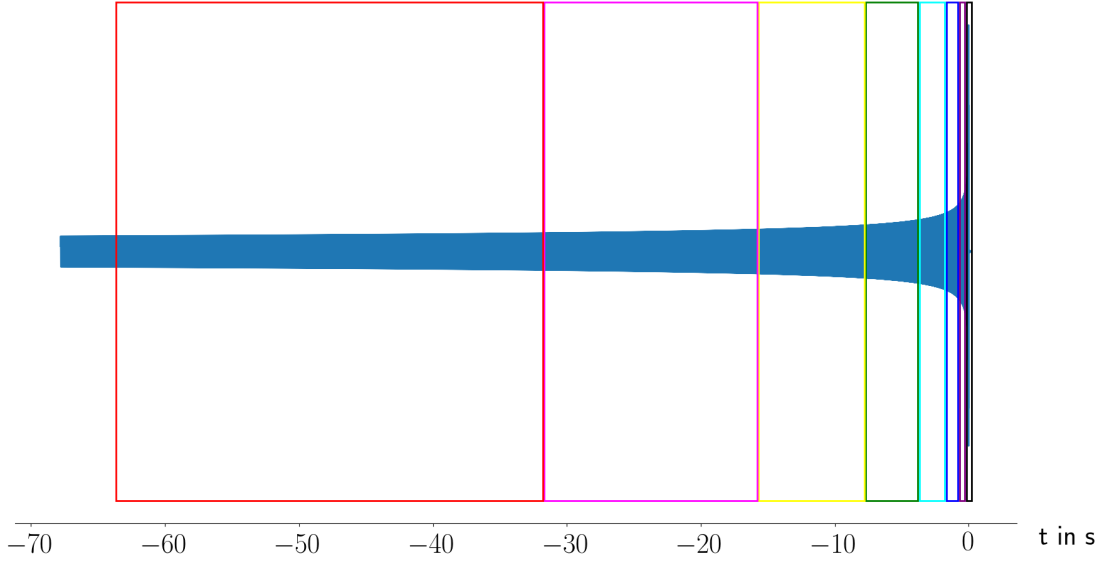
Figure 5.1: The plot shows how each BNS signal is sampled at multiple rates. Only the last $68\,\mathrm{s}$ of the entire waveform are shown. Each sample rate and interval has its own color attributed. Specifically they are given by: black=$(0.5\,\mathrm{s},\ 4096\,\mathrm{Hz})$, purple=$(0.5\,\mathrm{s},\ 4096\,\mathrm{Hz})$, blue=$(1\,\mathrm{s},\ 2048\,\mathrm{Hz})$, cyan=$(2\,\mathrm{s},\ 4096\,\mathrm{Hz})$, green=$(4\,\mathrm{s},\ 1024\,\mathrm{Hz})$, yellow=$(8\,\mathrm{s},\ 512\,\mathrm{Hz})$, magenta=$(16\,\mathrm{s},\ 256\,\mathrm{Hz})$ and red=$(32\,\mathrm{s},\ 64\,\mathrm{Hz})$, where each tuple gives (segment duration, sample rate).

arbitrary amount. To do so we use the PyCBC software package [18]. 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 LalSuite [54]. Therefore all results obtained using this data only hold for stationary gaussian noise. All waveforms were generated using the approximant "TaylorF2" as implemented by LalSuite [54]. Out of the 17 parameters that could have been varied, we fixed the spins to 0 and neglected tidal effects for simplicity. Both of these effects are expected to have a very minor effect on actual physical waveforms. 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 $\arccos\,(\mathrm{uniform}(-1, 1))$. Finally the sky-position is isotropic, i.e. θ is distributed like $\arccos\,(\mathrm{uniform}(-1, 1))$ and φ uniform in $[-\pi, \pi]$.

The luminsoity distance r is chosen indirectly by fixing the SNR to some value. This is valid, as the SNR scales inversely with the distance. 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

50

used. According to subsection 2.2, cutting off the waveform early might result in a lower or at least inaccurate value of the SNR, as parts of the signal are unused. 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. 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 for short signals. 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 LalSuite [54]. 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 filter (compare equation (2.72)), thus giving us an upper limit on 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 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 network. 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[18]. We then 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 $3/4 s_t \notin \mathbb{N}$ or $3/4 n_t \notin \mathbb{N}$, 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[19] 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 have 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 $96\,\mathrm{s}$. The resulting chunk is whitened by dividing out the amplitude spectral density associated to the analytic PSD `aLIGOZeroDetHighPower` and re-sampled to match the criteria of the network input. The window is than shifted by $0.25\,\mathrm{s}$ and the process repeated. The temporal resolution of $0.25\,\mathrm{s}$ was chosen because the waveforms are shifted around in the training set by $\pm 0.25\,\mathrm{s}$. 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.

---

[18]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.

[19]Keras calls this function a generator.

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 these 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 Network Metrics

This section describes the different metrics we use to evaluate the performance of different networks. Not all of them were used throughout the entire time of testing architectures. For the final results we will only quote sensitivities at a given false alarm rate, which were also used to optimize the final networks. Early versions, however, were adjusted to optimize the variance and accuracy of the given predictions.

For the following it is important to note that we train all networks for two different output values. The first one is trained to estimate the SNR of a signal, the second one is trained to give the binary decision of whether or not a GW signal is present. We will refer to these outputs by SNR and p-score respectively.

5.2.1 Variance and Mean Squared Error

The two metrics used to judge the performance of networks early on in the design process were the variance of the data and the MSE compared to the target value. Both of them were only applied to the SNR output. The reason for using these two was the simplicity of implementation and the fact that MSE is used as the loss function to optimize the network. As metric, the MSE gives an estimate of the average error for a prediction and thus is an estimator for the quality of predictions. The variance is an estimate of how far the recovered values spread and is hence a measure for the reliability of the prediction. The assumption was that these two metrics are strongly correlated with the sensitivity, which is the metric we want to report in the end.

Calculating these metrics is rather simple. We define the label values of our data by y_{label} and the predictions by y_{pred}. If y_{label} and y_{pred} contain n samples, the MSE is defined by

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(y_{\text{pred},i} - y_{\text{label},i} \right)^2. \tag{5.2}$$

The variance is given by

$$\text{Variance} = \frac{1}{n} \sum_{i=1}^{n} \left(y_{\text{pred},i} - y_{\text{label},i} - \text{mean}(y_{\text{pred}} - y_{\text{label}}) \right)^2, \tag{5.3}$$

with

$$\text{mean}(x) = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{5.4}$$

We aim to minimize both these metrics.

5.2.2 False Alarm Rate

The false alarm rate is a measure of significance for any reported value. If we recover some value of SNR for given data, we need to know how reliable this value is and how likely it is to originate from pure noise. The false alarm rate therefore gives the number of noise samples per month[20] that produce a higher ranking statistic than a given threshold. To estimate this function we need to sample

$$f(x) = \text{number of noise samples estimated louder than } x \text{ per month.} \quad (5.5)$$

As we have two different ranking statics with our two outputs SNR and p-score, we also need to calculate two different false alarm rates. We furthermore need to differentiate between the discrete training/validation set and the continuous testing set. We will start by describing the process of estimating the false alarm rates for the training/validation set and afterwards go into detail about the differences for the testing set.

To sample f, we use the prediction values x_i of our network for either of the two ranking statistics on pure noise samples. Let m be the number of noise samples we have in the set and denote any specific noise sample by n_i. To estimate $f_i := f(x_i)$, we define

$$f_i' := |\{x_j > x_i | j \in [1, m]\}|. \quad (5.6)$$

A month for this work is defined to be 30 days.
f_i' is the number of noise samples n_j that got assigned a ranking statistic x_j larger than x_i. It can therefore be understood as a cumulative sum. To convert this number to samples per time, we need to know how much time the m samples span. To do so, we observe that for each signal the waveform is allowed to move around in the noise background by ± 0.25 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 s. With this information the observation time m samples cover is $m \cdot 0.25$ 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, where $2,592,000$ is the number of seconds per month.

For a continuous stretch of data the notion of a noise sample is not as strictly defined. If we shift the window across the data, it might at some point contain only parts of a waveform. To resolve this issue, we set a threshold for the ranking statistic. If for any position of the window this threshold is exceeded we take this as a signal candidate. We then shift the network across the entire data with a step size of 0.25 s and take note of every position where the threshold is exceeded. In a next step these positions in time are clustered as a signal usually exceeds the threshold for multiple successive window positions. We do so by requiring two signal candidates to be within 1 s of each other to belong to the same cluster. Each cluster of signal candidates is then reduced to a single point in time by choosing the time where the ranking statistic is maximal. The signal candidate is finally assumed to be a false positive, i.e. originate from noise, if there is

---

[20]We define a month as 30 days.

no injection within a time span of $\pm 3\,\mathrm{s}$ of it. The injection times are the times in the data where the waveform of an injection shuts off.

The false positive rate for continuous data is therefore a function of the threshold and gives the number of false positives per month at threshold x. To convert the number of false positives to samples per month, we again divide by the observation time, which in this case is given by the duration of analyzed data, i.e. $4\,927\,488\,\mathrm{s}$ for the final testing set. Finally we multiply it by $2,592,000$ to get it in units samples per month.

5.2.3 Sensitivity

The sensitivity is the metric we are mostly trying to optimize. It gives the ratio of detected signals over total signals. To claim a detection we need to set a threshold for the detection statistic. Choosing this value in turn fixes the false alarm rate. So we can only claim statistical significance of the sensitivity if it is quoted alongside the associated false alarm rate.

As for the false alarm rate, we evaluate the sensitivity for both outputs individually. There are also slight differences between calculating it for the discrete training/validation set and the continuous testing set. Finally we can compute a volume from the found and missed signals in which our analysis is sensitive enough to find GW-sources. We give these volumes in terms of a radius of a corresponding sphere. We will again start off by describing the process for the training/validation set.

The first step to calculating the sensitivity is fixing a threshold above which the network is thought of to have found a signal. As the training/validation set usually contain comparatively few samples, the false alarm rate can't be resolved at high thresholds or in other words we are not able to sample low false alarm rates. For this reason we choose the maximum value of the detection statistic of any noise sample as threshold. This usually corresponds to a false alarm rate of $\mathcal{O}(10^3)\frac{\text{samples}}{\text{month}}$.

We then bin the training samples that contain a signal by their injected SNR and compute the ratio of how many samples in a bin get marked as signal over the total number of samples in that bin. For the training/validation set we do not compute the sensitive volume, as that would add another layer of complication and isn't useful for optimizing the performance, as the sensitive volume scales with the sensitivity.

For the testing set we plot the sensitivity against the false alarm rate rather than plotting it against SNR at fixed false alarm rate. We choose to plot against false alarm rate in order to have results that are comparable to matched filter results.

To obtain the sensitivity we need to figure out which injections have been found by the network and which have been missed. Therefore, the network is shifted across the continuous test set with a step size of $0.25\,\mathrm{s}$. This creates a time series of detection statistic values. We then apply a threshold to this time series and flag every time that exceeds it as possible detection. These possible detections are then clustered as explained in subsubsection 5.2.2 and declared a true detection if an injection is within a time span of $\pm 3\,\mathrm{s}$. This generates a list of found injections as well as a list of missed injections. We furthermore know the parameters of each injection and can therefore infer an estimate

of the sensitive volume using the PyCBC function `volume_montecarlo`.

Fixing the threshold to some value also fixes the false alarm rate. We can therefore plot the sensitive volume against false alarm rate.

5.3 Evolution of the Architecture

This section gives an overview of the steps that were taken to arrive at the final architecture. It chronologically highlights the pivotal points along the way and showcases some ideas that did not work out.

5.3.1 Convolutional Network

To start out we decided to test how well an architecture that is known to work for BBH signals holds up when it is trained on BNS signals. We therefore used an adaptation of the convolutional architecture introduced in [49].

As we only wanted to test how well the network could adapt to the new and different challenge, we decided to start with signals that are easy to find with traditional methods. Therefore, 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 modeled the output of the two detectors Hanford and Livingston, using simulated noise and the waveform approximant TaylorF2. It contained samples of signals and pure noise realizations with a split of $1/2$ between the two. This data was further used until subsubsection 5.3.4.

The neural network consisted of 3 stacked convolutional layers, doubling the number of filters with every layer. The kernel size on the other hand was kept fixed at 16. We also introduced batch normalization layers in between each convolutional layer and the corresponding ReLU activation, as doing so had shown slight improvements when testing the network with BBH data. As suggested by the original work, a maximum pooling layer was used after every convolution stack. Since the input data to our network is sampled at multiple rates, this network has 14 input channels instead of the 1 used in [49]. (7 input channels per detector, where each of the 7 channels corresponds to a single sample rate) We also introduce a second output to the network and therefore use the output of the final convolutional stack as input for two different stacks of dense layers. The first reduced the number of neurons down to a single one, which has the SNR as training goal, the second one condenses the output down to 2 values, which correspond to a binary decision. We will refer to these two outputs by the SNR output and p-score output respectively. The two outputs use different loss functions to optimize the network. The SNR output is trained to match the injected SNR using MSE whereas the p-score output gives two numbers that add up to 1. The first of these two numbers is trained to be 1 if the data contains a GW and to be 0 if it is a noise sample, the other is simply the inverse of the first. For this output we use a categorical crossentropy as loss. The total loss of the network is a weighted sum of these two outputs. We apply a weight of 1 to the SNR output and a weight of 0.5 to the p-score output, emphasizing the importance of recovering the SNR.

To rate the performance of this network we used only the MSE and variance of the SNR output and the false positive ratio of the p-score output with a threshold of 0.5. From these the sensitivity and false alarm rate can only be eyeballed. We will therefore also include crude estimates of these statistics here to put the performance of these networks into context of later improvements.

With a MSE of ~ 6.5 the network was able to recover the SNR rather well. The variance was on the same scale, also evaluating to ~ 6.5. The spread in recovered values is mostly down to the large SNR range attributed to pure noise samples, reaching from 0 up to 20. Using SNR 20 as a threshold to claim detection, the sensitivity can be eyeballed to reach 100% only above SNR ~ 25 and dropping close to 0% below SNR ~ 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, we try to optimize our network for the SNR-range of 5-15. 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 $6 \times 10^5 \frac{\text{samples}}{\text{month}}$[21]. Ideally the false alarm rate should not exceed 1 per 2 months above an SNR of about 8, as that false alarm rate is the threshold used to alert astronomers [25].

From this we concluded that it is possible for a NN to learn to classify BNS data but further improvements to the architecture are necessary to reach desirable sensitivities at acceptable false alarm rates.

5.3.2 Inception Network

The first major improvement to the original architecture was the introduction of the inception module [34]. This new module had shown strong improvements for the image recognition challenge ILSVRC in 2014, replacing traditional convolutional layers. As image recognition is one of the most active fields for the development of new NN architectures, we decided to test if their findings are applicable to time series data as well.

We applied the same data and used the same output data as in subsubsection 5.3.1.

The inception modules were a direct adaptation of the 2 dimensional variant presented in [34]. Details can be found in subsubsection 3.4.3. To get the best comparison of performance to the pure convolutional architecture of subsubsection 5.3.1, we used basically the same architecture. The three convolutional layers of the old architecture were simply replaced by inception modules. The number of filters, however was kept constant and the kernel size was adjusted. The original work suggests to use kernel sizes of $(1, 3, 5)$ within the inception module. We found that using slightly larger kernel sizes $(4, 8, 16)$ improved our statistics significantly. Another adaptation to the architecture used in subsubsection 5.3.1 is the use of two further convolutional layers before the first inception module. These again improved performance and were inspired by [34]. Overall the network consisted of two convolution layers, followed by three inception modules and

---

[21]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.

57

funneled down to the final outputs using two dense layers.

With these alterations to the architecture the MSE dropped to ~ 4.5 and the variance to ~ 3.9. These are significant improvements over subsubsection 5.3.1. As reference, the goal was to reach values of MSE and variance around 2 in order to use more difficult to detect data. Furthermore, the false positive rate dropped to $\sim 0.015\%$ which corresponds to a false alarm rate of only $\sim 10 \times 10^3 \frac{\text{samples}}{\text{month}}$. This is already pretty good for first tests. The sensitivity at this false alarm rate went up to 100% around SNR 20 and only dropped to 0% below SNR ~ 12. This improvement is down to the loudest false positive only being estimated to have SNR ~ 15.

Though this is still not an impressive performance, it is a considerable improvement over the simple convolutional approach and shows the potential the inception architecture has for time series data.

We also tried to evaluate the necessity of all the different sample rates for the network, as it can be beneficial in multiple ways to reduce the number of input samples. We therefore trained the same network using different sample rates. Adding sample rates one by one, the first network was trained using only the 2048 Hz data, the second used both the 2048 Hz and the 1024 Hz data and so on. We also tried to use only the three sample rates 2048 Hz, 512 Hz and 128 Hz, as we expected them to contain the bulk of the information. We found however that each sample rate, except for 64 Hz, improved the performance of the network. Using only the three sample rates that we expected to work best, however, was also very close in performance to using all sample rates in this test. The 64 Hz sample rate on the contrary seemed to hurt performance.

We also tested different depths of inception architectures and found that shallower networks usually perform a bit better than deeper ones. This probably points to a vanishing gradient problem. We however did not verify this hypothesis.

5.3.3 Collect-Inception Network

As we are using multi-rate sampling for our input data, we wanted to test if it would be beneficial to the performance to have the network analyze each sample rate separately and and only combine the results in the end. Considering the findings of subsubsection 5.3.2 only the three sample rates 2048 Hz, 512 Hz and 128 Hz were used in the beginning. The data was split for the three inputs, each containing the data for 2 detectors.

The architecture consisted of three inception networks side by side, the results of which were concatenated and put through some final dense layers. Each inception network consisted of multiple stacked inception modules. In the beginning these did not all have the same depth, but we empirically found the network would perform best if each stack contained the same number of inception layers. With further testing we found that shallower networks for each sample rate and a small inception network after concatenating the stacks improved results even further. The final iteration of improvement found that using all sample rates improved the performance of this architecture even further. Therefore the final network, for which we will give the metrics below, consisted of 7

parallel stacks of three inception modules each. The results of these stacks are then concatenated and used as input for two further inception modules. Their output is finally passed to two further dense layers for each output.

This architecture managed to bring down the MSE to ~ 2.5 and the variance to ~ 2.4. With this we had just about reached our goals for the simple data. The false positive rate, however, was higher by about 20 times for this network when compared to the architecture presented in subsubsection 5.3.2. Furthermore, the false alarm rate for the loudest noise sample is about twice that of the previous network. The sensitivity at this higher false alarm rate on the other hand also improved slightly. It didn't drop to 0% even at SNR 10 and reaching 100% at SNR ~ 18. However, we did not consider the latter metrics at the time of testing the networks and only looked at MSE and variance. For this reason, the new collect-inception architecture was the clear choice to move on. Later findings will back this conclusion up, even though we should not have drawn it for this architecture.

Having a worse estimate of the false alarm rate is due to the a lower number of noise samples used in the validation set. At the time we were testing many different architectures and data setups. For that reason not all validation sets were similar enough to draw good conclusions. For others who try to compare different architectures we therefore want to point out that using a single validation set for all networks is vital for the efficient improvement of architectures.

As we had reached our goal of MSE and variance for the easy data with this network we changed to use more difficult data from here on out. For details see the following section. We will refer to a network consisting of multiple parallel stacks of inception layers as "collect-inception network" from here on out.

5.3.4 Temporal Convolutional Denoiser

After the success of the collect-inception network introduced in subsubsection 5.3.3 we were inspired by [52] to try a new kind of network, called temporal convolutional network (TCN). Their results suggested that a stack of dilated convolutional layers outperforms normal stacked convolutional layers when classifying data containing GW signals. Their idea was backed up by findings of [44], who showed that TCN are better suited for sequence data than other architectures.

Since the collect-inception network achieved our metric goals for the simple data, we started to use more difficult data from this point on. We still kept all parameters but the SNR fixed, but varied the latter one between 8 and 15 instead. We furthermore feed the network noise and signals separately now and add them together on the first layer of the network. Doing so has many advantages. First of all it gives us more effective samples (see subsection 5.1 for details). The second reason comes from the properties of the TCN. 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. 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 [51]). It is, however, not trivial to do so and for our purposes impractical[22]. 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 [53], we used the TCN as a denoising stage in front of every input of the collect-inception network. To train the TCN to be a denoiser, we need the pure waveform as training goal. It is therefore convenient to store waveform and noise separated.

The TCN consisted of 12 stacked dilated convolutional layers, where the dilation factor was set to 2^i with i being the number of convolutional layers that came before. The number of convolutional layers was chosen such that the receptive field of the network spanned the entire 4096 input samples. For details of the exact implementation of the TCN see subsubsection 3.4.4. We used one TCN in front of every input to a collect-inception network. To train the TCNs to be denoisers, each of them was trained to recover the pure waveform from noisy data. Therefore an auxiliary output was added after every TCN that got the pure input waveform as training goal. The used loss was MSE and assigned a weight of 1. Its output is then added back onto the sum of noise and signal and collectively used as input for a collect-inception network with an architecture that was discussed in subsubsection 5.3.3. 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.

This new architecture and data was also the point in time where we started to use the false alarm rate and sensitivity as metrics to judge the performance. The false alarm rate can be estimated to be $\sim 250\,\frac{\text{samples}}{\text{month}}$ at SNR ~ 9.5. Fixing the false alarm rate to this value allows us to calculate the sensitivity, which outperformed any previous architecture. It did not drop below 20% in any SNR bin even down to SNR 8. Furthermore the sensitivity stayed above 80% for SNRs ≤ 12. All these values are achieved at lower false alarm rates than those of subsubsection 5.3.2 and subsubsection 5.3.3. To still compare the MSE and variance to earlier architectures, the MSE reached a value of 2.7 and the variance 2.3. These are slightly worse values than recorded before but achieved using more difficult data. We also tested to use the architecture presented in subsubsection 5.3.3 on this more difficult data set and found that even though the MSE and variance at 2.6 and 1.9 are lower, the sensitivity is worse. This is especially true for the lower SNR bins. However, the false alarm rate used to determine the sensitivity again was different to that used to estimate the sensitivity of the collect-inception network using a TCN as a denoiser. For the pure collect-inception network the false alarm rate was set to be $\sim 100\,\frac{\text{samples}}{\text{month}}$. We did, however, not consider this offset in false alarm rate at the time. A collect-inception network using a TCN as a denoiser will be called a TCN-collect-inception network or short TCIN from here on out.

Even though the TCIN architecture was considered to work a lot better than previous

---

[22]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.

attempts it had the major downside of complexity. Using an individual TCN for each sample rate and following it up with a collect-inception network required about 30 GB of memory with a mini-batch size of only 24 samples. The only graphics card that can has enough video memory is the NVIDIA GV100. The availability of this hardware is very limited and thus does not allow for rapid development. A smaller alternative would therefore be highly 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.

5.3.5 Evolution using Bad Data

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 performance, residual connections were added to every inception layer 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 [35]. 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 convolutional 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
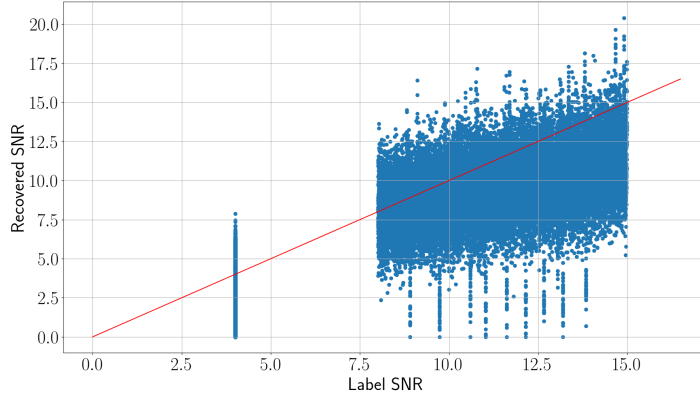
Figure 5.2: Shown are the recovered SNR values on the y-axis and the label values on the x-axis. Each SNR label has multiple y-values assigned to it, as the same waveform is submerged in multiple different noise realizations. The red line indicates where the dots should lie if the network recovered them 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 reduced the number of streaks down to a single one.

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 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 being assigned 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.

After the error in the data generator from subsubsection 5.3.5 was caught, we tried to evaluate how large of an impact it made. We therefore tested the different architectures we had found to be working well for the flawed data on the corrected data, which still varied component masses, coalescence phase, sky position and inclination.

The inception-res network experienced a drop in sensitivity to below 20% in the loudest SNR bin. The collect-inception-res network held up a bit better, with its sensitivity dropping only to 40% in the loudest SNR bin.

We then tied some minor alterations of the collect-inception-res network architecture like changing the number of inception modules or training only for p-score, but nothing helped to improve the performance.

5.3.6   Testing Alternative Loss Functions and Layers

After the performance of our networks on the difficult data could not be recovered by simple alterations of the architecture, we decided to try and optimize our algorithm to the specific problem of GW detection.

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 swapped, i.e. the loss grows exponentially for underestimating signals and linearly for overestimating them. The details on how this loss looks like
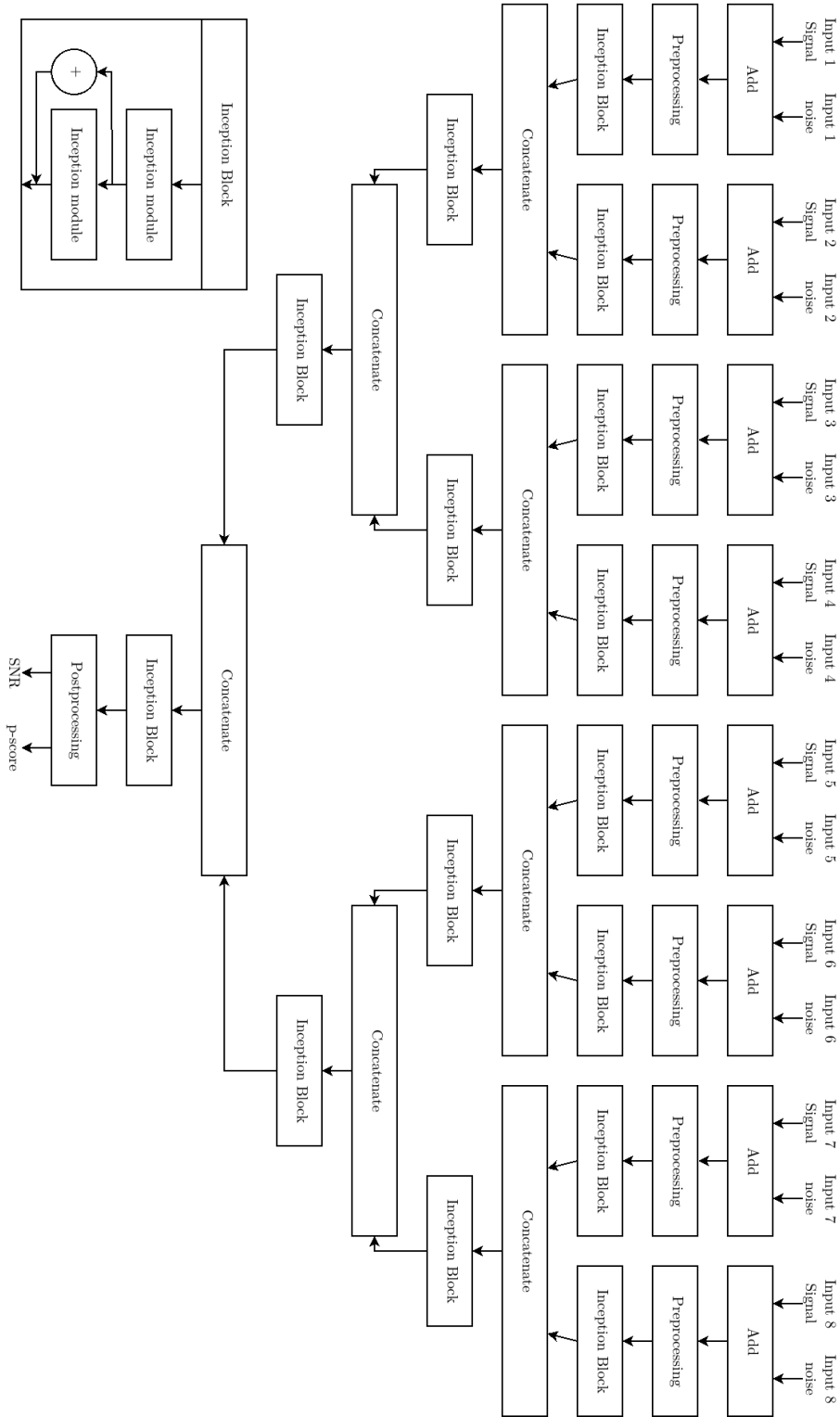
Figure 5.3: The architecture of a cascading collect-inception-res network. The preprocessing layers are shown in Figure C.3. The postprocessing layer is shown in Figure C.5.

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.

We also briefly tried to use a mean average percentage error as loss function for the SNR output to penalize errors at low values more strongly. The results of this different loss function were mixed but mostly resulted in worse performance.

Another approach we took was close in nature to that of SincNet [55]. 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 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 [56]. 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 convolutional layer. All other parameters are kept constant. For the convolutional 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 convolutional 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 convolutional 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 perfor-

mance may be down to pure chance. Secondly, the networks were only followed by a single convolutional 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 convolutional 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 in subsubsection 5.3.1 to subsubsection 5.3.5. 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 then 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 5.4.1. We then used this final architecture and trained it on the data described in subsection 5.1 to finally evaluate it.

5.3.7 Miscellaneous Findings

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. On the other hand it generally 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.

5.4 Final Network

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.

5.4.1 Architecture

All illustrations for this section can be found in Appendix C.

The final architecture is a collect-inception-res network with TCNs as denoisers for every stack. Each stack is attributed its own sample rate and they cascade down to two outputs. The first output is supposed to give an estimate of the SNR of the input data whereas the second one gives a p-score, a value that is used for binary classification, i.e. gives information about whether or not a GW is present in the data. To stress again, the p-score is not a probability but just a value that is bounded by 0 and 1 that, when a threshold is applied, gives a binary answer. A high-level overview of the network is shown in Figure C.1.

We will now give a detailed description of each module depicted in Figure C.1, working our way down from the input. We will only discuss each module once, as all of them share the same parameters.

The inputs are labeled 1 to 8, each consisting of an individual input for signal and noise. The order of these inputs is the reversed order of the chopped up time series, i.e. input 1 corresponds to the last $0.5\,\text{s}$ of data sampled at $4096\,\text{Hz}$, input 2 is the interval from $0.5\,\text{s}$ to $1\,\text{s}$ sampled at $4096\,\text{Hz}$ and so on. The final input thus is the last $32\,\text{s}$ sampled at $64\,\text{Hz}$ (for details see subsection 5.1). The first layer in each stack simply adds the two stack-inputs for signal and noise together. This sum is then fed to a TCN.

The TCN stacks 11 blocks of dilated convolutional layers with causal connections on top of each other. Every block has a total kernel size of three and the dilation is set to 2^i, where i is the depth of the TCN. Furthermore, each block consists of two stacked convolutional layers with dilation rate 2^i, each followed by a batch normalization, ReLU activation and dropout layer. The latter has a dropout rate of 0.1. Finally, the entire block is preceded by a dimensional reduction layer and wrapped with a residual connection. (see Figure C.2) The implementation follows [44], replacing their weight normalization by batch normalization, as Keras does not provide a weight normalization layer. The depth was chosen such that the receptive field of the TCN encapsulates all 2048 input samples.

Each of the TCN has their own training goal, which is given by the corresponding signal input. The loss for this part is chosen to be MSE and added to the total loss with a weight factor of 0.1. The weight is set so low as the TCN is not able to reliably recover the waveform from the input data and we don't want the network to try and optimize just the TCNs without optimizing the final outputs we care about. With this training goal, the TCN is meant to act as a denoiser to the input data. Its output is then added back onto the input, to amplify any signal it found. To avoid throwing away information the TCN could not recover, we only amplify the signal and don't just feed the output itself to the following layers.

As a next step we do some preprocessing for the following collect-inception-res network. The authors of [34] found that using a few simple convolutional layers before a stack of inception modules helps the network learn. We could verify these findings when testing different inception networks and thus included this preprocessing step. It starts of with a batch normalization layer, followed by a dropout layer with a dropout rate of 0.25.

They are followed by two stacked blocks of convolution, batch normalization and ReLU activation, separated by a max pooling layer with pool size 4. See Figure C.3 for a visual aid.

The main workhorse for the collect-inception-res network are the inception blocks. Each of them contain two stacked inception modules, separated by a batch normalization layer. The second inception module also utilizes a residual connection. The first one could be equipped with a residual connection as well if it was preceded by a dimensional reduction layer, adjusting the number of channels to 224. Within the inception modules, we empirically chose the number of filters to be descending for an increasing kernel size. We furthermore found that kernel sizes $(1, 2, 3)$ work best in these modules. These parameters could probably be optimized even further, but we could not do so in a feasible amount of time. The inception block is depicted in Figure C.4.

All concatenation layers except the final one are furthermore equipped with another auxiliary output that use the SNR label as a training goal. To reduce each of these layers down to a single value we use a stack of one average pooling layer with pool size 8, one dimensional reduction layer with 16 filters and one final dense layer with a single neuron and a ReLU activation function. The output of the dimensional reduction is flattened to be usable for a dense layer. This action was taken to help against vanishing gradients and was inspired by the architecture of [34]. The loss used is MSE and is also weighted by 0.1.

The post-processing consists of a max pooling layer with pool size 4, a dimensional reduction layer with 32 filters and, depending on the output, two dense layers with neuron numbers $(2, 1)$ for the SNR output and $(3, 2)$ for the p-score output. For the SNR output we use a ReLU activation function and for the p-score output a softmax activation function. The loss for the SNR output is MSE and has a weight of 1 whereas the p-score output uses a categorical crossentropy as loss and a weight of 0.5 to have more of an emphasis on the SNR. The postprocessing step is shown in detail in Figure C.5.

This architecture was derived empirically by the process described in subsection 5.3 as it showed the best performance in regards to the sensitivity on the validation set. The specifics of the performance are discussed in the next section. It does, however, come with multiple drawbacks. The general problem of this architecture is its size and complicated structure. The first obvious complication is the amount of necessary video memory. The model takes about $18\,\mathrm{GB}$ to train with a mini-batch size of 24. The GPU with the largest memory we could use for a full training run was a NVIDIA Titan V with a capacity $\sim 12\,\mathrm{GB}$. To train this network on such limited memory required us to decrease the mini-batch size to 16. The network just barely fit into memory and Tensorflow stated that additional performance might be gained if more memory was available. This is backed by the GPU utilization that constantly jumped between 20% and 100%. Overall each epoch training on the stated $322,500$ sample plus the validation step took $\sim 6\,\mathrm{h}$ to finish.

As each epoch takes such a long time, it is very difficult to optimize the hyper parameters, like number of inception modules, kernel sizes within these modules, number of filters in these modules, dropout rate, etc.

Finally, a network of this depth is always vulnerable to the vanishing gradient problem. We try to combat this with the auxiliary outputs and their loss functions but cannot rule it out.

5.4.2 Network Performance

To evaluate the performance of the final architecture we use a testing set that contains $4\,927\,488\,\text{s} = \sim57\,\text{d}$ of continuous data. This data was created independently of the training and validation set and split into 1203 parts of length 4096 s. Each of these parts is evaluated by the trained network, which was described in detail in subsubsection 5.4.1, using a moving window with step size 0.25 s to generate the correct input data format. Therefore, each window is whitened by the analytic PSD `aLIGOZeroDetHighPower` which is provided by the software library LalSuite [54] and re-sampled. The details on how these processing steps take place are discussed in subsection 5.1. The output of this procedure are two time series of duration 4024 s and time resolution 0.25 s. These two time series contain the predicted SNR and p-score values at each point where a window was evaluated. The difference of 72 s between the input and the output is due to the time span of the window that is used.

The testing set contained 37758 injections. They are spaced out to appear every 200 s with the exact position varying by $\pm20\,\text{s}$, but are not placed within the first or final 256 s of any of the 1203 parts. The source parameters are distributed in the following way: Both component masses are drawn from a uniform distribution between $1.2\,\text{M}_\odot$ to $1.6\,\text{M}_\odot$, the coalescence phase and polarization angle are uniformly distributed within the interval $[0, 2\pi]$ and the inclination is drawn from $\arccos(x)$, where x is uniformly distributed between -1 and 1. We still neglect spins and tidal deformabilities, as both of these effects are expected to be small for astrophysical events. The sky position is distributed volumetrically, meaning that the right ascension is uniformly distributed between 0 and 2π, the declination is drawn from $\arccos(x) - \pi/2$, where x again is uniformly distributed between -1 and 1, and the distance goes as r^2 between $0\,\text{Mpc}$ to $400\,\text{Mpc}$. The waveforms are generated using the model TaylorF2. All noise is generated using the PSD `aLIGOZeroDetHighPower` of LalSuite.

To evaluate the performance of our analysis, we calculate the sensitivity of the search at given false alarm rates. At this point we want to address the comments of [51]. They suggest that no statistically meaningful false alarm rate can be derived from a NN-based approach that classifies data into the two categories "noise" and "signal". They hinge their argument on multiple points. Firstly, the network is trained with a certain ratio of examples for signals and for pure noise. False alarm rates derived from the false positive rate on the training/validation set are therefore only applicable if the final distribution of signals within the data is similar to that within the training set. This is however a problem as signals in real data are comparatively rare, whereas for efficient training the ratio of signals to noise should be close to 1 : 1. We address this issue by using different ratios in all three sets used for the network. Specifically the training set has a ratio of roughly 1 : 1, the validation set has a ratio of roughly 3 : 1 and the testing set has on

average a ratio of 6 : 194.

The second issue the authors of [51] point out is the significance assigned to each detection candidate. In the case of binary classification the output would always be thresholded to yield either 1 or 0. Therefore, the false alarm rate assigned to every event would be the same. As we are training for SNR and p-score, we do have the means of assigning different false alarm rates to different events, as we can determine the false alarm rate at the recovered SNR. This could however also be done by using the p-score as a ranking statistic.

The third point of critique of [51] is the use of a sliding window approach and how to treat parts of the output, where the network jumps between claiming a detection and claiming no detection at successive points in time. To get around this issue, we cluster detection candidates into blocks, where the network is confident to have found a signal. If any detection candidates are within $\pm 1\,\mathrm{s}$ of another detection candidate they are clustered together and thus believed to belong to the same event. Any times in between are also believed to belong to the event regardless of whether or not they exceed the threshold value that is set to claim detection. To give a single point in time where we believe the event to have taken place, the time that is associated to each cluster is the maximum of either the SNR or the p-score output. As such we get two sets of detection candidates. We get another one if we only allow detection candidates that exceed the threshold in both ranking statistics.

The final issue the authors of [51] address is the fact that during training all waveforms are provided to the network at within a very specific time interval. They say that NNs are known to pick up on such subtle details and thus might not be able to handle waveforms that are not correctly aligned. Therefore a sliding window approach might not be suitable. We think that using a sliding window on continuous data to generate false alarm rates and sensitivities addresses this problem.

We start the discussion of the new architecture by examining both outputs and their statistics separately. The results are compared to matched filtering based searches and to the findings of [30]. We then consider the combination of both outputs to form a single statistic and discuss the differences to using a single ranking statistic. Finally, we finish this section by comparing the computational efficiency of our algorithm to a matched filtering based approach.

The false alarm rate of the SNR output is shown in the top row of Figure 5.4. To use our algorithm to alert astronomers of a GW event, we would need to reach false alarm rates of about 1 per 2 month. Evaluating the false alarm rate to SNR 20 yields a false alarm rate of 38.4 false alarms per month. It is even questionable if the network is capable to reach lower false alarm rates, as some false positives were attributed very high SNR values. The loudest false positive was assigned a value of 123672.85. One could possibly introduce a high cutoff above which detection candidates are vetoed. Choosing such a cutoff, however, would need to be done with great care to not veto a lot of loud physical events.

We trained the network using waveforms with injection SNR down to values of 8. Using this value as a threshold gives a false alarm rate of 107.8. The upper limit for training

waveforms was given by SNR 15. At this value we have a false alarm rate of 44.7. Even though the false alarm rate does improve throughout the range of provided training examples, it is not by much if we use the total change as reference. Most of the false alarm are given below the training range, i.e. most noise samples are assigned a SNR value < 8. As the high value for the loudest false positive and the overall flatness of the false alarm rate at high SNRs suggests, the greatest contribution to the false alarm rate comes from very loud glitches. Early tests suggested that these glitches usually exceed the threshold value only for 1 or 2 samples. Therefore, vetoing detection candidates that have high values and exceed the threshold only briefly might improve overall performance. Further careful analysis would be necessary to give a final conclusion.

Pure noise samples were attributed a label of SNR 4. We therefore expect very high false alarm rates at that value. Indeed we get a false alarm rate of 14704.6 at that threshold. Finally we want to compare the false alarm rate of the loudest noise sample from the validation set with its corresponding false alarm rate on the testing set. The highest value a pure noise sample from the validation set was assigned was 6.7 with a false alarm rate of 192.9. The corresponding false alarm rate on the testing set is 366.6, which is about double the estimate from the validation set. These findings agree with [51] in that it does not seem to be possible to translate a false positive rate on the validation set to a false alarm rate on continuous data.

The false alarm rate of the p-score is shown in the bottom row of Figure 5.4. The left of the two plots shows the false alarm rate against the p-score directly. The right one re-scales the x-axis to give a better overview of the evolution at high p-scores. Overall the false alarm rate drops quite significantly above p-score ~ 0.9. This is caused by a behavior we found for all architectures. The p-score gets pushed to the extreme values 0 and 1 for all cases, i.e. there are only few samples evaluated to a p-score in the range $[0.2, 0.8]$. Therefore, also pure noise samples occasionally get attributed a high p-score, which in turn causes a sudden drop of the false alarm rate only at high values.

At p-score 1 the false alarm rate evaluates to 0, as no sample is allowed to receive a p-score larger 1 by the design of the network. The closest p-score to 1 we evaluated the false alarm rate at is $1 - 10^{-5}$ and has a value of 31.6. This value is again too high to consider using this network to alert astronomers. Based on the findings of the testing set, the smallest false alarm rate this network can achieve is 23.7. This is due to the fact that 45 false positives were assigned a p-score of 1. The smallest p-score we evaluated the false alarm rate for was 0.05 with a false alarm rate of 14717.2.

The highest p-score assigned to any noise sample of the validation set was 0.408 and also had the false alarm rate of 192.9 attributed to it as an estimate. Evaluating the false alarm rate at this threshold on the testing set gives a value of 631.2 which is, again, a lot larger than the one recovered from the validation set. This is further proof that the false alarm rate may not be derived from the validation set but should be estimated from a long continuous stretch of data.

The false alarm rate alone is not useful to us. Instead we want to know how many signals we can reliably recover. For this reason we calculate the volume our search is sensitive to and give the result in terms of the radius of a corresponding sphere. The result is plotted
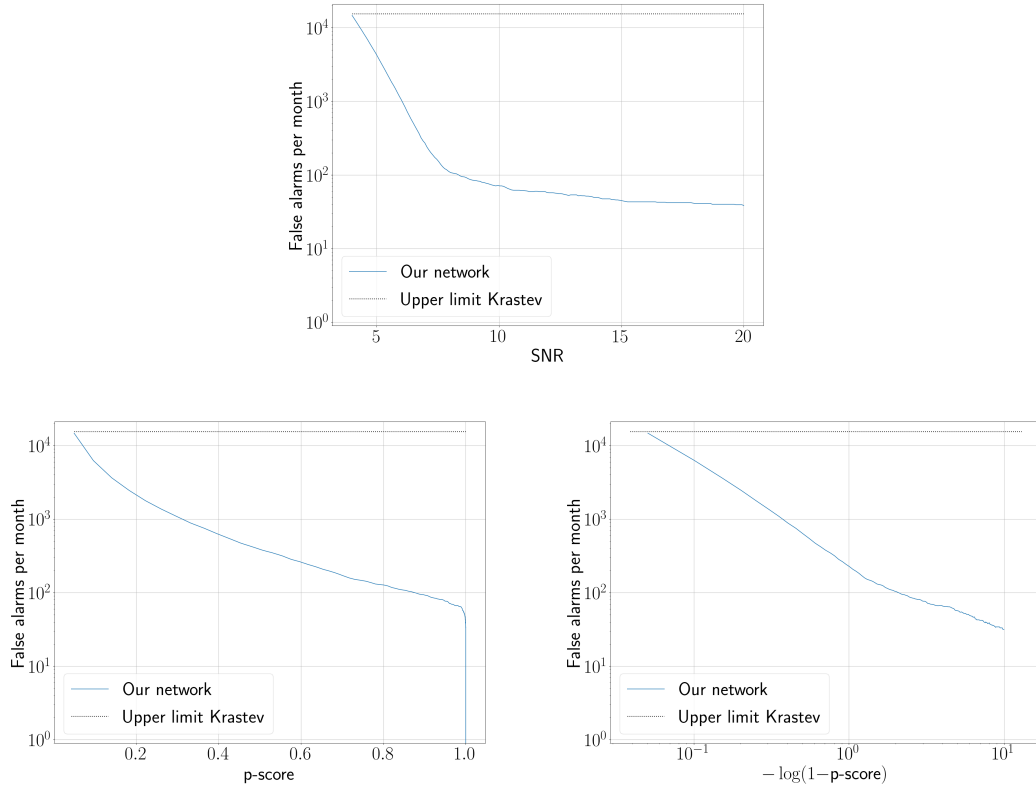
Figure 5.4: The black dotted line in each of the three plots gives the upper limit of the false alarm rate used in [30]. **Top:** The blue curve shows the false alarm rate of the SNR output as a function of the threshold value used to determine detection candidates. **Bottom left:** The blue curve gives the false alarm rate of the p-score output as a function of the threshold used to determine detection candidates. **Bottom right:** This plot rescales the x-axis of the bottom left plot to give a better overview of the evolution of the false alarm rate as the threshold approaches 1.

on the left side of Figure 5.5. Using the SNR output as detection statistic, the search reaches a sensitive radius of 179.2 Mpc at a false alarm rate of $14\,704.6\,\frac{\text{samples}}{\text{month}}$ and drops to 69.2 Mpc at a false alarm rate of $38.4\,\frac{\text{samples}}{\text{month}}$. Using the p-score as detection statistic yields very similar results for high false alarm rates but is advantageous at low values. Here the radius of the sphere to which our search is sensitive drops only to 84.3 Mpc.

This behavior of the sensitivity translates well to the general notion we got from testing multiple different architectures: Using the p-score to binary classify data usually yields better sensitivities at low false alarm rates than using the SNR to predict the strength of the signal.

We can compare these sensitivities to the sensitivity of a matched filter search. As a rule of thumb, a matched filter search comparable to our approach is roughly sensitive to sources that emit GWs which are assigned a SNR of 8 by the search. To get an estimate of the average distance, we calculate the SNR of a BNS system with equal component masses of $1.4\,\text{M}_\odot$ that is oriented face on to our detector at distance 1 Mpc. The distance in Mpc the source would need to be away in order to be detected at SNR 8 is then given by

$$\text{distance}_{\text{SNR}=8} = \frac{\text{SNR}_{\text{distance}=1}}{8}. \tag{5.7}$$

As the system we modeled is face on and perfectly positioned, this is the furthest distance the source may be away from the detector to be assigned a SNR of 8. To get the average distance, this distance needs to be divided by a factor of 2.26 (Alexander Harvey Nitz, personal communication, September 17, 2019). With this the sensitive radius of a matched filter search can be estimated to be $\sim$194.8 Mpc.

Our NN based approach can't achieve this sensitivity even at very high false alarm rates. At somewhat sensible false alarm rates (false alarm rates $< 100\,\frac{\text{samples}}{\text{month}}$) we are sensitive to sources out to about half the distance and therefore have roughly 1/8 of the sensitive volume of a matched filter search. Our algorithm is therefore not able to rival matched filtering in its current state.

We can also compare our results to the findings of [30] in more detail now. To do so, we transfer our findings to their notation to the best of our abilities. The data we use is different in many ways to the one that is used in [30]. Therefore, the results of this comparison have to be taken with great care. The most notable differences are that we use two detectors and vary a lot more parameters, while keeping the tidal deformability fixed. The authors of [30] on the other hand use data from a single detector, vary only component masses and tidal deformabilities and also differentiate between BBH and BNS signals.

To convert our findings to the framework of [30] we calculate the network SNR for every injection. We then convert it to a single detector SNR by dividing by $\sqrt{2}$ and finally convert this single detector SNR to a pSNR. For this conversion we use the factor 39.225 we found in section 4. Furthermore, we evaluate our network at a false alarm rate of $\sim$14\,500 $\frac{\text{samples}}{\text{month}}$ as the false alarm rates of [30] can only be constrained by an upper limit

of $15\,500\,\frac{\text{samples}}{\text{month}}$[23]. The results of our network can be seen on the right side of Figure 5.5. For low values of the pSNR the sensitivity follows an expected curve, growing steadily. Above pSNR ~ 0.3 however the sensitivity starts to become jumpy and inconsistent. This behavior is probably due to the low number of samples above this value. (pSNR $0.3 \approx$ SNR 17) There are only 1484 samples above pSNR 0.3 and only 859 above pSNR 0.4. Below pSNR 0.3 on the other hand lie $36\,274$ samples. Therefore, our comparison will focus mostly on the sensitivity between pSNR 0 and 0.3.

Our analysis reaches a sensitivity of 30% at pSNR 0.27, which is roughly 10% above the sensitivity of [30] at that value. Doing the same comparison for all values below pSNR 0.3 yields the same results: Our network is about 10%-points more sensitive than the network of [30] at low pSNRs. Above pSNR 0.3, however, this behavior breaks down and our approach becomes unstable. This might change if the training set contained louder signals up to pSNR $1 \approx$ SNR 55.5.

We do want to stress again that this comparison is by no means complete, as the data used in both approaches is vastly different.

So far we have only evaluated both ranking statistics individually. To reduce false alarms one can however also combine the two outputs. To do so, we only consider a data point a detection candidate if a threshold in both ranking statistics is exceeded. We then cluster the resulting times the same way we did for a single detection statistic. To determine the final time we report as the recovered position we need to find the time where one detection statistic is maximized. This gives us two options. Either we choose the time where the SNR is maximized or where the p-score is maximized. We evaluated the network in both ways and generated the according false alarm rates (Figure 5.6) and sensitivities (Figure 5.7). We choose to maximize the SNR in the plots where a fixed p-score is chosen and vice versa if the SNR is kept constant.

Using this combined approach reduces the false alarm rates at low values of the detection statistic slightly. The values are however mostly limited by the minimum value of the individual ranking statistics. To be specific, the highest value of the false alarm rate when keeping the p-score threshold fixed at 0.4 is given by 531.8 (compare top of Figure 5.6). Using just the p-score to determine a false alarm rate at threshold 0.4 results in a false alarm rate of 631.2, which is on a similar scale. Once the threshold SNR reaches a value of 20, both the combined output and the the individual output yield a false alarm rate of 38.4. Therefore, the false alarm rate of the combined output seems to be well approximated by choosing the minimum of the false alarm rate of both individual statistics.

Using the combined output also has an effect on the sensitivity of our search. The overall lower false alarm rates lead to lower sensitivities. None of them are capable

---

[23]The authors of [30] provide a false alarm probability of 0.03% which in turn would result in a false alarm rate of $\sim 7750\,\frac{\text{samples}}{\text{month}}$. We can, however, not find any evidence how they are able to achieve these false alarm probabilities, as their testing set contains roughly 1667 noise samples. This in turn means if a single one of them is classified as BNS it would result in a false alarm probability of 0.06%, which is consistent with the false alarm rate we use to compare our results. We did also compare the two analysis at the lower false alarm rate but found very similar results.
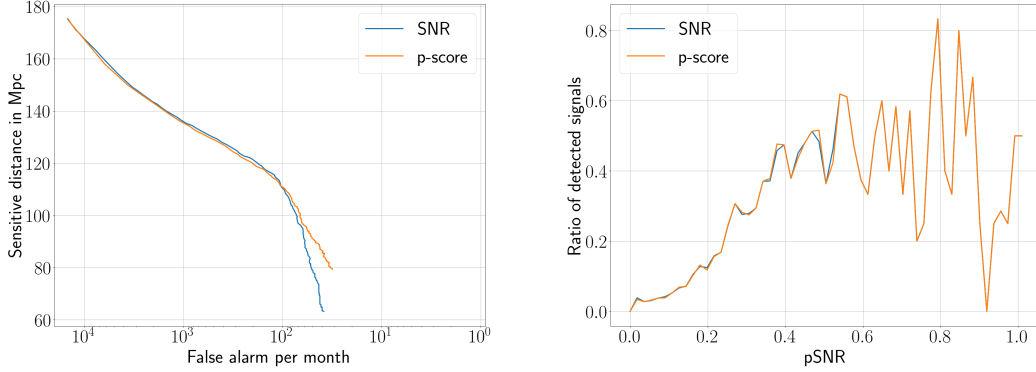
Figure 5.5: The figures give the sensitivity of the search algorithm as a function of the false alarm rate. **Left:** The two curves give the radius of a sphere to which our search is sensitive to. For high false alarm rates both detection statisics yield the same results. For low false alarm rates on the other hand the p-score output becomes more sensitive. **Right:** This plot gives the sensitivity in terms of fraction of detected injections at a given pSNR. The pSNR of an injection was determined by calculating the SNR for both the Hanford and Livingston detector, combining them by taking the root of the sum of the squares, dividing this value by a factor of $\sqrt{2}$ to get an estimate of the single detector SNR and finally devide it by the factor of 39.225 to convert SNR to pSNR. Doing so we hope to achieve results that are comparable to [30]. The sensitivities for both the p-score and SNR output are quoted at the highest false alarm rate of $\sim 14\,500\,\frac{\text{samples}}{\text{month}}$, as this is the closest value to the estimate of the false alarm rate used in [30].

of reaching the sensitivities of using just a single detection statistic, but are lower by 5 Mpc to 20 Mpc. Overall using both outputs enables us to be a bit more confident in found detection candidates but on the contrary makes real signals harder to find. It does therefore not give a real improvement over using just a single detection statistic.

We finally want to compare the computational efficiency of our search to that of matched filtering. Current template based algorithms are capable of evaluating about 5000 templates per core such that results arrive in real time. To be able to compare our search to a matched filtering based one, we calculate a template bank that spans the parameter space our network is optimized for. To do so, we use the program `pycbc_geom_nonspinbank` that comes with the PyCBC library and count the number of templates it generates for component masses in the range from $1.2\,\mathrm{M}_\odot$ to $1.6\,\mathrm{M}_\odot$ at 3.5PN order. The maximum mismatch was set to be 0.965. Such a template bank turns out to contain 1960 templates. Thus a two detector network needs to evaluate 3920 templates to run in real time. Based on the estimate, this is possible on a single core.

To evaluate the speed of our network, we feed it 1024 samples and use the verbose output of Keras for timing. Once all samples are evaluated, Keras prints the average time it took to evaluate one sample. On a dual core, 4 thread laptop CPU (i7-6600U) Keras reported an average time of 245 ms per sample. As each of our samples is equivalent to evaluating 0.25 s of data, our network manages to evaluate data in real time. Using a GPU or more cores would speed up this process significantly. It is however not fast
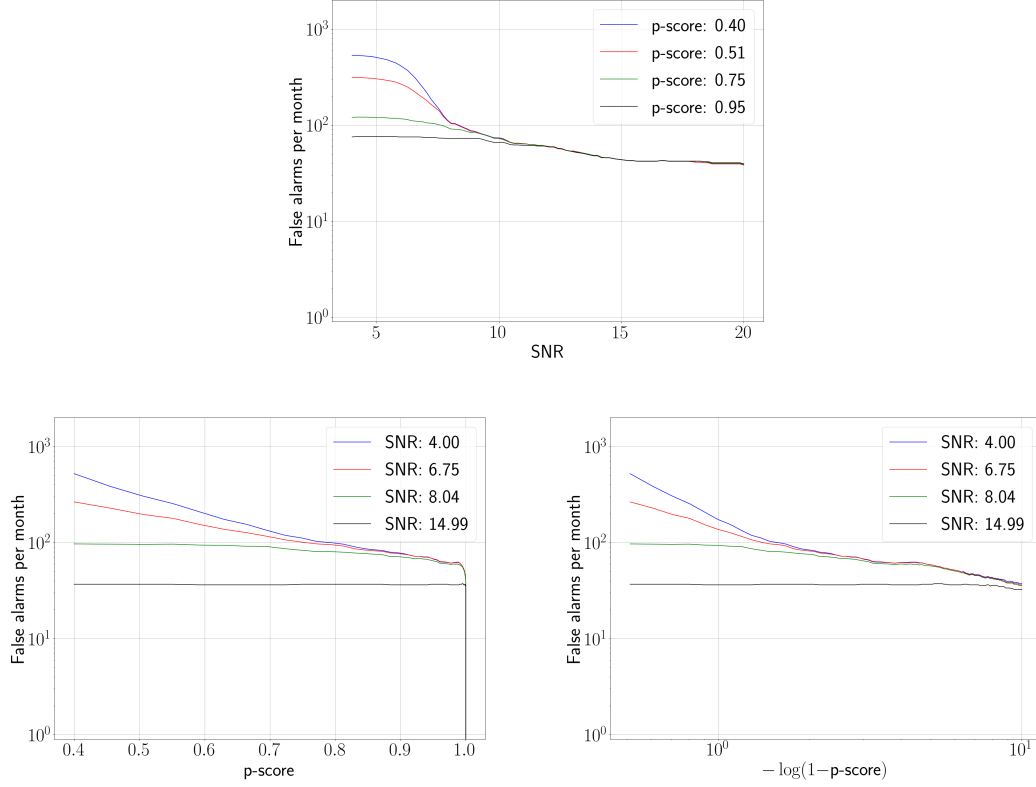
Figure 5.6: The plots show false alarm rates as measured on the test set when we require both detection statistics to exceed a threshold value. **Top:** Shown is the false alarm rate when the threshold for the p-score is fixed to some specific value and the SNR threshold is varied. For this case we use the SNR output to determine the position of a detection candidate, by using the maximum in each cluster. **Bottom left:** Shown is the false alarm rate of the network when fixing a SNR threshold and varying the p-score threshold. The maximum of the p-score in each cluster is used to determine the recovered position of a detection candidate. **Bottom right:** Shown is the false alarm rate when fixing the SNR threshold and varying the p-score threshold. It shows the same data as the bottom left plot but rescales the x-axis to give a better overview of the evolution for low false alarm rates.

76
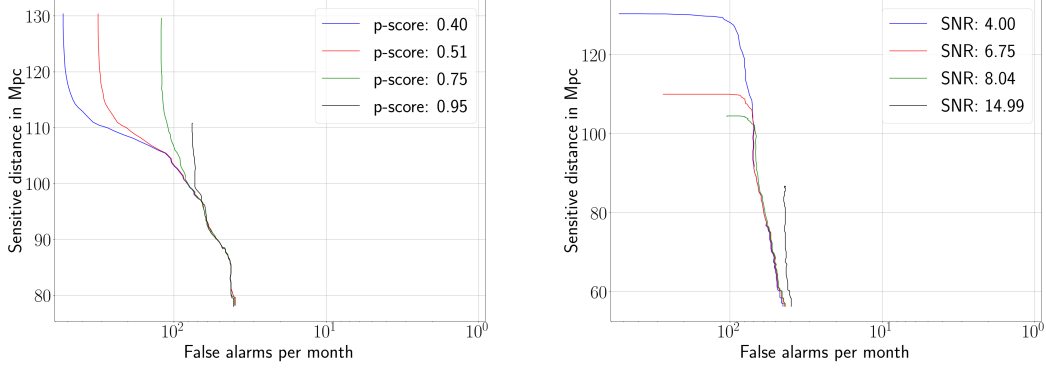
Figure 5.7: Shown are the sensitive distances when using only detection candidates that exceed the threshold for both ranking statistics. **Left:** The sensitive range is plotted at a fixed threshold for the p-score. The false alarm rate is therefore varied by changing the value of the SNR threshold. Accordingly, the time that is reported for each cluster of detection candidates is found by choosing the sample that maximizes the SNR ouput. **Right:** The sensitive range is plotted at a fixed threshold for the SNR output. The false alarm rate is varied by changing the p-score. Accordingly, the time that is reported for each cluster of detection candidates is found by choosing the sample that maximizes the p-score ouput.

enough to beat matched filtering.

To evaluate the speed of the network we did not use real or simulated data but simply fed random numbers that were uniformly distributed between 0 and 1 to the network. This does not have any impact on the speed of the network but was more easy to realize. Curiously the SNR output of the network returned values above 8 for more than 97% of the input samples. Furthermore, with a mean value of 41 the values were not absurdly large, such that it is difficult to spot them by vetoing unreasonably high values for the SNR output. The p-score on the other hand consistently returned 0 for all these samples. It might therefore be sensible to use the p-score output rather than the SNR output, as this one seems to be more robust.

6 Conclusions

This work has explored a completely new set of architectures in their application to GW data analysis. We introduced the concept of using multiple sample rates to reduce the number of input samples to the network and thus decrease the size of the network, while retaining a lot of the information of the long signals that are emitted by BNS systems. Testing multiple different architectures led to the use of inception modules and the TCNs as amplifiers for the input signal. Our design process led to a very deep and wide architecture utilizing multiple techniques pioneered for image recognition tasks. To the best of our knowledge we are also the first to use the SNR as a training goal, thus providing a direct ranking statistic for every event.

We have for the first time tested the sensitivity of a NN-based search at very low false alarm rates for BNS signals. The final architecture suggests that it might be possible to reach meaningful false alarm rates that are low enough to alert astronomers of possible GW events. Though we have not managed to fully reach the goal of acceptable sensitivities at a false alarm rate of $1 \frac{\text{sample}}{\text{month}}$, our network significantly improves upon previously tested architectures.

We do, however, find that a search pipeline using our NN to detect BNS signals is not yet sensible. Current matched filter pipelines are superior in every aspect. Not only are they more sensitive to the signals we used to train our network, but are also computationally more efficient, which was the main reason to explore machine learning search pipelines. The gap in efficiency is mostly down to the high complexity of the final architecture. We furthermore did not test the sensitivity of the network on real detector data which contains non Gaussian noise transients and can thus not claim any implications for a real search.

Using the SNR as training goal has proven to be a difficult problem. Using it to generate detection candidates yields slightly worse sensitivities at low false alarm rates. At high false alarm rates this difference vanishes. We did, however, find that this output is a lot more fragile and can produce realistic detection values for unrelated data. The p-score is a lot more resilient to such inputs and can also be interpreted as a ranking statistic, rather than a binary output.

We hope that our work, although not yielding a useful search pipeline, will prove useful to the field of utilizing machine learning algorithm to detect GWs. This thesis has tested multiple new approaches to the architecture and has found great improvements over the current state of the art convolutional architectures, by deploying inception modules. We have furthermore given an overview of a process which we believe produces reliable estimates of false alarm rates and sensitivities by sliding our network across a continuous input.

As a next step we would like to decrease the complexity of our architecture while retaining its sensitivity. Doing so would yield better computational efficiency and enable us to do more efficient hyper parameter exploration, which in turn will hopefully improve the performance even further. We also want to test different formats for the input data, as others have used shorter parts but sampled at a higher rates. Finally, as a long term

goal, we would like to incorporate and test real detector data and turn this approach into an independent search pipeline. To give security against data that accidentally resonates strongly with the network this pipeline should be backed up by a matched filter pipeline that only covers a small parameter space.

7 Acknowledgments

A   Full Adder as Network

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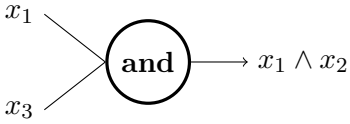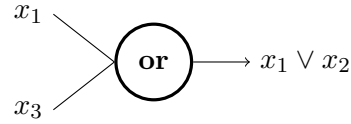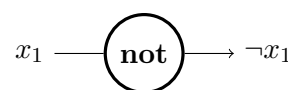
"and"-neuron			"or"-neuron			"not"-neuron	

$\vec{w} = (1, 1)$ $b = -1.5$			$\vec{w} = (1, 1)$ $b = -0.5$			$w = -1$ $b = 0.5$	
x_1	x_2	$a(x_1 + x_2 - 1.5)$	x_1	x_2	$a(x_1 + x_2 - 0.5)$		
0	0	0	0	0	0	x_1	$a(-x_1 + 0.5)$
0	1	0	0	1	1	0	1
1	0	0	1	0	1	1	0
1	1	1	1	1	1		

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

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

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

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

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

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

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

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

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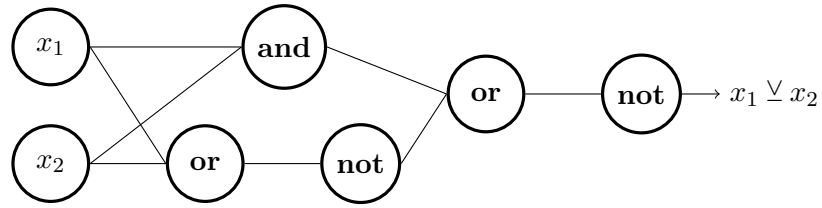

Figure A.1: The definition of a network that is equivalent to an "XOR"-gate.
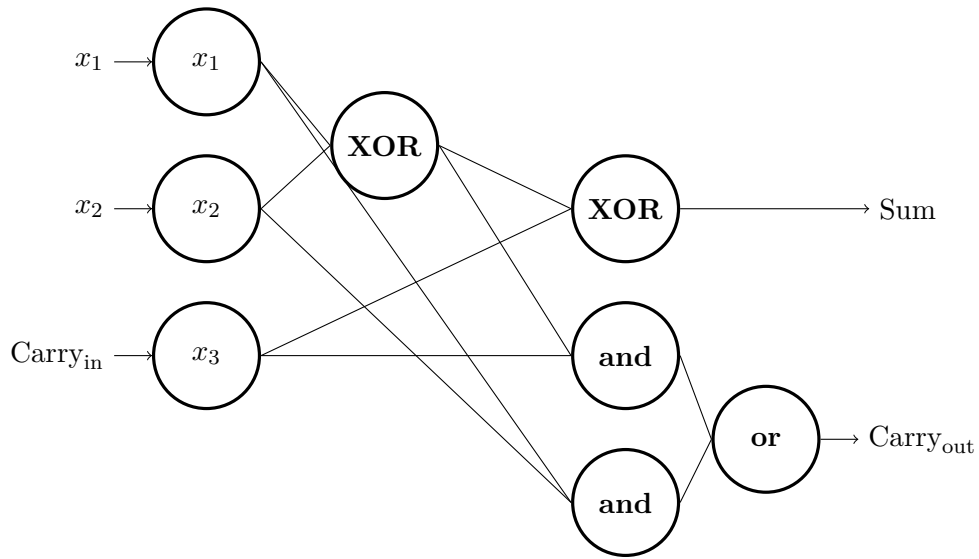

Figure A.2: A network replicating the behavior of a binary full adder.

B   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

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

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

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

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

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

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

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

For $\varphi \in [0, \pi]$ this function is defined everywhere but $\varphi = \frac{\pi}{2}$.
To get a loss as defined in (3.8) 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

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

with

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

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

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

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

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

86

and

$$p_2(z) := a_1(z - k + 1)^3 + a_2(z - k + 1)^2 + \Delta_1(z - k + 1) + l \qquad \text{(B.12)}$$

one gets

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

and

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

The complete loss is than given by

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

For this work, we choose values $a = 4, b = 8$ and $k = 2.2$.
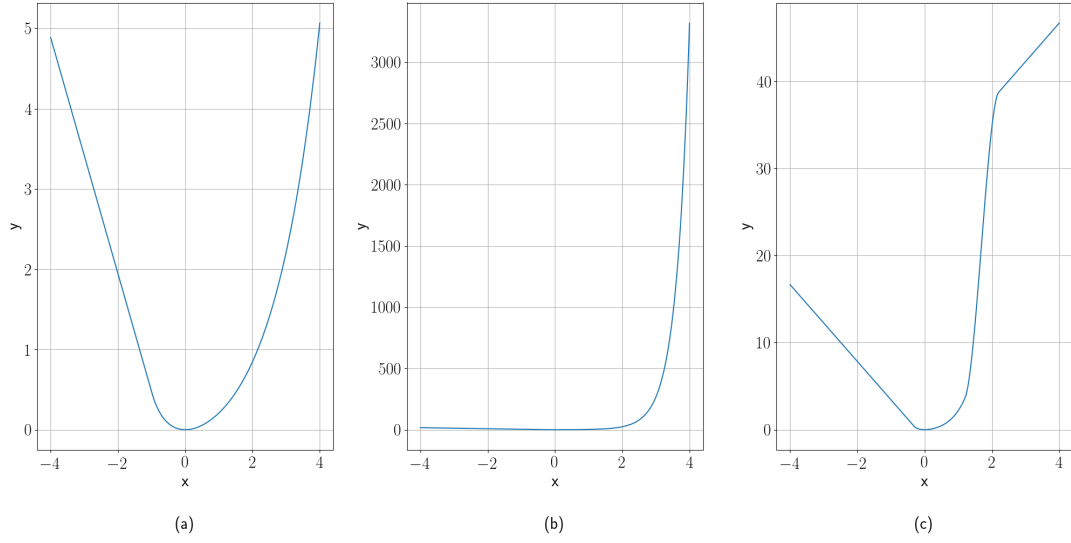
Figure B.1: Three different stages of the custom loss function. (a): The loss as defined in (B.9). For $|x| < 2$ the eyponential part is actually smaller than the linear part. This is not desirable, as for most of our testing the label for pure noise is about 4 away from the smallest label for a signal. (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.

C   Illustrations Final Architecture
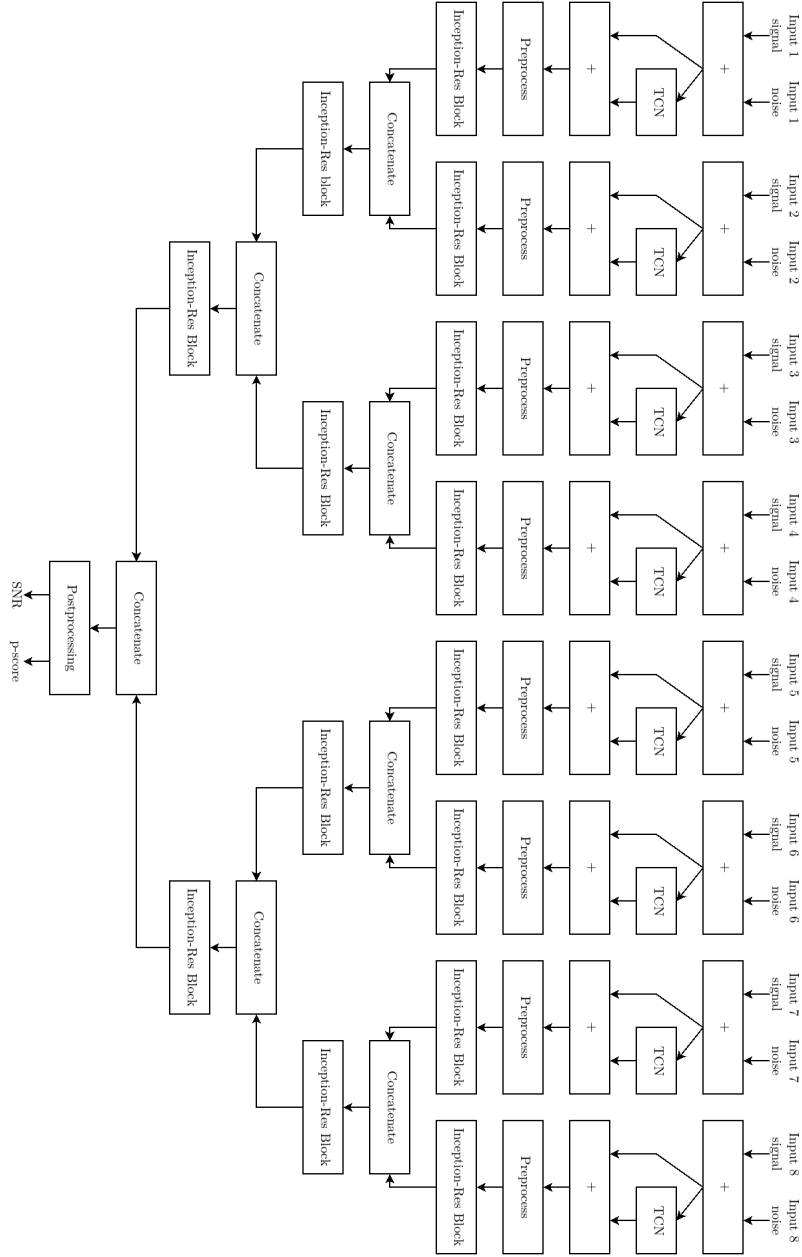
Figure C.1: Shown is a high level overview of the final neural network architecture. The illustration does not show the auxilliary outputs after every TCN and after all but the last concatenation layer. The first set of auxilliary ouputs are simply the outputs of the TCN and trained for the pure waveform. The second set of auxilliary outputs takes the output of the concatenation layers and feeds them through an average pooling and a dimensional reduction layer, before condensing them down to a single number. This number is trained to be the SNR label. The reason or these auxilliary outputs is to combat the vanishing gradient problem.
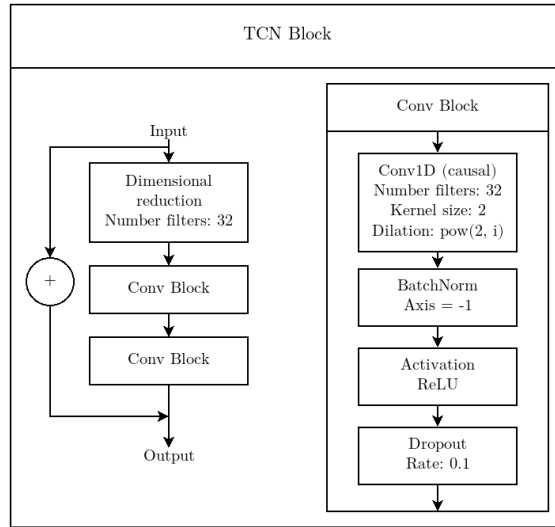
Figure C.2: Shown is a single TCN-block at depth i. A complete TCN stacks multiple of these on top of each other.
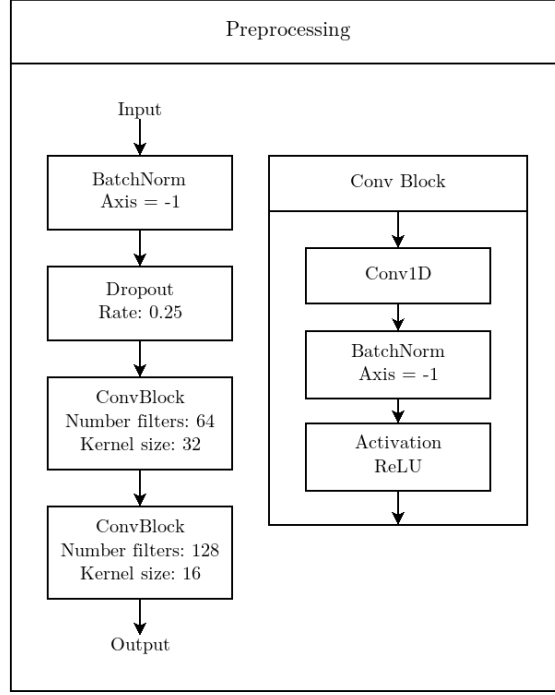
Figure C.3: Shown are the specific layers of the preprocessing step in the final network. A ConvBlock with a specified kernel size and number of filters is understood to have a convolution layer with these properties.
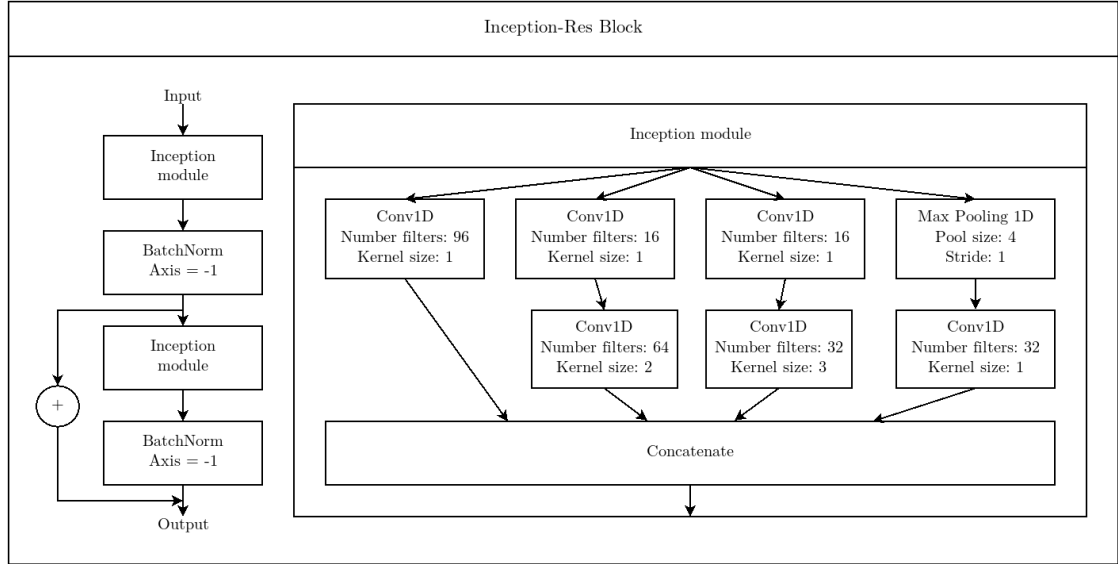
Figure C.4: Shown is an inception block of the final architecture. Each one contains two inception modules, which are depicted to the right. Each layer of the inception module is equipped with a ReLU activation function and all convolution layers pad their input in such a way that the output has the same size as the input.
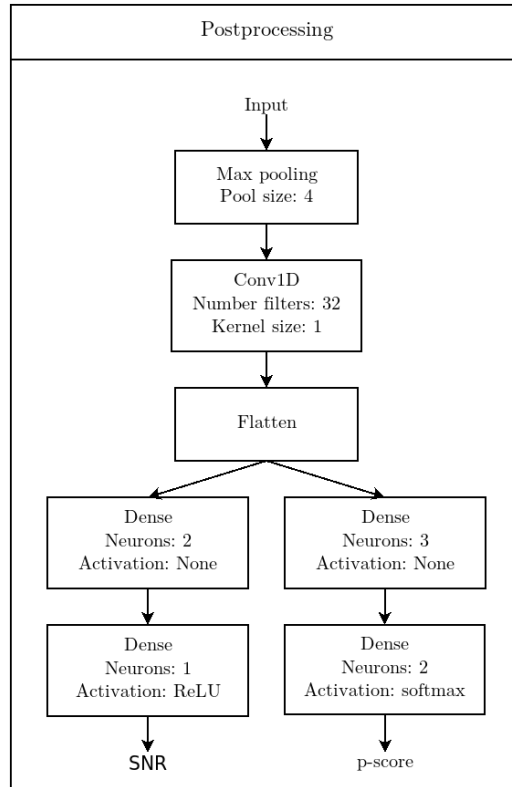
Figure C.5: Shown are the specific layers of the postprocessing step. They are mainly used to reduce the output of the collect-inception network that preceed it to the wanted quantities and their shapes.

Abbreviations

ASD Amplitude spectral density.

BBH Binary black hole.

BNS Binary neutron star.

CNN Convolutional neural network.

EM Electromagnetic.

FFN Feed forward (neural) network.

GW Gravitational wave.

ILSVRC ImageNet Large Scale Visual Recognition Challenge.

MSE Mean squared error.

NN Neural network.

PM Post-Minkowskian approximation.

PN Post-Newtonian approximation.

PSD Power spectral density.

pSNR Peak signal-to-noise ratio.

ReLU Rectified linear unit.

RNN Recurrent neural network.

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 pp. 1, 5).

[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]   LIGO Scientific Collaboration and VIRGO Collaboration. *GraceDB — Gravitational-Wave Candidate Event Database*. URL: https://gracedb.ligo.org/superevents/public/O3/ (visited on 09/11/2019) (cit. on p. 1).

[9]   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, 9, 49).

[10]   Benjamin P Abbott et al. "Multi-messenger observations of a binary neutron star merger". In: (2017). arXiv: 1710.05833. URL: https://arxiv.org/abs/1710.05833 (cit. on pp. 1, 45).

[11] LIGO Scientific Collaboration and VIRGO Collaboration. *Online Pipelines*. 2018. URL: https://emfollow.docs.ligo.org/userguide/analysis/searches.html (visited on 09/10/2019) (cit. on pp. 1, 18).

[12] 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, 39, 40).

[13] Aaron van den Oord et al. "Wavenet: A generative model for raw audio". In: *arXiv preprint arXiv:1609.03499* (2016). arXiv: 1609.03499. URL: https://arxiv.org/abs/1609.03499 (cit. on p. 2).

[14] 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).

[15] 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, 23, 45).

[16] François Chollet et al. *Keras*. https://keras.io. 2019 (cit. on p. 2).

[17] 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 p. 2).

[18] 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. 3, 50).

[19] Collin D Capano et al. "GW170817: Stringent constraints on neutron-star radii from multimessenger observations and nuclear theory". In: *arXiv preprint arXiv:1908.10352* (2019). arXiv: 1908.10352. URL: https://arxiv.org/abs/1908.10352 (cit. on p. 5).

[20] Michael Zevin et al. "Constraining formation models of binary black holes with gravitational-wave observations". In: *The Astrophysical Journal* 846.1 (2017), p. 82. URL: https://iopscience.iop.org/article/10.3847/1538-4357/aa8408/meta (cit. on p. 5).

[21] Marlin Schäfer. "Massenbestimmungen kompakter Binärsysteme durch Analyse von Gravitationswellen". Apr. 2018 (cit. on pp. 5, 7).

[22] Michele Maggiore. *Gravitational Waves. Volume 1: Theory and experiments*. Oxford University Press, 2008. ISBN: 978-019-857074-5 (cit. on pp. 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19).

[23] Domenico Giulini. "Einführung in die Allgemeine Relativitätstheorie". Lecture notes. 2017 (cit. on p. 7).

96

[24] Frank Ohme. "Bridging the gap between Post-Newtonian theory and numerical relativity in gravitational-wave data analysis". PhD thesis. Universität Potsdam, Mathematisch-Naturwissenschaftliche Fakultät, July 2012 (cit. on pp. 17, 18).

[25] Alexander H. Nitz et al. "Rapid detection of gravitational waves from compact binary mergers with PyCBC Live". In: *Phys. Rev. D* 98 (2 July 2018), p. 024050. DOI: 10.1103/PhysRevD.98.024050. URL: https://link.aps.org/doi/10.1103/PhysRevD.98.024050 (cit. on pp. 18, 20, 45, 57).

[26] Leo P. Singer and Larry R. Price. "Rapid Bayesian position reconstruction for gravitational-wave transients". In: *Phys. Rev. D* 93 (2 Jan. 2016), p. 024013. DOI: 10.1103/PhysRevD.93.024013. URL: https://link.aps.org/doi/10.1103/PhysRevD.93.024013 (cit. on p. 21).

[27] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259 (cit. on p. 23).

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org (cit. on pp. 23, 25, 26, 27, 28, 31, 33, 39, 42, 44).

[29] N. Mukund et al. "Transient classification in LIGO data using difference boosting neural network". In: *Phys. Rev. D* 95 (10 May 2017), p. 104059. DOI: 10.1103/PhysRevD.95.104059. URL: https://link.aps.org/doi/10.1103/PhysRevD.95.104059 (cit. on p. 23).

[30] 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. 23, 46, 70, 72, 73, 74, 75).

[31] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/index.html (cit. on pp. 23, 27, 33).

[32] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323. URL: http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf (cit. on p. 25).

[33] 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. 33).

[34] 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. 33, 40, 57, 67, 68).

[35] 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. 33, 61).

[36] 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. 35).

[37] 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. 39).

[38] 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. 39).

[39] 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. 39).

[40] 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. 39).

[41] 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. 39).

[42] 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. 41).

[43] 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. 41).

[44] 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. 41, 43, 59, 67).

[45] 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. 42).

[46] 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. 43).

[47] 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. 44).

[48] B P Abbott et al. "Characterization of transient noise in Advanced LIGO relevant to gravitational wave signal GW150914". In: *Classical and Quantum Gravity* 33.13 (June 2016), p. 134001. DOI: 10.1088/0264-9381/33/13/134001. URL: https://doi.org/10.1088%2F0264-9381%2F33%2F13%2F134001 (cit. on p. 45).

[49] 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. 45, 56).

[50] 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. 45).

[51] 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. 46, 60, 69, 70, 71).

[52] 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. 47, 59).

[53]   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. 47, 60).

[54]   LIGO Scientific Collaboration. *LIGO Algorithm Library - LALSuite*. free software (GPL). 2018. DOI: 10.7935/GT1W-FZ16 (cit. on pp. 50, 51, 69).

[55]   Mirco Ravanelli and Yoshua Bengio. "Interpretable convolutional filters with Sinc-Net". In: *arXiv preprint arXiv:1811.09725* (2018). arXiv: 1811.09725. URL: https://arxiv.org/abs/1811.09725 (cit. on p. 65).

[56]   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. 65).