Early warning for gravitational wave signals from binary neutron star coalescence using field programmable gate arrays

Master's Thesis

submitted in conformity with the requirements for the degree of

Master of Science in Physics (MSc)

Master's degree programme **Experimental Physics**

45 EC

Supervisor: Prof. dr. Chris Van Den Broeck

Second Examiner: Dr. Marc van der Sluys

Daily Supervisors: Melissa López, Quirijn Meijer

Submitted by: Ana Isabel Silva Martins

June 2024

"It is better to ask for forgiveness than to ask for permission."

– Melissa López, my daily supervisor, about cluster usage

Abstract

The present thesis is a proof-of-concept study for the possibility for a machine learning-based pipeline to run on an field programmable gate array in order to detect gravitational waves emitted by the early inspiral of binary neutron star mergers. It explores two main neural network models, GregNet and GWaveNet, and their performance at different parts of the inspiral phase of binary neutron star mergers. At a false alarm probability of 1%, they achieve accuracies of 66.81% and 76.22% respectively. It is found that GregNet is 1.5 times faster at inference than GWaveNet on the graphical processing unit and 5.6 times on the central processing unit. Still, the models would be very similar in cost when run as part of a pipeline. The models are successfully adapted to run on the field programmable gate array and quantized. GregNet is compiled to run on the field programmable gate array. The models in general run the fastest in the graphical processing unit. However, the graphical processing unit is also the least energy-efficient and most costly, while the field programmable gate array is the least costly.

In Chapter 1, we motivate the need for this study. In Chapter 2, we go over the formalism of gravitational waves and the functioning of the detectors; we explore related work and set some research aims. In Chapter 3, the process behind the study is explained, from describing and the treatment of the time series data we work with, to creating the neural networks and embedding a field programmable gate array with it. In Chapter 4, we present the results of the study, including the training of the neural networks, their performance, how long and how much energy they take to test and what making these models part of a pipeline would mean in terms of costs. In Chapter 5, we present the main results of the study and our outlook on future work. Appendix D contains a comprehensive manual for how to work with the Kria KV260, including all of the details, from the start-up to how to quantize and compile personalized neural network models. Additionally, a GitHub repository is provided, with all of the code and supporting documents used to generate the results for this study [1].

The author would like to firstly thank her supervisors, Prof. dr. Chris van den Broeck, Dr. Marc van der Sluys, Melissa López and Quirijn Meijer for their continuous support and valuable feedback, with a special thank you to Melissa for not only giving me academic, but also emotional support.

Thank you to Nikhef for lending me the necessary computing resources for this study and thank you especially to the people behind the Stoomboot cluster for lending me a helping hand multiple times, in particular Dennis van Dok, Mary Hester and Roel Aaij. This project would not have been possible without you.

Thank you to my cohort for never letting me eat lunch alone and for always pushing me to be better.

Lastly, thank you to my family and friends for never complaining about my worsening humour as the deadline for this thesis got closer and closer and for their permanent encouragement all the way from the other side of Europe. I could not have made it without you.

Dedicated to my parents.

Dedicado aos meus pais.

Contents

1 Preface **1**

2 Introduction **3**

2.1 Einstein's theory of general relativity: The formalism of gravitational waves 3

2.1.1 The metric tensor 3

2.1.2 Einstein field equations 3

2.1.3 Linearized general relativity 4

2.1.4 Constraints on gravitational waves 6

2.1.5 Solution for gravitational waves: The stretching and squeezing of space 7

2.2 Gravitational waves: Detectors and sources 8

2.2.1 Sources of gravitational waves 8

2.2.1.1 On neutron stars 10

2.2.2 Gravitational wave detectors 12

2.2.2.1 Interferometer response 14

2.2.3 A detection pipeline 16

2.2.3.1 State of the art: Matched filtering 16

2.2.3.1.1 Foundations 16

2.2.3.1.2 Evidence for gravitational waves through matched filtering pipelines 18

2.2.3.1.3 Challenges 19

2.2.3.2 Towards multimessenger astrophysics 20

2.2.3.2.1 Deep learning 21

2.2.3.2.2 Field programmable gate arrays 22

2.3 Related work 24

2.4 Research aims: Low latency neural network deployment 24

3 Methodology 26

3.1 Working with time series data from binary neutron star coalescence gravitational wave signals 26

3.1.1 Description of data 26

3.1.1.1 O3 Gaussian data 31

3.1.2 Data preparation 31

3.1.3 Data storing and loading 32

3.2 Creating a neural network to distinguish BNS coalescence GW signals from noise 32

3.2.1 Reference neural network: GregNet 33

3.2.1.1 Batch normalization 33

3.2.1.2 One-dimensional convolution 33

3.2.1.3 Activation functions 35

3.2.1.4 Maximum pooling 35

3.2.1.5 Flattening 35

3.2.1.6 Linear layer 36

3.2.1.7 Overview of the architecture 36

3.2.2 GWaveNet 39

3.2.2.1 Gate activation 39

3.2.2.2 1x1 convolutions 40

3.2.2.3 Skip-connections 41

3.2.2.4 Dilated causal convolutions 42

3.2.2.5 Overview of architecture 43

3.2.3 Hyper-parameter tuning 46

3.2.4 Training features 46

3.2.4.1 Curriculum Learning 46

3.2.4.2 Adaptive learning rate 47

3.2.4.3 Early stopping 47

3.3 Embedding a field programmable gate array with a neural network 48

3.3.1 Specifications of the field programmable gate array 48

3.3.1.1 Comparison with state-of-the-art hardware 50

3.3.2 Inspecting model compatibility 50

3.3.3 Post-training quantization 55

4 Results **57**

4.1 Training results 57

4.2 Performance results 60

4.3 Timing results 65

4.4 Energy consumption 67

4.5 How much does this cost? 69

5 Conclusions **70**

5.1 Future work 71

Appendices **72**

A Model details **73**

A.1 GregNet 73

A.2 GWaveNet 73

B Optimizer algorithms **76**

B.1 AdaMax 76

B.2 AdamW 76

C Code snippets **78**

C.1 Dilated causal convolutions 78

C.2 ModuleWavenet 78

D Manual for the Kria KV260 **80**

D.1 Versions 80

D.2 Start-up 80

D.3 Preparing the neural network for the field programmable gate array: Vitis AI 83

D.3.1 Preparation 83

D.3.2 Docker 83

D.3.2.1 If you have access to a GPU 84

D.3.2.2 If you do not have access to a GPU 85

D.3.2.3 Running the Docker container 85

D.3.3 Conda environment 86

D.3.4 Model inspector 86

D.3.5 Quantizer 87

D.3.6 Compiler 88
D.4 Model deployment 89

Bibliography **90**

Acronyms

1D one-dimensional

2D two-dimensional

AdaMax adaptive moment estimation with maximum

AI artificial intelligence

AMD Advanced Micro Devices

ANN artificial neural network

APU application processing unit

aLIGO advanced LIGO

BBH binary black hole

BH black hole

BNS binary neutron star

CBC compact binary coalescence

CE cosmic explorer

CNN convolutional neural network

CPU central processing unit

CUDA compute unified device architecture

DP DisplayPort

DNN deep neural network

DL deep learning

DPU deep learning/data processing unit

EOS equation of state

ET Einstein Telescope

FAP false alarm probability

FLOP floating point operation

FN false negative

FP false positive

FPGA field programmable gate array

GAN generative adversarial network

GNN graph neural network

GPU graphical processing unit

GR general relativity

GRB gamma-ray burst

GW gravitational wave

IP intellectual property

KAGRA Kamioka Gravitational Wave Detector

HDL hardware description language

LIGO Laser Interferometer Gravitational-Wave Observatory

LISA Laser Interferometer Space Antenna

LR learning rate

LSTM Long Short-Term Memory

LTS long term support

LUT look-up table

MMA multi-messenger astrophysics

ML machine learning

NN neural network

NS neutron star

NSBH neutron star-black hole binary

O3 observing run 3

O4 observing run 4

OS operating system

PE processing engine

PINN physics-informed neural network

PISNR partial inspiral signal-to-noise ratio

PL programmable logic

PLD programmable logic device

PS processing system

PSD power spectral density

PTQ post-training quantization

QAT quantization-aware training

RAM random access memory

ReLU rectified linear unit

RNN residual neural network

ROC receiver operating characteristic

sGRB short gamma-ray burst

SNR signal-to-noise ratio

SoC system-on-a-chip

SOM system on module

SSH secure shell protocol

TAP true alarm probability

TN true negative

TP true positive

UU Utrecht University

Chapter 1

Preface

In recent years, Astrophysics and adjacent sciences have been expanding at an astronomical pace. A key reason for this growth is the constant detection of gravitational waves (GWs).

The first GW detection was made on 14 September 2015, and it was caused by the merger of two black holes (BHs). Soon after, on 17 August 2017, we detected the first merger of two neutron stars (NSs). This detection was historic and paved the way for a new field: multi-messenger astrophysics (MMA). Having detected this merger with GW detectors allowed for other messengers to take a look at the event and, as a consequence of this, we were able to link binary neutron star (BNS) mergers to short gamma-ray bursts (sGRBs), to further constrain the equation of state (EOS) for the matter inside a NS and study the tidal effects acting on the two coalescing bodies, to make a new independent measurement of the Hubble constant, to measure the speed of GWs with great accuracy and to suggest the origin for many heavy metals.

However, the event was very particular, as the merger happened very close to Earth ($\sim$ 40 Mpc away, compared to the earlier detected binary black holes (BBHs), which were $\sim$ 10 times farther) and a similar detection and follow-up has not happened since. As such, we are in need of a fast and accurate detection method for the early stages of BNS mergers, which the present thesis touches on by exploring the possibility of running a machine learning (ML)-based pipeline on an field programmable gate array (FPGA) for low-latency detection, i.e. with minimal delay in processing signals.

The author recognizes the significant benefits these large-scale scientific experiments can bring to society and the world, but she is also conscious of the potential harm they can cause. The manufacturing and use of state-of-the-art hardware are causing substantial damage to the environment and this is only predicted to increase, with the manufacturing of computer hardware predicted to take up more than 20% of global energy usage by 2030 [2] and computation energy predicted to hit the world's energy production capacity by 2040 [3].

As such, this study was made with that in mind. The current search pipelines take too much time and computing power. Their continued use will be unfeasible, especially once we enter the era of third-generation detectors. Additionally, the hardware that is currently used is part of the significant problem that is techno-

logical waste. The turnover for central processing units (CPUs) and graphical processing units (GPUs) is even smaller than their lifetimes, with the technology becoming outdated in a matter of a couple of years [4] and there being a constant need to produce a new generation of hardware. Also, the energy needed for the computations, of course, only adds to the problem.

The detection method, based on deep learning (DL), proposed in this study takes significantly less computing resources than the current state of the art. Furthermore, it proposes the use of hardware that is much more environmentally-friendly, production-wise, as it lasts longer than the current state-of-the-art hardware, and computationally-wise, as it has low power consumption and is efficient at processing tasks: the FPGA.

Chapter 2

Introduction

2.1 Einstein's theory of general relativity: The formalism of gravitational waves

GWs are an astronomical phenomenon that were first introduced by Oliver Heaviside in 1893 [5] and later by Henri Poincaré [6] in 1905 as the gravitational equivalent of electromagnetic waves. They were first mathematically demonstrated by Albert Einstein [7, 8] as a consequence of his Theory of general relativity (GR) [9]. In this Section, we dive into Einstein's theory of GR and its solution for GWs.

2.1.1 The metric tensor

The metric tensor encodes the information needed to describe the geometry of spacetime, including its curvature and the effects of gravity. Spacetime is the model GR is based on; it fuses the three known dimensions of space with the one known dimension of time in one continuum space. It can be written as the flat Minkowski metric of special relativity, $\eta_{\mu\nu}$, with, in the weak field limit, a small perturbation due to weak gravitational fields, $h_{\mu\nu}$, where $|h_{\mu\nu}| \ll 1$:

$$g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu}. \tag{2.1}$$

2.1.2 Einstein field equations

Einstein first proposed the Einstein field equations in 1915 [10], and they relate matter with spacetime deformity. They are given by:

$$G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}, \tag{2.2}$$

where G and c are the gravitational constant and the velocity of light in vacuum, and $T_{\mu\nu}$ is the energy-momentum tensor, whereas the left-hand side of the equation is the Einstein tensor, given by

$$G_{\mu\nu} \equiv R_{\mu\nu} - \frac{1}{2} R g_{\mu\nu}, \tag{2.3}$$

where $R_{\mu\nu}$ and R are, respectively, the Ricci tensor and the Ricci scalar, which are defined from the Riemann tensor, $R^{\alpha}_{\mu\beta\nu}$:

$$R^{\alpha}_{\mu\beta\nu} = \partial_\beta \Gamma^{\alpha}_{\nu\mu} - \partial_\nu \Gamma^{\alpha}_{\beta\mu} + \Gamma^{\alpha}_{\beta\rho}\Gamma^{\rho}_{\nu\mu} - \Gamma^{\alpha}_{\nu\rho}\Gamma^{\rho}_{\beta\mu}, \tag{2.4}$$

where $\Gamma^{\alpha}_{\mu\nu}$ are called the Christoffel symbols, defined as:

$$\Gamma^{\alpha}_{\mu\nu} = \frac{1}{2} g^{\beta\alpha}(\partial_\nu g_{\mu\beta} + \partial_\mu g_{\nu\beta} - \partial_\beta g_{\mu\nu}). \tag{2.5}$$

The Ricci tensor follows from taking the trace of the Riemann tensor, i.e., when there is a repeated index on the Riemann tensor, such that $R_{\mu\nu} \equiv R^{\sigma}{}_{\mu\sigma\nu}$, and, in turn, the Ricci scalar is defined as $R \equiv R^{\alpha}{}_{\alpha}$.

2.1.3 Linearized general relativity

In order to describe how GWs behave, it is useful first to linearize the Einstein field equations. To do this, the metric tensor, $g_{\mu\nu}$ should be filled in with its expanded version (Equation (2.1)) and, since $h_{\mu\nu}$ is only a small perturbation, only linear terms with it and its derivatives should be kept. Starting with the linearized Christoffel symbols, we use that $\partial_\alpha \eta_{\mu\nu} = 0$ and neglect terms with higher orders of $h_{\mu\nu}$:

$$\begin{aligned}
\Gamma^{\alpha}_{\mu\nu} &= \frac{1}{2} g^{\beta\alpha}(\partial_\nu g_{\mu\beta} + \partial_\mu g_{\nu\beta} - \partial_\beta g_{\mu\nu}) \\
&= \frac{1}{2}(\eta^{\beta\alpha} + h^{\beta\alpha})(\partial_\nu(\eta_{\mu\beta} + h_{\mu\beta}) + \partial_\mu(\eta_{\nu\beta} + h_{\nu\beta}) - \partial_\beta(\eta_{\mu\nu} + h_{\mu\nu})) \\
&= \frac{1}{2}\eta^{\beta\alpha}(\partial h_{\mu\beta} + \partial_\mu h_{\nu\beta} - \partial_\beta h_{\mu\nu}).
\end{aligned} \tag{2.6}$$

With this, we now turn to the linearized version of the Riemann tensor by using this definition and neglecting the higher-order terms of $h_{\mu\nu}$ once again, which means we can neglect the last two terms of the original expression. We start from our definition in (2.4):

$$\begin{aligned}
R^{\alpha}_{\mu\beta\nu} &= \partial_\beta \Gamma^{\alpha}_{\nu\mu} - \partial_\nu \Gamma^{\alpha}_{\beta\mu} + \Gamma^{\alpha}_{\beta\rho}\Gamma^{\rho}_{\nu\mu} - \Gamma^{\alpha}_{\nu\rho}\Gamma^{\rho}_{\beta\mu} \\
&= \frac{1}{2}\left[\partial_\beta \eta^{\rho\alpha}(\partial_\mu h_{\nu\rho} + \partial_\nu h_{\mu\rho} - \partial_\rho h_{\nu\mu}) - \partial_\nu \eta^{\sigma\alpha}(\partial_\mu h_{\beta\sigma} + \partial_\beta h_{\mu\sigma} - \partial_\sigma h_{\beta\mu})\right] \\
&= \frac{1}{2}[\eta^{\rho\alpha}(\partial_\beta\partial_\mu h_{\nu\rho} + \partial_\beta\partial_\nu h_{\mu\rho} - \partial_\beta\partial_\rho h_{\nu\mu}) - \eta^{\sigma\alpha}(\partial_\nu\partial_\mu h_{\beta\sigma} + \partial_\nu\partial_\beta h_{\mu\sigma} - \partial_\nu\partial_\sigma h_{\beta\mu}].
\end{aligned} \tag{2.7}$$

In order to compress this expression, we can multiply the equation by the metric on the left:

$$\begin{aligned}
R_{\tau\mu\beta\nu} &= g_{\tau\alpha} R^{\alpha}_{\mu\beta\nu} \\
&= \frac{1}{2}[\delta^{\rho}_{\tau}(\partial_\beta\partial_\mu h_{\nu\rho} + \partial_\beta\partial_\nu h_{\mu\rho} - \partial_\beta\partial_\rho h_{\nu\mu}) - \delta^{\sigma}_{\tau}(\partial_\nu\partial_m u h_{\beta\sigma} + \partial_\nu\partial_\beta h_{\mu\sigma} - \partial_\nu\partial_\sigma h_{\beta\mu})] \\
&= \frac{1}{2}(\partial_\beta\partial_\mu h_{\nu\tau} + \partial_\beta\partial_\nu h_{\mu\tau} - \partial_\beta\partial_\tau h_{\nu\mu} - \partial_\nu\partial_\mu h_{\beta\tau} - \partial_\nu\partial_\beta h_{\mu\tau} + \partial_\nu\partial_\tau h_{\beta\mu}) \\
&= \frac{1}{2}(\partial_\beta\partial_\mu h_{\nu\tau} - \partial_\beta\partial_\tau h_{\nu\mu} - \partial_\nu\partial_\mu h_{\beta\tau} + \partial_\nu\partial_\tau h_{\beta\mu}).
\end{aligned} \tag{2.8}$$

From this, we get that the linearized Ricci tensor is given by:

$$R_{\mu\nu} = g^{\beta\tau}T_{\tau\mu\beta\nu} = \frac{1}{2}(\partial^\tau\partial_\mu h_{\nu\tau} - \partial^\tau\partial_\tau h_{\nu\mu} - \partial_\nu\partial_\mu h^\tau_\tau + \partial_\nu\partial_\tau h^\tau_\mu). \tag{2.9}$$

h^τ_τ is simply the trace of $h_{\mu\nu}$, h, and $\partial^\tau\partial_\tau$ can be defined as the d'Alembertian operator, $\Box$. Rewriting:

$$R_{\mu\nu} = \frac{1}{2}(\partial^\tau\partial_\mu h_{\nu\tau} - \Box h_{\nu\mu} - \partial_\nu\partial_\mu h + \partial_\nu\partial_\tau h^\tau_\mu). \tag{2.10}$$

Moreover, by consequence, the linearized Ricci scalar can be written as:

$$\begin{aligned} R = R^\mu_\mu = g^{\mu\nu}R_{\mu\nu} &= \frac{1}{2}(\partial^\tau\partial_\mu h^\mu_\tau - \Box h^\mu_\mu - \partial^\mu\partial_\mu h + \partial^\mu\partial_\tau h^\tau_\mu) \\ &= \partial^\tau\partial_\mu h^\mu_\tau - \Box h. \end{aligned} \tag{2.11}$$

Finally, the linearized Einstein tensor is given by substitung the linearized identities we found into Equation (2.3):

$$\begin{aligned} G_{\mu\nu} &= R_{\mu\nu} - \frac{1}{2}Rg_{\mu\nu} \\ &= \frac{1}{2}(\partial^\tau\partial_\mu h_{\nu\tau} - \Box h_{\nu\mu} - \partial_\nu\partial_\mu h + \partial_\nu\partial_\tau h^\tau_\mu) - \frac{1}{2}\eta_{\mu\nu}(\partial^\rho\partial_\sigma h^\sigma_\rho - \Box h). \end{aligned} \tag{2.12}$$

However, the usual form of the linearized Einstein field equations is in terms of a new tensor $\overline{h}_{\mu\nu} \equiv h_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu}h$, for simplicity of notation. By switching out $h_{\mu\nu} = \overline{h}_{\mu\nu} + \frac{1}{2}\eta_{\mu\nu}h$ into equation (2.12), we can get to a simpler version of the linearized Einstein tensor and, in turn of the linearized Einstein field equations:

$$\begin{aligned} G_{\mu\nu} &= \frac{1}{2}\left[\partial^\tau\partial_\mu\left(\overline{h}_{\mu\tau} + \frac{1}{2}\eta_{\nu\tau}h\right) - \Box\left(\overline{h}_{\mu\nu} + \frac{1}{2}\eta_{\mu\nu}h\right) - \partial_\nu\partial_\mu h + \partial_\nu\partial_\tau\left(\overline{h}^\tau_\mu + \frac{1}{2}\delta^\tau_\mu h\right)\right] \\ &- \frac{1}{2}\eta_{\mu\nu}\left[\partial^\rho\partial_\sigma\left(\overline{h}^\sigma_\rho + \frac{1}{2}\eta^\sigma_\rho h\right) - \Box h\right] \\ &= \frac{1}{2}\left(\partial^\tau\partial_\mu\overline{h}_{\nu\tau} + \frac{1}{2}\partial_\nu\partial_\mu h - \Box\overline{h}_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu}\Box h - \partial_\nu\partial_\mu h + \partial_\nu\partial_\tau\overline{h}^\tau_\mu + \frac{1}{2}\partial_\nu\partial_\mu h\right. \\ &\left. - \eta_{\mu\nu}\partial^\rho\partial_\sigma\overline{h}^\sigma_\rho - \frac{1}{2}\eta_{\mu\nu}\Box h + \eta_{\mu\nu}\Box h\right) \\ &= \frac{1}{2}(\partial^\tau\partial_\mu\overline{h}_{\nu\tau} - \Box\overline{h}_{\mu\nu} + \partial_\nu\partial_\tau\overline{h}^\tau_\mu - \eta_{\mu\nu}\partial^\rho\partial_\sigma\overline{h}^\sigma_\rho), \end{aligned} \tag{2.13}$$

$$\Box\overline{h}_{\mu\nu} + \eta_{\mu\nu}\partial^\alpha\partial^\beta\overline{h}_{\beta\alpha} - \partial^\alpha\partial_\nu\overline{h}_{\mu\beta} - \partial^\alpha\partial_\mu\overline{h}_{\alpha\nu} = -\frac{16\pi G}{c^4}T_{\mu\nu}. \tag{2.14}$$

In order to further simplify this, we consider a coordinate transformation to our metric that will not spoil its form by constraining ourselves to small changes in coordinates: $x'^\mu = x^\mu + \xi^\mu$. Under this transformation,

it can be shown that:

$$g'_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu} - \partial_\mu \xi_\nu - \partial_\nu \xi_\mu, \tag{2.15}$$

where we identify:

$$h'_{\mu\nu} = h_{\mu\nu} - (\partial_\mu \xi_\nu + \partial_\nu \xi_\mu). \tag{2.16}$$

Because of their close form to electromagnetism, these are called gauge transformations. The Lorentz gauge condition dictates that there always exists a gauge transformation such that:

$$\partial^\mu \bar{h}_{\mu\nu} = 0. \tag{2.17}$$

Under this choice of gauge, Equation (2.14) further simplifies into:

$$\Box \bar{h}_{\mu\nu} = -\frac{16\pi G}{c^4} T_{\mu\nu}. \tag{2.18}$$

Furthermore, away from the source (e.g. a detector of GWs on earth), the energy-momentum tensor tends to zero, and we get:

$$\Box \bar{h}_{\mu\nu} = 0. \tag{2.19}$$

2.1.4 Constraints on gravitational waves

We are interested in knowing how many degrees of freedom GWs have. A spacetime metric is a 4x4 tensor, thus, having 16 components. However, $\bar{h}_{\mu\nu}$ is a symmetric tensor, so the amount of degrees of freedom reduces to 10. In the Lorentz gauge condition (Equation (2.17)), there is a free ν index, thus reducing the number of degrees of freedom by 4.

We can further use gauge transformations, $h'_{\mu\nu} = h_{\mu\nu} - (\partial_\mu \xi_\nu + \partial_\nu \xi_\mu)$, to impose the following constraints on $\bar{h}_{\mu\nu}$:

$$\bar{h}^\mu_\mu = 0, \tag{2.20}$$

$$\bar{h}_{\mu 0} = 0. \tag{2.21}$$

Equation (2.20) indicates that $\bar{h}_{\mu\nu}$ is traceless and so $\bar{h}_{\mu\nu} = h_{\mu\nu}$. With no free indices, this condition reduces the number of degrees of freedom of GWs from 6 to 5. Equation (2.21) lets us drop the time-time and time-space components. This condition has a free μ index but, with the previous contraint from the Lorentz gauge, $\partial^\mu \bar{h}_{\mu 0} = 0$, so it only restricts us 3 degrees further. In the end, we are left with two degrees of freedom.

2.1.5 Solution for gravitational waves: The stretching and squeezing of space

What do these two degrees of freedom look like? Looking at Equation (2.19), we can recognise a wave equation, whose solution for the plane wave can give us an intuition for later reaching a more general form (now using only space components):

$$h_{ij} = A_{ij}(\vec{k})\cos(\omega t - \vec{k}\cdot\vec{x}), \tag{2.22}$$

where $\omega = ck$. From the Lorentz gauge (2.17), we get that $k^i A_{ij} = 0$. Taking this and the traceless condition (2.20), and choosing the direction of propagation along the z-axis, without loss of generality, h_{ij} must be of the form:

$$h_{ij}^{TT} = \begin{pmatrix} h_+ & h_\times & 0 \\ h_\times & -h_+ & 0 \\ 0 & 0 & 0 \end{pmatrix} \cos[\omega(t - z/c)]. \tag{2.23}$$

h_+ and $h_\times$ can be any expression, but we choose this convenient name whose reason will be clarified later. The gauge that leads to this solution is called the transverse-traceless gauge.

The spacetime metric will then take the form:

$$\begin{aligned} g_{\mu\nu} &= \eta_{\mu\nu} + h_{\mu\nu} \\ &= \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 + h_+\cos[\omega(t-z/c)] & h_\times\cos[\omega(t-z/c)] & 0 \\ 0 & h_\times\cos[\omega(t-z/c)] & 1 - h_+\cos[\omega(t-z/c)] & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \end{aligned} \tag{2.24}$$

and $ds^2 = g_{\mu\nu}dx^\mu dx^\nu$ is given by:

$$\begin{aligned} ds^2 = &-c^2dt^2 + (1 + h_+\cos[\omega(t-z/c)])dx^2 + (1 - h_+\cos[\omega(t-z/c)])dy^2 \\ &+ 2h_\times\cos[\omega(t-z/c)]dxdy + dz^2, \end{aligned} \tag{2.25}$$

where ds^2 is known as the line element and it describes the infinitesimal displacement between two events in spacetime.

To make it more explicit what this means physically, it is useful to look at the effects of h_+ and $h_\times$ separately. If $h_+ \neq 0$ and $h_\times = 0$, we get:

$$ds^2 = -c^2dt^2 + (1 + h_+\cos[\omega(t-z/c)])dx^2 + (1 - h_+\cos[\omega(t-z/c)])dy^2 + dz^2, \tag{2.26}$$

which demonstrates that space in the x and y directions is stretched and squeezed periodically. On the other hand, if $h_+ = 0$ and $h_\times \neq 0$, we get:

$$ds^2 = -c^2dt^2 + dx^2 + dy^2 + 2h_\times\cos[\omega(t-z/c)]dxdy + dz^2. \tag{2.27}$$

To get a proper intuition for what this means, it is convenient to change coordinate systems, $(ct, x, y, z) \rightarrow (ct, x', y', z)$, where the prime coordinates are rotated 45° from the original coordinates over the z-axis. Doing this, we get:

$$ds^2 = -c^2dt^2 + (1 + h_\times \cos[\omega(t - z/c)])dx'^2 + (1 - h_\times \cos[\omega(t - z/c)])dy'^2 + dz^2. \tag{2.28}$$

Once again, this demonstrates the periodical stretching and squeezing of space, but this time at a 45° angle to the previously shown polarisation. This effect is shown in Figure 2.1.

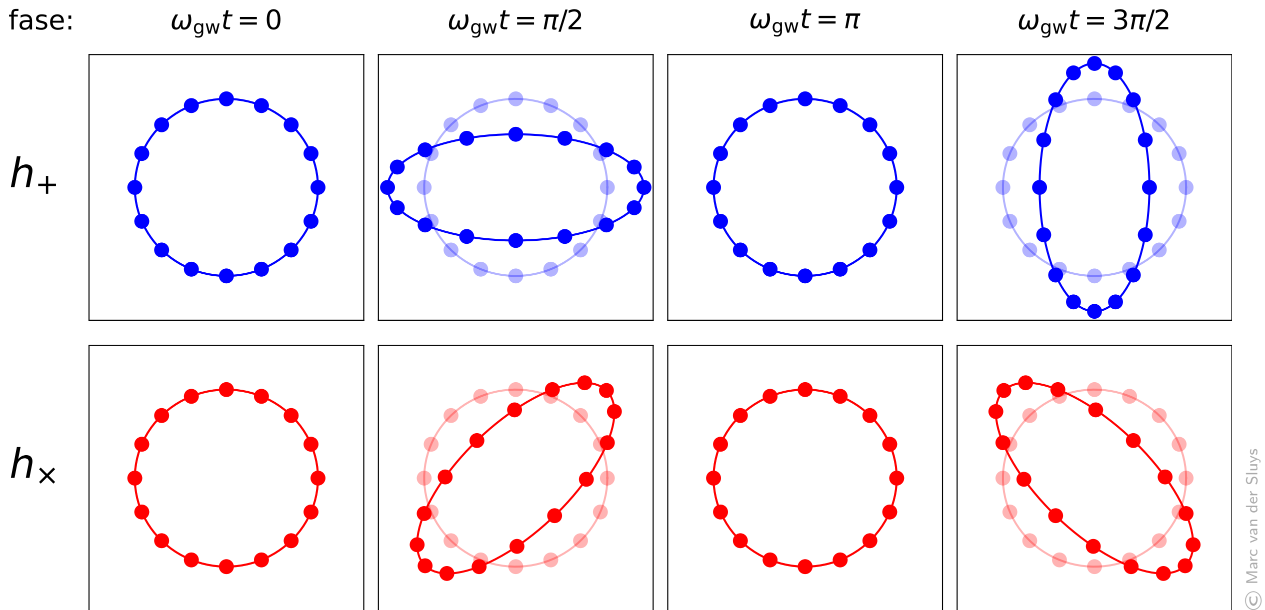


Figure 2.1: Effect of the + and × polarisations of a GW on a ring of test particles. For the + polarization, once space is stretched in the vertical direction, it is also squeezed in the horizontal direction. After half a period, the opposite happens. For the × polarization, when space is stretched at a 45° angle, it is also squeezed at a -45° angle, and vice-versa.

2.2 Gravitational waves: Detectors and sources

In the previous Section, we derived the mathematical formulation of GWs. In this Section, we dive into what the sources of GWs are and which one we will focus on, GW detectors and their functioning, how detections are made from detector data and what our proposal for a new detection method is.

2.2.1 Sources of gravitational waves

GWs can be emitted by several different sources, such as [11, 12]:

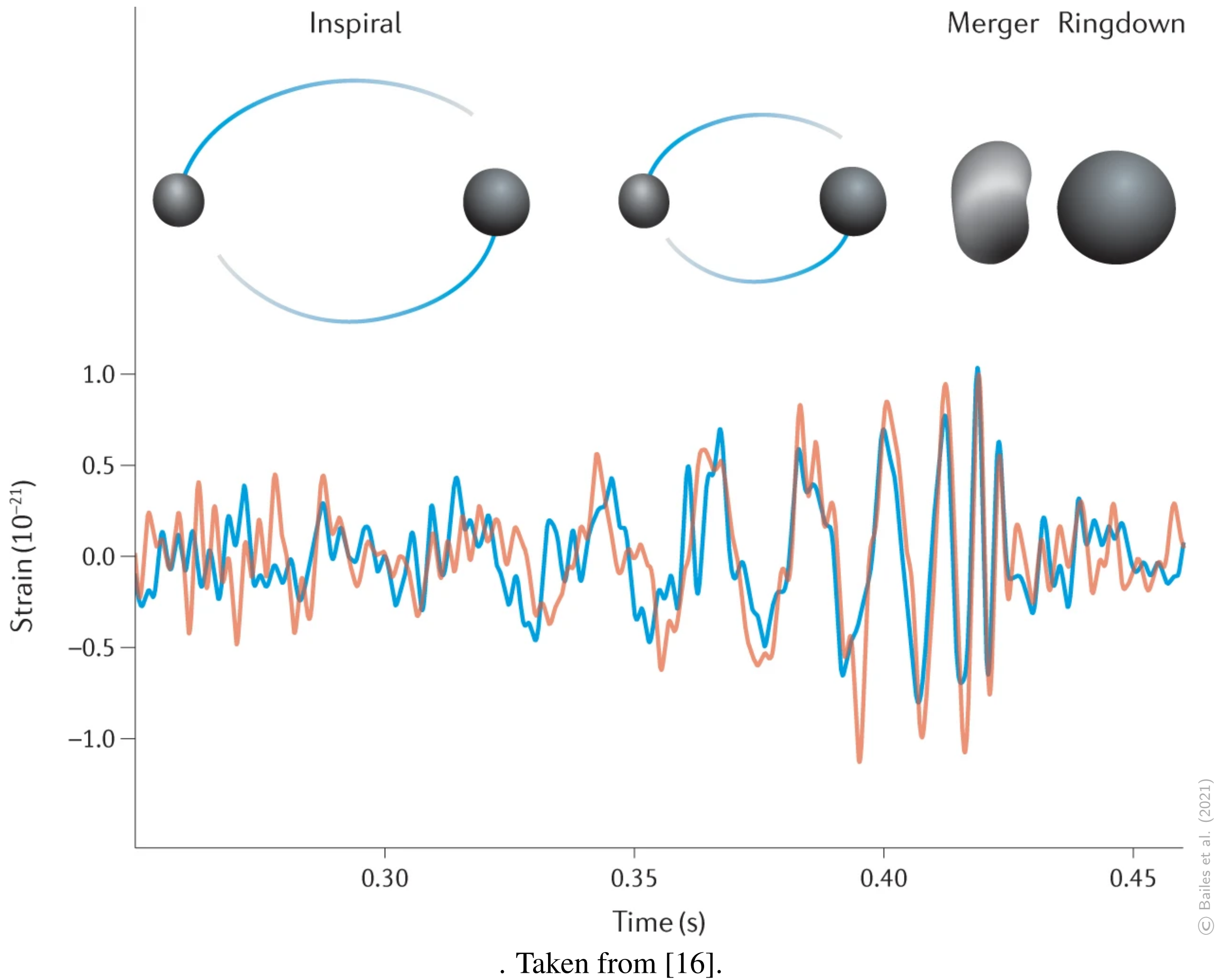
- A merger of two compact astrophysical objects, also referred to as a compact binary coalescence (CBC), such as a BBH, a BNS or a neutron star-black hole binary (NSBH),
- A single fast-spinning massive object, with any bumps on or imperfections in its spherical shape, such as an asymmetric NS,

- A burst, such as a supernova explosion, a magnetar flare or a cosmic string cusp,
- A primordial background of GWs.

The main modelled sources of GWs are CBCs.

A CBC event is made up of two compact astrophysical objects orbiting around each other in a wide orbit, and GWs are emitted, due to a break in symmetry, consequence of the two objects not being identical. The emission of GWs consumes energy, so this orbit will shrink. The objects will orbit closer and closer to each other, at higher and higher frequencies, until their orbit is close to circular, called a quasi-circular inspiral. At around 10 Hz, the GW signals fall in the sensitivity band of our current detectors [13] and they start being able to detect. As the objects orbit around each other, we are in the so-called inspiral phase. The system keeps losing energy until the objects are sufficiently close and then collapse towards each other, merging into a new, highly excited compact object. This part of the CBC event is called the merger. The new object will then de-excite, a process called the ringdown, and settle into a dormant state. A visual representation of a CBC event is shown in Figure 2.2, as is a graph showing the corresponding strain over time as detected by the Laser Interferometer Gravitational-Wave Observatory (LIGO) Hanford and LIGO Livingston detectors [14] for GW150914 [15], the first ever detected GW. [16]

Out of the sources for CBC events, in this study we are interested in BNSs. GW signals from BNSs are particularly interesting for MMA, due to their long inspiral phase and their production of observable signals across multiple wavelengths and messengers. MMA based on BNS mergers is focused not only on observing the binary system and its remnants but also on learning more about the individual components, diving deeper into dense matter physics and the nature of NSs.



. Taken from [16].

Figure 2.2: Representation of a CBC event and the corresponding GW strain over time as detected by LIGO Hanford (red) and LIGO Livingston (blue) for the GW150914 event

2.2.1.1 On neutron stars

NSs are highly compact astrophysical objects, and they appear as the outcome of the collapsing of the core of a massive star [17]. The existence of NSs was predicted shortly after the discovery of the neutron but only confirmed in 1968, when regular radio pulses emitted by PSR B1919+21 were detected [18].

Even though the EOS for nuclear matter under the extreme conditions of a NS has been studied since the 1930s [19] and many restrictions have been set, its final form is still unknown. Figure 2.3 shows many possibilities for EOS, as well as constraints set by GR, causality ($dP/d\rho = c^2_{\text{sound}} \leq c^2$ [20]) and observations. It further shows the masses of observed binary radio pulsars. Detections of GW signals from NSs are vital to constrain the possible tidal effects suffered by coalescing objects. Tidal forces arise from the variation in gravitational attraction in the objects involved in a coalescence, with the side facing the other object experiencing a stronger gravitational pull than the one facing away. Consequently, a tidal bulge is created, deforming the objects. Understanding the extent of this effect and its factors will lead us to further constrain the NS EOS, since knowing how tidal deformability acts will give us the information of mass as a function of radius.

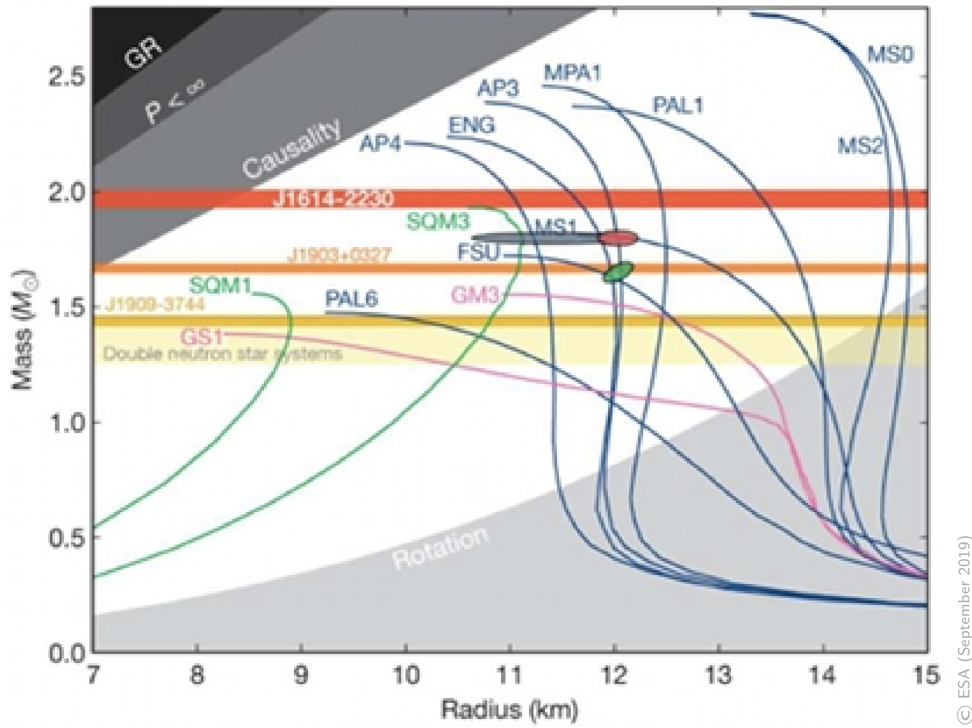


Figure 2.3: Theoretical mass-radius relation formulations for the EOS of the matter inside a NS for the possibilities of only nucleonic matter (blue), nucleonic plus exotic matter (pink), and strange quark matter (green). What each individual formulation means is beyond the scope of this study, and this Figure simply aims to demonstrate the landscape of the current theoretical predictions for the NS EOS. The horizontal lines represent the masses determined for observed binary radio pulsars. NSs with mass-radius in the upper left region are disallowed as a consequence of GR and causality and NSs with mass-radius in the lower right region are limited by the observation of the highest-frequency known pulsar, at 716 Hz. Taken from [21].

There are currently no firm theoretical constraints on the maximum possible mass of a non-rotating NS. However, the most massive NS measured to date, J0740+6620, had the mass of $2.14^{+0.10}_{-0.09}M_\odot$ [22], for a 68.3% credibility interval. The theoretical predictions for the lower bound are more contentious. The minimum predicted mass from GR, nearly independent from the EOS, is $0.1M_\odot$ [23]. However, it is thought that a NS with mass that low would not form since it depends on the mass of the original massive star whose core collapsed, and according to this, a more realistic estimate is closer to $1M_\odot$ [24]. The NS with the smallest mass measured to date was part of the binary pulsar J0453+1559, with a mass of $1.174 \pm 0.004M_\odot$ [25].

A BNS merger is a CBC event where both of the compact objects are NSs. BNS mergers have a long observable quasi-circular inspiral time, in the order of minutes to hours [26], depending on detector sensitivity. We are interested in being able to detect a GW in this inspiral phase, in order to prepare for the merger.

For a long time before current detections, it has been predicted that once the merger happens, a short, hard gamma-ray burst (GRB) is emitted from the system [27], furthering the possibility for MMA and the possibility to learn more about BNS systems in general.

2.2.2 Gravitational wave detectors

Current state-of-the-art GW detectors are called interferometric detectors [28], and they are inspired by the Michelson interferometer [29]. Figure 2.4 shows the layout of a detector. In these, a source laser beam is split in two by a beam-splitting mirror. Each of these two laser beams then travels through a long tube, called the detector arm, and gets reflected by a mirror at the end of it. The light then travels back to the beam-splitting mirror, where it is redirected to a photodiode and read by a light detector. In this basic setup, the laser interferometer is set up so that, under no perturbation, once the beams get to the light detector, they are out of phase by half a wavelength. This causes destructive interference, so no signal is read. If there is a perturbation in space, causing the stretching and squeezing of the length of the arms, then the light beams will not be perfectly out of phase, and there will be a signal. This is due to the beams still travelling at the speed of light and not being affected by the ripples caused by the GW.

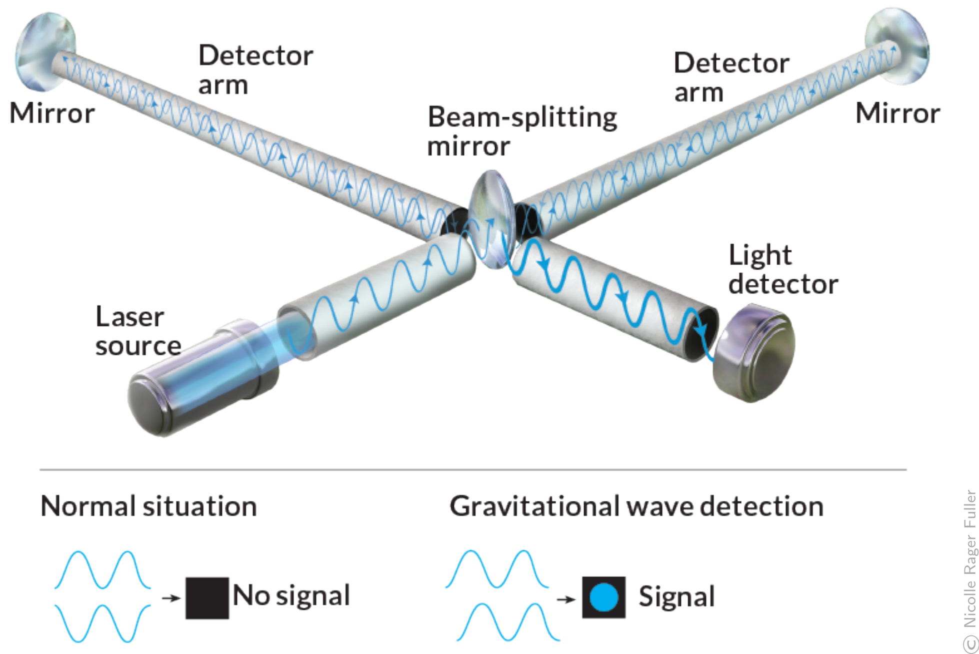


Figure 2.4: Layout of a laser interferometer. Taken from [28].

As seen in Section 2.1, space is perturbed by GWs, which in a detector translates to its arms being stretched and squeezed, changing their length and, hence, the path that light has to travel is different than in the unperturbed case. To determine how much this change in path is, let us momentarily consider a GW with only plus polarisation approaching the detector arms, with possibly perturbed lengths L_x and L_y, and no z component. Equation (2.25) simplifies to:

$$\begin{aligned} ds^2 &= -c^2dt^2 + (1+h_+\cos[\omega(t-z/c)])dx^2 + (1-h_+\cos[\omega(t-z/c)])dy^2 \\ &\quad + 2h_\times\cos[\omega(t-z/c)]dxdy + dz^2 \\ &= -c^2dt^2 + (1+h)dx^2 + (1-h)dy^2. \end{aligned} \tag{2.29}$$

The geodesic for a photon in this light path is null, thus $ds^2 = 0$. Hence, Equation (2.29) gives us the

displacement of each coordinate:

$$dx = \frac{cdt}{\sqrt{1+h}} \approx c\left(1+\frac{h}{2}\right)dt \tag{2.30}$$

$$dy = \frac{cdt}{\sqrt{1-h}} \approx c\left(1-\frac{h}{2}\right)dt, \tag{2.31}$$

where we Taylor expanded over h. Integrating both sides, we get:

$$\int_{x_{\text{beam splitter}}}^{x_{\text{mirror}}} dx = c\Delta t\left(1+\frac{h}{2}\right) \Rightarrow L_x = c\Delta t\left(1+\frac{h}{2}\right), \tag{2.32}$$

$$\int_{y_{\text{beam splitter}}}^{y_{\text{mirror}}} dy = c\Delta t\left(1-\frac{h}{2}\right) \Rightarrow L_y = c\Delta t\left(1-\frac{h}{2}\right). \tag{2.33}$$

The difference between the lengths of the two arms is then:

$$\Delta L = L_x - L_y = c\,\Delta t\,h = Lh, \tag{2.34}$$

where L is the unperturbed length of the arms.

The amplitude of a gravitational wave, h, also called gravitational wave strain for the CBC signals we are interested in studying is of the order of 10^{-21} [30]. We can relate the strain to the detectors using Equation (2.34). For example, for the Virgo detector, for which each detector arm has a length of 3 km [31], we are detecting differences in the length of the arms in the order of 10^{-18} m, around a hundred times smaller than the size of a proton!

Since the effects we are aiming to detect are so small, even LIGO's 4 km detector arms [32] are not enough to detect what we aim to. To solve this, one of the main features that was introduced is called the Fabry-Perot cavities [33]. These consist of introducing two new mirrors, one in each arm, near the beam splitter [32], as shown in Figure 2.5. The laser in each arm gets reflected inside the each cavity about 300 times, increasing the effective distance travelled from 4 km to about 1200 km.

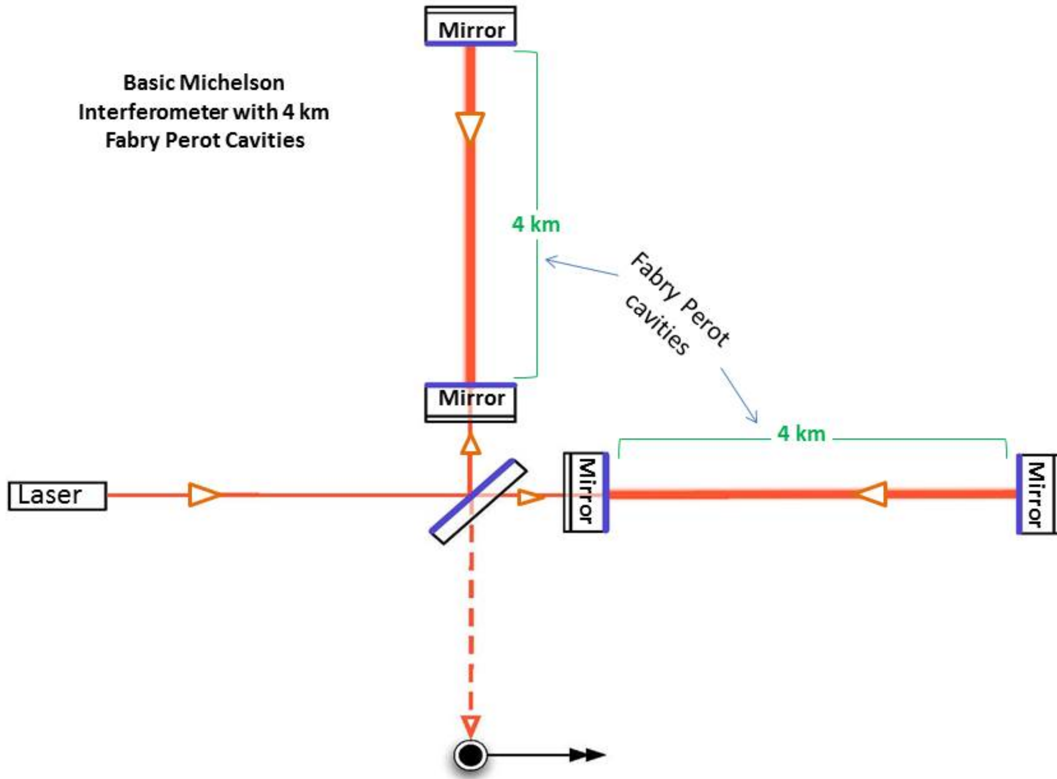


Figure 2.5: Layout of a laser interferometer with Fabry-Perot cavities. Mirrors are inserted near the beam-splitter, forming the Fabry-Perot resonant cavities. Taken from [34].

2.2.2.1 Interferometer response

In order to understand how GWs truly affect a detector, let us think about them from the reference frame of the detector, where the x-axis spans one arm and the y-axis spans the other. As such, we can write the line element as:

$$ds^2 = -c^2\,dt^2 + (\delta_{ij} + h_{ij})dx^i\,dx^j, \tag{2.35}$$

where δ^{ij} is the Kronecker delta and only the space components are considered for the perturbation, as we verified is valid in Section 2.1.4. Since light moves in null geodesics, $ds^2 = 0$, and, when considering what happens to the x-arm, we can set $dy = dz = 0$, and vice-versa for the y-arm, leaving us with:

$$c^2\,dt^2 = (1 + h_{xx})dx^2, \qquad c^2\,dt^2 = (1 + h_{yy})dy^2. \tag{2.36}$$

Similarly to Equations (2.32) and (2.33), we can integrate over the space between the mirror and the beam splitter (ignoring the Febry-Perot cavities for this demonstration) in order to get the time it takes for light to travel between the two, when there is a perturbation:

$$\Delta t_x \approx \left(1 + \frac{1}{2}h_{xx}\right) L/c, \qquad \Delta t_y \approx \left(1 + \frac{1}{2}h_{yy}\right) L/c. \tag{2.37}$$

The output of a detector, called gravitational wave strain, is then given by:

$$h(t) = \frac{\Delta t_x - \Delta t_y}{\Delta t_0} = \frac{1}{2}(h_{xx} - h_{yy}), \tag{2.38}$$

where Δt_0 is the time it takes for light to travel along the unperturbed length of the arms.

We now want to relate the gravitational wave strain to our original + and $\times$ polarizations. For that, we perform a coordinate transformation according to the rotation matrix:

$$\mathbf{R} = \begin{pmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{2.39}$$

with the corresponding angles as shown in Figure 2.6. As such, we perform the following operation:

$$h_{ij} = (\mathbf{R} h'^{TT} \mathbf{R}^T)_{ij}, \tag{2.40}$$

where $h'^{TT}_i j$ is the metric perturbation in the (x', y', z') frame, in its transverse-traceless form, as defined in Section 2.1.5. Finally, using Equation (2.38), we get the strain in terms of the + and $\times$ polarizations:

$$h = \frac{1}{2}(h_{xx} - h_{yy}) = F_+(\theta,\phi,\psi)h_+ + F_\times(\theta,\phi,\psi)h_\times, \tag{2.41}$$

where $F_{+,\times}$ are the beam pattern functions, defined by:

$$F_+(\theta,\phi,\psi) = \frac{1}{2}(1+\cos^2\theta)\cos(2\phi)\cos(2\psi) - \cos\theta\sin(2\phi)\sin(2\psi), \tag{2.42}$$

$$F_\times(\theta,\phi,\psi) = \frac{1}{2}(1+\cos^2\theta)\cos(2\phi)\sin(2\psi) + \cos\theta\sin(2\phi)\cos(2\psi). \tag{2.43}$$

By analysing Equations (2.42) and (2.43), one can easily see that the current two-armed perpendicular detectors have a very obvious fault: they are blind to certain directions. If $\phi = \frac{\pi}{4}$, then $\cos(2\phi) = \cos(\pi/2) = 0$, making both equations equal to zero.

In order to get accurate and useful readings, it is essential to have a network of detectors. A network of at least three detectors allows us to localise the source in the sky, allowing for MMA. Additionally, a network of detectors allows us to reduce the impact of other sources of perturbation, called glitches.

Glitches are short-duration noise artifacts that appear in detector data and can mimic GWs. These can be caused by a number of non-astrophysical sources, such as instrumental artifacts or environmental disturbances [35]. Glitches are classified according to their nature and shape, with the main types being the blip, fast scattering, koi fish, low-frequency burst, tomte, and whistle [36].

GWs are currently detected by pipelines based on Bayesian statistics [37], with help from tools like BayesWave [38]. More recently, neural networks (NNs) [39] have started to enter the scene by acting as triggers for events. This study is a proof-of-concept for the latter.

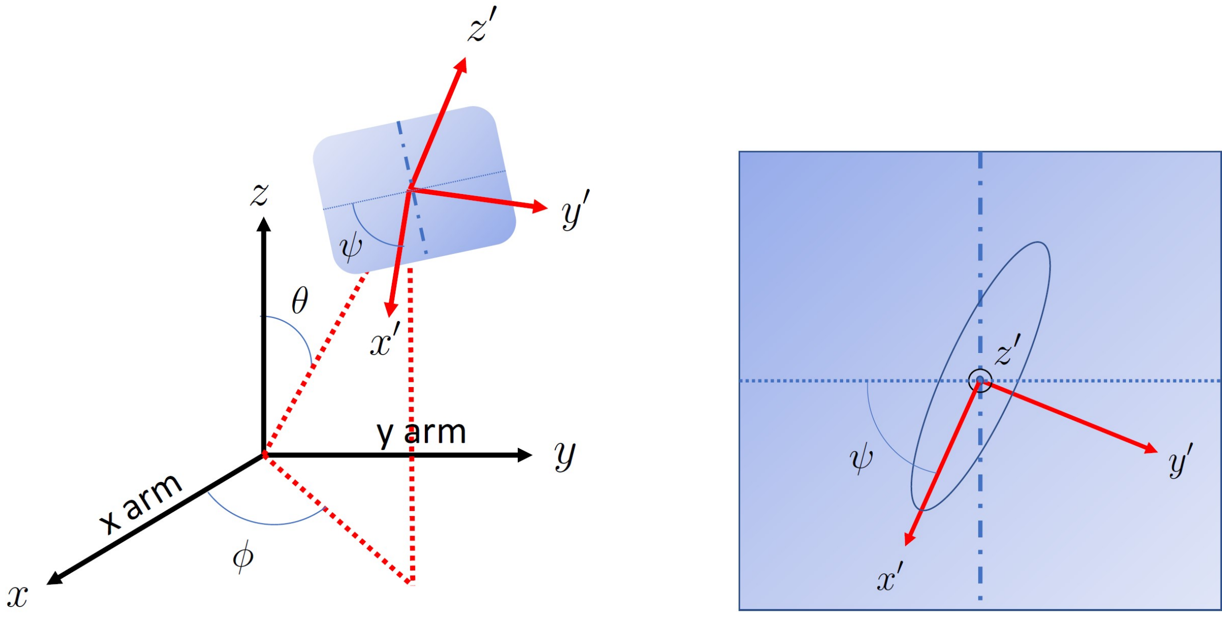


Figure 2.6: Coordinate system where a GW is propagating in the direction z', stretching and squeezing space in the x' and y' directions; it is angled according to θ and ϕ from the detector coordinates, whose arms extend along the x and y directions and are in the plane perpendicular to z.

2.2.3 A detection pipeline

2.2.3.1 State of the art: Matched filtering

2.2.3.1.1 Foundations Take a time series, $s(t)$, such that:

$$s(t) = h(t) + n(t), \tag{2.44}$$

that represents a GW signal $h(t)$ buried in, ideally, stationary Gaussian noise with zero mean $n(t)$. In a real-world setting, the noise is not ideal, in part due to the aforementioned glitches. There are two possible hypothesis for this problem:

Hypothesis 1 (H1): *The null hypothesis, when there is no signal buried in the noise, and* $s(t) = n(t)$.

Hypothesis 2 (H2): *The alternative hypothesis, when there is a signal buried in the noise, and* $s(t) = h(t) + n(t)$ *with* $h(t) \neq 0$.

Matched filtering is a method for modelled searches that aims to find the optimal filter $K(t)$, or corresponding GW template, that is able to identify a waveform in $h(t)$, assuming Hypothesis 2, while filtering out the noise $n(t)$ through maximizing its detection statistic. Let S be the expected value of $\hat{s}$, our observation, when a signal is present ($h(t) \neq 0$) and N be the root-mean-square of $\hat{s}$ when no signal is present ($h(t) = 0$):

$$S = \langle \hat{s} \rangle_h, \qquad N = \langle \hat{s}^2 \rangle_{h=0}^{1/2}. \tag{2.45}$$

We can integrate the signal against our optimal filter $K(t)$ to reach expressions for S and N, where * repre-

sents the complex conjugate and $\tilde{\cdot}$ represents the Fourier transform of the function:

$$
\begin{aligned}
S &= \int_{-\infty}^{\infty} dt \, \langle s \rangle K(t) = \int_{-\infty}^{\infty} dt \, \langle n(t) + h(t) \rangle K(t) = \int_{-\infty}^{\infty} dt \, h(t) K(t) \\
&= \int_{-\infty}^{\infty} df \, \tilde{h}^*(f) \tilde{K}(f), \qquad (2.46)\\
N &= \sqrt{[\langle \hat{s}^2(t) \rangle - \langle \hat{s}(t) \rangle^2]_{h=0}} = \sqrt{\langle \hat{s}^2(t) \rangle} \\
&= \sqrt{\left\langle \int_{-\infty}^{\infty} dt \int_{-\infty}^{\infty} dt' \, n(t) n(t') K(t) K(t') \right\rangle} \\
&= \sqrt{\int_{-\infty}^{\infty} dt \int_{-\infty}^{\infty} dt' \, \langle n(t) n(t') \rangle K(t) K(t')} = \sqrt{\int_{-\infty}^{\infty} df \, \frac{1}{2} S_n(f) |\tilde{K}(f)|^2}, \qquad (2.47)
\end{aligned}
$$

where we used that $\langle n(t) \rangle = 0$ and $\langle h(t) \rangle = h(t)$, with the averaging happening over the whole GW signal. $S_n(f)$ represents the detector noise power spectral density (PSD) in the frequency domain and is defined by the following equation:

$$
\langle \tilde{n}^*(f) \tilde{n}(f) \rangle = \frac{1}{2} \delta(f - f') S_n(f). \qquad (2.48)
$$

The detection statistic for matched filtering is called signal-to-noise ratio (SNR) (denoted as ρ), and it can be defined as:

$$
\rho = S/R = \frac{\int_{-\infty}^{\infty} df \, \tilde{h}^*(f) \tilde{K}(f)}{\sqrt{\int_{-\infty}^{\infty} df \, \frac{1}{2} S_n(f) |\tilde{K}(f)|^2}} = \frac{2 \int_{-\infty}^{\infty} df \, \frac{\tilde{h}^*(f) K'(f)}{\frac{1}{2} S_n(f)}}{\sqrt{2 \int_{-\infty}^{\infty} df \, \frac{|K'(f)|^2}{\frac{1}{2} S_n(f)}}}, \qquad (2.49)
$$

where we introduce $K'(f) \equiv \frac{1}{2} S_n(f) \tilde{K}(f)$. More compactly, we can write:

$$
\rho = \frac{\langle h | K' \rangle}{\langle K' | K' \rangle^2} = \langle h | \hat{K}' \rangle, \qquad (2.50)
$$

where $\langle a | b \rangle$ is the inner product defined as:

$$
\langle a | b \rangle \equiv 4 \mathrm{Re} \int_0^{\infty} \frac{\tilde{a}^*(f) \tilde{b}(f)}{S_n(f)} df. \qquad (2.51)
$$

Thus, the SNR is the inner product between the GW strain h and a unit vector $\hat{K}'$. The SNR will then be maximal when h and $\hat{K}'$ are aligned, such that the optimal filter function will take the form:

$$
\tilde{K}(f) \propto \frac{\tilde{h}(f)}{S_n(f)}. \qquad (2.52)
$$

Matched filtering-based pipelines have been used ever since we started using the current interferometric detectors to search for GWs.

2.2.3.1.2 Evidence for gravitational waves through matched filtering pipelines The existence of GWs was first verified on 14 September 2015 after the detection of a GW signal emitted by a BBH merger, referred to as GW150914 [15], discovered by the LIGO [14] and Virgo [40] collaborations through a matched filtering pipeline. The signal was detected by the two active detectors belonging to the LIGO collaboration: LIGO Livingston and LIGO Hanford.

The first time a GW signal from a BNS was detected was on 17 August 2017 by the LIGO-Virgo network of detectors, and it was named GW170817. Its electromagnetic counterpart, GRB 170817A, was detected by INTEGRAL SPI-ACS [41] and Fermi-GBM [42] independently 1.7 seconds after the coalescence, which was the first experimental evidence that BNS mergers and GRB are connected [26]. The host galaxy NGC4993 was localized by the 1M2H Collaboration [43] at 10.9 hours after the merger. Figure 2.7 displays the projections of the 90% confidence regions from the localisations of the experiments. It also shows the discovery image 10.9 hours after the merger and an image from the DLT40 Collaboration [44] of the galaxy 20.5 days pre-merger, where we can clearly see that a new transient appeared [45]. Following the localisation of the source, optical telescopes closely monitored the afterglow of the event. It was observed that what started as a GRB went on to lose energy until only radio waves remained [46]. GW170817 / GRB 170817A was the first event that truly benefited from MMA, as experiments based on GW detectors, electromagnetic telescopes and, in particular, optical telescopes were all able to contribute. The event was revolutionary in many ways. In particular, other than, as already mentioned, proving the relation between BNS mergers and sGRB and further constricting the NS EOS, it allowed for an independent measurement of the Hubble constant [47], it enabled an accurate measurement of the speed of GWs (the same as the speed of light) [45] and it suggested the origin for many heavy metals [48]. [49]

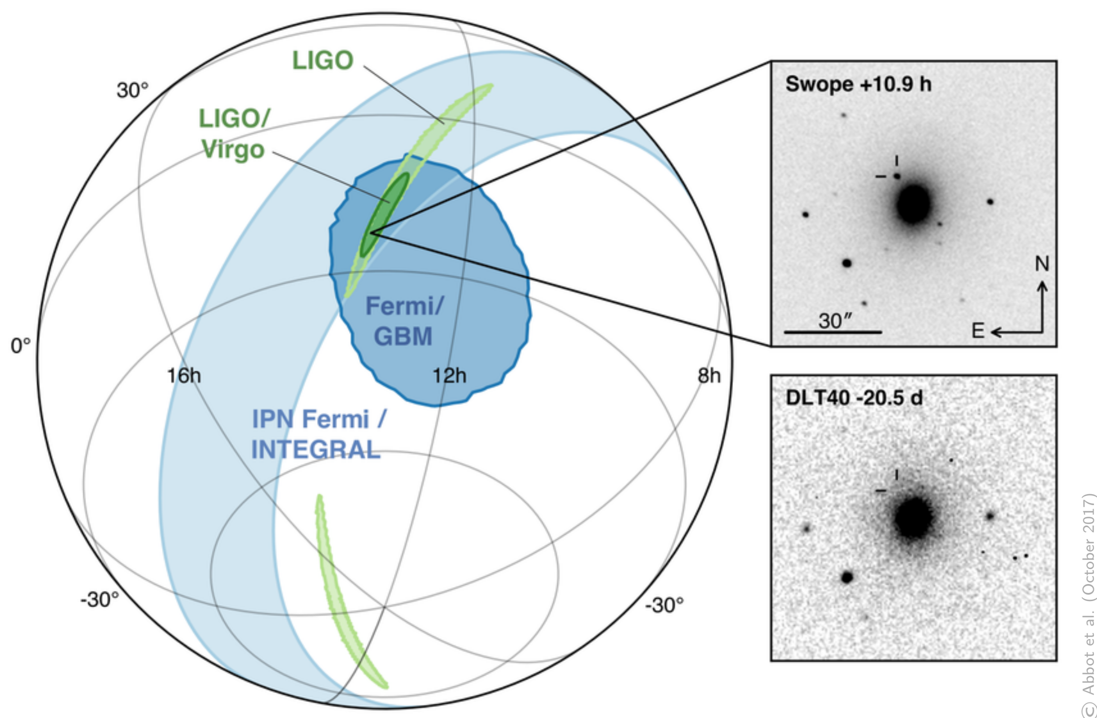


Figure 2.7: Sky localization of GW170817 / GRB 170817A. Taken from [45].

Since 2007, the LIGO and Virgo Collaborations have agreed to share and analyse the data collected by their

detectors and jointly publish their results. The Kamioka Gravitational Wave Detector (KAGRA) Collaboration [50] joined the agreement in 2019.

Thanks to the existence of this network of detectors, we are able to confirm detections by looking for detector coincidences and, as such, many GWs have been detected to date, some of whose components and remnants can be viewed in Figure 2.8. The most recent at the time of writing is GW230529_181500 [51], which was detected during the fourth observing run of the LIGO-Virgo-KAGRA detectors network. The GW is thought to have been emitted by a merger between a NS and an object in the lower mass gap. The lower mass gap is a gap in the mass distribution of compact objects, spanning approximately from $3M_{\odot}$ to $5M_{\odot}$, thought to separate the heaviest NSs from the lightest stellar-mass BH [52]. This makes us believe that, as detector sensitivity improves and new generations of detectors are introduced, it will soon be possible to detect more exotic objects, such as NSs and objects in the mass gap, more frequently. With more and more data about harder-to-detect objects incoming, the present study is particularly relevant, as it will allow for fast and sustainable first trigger alerts. Through this, the correct identification of early BNS inspirals will be able to provide more insight at the intersection of astrophysics, dense matter, gravitation and cosmology.

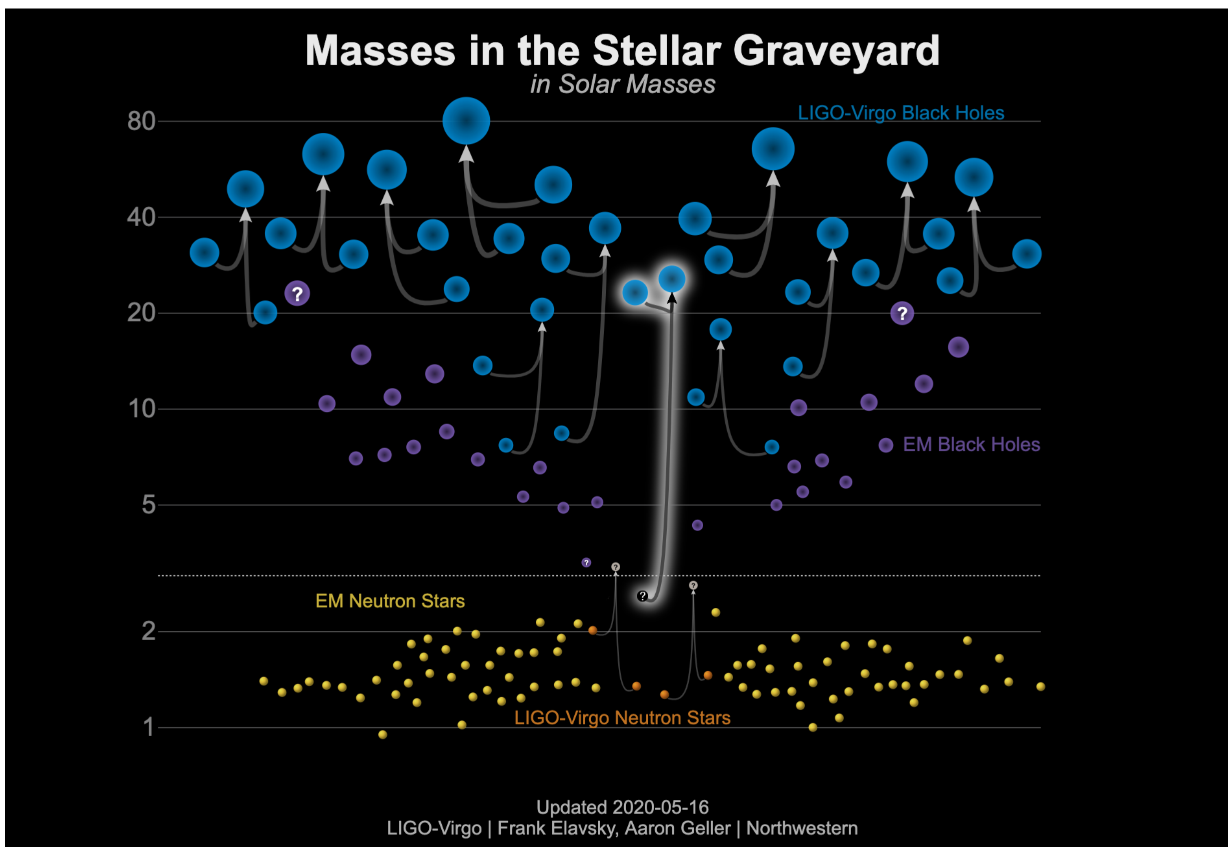


Figure 2.8: Masses of components of CBC signals and the corresponding remnants. Taken from [53].

2.2.3.1.3 Challenges As much as matched filtering has contributed to GW searches up until now, it also has its faults.

Matched filtering requires modelled waveforms to compare the signals to and is performed based on template banks. Template banks are composed of modeled waveforms with specific combinations of system

parameters, spanning the likely parameters for the astrophysical sources of GWs we expect to detect [54]. However, this method lacks the flexibility to detect unanticipated phenomena. Let us say that GR does not predict certain physical phenomena well enough or that there are phenomena that it does not even predict at all: matched filtering would never be able to detect those. The method biases our predictions towards the theoretical ones, not allowing us to reflect the true population of GWs and not granting us the opportunity to explore new physics outside of GR.

Matched filtering is also highly computationally intensive, which makes the time required to analyse the data and obtain results significant. It is important to be able to do a low-latency follow-up of any candidate events in order to allow for MMA. Furthermore, matched filtering costs a lot of resources and energy. This is especially relevant for the current scientific landscape, where we build huge experiments that consume vast amounts of energy and contribute massively to climate change. Thus, whilst trying to create a good societal impact by exploring the unexplored, we are indeed just creating more pollution and leaving the world worse off than we initially found it.

Third-generation detectors, Laser Interferometer Space Antenna (LISA) [55] and the Einstein Telescope (ET) [56], will be the biggest challenge for matched filtering. More and more different data will become available, as different bands of frequency emerge and sensitivity increases, changing the paradigm of how we currently use matched filtering. Third-generation detectors will increase the volume of the Universe to be explored. The template banks matched filtering is based on would have to be expanded to accommodate these new possibilities, requiring more resources and increasing the computational load on matched filtering-based pipelines. Using matched filtering-based pipelines for GW searches in third-generation detectors will be unfeasible.

2.2.3.2 Towards multimessenger astrophysics

BNS coalescence signals have a very long orbiting stage before their merger, which means that out of the common detectable sources, this would be the one most likely to allow for early detection.

Since it is now known that it is highly likely that BNSs emit a sGRB, and its subsequent afterglow, detecting the GWs emitted by the early inspiral of the BNS and locating it is essential for MMA. If other types of detectors, such as electromagnetic and, in particular, optical telescopes, look for the upcoming merger in the sky, we will not only be able to gather more information about the event, but it will also further help us confirm that the event was indeed a BNS merger. With the event information, we can further restrict the BNS EOS, bringing us closer to discovering how dense matter behaves, through analysing how tidal deformability affects the individual components. Moreover, in particular, if the electromagnetic messengers can constrain distance and inclination, masses can be better constrained from GW data.

The (almost) simultaneous detection of GW170817 and GRB 170817A, as well as the identification of the host galaxy and the observation of the afterglow, was a revolutionary event that showed just the tip of the iceberg of what MMA can achieve.

The goal of this study is to explore the possibility of setting a DL-based pipeline to run on an FPGA

(introduced in Section 2.2.3.2.2) that is able to detect GWs emitted by the early inspiral of a BNS at the lowest possible frequency.

2.2.3.2.1 Deep learning DL [57] is the subfield of ML based on artificial neural networks (ANNs). NNs are often used in the context of supervised learning, where input examples X, associated with a label y, also known as the target, perform future predictions based on past evidence. NNs take inspiration from the human brain and how humans learn. The simplest instance of a NN, invented in 1943 [58] and first implemented in 1957 [59], is called the perceptron, and it imitates the neuron. Figure 2.9 shows the layout of the perceptron model.

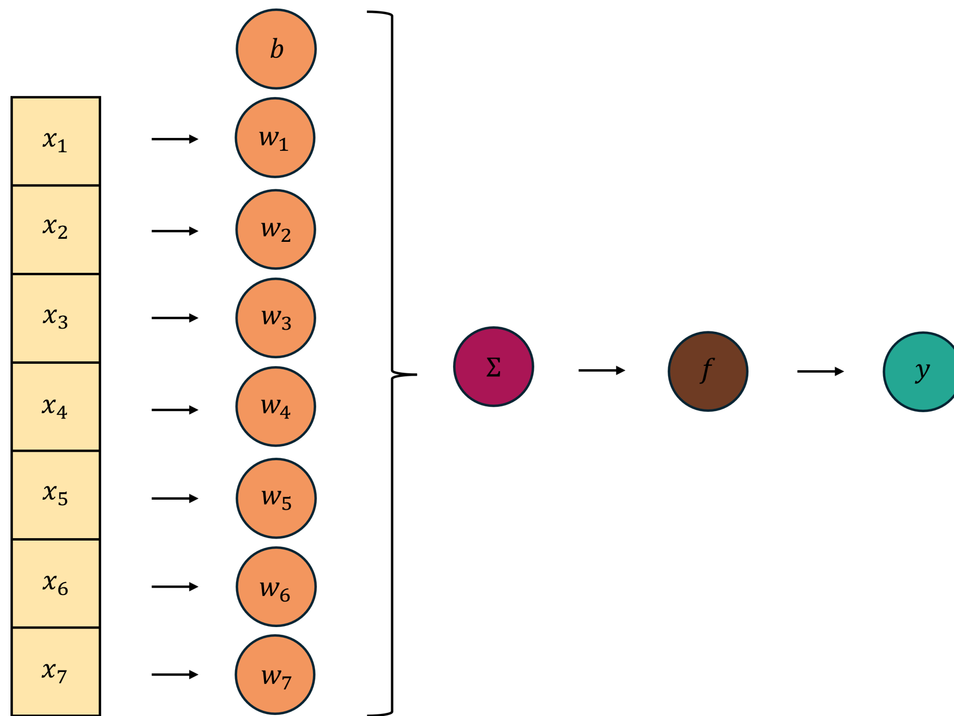


Figure 2.9: Visual representation of the perceptron model. The inputs x_i are multiplied by learned weights w_i and added together, along with a bias term b. They are then fed to an activation function f to give an output y.

The output of the perceptron is given by:

$$y = f\left(\sum_{i}^{n} x_i w_i + b\right), \tag{2.53}$$

where x_i are the inputs of the network, $w_{i>0}$ are the learnable weights, which are between 0 and 1, b is a learnable bias, y is the output, and f is the activation function. The activation function mimics the firing of a neuron and decides what the output will look like from the previous sum. A simple example of an activation function is the binary step, where:

$$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0. \end{cases} \tag{2.54}$$

The model weights are learned and optimized through the evaluation of a cost function, also referred to as

the loss function, which can be defined in many different ways, depending on the application. A common example is the L2-norm cost function:

$$J(w,b) = \frac{1}{2}||h_{w,b} - y||^2, \quad (2.55)$$

where we aim to minimize the difference between the ground truth, y, and the output of our model $h_{w,b}$, based on the trainable weights and bias. To minimize the cost function, NNs use an optimizing function, the best-known being gradient descent. Gradient descent is a first-order iterative algorithm that updates the model weights in the negative direction of the cost function, gradually reducing the error until a minimum is reached.

A type of architecture that has been gaining momentum for some years now is the convolutional neural network (CNN) [60], particularly for vision applications and, such as for this study, evaluation of time series data. CNNs use convolutional layers as part of their architecture, which are layers based on the mathematical notion of convolution. A convolution operation applies a sliding (learned) filter to the input, to produce feature maps. These filters capture specific patterns in the data; for example, a horizontal line filter might detect horizontal strokes typical of digits like 2, 4, 5, and 7, while a (semi-)circle filter might identify rounded shapes present in digits such as 2, 3, 5, 6, 8, and 9. By combining these filters, the CNN can infer the presence of digits like 2 or 5 based on the patterns detected. Further refinement is achieved by incorporating additional filters tailored to recognize other distinctive features.

Information like which activation function to use, the kernel size, or the learning rate to use with the optimizing function are called the model's hyperparameters.

NNs are quickly becoming the standard predictors for applications in all sciences and engineering. Once they are trained and the optimal weights are learned, NNs are exceptionally fast at predicting and highly accurate, given an informed architecture for each application. However, it is worth to mention that NN are difficult to interpret, as they are somewhat of a black box. This problem may, nevertheless, be mitigated by using architectures fit for each problem, particularly physics-informed neural networks (PINNs) [61], and by inspecting the relation between the intrinsic parameters of the input and the NN's output.

2.2.3.2.2 Field programmable gate arrays FPGAs are a subset of programmable logic devices (PLDs), consisting of an array of programmable logic blocks and routing channels, as shown in Figure 2.10.

Each logic block is constituted by a 4-input look-up table (LUT) and a flip-flop, as shown in Figure 2.11. A LUT is a data structure that contains pre-computed values of certain operations, in order to bypass the need for real-time computations and speed up processes running on the FPGA. On the other hand, a flip-flop is a digital memory circuit that switches between two states based on its input, storing information at a low cost. A logic block has four LUT inputs, a clock input, and a single output.

FPGAs are able to perform multiple tasks simultaneously due to their hardware design, based on multiple logic gates and flip-flops, where each task can have their own set. On the other hand, parallel computing in CPUs or GPUs involves software, sharing hardware resources, and its efficiency is limited by the soft-

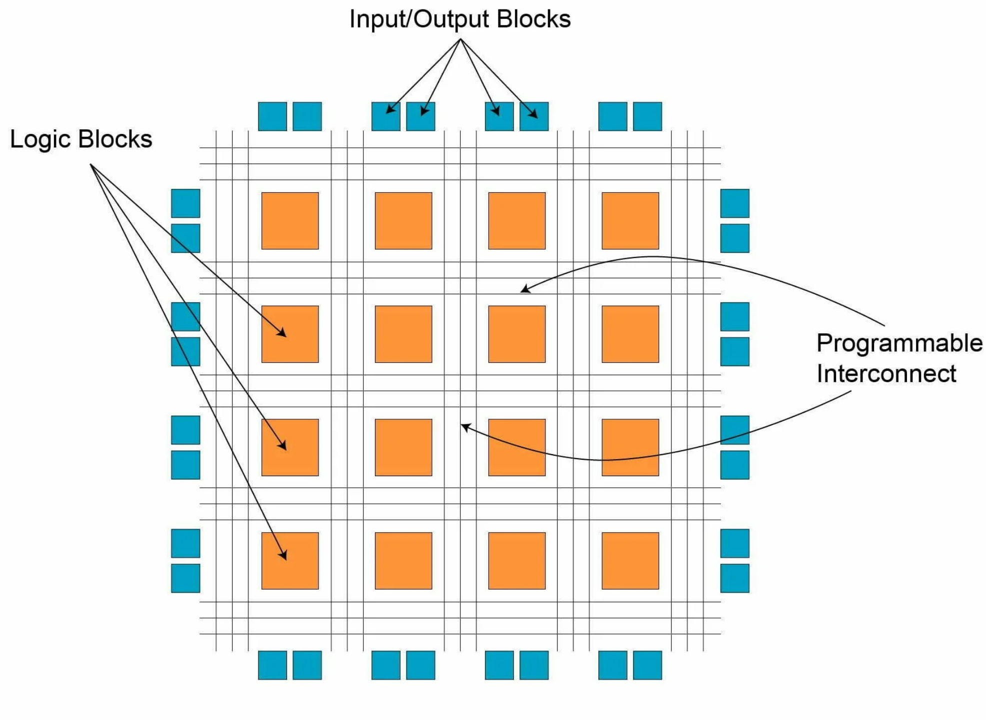


Figure 2.10: Architecture of an FPGA. Taken from [62].

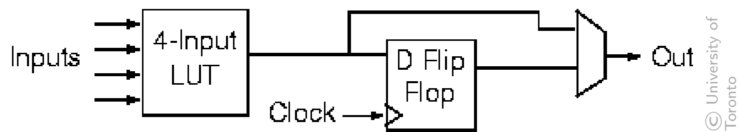


Figure 2.11: Architecture of a logic block. Taken from [63].

ware's ability to manage the concurrency of the tasks. Furthermore, this type of architecture allows for custom digital circuits using hardware description languages (HDLs), meaning the FPGA architecture can be customized using software, such as Vivado [64], making it so that it can become customized for particular tasks. The customization of the hardware of FPGAs is outside of the scope of this study, but it is an interesting notion to explore in future work.

The main processing unit in recent FPGAs is the deep learning/data processing unit (DPU), but FPGAs also have an integrated CPU. The DPU is able to concentrate on data-centric computations and data transfer operations, while its CPU can handle application support tasks, such as interacting with the user.

In general, FPGAs are an affordable high-performance computing alternative to the traditional CPUs and GPUs, built to be faster and more energy-efficient. FPGAs are used in a multitude of industries, such as but not limited to aerospace, automotive, broadcasting, data centres, medical and video and image processing [65], usually for low-level bulk data processing.

In this study, we undertake the task of embedding an FPGA with a NN built for the detection of the early inspiral of BNS mergers. FPGAs have been used in physics applications for many years, mainly in decision-making tasks that need to be performed in the order of microseconds, such as the ones at CERN [66]. Similarly, GW searches need low-latency triggers to detect the signal coming before the merger happens,

in order to allow for MMA. As such, we are confident in the potential of FPGAs for GW searches.

2.3 Related work

Baltus et al. (2021) [67] used a CNN as a binary classifier for BNS merger signals, as a proof-of-concept for a pipeline based on unmodelled prediction. The work used only the inspiral part of the signal, i.e. before the merger happened, to train and test the model, aiming for early alert. Baltus et al. (2022) [68] complemented the first study by introducing curriculum learning, starting to train the network with easier cases (bigger frequencies as the maximum cut-off) and later leaning onto more difficult cases (smaller frequencies). The work achieved much of what it aimed to do, but it had some limitations. The proposed model was very shallow for very large data, resulting in insufficient constraints on the NN. The precision of the model was not optimal, with many unexplained false positive alerts. The whitening portion of the data preparation took a long time, which slowed down the whole pipeline.

The current state of the art for audio generation is a model called WaveNet [69]. This work takes ideas from other revolutionary works, such as 1x1 convolutions [70] and residual NNs [71]. It also explores the idea of gated activation, making it easier for the network to filter through irrelevant information. WaveNet has also recently started to be adapted to be used as a classifier [72]. WaveNet takes a lot of state-of-the-art ideas for NNs, so it is very complex and challenging to implement.

FPGAs have recently started to be used for DL applications [73] [74]. A focus has been recently put on physics applications [75], with a relevant study by Que et al. on embedding FPGAs with graph neural networks (GNNs) for high-energy applications [76]. These studies are all based on up-and-coming software for programming the NNs for deployment on the FPGA, with significant work needed to make them work.

2.4 Research aims: Low latency neural network deployment

Matched filtering is widely used in GW search pipelines, but NN-based pipelines are slowly entering the scene, with their inference time being much faster than that of matched filtering [39]. The present study similarly leverages NN models for their rapid classification abilities.

Furthermore, current GW search pipelines are run in CPUs and, more recently, GPUs. GPUs were designed to render video and graphics, optimising them for matrix calculations. Thus, NN predictions can be made quickly since NNs boil down to simple matrix calculations. The main processing unit in an FPGA, the DPU, is also excellent at fast matrix multiplication. FPGAs also have an embedded CPU; the existence of the DPU offloads networking and communication workload from the CPU, making it free to handle application support tasks. FPGAs have a much lower upfront and upkeep cost than that of CPUs and GPUs and a longer lifetime: FPGAs can last up to decades, whereas a GPU under heavy use, such as running GW search pipelines, will last around 5 years [77]. Possibly the biggest advantage of FPGAs is their low power consumption compared to other high-performance computers. The energy efficiency of FPGAs makes them an even more affordable option, and they will have much less environmental impact than the current state-

of-the-art hardware. FPGAs are also able to perform multiple tasks in parallel thanks to their architecture. Lastly, FPGAs have the capability to have their hardware redesigned to suit the needs of the NN application, which will not be touched on by this study but is a great exploration point for future work.

This study aims to combine fast detection methods with hardware built for low latency to detect the early inspiral of BNSs. The early detection of GWs from BNS mergers is crucial for advancing our understanding of the Universe through MMA. By identifying these events promptly, we can trigger follow-up observations across the electromagnetic spectrum, enhancing our ability to study these events and, consequently, their individual components. This will not only deepen our understanding of these astrophysical phenomena but also contribute to broader fields, such as dense matter physics, cosmology, and heavy element formation. Through our innovative approach, we aspire to set a new standard for early detection, paving the way for groundbreaking discoveries in the era of MMA.

Chapter 3

Methodology

After having defined the current state of the art for GW searches and our competing proposal, in this Chapter we turn to the specifics that make this study so remarkable.

3.1 Working with time series data from binary neutron star coalescence gravitational wave signals

In this Section, we delve into working with time-series data extracted from BNS coalescence GW signals. The data used are the foundation for training and testing our NN models.

3.1.1 Description of data

In this study, we employ the dataset used by Baltus et al. [68], as we are interested in enhancing the performance, in terms of accuracy and speed, of this previous work. Each data sample consists of a one-dimensional (1D) whitened time series, composed of solely noise in 50% of cases, and noise with a simulated GW signal, also referred to as an injection, from a BNS coalescence added in the other 50% of cases. Whitening is a data preprocessing step that involves transforming the data so that the spectrum of the signal is flattened [78]. For each time series, a sample of what the detector signal would look like is generated for each of the LIGO-Virgo detectors, using the same parameters.

The signals mimic GW signals from BNS coalescence with individual component masses between $1 M_{\odot}$ and $3 M_{\odot}$, to encompass all the possible masses for NSs, as explored in Section 2.2.1.1. The sources are uniformly distributed over the sky and spin effects were considered, including precession effects. The waveform approximant used to generate the injections is `SpinTaylorT4`, and they are generated in the time domain. Taylor waveforms are based on post-Newtonian theory, thus being ideal for approximating the inspiral phase of a CBC [79]. In this study, we use only the early inspiral parts of the mergers, making a Taylor waveform the most logical choice for simulating the effect of a GW. We use the `SpinTaylorT4` approximate in particular as it simulates solely the inspiral phase of the signal.

The original injection is over 300 seconds of data. However, the data are divided into five subsets, with cut-off maximum frequencies sampled from a Gaussian distribution with a standard deviation of 2.5 Hz and mean of 40, 35, 30, 25, and 20 Hz for each subset, as shown in Figure 3.1, each with 300 seconds of data, corresponding to 155648 data points per sample. The averages of the minimum and maximum frequencies in each data set are described in Table 3.1.

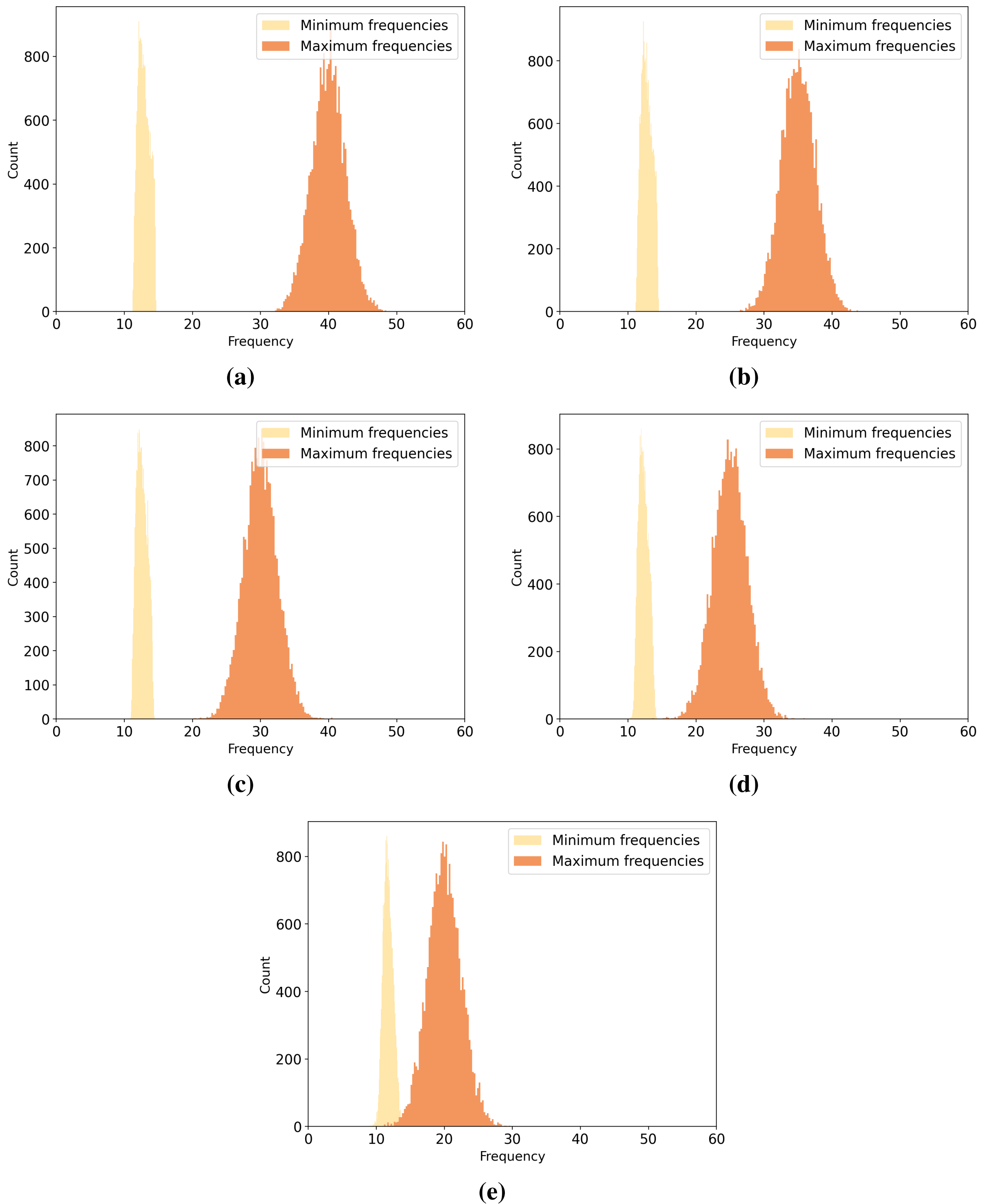


Figure 3.1: Histogram of the minimum and maximum frequencies of the signals for training **(a)** data set 1, **(b)** data set 2, **(c)** data set 3, **(d)** data set 4 and **(e)** data set 5.

Data set	Minimum frequency (Hz)	Maximum frequency (Hz)
1	12.9	40.0
2	12.8	35.0
3	12.6	30.0
4	12.3	25.0
5	11.7	20.0

Table 3.1: Averages of maximum and minimum frequencies for each data set. The minimum frame is chosen such that all signals are 300 s long.

Together with the time series, we have the parameters of the event, such as the component masses, the component spins, the SNR, and the partial inspiral signal-to-noise ratio (PISNR) of each detector and the network. The PISNR is a concept that is introduced by Baltus et al. [67] and describes the corresponding SNR for the signal up until the cut-off point before the merger. We choose to use the PISNR as a main parameter of the signals, instead of the usual SNR, as the merger and ringdown phases of the events are not included in our data strains, and the PISNR ends up being the effective SNR of our data. Figure 3.2 shows the difference in distribution between the SNR and the PISNR and Figure 3.3 shows the PISNR distribution for each of the data sets. The highest PISNRs reach between 100 and 200, making these samples extremely quiet when compared to samples including the merger phase of the signal.

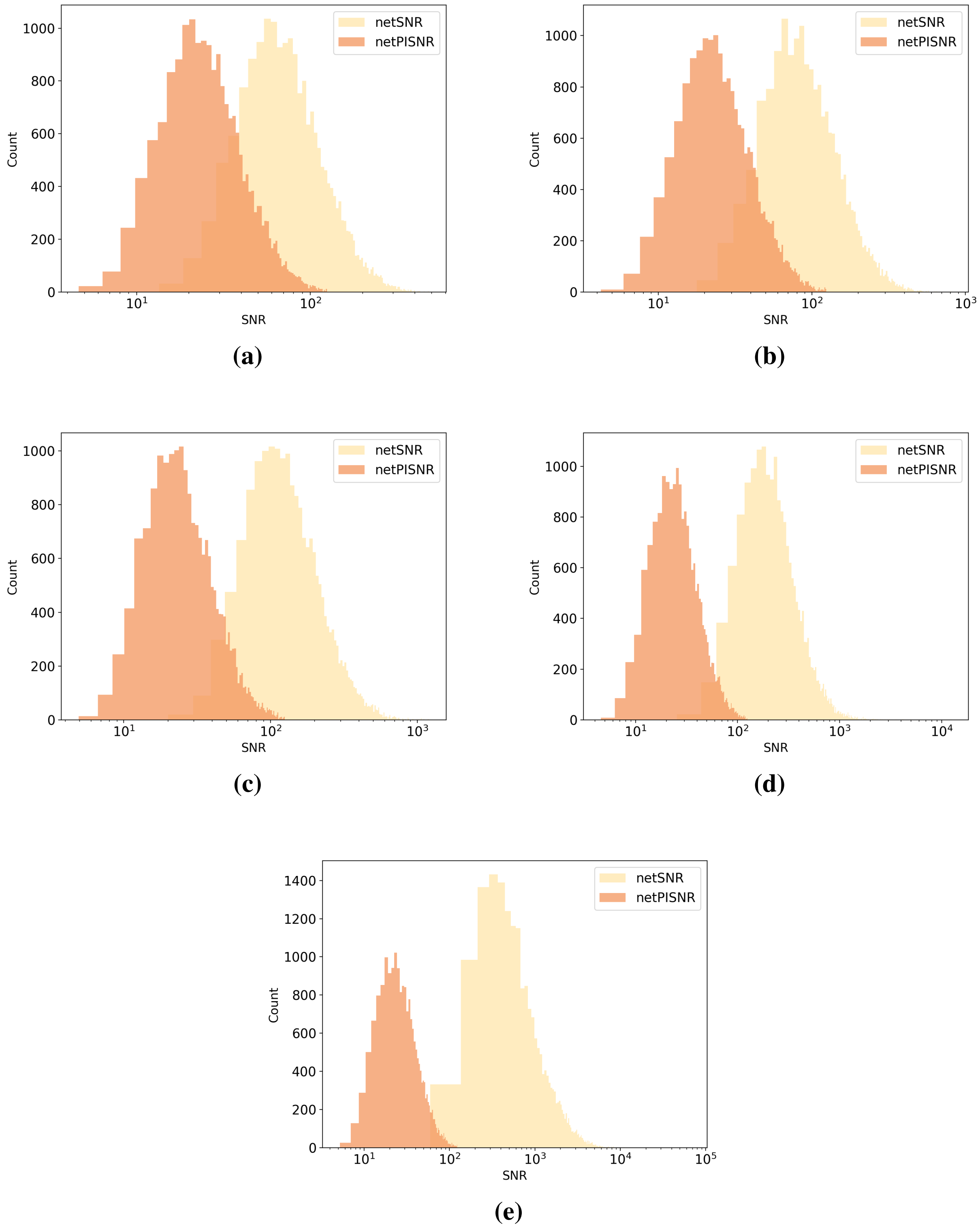


Figure 3.2: Histogram of the SNR and PISNR of the network of the detectors for training **(a)** data set 1, **(b)** data set 2, **(c)** data set 3, **(d)** data set 4 and **(e)** data set 5. The x-axes are set to logarithmic scale. See Table 3.1 for a definition of the data sets.

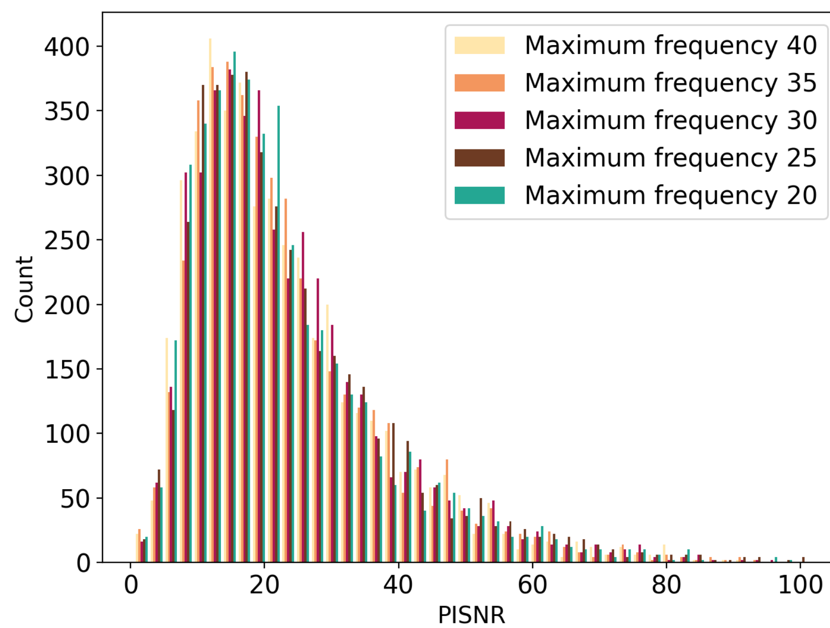


Figure 3.3: Distribution of PISNR of the test subset for each of the data sets.

3.1.1.1 O3 Gaussian data

The NNs were trained on Gaussian-noise data, based on the theoretical predictions for observing run 3 (O3) of the LIGO-Virgo detector network. The O3 Gaussian noise is generated from the design-sensitivity PSD for each of the three detectors, advanced LIGO (aLIGO) [80] and Advanced Virgo [31]. The generation of the data was performed using the PyCBC package [81]. In particular, the noise time series were generated from the `aLIGOaLIGO140MpcT1800545` and `aLIGOAdVO3LowT1800545` theoretical PSDs, respectively for the LIGO and Virgo detectors.

The data highly depends on the PSDs of each detector. The PSD is a measure of power over different frequencies in a signal and it provides a comprehensive description of the noise characteristics of a detector. Through the PSD of a detector, we can better understand its sensitivity at different frequencies. The PSDs of the Advanced Virgo and aLIGO detectors and, in contrast, of the ET and cosmic explorer (CE) (still in the planning phase) detectors are shown in Figure 3.4.

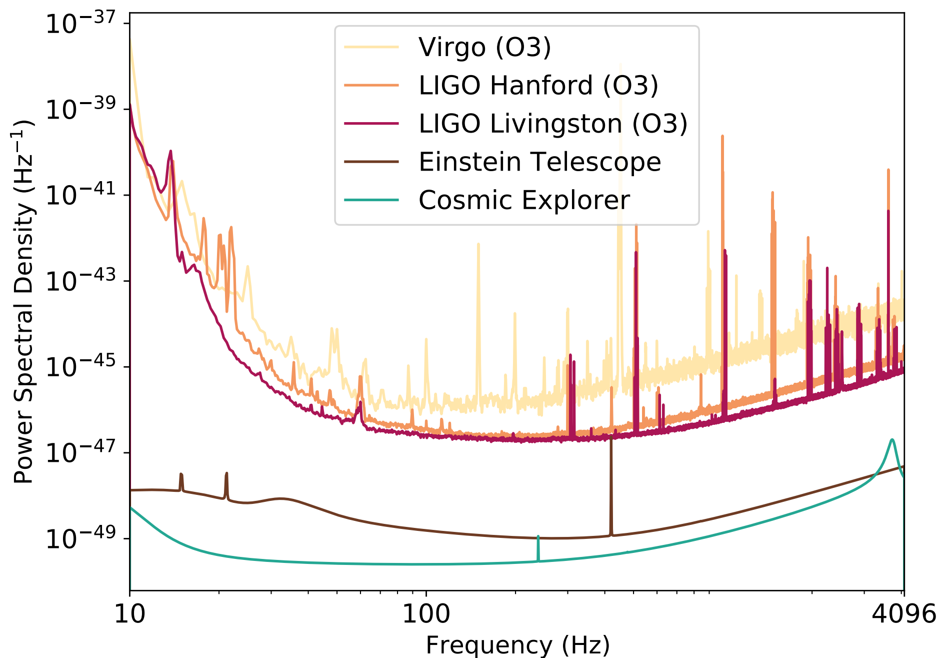


Figure 3.4: PSDs of the aLIGO (Hanford and Livingston), Advanced Virgo, ET and CE detectors.

3.1.2 Data preparation

Some data preparation is done before the data are fed to the NNs. Firstly, the 1D data are loaded and stacked such that we end up with a $N \times d \times L$ tensor, where N is the number of samples, d is the number of detectors, and L is the time series length. A visual representation can be seen in Figure 3.5. For the NNs described in this study, we always use $d = 3$. Tests were done with $d = 1$ but they showed that the NNs perform better for $d = 3$.

The order in which the samples are stored is randomized, taking an array with numbers from 1 to n and drawing without replacement and following that order.

We used approximately 70% of the data for training the NNs, 10% for validation and 20% for testing. A

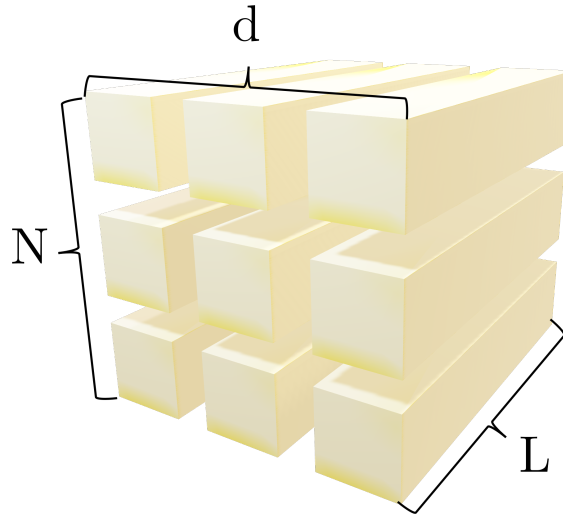


Figure 3.5: Shape of the input data.

tensor like the one described above is created for each of the three (training, validation, and testing) batches, where, for each data set, as described in Table 3.1, a total of $N = 11200$ samples were used for each class, totalling 80000 samples for training, 10000 for validation and 22000 for testing the NNs.

3.1.3 Data storing and loading

To speed up training and testing, we carefully designed the way data is stored and then loaded into memory for when we are ready to train the NNs and how much random access memory (RAM) is taken up by loading up the data since we are using a GPU to train the NNs.

The data are laid out in the tensors described in Section 3.1.2, being divided into different batches because of RAM constraints and, once all of the tensors are ready, they are all stored in `.npz` files from the NumPy package [82]. This file type allows for multiple NumPy arrays to be stored in a single file and then be loaded one by one according to a key. Using this type of data file allows for the data to be well organised as well as quickly loaded into memory once it is needed for training and testing.

The GPU used to train the NNs for this study is the NVIDIA Tesla V100 [83], with 32GB of available RAM. Thus, the memory is a limiting factor, and the data were stored in batches of 200.

3.2 Creating a neural network to distinguish BNS coalescence GW signals from noise

The goal of creating a NN for this study is to have a binary classifier that distinguishes between when the data contain only noise (the negative class) and when the data contain noise and an injected GW signal (the positive class).

In this study, two NNs were trained and embedded into the FPGA. We started by recreating the NN described by Baltus et al. [68], henceforth referred to as GregNet, to have a point of comparison, and then explored an implementation of WaveNet, henceforth referred to as GWaveNet. Both of the architectures are implemented using the `PyTorch` package [84].

3.2.1 Reference neural network: GregNet

We started by recreating the network described by Baltus et al.1,[68], with some slight changes, to have a reference point. It is useful to first look at what types of transformations make up the architecture, as well as other building blocks, such as activation functions, and later move to an overview of the full architecture.

3.2.1.1 Batch normalization

Batch normalization [85] is used to combat the problem of internal covariate shift, through normalizing layer inputs. It normalizes the inputs to have a mean of zero and a standard deviation of one, stabilizing and accelerating the training process.

The formula for batch normalization is given by:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \varepsilon}}\gamma + \beta, \tag{3.1}$$

where x is the input data and $\mathrm{E}[x]$ and $\mathrm{Var}[x]$ are the correspondent expectation value and variance, ε is a given parameter used for numerical stability, γ and β are learnable bias parameters and y is the output of the batch normalization layer.

3.2.1.2 One-dimensional convolution

1D convolutional layers are inspired by the mathematical notion of convolution and are now used broadly in DL [86]. A way to visualize a 1D convolution is shown in Figure 3.6.

A layer with input size $(N, C_{\mathrm{in}}, L_{\mathrm{in}})$ will have output size $(N, C_{\mathrm{out}}, L_{\mathrm{out}})$, where the output for each sample i is given by:

$$y(N_i, C_{\mathrm{out}_j}) = b(C_{\mathrm{out}_j}) + \sum_{k=0}^{C_{\mathrm{in}}-1} w(C_{\mathrm{out}_j}, k) \star x(N_i, k), \tag{3.2}$$

where N is the batch size, C is the number of channels, L is the length of the signal, b is the bias, w are the weights, x is the inputs and y is the outputs. The operator $\star$ is the cross-correlation operator, defined for two discrete functions f and g as:

$$(f \star g)[n] \equiv \sum_{m=-\infty}^{\infty} \overline{f[m]}\, g[m+n], \tag{3.3}$$

where $\overline{f[m]}$ denotes the complex conjugate of $f[m]$.

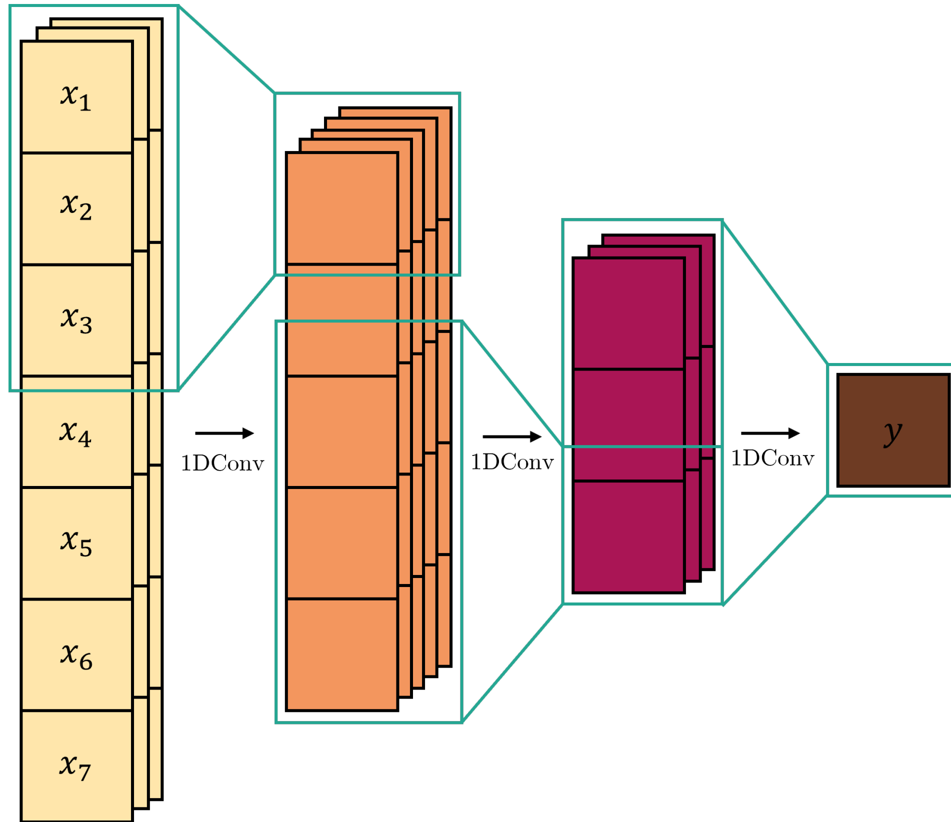


Figure 3.6: Visual representation of an example of 1D convolutions. The cream blocks represent the input data, with a length of seven and three channels. The orange and burgundy blocks represent hidden layers, with lengths of five and three, respectively, and five and three channels. The brown block represents the output. The blue outlines represent the 1D convolutions, with kernel size three.

The length of the output, L_{out} is given by:

$$L_{\text{out}} = \frac{L_{\text{in}} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel_size} - 1)}{\text{stride}} + 1, \tag{3.4}$$

where padding, dilation, kernel_size, and stride are parameters of the convolution[1]. The depth of the output, on the other hand, is governed by the chosen number of output channels, alternatively called filters. The kernel size of the convolution is how many data points the convolution can see at once. In the example in Figure 3.6 the kernel size is three. The padding is how much is added to the edges of the output to extend the input to the convolution. In Figure 3.6 padding is zero, but if we added one zero on each end of the input and used those points as part of the convolutions, then padding would be one and so on. The dilation is the spacing between the elements in the kernel. In Figure 3.6 the dilation is one, but if the input for the first pass were x_1, x_3 and x_5 instead, then the dilation would be two. The stride is how many data points the network skips between each pass. In Figure 3.6 the stride is zero, but it would be one if the second pass was for x'_3, x'_4 and x'_5, instead of what it currently is, x'_2, x'_3 and x'_4.

[1]These concepts are difficult to explain with words. *vdumoulin* has comprehensive visualizations for each in two-dimensional (2D), with the concepts being able to be generalised for 1D [87].

3.2.1.3 Activation functions

As introduced in Section 2.2.3.2.1, activation functions calculate the output of an operation layer based on its inputs and weights.

Two different activation functions were used overall in the making of this architecture: the rectified linear unit (ReLU) activation function for the hidden layers and the Sigmoid activation function for the activation of the output of the NN. ReLU is a fast and simple activation function, linear for positive inputs, that mitigates the vanishing gradients problem (where gradients become very small during backpropagation, making the training process less effective), and thus is very popular as an activation function for hidden layers. Sigmoid, on the other hand, is non-linear, outputting values between zero and one and thus being a good choice for a final activation function in binary classification, also since it does suffer from the vanishing gradients problem.

Given an input x, the ReLU activation function [88] is described by:

$$\mathrm{ReLU}(x) \equiv \max(0, x), \tag{3.5}$$

and the Sigmoid activation function by:

$$\mathrm{Sigmoid}(x) = \sigma(x) \equiv \frac{1}{1 + \exp(-x)}. \tag{3.6}$$

3.2.1.4 Maximum pooling

Max pooling is a downsampling technique that reduces the length dimension of the input. In max pooling [89], a window slides over the input, similar to what is represented in Figure 3.6, and takes the maximum value out of the values in its field of vision (kernel). The output layer for each sample N_i with class C_j is given by:

$$y(N_i, C_j, k) = \max_{m=0,\ldots,\mathrm{kernel_size}-1} x(N_i, C_j, \mathrm{stride} \times k + m), \tag{3.7}$$

and the length of the output layer is once again given by Equation (3.4).

3.2.1.5 Flattening

Flattening is necessary as it converts the multi-dimensional output from the convolutional layers into a 1D input, necessary for fully-connected layers.

The flattening layer [90] flattens the input with C channels into having only one channel, by taking each channel and putting it at the end of the previous one in a single dimension, as shown in Figure 3.7.

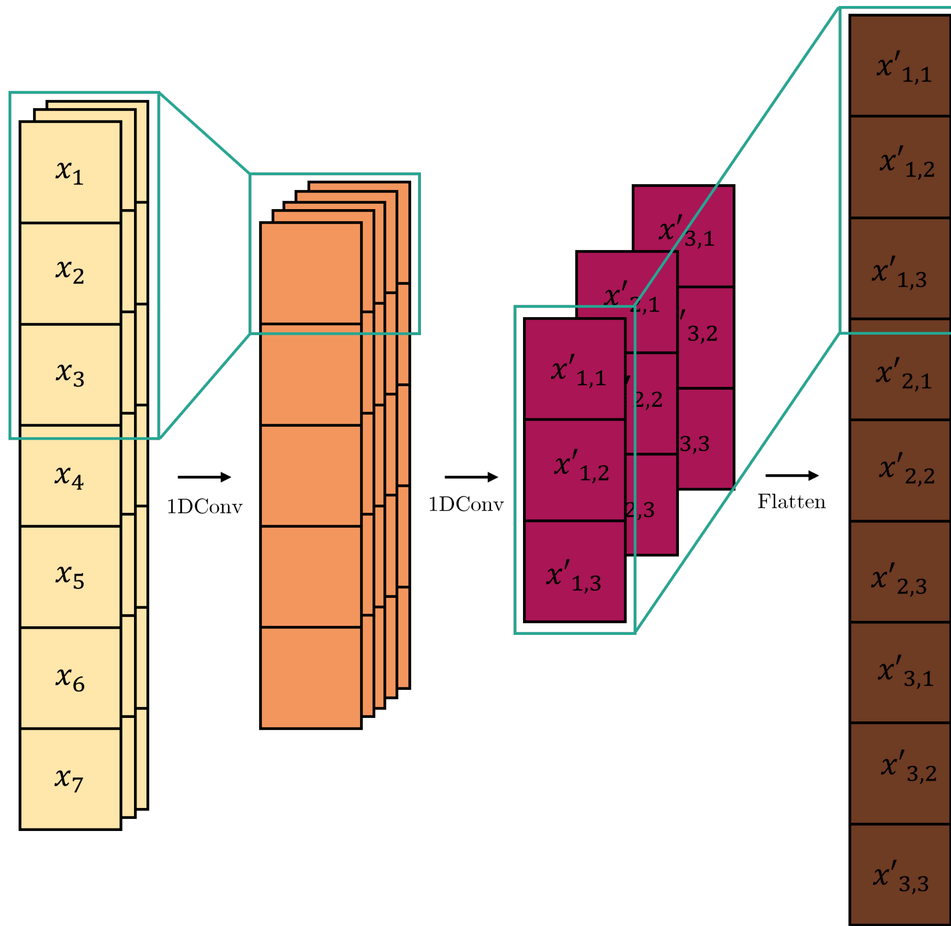


Figure 3.7: Visual representation of a flattening layer. We go from three channels in the burgundy blocks into one channel in the brown blocks through the use of a flattening layer, that adds each channel at the end of the previous one in 1D.

3.2.1.6 Linear layer

The linear, or dense, layer [91] follows the principle of the perceptron, explained in Section 2.2.3.2.1, by using Equation (2.53).

3.2.1.7 Overview of the architecture

The architecture of GregNet is illustrated in Figure A.1 and further described in Table A.1 in Appendix A, where the layers are the ones defined in PyTorch. The size of the padding and the dilation are omitted as they are always set to zero and one, respectively. The full model, whose visualization is generated by the `torchviz` package [92], can be seen in Figure 3.8, totalling 6,179,303 trainable parameters.

The optimizer used was adaptive moment estimation with maximum (AdaMax), a variant of Adam based on the infinity norm, with a weight decay of 10^{-5} and the learning rate (LR) used was 8×10^{-5}. Weight decay is a regularization technique used to penalize complexity, by multiplying the sum of squares of the weights by a regularization parameter and adding it to the loss function. The Adam and AdaMax algorithms are both used for stochastic optimization [93]. The AdaMax optimizer is based on Algorithm 1 in Appendix B.

The chosen loss function was the cross-entropy loss [94], as the study is based on a classification task. The cross-entropy loss, or logistic regression loss, is given by:

$$l(x,y) = \frac{\sum_{n=1}^{N} l_n}{N}, \qquad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}, \tag{3.8}$$

where N is the batch size, C is the number of classes, and $C = 2$ for our case, w_c is the weight of each class c, $x_{n,c}$ is the input of sample n and class c, and $y_{n,c}$ is the ground truth of sample n and class c.

In the original work, the authors weighted the positive class in the `BCELoss` function by a factor of 0.4 to reduce the number of false positives. We opted for using a different implementation of the cross-entropy loss in PyTorch, the `BCEWithLogitsLoss` [95]. The `BCEWithLogitsLoss` function is more numerically stable than using a sigmoid function followed by the plain `BCELoss` function implemented in PyTorch, as it uses the log-sum-exp trick to fuse the two operations into one layer. The original weighting factor was used by passing it through the `pos_weight` argument.

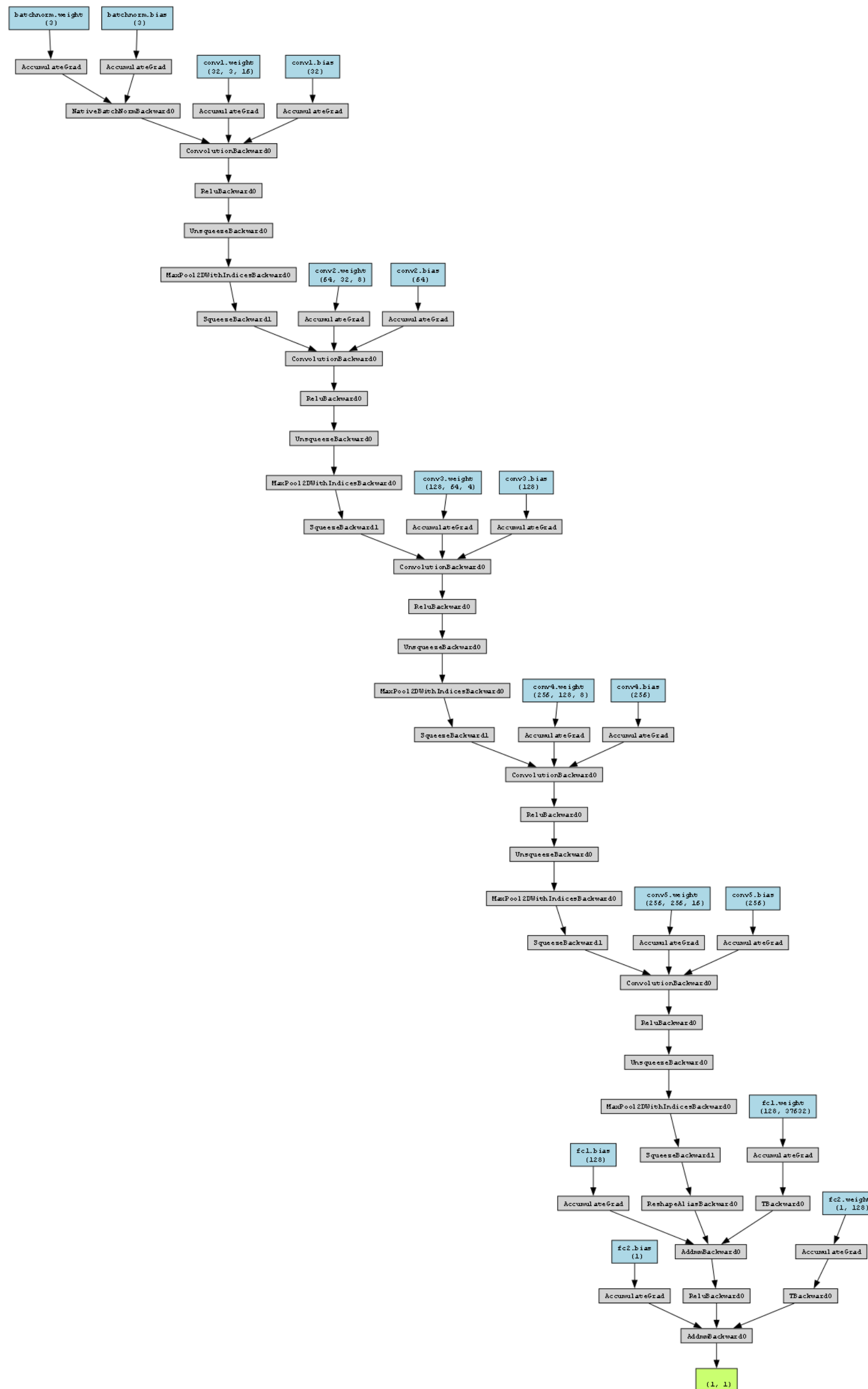


Figure 3.8: Full architecture of GregNet. We start by using batch normalization on the input. Next, we perform a convolution with a kernel size of 16 and 32 filters, followed by a ReLU activation function, and a max pooling layer with a kernel size of four and a stride of four. Then, we perform a convolution with kernel size of eight and 64 filters, again followed by a ReLU activation function and a max pooling layer with the same sizes. This three-layered structure is repeated three more times, with convolution kernel sizes four, eight and 16 and 128, 256 and 256 filters. Then, a flattening layer is applied, bringing the number of channels down to one and the length of the hidden layer to up 37632. Lastly, two linear layers are applied, with lengths of the output 128 and one, respectively, with a ReLU activation function after the first one and a Sigmoid activation function after the second.

3.2.2 GWaveNet

GWaveNet is inspired by the model by van den Oord et al. [69], called WaveNet. WaveNet was created with the goal of audio generation and is the state of the art. We take inspiration from it based on the fact that our time series detector data is audio-like. Additionally, the concept of using a WaveNet-like model for GW data purposes has been briefly successfully explored [96, 97].

Once again, we will first go over the transformations and building blocks of GWaveNet, and then take a look at the full architecture implemented. Let us go over these innovative ideas in chronological order!

3.2.2.1 Gate activation

A first example of gate activation was introduced in 1997 as a feature of the Long Short-Term Memory (LSTM) architecture [98], incorporating a specialized gating mechanism that allows the NN to learn and remember patterns in sequential data, such as time-series data. It enables the NN to selectively remember useful information and forget useless one based on the context, acting as a logic gate.

Gate activation is implemented by passing the output of two independent instances, duplications of the output at that point in the model, after passing through a causal convolution layer and a batch-normalization layer into hyperbolic tangent (tanh) [99] and logistic sigmoid [100] activation functions. After this, the output from each activation function is multiplied. This is illustrated in Figure 3.9.

The hyperbolic tangent activation function is given by its usual mathematical definition:

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}, \tag{3.9}$$

and the sigmoid function is the same one as used for the activation of the output of GregNet, defined in Equation (3.6).

Gate activation is solely used as the activation for the Module Wavenet layers, whereas for the activation of the hidden linear layers, we used the ReLU activation function (Equation (3.5)), and for the activation of the output, we used the sigmoid activation function (Equation (3.6)), as in GregNet. We use ReLU for the linear layers due to its simplicity and efficiency, whereas gated activation is ideal for capturing complex relations in sequential data. Using a combination of the two, we can make the most out of computational efficiency while still maintaining the ability to model intricate data patterns.

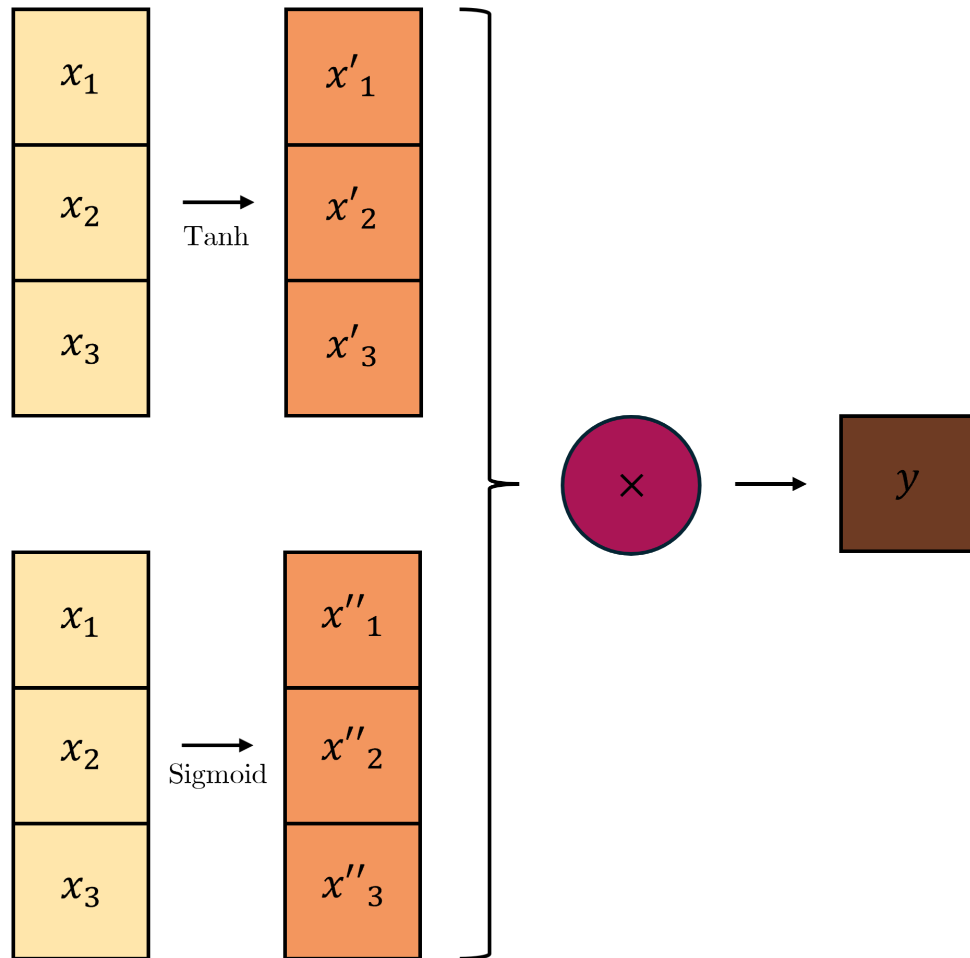


Figure 3.9: Visual representation of gate activation. The cream blocks are two instances of a duplicated input. The orange blocks are hidden layers, with the first being a product of using the tanh activation function on the input and the second of using the sigmoid activation function. These two are then multiplied, in burgundy, and we get to an output, in brown.

3.2.2.2 1x1 convolutions

1x1 convolutions were first introduced by Szegedy et al. in 2014 as a feature of GoogLeNet [70], to reduce dimensionality and perform feature combination. 1x1 convolutions can be seen as a non-linear counterpart to the usual max-pooling layer, but pooling depth-wise instead of length-wise, or to the usual flattening layer, as they can learn non-linear relations and lower the number of input channels without being computationally costly.

1x1 convolutions arise from usual convolutions, with kernel size one and number of output channels smaller than the number of input channels. In our case, we choose to set the number of output channels of the 1x1 convolution to one. 1x1 convolutions earn the name "1x1" as they are usually implemented in 2D problems, but in our case, they are used in 1D convolutions: as such, the term "1x1" is an abuse of notation, kept for consistency with the original work. 1D convolutions of kernel size one are performed, looking at each data point individually and computing the weighted sum for all channels. This is illustrated in Figure 3.10.

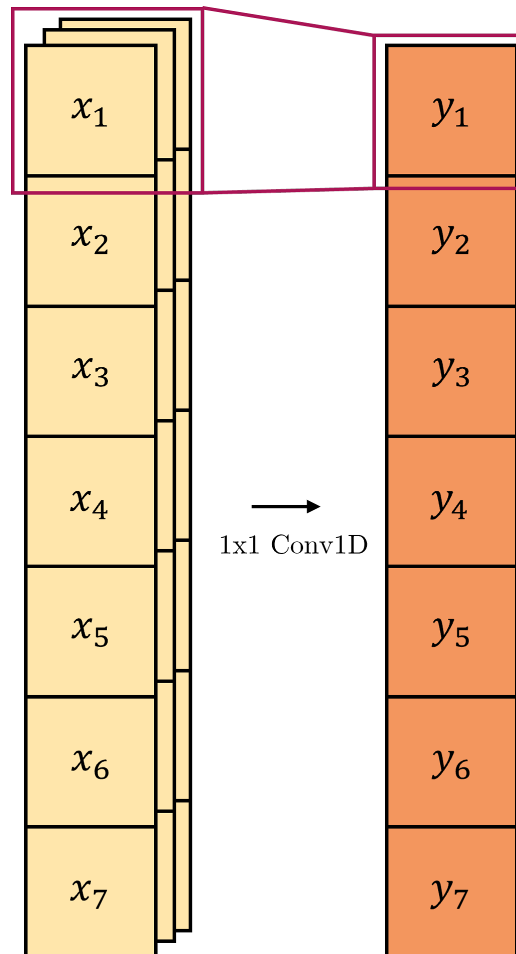


Figure 3.10: Illustration of a 1x1 convolution. The cream blocks represent the input, with a length of seven and three channels. The 1x1 convolution is represented in burgundy, and it has a kernel size of one and one filter. This results in an output, in orange, with the same length but with a new depth of one.

3.2.2.3 Skip-connections

Together with gate activation, skip-connections help the NNs learn and remember relations in the data. They were first introduced in 2015 by He et al. as the main selling point of the residual neural network (RNN) architecture [71], in order to explore deeper NN models without them forgetting about the nature of the input.

Skip-connections provide shortcut connections that skip certain layers of the NN, allowing the network to learn the differences between input and output, also referred to as residual mappings, as opposed to the full mappings. This is illustrated in the original paper, as well as in Figure 3.11, where the x identity is fed to the to the NN again after it has undergone other transformations, in order to help it remember its nature.

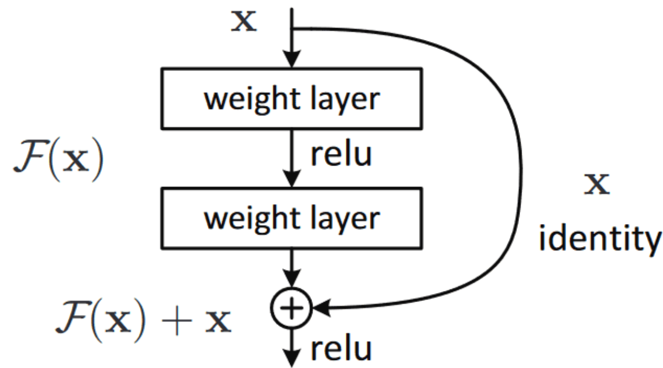


Figure 3.11: Illustration of a skip-connection. The initial input x is saved and then added to the output of the second weight layer in order for the model to remember the input identity and mitigate the risk of overfitting. Taken from [71].

3.2.2.4 Dilated causal convolutions

Causal convolutions were introduced in 2016 by van den Oord et al. [69] as a way to treat temporal data such that the model will not violate the time ordering of the samples. A comprehensive visualization for causal convolutions is included in their paper and can be seen in Figure 3.12.

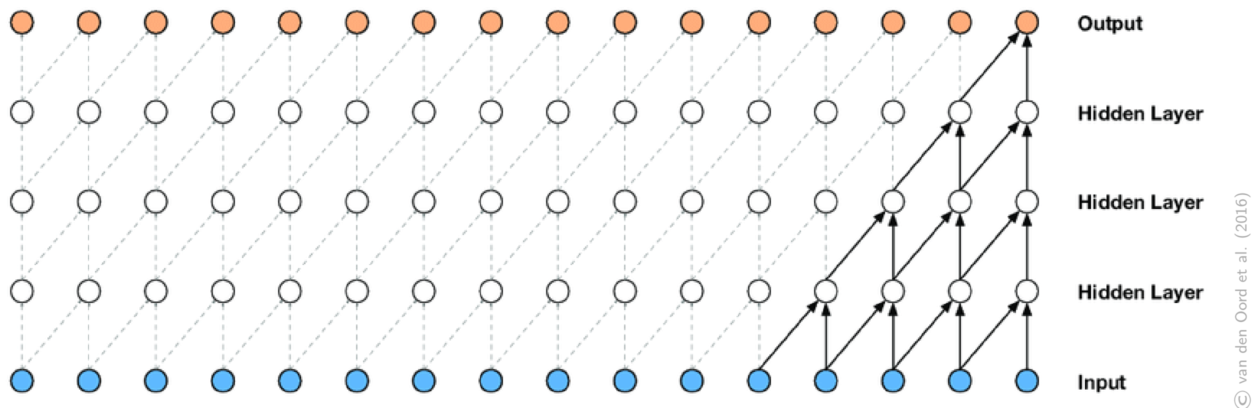


Figure 3.12: Visual representation of a causal convolution. For each new hidden layer, time laws are respected, as we can see from there not being any arrows going from right to left. Taken from [69].

Next, dilation, introduced in Section 2.2.3.2.1, is added to the convolutional layers with exponentially growing values for each of the convolutions. This approach allows the NN to capture a larger context of the signal without increasing the computational load significantly. As such, the NN's ability to detect complex patterns over long time scales in the data is enhanced.

The dilated causal convolution effect is achieved by leveraging padding and dilation in a usual convolution (see Section 3.2.1.2) and a code implementation for PyTorch is shown in Listing C.1 in Appendix C.

An earlier implementation of WaveNet for GW searches [97] opted against using the causal convolutions feature. However, we decide to include it, since it makes for a more closely PINN, as it forces the NN to follow time laws.

3.2.2.5 Overview of architecture

The architecture of the GWaveNet model is as described in Table A.2 in Appendix A, where the layers are the corresponding functions in PyTorch and *ModuleWavenet* is defined as described in Listing C.2 in Appendix C. The ModuleWavenet blocks are further illustrated in Figure 3.13, where i is the layer counter, from 0 to 6. The full architecture of the NN is displayed in Figure 3.14, totaling 522,598 parameters.

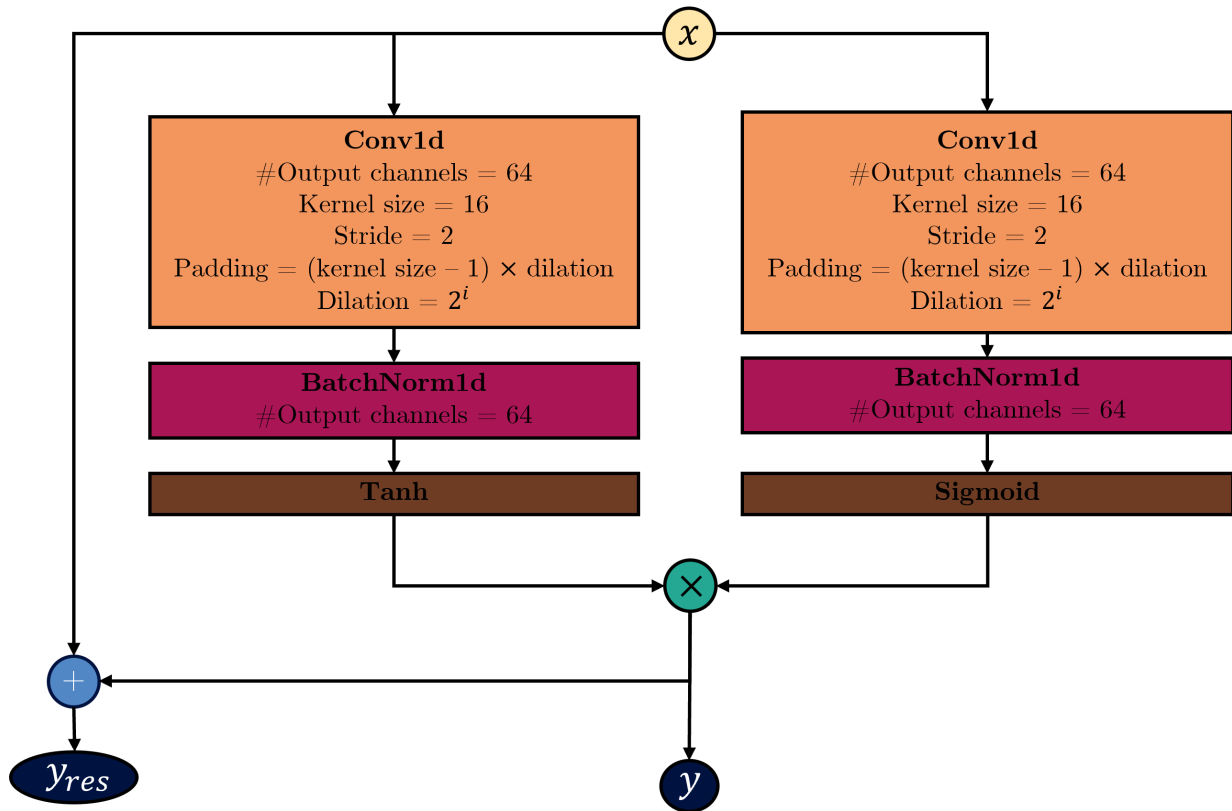


Figure 3.13: Architecture of a ModuleWavenet block. An input x, in cream, gets duplicated and each copy gets fed into a causal convolution block, in orange, followed by a batch normalization layer, in burgundy. Then, one of the outputs gets fed into a tanh activation function, in brown, and the other into a sigmoid, also in brown, and the two instances are multiplied, in green. This is the main output of the ModuleWavenet block, in dark blue. Still, the output also gets added to a copy of the initial input, in light blue, and that is saved as the residual output of the block, in dark blue.

We use the Kaiming uniform weight initialization method in order to initialize the weights of the filters of our convolutions, so that training becomes more stable. This method was introduced in 2015 by He at al. [101] in order to address problems with non-linearities that older weight initilization methods had and it is particularly useful for NNs using the ReLU activation function. As such, we use weight initialization after the 1x1 convolution and before the first ReLU unit. The weights will have values sampled from a uniform distribution between $-b$ and b, the bounds, where:

$$b = \text{gain} \times \sqrt{\frac{3}{\text{fan_in}}}. \tag{3.10}$$

The gain for using the ReLU function is $\sqrt{2}$ and fan_in is the number of input units. This weight initialization layer is placed after the 1x1 convolution and before the expected ReLU activation function.

We opted for using the AdamW optimizer [102], since recently it has been shown that models trained with Adam may not generalize well. Thus, AdamW emerges as an improved version of Adam where weight decay is performed outside of the moving averages and it is only proportional to the weight itself. AdamW is based on Algorithm 2 in Appendix B [103].

The chosen loss function was the cross-entropy loss (Equation (3.8)), the same one as used for GregNet, since it is a widely used cost function for this type of application.

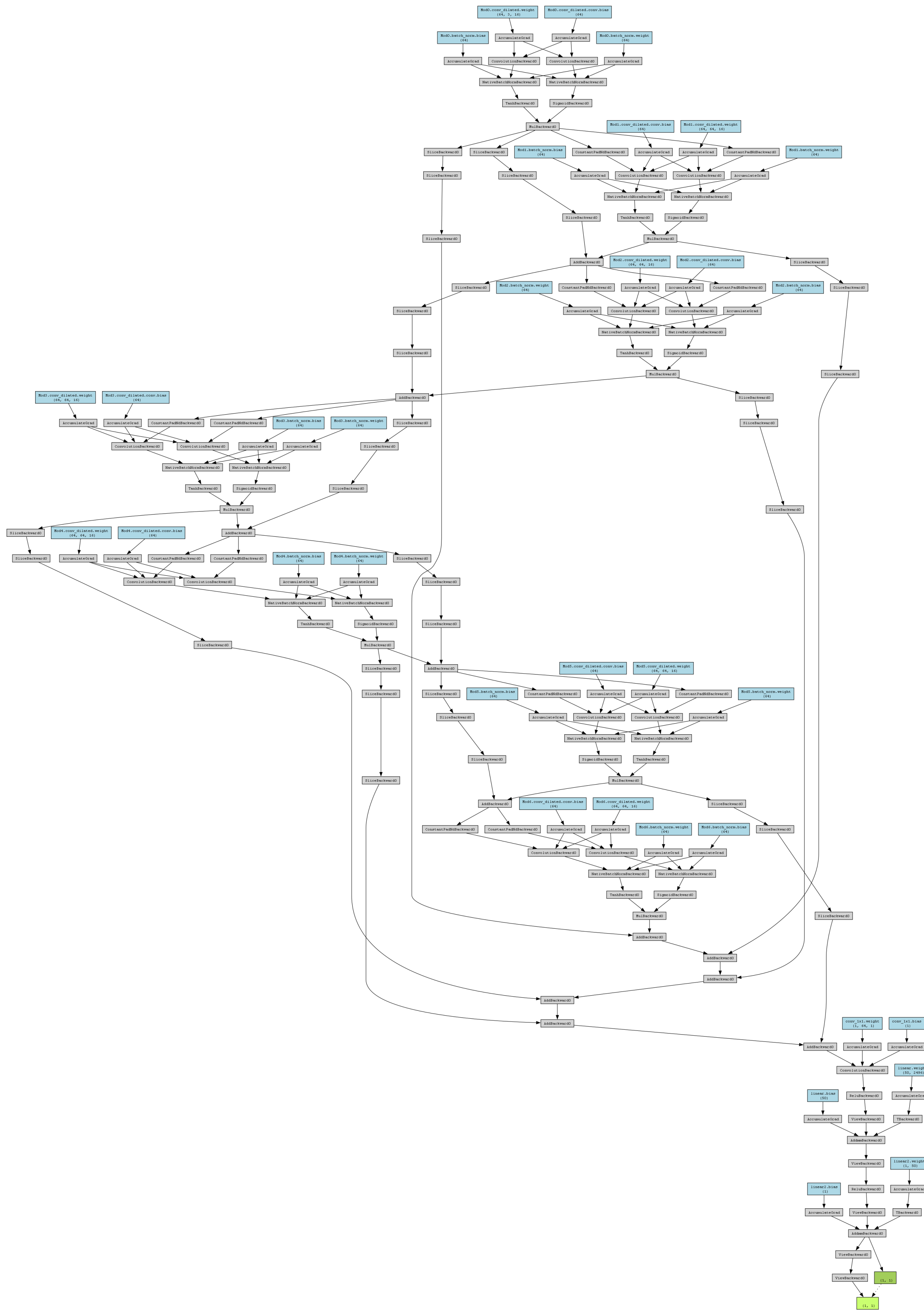


Figure 3.14: Full architecture of GWaveNet. The input goes through seven Module Wavenet blocks, described in Figure 3.13, and the residual outputs of each are saved and added at the end of the module blocks. After that, a 1x1 convolution is used to bring the number of channels to one, followed by a ReLU activation function. Next, two linear layers are applied, with a ReLU in between, and final lengths of 50 and one. Lastly, a sigmoid activation function is used to distribute the predictions between zero and one.

3.2.3 Hyper-parameter tuning

Having the NN architecture set, one still needs to find its optimal hyper-parameters for the application, as introduced in Section 2.2.3.2.1. This means deciding on hyper-parameters, such as the LR during training, what type of activation functions should be used where, how many filters should be used in the convolutions, or even what the minibatch size should be within the constraints of the available hardware. Hyper-parameters are everything to do with how the training process goes.

During the early stages of this project, the hyper-parameter optimization framework Optuna [104] was used. Optuna does a hyper-parameter search automatically, given certain constraints, based on the ML algorithm of choice. It also has pruners implemented, making unfruitful trials be pruned without having to run the full algorithm. Even though Optuna is very helpful, with the amount and size of data we are using, training our models quickly started taking too long to be able to use Optuna, since its point is to try different combinations of hyper-parameters, to find the best one. After working with it for this study, we found that indeed Optuna is better-suited for ML models that take a shorter time to train, such as tree-based models or shallower NNs.

Thus, as a trade-off between high performance and computational complexity, the final hyper-parameters presented were found by hand, starting on the foundation that the trials with Optuna left us.

3.2.4 Training features

As mentioned in Section 3.1.2, our data was divided into training, validation, and testing portions. Our training procedure followed this division as well. The NNs were trained for a certain amount of epochs (iterations over the entire training set), where, for each, the NN saw and learned from the training set, and saw but did not learn from the validation set, i.e. the weights were not adapted according to it. Thus, every time the NNs saw the validation set, it was as if they were seeing it for the first time. This is done to control the performance of the NN and avoid overfitting. Overfitting is a term used for when a supervised learning algorithm fits too well on the data that it is seeing and learning from and does not perform equally on data that it is seeing for the first time. This can be observed through the loss function: as the training loss continues to decrease, the validation loss starts increasing, indicating that the model is not generalising well. After all of the epochs and once training is finished, the NNs were tested on a completely new set of data, the test set. This approach is used commonly in ML, as the validation set is used for hyper-parameter tuning purposes. The final results are, thus, based on the test set and are shown in Chapter 4 are all based on the performance of the NNs on this set.

To improve performance and avoid overfitting, our training procedure implemented the methods described in the following Sections.

3.2.4.1 Curriculum Learning

To push our NNs to be able to detect a signal as early as possible, curriculum learning [105] was used. Curriculum learning is a training feature where the NN first gets shown the easy cases and then gets shown

progressively harder cases to detect.

In this study, curriculum learning was implemented by first showing the NNs cases where we were closer to the merger of the BNSs in time, i.e. at higher frequencies, and then showing cases further and further away, i.e. at lower frequencies, with the specifications shown in Table 3.1. The standard procedure for using curriculum learning for GW searches is through starting with higher SNR and lowering it progressively [106], so this method, also implemented by Baltus et al. [68], is a new feature.

Thus, the NNs initially trained on 16000 samples and validated on 2000 samples from data set 1 for a certain amount of epochs. Then, the already trained NNs were trained on 16000 samples from data set 1 and 16000 samples from data set 2 and validated on 2000 samples from data set 1 and 2000 samples from data set 2, and so on, until, on the fifth run, the NNs were trained and validated on data from all data sets, 1, 2, 3, 4 and 5. At the very end, the NNs were tested on 4400 samples from each of the data sets.

3.2.4.2 Adaptive learning rate

During this study, it was verified that it was harder and harder for GWaveNet to not overfit as more data sets were introduced, and overfitting was happening much earlier. As such, an adaptive LR was introduced for the training of this model. The LR starts as 2×10^{-5} and drops as the runs go with $\mathrm{LR} = \mathrm{LR}_{\mathrm{initial}}/\#\text{data set}$, where the data sets go from 1 to 5.

3.2.4.3 Early stopping

To further avoid overfitting, two early stopping algorithms were implemented.

The first algorithm consists of initially giving an arbitrarily large number as the maximum number of epochs that the NN will train for, and then stopping the training earlier, depending on the performance of the NN on the validation set. In particular, the training would stop if the validation loss of epoch e, $J_{\mathrm{validation}}(e)$ was a minimum delta value, ε, away from the best (i.e. lowest, since we want to minimise the loss), $J^{\mathrm{best}}_{\mathrm{validation}}$ for more than or equal to n patience epochs, or if the validation loss was more than ε worse than the best validation loss for more than or equal to n patience epochs. Mathematically, we stop training if:

$$|J_{\mathrm{validation}}(e) - J^{\mathrm{best}}_{\mathrm{validation}}(e^{\mathrm{best}})| < \varepsilon \qquad\qquad (e - e^{\mathrm{best}}) \geq n. \tag{3.11}$$

For our NNs, $\varepsilon = 0.0001$ and $n = 2$, with the maximum number of epochs to train for set as 20.

The second early stopping algorithm consists of getting the best model post-training. For each run, even if the first algorithm did not stop the run early, this algorithm goes and picks the model at the epoch where the validation loss was the best (i.e. lowest) and saves that model, to then be loaded on the next run, and so on, until the fifth and last round, where the best model for that run is also saved, and the test set is tested on this best saved model.

3.3 Embedding a field programmable gate array with a neural network

The goal of embedding an FPGA with our pre-trained NNs is to have it running on low-latency inference mode, to detect the signals as early as possible. Furthermore, the environmental impact of an FPGA is significantly lower than a GPU, as well as its energy consumption.

Details about the FPGA, as well as the whole process needed to embed the NNs into it, can be found on Appendix D.

3.3.1 Specifications of the field programmable gate array

The FPGA used for the study is part of the Advanced Micro Devices (AMD) Kria KV260 Vision AI Starter Kit [107]. As such, we opted for using the software associated with it, Vitis AI [108], for the NN deployment.

The DPUCZDX8G intellectual property (IP) is a standalone module and can be used in any FPGA, having, however, been specifically designed for the Zynq UltraScale+ MPsystem-on-a-chip (SoC), included in the Kria KV260, and it is optimized for pre-built vision applications based on CNNs. A diagram of the multiple blocks that make up the DPUCZDX8G is shown in Figure 3.15. The DPU is composed of a high-performance scheduler module, a hybrid computing array module, an instruction fetch unit module, and a global memory pool module. The DPUCZDX8G IP is implemented in the programmable logic (PL) of the Zynq UltraScale+ MPSoC with direct connections to the processing system (PS). It requires accessible memory locations to read and store the data used for the NNs. A program running on the application processing unit (APU) also needs to connect with the DPU for data transfers. [109]

The hardware architecture of the DPUCZDX8G is outlined in Figure 3.16. After start-up, the DPUCZDX8G fetches instructions from off-chip memory to control the given program that was passed on to it. The on-chip memory focuses on more particular low-latency instructions, like buffer activation functions and intermediate feature maps. The PEs use the built-in multipliers, adders, and accumulators to further accelerate the applications. [110]

The DPUCZDX8G IP can be configured with different provided architectures, namely B512, B800, B1024, B1152, B1600, B2304, B3136, and B4096. These architectures are related to the parallelism of convolutions, with the most recent, fastest, and most flexible one being B4096, at 4096 peak operations per cycle, and thus the one used for this study. [111]

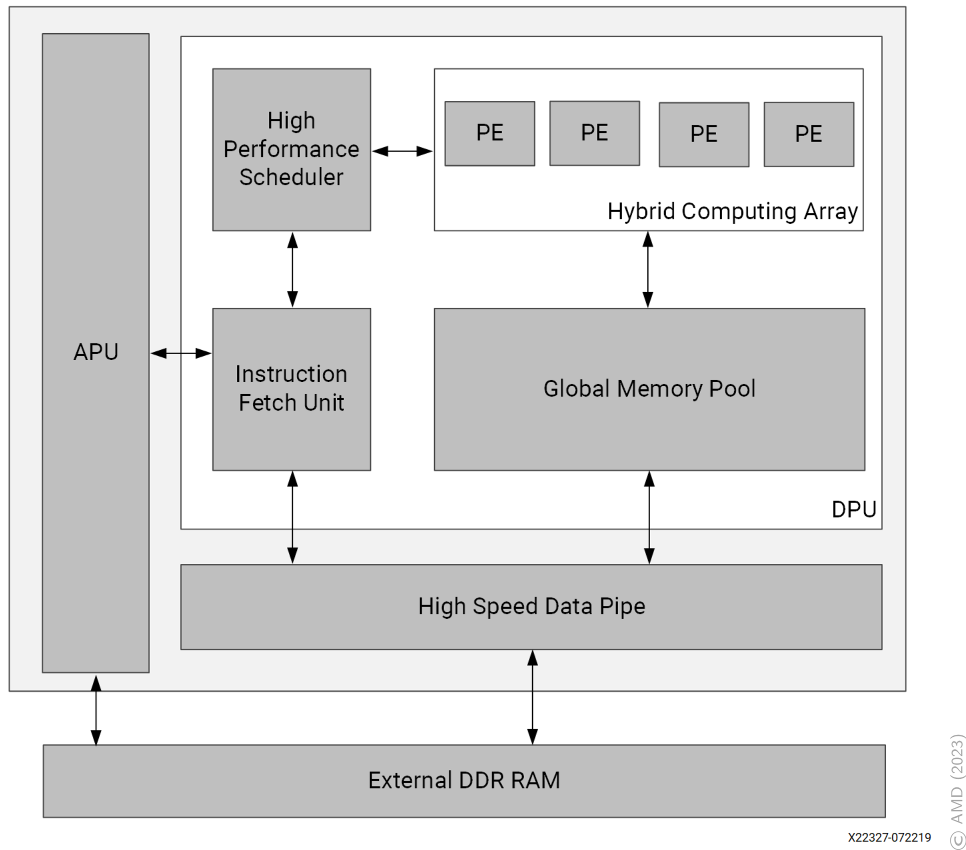


Figure 3.15: Block diagram of the DPUCZDX8G, with its distinct components, namely the APU, the DPU and the processing engine (PE). Taken from [109].

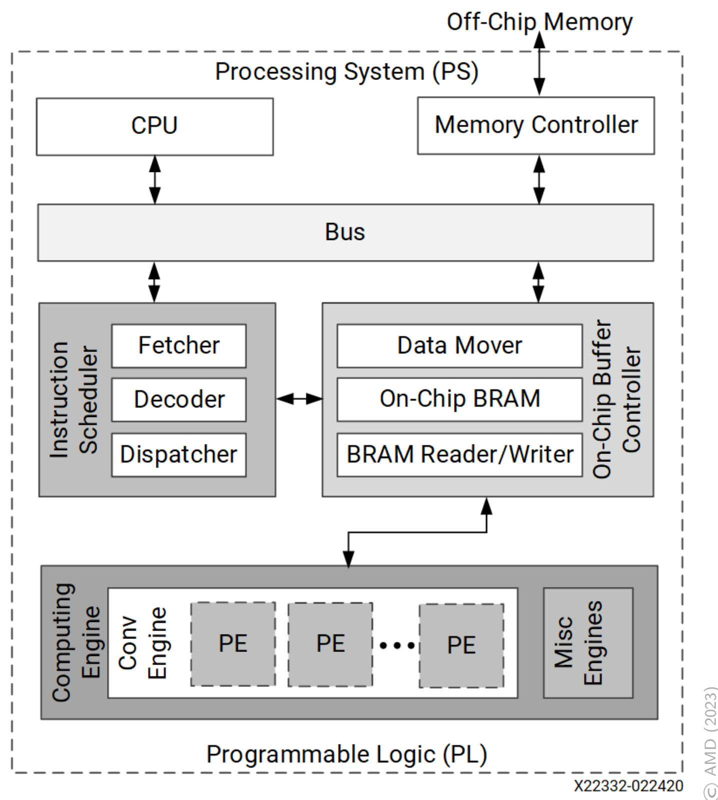


Figure 3.16: Hardware architecture of the DPUCZDX8G. Taken from [110].

3.3.1.1 Comparison with state-of-the-art hardware

The CPU used for the inference testing in this study was the AMD EPYC 7551P 32-Core Processor CPU [112]. At launch date, it cost around 9 times [113] as much as the AMD Kria KV260 [107] and it has an expected lifetime of around 5 to 10 years. The GPU used for inference was the NVIDIA GeForce GTX 1080 GPU [114]. At launch date, it cost around three times [115] the amount of the AMD Kria KV260. The expected lifetime is 3 to 5 years. However, both CPU and GPU technology quickly advance. These last-generation devices will become obsolete even before their lifetime is over, unable to compete with other newer devices. In fact, the GPU used for inference is now, 8 years after its launch, considered obsolete and the one we used will be replaced soon. The Kria KV260, on the other hand, has an expected lifetime of around 10 to 15 years. Additionally, because it is customizable, it is easy to make changes to it so that it is always competitive and does not quickly become obsolete. The carbon footprint of using FPGAs instead of these traditional processing units is significantly lower, not only because it uses less energy when operating, but also thanks to their effective lifetime.

3.3.2 Inspecting model compatibility

To run our NNs on the FPGAs, we first need to prepare the models, by first checking if they will be compatible with the FPGA and the chosen architecture, as described in the present Section, and then quantizing and compiling them, as outlined in Section 3.3.3.

To indeed check if our models are compatible with the DPU, we pass our models through the model inspector provided by AMD, with some slight changes to their code. Once again, all details are outlined in Appendix D.

The first modification needed for the models to work on the FPGA is that all layers that are based on 1D transformations be passed onto their 2D versions. This is such because the Kria KV260 is designed for vision applications and as such it is only prepared for operations with 2D samples. The 2D version of the NNs will henceforth be referred to as their original names with "2D" at the end. This could be reworked in future work by redesigning the FPGA hardware to suit these needs.

The output of the model inspector for GregNet2D and GWaveNet2D can be observed respectively in Figures 3.17 and 3.18. The black ellipses represent the input and output, the red ellipses represent the layers running on the CPU of the FPGA, and the blue ellipses represent the layers running on the DPU of the FPGA. We want to maximize the number of layers running on the DPU, since this is specifically designed for data processing and thus should perform inference of the NNs faster. However, and because the FPGA is also able to run processes on the DPU and the CPU at the same time, on top of being able to perform multi-threading, we are not necessarily concerned with making every layer run on the DPU.

Since our goal is both to have a good performance and a fast performance, changes were made to the NNs for more layers to be able to run on the DPU instead of the CPU, to make the process faster.

For GregNet, the first three layers, that deal with reshaping the data from 1D to 2D, are flagged as not able

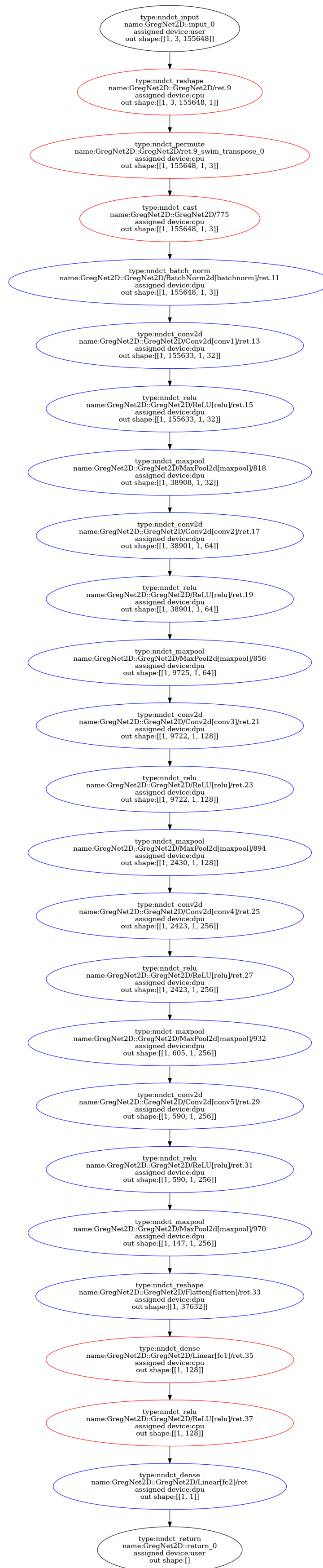


Figure 3.17: Output of the model inspector for the 2D version GreNet, describing where each layer is run.

to run on the DPU, as well as the first linear layer and its activation function. Unfortunately, we are not able

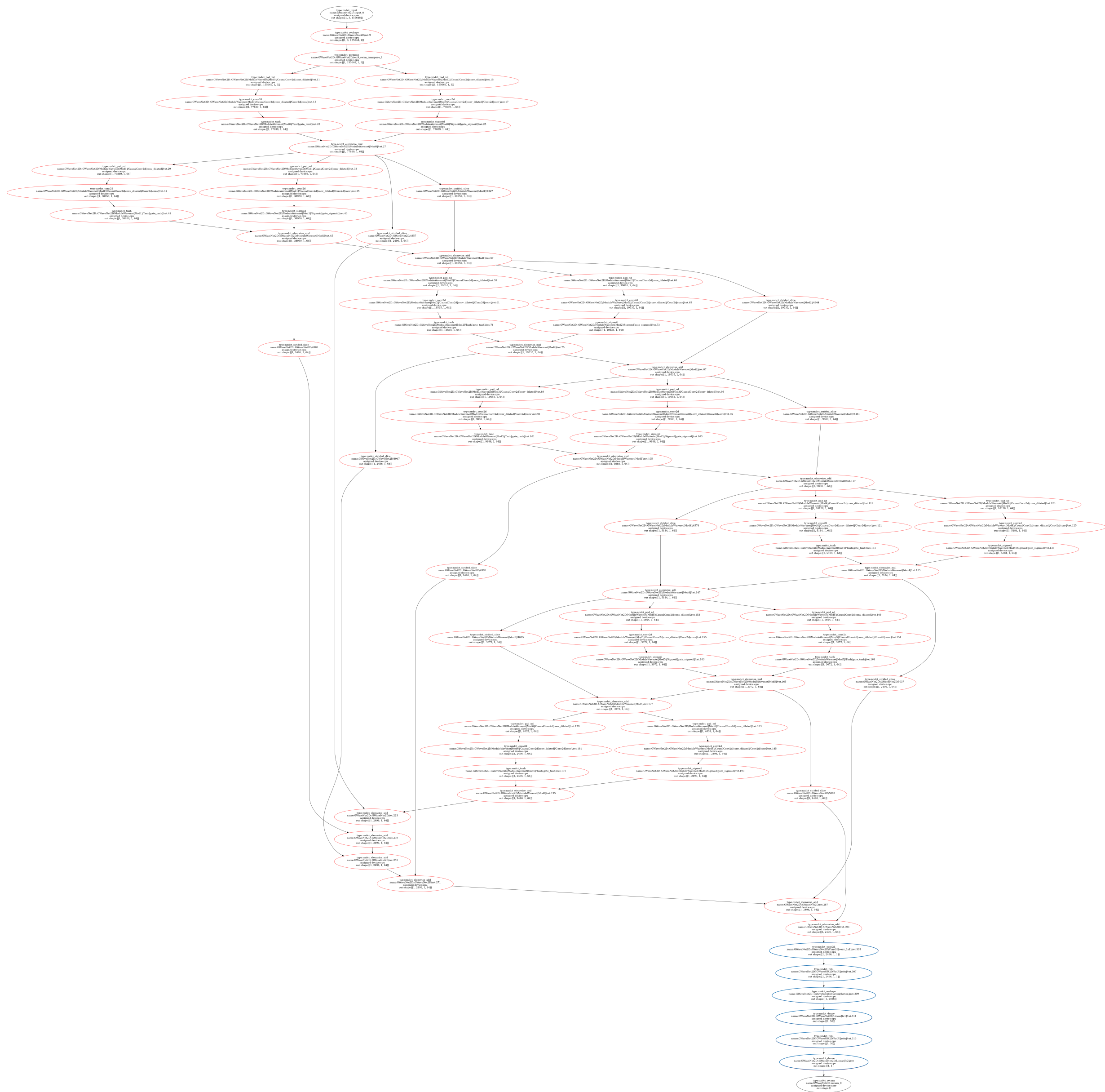


Figure 3.18: Output of the model inspector for the 2D version of GWaveNet, describing where each layer is run. Modified according to compiler instructions.

to convert the reshaping layers to run on the DPU and they are essential to the model, as the DPUCZDX8G IP does not have support for 1D layers. However, we were able to resolve the second conflict by adding a 1x1 convolution layer before the flattening layer to cut the amount of channels in half. The output of the model inspector with the new architecture with these modifications is shown in Figure 3.19.

For GWaveNet, multiple layers are flagged as not being able to run on the DPU, mainly due to their size (derived from the exponential growth in dilation of the convolutions). The first challenge we tackle is the artificial padding used to implement the causal convolutions: the padding mode used, *constant*, is not allowed to run on the DPU, so we change it to the only mode allowed to run on it: *replicate*, where padding is set as symmetric instead of constant. After that, we can notice that the sigmoid and tanh activation functions are also not running on the DPU. This happens because the functions are too non-linear to run on int8 in the DPU, and as such we switch the activation functions respectively to `Hardsigmoid` [116] and

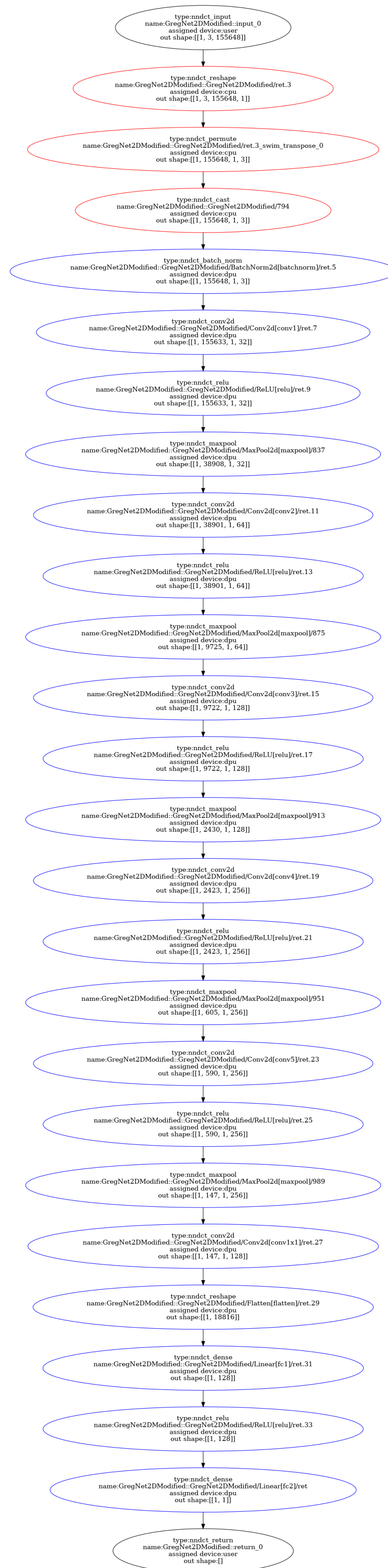


Figure 3.19: Output of the model inspector for the final modified version of GregNet, describing where each layer is run.

`Hardtanh` [117], which are more linearized versions of the old activation functions. Still, the `Hardtanh` activation function was not able to run on the DPU, and, looking at the documentation for the supported functions [118], the `Hardtanh` function is transformed into Vitis AI's implementation of `ReLU6` anyways, so we set the previously tanh activation function as `ReLU6` [119], which has a similar shape. Lastly, some of the causal convolutions are not able to run in the DPU due to their size. As such, and after varying the parameters in many ways to find the closest to the initial implementation, we lowered the convolution kernel size to 2. Additionally, the sixth convolution was still too large to run, so we cut the number of filters it was working with in half, to 32. Because of this, and because we are working with an RNN, all of the outputs of every ModuleWavenet need to have the same amount of channels. As such, we introduce a 1x1 convolution after the gated activation to lower the channels after the other convolutions from 64 to 32. The output of the model inspector with the new architecture for GWaveNet with these modifications is shown in Figure 3.20. Still, with these modifications, the reshaping layer changing the input from 1D to 2D is not able to run on the DPU, similarly to GregNet, and a layer cropping the input for the skip-connection is also not feasible to run on the DPU.

Performance and timing results for all models, before and after the modifications, are presented in Chapter 4.

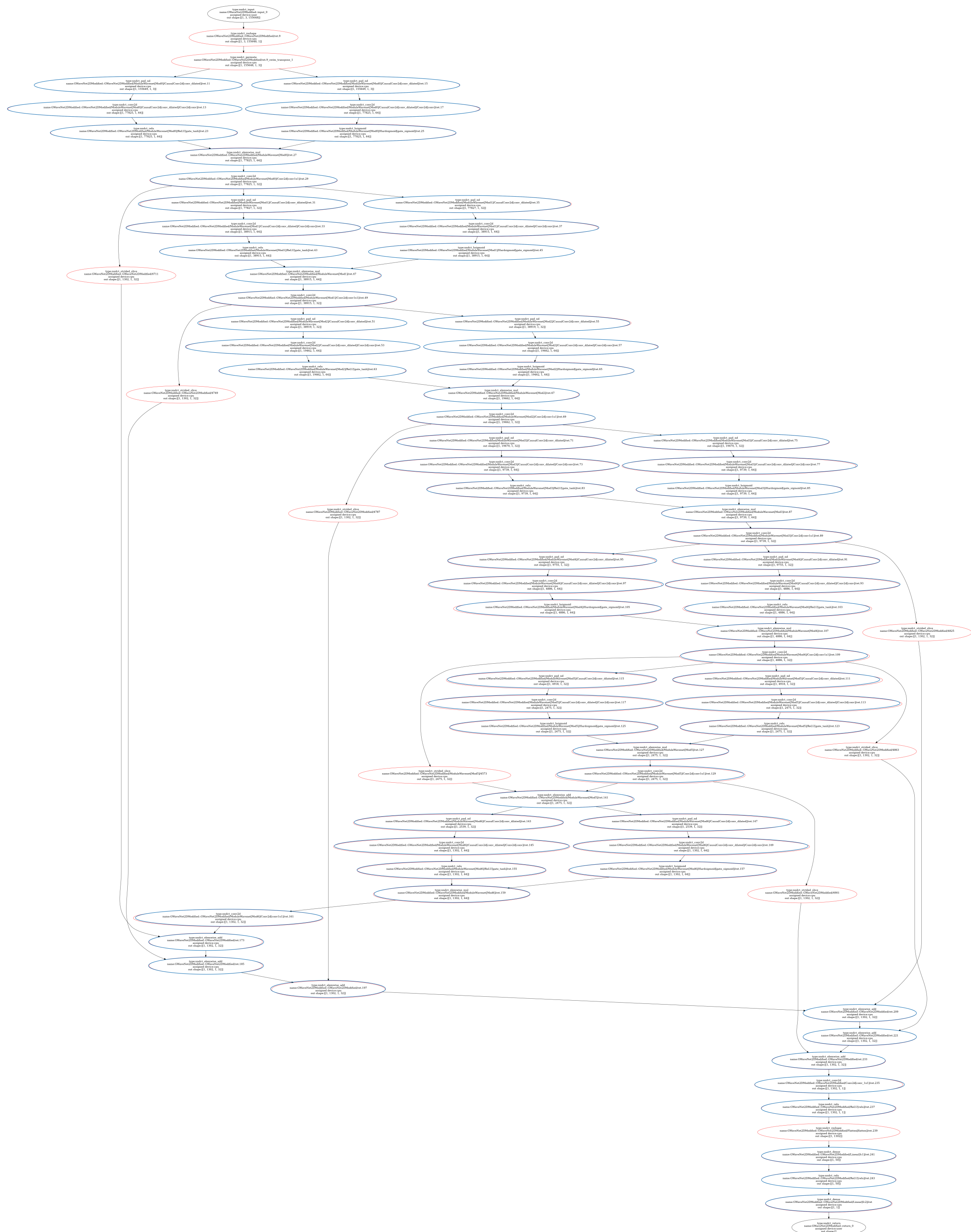


Figure 3.20: Output of the model inspector for the final modified version of GWaveNet, describing where each layer is run. Modified according to compiler instructions.

3.3.3 Post-training quantization

Quantization is a process that turns the NN's weights from floating point to lower bitwidths to save memory and accelerate inference. The memory resources of an FPGA are very limited compared to a standard CPU or GPU. As such, to compile and embed the NN models into the FPGA, we need to quantize them.

We chose to explore post-training quantization (PTQ), as it was what made the most sense for the size and quantity of data used for training, but the notion of quantization-aware training (QAT) might be worth exploring in future work.

In particular, in this study we quantize our float32 weights into int8. It has been shown that this type of quantization reduces the model size and the memory bandwidth requirements 4 times, as well as being able to accelerate computations 2 to 4 times [120]. However, this may affect the model's performance negatively, notably more towards smaller and shallower models.

A quantization operator $Q(r)$ following a uniform quantization strategy is of the form:

$$Q(r) = \text{Int}(r/S) - Z, \tag{3.12}$$

where r is the target weight to quantize, S is a scaling factor and Z is an integer zero point. The quantization strategy can be symmetric or asymmetric, depending on the choice of the scaling factor S and the zero point Z. The scaling factor S is often defined as:

$$S = \frac{\beta - \alpha}{2^b - 1}, \tag{3.13}$$

where $[\alpha, \beta]$ is the clipping range and b is the quantization bit width, and $b = 8$ in our case. Choosing the clipping range is referred to as calibration, and it is a process that Vitis AI does automatically. A straightforward and asymmetric strategy is choosing α and β to be the minimum and maximum of the range of values to quantize. Another popular choice, that takes the symmetric route, is making α and β symmetric, where $-\alpha = \beta = \max(|r_{min}|, |r_{max}|)$, and this naturally sets Z to 0. [121]

Most of the existing studies employ linear quantization, which usually significantly affects a model's performance. Additionally, weight distribution is usually non-uniform. This prompts non-uniform non-linear quantization strategies to be better. [122]

Power-of-two quantization is shown to be a robust logarithm-based quantization strategy and, in particular, it is the strategy that the Vitis AI quantizer uses [123]. It arises from the following equation:

$$y = w^T x \simeq \sum_{i=1}^{n} w_i \times 2^{Q(\log_2(x_i))} = \sum_{i=1}^{n} \text{Bitshift}(w_i, Q(\log_2(x_i))), \tag{3.14}$$

where $y = w^T x$ is a matrix multiplication mimicking what happens in fully connected or convolutional layers of NNs, with w denoting the weights and x the input, and $Q(r)$ is the quantization operator introduced in Equation (3.12) [124]. From this equation, it becomes clear that the power-of-two quantization strategy is based on a simple, bitshift operation, which is native to the encoding in digital hardware, and thus fast to perform. Furthermore, logarithmic representation natively encodes data with a large range in fewer bits than linear fixed-point representation.

Post-quantization accuracies of the possible models (2D and 2DModified versions) are presented in Chapter 4, as well as inference testing.

Chapter 4

Results

In this chapter, we delve into the final results of this study, including what the training of the NNs looks like in Section 4.1, how the NNs perform in Section 4.2, how much time they take to predict in Section 4.3 and how much energy they consume in Section 4.4. We will compare between GregNet, GWaveNet and their 2D and 2DModified counterparts.

4.1 Training results

We are first interested in observing how the NNs performed during training and validation and if our training features were effective. This will give us a first overview of the performance of the models.

Figure 4.1 shows the training and validation losses over the full number of epochs the NNs trained for, and Figure 4.2 shows the corresponding accuracies for a default decision threshold of 0.5. The decision threshold is the threshold that the NNs take into account for choosing between a negative (0) or positive class (1). In particular, during training, we monitor the accuracy for a default decision threshold of 0.5 as the performance of the NNs is only really known at the end of training. Additionally, the decision threshold can be chosen with a specific goal in mind, which we explore later on in this Chapter.

The vertical lines point out when a new data set was added into training, signalling a new curriculum learning stage, with new data farther away from the merger added into the training and validation sets. It is particularly obvious in Figure 4.2 that adding new data significantly worsens the performance of the NNs, for both the loss and accuracy metrics, as expected when using curriculum learning, as explored in Section 3.2.4.1. Usually, the performance starts getting better again as the epochs go on, as learning converges. This is the case for ideal curriculum learning, but we can verify that, in fact, this is only true for GWaveNet, as GregNet actually gets worse as the epochs go on, in particular for the last two curriculum learning stages.

For the loss plots in particular, we are interested in seeing this measure minimized. Comparing the losses between the two models, we can hypothesise that GWaveNet has a better performance from the very first epoch, never reaching a loss as high as GregNet, which may mean that the GWaveNet model is better at

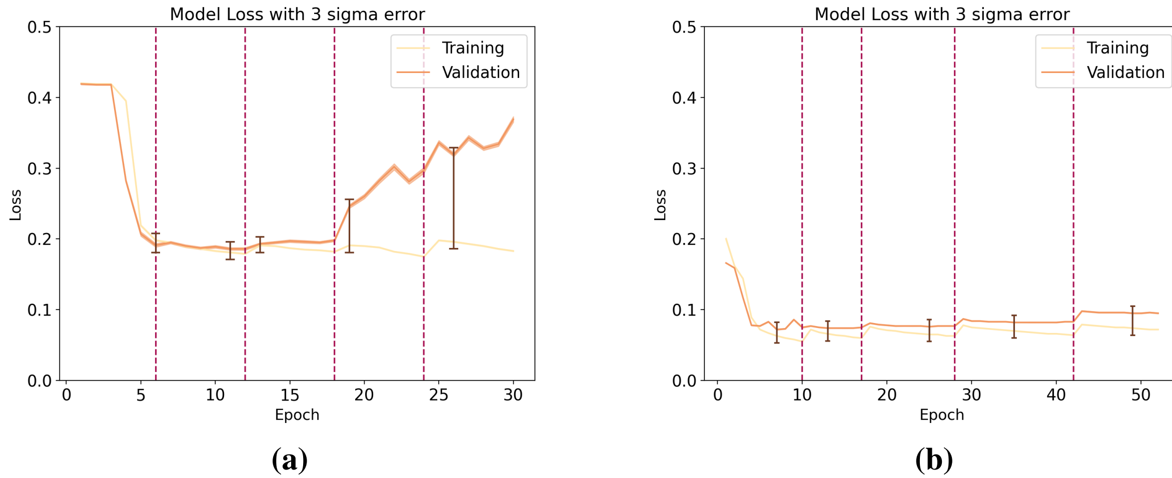


Figure 4.1: Training and validation losses over number of epochs for **(a)** GregNet and **(b)** GWaveNet, with the corresponding 3σ error for each. The burgundy dashed lines signal the end of a curriculum learning stage, with more data from a new data set being added after them. The solid brown short lines signal the best-performing epoch in each stage.

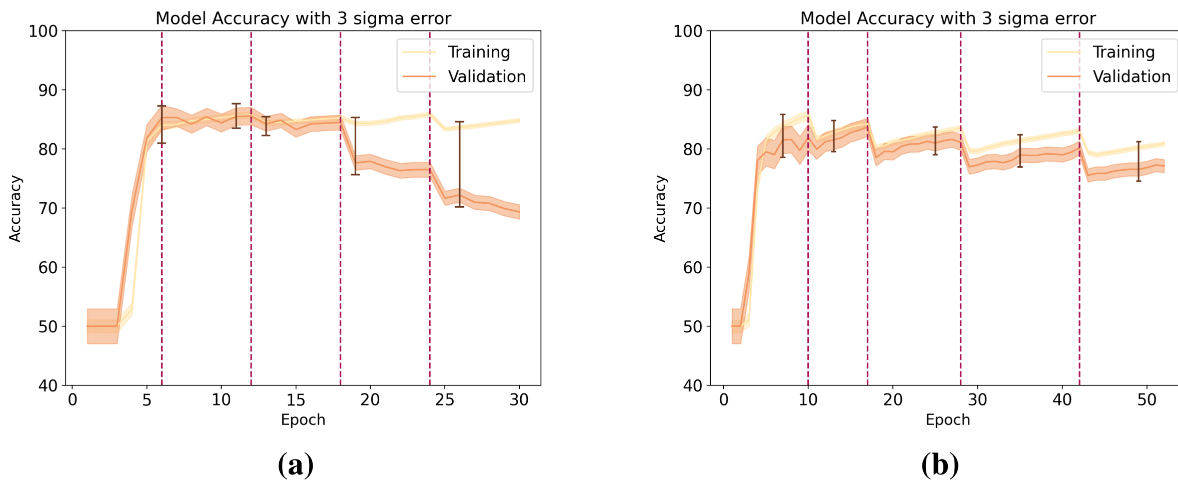


Figure 4.2: Training and validation accuracies over number of epochs for **(a)** GregNet and **(b)** GWaveNet, with the corresponding 3σ error for each. The burgundy dashed lines signal the end of a curriculum learning stage, with more data from a new data set being added after them. The solid brown short lines signal the best-performing epoch in each stage.

learning the patterns in the data independently of the training features used. However, it is important to point out that the loss functions for each model are weighted differently. Still, we can observe that GWaveNet does a better job at consistently minimising its loss function.

As described in Section 3.2.4.3, a large number of epochs was given as the maximum number of epochs the training could go on for for GWaveNet, but with the first early stopping algorithm we set, the training did not go on for the full length of those epochs, and this can be verified by the fact that the different stages of curriculum learning trained for different numbers of epochs. This does not, in turn, happen for GregNet, since the same training features as in the original paper were used, and the model trained for exactly six epochs for each curriculum learning stage. This feature allows the model to train efficiently, as it allows

GWaveNet to train for the necessary epochs to lower the loss as much as possible for each curriculum learning stage, whilst stopping the training early if it is no longer useful, saving time and computational resources.

From our second early stopping algorithm, we choose the best epoch out of the ones that GWaveNet trained for in each curriculum learning stage and use the weights from that epoch to then go on to add more data and, at the end of training, we choose the best epoch out of the last stage and test the test set in the NN set with those weights, as described in Section 3.2.4.3. In Figures 4.1 and 4.2, the brown vertical lines point out the best epoch from each curriculum learning stage. For GWaveNet, the aforementioned early stopping algorithm picks up the weights from those epochs to carry on training. However, for GregNet, that is not the case, as we followed the same training standards as in the original work, and we can see that for most the curriculum learning stages, most of the training is not effective and it turns into overfitting. Overfitting so soon after a new curriculum learning stage is reached may point to a LR that is too high, and GregNet might have also benefited from an adaptive LR.

Indeed, it is easy to see that GWaveNet's training is much more stable than GregNet's, as the "bumps" one can see in GWaveNet's training are simply due to the adding of new data, whereas the "bumps" in GregNet's training are due to the model overfitting massively and its unstable training.

It is also useful to take a look at how the loss evolves in terms of the number of floating point operations (FLOPs) for each of the NNs and compare how it evolves for each with the amount of calculations, shown in Figure 4.3. To do this, we use the Flop Counter [125] in the `fvcore` library [126].

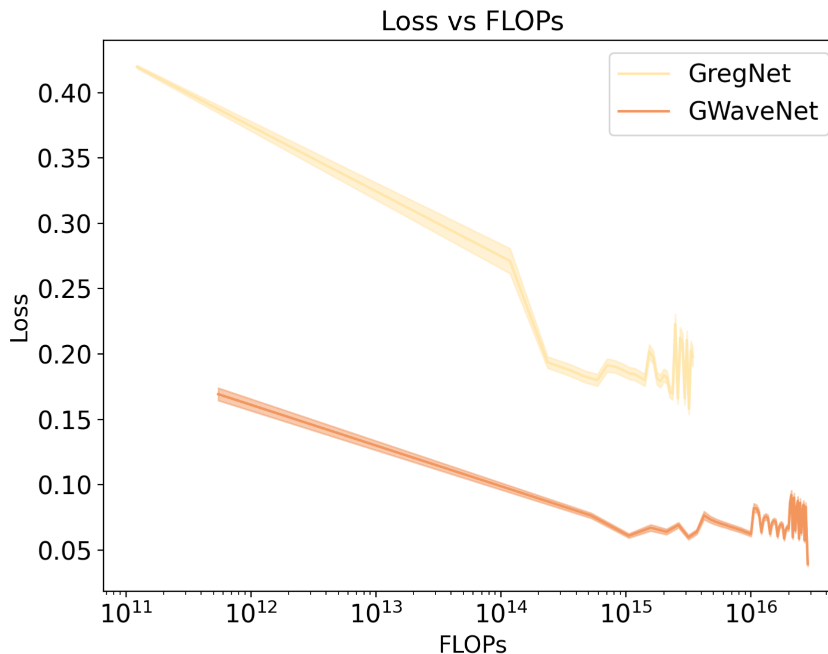


Figure 4.3: Training losses over number of FLOPs for GregNet (cream) and GWaveNet (orange). We present the average loss per epoch for the cumulative amount of FLOPs and the corresponding 3σ error. The vertical axis is in logarithmic scale.

Once again, GWaveNet's loss not only is lower from the beginning, but it also varies less throughout train-

ing. However, GregNet does make fewer FLOPs, which was already established in Section 3.2, not only because it trains for fewer epochs, but also because it is a shallower model, which can be verified by the curve for GWaveNet starting at higher FLOPs. This will later also impact the comparison between the efficiency of the models.

4.2 Performance results

The goal of this proof-of-concept study is to evaluate how these methods would work in a search. Hence, we want the number of false positives (FPs) to be as low as possible.

Let us first take a look at how the probabilities output by the NNs compare to the ground truth class in Figure 4.4.

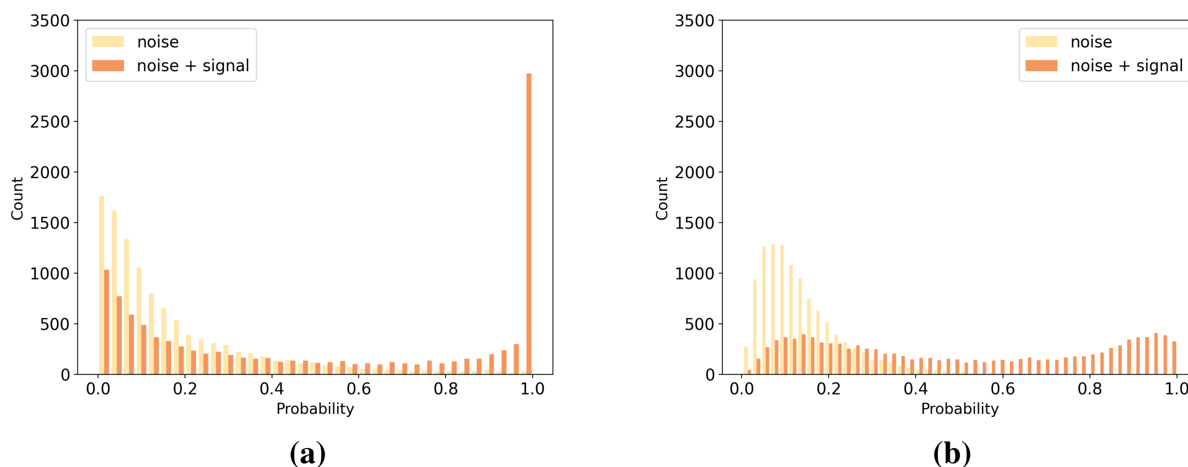


Figure 4.4: Probability distribution for both noise and signal classes for **(a)** GregNet and **(b)** GWaveNet.

GregNet appears to be more confident in its predictions, evidenced by having its probabilities distributed more towards the extremes of the plot. However, even though it works mostly as a yes classifier, evidenced by the big peak next to 1.0, it misses many signals. Furthermore, its distribution of probabilities of the negative class spans a big part of the plot, with significant counts until around 0.9. GWaveNet, on the other hand, is more conservative in the sense that it prefers to be correct about its negative class predictions, with these ceasing before 0.5. The predictions for the positive class are quite spread along the plot, which is likely due to the nature of the individual signals. This will be addressed later when we look deeper into the nature of the signals. Furthermore, GWaveNet is not saturating on the positive class, giving it more decision power.

At the default decision threshold of 0.5, the false alarm probability (FAP) varies as shown in Figure 4.5 for the different data sets along the PISNR, whose distribution was shown previously in Figure 3.3 and is the reason why the points are not evenly spaced throughout the horizontal axis.

No trend is really noticeable in these plots, but we can already see that for both models, the FAP is somewhat low for the default decision threshold, never surpassing 20%, as the models' losses are biased towards the

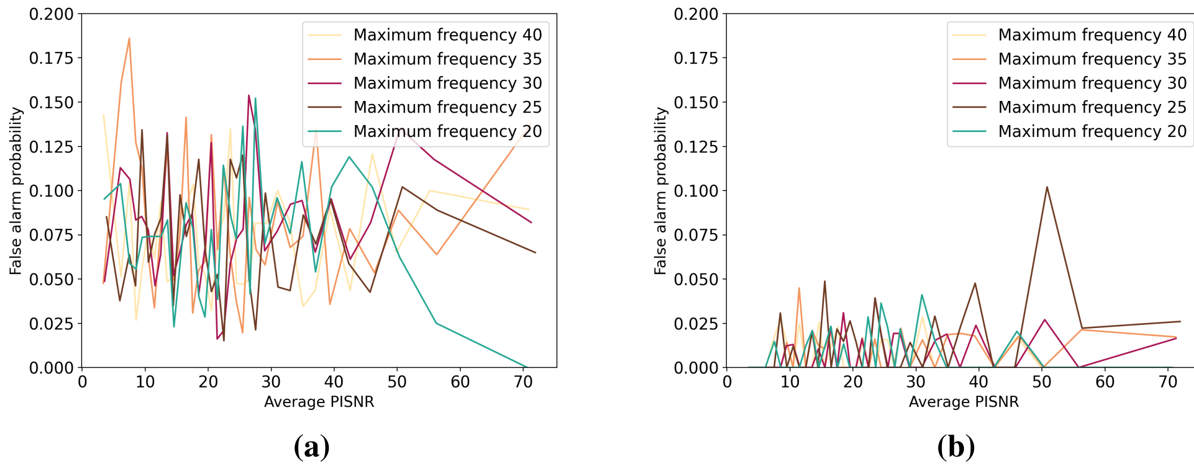


Figure 4.5: FAP over the PISNR for the different data sets for **(a)** GregNet and **(b)** GWaveNet, for a default decision threshold of 0.5.

negative class during training. If the losses were not biased, these FAPs would have been expected to be much higher, and, yet the overall accuracy would likely be higher. That is not, however, the purpose of this study: we want the accuracy to be high, but not at the cost of a higher FAP. When comparing the two models, GWaveNet has lower FAPs in general, never really even reaching the mean FAP for GregNet.

Let us look at both the FAP and the true alarm probability (TAP) for various decision thresholds around the default decision threshold of 0.5 for each of the models in Figure 4.6, to see how the differences between TAP and FAP compare.

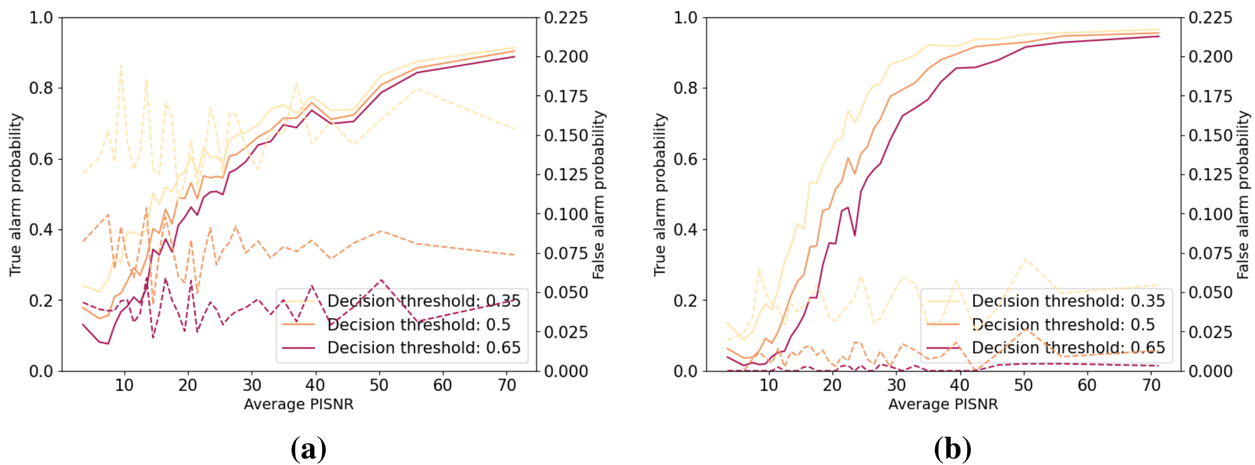


Figure 4.6: TAP (solid line, left vertical axis) and FAP (dashed line, right vertical axis) over the PISNR for different decision thresholds for **(a)** GregNet and **(b)** GWaveNet.

The most noticeable difference between the two models is that the FAP is much higher for GregNet than for GWaveNet, for all considered thresholds. There is a general upward trend for both models in the relation between the TAP and the PISNR. This was expected, as it means that the signals that were already louder before their merger are easier to detect. Once again, GWaveNet shows evidence of its better performance, with its TAP being better overall, and in particular at higher PISNRs, converging to 1 quicker, as opposed

to GregNet that indeed ends up never converging to 1. However, GregNet is better at predicting for lower PISNRs.

Even with an already somewhat low FAP, we want it to be much lower and stable so that we can properly compare the two methods. As such, we set the FAP to 1%, which we consider to be a reasonable constraint for this study, as it is the same one used in the study by Baltus et al. [68]. This constraint translates into a decision threshold of 0.86 for GregNet and 0.46 for GWaveNet, as shown in Figure 4.7.

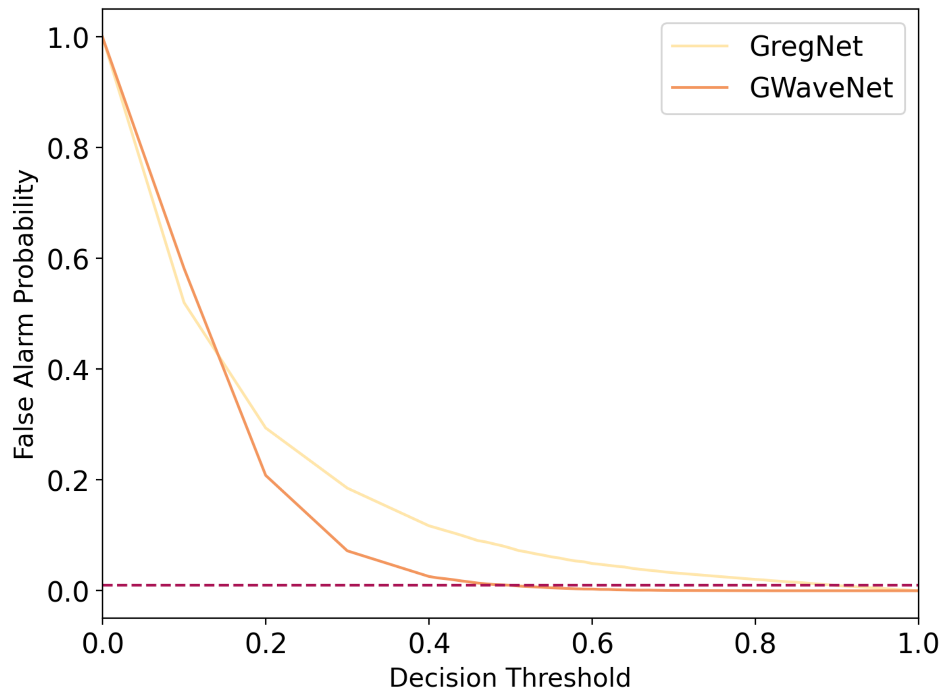


Figure 4.7: FAP over the decision threshold for GregNet (yellow) and GWaveNet (orange). The burgundy dashed line represents the FAP at 1%, and it intercepts the curves at 0.86 and 0.46 respectively.

As can also be interpreted from Figure 4.4, GWaveNet has a lower FAP for lower probabilities. In fact, GWaveNet's FAP tends to low values so quickly that we could set the FAP even lower, but we choose to keep it at 1%, in order to be consistent with the original study. The fact that GWaveNet settles at a lower decision threshold means the model can better differentiate between positive and negative classes, as opposed to when the threshold is higher and the predictions are closer together.

A good overall measure of compromise between the FAP and the TAP is the receiver operating characteristic (ROC) curve, and we can see the difference of the ROC curves between the two main models in Figure 4.8.

The ideal ROC curve is a right angle at the top left, where for a FAP of 0%, we have a TAP of 100%. So, the higher the curve, the better the performance. Once again, we are able to confirm that GWaveNet indeed has a better performance. In particular, since we are interested in having a low FAP, we should bring out attention to the left-most part of the plot, where, still, GWaveNet outperforms GregNet by close to 20%.

Setting the FAP at 1%, we can take a look at how the TAP over the PISNR for the different data sets looks in Figure 4.9.

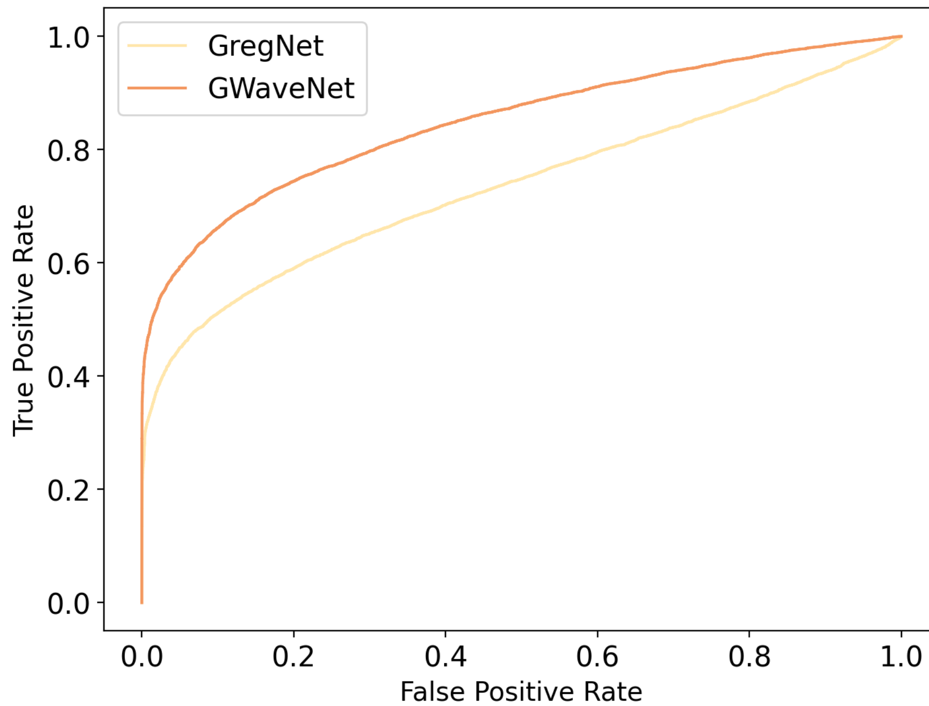


Figure 4.8: ROC curve for GregNet and GWaveNet.

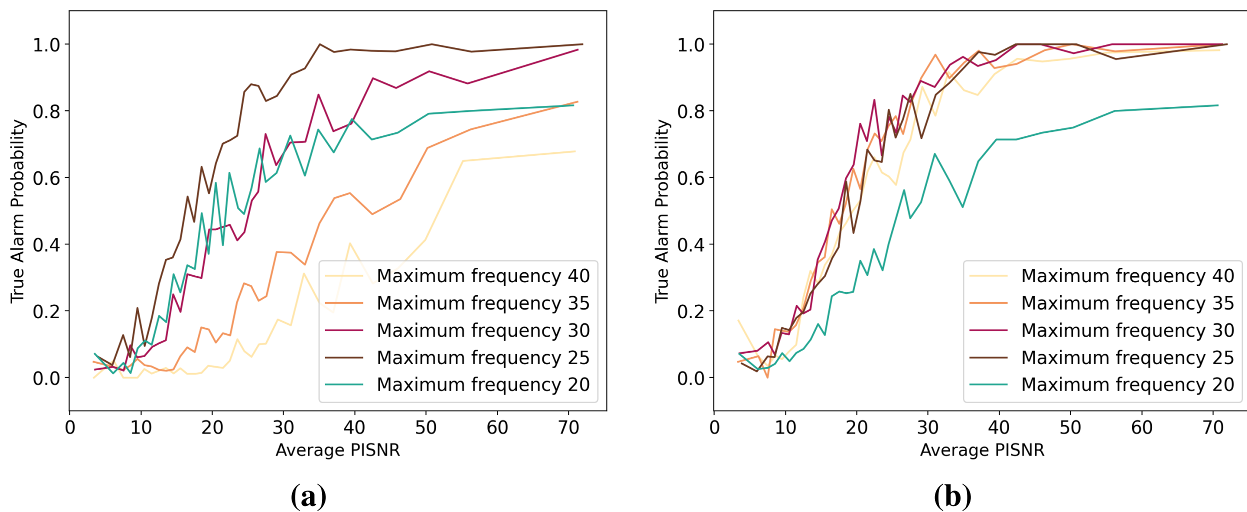


Figure 4.9: TAP over the PISNR for the different data sets for **(a)** GregNet and **(b)** GWaveNet, for a set FAP of 1%.

We can, again as in Figure 4.6, see a general upward trend in both the models between the TAP and PISNR. GregNet appears to have learned the most difficult cases better than the easier ones, pointing to it having forgotten the easier cases, even though they were still fed to it in the last curriculum learning stages. Indeed, GWaveNet's residual blocks seem to have taken care of that, as the same does not happen for its predictions. We can see, however, that the most difficult data set was not learned as well as the others, which is expected, since the NN saw these examples less often than it did the others, and they are indeed harder examples to detect.

For one last in depth look at how the NNs perform, we present the confusion matrices, representing the

percentages of true positives (TPs), false negatives (FNs), FPs and true negatives (TNs), produced by each model for their threshold at FAP = 1% in Figure 4.10.

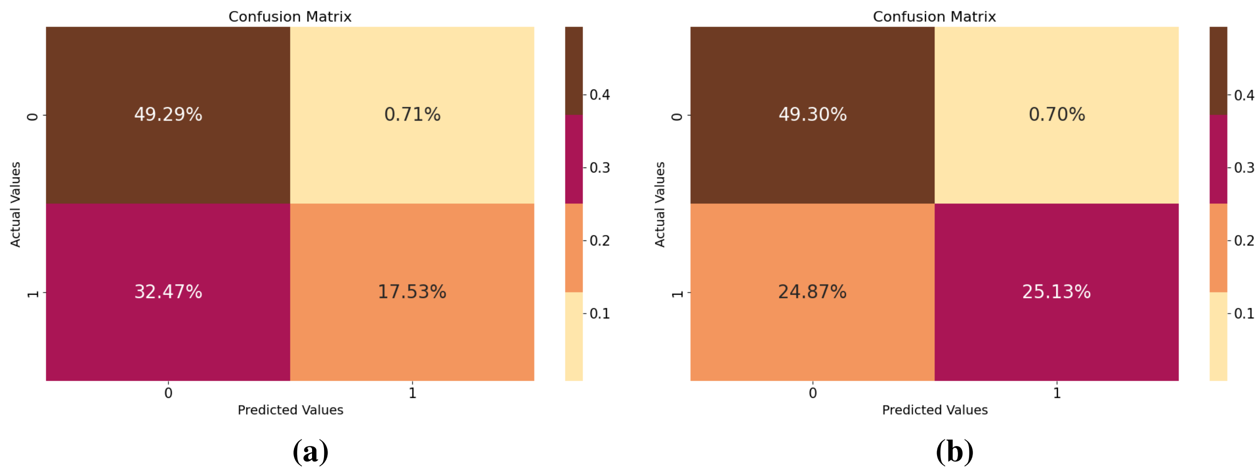


Figure 4.10: Confusion matrices for **(a)** GregNet and **(b)** GWaveNet, for a set FAP of 1%.

The ideal confusion matrix is a diagonal matrix, and, as such, we are looking for a confusion matrix with most of its values along the diagonal. For both models, the NNs are very good at predicting the negative class, not only because the loss is biased towards it during training, but also because the FAP was fixed at 1%. GregNet has more wrong predictions for the positive class than right predictions, which does not happen for GWaveNet, even though its ratio is not ideal either. It is interesting to point out once again that the decision threshold for GWaveNet is lower than GregNet's, meaning that it accepts more probabilities as the positive class, which could have meant that more FPs would appear, but it seems that the NN has learned the data well enough for that to not be the case.

These numbers of TPs, FNs, FPs and TNs translate into the accuracies shown in Table 4.1, where the accuracies also for the 2D and 2DModified counterparts of the models are shown, as well as their accuracies after quantization is performed.

Model	Test accuracy (before quantization)	Test accuracy (after quantization)
GregNet	66.81%	-
GregNet2D	**69.50%**	69.37%
GregNet2DModified	66.70%	65.07%
GWaveNet	76.22%	-
GWaveNet2D	79.42%	77.28%
GWaveNet2DModified	**71.72%**	72.42%

Table 4.1: Test accuracies of all models for a FAP set at 1%, tested on a test set of 2200 samples per class per data set, for a total of 22000 samples, except for the GWaveNet2D and GWaveNet2DModified post-quantization models, which were tested on a total of 5000 samples, equally distributed over the classes and the data sets, due to timeout constraints

.

There is an around 2 percentage points difference between the original GregNet model and its 2D counterpart, and the same happens for GWaveNet, with the 2D versions outperforming the 1D ones. The decision threshold for GregNet2D is moved lower than GregNet's for the FAP set at 1%, standing at 0.82, but on

the other hand, the decision threshold for GWaveNet2D is higher than GWaveNet's, at 0.54. If only GregNet was affected, these differences would most likely be due to the oscillations in training, since the LR applied to GregNet is high and, as we saw in Section 4.1, its training is not very stable. As such, and also because NNs are probabilistic models, the training of GregNet2D would have likely been a lucky run, and this difference would not due to changing the operations from the original 1D ones to their 2D counterparts. However, because the same is true for GWaveNet, it suggests that indeed changing the layers from 1D to 2D makes a difference. 2D layers are more widely used in the field of ML, as the applications in vision are broad, which may lead to `PyTorch`'s implementations of them being more optimized than their 1D counterparts. This might result in better numerical stability, which leads to higher accuracy. Furthermore, because the training and testing were performed in an Nvidia GPU, which is likely optimized for these 2D operations, once again, due to their common use in image processing, this might also contribute to the numerical stability and consequent higher accuracy.

Keeping this in mind, and indeed hypothesizing that the 2D operations are optimized, it seems that the 2DModified versions of GregNet and GWaveNet result in a drop in accuracy, due to the simplification of the original models. As such, and depending on how much the speed up for the FPGA ends up being, this might, or not, be worth it.

GWaveNet outperforms GregNet by almost 10% before quantization. In fact, the best-performing GregNet-like model, GregNet2D, is still worse than the worst-performing GWaveNet-like model, GWaveNet2DModified. As such, and taking also the results from the previous Sections into account, we can declare that the search for a new better-performing model in terms of accuracy was successful.

The quantization process appears to not affect the accuracies of the GregNet models in a very significant way, but GregNet2DModified, with its already lower accuracy, could benefit from QAT. For the GWaveNet2D model, quantization affects the accuracy similarly to what happens to GregNet2DModified, pointing out that the model might also benefit from QAT. However, it is interesting to notice that the test accuracy post-quantization of GWaveNet2DModified is higher than pre-quantization. This is likely due to the model pre-quantization suffering from overfitting, which gets lessened with the loss of precision as a consequence of the quantization process.

4.3 Timing results

In order to compare also the models' efficiency, the NNs were timed in testing 22000 samples in the CPU and GPU used for this study and in testing 1000 samples in the FPGA used for this study. The results are presented in Table 4.2. The library used for inference testing in the CPU and GPU was the `pyJoules` library [127] and in the FPGA we used the standard `time` library. The tests for the CPU and GPU were performed on the Nikhef cluster, with the machines being reserved for our inference testing to ensure a fair comparison. As such, the results for the CPU are shown for using the full 32 cores of our inference CPU. The tests for the FPGA were done using 10 threads, as it was found to be the optimal number of threads.

The GregNet and GWaveNet original models were not tested for inference on the FPGA, as their 1D oper-

Model	Time (ms)		
	CPU	GPU	FPGA
GregNet	27.53 ± 3.69	21.75 ± 1.89	-
GregNet2D	29.58 ± 0.10	20.90 ± 1.56	84.54 ± 0.10
GregNet2DModified	29.54 ± 0.11	24.87 ± 1.65	84.28 ± 0.10
GWaveNet	154.75 ± 0.38	33.60 ± 2.76	-
GWaveNet2D	85.28 ± 2.87	29.41 ± 1.77	-
GWaveNet2DModified	65.54 ± 3.67	26.02 ± 3.25	-

Table 4.2: Time it takes for each of the models to predict one sample, averaged over 22000 samples for the CPU and GPU and over 1000 samples for the FPGA, including the time it takes to load the data, with the corresponding 3σ error after having tested around 10 times.

ations are not able to be compiled to run in the FPGA. Furthermore, we found that the 2D and 2DModified versions of the GWaveNet model were also not able to be compiled to run on the FPGA. We believe this is so because of the lack of support for the padding needed for the causal convolutions as, even though it is stated as supported [118], it seems this is a common problem among users online, to which AMD Xilinx has not provided support. As such, and because the FPGA used is an entry-level one that does not support a big multitude of layer types, we will only evaluate the performance of the FPGA for the simpler model, GregNet.

All of the GregNet models have comparable inference times, across the different devices used. Still, it is worth pointing out that the GregNet2DModified model was designed specifically for faster inference on the FPGA, as all of its layers are running on its DPU. It seems this goal was accomplished, but not significantly, with the GregNet2D model being much preferable, with its accuracy post-quantization being more than four percentage points higher.

The GWaveNet models have more differences between themselves, notably on the CPU, where GWaveNet takes almost around double the time to predict as the other models. Additionally, the 2D version consistently outperforms the original model and the 2DModified version consistently outperforms both in terms of speed. The difference between the original and 2D versions may be explained for the GPU due to it being designed to perform matrix multiplications. However, there is no obvious explanation for why this difference is so significant on the CPU, especially when taking the small 3σ error associated with the measurement into account. The 2DModified version is faster due to its lighter nature (less filters in most convolutional layers).

For the GPU, GWaveNet takes approximately 1.5 times longer than GregNet to test on the GPU and 5.6 times longer on the CPU. However, considering the improvement in performance that GWaveNet presents in relation to GregNet, and the small difference in these times, almost negligeable compared to the size of the input signal, we believe GWaveNet to still be a better choice.

Comparing the times between the different devices, we can infer that the models perform the fastest on the GPU, due to its fast matrix multiplication nature. However, the inference times on the CPU for the GregNet models are very close, likely due to the CPU used for inference being a high-end one and the GPU used being now considered more low-end. We expected the FPGA to perform better or similarly to the other

devices, since it is optimized for operations with data, but it takes around four times longer. We believe this is so because the FPGA and pre-built DPU that were used were not meant for this time of data (very large and 1D) and operations (complex NNs). A previous Master thesis at Utrecht University (UU), by Steven Bos, on the use of FPGAs for particle shower classification demonstrated that the FPGA showed an improvement of around 10 times over the CPU. However, the data in this previous study was ideal for this FPGA (small images, with the biggest being 128x128 and the smallest 16x16) and the deployed model was very simple (only three convolutional layers with smaller kernels and a single dense layer). It is also possible that their CPU testing conditions were not ideal, as there is no specific mention of those.

Still, we decided to further investigate the bottleneck for this unexpectedly large amount of time to run on the FPGA. We found that the batch normalization layer was the one creating problems, as Vitis AI has a very specific and complicated way of dealing with batch normalization, as described in [118]. The inference results for the models without their batch normalization layers are shown in Table 4.3. The performance results for these models are not presented, as the models would have to be re-trained to show accurate results.

Model	Time (ms)		
	CPU	GPU	FPGA
GregNet2D	29.44 ± 0.14	24.53 ± 2.62	42.89 ± 0.05
GregNet2DModified	28.72 ± 0.06	30.67 ± 1.45	42.69 ± 0.01
GWaveNet2D	63.64 ± 0.20	36.84 ± 1.23	-
GWaveNet2DModified	40.45 ± 0.30	27.47 ± 1.69	-

Table 4.3: Time it takes for each of the models without their batch normalization layers to predict one sample, averaged over 22000 samples for the CPU and GPU and over 1000 samples for the FPGA, including the time it takes to load the data, with the corresponding 3σ error after having tested around 10 times.

Removing the batch normalization layers cuts the FPGA times in half, while for the CPU it slightly lowers the inference time and for the GPU, on the other hand, it slightly increases it.

In any case, it is worth pointing out once again that the length of the signals used is of 300 seconds, so all of these analysis times are incredibly low compared to that and any combination of model/device would be able to perform real-time low-latency predictions.

4.4 Energy consumption

A useful comparison is between how much energy each NN consumes in each of the used devices. This is displayed in Table 4.4, and the model versions without batch normalization are shown in Table 4.5. The library used for inference of the CPU and GPU energy consumption was, once again, `pyJoules`. The tests for the CPU and GPU were performed on the Nikhef cluster, with the machines being reserved for our inference testing to ensure a fair comparison. On the other hand, in order to test the models for inference on the FPGA, an application that is native to the Kria KV260, `xmutil` [128], was used, in particular, the `xlnx_platformstats` tool [129]. This tool monitors the power consumed by the FPGA, so we saved the power for every 10 seconds that the script was running, averaged it and multiplied it by the time it

took to run in order to get the consumption. It is worth noting that we believe that the measurements for the FPGA are more accurate than the measurements for the CPU and the GPU, as the first has a native application to measure consumption. However, the tool used for the other two is a library that leverages different properties of the hardware to be able to get these readings but is still under heavy development. In particular, it is worth noting that, even though we used 32 CPU cores for the inference tests, only 4 were listed in the output, which may mean that these values are highly biased, but we have no way of confirming.

Model	Consumption (mJ)		
	CPU	GPU	FPGA
GregNet	107.24 ± 4.26	617.12 ± 37.11	-
GregNet2D	177.93 ± 1.60	606.74 ± 32.06	468.59 ± 2.40
GregNet2DModified	178.35 ± 1.34	694.59 ± 28.46	471.51 ± 2.96
GWaveNet	583.66 ± 5.56	983.83 ± 60.52	-
GWaveNet2D	568.92 ± 4.95	858.11 ± 36.83	-
GWaveNet2DModified	240.76 ± 5.85	752.49 ± 57.90	-

Table 4.4: Energy it takes to process and predict one sample, averaged over 22000 samples for the CPU and GPU and over 1000 samples for the FPGA, for the different models in the different devices, with the corresponding 3σ error after having tested around 10 times.

Model	Consumption (mJ)		
	CPU	GPU	FPGA
GregNet2D	177.19 ± 1.00	697.32 ± 54.59	239.80 ± 0.64
GregNet2DModified	176.13 ± 0.97	834.39 ± 27.95	237.48 ± 0.61
GWaveNet2D	539.07 ± 0.90	$1,032.67 \pm 22.61$	-
GWaveNet2DModified	207.74 ± 1.19	788.29 ± 34.18	-

Table 4.5: Energy it takes to process and predict one sample, averaged over 22000 samples for the CPU and GPU and over 1000 samples for the FPGA, for the different models without their batch normalization layers in the different devices, with the corresponding 3σ error after having tested around 10 times.

The difference between the consumption of GregNet and GWaveNet for both the CPU and the GPU is due to the difference in inference time, as GWaveNet spends, once again, 1.5 times more than GregNet in the GPU and 5.6 times in the CPU, meaning the models themselves spend the same amount of energy per amount of time.

As expected, the GPU uses the most energy, even though it takes the least time to predict. This is not only due to is being a heavier machine, needing more power to operate, but also due to it having a support CPU that also spends energy.

Even though the CPU appears to spend the least energy, we do have an unmeasurable uncertainty associated with the used library. Taking into account the prediction time, the FPGA is the most power-efficient and, as such, we believe that a model that is fully optimized for the FPGA would have an overwhelmingly better energy performance running on this device.

4.5 How much does this cost?

If we were to turn these algorithms into pipelines that are running 24/7, how much would that cost over a year? Considering the values we found and that the average industry electricity price in the United States of America, where most search pipelines are run, in 2022 was 8.45 cents per kilowatt [130], Table 4.6 shows these values. The values shown are for the lowest power consumption for each, i.e. the most energy-efficient out of the models with or without batch normalization.

Model	Money ($/year)		
	CPU	GPU	FPGA
GregNet	28.83	210.01	-
GregNet2D	44.52	210.40	41.03
GregNet2DModified	44.69	201.36	41.17
GWaveNet	27.92	216.72	-
GWaveNet2D	49.38	207.47	-
GWaveNet2DModified	27.19	212.40	-

Table 4.6: How much money it costs to constantly be running the models over a year, for the different models in the different devices.

Indeed, as expected, the GPU is the most costly device to run the models in, being 5 to 10 times more expensive than the CPU or the GPU. The FPGA is slightly cheaper than the CPU, by a factor of around 1.1. Still, it is worth mentioning once more that the library used for energy measurement on the CPU and GPU is less accurate than the one used on the FPGA. Furthermore, these prices only take into account the electricity prices, but adding the upfront and upkeep prices of the devices, the FPGA is by far the least costly.

Lastly, we verify that GregNet and GWaveNet end up costing around the same price when run constantly as a pipeline, making us confident that this project was successful in finding a more worthy candidate, GWaveNet, for a precise and sustainable ML-based BNS search pipeline.

Chapter 5

Conclusions

The present thesis was set up as a proof-of-concept study about detecting BNS mergers from their early inspirals using ML on FPGAs. The first question that we had was "Can you even do ML on FPGAs?" and, when brought up to many computing experts we talked with at the start of the project, their answer was usually along the lines of "No, FPGAs are too hard for that" or "No, FPGAs are not designed for that". This author begs to differ.

In this Chapter, we will discuss what the project was able to achieve and to what extent, as well as discuss the most relevant results and the outlook for future work.

In this study, we created two main NN models, trained on the early inspiral data of BNS mergers, with maximum frequencies ranging from 40 to 20 Hz.

We were able to successfully recreate the NN described in the work by Baltus et al. [68], denominated in this thesis as GregNet. Like the authors, we verify that indeed the model seems to have predictions that are not explainable, because it is a very shallow and simple model, not necessarily inspired by its input.

On the other hand, we create a new model, GWaveNet, inspired by the audio generating generative adversarial network (GAN) WaveNet [69]. The model is adapted from a GAN into a binary classifier, that successfully differentiates between noise and noise + signal classes from data from the early inspiral of BNS mergers, at a significantly low FAP. The model is much deeper and more complex than GregNet, taking some inspiration from the physics of the input through the implementation of causal convolutions and using residual blocks to avoid forgetting the earlier taught cases. It also takes advantage from some particular training features, such as having an adaptive learning rate and two early stopping algorithms in place, which ensures its training is stable, as discussed in Section 4.1. As such, the output of GWaveNet is as expected, having more difficulty to decipher the harder cases and being more confident and correct about the easier ones, as shown in Figure 4.9.

When comparing the two, the first is faster and lighter to train, and the same is true for its inference. GWaveNet takes much longer to train, and, even though it also takes somewhat longer to test, these times

are more comparable, with both taking in the order of 10 ms to predict one sample, across multiple devices. In terms of performance, GWaveNet outperforms GregNet by almost 10 percentage points, achieving an accuracy of 76.22% for a fixed FAP of 1%, while GregNet sits at 66.81%. As such, even with GregNet being slightly faster for inference, we believe GWaveNet is the preferred model.

GregNet was successfully adapted to be able to compile for and run in an FPGA, even with all of its restrictions. For this, new 2D versions of the original model were created and it was found that, for the library and hardware used, these appear to be more numerically stable and faster at inference.

The use case in this study is not the ideal one for the specific FPGA model that was used, as this model not only does not have support for all of the types of layers used, making it necessary for a modified version of the models to be created. In addition, it does not support all sizes, with smaller sized-operations being favoured. We are confident that projects that require vision-based tasks or smaller sizes of data (such as the detection of BBHs) would perform better and be able to use the full potential of the Kria KV260. Furthermore, smaller data and smaller models would also open the possibility of using quantization-aware training, which would not deplete the models' accuracies as much during the quantization process, as opposed to PTQ, the method used in this study.

Still, in terms of costs, the FPGA is shown to be overwhelmingly the most sustainable, while even though a GPU is the fastest device for NN inference, it is the one with the highest consumption.

5.1 Future work

This study can be extended in different directions.

Firstly, we can train and test the NNs in new noise settings, such as real noise from O3 and theoretical and real noise from observing run 4 (O4). We plan on doing so in a follow-up paper to this thesis.

Furthermore, the WaveNet architecture could be further explored for different sizes of the operations. In this study, we focused on starting with an architecture that would be close to what was needed for the model to compile for the FPGA. However, in a context without the FPGA, there is room for growth.

Even with keeping the use of the FPGA, it is expected that the support for different layer types and sizes will be extended in the near future. However, we would recommend to use a different FPGA for future work, with more documentation and support from the manufacturer. In particular, the FPGAs by Intel [131] seem to have more support. Additionally, we would recommend the use of a different quantization software, such as `hls4ml` [132], which has more support and functionalities, even though it is still under development.

Still, a way to overcome any arising problems with software/hardware incompatibility is to introduce hardware and software design into the project: designing a DPU architecture for the Kria KV260 by using HDL and accompanying encodings for the layer types and sizes. This would add a completely new dimension to the study and would let one use the FPGA to its full potential.

Appendices

Appendix A

Model details

A.1 GregNet

Layer	#Input channels	#Output channels	Length of output	Kernel size	Stride	Activation
BatchNorm	3	3	155648	-	-	-
Conv1d	3	32	155633	16	1	ReLU
MaxPool1d	32	32	38908	4	4	-
Conv1d	32	64	38901	8	1	ReLU
MaxPool1d	64	64	9725	4	4	-
Conv1d	64	128	9722	4	1	ReLU
MaxPool1d	128	128	2430	4	4	-
Conv1d	128	256	2423	8	1	ReLU
MaxPool1d	256	256	605	4	4	-
Conv1d	256	256	590	16	1	ReLU
MaxPool1d	256	256	147	4	4	-
Flatten	256	1	37632	-	-	-
Linear	1	1	128	-	-	ReLU
Linear	1	1	1	-	-	Sigmoid

Table A.1: Architecture of GregNet.

A.2 GWaveNet

Layer	#Input channels	#Output channels	Length of output	Kernel size	Dilation	Activation
ModuleWavenet	3	64	77832	16	1	Gate activation
ModuleWavenet	64	64	38931	16	2	Gate activation
ModuleWavenet	64	64	19496	16	4	Gate activation
ModuleWavenet	64	64	9808	16	8	Gate activation
ModuleWavenet	64	64	5024	16	16	Gate activation
ModuleWavenet	64	64	2752	16	32	Gate activation
ModuleWavenet	64	64	1856	16	64	Gate activation
Conv1d	64	1	1856	1	1	ReLU
Linear	1	1	50	-	-	ReLU
Linear	1	1	51	-	-	Sigmoid

Table A.2: Architecture of the NN inspired by WaveNet.

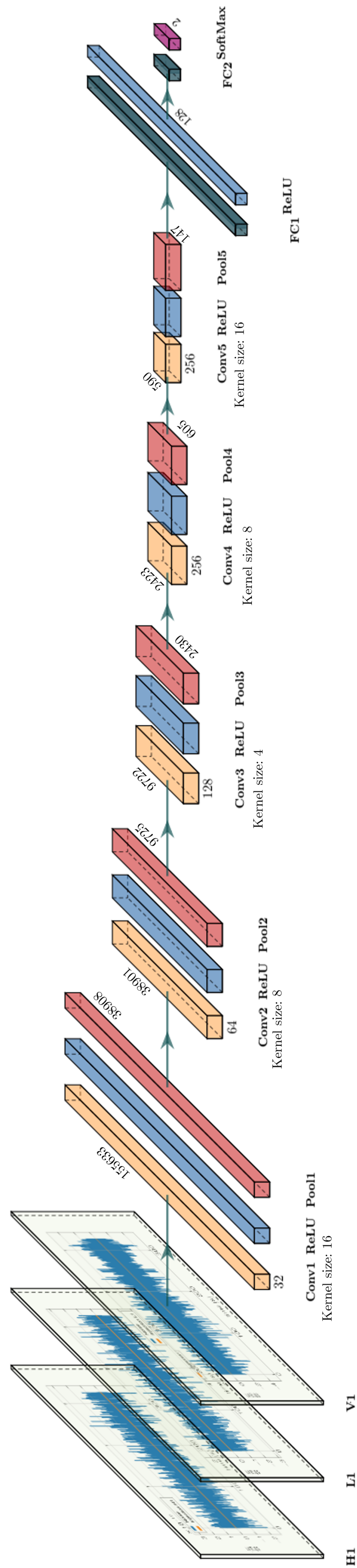


Figure A.1: Visual representation of the architecture of GregNet. Adapted from [68].

Appendix B

Optimizer algorithms

B.1 AdaMax

Algorithm 1 Algorithm for AdaMax.

Input

γ	learning rate
β_1	
β_2	
θ_0	parameters
$f(\theta)$	objective
λ	weight decay
ε	

Initialize

$m_0 \leftarrow 0$	first moment
$u_0 \leftarrow 0$	infinity norm

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda\, \theta_{t-1}$
  end if
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $u_t \leftarrow \max(\beta_2 u_{t-1}, |g_t| + \varepsilon)$
  $\theta_t \leftarrow \theta_{t-1} - \frac{\gamma m_t}{(1-\beta_1^t) u_t}$
end for

Output
  θ_t

B.2 AdamW

Algorithm 2 Algorithm for AdamW.

Input

γ	learning rate
β_1	
β_2	
θ_0	parameters
$f(\theta)$	objective
λ	weight decay
ε	
amsgrad	variation of the algorithm [133] (default: false)
maximize	maximize the objective instead of minimizing (default: false)

Initialize

$m_0 \leftarrow 0$	first moment
$v_0 \leftarrow 0$	second moment
$\widehat{v_0}^{max} \leftarrow 0$	

for $t = 1$ **to** ... **do**
  if *maximize* **then**
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  end if
  $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\,\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$
  $\widehat{v_t} \leftarrow v_t/(1-\beta_2^t)$
  if *amsgrad* **then**
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_t - \gamma \widehat{m_t}/(\sqrt{\widehat{v_t}^{max}} + \varepsilon)$
  else
    $\theta_t \leftarrow \theta_t - \gamma \widehat{m_t}/(\sqrt{\widehat{v_t}} + \varepsilon)$
  end if
end for

Output
  θ_t

Appendix C

Code snippets

C.1 Dilated causal convolutions

```
class CausalConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, dilation):
        super(CausalConv1d, self).__init__()
        self.padding = (kernel_size - 1) * dilation
        self.conv = nn.Conv1d(
            in_channels,
            out_channels,
            kernel_size,
            self.padding,
            dilation,
        )

    def forward(self, x):
        x = nn.functional.pad(x, (self.padding, 0))
        x = self.conv(x)
        return x
```

Listing C.1: Code snippet to implement causal convolutions in PyTorch.

C.2 ModuleWavenet

```
class ModuleWavenet(nn.Module):
    def __init__(
        self,
        in_channels,
        filters,
        dilation,
        kernel_size,
    ):
        super(ModuleWavenet, self).__init__()
```

```
10
11          self.input = in_channels
12          self.output = filters
13
14          self.dilation = dilation
15          self.kernel_size = kernel_size
16
17          self.batch_norm = torch.nn.BatchNorm1d(self.out)
18
19          self.gate_tanh = torch.nn.Tanh()
20          self.gate_sigmoid = torch.nn.Sigmoid()
21
22          self.conv_dilated = CausalConv1d(
23              in_channels=self.input,
24              out_channels=self.output,
25              kernel_size=kernel_size,
26              dilation=dilation,
27          )
28
29      def forward(self, x):
30          x_old = x
31
32          x_a = self.conv_dilated(x)
33          x_b = self.conv_dilated(x)
34
35          # Gate activation
36          x_a = self.batch_norm(x_a)
37          x_b = self.batch_norm(x_b)
38          x = self.gate_tanh(x_a) * self.gate_sigmoid(x_b)
39
40          # Skip-connection
41          if self.input != self.output:
42              x_res = x
43          else:
44              x_res = x + x_old[:, :, -x.size(2) :]
45
46          return x_res, x
```

Listing C.2: Code snippet to implement ModuleWavenet in PyTorch.

Appendix D

Manual for the Kria KV260

The Kria KV260 Vision artificial intelligence (AI) Starter Kit by AMD [107], used for this study, is a kit that includes an FPGA with a pre-configured system on module (SOM) with a DPU, carrier card and thermal solution. The tutorials that were mainly followed for the usage of the FPGA were the Getting Started with Kria KV260 Vision AI Starter Kit [134] and the Quick Start Guide for Zynq UltraScale+ by Xilinx [135], as well as the Vitis AI User Guide [108], but they have many faults and missing information. So, in this manual, we hope to clear any doubts, reduce the amount of trial and error needed in future work and make the usage of the FPGA smoother in general.

D.1 Versions

This section aims to list the versions of the software used, such that the reader can trace back any errors by up/down-grading to these:

- Ubuntu 20.04 long term support (LTS) for the host machine,
- Vitis AI v3.0 for use in the host machine,
- Vitis AI pre-build image for the Kria KV260, including the B4096 DPU architecture.

D.2 Start-up

The first step to working with the Kria KV260 is preparing an image from which it can boot. The module can leverage its own operating system (OS). Still, in this study, we used an SD card image pre-built specifically for the Kria KV260 for use with Vitis AI [136], as this version already comes prepared for use with Vitis AI and can compile in the B4096 architecture, the latest and faster (at the time of writing) architecture for the KV26 SOM, which, for example, an Ubuntu 20.04 LTS version-based system cannot.

The image should be burned on a microSD card with at least 8GB of memory, but at least 32GB is recom-

mended. The Balena Etcher [137] tool can be used to do this.

Once the microSD card is ready, we can start booting up the Kria KV260. For this, the following materials are needed:

1. MicroSD card burned with the Vitis AI pre-built image
2. USB keyboard and mouse
3. Monitor with a HMDI or DisplayPort (DP) cable
4. Ethernet cable
5. Power adapter

Those should be connected, in order, to the following ports in the Kria KV260 as shown in Figure D.1:

1. J11 MicroSD
2. 2x USB 3.0
3. J5 HDMI or J6 DisplayPort
4. J10 RJ-45 Ethernet
5. J12 DC Jack

After a couple of minutes, the monitor should show the FPGA booting. The default username and password are both *root*.

To run in the FPGA more smoothly, one can connect via secure shell protocol (SSH) [138] from the host machine. This is also useful to pass files between the host and the target (the FPGA). For this, first one needs to know the IP address of the target, which can be found with the following command:

```
1 ifconfig eth0
```

Listing D.1: Code snippet to get IP address. Run on the target.

If you are using a monitor, you can do this directly on the FPGA by using the USB mouse and keyboard to reach the terminal. However, it is also possible to do this without a monitor, keyboard or mouse, by connecting a micro-USB to USB cable that allows for data transfer between the target and the host and accessing the FPGA terminal via UART connection.

To connect via SSH from the host to the target, one can use the following command:

```
1 ssh -X root@[TARGET_IP_ADDRESS]
```

Listing D.2: Code snippet to connect via SSH. Run on the host.

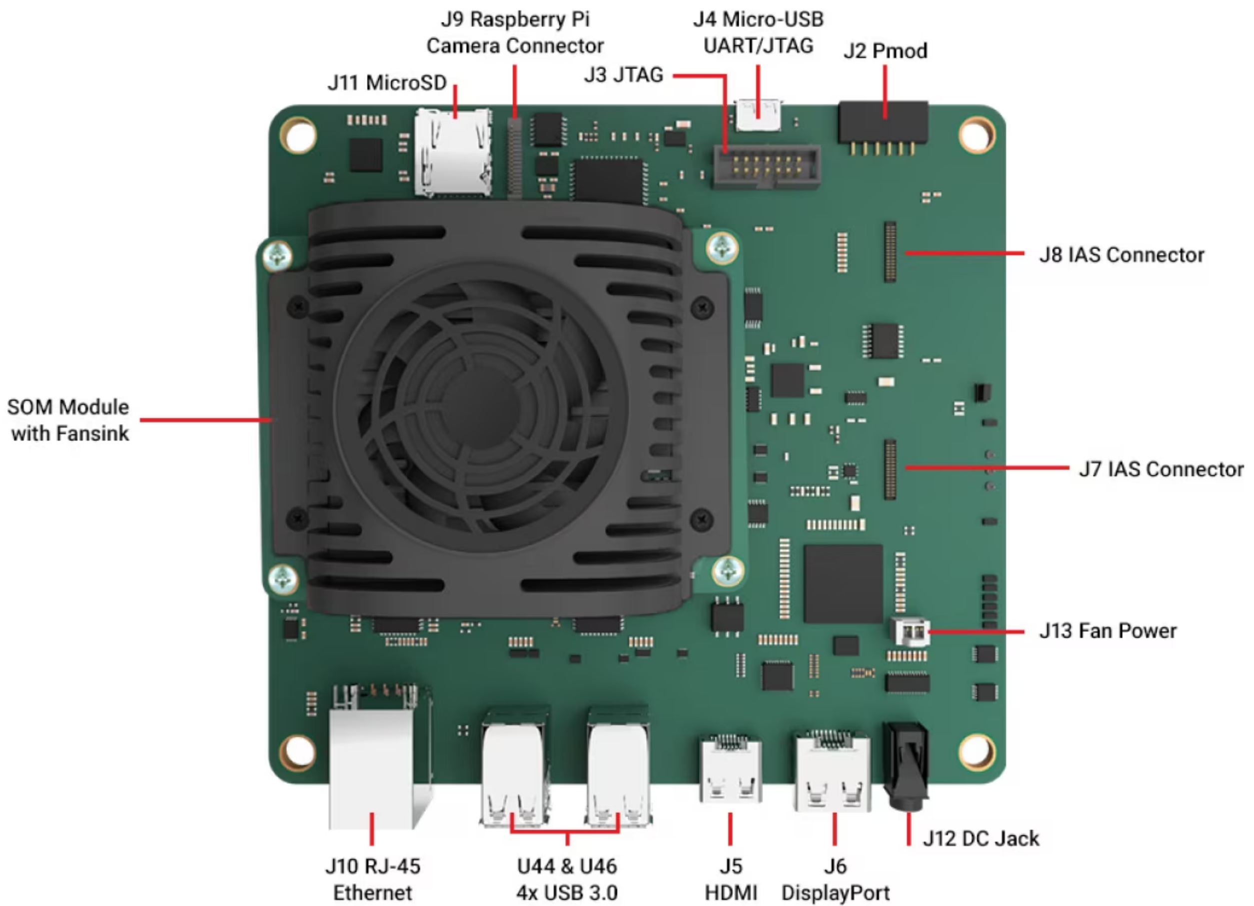


Figure D.1: Kria KV260 board with labelled inputs [134].

The terminal on your host can now control the target's terminal and run whatever you want from there. Note that running anything on that specific terminal will run it on the FPGA, not on the host, so if you need to use the terminal on your own machine, open a new one. Now, the monitor, keyboard and mouse are no longer needed.

In order to pass files between the host and the target, one can use:

```
1 scp [FILE] root@[TARGET_IP_ADDRESS]:~/
```

Listing D.3: Code snippet to pass files via SSH to the home directory of the target. Run on the host.

Note that you might need to add the flag `-O` (capital o) when using `scp` if you have a recent OpenSSH release installed on your machine.

The FPGA is ready for the deployment of a NN model.

D.3 Preparing the neural network for the field programmable gate array: Vitis AI

D.3.1 Preparation

To prepare the NN trained and tested in a GPU for the FPGA, external software needs to be leveraged. Vitis AI [139] was used for this study. At the time of writing, the latest stable version for the Kria KV260 is v3.0, so this one was used.

To install Vitis AI v3.0, clone the Vitis AI GitHub repository [140] into a personal machine and checkout into the desired version by using the following on a terminal:

```
git clone https://github.com/Xilinx/Vitis-AI.git
git checkout v3.0
```

Listing D.4: Code snippet to install Vitis AI. Run on the host.

This study used a machine with the Ubuntu 20.04 LTS version as the host.

D.3.2 Docker

Docker [141] will also be needed to use Vitis AI. To install docker in the Ubuntu 20.04 LTS version, the next steps should be followed, as described in the official Docker documentation [142]:

```
# to uninstall any conflicting packages:
for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd
    ↪ runc; do sudo apt-get remove $pkg; done

# add docker's official gpg key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# add the repository to apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://
      ↪ download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

# install the latest version of docker
```

```
19 sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
     ↪ compose-plugin
20
21 # check if the installation is successful; a hello world message should be printed on the
     ↪ terminal after running this command
22 sudo docker run hello-world
```

Listing D.5: Code snippet to install Docker. Run on host.

To be able to run Docker freely, non-root users should be added to the Docker group, like so [143]:

```
1 # create the docker group; this command may return saying that the group already exists,
    ↪ continue to the next step
2 sudo groupadd docker
3
4 # add your user to the docker group
5 sudo usermod -aG docker $USER
6
7 # activate the changes
8 newgrp docker
9
10 # verify that the command docker works without the command sudo
11 docker run hello-world
```

Listing D.6: Code snippet to add current user to the Docker group. Run on host.

Now, we can leverage the pre-built Docker containers by Xilinx. If you have access to a compute unified device architecture (CUDA)-enabled GPU [144], you can leverage a container that allows for GPU usage and the processes will run much faster. To verify this, you may check the list of CUDA GPUs [145] or look up your own GPU online to check if it is CUDA-enabled (for example, the NVIDIA GeForce MX330 GPU was used in this study [146] – you can see the CUDA version is 6.1). If you do not know if you have access to a GPU or which one you have, you can run the following on a terminal:

```
1 sudo lshw -C display
```

Listing D.7: Code snippet to verify which, if any, GPU is installed. Run on host.

D.3.2.1 If you have access to a GPU

Prepare the host for the installation of the CUDA-powered Docker container by running the following to uninstall conflicting libraries and installing dependencies:

```
1 sudo apt purge nvidia* libnvidia*
2 sudo apt install nvidia-driver-510
3 sudo apt install nvidia-container-toolkit
```

Listing D.8: Code snippet to install the NVIDIA driver and the Container Toolkit. Run on host.

You may install another version of the driver by switching out *nvidia-driver-510* with *nvidia-driver-[VERSION]*, but the version 510 is recent and stable at the time of writing. To confirm that the installation was successful, run:

```
1 nvidia-smi
```

Listing D.9: Code snippet to check driver status. Run on host.

If the installation was successful, you should be able to see the Driver and CUDA version.

To build a CUDA-powered Docker container, first navigate to the *docker* folder inside the Vitis AI folder:

```
1 cd [VITIS_AI_INSTALLATION_PATH]/Vitis-AI/docker
```

Listing D.10: Code snippet to navigate to the pre-built docker containers folder. Run on host.

There are various options to build Docker containers. In this study, we used PyTorch [84] and allowed for the possibility of using the Vitis AI Optimizer, in which case the target framework is *opt_pytorch*. If you do not have a license for the Vitis AI Optimizer, the target framework is simply *pytorch*.

```
1 ./docker_build.sh -t gpu -f [TARGET_FRAMEWORK (e.g. opt_pytorch)]
```

Listing D.11: Code snippet to build the pre-built docker container for a specific target framework on the GPU. Run on host.

The containers with access to a GPU have some faults, namely that the Conda [147] environments that are already set up on them do not have all of the libraries needed to run some of the programs for quantizing and compiling the model. However, most of them are fairly easy to resolve by searching on the internet for the error code (and most will be resolved by using *pip install [LIBRARY]*). Still, if you find yourself stuck, you can always rely on the CPU-based docker image, as it does the same things; it will simply take longer.

D.3.2.2 If you do not have access to a GPU

Installing the CPU-based Docker is much easier and can be done with the simple command:

```
1 docker pull xilinx/vitis-ai-pytorch-cpu:latest
```

Listing D.12: Code snippet to build the pre-built docker container for PyTorch [84] on the CPU. Run on host.

D.3.2.3 Running the Docker container

After the containers are built (it might take a while), you are ready to run them. To do this, make sure you are back in the main Vitis AI folder:

```
1 cd [VITIS_AI_INSTALLATION_PATH]/Vitis-AI
```

Listing D.13: Code snippet to go to the Vitis AI main folder. Run on host.

Afterwards, use the file *docker_run.sh* to run your container followed by the container you chose to leverage. If you have forgotten the name/keywords for your container, you may find your installed containers by running the following command:

```
1 docker images -a
```

Listing D.14: Code snippet to list built Docker containers. Run on host.

Select the one you want to work with. This manual will use the CPU-based one from here on out for simplicity, except when mentioned otherwise, but everything is also applicable to the GPU-based applications. To run the container, type out the following:

```
1 ./docker_run.sh xilinx/vitis-ai-pytorch-cpu:latest
```

Listing D.15: Code snippet to run the CPU-based Docker container for PyTorch [84]. Run on host.

If this was successful, you should get a message saying "Vitis AI" in big letters and you should be logged in as *vitis-ai-user*

D.3.3 Conda environment

After you are inside the Docker container, we need to activate the Conda [147] environment already built inside this Docker container. To find the name of the environment, run:

```
1 conda info --envs
```

Listing D.16: Code snippet to list the pre-built Conda environments. Run on host inside Docker.

If you are using the CPU-based Docker and the PyTorch framework, activating the container will be something along the lines:

```
1 conda activate vitis-ai-pytorch
```

Listing D.17: Code snippet to activate the pre-built Conda environment. Run on host inside Docker.

If this is successful, the prompt will show that you are inside the Conda environment.

D.3.4 Model inspector

Vitis AI provides a program that can inspect your model to see whether it will run on the DPU. Some layers are currently not supported by the DPU and some layer orderings are not optimal, and the model inspector [148] can point those out. The supported layers are mentioned in the Vitis AI User Guide [118], as well as the supported sizes [149]. Some popular layer types, like Conv1d, are not possible without modifications to the hardware.

In order to run the model inspector, you can use the code provided by Xilinx [150] or use the slightly modified version that was used for this study [1].

To open the Jupyter Notebook inside the Docker container and inside the the Conda environment, navigate to where your *model_inspector.ipynb* file is and run it as follows:

```
1 jupyter notebook model_inspector ipynb
```

Listing D.18: Code snippet to run the model inspector. Run on host inside Docker inside Conda environment.

The output of the Jupyter Notebook will be an image indicating where the layers of the NN will be able to run in the FPGA (CPU or DPU), as well as a text file with the same information.

In order to take best advantage of the FPGA and its DPU, all layers should be able to run in the DPU, and this is what will make the most difference in the latency of your model. The inspector will give tips on how to optimize the model for the DPU, such as layer orderings, but it might also just throw an error if you are using layers that are not supported [151], which should be easily fixed by searching online.

D.3.5 Quantizer

In this study, we used PTQ and, hence, this is the method presented in this manual.

In order to quantize your pre-trained model, you need to use the quantizer provided with the pytorch_nndct library [152], which is pre-installed in the conda environment inside the docker container.

Xilinx provides a pre-made script for quantization, but once again it is very specific for vision-based applications. As such, a more general script can be found in the GitHub for this study [1].

In order to be able to run the quantizer, the Python file of the model (.py) and the trained model file (.pt) should be in a folder. In this case, it was a folder called "model", so we will refer to it as that, but this is possible to change without touching the script, via the arguments passed to it. More on that later. Additionally, a folder with validation samples should also be made, with between 100 and 1000 samples. This folder was called "dataset".

You can evaluate the accuracy of your initial floating point model, but this is an optional step in the quantization procedure. If you wish to do so, you can use the following command:

```
1 python quantize.py --quant_mode float --data_dir dataset --model_dir model --model_name
    ↪ GregNet2D --decision_threshold 0.86
```

Listing D.19: Code snippet to evaluate the accuracy of the floating point model. Run on host inside Docker inside Conda environment.

The arguments passed to the scrip include:

- **quant_mode**: the mode in which the script is being run,
- **data_dir**: the name of the folder where the validation samples are,

- **model_dir**: the name of the folder where the .py and .pt model files are,
- **model_name**: the name of the model you want to run the script for, if you have more than one,
- **decision_theshold**: the decision threshold that should be used to evaluate the accuracy.

These are used in a similar way for the rest of the commands.

In order to start quantization, run the following command:

```
1 python quantize.py --quant_mode calib --data_dir dataset --model_dir model --model_name [
    ↪ MODEL_NAME] --decision_threshold [DECISION_THRESHOLD]
```

Listing D.20: Code snippet to start quantization and generate the quantized vai_q_pytorch format model and the quantization steps of tensors. Run on host inside Docker inside Conda environment.

This command will generate a new folder called "quantize_result", that includes a quantized vai_q_pytorch format model (.py) and a file with the quantization steps of tensors information (Quant_info.json).

In order to optionally evaluate the accuracy of the quantized model, you can use the following command:

```
1 python quantize.py --quant_mode test --data_dir dataset --model_dir model --model_name [
    ↪ MODEL_NAME] --decision_threshold [DECISION_THRESHOLD]
```

Listing D.21: Code snippet to evaluate the accuracy of the quantized model. Run on host inside Docker inside Conda environment.

In order to generate the .xmodel quantized file, needed for the compilation for the FPGA, you can use the following command:

```
1 python quantize.py --quant_mode test --data_dir dataset --model_dir model --model_name [
    ↪ MODEL_NAME] --decision_threshold [DECISION_THRESHOLD] --subset_len 1 --batch_size 1 --
    ↪ deploy
```

Listing D.22: Code snippet to generate the .xmodel file. Run on host inside Docker inside Conda environment.

The quantization step is finished, and we can now move on to compiling the quantized INT8.xmodel into a deployable DPU.xmodel, fit for the particular architecture being used.

D.3.6 Compiler

In order to compile the model for the correct architecture, check and modify the arch.json file inside the folder /opt/vitis_ai/compiler/arch/DPUCZDX8G/KV260. The default is already the B4096 architecture, so nothing needs to be done if you are using this one.

To compile the model, run the following command:

```
1 vai_c_xir -x quantize_result/[MODEL_NAME]_int.xmodel -a /opt/vitis_ai/compiler/arch/DPUCZDX8G
    ↪ /KV260/arch.json -o [MODEL_NAME] -n [MODEL_NAME]
```

Listing D.23: Code snippet to compile the model. Run on host inside Docker inside Conda environment.

If the compilation is successful, a new folder with the name of your model will be generated containing the .xmodel file, generated according to the specified DPU architecture.

Additionally, it is advised to create a .prototxt file with configurations of the model. An example is given in the GitHub [1].

D.4 Model deployment

In order to deploy the model, we first need to copy the necessary files to the FPGA. We start with copying the model folder, which contains the `.xmodel`, the `md5sum.txt` and the `meta.json` files. We can do that with the following command, from the main folder where you run Vitis AI:

```
1 scp -r [MODEL_NAME] root@[TARGET_IP_ADDRESS]:/usr/share/vitis_ai_library/models/
```

Listing D.24: Code snippet to pass the model files via SSH to a specified location on the target. Run on the host.

The location can be whatever you prefer, but this is the default one according to the Linux folder structure and what the OS already comes prepared with from Vitis AI.

Gather test samples in a folder and similarly copy them to the FPGA. We chose to make a new folder inside the Vitis-AI folder to keep the program and data together:

```
1 scp -r dataset root@[TARGET_IP_ADDRESS]:~/Vitis-AI/[PROJECT_NAME]
```

Listing D.25: Code snippet to pass the samples folder via SSH to a specified location on the target. Run on the host.

An example of a program used to deploy the model can be found in the project GitHub [1]. To copy it to the organised directory in the target, run the following command:

```
1 scp -r app_mt.py root@[TARGET_IP_ADDRESS]:~/Vitis-AI/[PROJECT_NAME]
```

Listing D.26: Code snippet to copy the app to the target. Run on the host.

To run it, use the following command:

```
1 python app_mt.py --data_dir dataset --threads [NUMBER_OF_THREADS] --model_name [MODEL_NAME]
    ↪ --decision_threshold [DECISION_THRESHOLD]
```

Listing D.27: Code snippet to deploy the model. Run on the target.

Bibliography

[1] Ana Martins. *Deploying Custom Neural Networks in FPGA - Model Inspector*. GitHub repository. Accessed 04 April 2024. Available from: https://github.com/anaismartins/deploying-custom-nn-in-fpga/blob/main/src/model_inspector.ipynb (cit. on pp. 1, 86, 87, 89).

[2] E. C. News. *Intel's Chip Manufacturing Has Massive Carbon Footprint: What Will It Mean for Ohio?* https://energycentral.com/news/intels-chip-manufacturing-has-massive-carbon-footprint-what-will-mean-ohio. Accessed 10 June 2024. 2022 (cit. on p. 1).

[3] M. N. Office. *How can we reduce the carbon footprint of global computing?* https://news.mit.edu/2022/how-can-we-reduce-carbon-footprint-global-computing-0428. Accessed 10 June 2024. 2022 (cit. on p. 1).

[4] CyberSided. *How Long Do GPUs Last?* https://cybersided.com/how-long-do-gpus-last/. Accessed 23 June 2024 (cit. on p. 2).

[5] O. Heaviside. "Gravitational and Electromagnetic Analogy". In: (1893). Accessed 23 June 2024 (cit. on p. 3).

[6] H. Poincaré. "Sur la dynamique de l'électron". In: *Comptes Rendus de l'Académie des Sciences* 140 (1905), pp. 1504–1508 (cit. on p. 3).

[7] A. Einstein. "Näherungsweise Integration der Feldgleichungen der Gravitation". In: *Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften* 1916 (1916), pp. 688–696 (cit. on p. 3).

[8] A. Einstein. "Gravitationswellen". In: *Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften* 1918 (1918), pp. 154–167 (cit. on p. 3).

[9] A. Einstein. "Die Grundlage der allgemeinen Relativitätstheorie". In: *Annalen der Physik* 354.7 (1916), pp. 769–822 (cit. on p. 3).

[10] A. Einstein. "Die Feldgleichungen der Gravitation". In: *Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften* 1915 (1915), pp. 844–847 (cit. on p. 3).

[11] P. D. Lasky. "Gravitational Waves from Neutron Stars: A Review". In: *Publications of the Astronomical Society of Australia* 32 (2015), e034. DOI: 10.1017/pasa.2015.35 (cit. on p. 8).

[12] LIGO. *Sources and Types of Gravitational Waves*. https://www.ligo.caltech.edu/page/gw-sources. Accessed 04 June 2024 (cit. on p. 8).

[13] NASA. *NSF's LIGO Has Detected Gravitational Waves*. https://www.nasa.gov/universe/nsfs-ligo-has-detected-gravitational-waves/. Accessed 08 June 2024. 2016 (cit. on p. 9).

[14] B. P. Abbott et al. "LIGO: the Laser Interferometer Gravitational-Wave Observatory". In: *Reports on Progress in Physics* 72.7 (June 2009), p. 076901. ISSN: 1361-6633. DOI: 10.1088/0034-4885/72/7/076901. Available from: http://dx.doi.org/10.1088/0034-4885/72/7/076901 (cit. on pp. 9, 18).

[15] 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. Available from: https://link.aps.org/doi/10.1103/PhysRevLett.116.061102 (cit. on pp. 9, 18).

[16] M. Bailes et al. "Gravitational-wave physics and astronomy in the 2020s and 2030s". In: *Nature Rev. Phys.* 3.5 (2021), pp. 344–366. DOI: 10.1038/s42254-021-00303-8 (cit. on pp. 9, 10).

[17] J. Kunz. *Neutron Stars*. 2022. arXiv: 2204.12520 [gr-qc] (cit. on p. 10).

[18] A. Hewish et al. "Observation of a Rapidly Pulsating Radio Source". In: *Nature* 217 (1968), pp. 709–713. Available from: https://api.semanticscholar.org/CorpusID:4277613 (cit. on p. 10).

[19] J. R. Oppenheimer and G. M. Volkoff. "On Massive Neutron Cores". In: *Phys. Rev.* 55 (4 Feb. 1939), pp. 374–381. DOI: 10.1103/PhysRev.55.374. Available from: https://link.aps.org/doi/10.1103/PhysRev.55.374 (cit. on p. 10).

[20] K. Sumiyoshi, T. Kojo, and S. Furusawa. "Equation of State in Neutron Stars and Supernovae". In: *Handbook of Nuclear Physics*. Ed. by I. Tanihata, H. Toki, and T. Kajino. Singapore: Springer Nature Singapore, 2023, pp. 3127–3177. ISBN 978-981-19-6345-2. DOI: 10.1007/978-981-19-6345-2_104. Available from: https://doi.org/10.1007/978-981-19-6345-2_104 (cit. on p. 10).

[21] *Equation of State for Neutron Stars*. European Space Agency (ESA). Accessed 13 April 2024. Available from: https://sci.esa.int/web/loft/-/49338-equation-of-state-for-neutron-stars (cit. on p. 11).

[22] T. C. H et al. "Relativistic Shapiro delay measurements of an extremely massive millisecond pulsar". In: *Nature Astronomy* 4.1 (2020), pp. 72–76. DOI: 10.1038/s41550-019-0880-2 (cit. on p. 11).

[23] G. H. BORDBAR and M. HAYATI. "COMPUTATION OF NEUTRON STAR STRUCTURE USING MODERN EQUATION OF STATE". In: *International Journal of Modern Physics A* 21.07 (Mar. 2006), pp. 1555–1565. ISSN: 1793-656X. DOI: 10.1142/s0217751x06028400. Available from: http://dx.doi.org/10.1142/S0217751X06028400 (cit. on p. 11).

[24] G. F. Burgio and H.-J. Schulze. "The maximum and minimum mass of protoneutron stars in the Brueckner theory". In: *Astronomy and Astrophysics* 518 (July 2010), A17. ISSN: 1432-0746. DOI: 10.1051/0004-6361/201014308. Available from: http://dx.doi.org/10.1051/0004-6361/201014308 (cit. on p. 11).

[25] J. G. Martinez et al. "PULSAR J0453+1559: A DOUBLE NEUTRON STAR SYSTEM WITH A LARGE MASS ASYMMETRY". In: *The Astrophysical Journal* 812.2 (Oct. 2015), p. 143. ISSN: 1538-4357. DOI: 10.1088/0004-637x/812/2/143. Available from: http://dx.doi.org/10.1088/0004-637X/812/2/143 (cit. on p. 11).

[26] B. P. Abbott et al. "GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral". In: *Physical Review Letters* 119.16 (Oct. 2017). ISSN: 1079-7114. DOI: 10.1103/physrevlett.119.161101. Available from: http://dx.doi.org/10.1103/PhysRevLett.119.161101 (cit. on pp. 11, 18).

[27] R. Narayan, B. Paczynski, and T. Piran. "Gamma-ray bursts as the death throes of massive binary stars". In: *The Astrophysical Journal* 395 (Aug. 1992), p. L83. ISSN: 1538-4357. DOI: 10.1086/186493. Available from: http://dx.doi.org/10.1086/186493 (cit. on p. 11).

[28] Science News. *Trio wins physics Nobel for gravitational wave detection*. Accessed 10 April 2024. Available from: https://www.sciencenews.org/article/trio-wins-physics-nobel-prize-gravitational-wave-detection (cit. on p. 12).

[29] A. A. Michelson and E. W. Morley. "On the relative motion of the Earth and the luminiferous ether". In: *American Journal of Science* s3-34 (1887), pp. 333–345. Available from: https://api.semanticscholar.org/CorpusID:124333204 (cit. on p. 12).

[30] J. Aasi et al. "Enhanced sensitivity of the LIGO gravitational wave detector by using squeezed states of light". In: *Nature Photonics* 7.8 (July 2013), pp. 613–619. ISSN: 1749-4893. DOI: 10.1038/nphoton.2013.177. Available from: http://dx.doi.org/10.1038/nphoton.2013.177 (cit. on p. 13).

[31] F. Acernese et al. "Advanced Virgo: a second-generation interferometric gravitational wave detector". In: *Classical and Quantum Gravity* 32.2 (Dec. 2014), p. 024001. ISSN: 1361-6382. DOI: 10.1088/0264-9381/32/2/024001. Available from: http://dx.doi.org/10.1088/0264-9381/32/2/024001 (cit. on pp. 13, 31).

[32] G. D. Meadors, K. Kawabe, and K. Riles. "Increasing LIGO sensitivity by feedforward subtraction of auxiliary length control noise". In: *Classical and Quantum Gravity* 31.10 (May 2014), p. 105014. ISSN: 1361-6382. DOI: 10.1088/0264-9381/31/10/105014. Available from: http://dx.doi.org/10.1088/0264-9381/31/10/105014 (cit. on p. 13).

[33] B. P. Abbott et al. "A guide to LIGO–Virgo detector noise and extraction of transient gravitational-wave signals". In: *Classical and Quantum Gravity* 37.5 (Feb. 2020), p. 055002. ISSN: 1361-6382. DOI: 10.1088/1361-6382/ab685e. Available from: http://dx.doi.org/10.1088/1361-6382/ab685e (cit. on p. 13).

[34] LIGO. *LIGO's Interferometers*. https://www.ligo.caltech.edu/page/ligos-ifo. Accessed 08 June 2024 (cit. on p. 14).

[35] M. A. T. Chowdhury. *Advancements in Glitch Subtraction Systems for Enhancing Gravitational Wave Data Analysis: A Brief Review*. 2024. arXiv: 2406.01318 [gr-qc] (cit. on p. 15).

[36] S. Bahaadini et al. "Machine learning for Gravity Spy: Glitch classification and dataset". In: *Information Sciences* 444 (2018), pp. 172–186. ISSN: 0020-0255. DOI: https://doi.org/10.

1016/j.ins.2018.02.068. Available from: https://www.sciencedirect.com/science/article/pii/S0020025518301634 (cit. on p. 15).

[37] J. Veitch and A. Vecchio. "Bayesian approach to the follow-up of candidate gravitational wave signals". In: *Physical Review D* 78.2 (July 2008). ISSN: 1550-2368. DOI: 10.1103/physrevd.78.022001. Available from: http://dx.doi.org/10.1103/PhysRevD.78.022001 (cit. on p. 15).

[38] N. Cornish et al. "The BayesWave analysis pipeline in the era of gravitational wave observations". In: *Physical Review D* 103 (Nov. 2020). DOI: 10.1103/PhysRevD.103.044006 (cit. on p. 15).

[39] E. Marx et al. *A machine-learning pipeline for real-time detection of gravitational waves from compact binary coalescences*. 2024. arXiv: 2403.18661 [gr-qc] (cit. on pp. 15, 24).

[40] The Virgo Collaboration. *The Virgo Collaboration*. Accessed 10 April 2024. Available from: http://public.virgo-gw.eu/the-virgo-collaboration/ (cit. on p. 18).

[41] European Space Agency (ESA). *INTEGRAL SPI Instrument*. Accessed 16 April 2024. Available from: https://www.cosmos.esa.int/web/integral/instruments-spi (cit. on p. 18).

[42] C. Meegan et al. "THEFERMIGAMMA-RAY BURST MONITOR". In: *The Astrophysical Journal* 702.1 (Aug. 2009), pp. 791–804. ISSN: 1538-4357. DOI: 10.1088/0004-637x/702/1/791. Available from: http://dx.doi.org/10.1088/0004-637X/702/1/791 (cit. on p. 18).

[43] University of California, Santa Cruz. *UC Santa Cruz Neutron Star Merger*. Accessed 2024. Available from: https://reports.news.ucsc.edu/neutron-star-merger/media/ (cit. on p. 18).

[44] University of California, Davis. *DLT40 Survey*. Accessed 2024. Available from: https://dark.physics.ucdavis.edu/dlt40/DLT40 (cit. on p. 18).

[45] B. P. Abbott et al. "Multi-messenger Observations of a Binary Neutron Star Merger*". In: *The Astrophysical Journal Letters* 848.2 (Oct. 2017), p. L12. ISSN: 2041-8213. DOI: 10.3847/2041-8213/aa91c9. Available from: http://dx.doi.org/10.3847/2041-8213/aa91c9 (cit. on p. 18).

[46] S. Yang et al. "Optical Follow-up of Gravitational-wave Events during the Second Advanced LIGO/VIRGO Observing Run with the DLT40 Survey". In: *The Astrophysical Journal* 875.1 (Apr. 2019), p. 59. ISSN: 1538-4357. DOI: 10.3847/1538-4357/ab0e06. Available from: http://dx.doi.org/10.3847/1538-4357/ab0e06 (cit. on p. 18).

[47] B. P. Abbott et al. "A gravitational-wave standard siren measurement of the Hubble constant". In: *Nature* 551.7678 (Oct. 2017), pp. 85–88. ISSN: 1476-4687. DOI: 10.1038/nature24471. Available from: http://dx.doi.org/10.1038/nature24471 (cit. on p. 18).

[48] P. S. Cowperthwaite et al. "The Electromagnetic Counterpart of the Binary Neutron Star Merger LIGO/Virgo GW170817. II. UV, Optical, and Near-infrared Light Curves and Comparison to Kilonova Models". In: *The Astrophysical Journal* 848.2 (Oct. 2017), p. L17. ISSN: 2041-8213. DOI: 10.3847/2041-8213/aa8fc7. Available from: http://dx.doi.org/10.3847/2041-8213/aa8fc7 (cit. on p. 18).

[49] T. Dietrich, T. Hinderer, and A. Samajdar. "Interpreting Binary Neutron Star Mergers: Describing the Binary Neutron Star Dynamics, Modelling Gravitational Waveforms, and Analyzing Detections". In: *Gen. Rel. Grav.* 53.3 (2021), p. 27. DOI: 10.1007/s10714-020-02751-6. arXiv: 2004.02527 [gr-qc] (cit. on p. 18).

[50] K. Collaboration et al. *Overview of KAGRA : KAGRA science*. 2020. arXiv: 2008.02921 [gr-qc] (cit. on p. 19).

[51] T. L. S. Collaboration, the Virgo Collaboration, and the KAGRA Collaboration. *Observation of Gravitational Waves from the Coalescence of a* $2.5-4.5\ M_\odot$ *Compact Object and a Neutron Star*. 2024. arXiv: 2404.04248 [astro-ph.HE] (cit. on p. 19).

[52] W. M. Farr et al. "THE MASS DISTRIBUTION OF STELLAR-MASS BLACK HOLES". In: *The Astrophysical Journal* 741.2 (Oct. 2011), p. 103. ISSN: 1538-4357. DOI: 10.1088/0004-637x/741/2/103. Available from: http://dx.doi.org/10.1088/0004-637X/741/2/103 (cit. on p. 19).

[53] LIGO Scientific Collaboration. *LIGO Detections*. Accessed 10 April 2024. Available from: https://www.ligo.org/detections.php (cit. on p. 19).

[54] S. Babak et al. "A template bank to search for gravitational waves from inspiralling compact binaries: I. Physical models". In: *Classical and Quantum Gravity* 23.18 (Aug. 2006), pp. 5477–5504. ISSN: 1361-6382. DOI: 10.1088/0264-9381/23/18/002. Available from: http://dx.doi.org/10.1088/0264-9381/23/18/002 (cit. on p. 20).

[55] P. Amaro-Seoane et al. *Laser Interferometer Space Antenna*. 2017. arXiv: 1702.00786 [astro-ph.IM] (cit. on p. 20).

[56] M. Maggiore et al. "Science case for the Einstein telescope". In: *Journal of Cosmology and Astroparticle Physics* 2020.03 (Mar. 2020), pp. 050–050. ISSN: 1475-7516. DOI: 10.1088/1475-7516/2020/03/050. Available from: http://dx.doi.org/10.1088/1475-7516/2020/03/050 (cit. on p. 20).

[57] Y. LeCun, Y. Bengio, and G. Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539 (cit. on p. 21).

[58] W. Mcculloch and W. Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147 (cit. on p. 21).

[59] Rosenblatt, Frank. *The Perceptron: A Perceiving and Recognizing Automaton*. Tech. Rep. 85-460-1. Cornell Aeronautical Laboratory, 1957 (cit. on p. 21).

[60] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541 (cit. on p. 22).

[61] M. Raissi, P. Perdikaris, and G. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. Available from: https://www.sciencedirect.com/science/article/pii/S0021999118307125 (cit. on p. 22).

[62] Logic Fruit Technologies. *FPGA Design Architecture and Applications*. Accessed 17 April 2024. Available from: https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/ (cit. on p. 23).

[63] Vaughn Betz. *FPGA Architecture*. Accessed 17 April 2024. Available from: https://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html (cit. on p. 23).

[64] Xilinx, Inc. *Xilinx Vivado Design Suite*. Accessed 17 April 2024. Available from: https://www.xilinx.com/products/design-tools/vivado.html (cit. on p. 23).

[65] Xilinx, Inc. *What is an FPGA?* Accessed 17 April 2024. Available from: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html (cit. on p. 23).

[66] CERN. *From capturing collisions to avoiding them*. https://home.cern/news/news/knowledge-sharing/capturing-collisions-avoiding-them. Accessed 04 June 2024. 2019 (cit. on p. 23).

[67] G. Baltus et al. "Convolutional neural networks for the detection of the early inspiral of a gravitational-wave signal". In: *Phys. Rev. D* 103 (10 May 2021), p. 102003. DOI: 10.1103/PhysRevD.103.102003 (cit. on pp. 24, 28).

[68] G. Baltus et al. "Convolutional neural network for gravitational-wave early alert: Going down in frequency". In: *Physical Review D* 106.4 (Aug. 2022). ISSN: 2470-0029. DOI: 10.1103/physrevd.106.042002. Available from: http://dx.doi.org/10.1103/PhysRevD.106.042002 (cit. on pp. 24, 26, 33, 47, 62, 70, 75).

[69] A. van den Oord et al. *WaveNet: A Generative Model for Raw Audio*. 2016. arXiv: 1609.03499 [cs.SD] (cit. on pp. 24, 39, 42, 70).

[70] C. Szegedy et al. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV] (cit. on pp. 24, 40).

[71] K. He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV] (cit. on pp. 24, 41, 42).

[72] N. Rahman et al. "Bengali Music Genre Classification Using WaveNet". In: Dec. 2023, pp. 1–6. DOI: 10.1109/ICCIT60459.2023.10441498 (cit. on p. 24).

[73] J. M. Duarte et al. "Low-latency machine learning inference on FPGAs". In: 2019. Available from: https://api.semanticscholar.org/CorpusID:209337198 (cit. on p. 24).

[74] R. Sang, Q. Liu, and Q.-J. Zhang. "FPGA-based acceleration of neural network training". In: July 2016, pp. 1–2. DOI: 10.1109/NEMO.2016.7561676 (cit. on p. 24).

[75] E. E. Khoda et al. *Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml*. 2022. arXiv: 2207.00559 [cs.LG] (cit. on p. 24).

[76] Z. Que et al. "LL-GNN: Low Latency Graph Neural Networks on FPGAs for High Energy Physics". In: *ACM Transactions on Embedded Computing Systems* 23.2 (Mar. 2024), pp. 1–28. ISSN: 1558-3465. DOI: 10.1145/3640464. Available from: http://dx.doi.org/10.1145/3640464 (cit. on p. 24).

[77] Intel Corporation. *FPGA and GPU Solutions*. Accessed 18 April 2024. Available from: https://www.intel.com/content/www/us/en/artificial-intelligence/programmable/fpga-gpu.html (cit. on p. 24).

[78] A. Akhshi et al. "A template-free approach for waveform extraction of gravitational wave events". In: *Scientific Reports* 11 (2020). Available from: https://api.semanticscholar.org/CorpusID:237499979 (cit. on p. 26).

[79] S. Isoyama, R. Sturani, and H. Nakano. "Post-Newtonian Templates for Gravitational Waves from Compact Binary Inspirals". In: *Handbook of Gravitational Wave Astronomy*. Springer Singapore, 2021, pp. 1–49. ISBN 9789811547027. DOI: 10.1007/978-981-15-4702-7_31-1. Available from: http://dx.doi.org/10.1007/978-981-15-4702-7_31-1 (cit. on p. 26).

[80] J. Aasi et al. "Advanced LIGO". In: *Classical and Quantum Gravity* 32.7 (Mar. 2015), p. 074001. ISSN: 1361-6382. DOI: 10.1088/0264-9381/32/7/074001. Available from: http://dx.doi.org/10.1088/0264-9381/32/7/074001 (cit. on p. 31).

[81] C. M. Biwer et al. "PyCBC Inference: A Python-based Parameter Estimation Toolkit for Compact Binary Coalescence Signals". In: *Publications of the Astronomical Society of the Pacific* 131.996 (Jan. 2019), p. 024503. ISSN: 1538-3873. DOI: 10.1088/1538-3873/aaef0b (cit. on p. 31).

[82] T. E. Oliphant. *NumPy: A Guide to NumPy*. Trelgol Publishing, USA. 2006 (cit. on p. 32).

[83] NVIDIA Corporation. *NVIDIA Tesla V100*. Accessed 20 March 2024. Available from: https://www.nvidia.com/en-gb/data-center/tesla-v100/ (cit. on p. 32).

[84] A. Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG] (cit. on pp. 33, 85, 86).

[85] S. Ioffe and C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG] (cit. on p. 33).

[86] S. Kiranyaz et al. "1D convolutional neural networks and applications: A survey". In: *Mechanical Systems and Signal Processing* 151 (2021), p. 107398. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2020.107398. Available from: https://www.sciencedirect.com/science/article/pii/S0888327020307846 (cit. on p. 33).

[87] vdumoulin. *Convolutional Arithmetic*. Accessed 22 April 2024. Available from: https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md (cit. on p. 34).

[88] V. Nair and G. E. Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN 9781605589077 (cit. on p. 35).

[89] PyTorch. *PyTorch Documentation - torch.nn.MaxPool1d*. Accessed 22 April 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.MaxPool1d.html (cit. on p. 35).

[90] PyTorch. *PyTorch Documentation - torch.flatten*. Accessed 22 April 2024. Available from: https://pytorch.org/docs/stable/generated/torch.flatten.html (cit. on p. 35).

[91] PyTorch. *PyTorch Documentation - torch.nn.Linear*. Accessed 22 April 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html (cit. on p. 36).

[92] Sergey Zagoruyko. *PyTorchviz GitHub Repository*. GitHub repository. Accessed 30 April 2024. Available from: https://github.com/szagoruyko/pytorchviz (cit. on p. 36).

[93] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG] (cit. on p. 36).

[94] PyTorch. *PyTorch Documentation - torch.nn.CrossEntropyLoss*. Accessed 21 April 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html (cit. on p. 37).

[95] PyTorch. *PyTorch Documentation - torch.nn.BCEWithLogitsLoss*. Accessed 24 April 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html (cit. on p. 37).

[96] W. Wei and E. Huerta. “Gravitational wave denoising of binary black hole mergers with deep learning”. In: *Physics Letters B* 800 (Jan. 2020), p. 135081. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2019.135081. Available from: http://dx.doi.org/10.1016/j.physletb.2019.135081 (cit. on p. 39).

[97] W. Wei et al. “Deep learning ensemble for real-time gravitational wave detection of spinning binary black hole mergers”. In: *Physics Letters B* 812 (Jan. 2021), p. 136029. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2020.136029. Available from: http://dx.doi.org/10.1016/j.physletb.2020.136029 (cit. on pp. 39, 42).

[98] S. Hochreiter and J. Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735 (cit. on p. 39).

[99] PyTorch. *PyTorch Documentation - torch.nn.Tanh*. Accessed 26 April 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.Tanh.html (cit. on p. 39).

[100] PyTorch. *PyTorch Documentation - torch.nn.Sigmoid*. Accessed 28 April 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html (cit. on p. 39).

[101] K. He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV] (cit. on p. 43).

[102] I. Loshchilov and F. Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: 1711.05101 [cs.LG] (cit. on p. 44).

[103] PyTorch. *PyTorch Documentation - torch.optim.AdamW*. Accessed 29 April 2024. Available from: https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html (cit. on p. 44).

[104] T. Akiba et al. *Optuna: A Next-generation Hyperparameter Optimization Framework*. 2019. arXiv: 1907.10902 [cs.LG] (cit. on p. 46).

[105] Y. Bengio et al. “Curriculum learning”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML ’09. Association for Computing Machinery, 2009, pp. 41–48. ISBN

9781605585161. DOI: 10.1145/1553374.1553380. Available from: https://doi.org/10.1145/1553374.1553380 (cit. on p. 46).

[106] M. B. Schäfer et al. "Training strategies for deep learning gravitational-wave searches". In: *Physical Review D* 105.4 (Feb. 2022). ISSN: 2470-0029. DOI: 10.1103/physrevd.105.043002. Available from: http://dx.doi.org/10.1103/PhysRevD.105.043002 (cit. on p. 47).

[107] Advanced Micro Devices, Inc. *AMD Kria KV260 Vision Starter Kit*. Accessed 27 March 2024. Available from: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html (cit. on pp. 48, 50, 80).

[108] Advanced Micro Devices, Inc. *AMD Vitis AI Documentation*. Accessed 02 April 2024. Available from: https://docs.amd.com/r/3.0-English/ug1414-vitis-ai (cit. on pp. 48, 80).

[109] Advanced Micro Devices, Inc. *AMD DPU Core Overview*. Accessed 02 May 2024. Available from: https://docs.amd.com/r/en-US/pg338-dpu/Core-Overview (cit. on pp. 48, 49).

[110] Advanced Micro Devices, Inc. *AMD DPU Hardware Architecture*. Accessed 04 May 2024. Available from: https://docs.amd.com/r/en-US/pg338-dpu/Hardware-Architecture (cit. on pp. 48, 49).

[111] Advanced Micro Devices, Inc. *Architecture of the DPUCZDX8G*. Accessed 04 May 2024. Available from: https://docs.amd.com/r/en-US/pg338-dpu/Architecture-of-the-DPUCZDX8G (cit. on p. 48).

[112] TechPowerUp. *AMD EPYC 7551P*. https://www.techpowerup.com/cpu-specs/epyc-7551p.c1929. Accessed 23 June 2024. 2024 (cit. on p. 50).

[113] ServeTheHome. *AMD EPYC 7551P Benchmarks and Review: A 32 Core Value Monster*. https://www.servethehome.com/amd-epyc-7551p-benchmarks-and-review-a-32-core-value-monster/. Accessed 23 June 2024. 2018 (cit. on p. 50).

[114] NVIDIA. *GeForce GTX 1080 Specifications*. https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1080/specifications/. Accessed 09 June 2024. 2024 (cit. on p. 50).

[115] A. Peters. *NVIDIA GeForce GTX 1080 release date, price and why it's the best gaming card ever*. https://www.pocket-lint.com/games/news/nvidia/137557-nvidia-geforce-gtx-1080-release-date-price-and-why-it-s-the-best-gaming-card-ever/. Accessed 09 June 2024. 2016 (cit. on p. 50).

[116] PyTorch Contributors. *torch.nn.Hardsigmoid*. Accessed 08 May 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.Hardsigmoid.html (cit. on p. 52).

[117] PyTorch Contributors. *torch.nn.Hardtanh*. Accessed 08 May 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.Hardtanh.html (cit. on p. 54).

[118] Advanced Micro Devices, Inc. *AMD Vitis AI Documentation - Operators Supported by PyTorch*. Accessed 2024. Available from: https://docs.amd.com/r/3.0-English/ug1414-vitis-ai/Operators-Supported-by-PyTorch (cit. on pp. 54, 66, 67, 86).

[119] PyTorch Contributors. *torch.nn.ReLU6*. Accessed 2024. Available from: https://pytorch.org/docs/stable/generated/torch.nn.ReLU6.html (cit. on p. 54).

[120] PyTorch Contributors. *PyTorch Quantization*. Accessed 06 May 2024. Available from: https://pytorch.org/docs/stable/quantization.html (cit. on p. 56).

[121] A. Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. arXiv: 2103.13630 [cs.CV] (cit. on p. 56).

[122] D. Przewlocka-Rus et al. *Power-of-Two Quantization for Low Bitwidth and Hardware Compliant Neural Networks*. 2022. arXiv: 2203.05025 [cs.LG] (cit. on p. 56).

[123] Xilinx, Inc. *Vitis-AI Quantization Configuration*. Accessed 07 May 2024. Available from: https://github.com/Xilinx/Vitis-AI/blob/3.0/src/vai_quantizer/vai_q_pytorch/doc/Quant_Config.md (cit. on p. 56).

[124] D. Miyashita, E. H. Lee, and B. Murmann. *Convolutional Neural Networks using Logarithmic Data Representation*. 2016. arXiv: 1603.01025 [cs.NE] (cit. on p. 56).

[125] Facebook Research. *Flop Counting with fvcore*. GitHub repository. Accessed 14 May 2024. Available from: https://github.com/facebookresearch/fvcore/blob/main/docs/flop_count.md (cit. on p. 59).

[126] Facebook Research. *Facebook Research fvcore*. GitHub repository. Accessed 2024. Available from: https://github.com/facebookresearch/fvcore (cit. on p. 59).

[127] PyPI. *pyJoules: A Python library for energy measurements*. Website. Accessed 27 March 2024. Available from: https://pypi.org/project/pyJoules/ (cit. on p. 65).

[128] Xilinx. *xmutil: Xilinx Machine Learning Utility*. https://github.com/Xilinx/xmutil. Accessed 30 May 2024. 2024 (cit. on p. 67).

[129] Xilinx. *xlnx_platformstats: Xilinx Platform Statistics*. https://github.com/Xilinx/xlnx_platformstats. Accessed 30 May 2024. 2024 (cit. on p. 67).

[130] Statista. *Energy Prices in the U.S.* https://www.statista.com/topics/6337/energy-prices-in-the-us/#topicOverview. Accessed 23 June 2024. 2024 (cit. on p. 69).

[131] I. Corporation. *Getting Started with FPGA Training*. https://www.intel.com/content/www/us/en/support/programmable/support-resources/fpga-training/getting-started.html. Accessed 21 June 2024. 2024 (cit. on p. 71).

[132] hls4ml. *hls4ml: Machine Learning Inference Made Fast and Easy*. https://fastmachinelearning.org/hls4ml/index.html. Accessed 21 June 2024. 2024 (cit. on p. 71).

[133] S. J. Reddi, S. Kale, and S. Kumar. *On the Convergence of Adam and Beyond*. 2019. arXiv: 1904.09237 [cs.LG] (cit. on p. 77).

[134] Advanced Micro Devices, Inc. *AMD Kria KV260 Vision Starter Kit - Getting Started*. Accessed 28 March 2024. Available from: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit/getting-started-ubuntu/getting-started.html (cit. on pp. 80, 82).

[135] Xilinx, Inc. *Xilinx Vitis-AI Quickstart Documentation*. Accessed 27 March 2024. Available from: https://xilinx.github.io/Vitis-AI/3.0/html/docs/quickstart/mpsoc.html (cit. on p. 80).

[136] Xilinx, Inc. *Xilinx KV260 DPU Download Page*. Accessed 02 April 2024. Available from: https://www.xilinx.com/member/forms/download/design-license-xef.html?filename=xilinx-kv260-dpu-v2022.2-v3.0.0.img.gz (cit. on p. 80).

[137] balena. *balenaEtcher*. Accessed 27 March 2024. Available from: https://etcher.balena.io (cit. on p. 81).

[138] SSH.COM. *SSH Protocol*. Accessed 02 April 2024. Available from: https://www.ssh.com/academy/ssh/protocol (cit. on p. 81).

[139] Xilinx, Inc. *Xilinx Vitis AI*. Accessed 28 March 2024. Available from: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html (cit. on p. 83).

[140] Xilinx, Inc. *Xilinx Vitis-AI GitHub Repository*. Accessed 28 March 2024. Available from: https://github.com/Xilinx/Vitis-AI (cit. on p. 83).

[141] Docker, Inc. *Docker*. Accessed 02 April 2024. Available from: https://www.docker.com (cit. on p. 83).

[142] Docker, Inc. *Install Docker Engine on Ubuntu*. Accessed 02 April 2024. Available from: https://docs.docker.com/engine/install/ubuntu/ (cit. on p. 83).

[143] Docker, Inc. *Linux post-installation steps*. Accessed 02 April 2024. Available from: https://docs.docker.com/engine/install/linux-postinstall/ (cit. on p. 84).

[144] NVIDIA Corporation. *What is CUDA?* Accessed 02 April 2024. Available from: https://nvidia.custhelp.com/app/answers/detail/a_id/2132/~/what-is-cuda%3F (cit. on p. 84).

[145] NVIDIA Corporation. *NVIDIA CUDA GPUs*. Accessed 02 April 2024. Available from: https://developer.nvidia.com/cuda-gpus (cit. on p. 84).

[146] TechPowerUp. *GeForce MX330 GPU Specifications*. Accessed 02 April 2024. Available from: https://www.techpowerup.com/gpu-specs/geforce-mx330.c3493 (cit. on p. 84).

[147] Conda Documentation Team. *Conda Documentation*. Accessed 03 April 2024. Available from: https://docs.conda.io/en/latest/ (cit. on pp. 85, 86).

[148] Xilinx, Inc. *Xilinx Vitis AI Documentation - Model Development Workflow - Model Inspector*. Accessed 04 April 2024. Available from: https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-model-development.html#model-inspector (cit. on p. 86).

[149] Advanced Micro Devices, Inc. *AMD Vitis AI Documentation - Currently Supported Operators*. Accessed 22 April 2024. Available from: https://docs.amd.com/r/3.0-English/ug1414-vitis-ai/Currently-Supported-Operators (cit. on p. 86).

[150] Xilinx, Inc. *Xilinx Vitis AI GitHub Repository - Inspector Tutorial*. GitHub repository. Accessed 04 April 2024. Available from: https://github.com/Xilinx/Vitis-AI/blob/3.0/examples/vai_quantizer/pytorch/inspector_tutorial.ipynb (cit. on p. 86).

[151] AMD. *Error Codes - Vitis AI Documentation*. https://docs.amd.com/r/3.0-English/ug1414-vitis-ai/Error-Codes. Accessed 02 June 2024. 2024 (cit. on p. 87).

[152] Xilinx. *Vitis AI Quantizer for PyTorch - README*. https://github.com/Xilinx/Vitis-AI/blob/master/src/vai_quantizer/vai_q_pytorch/README.md. Accessed 02 June 2024. 2024 (cit. on p. 87).

Further Reading

[153] Canonical Ltd. *Ubuntu for AMD*. Accessed 27 March 2024. Available from: https://ubuntu.com/download/amd.

[154] NVIDIA Corporation. *NVIDIA GeForce MX330 Gaming Laptops*. Accessed 28 March 2024. Available from: https://www.nvidia.com/nl-nl/geforce/gaming-laptops/mx-330/.

[155] Xilinx, Inc. *Xilinx Vitis AI Documentation - FAQ - What layers are supported for hardware acceleration?* Accessed 04 April 2024. Available from: https://xilinx.github.io/Vitis-AI/3.0/html/docs/reference/faq.html#what-layers-are-supported-for-hardware-acceleration.

[156] A. Ali et al. "Bayesian Inference on Gravitational Waves". In: *Pakistan Journal of Statistics and Operation Research* 11 (Dec. 2015), p. 645. DOI: 10.18187/pjsor.v11i4.1053.

[157] Y.-Z. Fan et al. "Maximum gravitational mass $M_T OV = 2.25^{+0.08}_{-0.07} M_\odot$ inferred at about 3% precision with multimessenger data of neutron stars". In: *Phys. Rev. D* 109 (4 Feb. 2024), p. 043052. DOI: 10.1103/PhysRevD.109.043052. Available from: https://link.aps.org/doi/10.1103/PhysRevD.109.043052.

[158] D. Lopez-Bernal et al. "Education 4.0: Teaching the Basics of KNN, LDA and Simple Perceptron Algorithms for Binary Classification Problems". In: *Future Internet* 13 (July 2021), p. 193. DOI: 10.3390/fi13080193.

[159] F. Acernese et al. *Virgo Detector Characterization and Data Quality during the O3 run*. 2022. arXiv: 2205.01555 [gr-qc].

[160] A. Shenfield and M. Howarth. "A Novel Deep Learning Model for the Detection and Identification of Rolling Element-Bearing Faults". In: *Sensors (Basel, Switzerland)* 20 (Sept. 2020). DOI: 10.3390/s20185112.

[161] GreenOrange. *Introduction to Machine Learning Loss Functions (Chinese)*. Accessed 22 April 2024. Available from: https://www.cnblogs.com/GreenOrange/p/15628140.html.

[162] D. M. Macleod et al. "GWpy: A Python package for gravitational-wave astrophysics". In: *SoftwareX* 13 (2021), p. 100657. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.

2021.100657. Available from: https://www.sciencedirect.com/science/article/pii/S2352711021000029.

[163] T. L. S. Collaboration and the Virgo Collaboration. *Advanced Virgo O4 High Frequency Noise and Background Characterization*. LIGO Document T2000012. https://dcc.ligo.org/DocDB/0165/T2000012/002/avirgo_O4high_NEW.txt. 2020.

[164] T. L. S. Collaboration and the Virgo Collaboration. *Advanced LIGO O4 High Frequency Noise and Background Characterization*. LIGO Document T2000012. https://dcc.ligo.org/DocDB/0165/T2000012/002/aligo_O4high.txt. 2020.

[165] AMD. *AMD EPYC 7702P CPU*. Website. Accessed 23 May 2024. Available from: https://www.amd.com/en/products/cpu/amd-epyc-7702p.

[166] Dell. *Dell NVIDIA Quadro GV100 32GB HBM2 Full-Height PCIe 3.0x16 4 DP Graphics Card*. Website. Accessed 23 May 2024. Available from: https://www.dell.com/nl-nl/shop/dell-nvidia-quadro-gv100-32-gb-hbm2-volledige-hoogte-pcie-30x16-4-dp-grafische-kaart/apd/490-benn/grafische-kaarten.

[167] Microway, Inc. *NVIDIA Tesla V100 Price Analysis*. https://www.microway.com/hpc-tech-tips/nvidia-tesla-v100-price-analysis/. Accessed 09 June 2024. 2018.