KU LEUVEN

Machine learning algorithms for the conservative-to-primitive conversion in relativistic hydrodynamics

Thibeau Wouters

Thesis submitted for the degree of Master of Science in Artificial Intelligence, option Engineering and Computer Science

Supervisors:
Prof. dr. Tjonnie G.F. Li
Prof. dr. ir. Johan Suykens

Assessor:
Dr. Kaze Wong

Assistant-supervisors:
Arthur Offermans
David Winant

Academic year 2023 – 2024

Bring forward what is true. Write it so that it is clear.
Ludwig Boltzmann

Preface

Confucius once said: *"isn't it a pleasure to study and practice what you have learned?"*, although probably in Chinese instead of English. After six year's worth of pleasant studies, this piece of work in front of you is, if all goes well, the final manuscript I will submit through KU Leuven's student platform.

First, I would like to thank those that helped me along the way during this research project. Tjonnie, thanks for supervising me, inviting master students to the journal club and the inspiring brainstorm sessions that continuously challenged me. Thanks for introducing me to my future boss and helping me make the first steps into an academic research career. I look forward to many more discussions, either in Leuven or in Utrecht.

Thanks to professor Suykens for reading my thesis and the valuable feedback and comments provided during the intermediate presentation. Thanks for the lectures which made me think critically about machine learning techniques.

Arthur, thanks for the fruitful discussions on the deep learning and Fortran code, double-checking my methods and results and helping me to better understand `Gmunu`. Thanks for finding that final obscure bug in the GRHD code. Thanks for providing the data of the convergence runs. David, thanks for the discussions on neural networks and other machine learning techniques. To both of you, thanks for reading through the final draft version. Thanks to Patrick Cheong for your Fortran expertise and improvement of the code, the feedback along the way, and for providing guidance through the jungle of module files of `Gmunu`. Thanks to Tobias Dieselhorst for interesting discussions on the topic and sending the Fortran code to exactly solve the shocktube.

Thanks to my assessor Kaze Wong for taking the time to read my thesis. I look forward to interesting discussions and insights!

Thanks to everyone at the ITF for allowing me yet again to have an office. Thanks to my office mates Milan, Vincent. Thanks to Simon for organizing the journal club.

Finally, an infinite amount of gratitude and love to my family and closest friends. Eline, my love and soulmate, thanks once again for believing in me and encouraging me at every moment, easy or hard, during the past year. I look forward to starting a new chapter in my life, but I could never imagine it become a perfect story without you by my side to share this adventure with.

Thibeau Wouters

Contents

Abstract

Future detections of gravitational waves originating from binary neutron star mergers or core-collapse supernovae offer the potential to gain unprecedented insights into the structure of matter at densities far beyond those probed by Earth-based experiments. In order to be able to identify the correct equation of state of matter, a template bank of waveforms has to be generated by general relativistic magnetohydrodynamics simulations. However, state-of-the-art solvers are slowed down by the conservative-to-primitive transformation, a central algorithmic step in any relativistic hydrodynamics solver. We investigate the potential of three machine learning algorithms to improve existing conservative-to-primitive schemes. We find that fully replacing either the conservative-to-primitive transformation or the evaluation of the equation of state by a machine learning model is unable to provide any significant advantage. We propose a novel, hybrid scheme that unifies machine learning and state-of-the-art schemes, resulting in an acceleration of numerical solvers by up to 25% for general relativistic magnetohydrodynamics simulations involving microphysical equations of state, without compromising accuracy or robustness.

List of Figures

List of Tables

Nomenclature

List of Abbreviations

C2P conservative-to-primitive transformation

DL deep learning

EOS equation of state

GR general relativity

GRHD general relativistic hydrodynamics

GRMHD general relativistic magnetohydrodynamics

GW gravitational wave

HPC high-performance computing

ML machine learning

MLP multilayer perceptron

NN neural network

NNC2P neural network replacement of the C2P conversion

NNEOS neural network replacement of the EOS

NR Newton-Raphson rootfinding algorithm

ONNX open neural network exchange

P2C primitive-to-conservative transformation

ReLU rectified linear unit

TTC time-to-completion

List of Symbols

α lapse function in the $3+1$ decomposition of space-time

β_i shift vector in the $3+1$ decomposition of space-time

Γ adiabatic index

γ_{ij} spatial three-metric in the $3+1$ decomposition of space-time

ρ mass density of fluid

τ conserved energy density relative to D

c_s^2 square of the speed of sound

D conserved density

E conserved energy density

h enthalpy

p specific internal energy

S_i conserved momentum density

v^i fluid 3-velocity

W Lorentz factor

Y_e electron fraction

Conventions
We use geometrized Heaviside–Lorentz units, for which the speed of light c, gravitational constant G, solar mass $M_\odot$, vacuum permittivity ϵ_0, and vacuum permeability μ_0 are all equal to one. By default, we use base-10 for the logarithm, *i.e.* $\log(\cdot) \equiv \log_{10}(\cdot)$. Latin indices on vectors that depend on space and time refer to spatial components (*e.g.* x^i, $i = 1, 2, 3$), whereas Greek indices refer to space-time components (*e.g.* x^μ, $\mu = 0, 1, 2, 3$).

Chapter 1

Introduction

Ever since humans can stand on their two feet and gaze into the night sky, we have wondered about the origin of our existence and the matter that surrounds us. This fundamental question has ignited the curiosity of countless generations throughout the ages. Over time, we have come to understand that matter is made out of atoms, which themselves consist of electrons and nuclei. Yet, we still lack a complete understanding of the origin of the wide variety of nuclei observed in nature. The intricate secrets of the theory of nuclear physics are told by the most cataclysmic events in the cosmos, the explosions of massive stars. These explosions, known as supernovae, achieve extremely high densities and temperatures, leading to the creation of heavy elements. Supernovae leave behind a collapsed core, a neutron star, which is one of the densest known objects in the universe.

While astronomers have pointed their telescopes at supernovae and neutron stars in the past, the mechanisms that trigger supernovae and the composition of neutron star matter are difficult to grasp from electromagnetic radiation alone, and other observational channels are required to complete the picture. Gravitational waves provide such a novel observational channel. Predicted by Albert Einstein's famous theory of gravity, general relativity, gravitational waves are a form of radiation emitted by violent, dynamical events which travel as ripples through the fabric of space-time. Unlike electromagnetic radiation, gravitational waves are unobstructed by matter and can travel all the way from the inner regions of supernovae and neutron stars towards Earth. Gravitational waves have been directly observed for the first time in 2015, almost a century after their prediction. A new emerging field of astrophysics, known as multi-messenger astrophysics, tries to gain more insight in the nature of matter at high densities by combining different observational channels, including gravitational waves.

Gravitational waves are detected by relying on the procedure of matched filtering, where an incoming signal is compared to a large bank of templates of waveforms, computed for different parameter settings of the source system. In case the source is a binary system of compact objects that spiral toward each other and eventually collide, these parameters can *e.g.* specify the masses of the objects and their rotations relative to each other and their plane of motion. For black holes, these large template

banks have been constructed in the past and the parameter space that determines the waveform is relatively well-understood. For events involving neutron stars and supernovae, the description and behaviour of matter at high density influences the resulting waveform. Therefore, this description has to be taken into account when generating the template banks used in the matched filtering process. Theoretical models account for this microphysical description by providing relations among the physical variables, which is known as the equation of state.

Unfortunately, the equation of state of neutron stars is not yet known, since Earth-based experiments are unable to probe the high density regime realized in neutron stars and supernovae. Currently, there exist different propositions of the equation of state, originating from various theoretical nuclear physics frameworks. Therefore, the waveforms of mergers of neutron stars and supernovae have a significant amount of variance due to the unknown equation of state. By analyzing the gravitational waves originating from events at high density, such as mergers of neutron stars and supernovae, we can gain valuable insights into the properties of dense nuclear matter. In particular, by comparing the observed gravitational wave signals to the theoretically predicted waveforms, we hope to identify the equation of state in the near future.

As such, theoretical models of relativistic hydrodynamics have to provide template banks of waveforms for each proposed equation of state. Since the theoretical models are intractable to compute analytically, one has to resort to numerical simulations of the systems of interest to compute the waveform. However, to obtain accurate waveforms, effects occurring at different length scales have to be properly taken into account, resulting in computationally demanding simulations. As a result, obtaining these template banks seems to be infeasible with the current state-of-the-art numerical solvers.

Therefore, to make progress in the field of multi-messenger astrophysics, we require optimized and accelerated relativistic hydrodynamics solvers. A central step in any relativistic hydrodynamics solver is a transformation from the conserved variables, which are the fluid variables measured in the laboratory frame, to the variables measured in the rest frame of the fluid, the primitive variables. This conservative-to-primitive transformation is a numerically expensive step, especially for simulations that model the microphysics of the system of interest. Moreover, this transformation has to be applied to each grid point and for every time step. To generate accurate waveforms more efficiently, a good starting point would be to optimize this crucial algorithmic step.

In this thesis, we explore the application of machine learning to improve the efficiency of the conservative-to-primitive transformation. Machine learning, a subfield of artificial intelligence, studies the development of numerical algorithms that are able to accomplish predefined tasks by learning from examples. The goal of a machine learning algorithm is to train a model that is able to extract meaningful patterns from provided data. After a machine learning model has been trained, it can be used to predict the values for similar, unseen cases. We aim to leverage this learning capability to construct a machine learning algorithm that is able to accelerate the conservative-to-primitive transformation.

1.1 Outline

This work is as organized as follows. In Chapter 2, we delve deeper into the motivation of our work and provide a more detailed discussion on the prospects and challenges of the field of gravitational waves and multi-messenger astrophysics.

In Chapter 3, we present the theoretical framework used to obtain waveforms of gravitational waves originating from neutron stars and supernovae, namely general relativistic magnetohydrodynamics. At the same time, we discuss the conservative-to-primitive transformation in more detail. Finally, this chapter discusses `Gmunu`, the numerical solver for which we designed and implemented machine learning algorithms.

Chapter 4 then provides an introduction to machine learning and the techniques that we have used to tackle the conservative-to-primitive problem. Our work mainly considered deep learning and artificial neural networks, such that we provide a comprehensive introduction to these methods. Since many hydrodynamics solvers are written in the Fortran programming language due to its efficiency for scientific computing, we present an extensive overview and discussion of using deep learning in Fortran.

In Chapter 5, we present three proposed solutions that aim to accelerate the conservative-to-primitive transformation. We discuss the numerical experiments we have conducted in the `Gmunu` solver to benchmark the performance of these methods against existing algorithms. Finally, Chapter 6 provides a discussion of our work and reflects on possible future extensions of the methods proposed.

Chapter 2

Gravitational waves

The most accurate theory of gravity to date is Albert Einstein's theory of general relativity, first formulated in 1915. Central to the theory of general relativity are the Einstein equations, which describe the relation between the curvature of space-time and matter. These equations predict the existence of gravitational waves, which were directly observed for the first time in 2015 and can deliver new insights into fundamental physics in the near future.

In this chapter, we aim to briefly introduce the basic concepts underlying the theory of general relativity and gravitational waves, as well as the prospects and challenges for the near future.

2.1 General relativity and the Einstein equations

The general theory of relativity, or simply *general relativity* (GR), is a theory of gravity that emerged out of the unification of gravity (as described by Newton) and the principles of electrodynamics and special relativity. Since a thorough introduction to GR is beyond the scope of this thesis, readers are referred to Refs. [1, 2] for more details and mathematical rigour.

Newton's theory of gravity assumed space and time to be separate and absolute concepts. In the theory of relativity, these notions are no longer absolute, but become relative to a specific observer, hence the origin of the name of the theory. Moreover, the clear separation between space and time disappears as well, and all four coordinates are collectively used to describe the universe as a *space-time*. That is, time t and space x^i, where $i = 1, 2, 3$ for three-dimensional spaces, are merged into one coordinate vector $x^\mu = (t, x^i)$, where $\mu = 0, 1, 2, 3$ accounts for all space-time coordinates. By convention, Latin indices refer to spatial components, whereas Greek indices refer to all space-time components. Vectors which carry four indices are hence also referred to as *four-vectors*.

In the theory of relativity, space-time is mathematically described as a manifold. These manifolds are equipped with a *metric* $g_{\mu\nu}$, which measures distances in space-time and describes the local geometry of the manifold. Moreover, it defines a symmetric bilinear transformation, the inner-product, on the space of four-vectors.

The inverse metric, which is the inverse of $g_{\mu\nu}$ viewed as a matrix, is written with upper indices $g^{\mu\nu}$. Without going into further details, we remark that indices of vectors and tensors (*i.e.*, multi-dimensional vectors) can be raised or lowered using the metric, *viz.*

$$V_\mu = g_{\mu\nu} V^\nu \,, \quad V^\mu = g^{\mu\nu} V_\nu \,, \tag{2.1}$$

where repeated indices are summed over, a convention known as the Einstein summation convention, *viz.*

$$V_\mu V^\mu \equiv \sum_{\mu=0}^{3} V_\mu V^\mu = V_0 V^0 + V_1 V^1 + V_2 V^2 + V_3 V^3 \,. \tag{2.2}$$

Using the metric, the inner-product can also be computed as

$$V_\mu V^\mu = V_\mu g^{\mu\nu} V_\nu = \sum_{\mu,\nu} g^{\mu\nu} V_\mu V_\nu = \sum_{\mu,\nu} g_{\mu\nu} V^\mu V^\nu \,. \tag{2.3}$$

We will encounter such an inner-product, for instance, in the computation of the square of the velocity and the Lorentz factor in relativistic hydrodynamics.

Importantly, manifolds in GR can curve and are not necessarily rigid and fixed. Their curvature can be expressed as a function of the metric and its derivatives, such that the metric fully specifies the space-time. For instance, the simplest metric consistent with relativity is the so-called *Minkowski metric* (usually denoted by $\eta_{\mu\nu}$ rather than $g_{\mu\nu}$) which describes a flat space-time without any curvature:

$$g_{\mu\nu} = \eta_{\mu\nu} \equiv \mathrm{diag}\,(-1, +1, +1, +1) \,. \tag{2.4}$$

Here, the positive eigenvalues are related to the spatial components, which gives the well-known Euclidean geometry, and the first entry refers to the time component.

Since manifolds in GR are not rigid, the metric is a dynamical quantity and obeys its own evolution equations. The central idea underlying these equations is neatly summarized by a famous quote ascribed to John Wheeler: "*Space-time tells matter how to move, matter tells space-time how to curve*" [3]. That is, GR describes the interplay between the curvature of space-time and the matter and energy sources present in the space-time. This relation is formalized by the famous *Einstein field equations* (or simply Einstein equations)

$$R_{\mu\nu} - \frac{1}{2} R g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu} \,, \tag{2.5}$$

where we explicitly wrote the physical constants for clarity. The Einstein equations constitute a set of sixteen non-linear, coupled differential equations, although not all sixteen components are independent of each other. The left hand side involves the metric, along with the Ricci tensor $R_{\mu\nu}$ and the Ricci scalar R, which are two quantities related to the curvature of the manifold and which can be expressed in terms of the metric. In short, the left hand side hence refers to space-time and its curvature. The right hand side, on the other hand, involves the *energy-momentum tensor* $T_{\mu\nu}$, which accounts for the energy and matter sources in the space-time. Readers who are unfamiliar with the mathematics of GR are encouraged to remember Eq. (2.5) simply as the mathematical translation of Wheeler's quote.

2.2 Theory of gravitational waves

The Einstein equations provide a mathematical framework in which we can study the interaction between matter and space-time for different systems. Of particular interest to us are systems that, through their dynamics, emit a type of radiation throughout space-time that is measurable far away from the source. This gravitational radiation, known as *gravitational waves* (GWs), was theoretically formulated shortly after the birth of GR itself and is currently being developed as a new and exciting experimental field. We briefly introduce GWs based on Refs. [4, 5].

Since it is infeasible to solve the Einstein equations analytically except for a handful of systems, GWs are best understood by incorporating approximations. For instance, one can assume that the background space-time is flat (with metric given by Eq. (2.4)) such that the GWs add a small perturbation $h_{\mu\nu}$ to this background, *viz.*

$$g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu} \,, \quad |h_{\mu\nu}| \ll 1 \,. \tag{2.6}$$

Rewriting the Einstein field equations up to first order in the perturbation, the solution of the GW can be written as

$$h_{\mu\nu} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & h_+ & h_\times & 0 \\ 0 & h_\times & -h_+ & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cos\left(\omega(t-z)\right) \,, \tag{2.7}$$

which represents a single plane wave travelling along the z-axis of the coordinate system. The wave has two polarizations, referred to as the plus and cross polarizations, with amplitudes h_+ and $h_\times$, respectively. Higher-order corrections can be obtained with the so-called post-Newtonian formalism. This gives a multipole expansion, which yields an expansion in the parameter v/c, where v is the typical velocity of the source and c the speed of light.

2.3 Detections and matched filtering

After briefly discussing the theory constituting GWs, we present an informal discussion of how GWs can be detected. For this, assume that our GW detector consists of a ring of test masses situated on a circle in the (x, y)-plane of a coordinate system. Suppose that a GW travelling along the z-direction passes over our detector. The waveform, in the linearized theory, is given by Eq. (2.7). The h_+, respectively $h_\times$, polarization of the GW distorts the ring into a periodic plus-like shape, respectively cross-like shape, hence explaining the name of the polarizations. These deformations are shown schematically in Figure 2.1. If the displacement of the particles is δL and the original separation is L, the *strain* of a GW is $\delta L/L$ and is precisely the quantity that GW detectors aim to detect. Actual detectors do not rely on a ring of test particles, but are interferometers which send light beams through two perpendicular arms. The detector outputs the phase shift of the light beams, recombined after travelling through the two arms, which depends on the relative distances of the two arms.

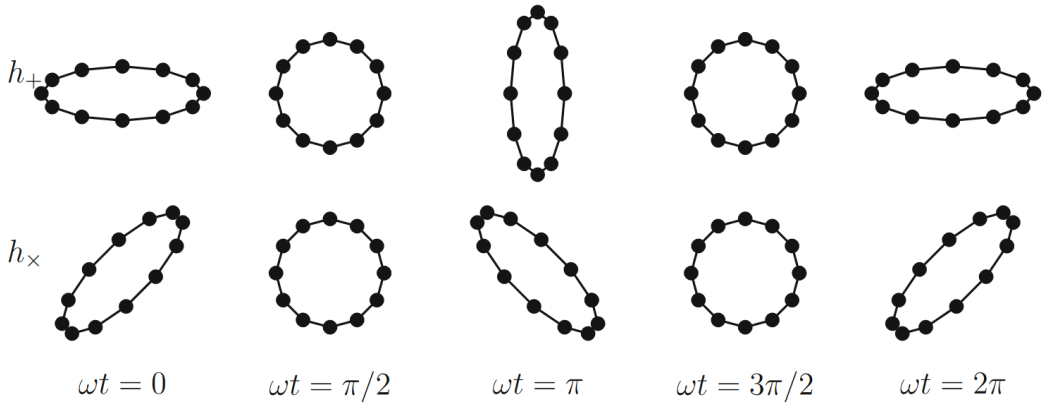

Figure 2.1: Deformations of a ring of test masses acting as an idealized detector system. The deformations due to the two GW polarizations h_+ and $h_\times$ are shown at various phases. Figure taken from [5].

Currently, the most sensitive interferometers are the LIGO detectors in Hanford and Livingston [6], that of Virgo in Italy [7] and Japan's KAGRA interferometer [8], collectively organized in the LVK collaboration.

Therefore, realistic GW detectors output a time series, the observed signal $s(t)$, which is the sum of the strain and noise:

$$s(t) = h(t) + n(t)\,. \tag{2.8}$$

Extracting the GW strain from the signal is a challenging task, since the detectors are flooded by a multitude of noise sources, ranging from quantum noise coming from the lasers to seismic noise. In fact, in realistic scenarios, the noise dominates over the actual GW signal. We briefly describe how GW signals are detected in practice. To be able to extract the GW signal, we need a reasonable guess of the waveform $h(t)$. GWs can be detected through the principle of *matched filtering*. In this process, we compute the quantity

$$\hat{s} = \int_{-\infty}^{+\infty} s(t)K(t)\,, \tag{2.9}$$

where K is some filter function. Signals are characterized by their signal-to-noise ratio (SNR) $\rho = S/N$, where S is the expected value of $\hat{s}$ when a GW is present and N is the root-mean-square value of $\hat{s}$ when the signal does not contain a wave. Hence, a signal should result in a sufficiently high SNR in order to confidently identify the presence of a GW. One can show that the filter function that maximizes the SNR when the signal contains a specific waveform is a function of the waveform itself. In other words, in using matched filtering to detect GWs, the best filter is provided by the wave itself.

Therefore, matched filtering requires prior knowledge on the waveforms $h(t)$. These have to be computed from theoretical frameworks and depend on several parameters, such as the distance to the source, or the masses of compact objects in a

binary system et cetera. A waveform for a specific combination of parameters hence gives a *template* $h(t;\theta)$ for the GW signal and filter to be used in the detection. To detect a GW and estimate the parameters of its source, we have to rely on large template banks. However, since GR is a complicated theory, constructing these template banks is infeasible analytically, and one has to rely on numerical simulations. For some systems, such as neutron stars and core-collapse supernovae, this requires us to model physics on a wide range of scales. Moreover, the dynamics of these systems depends on unknown physics, which increases the difficulty of creating a template bank.

The first direct[1] detection of GWs, and the dawn of gravitational wave astrophysics as an experimental field, was reported for the first time in 2016 [10]. The observed signal, called GW150914, is shown in Figure 2.2, after applying a few postprocessing steps.[2] The signal originates from a binary system of black holes that spiral towards each other, which increases the frequency and amplitude of the GW strain, and eventually merge into one, final black hole, which happens at $t = 0$ in Figure 2.2. As

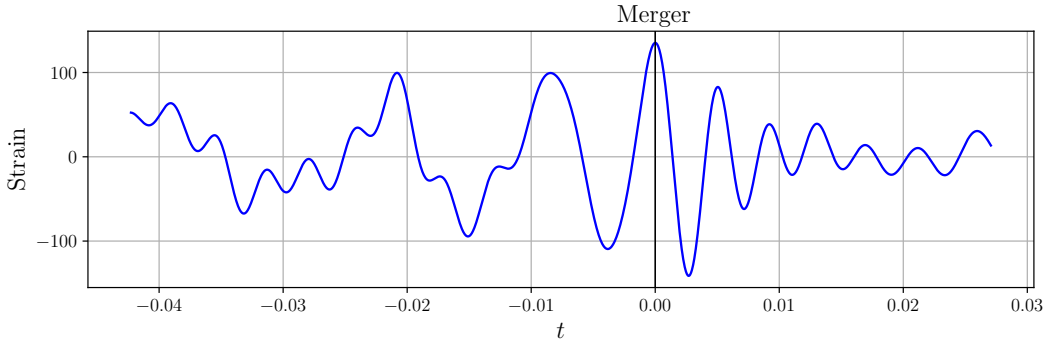

Figure 2.2: The postprocessed GW150914 signal with time of merger at $t = 0$.

a small demonstration of matched filtering, we determine an unknown parameter, namely the masses of the black holes. For simplicity, we assume that the spins of the black holes are aligned and that the black holes have an equal mass M, which we express in numbers of solar masses $M_\odot$. Figure 2.3 shows the theoretical waveforms for three different values of the parameter M, computed by a phenomenological model. These signals are our toy version of a template bank. After running the matched filtering process, we can overlay the theoretical waveforms with the signal and compute the SNR of each waveform. The results, shown in Figure 2.3, show that the signal is most likely due to the merger of two black holes with $M = 36M_\odot$. Indeed, a more thorough analysis revealed that the black holes have masses of around 29 and 36 solar masses. This analysis assumed that the spins of the black holes were aligned with the orbital angular momentum and used a template bank containing around 250 000 waveforms.

[1]The existence of GWs was first inferred indirectly from the study of the Hulse-Taylor binary pulsar system [9].

[2]The data used for the demonstration is obtained with the `PyCBC` tutorials from Ref. [11].

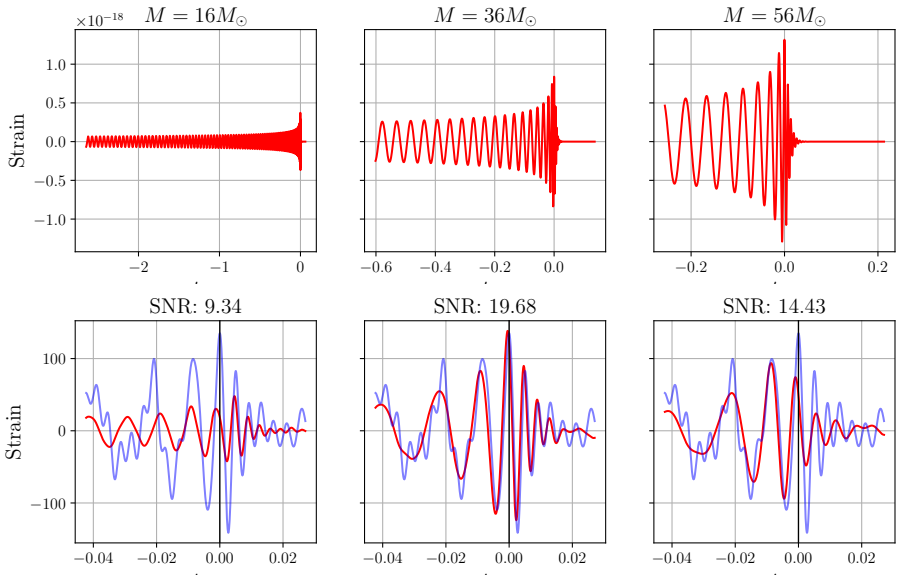

Figure 2.3: Top: Theoretical waveforms of GWs from a binary system of black holes with equal masses M. *Bottom*: Comparison between theoretical waveforms (red) and GW150914 signal (blue) with corresponding SNR.

2.4 Multi-messenger astrophysics

Up until today, around a hundred GWs have been observed, all of them generated by coalescences of binary systems of compact objects formed by heavy stars. The evolution of these heavy stars eventually results in a *supernova* explosion. In *core-collapse supernovae* (CCSNe), the core collapses and forms a compact, dense object such as a neutron star or a black hole. *Neutron stars* are among the smallest and densest stellar objects and, together with supernovae, harbour important information on the properties of nuclear matter at high densities. GWs originating from binary systems of neutron stars, and even of binary systems containing a black hole and a neutron star have been observed [12, 13].

In the near future, additional detectors such as Einstein Telescope, Cosmic Explorer and LISA will come online. Together with the upgraded versions of existing ones, the frequency range over which we will be able to detect GWs will be broadened and the number of detections will vastly increase. Moreover, GW sources will be located more accurately in the sky. Future GW detections can hence be combined with observations of the electromagnetic radiation originating from binary neutron star mergers or supernovae. In 2017, the first-ever example of such a joint detection of the gravitational and electromagnetic radiation of a binary neutron star coalescence led to the birth of the field of *multi-messenger astrophysics* [14].

Together with neutrinos, small and weakly-interacting subatomic particles, the combination of all these observational channels will allow us to study in great detail the dynamics of the core collapse of massive stars as well as the mechanism of the explosion [15]. It is currently not yet known which mechanism dominates the onset

of core-collapse supernovae. Different mechanisms depending on the microphysics, such as heating from neutrinos, have been proposed, and these results can also be influenced by rotation and magnetic fields. Since electromagnetic radiation mostly arises from the outer layers of the stars, the intricacies of these mechanisms, and our hope to understand them, are buried deep in the interior of these stars. Since the GWs of these events originate from the inner regions, they hold crucial information regarding these formation mechanisms.

In lack of experimental data, most of our understanding of CCSNe and their gravitational waveforms comes from simulations. While these simulations are indispensable, they come with considerable challenges [16]. The simulated waveforms are highly dependent on the explosion mechanism, and parameters such as the mass and rotation of the progenitor. For future detections, we require the simulations to fully take the theory of GR into account. However, most of the sophisticated simulations to date model gravity through effective potentials. These simulations also have a long duration, as the GW signal is influenced by the continued accretion of matter. Taking neutrino physics into account requires us to simulate the impact of microphysical effects, such as neutrino oscillations, on the waveform.

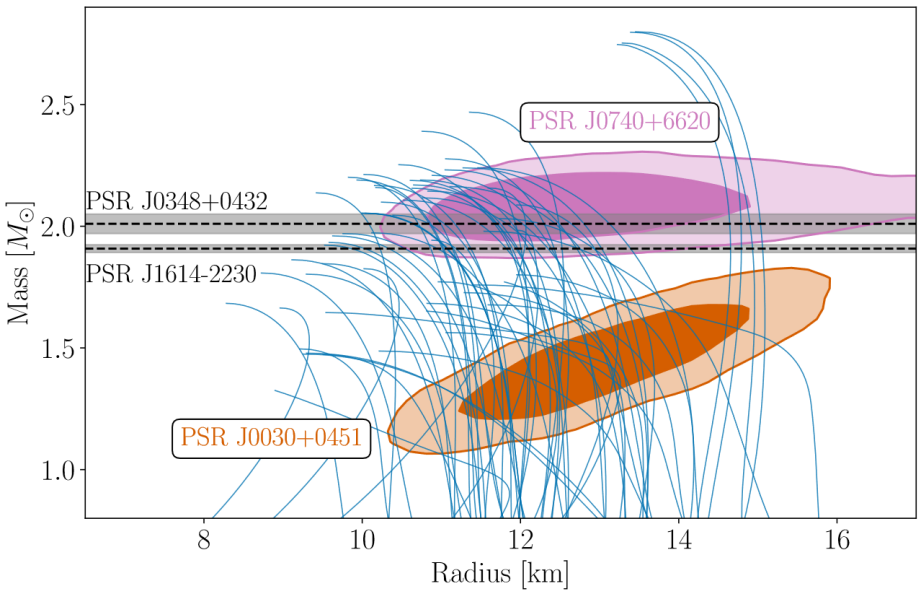

Figure 2.4: Mass-radius relationship of neutron stars for 65 different equations of state (blue lines), along with constraints by experiments (black dashed lines, contours). Figure taken from Ref. [17].

These challenges are not only limited to CCSNe, but also appear when simulating neutron stars. One currently open problem in nuclear physics is the precise description of dense nuclear matter, as experiments on Earth are unable to probe the density regimes explored by neutron stars. Therefore, neutron stars offer a natural laboratory to study matter at high density. This microscopic description, which is encoded in the nuclear *equation of state*, influences the observables of neutron star systems. For

instance, the relation between the mass and the radius of neutron stars depends on the equation of state [18]. Figure 2.4 shows the theoretical mass-radius relationships predicted by 65 different equations of state along with constraints from observations of pulsars. The experiments taken into account here are unable to identify a single equation of state, such that there are still several possible candidates for the equation of state today.

Especially in the parameter regions explored by mergers of neutron stars and CCSN, the equation of state is poorly constrained, and further work and observations are required to make significant progress [17, 19]. Different equations of state then lead to different dynamics in simulations and a plethora of gravitational waveforms to be generated. Future observations of GWs, when compared against such template banks of accurate waveforms, can potentially further improve the current constraints on the nuclear equation of state.

In short, the novel field of multi-messenger astrophysics could answer important unsolved problems regarding the properties of dense matter. However, the uncertainties and variance that the unknown equation of state introduces, create new challenges for numerical simulations. In this thesis, we wish to alleviate these problems by improving current simulations with machine learning. Before delving into our work, we present the theoretical framework that is able to model systems such neutron stars and supernovae within dynamical space-times.

Chapter 3

General relativistic magnetohydrodynamics

In the previous chapter, we introduced the field of multi-messenger astrophysics, its exciting prospects and its current challenges to be tackled. One of those challenges is to obtain accurate templates of GWs from simulations of neutron stars and supernovae. In these systems, one has to simulate both the fluid dynamics as well as the space-time metric. As such, these simulations take place in the framework of *general relativistic hydrodynamics* (GRHD). When electromagnetic fields are taken into account as well, we speak of *general relativistic magnetohydrodynamics* (GRMHD). This chapter provides the necessary background of the GRMHD framework and simulations. We formulate the conservative-to-primitive transformation, the central problem of this thesis, in more detail. Finally, we introduce the simulation code `Gmunu`, for which we developed deep learning methods to optimize the conservative-to-primitive transformation. This chapter is largely based on Refs. [20, 21].

3.1 Relativistic hydrodynamics equations

Let us first discuss numerical simulations of space-times assuming that the energy-momentum tensor (the right hand side of the Einstein equations) is fixed and known. Due to the complexity of GR, only a limited number of analytic solutions to the theory are currently known. Hence, for systems of astrophysical relevance, one has to resort to *numerical relativity* [22]. In the widely used $3+1$ formalism, space and time, which were unified into one entity in GR, are decomposed again. Specifically, space-time is sliced into 3D spacelike hypersurfaces Σ_t and a *spatial three-metric* γ_{ij} is defined on each hypersurface. The metric[1] is then written as

$$\mathrm{d}s^2 = -(\alpha^2 - \beta_i\beta^i)\,\mathrm{d}t^2 + 2\beta_i\,\mathrm{d}x^i\,\mathrm{d}t + \gamma_{ij}\,\mathrm{d}x^i\,\mathrm{d}x^j\,, \tag{3.1}$$

where α is called the *lapse function* and β_i the *shift vector*. Their geometric interpretation is shown in Figure 3.1.

[1] Here, we show the line element $\mathrm{d}s^2$, which is related to the space-time metric by $\mathrm{d}s^2 = g_{\mu\nu}\,\mathrm{d}x^\mu\,\mathrm{d}x^\nu$, where $\mathrm{d}x^\mu = (\mathrm{d}t, \mathrm{d}x, \mathrm{d}y, \mathrm{d}z)$ are differentials.

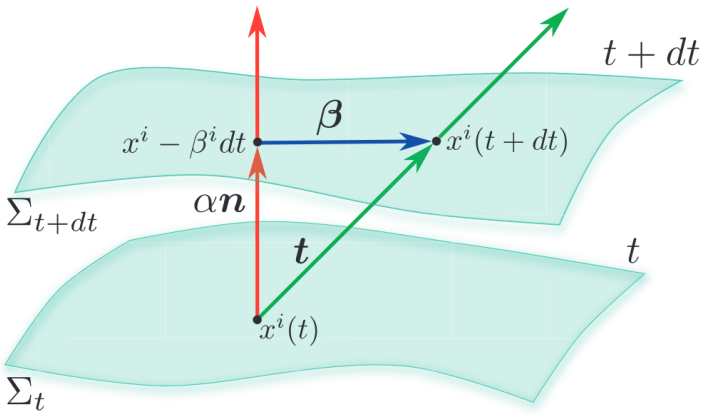

Figure 3.1: The $3 + 1$ decomposition of space-time. Hypersurfaces of constant coordinate time Σ_t are related by the four-vector $\boldsymbol{t}$ representing the direction of evolution of time, split into a timelike component perpendicular to Σ_t, $\alpha \boldsymbol{n}$, and a spacelike component, $\boldsymbol{\beta}$.

In the past decades, several schemes have been introduced that cast the Einstein equations into a form that is appropriate for numerical evolution [20]. One of the most well-known schemes is the ADM formulation, which applies the $3 + 1$ decomposition to the Einstein equations. This formulation splits the Einstein equations into a set of evolution equations and a set of constraint equations. Unfortunately, these ADM equations turn out to be weakly hyperbolic and are therefore not guaranteed to be well-posed. We will further consider numerical schemes below in the context of relativistic hydrodynamics.

So far, our discussion only considers evolution equations for the space-time metric, *i.e.* the left hand side of the Einstein equations. The right hand side of the Einstein equations involves the energy-momentum tensor, which describes the properties of matter and energy present in the space-time. In case the system contains matter that is best described as a fluid, this information is determined by solving the evolution equations of the fluid. Hence, GRHD simulations numerically solve a set of relativistic hydrodynamics equations which we now briefly discuss. The original formulation of these equations involves the so-called *primitive variables*, defined in the local rest frame of the fluid. These are the *mass density* ρ, *velocity* field v^i of the fluid and *specific internal energy* ε. These primitive variables are grouped together in a state vector $\boldsymbol{\mathcal{P}}$:

$$\boldsymbol{\mathcal{P}} \equiv (\rho, v^i, \varepsilon)\,. \tag{3.2}$$

The pressure p of the fluid is a derived quantity and depends on the equation of state of the fluid, as we will discuss in further detail below.

The hydrodynamics equations that evolve these primitive variables can be written in the form[2]

$$\partial_t \boldsymbol{\mathcal{P}} + \boldsymbol{A} \cdot \boldsymbol{\nabla} \boldsymbol{\mathcal{P}} + \boldsymbol{B} = 0\,, \tag{3.3}$$

[2]Readers are referred to Chapter 7 of [20] for details and derivations, which are beyond the scope of this thesis.

representing a system of hyperbolic equations. Hyperbolicity ensures that the equations are suited for numerical solvers.

However, using evolution equations in terms of the primitive variables has disadvantages for simulations containing shocks and discontinuities, as we will explain shortly. These issues are resolved by relying on a reformulation of the hydrodynamics equations, called the conservative formulation. In a *conservative formulation*, a different state vector $\boldsymbol{\mathcal{C}}$ is used for which the evolution equations take the form

$$\frac{\partial \boldsymbol{\mathcal{C}}}{\partial t} + \boldsymbol{\nabla} \boldsymbol{F}(\boldsymbol{\mathcal{C}}) = \boldsymbol{S}\,, \tag{3.4}$$

where $\boldsymbol{F}$ and $\boldsymbol{S}$ are the flux and source vector, respectively. The variables in the state vector $\boldsymbol{\mathcal{C}}$ are called the *conserved variables* (or conservative variables), which are the fluid variables measured in the laboratory frame rather than the rest frame. These are the *conserved density* D, the *conserved momentum density* S_i and the *conserved energy* E. Since linear combinations of conserved variables are still conserved, numerical schemes often use the quantity

$$\tau \equiv E - D \tag{3.5}$$

instead of E for numerical reasons. With these considerations, the state vector consisting of the conserved variables is then

$$\boldsymbol{\mathcal{C}} \equiv (D, S_i, \tau)\,. \tag{3.6}$$

The importance of conservative schemes is highlighted by several mathematical theorems. One such theorem states that numerical schemes in which the evolution equations are not written in conservative form are unable to converge to the true solution in case a shock is present, which is very likely to occur in hydrodynamical systems.

3.2 Equation of state

The system of relativistic hydrodynamics equations does not constitute a closed system. To close the system, additional thermodynamic relations, encoded in the *equation of state* (EOS) are required. We restrict the following discussion to two classes of equations of state which appear throughout this work.

3.2.1 Ideal-fluid equation of state

One of the simplest equations of state is that for a classical monatomic fluid, which takes the well-known form $p = nk_BT$. It can be shown that this equation of state is also valid for non-degenerate relativistic fluids, non-degenerate non-relativistic fluids and non-degenerate ultrarelativistic fluids, to which we will refer simply as *ideal fluids*. An equivalent expression of this EOS is the relation

$$p(\rho, \varepsilon) = (\Gamma - 1)\rho\varepsilon\,, \tag{3.7}$$

where p is the pressure of the fluid and Γ is the *adiabatic index* of the fluid (*i.e.*, the ratio of the specific heat at constant volume over the specific heat at constant pressure). This EOS is often referred to as the *ideal-fluid EOS* or the Γ*-law EOS*. There are two noteworthy aspects of this equation of state. First, the EOS is analytic, meaning that the relation between the pressure, density and energy is represented by a closed-form expression which is easy to evaluate in numerical solvers. Second, the EOS does not take the microphysics, such as the composition of the fluid, into account, and hence is an unrealistic equation of state for systems such as neutron stars and core-collapse supernovae, for which microphysical effects become important. GRHD simulations also have to keep track of the *speed of sound*. Its square, denoted by c_s^2, can easily be computed for an analytic EOS from the *enthalpy*

$$h = 1 + \varepsilon + p/\rho\,, \tag{3.8}$$

and derivatives of the pressure using the equation [23]

$$c_s^2 = \frac{1}{h}\left(\frac{\partial p}{\partial \rho} + \frac{p}{\rho^2}\frac{\partial p}{\partial \varepsilon}\right)\,. \tag{3.9}$$

3.2.2 Tabulated equations of state

When the microscopic time-scales are comparable to the macroscopic ones, the assumption of local thermodynamic equilibrium breaks down, and the ideal-fluid description has to be modified in order to take microphysical effects into account. This is the case, for instance, in neutron star matter. As a consequence, the EOS depends on the *electron fraction* of the fluid, Y_e, and generically takes the form $p = p(\rho, \varepsilon, Y_e)$. Besides the pressure, these EOS contain additional variables, to which we will refer as the dependent variables of the EOS. These include the compositions of particles, chemical potentials or thermodynamic quantities of interest. Therefore, the EOS must be viewed as a function $f(\rho, \varepsilon, Y_e)$ that specifies several of such dependent variables, the pressure only being one of them.

These microphysical EOS introduce several complications for numerical solvers compared to an analytic EOS, such as the ideal-fluid EOS. First, the thermodynamic potential from which the physical quantities are derived is the Helmholtz free energy [24]. As a consequence, the EOS is a function of temperature T rather than the energy density ε, *i.e.* the EOS has the functional form $f(\rho, T, Y_e)$. While a relation between ε and T exists, this relation cannot be expressed analytically for these EOS. Instead, rootfinding methods have to be used to convert ε, which is evolved during simulations, to the corresponding temperature, which serves as the input of the EOS. Second, the microphysical EOS cannot be written in a closed-form expression and are too costly to compute during simulations. Therefore, GRHD codes rely on a *look-up table*, which is a table of gridpoints at which the EOS variables are evaluated. For instance, the pressure is computed as

$$p = p(\rho, T, Y_e) \approx f_{\texttt{[rho, T, Ye]}}(p)\,, \tag{3.10}$$

where the notation on the right hand side indicates that f is represented as an array. Hence, one often refers to these EOS as *tabulated EOS*. When the EOS is queried at a point within the domain of the EOS table, a look-up procedure determines the value at this point based on the values of the surrounding gridpoints. Most solvers rely on trilinear interpolation, which is discussed in more detail in Section B.2. A third complication arises from the fact that the values in the EOS tables are not smooth and can be noisy or unphysical. Besides introducing errors, this also implies that the speed of sound cannot be computed exactly as in the ideal-fluid case, since derivatives of the pressure are inaccurate. Hence, these values also have to be supplied in the EOS table. A final complication is that EOS tables are memory intensive, with typical EOS tables easily taking around 800 MB, although the exact size depends on the resolution of the grid. At higher resolutions, these EOS table can even take up more than 4 GB of memory which leads to a dramatic impact on the performance of simulations. It is up to the practitioner running a simulation to choose a resolution, which leads to a trade-off between accuracy and speed or memory requirements.

Microphysical EOS are derived from finite-temperature theoretical frameworks of nuclear interactions and are constructed using the methodologies described *e.g.* in Ref. [25]. Several EOS tables are provided at Ref. [26]. Each table has 19 dependent variables (also called columns) which are tabulated as a function of $\log \rho, \log T$ and Y_e. Besides thermodynamic relations, the table provides chemical potentials and compositions of various particles and chemical potentials. As an example, we show a subset of the pressure, specific internal energy and square of the speed of sound divided by the speed of light for the SLy4 tabulated EOS in Figure 3.2. Note that the speed of sound values at high density exceed the speed of light, which is unphysical.

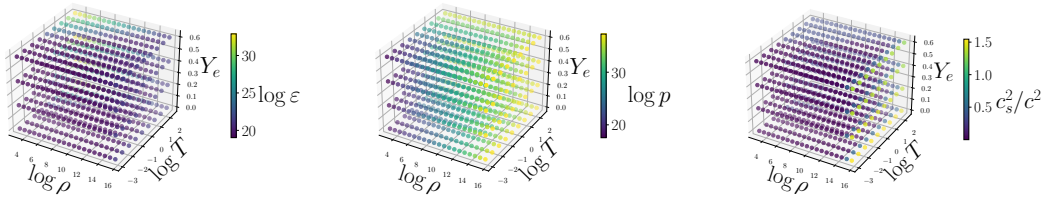

Figure 3.2: Visualization of the SLy4 tabulated EOS, showing the pressure, specific internal energy and the square of the speed of sound.

3.3 Conservative-to-primitive transformation

As mentioned, numerical GRMHD codes evolve the conserved variables, as such schemes are guaranteed to converge to the correct result in case the flow develops shocks or discontinuities. However, the fluxes of the evolution equations depend on the primitive variables, and hence one has to compute the primitive variables corresponding to the conserved variables at each time step. This so-called recovery procedure is hence a central algorithmic step in any GRMHD simulation codes.

Let us first mention that the transformation in the reverse direction, from primitive to conservative variables, can be expressed analytically. This *primitive-to-conservative transformation* (P2C) is given by

$$D = \rho W(v) \tag{3.11a}$$
$$S_i = \rho h W(v)^2 v_i \tag{3.11b}$$
$$\tau = \rho h W(v)^2 - p - D\,, \tag{3.11c}$$

where we have defined the *Lorentz factor* $W(v)$ as

$$W(v) = \frac{1}{\sqrt{1 - v_i v^i}}\,. \tag{3.12}$$

Recall from Chapter 2 that repeated indices implicitly represent a summation, *viz.*

$$v_i v^i = \gamma^{ij} v_i v_j = \sum_{i,j=1}^{3} \gamma^{ij} v_i v_j\,. \tag{3.13}$$

The above P2C transformation is valid for GRHD. In GRMHD, contributions from the magnetic field have to be taken into account. For simplicity of presentation, we show the complete GRMHD equations in Appendix A. Note that the transformation, due to the Lorentz factor, neatly demonstrates that the difference between the conserved and primitive variables can be ascribed to a change of reference frame.

Unfortunately, the inverse transformation cannot be written in a closed form expression in relativistic hydrodynamics. While the recovery of the primitive variables from the conservative ones is trivial in Newtonian hydrodynamics, in the relativistic setting, one has to numerically solve a system of coupled non-linear equations [27]. The *conservative-to-primitive* (C2P) transformation must hence be accomplished using numerical techniques such as rootfinding algorithms. As such, the C2P step can be computationally expensive and a source of numerical errors, especially for simulations employing a tabulated EOS. Rootfinding algorithms commonly used by the `Gmunu` solver, to be introduced shortly, are the Newton-Raphson (NR) algorithms and Brent's method, which are discussed in more detail in Appendix B. Instead of computing all the primitive variables, it is sufficient to obtain a value for the pressure p, since the other primitive variables can be computed from the pressure and the conservative variables. This is achieved by the equations [23]

$$v^i(p) = \frac{S^i}{\tau + D + p} \tag{3.14a}$$
$$\varepsilon(p) = \frac{\tau + D(1 - W) + p(1 - W^2)}{DW} \tag{3.14b}$$
$$\rho(p) = \frac{D}{W}\,. \tag{3.14c}$$

The importance of an efficient and reliable C2P scheme for GRMHD simulations cannot be underestimated. The cost of recovering the primitive variables can be

up to 40% of the total simulation cost [28]. Moreover, this calculation is performed at each grid point for each time subcycle, easily leading to 10^9 calls to the C2P per millisecond [23]. Furthermore, recent work comparing different GRMHD codes found quantitative differences in simulations involving mergers of binary neutron stars. Some reported differences are a 10% difference between merger times, different survival times of the post-merger remnant and different preservation of the magnetic fields during inspiral. The authors, although without providing solid proof, ascribe these differences to the different implementations of the C2P algorithms between solvers. As such, it is clear that the computational efficiency and accuracy of GRMHD simulations crucially depend on the C2P step of numerical solvers.

Different recovery schemes can be assessed and compared to each other based on several criteria [28]. First, the speed of the C2P scheme is important. The speed can be measured in terms of CPU time, although often, one also considers other measures such as the number of iterations in rootfinding algorithms or calls to the EOS. Reducing the number of calls to the EOS can be beneficial since, as mentioned before, microphysical EOS rely on look-up procedures and interpolations, which can be costly to compute and easily make EOS calls the most expensive part of the C2P procedure. Second, the C2P naturally has to be performed accurately. Accuracy can be measured in an artificial test case as follows. Starting from given values for the primitive variables $\boldsymbol{\mathcal{P}}$, we compute the corresponding conserved variables using the analytic P2C, and make use of the C2P procedure to numerically compute an approximation of the primitive variables $\boldsymbol{\mathcal{P}}'$, which we compare against $\boldsymbol{\mathcal{P}}$ to assess the accuracy. The accuracy can also be tested in numerical solutions which have a known, exact solution. Finally, C2P schemes have to be robust. Robustness is often assessed more qualitatively and relates to aspects such as independence of the initial guess of rootfinding methods, guarantee of convergence and independence of derivatives of thermodynamic quantities. Moreover, not all combinations of evolved variables correspond to primitive variables which are physically valid, and robust schemes have to detect and correct such errors during the evolution [27].

Various recovery schemes have been introduced in the literature. The most commonly used schemes are those proposed by Nobel [29], Duran [30], Neilsen [31, 32] and Newman [33]. These schemes were compared in Ref. [28], where additionally another novel C2P scheme was discussed. The main conclusion of this work states that all state-of-the art recovery schemes can fail in certain regimes, and the efficiency of a C2P algorithm depends on the EOS that is used. Overall, a three-dimensional NR scheme is the fastest and most accurate scheme, although not the most robust one, since the convergence of an NR scheme is sensitive to the provided initial guess. Moreover, the dependence on thermodynamic derivatives could lead to failure in an NR scheme. Schemes such as a one-dimensional version of Brent's method, on the other hand, are guaranteed to converge. Moreover, in case the derivatives have to be estimated numerically for a one-dimensional function, Brent's method is guaranteed to dominate over an NR scheme [34].

However, the schemes investigated in Ref. [28] are unable to reliably handle and correct invalid evolved variables.

More recently, a robust recovery scheme for ideal GRMHD[3] simulations was suggested in Ref. [27], which we will refer to as *Kastaun's scheme*. The scheme applies a rootfinding method to a one-dimensional master function $f(\mu)$, where μ is defined by

$$\mu \equiv \frac{1}{Wh}\,, \tag{3.15}$$

where h is the enthalpy and W is the Lorentz factor. The scheme comes with a guarantee, which is proven mathematically, to find a unique solution and to detect invalid evolved variables. Moreover, the scheme is EOS-agnostic and in particular does not depend on derivatives of the EOS, such that it is able to efficiently work with any EOS that is supplied during simulations. The tests carried out in Ref. [27] demonstrate robustness of the scheme across a range of the magnetic field that is typically encountered in GRMHD simulations. Compared to the schemes reviewed in Ref. [28], Kastaun's scheme is much simpler, since it is formulated as a one-dimensional problem whereas most of the older schemes tend to be formulated in two or three dimensions. Overall, Kastaun's scheme has an improved efficiency and accuracy over the aforementioned C2P schemes such that `Gmunu`, the simulation code adopted in this work, by default relies on Kastaun's scheme for the C2P conversion.

3.4 GRMHD simulations in `Gmunu`

The GRMHD simulation code that we use in this thesis is the `g`eneral-relativistic `mu`ltigrid `nu`merical solver, or `Gmunu` for short [36–39]. As mentioned before, numerical relativity codes rely on a $3+1$ decomposition of space-time. As a consequence of this decomposition, the Einstein equations are split into two distinct sets, a set of evolution equations and a set of constraints that have to be satisfied during the evolution. Most NR schemes adopt a free-evolution approach, where the constraints are only solved to provide initial data. They are then used to monitor the accuracy of simulations but are not explicitly solved. Alternatively, one can adopt a fully constrained-evolution approach, where the constraint equations are also solved during the simulations. While this approach is less popular since the constraint equations are expensive to solve, it is the approach considered in the `Gmunu`. To deal with this issue, `Gmunu` combines a standard finite-volume grid for the hyperbolic hydrodynamic equations with a multigrid method for the elliptic, space-time metric equations. Since such multigrid methods employ a hierarchy of different discretizations, their computational cost is generally lower than most standard methods. `Gmunu` is the first solver to introduce the multigrid approach in the field of constrained-evolution GRMHD codes. As already mentioned, `Gmunu` by default relies on Kastaun's C2P scheme for both GRHD and GRMHD. Recent developments on the `Gmunu` have included non-vanishing electric resistivity in the GRMHD formulation, which is important to realistically model plasmas, and have implemented radiation transport. `Gmunu` is written in the Fortran programming language. *Fortran*, derived from 'formula translation', is

[3] In the ideal MHD limit, fluids are assumed to have vanishing resistivity such that electric fields vanish in the co-moving frame of the fluid [35].

a compiled programming language commonly used in scientific high-performance computing codes [40]. Further details on the implementation of the solver can be found in the aforementioned references.

A few studies have been published based on simulations performed in Gmunu. Ref. [41] proposed a link between the GW signal of a binary neutron star merger and the fundamental oscillation modes and the mass ratio of the binary system. Ref. [42] studied the formation process and properties of magnetized hybrid stars, which are stars consisting of exotic matter such as deconfined quarks besides hadrons. Finally, Ref. [43] investigated the effects of magnetization on the pulsations of highly magnetized, non-rotating neutron stars and the GWs they emit.

Our goal in this thesis is to find a way to optimize the C2P conversion in Gmunu. The approach that we adopt is to cast the problem into a form that can be solved with the help of machine learning algorithms, which have already demonstrated remarkable abilities to learn from datasets and speed up computations in various fields in the past. In the next chapter, we will therefore introduce an overview of machine learning and the techniques that we use.

Chapter 4

Machine learning

Machine learning is a large subfield of artificial intelligence, an interdisciplinary field that studies artificial learning behaviour. Machine learning algorithms are designed with the capacity to improve their performance by learning from examples, either provided as input or obtained through interaction with an environment. Different from standard computer algorithms, these algorithms do not explicitly formulate step-by-step procedures determining how a task has to be solved. Rather, by specifying a desired outcome and comparing the algorithm's behaviour against it, the parameters of the algorithm can be adjusted in such a way that the final algorithm is able to achieve the desired task. Machine learning offers tools to extract complicated patterns from data and learn a model from them. Distilling knowledge from data is a common goal of both science and machine learning, and combining these two fields is determined to lead to new, exciting opportunities in the future.

In this thesis, we investigate the possibility of optimizing the complicated C2P conversion in GRMHD simulations through machine learning algorithms. This chapter introduces machine learning from a high-level perspective and discusses deep learning, the technique used in the remainder of the thesis. Moreover, we provide an extensive overview of past work on the applications of deep learning in Fortran codes. We refer readers to Refs. [44–46] for more details.

4.1 Introduction to machine learning

Artificial intelligence (AI) can be broadly defined as the study of agents that are able to receive certain perceptions from an environment and can take actions within that environment. Viewed as a program, such agents are mappings from percept sequences to actions. The key distinction in agents studied in AI is rationality. A rational agent is one that acts in such a way to maximize the expected utility of the outcomes due to its chosen actions. This broad and vague definition reflects the variety observed in AI algorithms and applications. The field of AI touches upon other fields such as psychology and neurology, employs algorithms based on probabilistic reasoning, logic, or parametric models which learn from tabulated data, text, images and videos or through experiences and rewards. Besides, agents can

take the shape of computers, robots, automated vehicles, smart devices and much more.

Here, we restrict ourselves to machine learning, one of the major subfields of AI. In the field of *machine learning* (ML), the agents are programs that build a model (a hypothesis of the environment) from observations of data. A formal definition which is often mentioned is one coined by Mitchell [47]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks T, as measured by P, improves with experience E". We will only deal with *supervised learning* problems, where these experiences consist of instances of input-output pairs (also called attribute-value pairs) and the algorithm has to learn the correct mapping from input to output. Often, the input values are called the *features* and the output values the *labels*. The algorithm is then provided a dataset of samples of such pairs of input-output examples, from which it has to learn the mapping, generalized to the population from which the dataset is sampled. If the domain of the labels constitute a finite set, then the problem is said to be a classification problem. When the output values are continuous variables, the learning problem is said to be a regression problem. In this thesis, we will only encounter regression problems.

4.1.1 Regression problems

A regression problem is more formally described as follows. Given a dataset

$$\{(\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathbb{R}^n \times \mathbb{R}^m \mid \boldsymbol{y_i} = f(\boldsymbol{x}_i)\} \,, \tag{4.1}$$

for $i = 1, \ldots, N$ and with f an unknown function, the goal is to find the hypothesis $h : \mathbb{R}^n \to \mathbb{R}^m$ (also called the *model*) in a hypothesis space $\mathcal{H}$ that provides the best approximation of the function f. To evaluate the approximation, one defines a *loss function* $\mathcal{L}$ that quantifies the error between the predictions of the model, denoted by $\hat{\boldsymbol{y}}$, and the actual values. A commonly used loss function is the *mean-squared error* (MSE) loss:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \,. \tag{4.2}$$

Therefore, the goal of a regression problem can be rephrased into finding the hypothesis that minimizes the loss. However, minimizing the loss function on the provided samples does not guarantee that the learned model will be a reliable approximation of the true underlying function across its entire domain. Therefore, when measuring the performance of a hypothesis h, we are more interested in the *generalization* of h. We say that h generalizes well if it has a high performance (low loss) on a dataset of previously unseen examples. Therefore, ML practitioners often keep a part of the available data (called the *test set*) separate to check the model's performance on unseen data.

Several techniques exist to ensure that ML models generalize well. One common technique is to split the available data into a *training set* and a *validation set*. The former is used to tweak the model's parameters, whereas the latter acts as a proxy to

the test set and monitors the performance on unseen data during the learning phase. By comparing the loss on the validation set, different ML models can be compared to each other, allowing us to select the best model among a set of proposals. We can now more formally state that with *training* an ML model, we mean iteratively presenting the training data to the model and updating its parameters[1] in each iteration in such a way to minimize the training loss up until a criterion is met, such as reaching a convergence condition or a maximal number of training iterations. In ML literature, a single training iteration is also commonly referred to as an *epoch*.

4.1.2 Generalization and overfitting

The hypothesis space $\mathcal{H}$ should be selected with care. Importantly, there is a trade-off between the *capacity* or expressiveness of models in a hypothesis space and the computational complexity required to determine a good hypothesis within the hypothesis space. Often, this choice is intertwined with the *bias-variance trade-off*, which is a trade-off commonly encountered in ML. That is, one must decide between a complex and expressive model that has low bias and can readily fit the data, but may exhibit high variance in the learned model with respect to the provided data, and a simpler model with high bias that may have difficulty fitting the data, yet can decrease variance and potentially improve generalization. This balance has two extremes which are catastrophic for any ML algorithm. When we are unable to find a hypothesis in the hypothesis space that reasonably fits the training data, we say that the model is *underfitting*. One example is a linear regression model trained on data sampled from a quadratic function. On the other hand, highly flexible models with too many parameters can easily fit the training data, but likely have poor generalization, which is known as *overfitting*. For instance, n datapoints can always be exactly fitted with a polynomial of degree $n+1$, but when the underlying function is linear, these models will have poor performance on unseen data.

Several techniques exist to prevent a model from overfitting. For instance, one can monitor the loss on the validation set during training. This allows us to gain insight into the generalization of the model during training. If the training loss is decreasing but the validation loss starts to increase, this signals that the model is overfitting on the training data, at which point the training must be stopped. This procedure is known as *early stopping*. Another commonly used technique is to modify the loss function by introducing a term that depends on the complexity of the model and penalizes more flexible models. Such a *regularization* term ensures that the optimal model achieves a trade-off between simplicity ("regularity" of the hypothesis) and accuracy and is inspired by Occam's razor dictating that simpler models likely generalize better.

During training, the precise procedure of updating a model's parameters depends on the model that is being learnt. For convenience, we interpret the loss function as

[1] In light of the techniques used in Chapter 5, our discussion is limited to parametric ML models with continuous parameters.

depending on the parameters $\boldsymbol{\theta}$ of the model in a continuous manner, *viz.*

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \mathcal{L}\left(f(\boldsymbol{x}), h(\boldsymbol{x}; \boldsymbol{\theta})\right) . \tag{4.3}$$

Training the ML model is then an optimization problem formulated in the parameter space of the model, and the goal of the training algorithm is to find the optimal set of parameter values $\boldsymbol{\theta}^\star$ found by

$$\boldsymbol{\theta}^\star = \arg\min_{\theta} \mathcal{L}\left(f(\boldsymbol{x}), h(\boldsymbol{x}; \boldsymbol{\theta})\right) . \tag{4.4}$$

We refer to the *loss landscape* as the loss function viewed as a function of the parameters $\boldsymbol{\theta}$, *i.e.* $\mathcal{L}(\boldsymbol{\theta})$. Only for a few ML models, such as the linear regression model, can we determine $\boldsymbol{\theta}^\star$ analytically by solving $\partial_{\boldsymbol{\theta}}\mathcal{L} = 0$ to determine the minimum of the loss landscape.

More often, one has to rely on numerical optimization algorithms to find the minima of the loss landscape. Many of these techniques are variants of the well-known *gradient descent* algorithm, which updates the parameters in the direction of decreasing loss value determined by the gradient of the loss landscape, *viz.*

$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} - \alpha \partial_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) . \tag{4.5}$$

Here, the proportionality constant α is called the *learning rate*. Artificial neural networks, a popular class of ML models to be introduced shortly, are precisely trained by such optimization algorithms.

4.1.3 Intermezzo: instance based learning

Before delving deeper into the techniques used in this thesis, we provide a few remarks on a different class of ML models outside of the supervised learning setting known as *instance based learning*, its relevance for this work and its relation to supervised learning.

The hypothesis of an instance based learner is in fact the dataset itself, and the "learning" phase consists of simply storing the provided data. Instance based learners are therefore also called *lazy learners*. In the context of predictive learning, the specific task that instance based learners achieve is transductive learning. That this, predictions on a new input are based on, for instance, the k nearest neighbours of the input in the dataset, with k a tunable hyperparameter. In the context of regression, the output can be determined by performing a weighted regression through these nearest neighbours. As such, lazy learners are non-parametric models that construct local models $h_{\boldsymbol{x}}$, that explicitly depend on the training data, at prediction time.

In supervised learning, on the other hand, the algorithms learn a global model h, which hence achieves *inductive learning*. These learners are called *eager learners* since they process the dataset to create the model before new, unseen instances are shown to the learner. There are a few differences between lazy and eager learners. For eager learners, the learning phase is typically slow, while the prediction phase can be faster than instance based methods. Moreover, these learners learn a global model with the

aim to generalize well to the population. Lazy learners have a fast learning phase, as they just store the data, but can be slow at prediction times depending on the data and application. On the other hand, computations can be more sophisticated because they locally use the provided data to make new predictions.

With our discussion of the use of tabulated EOS in GRMHD simulations from Section 3.2.2 and the trilinear interpolation algorithm discussed in Section B.2 in mind, it is clear that the look-up procedures of EOS tables can be seen as a form of instance based learning. In Chapter 5, we compare the efficiency of lazy learners against eager learners at predicting new values of the EOS variables.

4.2 Artificial neural networks and deep learning

Deep learning (DL), a popular and dominant subfield of machine learning, encompasses a wide variety of hypothesis spaces that can be described as algebraic circuits made of computational units with tunable connections between them. These circuits are "deep" in the sense that their computational graph can easily involve many steps in more complicated deep learning architectures. Here, we will mainly refer to these circuits as[2] *neural networks* (NNs). The name originates from the fact that, historically, NNs were introduced as an artificial model of the brain. As such, the nodes of the networks are also commonly referred to as *neurons*.

4.2.1 Multilayer perceptrons

Let us now consider how these networks can be applied to regression problems. We will restrict ourselves to *feedforward* neural networks, which only have connections from the input to the output of the network. The neurons are organized into several *layers*. To represent functions, they hence should certainly have an input layer and an output layer, where the dimensionality of the input and output space determine the size of the input and output layers, respectively. Let us build up intuition with the simplest NN architecture, known as the *perceptron*, consisting of one input neuron and one output neuron. The result of the output layer depends on the value of the input unit and the interconnection strength between the two neurons, called the *weight* w. Qualitatively, the output is determined by the Hebbian rule "neurons that wire together, fire together". That is, a high value for w means that the output will amplify the value coming from the input, while a low value for w implies that the input will not affect the output at all. The output is also affected by a constant offset, the *bias* b. If the input node has value x, then the value of the output node of the perceptron is computed as

$$y = wx + b\,. \tag{4.6}$$

Often, one considers the bias term as originating due to an additional, dummy input node with value 1 and corresponding weight b. That way, we can write the

[2]Whereas the precise term should be *artificial* neural networks, we will refer to these circuits as just "neural networks" for simplicity.

computation for the output as a matrix-vector product:

$$y = \boldsymbol{W}^T \cdot \boldsymbol{x}\,, \quad \boldsymbol{W} = \begin{pmatrix} b \\ w \end{pmatrix}\,, \quad \boldsymbol{x} = \begin{pmatrix} 1 \\ x \end{pmatrix}\,. \tag{4.7}$$

Therefore, when referring to the weights of the network, we will always implicitly include the bias terms as well.

While the perceptron seems simplistic, it is in fact the NN representation of linear regression, a staple among statistical tools. In Figure 4.1, we show the graph representation of the perceptron and the loss landscape as a function of the weights of the perceptron. While analytic relations exist to determine the optimal parameters for linear regression, one can imagine the perceptron to learn these optimal parameters by traversing the loss landscape towards its minimum through the principle of gradient descent.

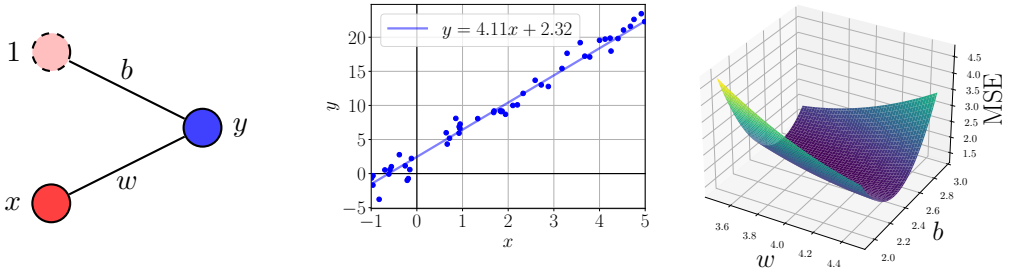

Figure 4.1: Illustration of the perceptron representation of linear regression and its loss landscape.

The true power of deep learning comes, as the name suggests, by extending the simple perceptron into deeper (more layers) and wider (more neurons) architectures that can approximate non-linear functions as well. To get more complicated architectures, we introduce additional layers between the input and output layer. These are called the *hidden layers*, since their computations are not directly observed, unlike the input and output layers. As before, each layer is the weighted sum of the neurons of the previous layer, *viz.*

$$\boldsymbol{z}_{(l)} = \boldsymbol{W}_{(l)}^T \cdot \boldsymbol{z}_{(l-1)}\,, \tag{4.8}$$

where l is a layer index.[3] We can think of each hidden layer as corresponding to a different representation built from the original input. Often, the sizes of the hidden layers are much larger than the dimensionality of the input. In the graph representation corresponding to Eq. (4.8), all neurons of layer $l-1$ are connected to all neurons of layer l. Such networks are said to be *fully connected* or *dense*.

Connecting several hidden layers using the transformation from Eq. (4.8) results in a class of linear networks. We can introduce non-linearity by including *activation*

[3]For simplicity, we will not mention the bias term explicitly anymore, although we always use bias terms for every layer unless stated otherwise.

functions. These are non-linear functions φ applied to the weighted sum of Eq. (4.8), such that the values computed in layer l become

$$\boldsymbol{z}_{(l)} = \varphi\left(\boldsymbol{W}_{(l)}^{T} \cdot \boldsymbol{z}_{(l-1)}\right) , \tag{4.9}$$

The activation functions that we consider in this thesis are the *sigmoid* activation function:

$$\varphi(z) = \frac{1}{1+e^{-z}} , \tag{4.10}$$

and the *rectified linear unit* (ReLU):

$$\varphi(z) = \max(0, z) , \tag{4.11}$$

which are easily extended to vectors by applying these operations element-wise.

Combining all of the above ingredients leads to the concept of *multilayer perceptrons* (MLPs), which are fully connected feedforward neural networks with non-linear activation functions applied to the hidden layers.[4] These are the network architectures that we employ in this thesis. Building on top of the intuition that we gathered from the perceptron, we can interpret MLPs as performing linear regression on the final hidden layer to compute the output. This final hidden layer is then a set of high-dimensional abstract features obtained by performing a non-linear transformation on the input data. This transformation is obtained by chaining the transformations of the previous hidden layers, where the NN learns, by adjusting the weights of the network, to build the representation that gives the best results in the linear output layer.

Mathematical theorems have demonstrated that MLPs are universal approximators. That is, they can be used to represent any continuous non-linear function to arbitrary accuracy. However, these theorems do not include constructive proofs, and finding a good architecture is often based on trial and error. While the loss landscape of the linear regression problem was a two-dimensional convex surface with a unique, global minimum, the loss landscape of most MLPs are high-dimensional surfaces with many local minima. Random initialization of the network weights implies that different training runs can converge to different minima. Therefore, when training MLPs, we are often optimizing the network weights until a local minimum is reached.

4.2.2 Optimization algorithms

The core idea of training MLPs builds on gradient descent. When presented a training example, the network performs a forward pass, where information propagates from the inputs to produce an output $\hat{\boldsymbol{y}}$. Comparing the prediction with the true label $\boldsymbol{y}$, we then compute a loss value. The *backpropagation* algorithm allows one to easily compute the backward pass, where information of the loss flows backwards through the network to compute gradients on the weights. Backpropagation relies on automatic differentiation, which applies the rules of calculus to the computational

[4] With a slight abuse of terminology, we will use the terms neural networks and multilayer perceptrons interchangeably.

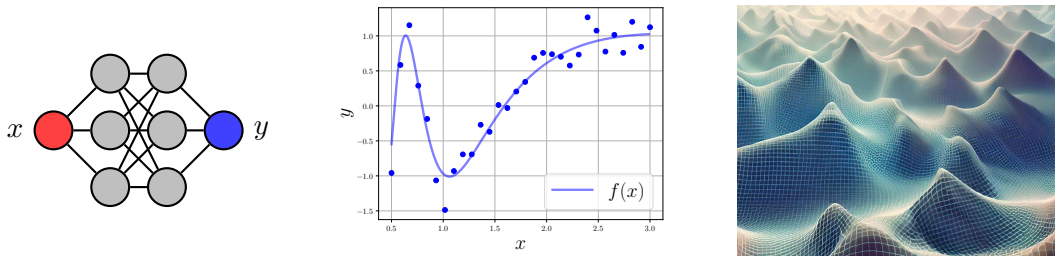

Figure 4.2: Illustration of the MLP as universal approximator. The loss landscape is high-dimensional and contains many local minima. Image generated by Ref. [48].

graph that MLPs represent. Indeed, computing the desired gradients is done by applying the chain rule in reverse mode differentiation. Details and derivations of the backpropagation algorithm can be found in Ref. [46].

Once the gradients are evaluated, it can be passed to the optimizer to update the parameters of the NN. Several optimization algorithms exist that improve upon the basic idea of gradient descent. One improvement, shared by many optimizers, is to use small, random subsets of the training data to estimate the gradient in a statistically robust manner. Such algorithms are called *minibatch* algorithms. The size of these minibatches is typically taken to be 16 or 32. When gradient descent uses an estimate for the gradient obtained in this way, it is referred to as stochastic gradient descent. This optimizer can be accelerated by adding a momentum term, which accumulates a decaying moving average of previously computed gradients to aid the search. Another popular extension is to use an adaptable learning rate. One popular algorithm is *Adam*, derived from "adaptive moments". Here, one computes an unbiased estimate for the first and second moments of the gradient. Adam is generally regarded as fairly robust to the choice of hyperparameters.

4.2.3 Neural architecture search and pruning

While the universal approximator theorem ensures that MLPs can approximate any function, it is not clear how to determine an architecture that is able to fit the data and generalize well. Generally, this is determined through trial and error. However, since we are interested in speeding up simulation codes, we would like to find an optimal architecture that is able to fit the data with as minimal neurons as possible in order to ensure that the networks can be evaluated efficiently. In fact, this determines a meta-optimization problem, formulated over the space of all MLPs. The field of neural architecture search deals with such architecure engineering issues.

There are several neural architecture search techniques discussed in the literature, which can be characterized based on three criteria: the search space (*i.e.*, the hypothesis space of possible network architectures), the search strategy and the performance estimation strategy [49]. Even when restricting to standard MLPs, where only the number of hidden layers, their sizes and the activation functions determine the search space, one encounters an enormous search space. Clever search strategies have to be designed that provide a trade-off between exploration and

exploitation during the search. Examples include using random search, Bayesian optimization, evolutionary algorithms, reinforcement learning or gradient-based methods in a continuous search space. To estimate the performance, one obvious strategy is to train each network configuration to convergence (using early stopping on a validation set, for instance). However, this is clearly computationally expensive even for small architectures. While several proxy metrics have been proposed in the literature to shorten the training, they can lead to biased estimates and are beyond the scope of this work to implement and experiment with.

A simpler procedure, which we adopted in this work, is to find (through trial and error) an architecture of reasonable size that is able to fit the data, and reduce its size through *pruning*. In the context of NNs, pruning refers to removing parameters from an existing, trained network in order to reduce the complexity of the network while maintaining similar performance. Implementations of pruning schemes differ on various levels [50]. For instance, unstructured methods prune an individual parameter, while structured methods remove groups of parameters, for instance by removing neurons which are connected by several weights. Scoring a pruned network can be done locally or globally, depending on whether we take into account the location inside the architecture where pruning occurred. Schemes can differ in their scheduling, which determines if weights are pruned iteratively or entirely at once. Finally, networks are often fine-tuned (*i.e.*, trained after pruning), although details differ depending on the implementation. The methods of optimal brain damage (OBD) and optimal brain surgeon (OBS), which prune individual weights, are among the most well-known pruning methods [51, 52]. OBD prunes those weights whose deletion have the smallest impact on the training error. OBS improves upon the methods of OBD by including information from the second-order derivatives of the loss function.

In this work, we have briefly explored a simple, greedy pruning scheme that prunes individual neurons based on their impact on the training loss. The pruning is performed iteratively and each iteration prunes a single neuron from the original network, such that we employ a hill climbing-like search strategy. The pseudocode of the pruning scheme is shown in Algorithm 1.

Algorithm 1 Greedy hill climbing neuron pruning

Require: *network*, H hidden neurons
Require: *threshold*, to determine whether to fine-tune a pruned network
Ensure: *best_network*, $K < H$ hidden neurons

while $K < H$ **do**
  best_network ← None
  best_loss ← ∞
  for i = 1,...,H **do** ▷ Find optimal neuron to prune
    pruned ← *delete_hidden_neuron*(*network*, i)
    loss ← *measure_loss*(*pruned*)
    if *loss* < *best_loss* **then**
      best_network ← *pruned*
      best_loss ← *loss*
    end if
  end for
  network ← *best_network*
  $H \leftarrow H - 1$
  if *best_loss* > *threshold* **then**
    train(*network*) ▷ Fine-tune the pruned network
  end if
end while

4.3 Deep learning in Fortran

We chose to work with NNs in this thesis for two reasons. First, MLPs are easy to implement and train in Python thanks to highly-optimized open-source libraries. In this work, we designed and trained neural networks with PyTorch [53]. Second, the computations of MLPs are created from simple building blocks such as matrix-vector multiplications and a few non-linear activation functions. As these building blocks are either supported by or straightforwardly implemented in any programming language, we can easily export NNs to `Gmunu`, which is written in Fortran.

At first sight, deploying deep learning models trained in Python in simulation codes written in Fortran seems merely an issue of programming language interoperability. However, large-scale simulations rely on parallelization such that additional considerations such as efficient threading and data exchange complicate the matter. Furthermore, simulations benefit from an efficient implementation which goes beyond integrating a functional version. The problem is not restricted to the `Gmunu` solver considered in this work, since many numerical simulation codes, including those from entirely different fields such as computational fluid dynamics and climate modelling, are written in Fortran. Therefore, communities dealing with such high-performance computing (HPC) codes currently lack a universal, optimized technique to integrate NNs into Fortran codes. Developers of HPC codes as well as Fortran developers both have pointed out this apparent gap and expressed the need for bridging this gap [54, 55].

In this section, we provide an overview of existing attempts to bridge the gap, discuss their benefits or disadvantages and point readers towards the most promising methods for future development. We start off by providing a discussion on the most important requirements for the integration of DL frameworks with Fortran HPC codes. Afterwards, we compare existing software packages that have been developed to achieve this integration.

4.3.1 Wishlist

Before discussing the existing approaches to integrate DL (or more generally, ML) frameworks in Fortran HPC codes, let us give an extensive overview of desirable properties that such an integration should have.

First, inference of the deployed ML model should be as efficient as possible such that ML methods offer a computational advantage. Here, efficiency can be measured either in terms of latency or inference throughput. While there has been some work regarding such a comparison in Ref. [56], a complete comparison has not been completed. Since efficiency depends on various factors (such as the source code of the simulation, the application under consideration, the trained model), it is hard to estimate the performance of strategies beforehand. Therefore, we will continue our comparison based on implementations of the code.

Second, the integration should require minimal adjustments to the simulation source code for several reasons. First, HPC simulations are typically not written with the interoperability between Python and Fortran in mind. Adjusting source codes to include this interoperability is time-consuming and tedious. Second, due to their popularity, existing DL frameworks are developed at a rapid pace which HPC communities are unable to follow. Third, developers of HPC simulation codes have limited resources which ideally should be dedicated towards their source code. Therefore, the integration of DL should ideally be achieved "under the hood".

Third, we want the integration between DL and Fortran solvers to be flexible. That is, we want to be able to easily switch between different models within a simulation. Moreover, since several DL frameworks exist in Python which are more or less equally popular (such as scikit-learn, TensorFlow or PyTorch), the integration should be agnostic of the specific library used to train networks. Hence, the exported models should be based on a universal format. In short, loading models in Fortran should require a minimal amount of preprocessing steps.

Fourth, the integration should rely on existing open-source libraries such as TensorFlow or PyTorch. These libraries are developed by large communities of developers and are evolving at a rapid pace. Rewriting these libraries in Fortran is unlikely going to provide the same optimized functionalities and similar support as existing state-of-the-art DL frameworks. However, the Fortran community seems divided on this idea [57].

Fifth, the integration should be achieved by an implementation used and shared by a larger community which is actively working on the implementation to provide support to new users and adapt the codebase to changes and additions to DL frameworks.

Sixth, the integration should offer hardware optimization and support the use of accelerators such as GPUs and parallelization of code, as these are standard practices for HPC simulations which often involve large-scale simulations.

Seventh, an ideal integration should enable the *online training* of ML models. That is, we envision future applications which train ML models based on data acquired during simulations. For such applications, batch learning is less suited, since large scale simulations involve many gridpoints over large timescales. Saving data to external files would require extensive use of an I/O system, slowing down the simulation and creating large datasets to process. Training a model from this data is infeasible due to the size of the dataset and the implied training time. Hence, we wish to rely on libraries that efficiently allow online learning. An additional advantage of online learning is that one can take the performance of the model into account before deciding to provide a datapoint to the model for training.

4.3.2 Overview of existing methods

We provide an overview of existing approaches that integrate deep learning with Fortran. A summary and comparison of the main approaches discussed in this section is provided by Table 4.1. The main criteria we consider are portability (easy to export and relying on universal formats), flexibility (easy to change models or to use several models at once), active development of the software, support for hardware optimization and the ability to train models in an online setting.

Table 4.1: Comparison between frameworks integrating deep learning with Fortran.

	API	`roseNNa`	`pytorch-fortran`	`neural-fortran/FKB`	`SmartSim`
Portability	✗	✓	✗	✗	✓
Flexibility	✗	✗	✗	✗	✓
Active development	✓	✗	✗	✓	✓
GPU/parallelization	✓	✓	✗	✓	✓
Online learning	✗	✗	✗	✗	✓

Hard-coded single-use neural networks

A first option, not included in the table above, is to recreate the computations of a trained NN directly in a Fortran program. For this, one can use Fortran's built-in matrix multiplication and implement the activation functions from scratch. The weights and biases of the network can easily be exported to CSV or HDF5 files after training, which can be processed in Fortran. This approach has been used before in the field of climate modelling [58].

Such an implementation does not offer the more advanced features highlighted in our wishlist. Nevertheless, it is an ideal approach for works that deal with proof of

concepts. As such, we have chosen to adopt this approach to integrate our NNs into the `Gmunu` simulation code. An example of our deep learning code in Fortran, which implements the NNC2P architecture that we are going to present in Section 5.2, is provided in Appendix C.

Language binding

One possibility is to interface other programming languages from Fortran, known as language binding. For instance, one can interface Python directly from Fortran code. In Ref. [59], the authors couple a trained NN with their Fortran simulation code of gravity waves (not to be confused with gravitational waves) using the `forpy` package [60]. They report that their Fortran implementation of the neural network is 2.5 times slower, such that this approach has to sacrifice performance.

However, another option, which is also supported in PyTorch, is to use an export API to convert the NN models to TorchScript [61, 62]. TorchScript compiles NN models to serializable versions which have the advantage that they can be loaded into processes without any Python dependency. Converting models to Torchscript is done through tracing and scripting, which track the computations that the network performs. Afterwards, a computation graph is created, which can be processed in other environments and programming languages, such as C++ [63]. One approach would be to call a C++ program to perform inference of the model from Fortran programs. In this work, we have not compared the efficiency of this implementation compared to the hard-coded implementation discussed above. In Ref. [56], the authors argue that using the TensorFlow C++ API has high portability but less flexibility. In terms of concurrent calls and inference throughput, the authors remark that the C++ API is outperformed by the `SmartSim` package, discussed below.

Fortran libraries

To circumvent the use of other programming languages, a few open-source libraries have been developed that run inference on trained models by rebuilding them directly in Fortran. Ordered in increasing popularity (measured by number of stars on Github), we will discuss the `roseNNa`, `pytorch-fortran`, `FKB` and `neural-fortran` libraries [64–67]. We dedicate the next subsection to `SmartSim`, a package adopting a similar approach but specifically designed and optimized for HPC simulations.

`roseNNa` is a fast and minimally intrusive library to perform neural network inference in Fortran. It is built on top of the open neural network exchange (ONNX) format, an open format that defines a collection of operators commonly used in NNs. This file format achieves interoperability, such that developers can create networks in their preferred framework without sacrificing performance when using other inference engines. Another key benefit is that ONNX offers hardware optimizations to increase performance. The ONNX format has wide support from developer communities and companies. Moreover, the software is actively being developed on Github, as demonstrated by several commits and bug fixes during the time of writing and having more than 200 collaborators, 3 000 forks and 14 000 stars. Furthermore,

ONNX is supported by popular DL frameworks such as PyTorch and TensorFlow, and exporting models to ONNX format is as easy as exporting to TorchScript [68]. `roseNNa` creates Fortran representations of NNs by postprocessing the ONNX files and compiling architectures from custom Fortran modules based on this information. Practitioners can easily integrate the `roseNNa` library in their simulation codes if they so desire. However, to switch to a different model, practitioners are required to apply a sequence of preprocessing steps to convert the model to an ONNX format and correctly parse it with `roseNNa`. Moreover, running several models requires extending the current implementation of `roseNNa`. Currently, `roseNNa` supports recurrent NNs, convolutional NNs and MLPs. Since their final commit to the repository was in April 2023, it is not clear if the developers aim to further extend the package.

`pytorch-fortran` provides a simple way to use PyTorch models in Fortran HPC codes. Besides running inference on trained NNs, `pytorch-fortran` also offers the possibility to train an NN within Fortran. However, this repository does not seem to rely on the ONNX format, which excludes the use of other popular deep learning frameworks such as scikit-learn and TensorFlow which support the ONNX format. Moreover, since the last commit to the repository was in April 2023 and the introduction mentions that the code is "very much work-in-progress", it is not clear if `pytorch-fortran` is a reliable package for integrating deep learning in Fortran.

`FKB`, short for Fortran-Keras bridge and built on top of the `neural-fortran` package, brings the deep learning tools from Keras, a high-level library built on top of TensorFlow, to Fortran HPC codes. The so-called Python anchor stores trained Keras models in HDF5 files containing all information regarding network architecture, weights, biases and information regarding the training such as optimizers and learning rates. The Fortran anchor then parses this information and recreates the network in Fortran. Networks can also be constructed and trained in Fortran and exported to Keras for additional training or hyperparameter tuning. However, it is not clear whether NNs can be trained during simulations as well. The `FKB` code has been successfully applied to case studies [69]. However, a clear disadvantage is the fact that `FKB` relies on custom configuration files that specifically process Keras models.

`neural-fortran` provides out-of-the-box support for MLPs with an arbitrary architecture, optimization, data-based parallelism and saving and loading trained networks. While written in Fortran, the code achieves a similar performance compared to training in Keras, although `neural-fortran` was only implemented as a proof of concept while Keras is highly-optimized [70]. While the original aim of `neural-fortran` seemed to be providing a Fortran implementation of deep learning from scratch, a more recent version of `neural-fortran` has integrated the features provided by `FKB`. Moreover, it has extended the work of `FKB` and also supports training and inference of convolutional networks. We also note that the main contributor of `neural-fortran` contributed to `FKB`. Since `neural-fortran` is more actively maintained at the time of writing, it has likely continued the development of the features offered `FKB`.

SmartSim

Currently, the most promising software package to achieve an ideal merger of DL frameworks and Fortran HPC codes is the `SmartSim` library, which solves the issues that were raised for the other libraries [71].

`SmartSim` consists of two libraries. The first one, called the infrastructure library, offers a Python workflow to facilitate the execution of HPC and ML scripts. This library allows applications written in Fortran, C, C++ and Python to easily communicate without the need of reading from or writing to the filesystem. The models can be efficiently executed on either a CPU or GPU. Furthermore, the library relies on the universal ONNX format. The striking advantage of this set-up is that the communication can be realized without adjusting the source code of either the ML model or the simulation code, making it the most portable and flexible framework covered so far. This is achieved by the second component of `SmartSim`, called `SmartRedis`. `SmartRedis` is built on top of RedisAI, a server optimized for serving ML models [72]. `SmartRedis` adds functionalities to RedisAI which are specifically designed for HPC codes, such as Fortran clients and distributed placements of the models and scripts to allow parallelization to maximize inference throughput.

The experiments performed in Ref. [71] demonstrate that `SmartSim` offers an ideal integration of deep learning with Fortran HPC codes. Moreover, the code is still actively developed. Finally, the fact that the `SmartSim` framework circumvents the use of the filesystem as intermediary makes it the only framework discussed here that enables online training of ML models. To reinforce this latter point, we note that Ref. [73] relied on `SmartSim` to augment a computational fluid dynamics solver with a reinforcement learning agent that learns a control strategy to select the viscosity in large eddy simulations. Contrary to supervised learning, a reinforcement learning agent learns through continuous interaction with the environment rather than optimizing weights based on a labeled training dataset. Therefore, the integration of ML and HPC codes is more involved for reinforcement learning applications, since the simulations and learning have to run in parallel. Despite these challenges, the results of Ref. [73] demonstrate that `SmartSim` is capable of achieving this, offering supporting proof that the techniques employed in `SmartSim` are beneficial for future applications.

Chapter 5

Machine learning for the C2P conversion

In previous chapters, we discussed that GRMHD simulations, from which we wish to obtain accurate waveforms of GWs, are obstructed by the conservative-to-primitive conversion as a significant bottleneck. We also discussed that modern machine learning algorithms can efficiently approximate any function by learning from examples. In this chapter, we now explore the possibility of improving the C2P conversion by integrating machine learning algorithms into GRMHD simulations.[1]

5.1 Earlier work and outline

Accelerating GRMHD simulations by ML techniques has been relatively unexplored in the past. In Ref. [23], the authors addressed the potential of using neural networks to speed up the C2P conversion. Their proof of concept analysis investigated simulations in GRHD using the ideal-fluid EOS, given by Eq. (3.7). This work discussed two possible novel methods to speed up the C2P transformation. First, the authors use an NN that is trained to replace the EOS. Such a replacement can be beneficial since evaluating tabulated EOS can be costly and can introduce numerical errors, while NNs may circumvent these issues. This first kind of architecture has been named *NNEOS*. Second, the authors consider replacing the entire C2P conversion with an NN. Since the current C2P schemes involve rootfinding procedures making several calls to the EOS, which can be costly, such a replacement can avoid using the EOS directly. Hence, such a replacement can be a promising method to accelerate the simulations. This second class of methods is referred to as *NNC2P*.

Besides offering a proof of concept implementation of these ideas for the case of an analytic EOS, the authors investigate the NNEOS and NNC2P methods in realistic simulations that make use of tabulated EOS. As a test case, the authors still consider the ideal-fluid EOS, but generate a mock table that tabulates the relevant EOS values rather than relying directly on the analytic expression. This allows

[1] All code written for this and the next chapter can be found at `https://github.com/ThibeauWouters/master-thesis-AI`

the comparison between standard interpolation methods used for tabulated EOS and the NNEOS and NNC2P methods in a more realistic, yet controlled problem. The architectures corresponding to the NNEOS and NNC2P methods are shown in Figure 5.1 and will be discussed in more detail below.

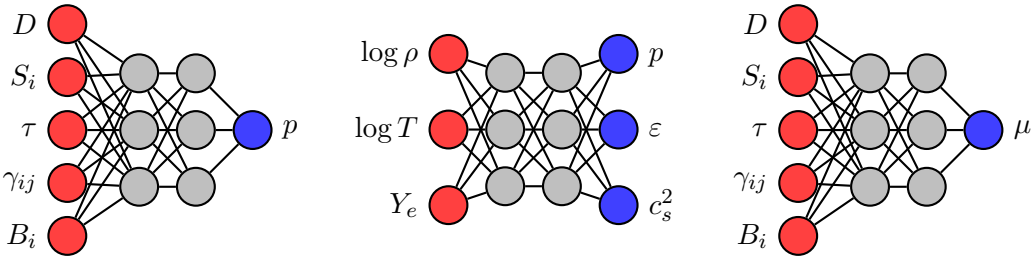

Figure 5.1: From left to right: the NNC2P, NNEOS and NN assist architectures considered in this chapter. Hidden layers are not drawn to scale.

Our work presented in this chapter is, first of all, intended to be an independent validation of the main results reported in Ref. [23]. However, the results from Ref. [23] were obtained from a solver implemented in Python. Scientific high-computing codes, such as GRMHD solvers like `Gmunu`, are instead implemented in Fortran. As such, our work extends the work of Ref. [23] by considering its application to such state-of-the-art solvers. Furthermore, we consider a novel idea which attempts to unify existing C2P schemes and ML models into a framework that improves upon the methods of Ref. [23], for reasons discussed in detail in the next chapter. This idea is able to combine the best of both worlds and, based on preliminary results, seems a promising method to speed up GRMHD simulations. Our hybrid method has a similar architecture as the NNC2P architecture, but rather than computing the pressure p, it outputs an initial guess for the rootfinding methods used in Kastaun's C2P scheme. Therefore, we refer to this method as the *NN assist*. While future work has to test this method in more realistic scenarios, we will argue in Chapter 6 that its extension is a more promising method than the ideas put forward in Ref. [23].

The outline of this chapter is summarized by Table 5.1. In Section 5.2 and Section 5.3, we consider the NNC2P method, where we replace the entire C2P conversion with an NN for simulations using analytic EOS and tabulated EOS, respectively. In Section 5.4, we consider the NNEOS method to replace the look-up procedures used for tabulated EOS with an NN. Finally, in Section 5.5, we consider our hybrid method which combines an NN with Kastaun's C2P scheme. In the next chapter, we discuss the results presented in this chapter and reflect on extensions for future work.

Table 5.1: Outline of this chapter.

EOS \ **C2P**	Kastaun		NNC2P
	Rootfinding	*Hybrid*	
Analytic EOS	*status quo*	Section 5.5.2	Section 5.2
Tabulated EOS	*status quo*	Section 5.5.2	Section 5.3
NNEOS	Section 5.4	–	–

5.2 NNC2P with analytic EOS

While the C2P conversion cannot be formulated in a closed-form analytic expression, we can view the conversion as a mapping from the conservative variables to the primitive variables. Since NNs are universal approximators, this transformation can be approximated by an NN. Recall from Eqs. (3.14) that it is sufficient to learn the pressure p as output variable, since the other primitive variables can be obtained from the pressure and the conservative variables.

Figure 5.1 shows the most general NNC2P architecture. The input consists of the conservative variables D, S_i and τ, where i is a spatial index of which the range is determined by the dimensionality considered in the simulation. Simulations involving dynamical space-times also have to consider the spatial metric γ_{ij} in the computation of the Lorentz factor, as the square of the velocity depends on the metric. Since `Gmunu` applies a conformal transformation to the space-time metric, the metric is diagonal and the node γ_{ij} effectively accounts for up to three additional input variables. Furthermore, the C2P in GRMHD also depends on the magnetic field B_i as input, as the equations of Appendix A show. However, in this section, we test the NNC2P method only in 1D GRHD test simulations with flat space-times, such that the relevant input nodes are D, S_x and τ.

5.2.1 Data generation

Training data can easily be generated by the P2C transformation. We uniformly sample datapoints for the primitive variables within the range

$$\rho \in (0, 10.1) \tag{5.1a}$$
$$\varepsilon \in (0, 2.02) \tag{5.1b}$$
$$v_x \in (0, 0.721)\,. \tag{5.1c}$$

This range is covered by the two GRHD test simulations considered below. From the sampled primitive variables, we compute the corresponding conservative variables

using the P2C transformation, *viz.*

$$D = \rho W \tag{5.2a}$$
$$S_x = \rho h W^2 v_x \tag{5.2b}$$
$$\tau = \rho h W^2 - p - D\,, \tag{5.2c}$$

where $h = 1 + \varepsilon + p/\rho$ is the enthalpy and

$$W = \frac{1}{\sqrt{1 - v_i v^i}} = \frac{1}{\sqrt{1 - \sum_{ij} \gamma_{ij} v_i v_j}}\,. \tag{5.3}$$

is the Lorentz factor, which for the simulations considered below simply reduces to

$$W = \frac{1}{\sqrt{1 - v_x^2}}\,. \tag{5.4}$$

We consider the ideal-fluid equation of state, given by

$$p(\rho, \varepsilon) = (\Gamma - 1)\rho\varepsilon\,, \tag{5.5}$$

with the adiabatic index set to $\Gamma = 5/3$.

5.2.2 Training and pruning

All neural networks considered in this work are trained with the Adam optimizer with an initial learning rate of 10^{-3} and the MSE loss function, using mini-batches of size 32. This learning rate is the suggested default for the Adam optimizer, although we have performed a line search and considered the learning curves for learning rates on a logarithmic grid to verify that this was the optimal choice for our application. We employ a learning rate scheduler during training. Specifically, if the loss in the previous 10 epochs did not improve upon the best loss value recorded so far by a certain threshold factor, the learning rate is decreased by multiplying with a factor between 0 and 1. We do not employ a full neural architecture search due to its computational complexity, but determined satisfactory architectures based on tuning procedures on small grids of interesting hyperparameter combinations. Since the training and test sets are sampled from the same distribution and can be made arbitrarily large in size, we did not easily observe overfitting during training. Eventually, we decided to use an NN with two hidden layers with 600 and 200 hidden neurons, respectively, and with sigmoid activation fuctions. Since this architecture is also used by Ref. [23], we can make a fair comparison between the observed results during training. We use a training set of 80 000 examples and a test set of 20 000 examples.

We trained the NN until a satisfactory MSE loss was obtained and the learning rate reached a specified small value indicating that the network converged to a local optimum. After training, we measure the performance of the network on newly

generated data samples using three different metrics:

$$\ell_1(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{5.6a}$$

$$\ell_2(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{5.6b}$$

$$\ell_\infty(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \max_i |y_i - \hat{y}_i| \,. \tag{5.6c}$$

In case $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are tensors rather than vectors, the above mean and max operations are taken across all axes. The ℓ_2-norm coincides with the MSE loss and hence gives insight into the training error. The other metrics provide important insights into the performance of the model for simulations. That is, the ℓ_1 and ℓ_∞ errors indicate the absolute deviations from the ground truth which are propagated throughout the simulation. The errors obtained with our NN are shown in Table 5.2.

Since we are interested in deploying NNs which can be evaluated efficiently, we explored the possibility of reducing the number of parameters in the network by pruning neurons. We have applied the neuron pruning scheme given by Algorithm 1. Figure 5.2 shows the performance of the NNs obtained by pruning a single neuron from our original, trained NN for all hidden neurons across the two layers compared to the original performance (dashed line). It is clear that the network has a lot of redundancy, especially in the second layer, since neurons can be pruned without a significant reduction in accuracy. We stopped the pruning scheme when fine-tuning

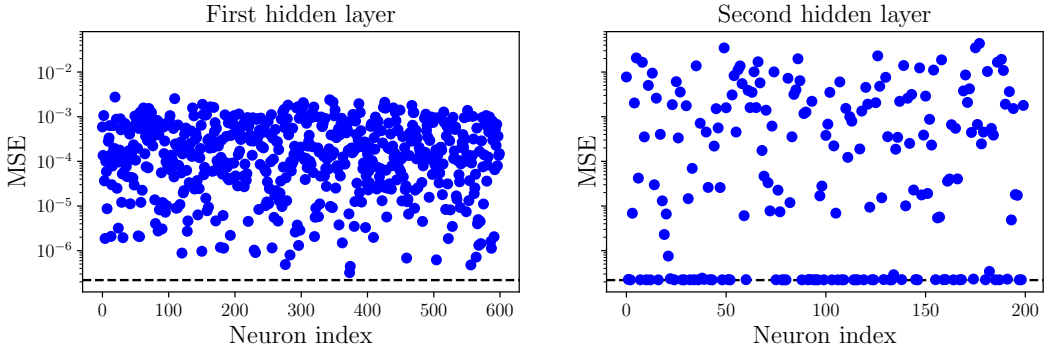

Figure 5.2: Performance of NNs obtained by pruning a single neuron from an NN with 600 and 200 hidden neurons, of which the performance is indicated by the dashed line.

(retraining) the network became computationally too expensive and infeasible. After pruning, we obtain a network with 504 and 127 neurons in the first and second layer, respectively. Figure 5.3 shows the same information as Figure 5.2 for the pruned network. The network we obtain no longer has any redundancy, as pruning a neuron would lead to a decrease of around two orders of magnitude in loss value. Remarkably, we were unable to achieve a similar performance by training an NN with an identical architecture from scratch. Hence, pruning proved to be a viable scheme to obtain balance between accuracy and efficiency. Table 5.2 compares the results of

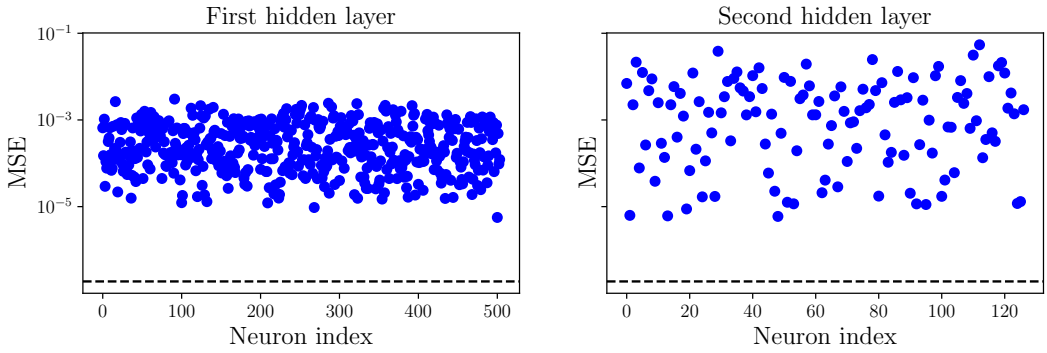

Figure 5.3: Performance of NNs obtained by pruning a single neuron from the NN obtained at the end of the pruning scheme with 504 and 127 hidden neurons, of which the performance is indicated by the dashed line.

the different NNs, also taking the work of Ref. [23] into account. We observe similar error measures as Ref. [23]. Moreover, we notice that the pruned network achieves a 1.85× reduction in the number of parameters. At the same time, the network slightly improved its performance, due to the fine-tuning (retraining) we applied after each pruning iteration.

Table 5.2: Errors between true values and predictions of NN.

Network	h_1	h_2	#params	$(\Delta p)_{\ell_1}$	$(\Delta p)_{\ell_2}$	$(\Delta p)_{\ell_\infty}$
Ref. [23]	600	200	122 801	3.84×10^{-4}	–	8.14×10^{-3}
Ours	600	200	122 801	2.67×10^{-4}	2.19×10^{-7}	9.18×10^{-3}
Ours (pruned)	504	127	66 279	2.66×10^{-4}	1.88×10^{-7}	7.95×10^{-3}

We check the performance of the model by plotting the reconstruction error in Figure 5.4. That is, we sample primitive variables from the range

$$\rho \in (0.05, 10) \tag{5.7a}$$
$$\varepsilon \in (0.1, 2) \tag{5.7b}$$

and fix $v = v_x = 0.35$. We perform the P2C to obtain the corresponding conservative variables which are used as input to the NN. Afterwards, we compare the absolute difference between the original and reconstructed pressure values. Overall, the performance of the network is quite robust across the considered parameter space. We have noticed that the largest errors arise from values close to the origin, *i.e.* when all the primitive variables are small. In this case, the conservative variables are small as well, which can easily lead to rounding errors in the computation of the NN, likely causing the results to deviate significantly from the ground truth. However, such situations are unlikely to occur in our simulations as they constitute an unphysical state.

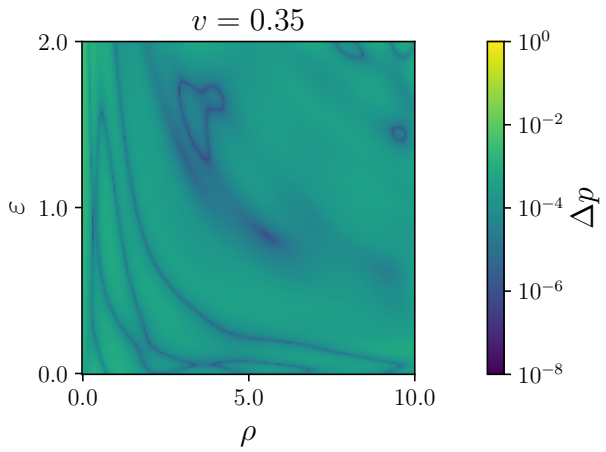

Figure 5.4: Reconstruction errors of NNC2P in the training domain.

5.2.3 Performance in simulations

After training, we deploy the NN in `Gmunu` to determine its performance in simulations. As mentioned in Section 4.3, we adopt a simple implementation and export the weights and biases of the NNs to a CSV file which is read at the start of the simulation. The NN computations are reproduced through the built-in matrix multiplication function and the activation functions are implemented from scratch. Afterwards, we assess the performance of the NNC2P scheme by comparing it with the Kastaun rootfinding scheme in two simple GRHD test simulations in 1D and in flat space-time.

Smooth sine wave

First, we consider the smooth sine wave problem [37]. Since the solution of this problem can be expressed analytically, it allows us to test the accuracy of the NNC2P method compared to the Kastaun scheme. Generally, the problem is formulated in 2D with the initial condition

$$\rho_0 = 1 + A \sin\left[2\pi \left(x\cos\theta + y\sin\theta\right)\right] , \tag{5.8a}$$

$$p = 1 , \quad v_x = v_0 , \quad v_y = 0 . \tag{5.8b}$$

We consider a 1D version of the problem by fixing $y = 0$, and consider the domain to be $x \in [0, 1]$. Moreover, we set $\theta = 0$, $A = 0.2$ and $v_0 = 0.2$. The exact solution can be expressed analytically by

$$\rho = 1 + A \sin\left[2\pi \left((x\cos\theta + y\sin\theta) - (v_x\cos\theta + v_y\sin\theta)t\right)\right] , \tag{5.9a}$$

$$p = 1 , \quad v_x = v_0 . \tag{5.9b}$$

The system is numerically evolved until $t = 5$ on a grid of size $N = 128$. Details of the solvers are given in Ref. [37]. The evolved variables are shown in Figure 5.5 together with the analytic solution. Both the Kastaun scheme and the NNC2P

scheme agree with the exact solution. The absolute errors are shown in Figure 5.6, which shows that the NNC2P method achieves a similar accuracy as the Kastaun scheme.

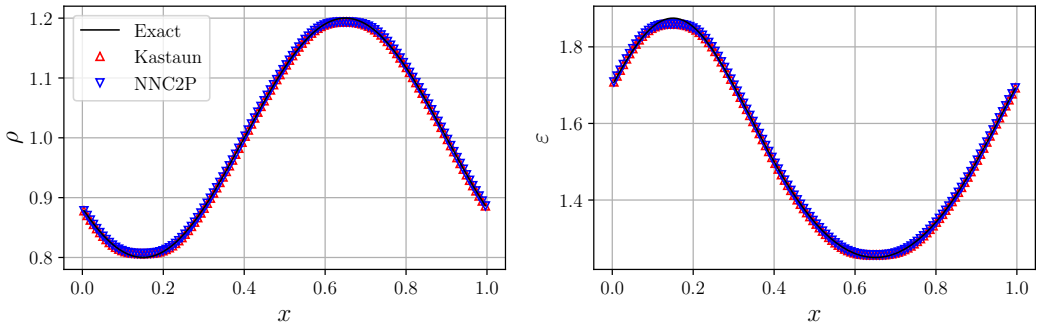

Figure 5.5: Solution of the smooth sine wave at $t = 5$ with $N = 128$.

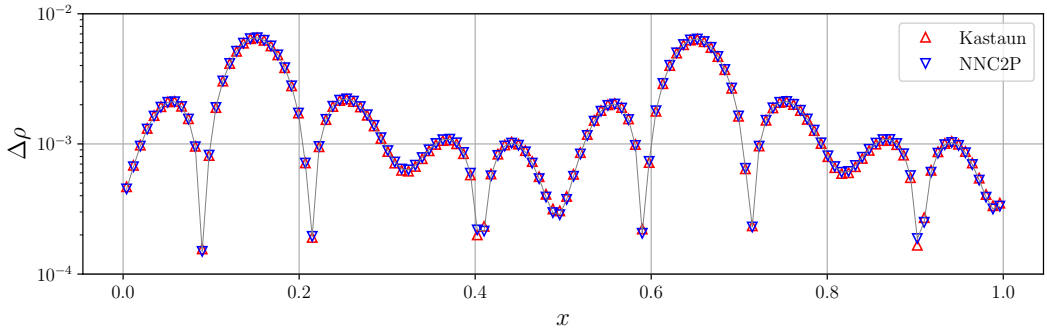

Figure 5.6: Absolute errors of Kastaun and NNC2P schemes in the smooth sine wave problem.

We report the order of convergence of both schemes. That is, we study the behaviour of the absolute errors as a function of the grid size N, following the discussion presented in Ref. [37]. More specifically, we are interested in the scaling of the relative numerical errors as a function of the grid size, δ_N, which we define as

$$\delta_N = \frac{\sum_{i=1}^{N} |\hat{\rho}_i - \rho_i|}{\sum_{i=1}^{N} |\rho_i|} , \tag{5.10}$$

where $\boldsymbol{\rho}$, $\hat{\boldsymbol{\rho}}$ represents the density of the exact solution and the numerical approximation, respectively. The convergence rate is defined as

$$R_N = \log_2 \left(\frac{\delta_{N/2}}{\delta_N} \right) . \tag{5.11}$$

We show these quantities, varying N on a logarithmic scale, in Figure 5.7. While both the original Kastaun scheme and NNC2P exhibit close to second-order convergence for lower grid sizes, we notice that the NNC2P method loses this second-order convergence for higher grid sizes, and essentially saturates at $N = 1600$. This is

likely ascribed to the fact that the computations of the NN are independent of the resolution of the grids in the simulation. Therefore, the error of the NN is dominating the numerical error for larger grid sizes. The Kastaun scheme does not suffer from this and exhibits second-order convergence across all grid sizes.

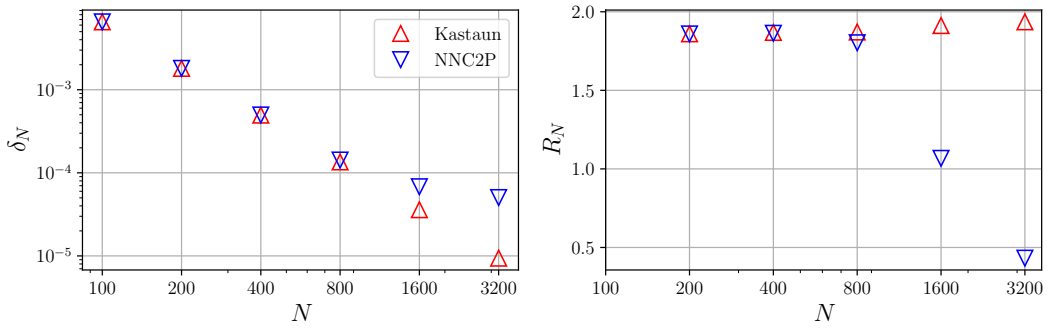

Figure 5.7: Convergence results for the Kastaun and NNC2P schemes for the smooth sine wave problem.

Shocktube problem

We also compare the accuracy of our implementation in a more demanding problem involving a discontinuity. In the well-known 1D shocktube problem, we consider a Riemann problem over the domain $x \in [0, 1]$ with two different states given by

$$(\rho, p, v_x) = \begin{cases} (10, 40/3, 0) & \text{if } x < 0.5\,, \\ (1, 0, 0) & \text{if } x > 0.5\,, \end{cases} \tag{5.12}$$

The system is evolved for $t = 0.4$ seconds and uses a grid of size $N = 16$ with an adaptable mesh that gets refined during simulations. Further details about the numerical set-up can be found in Ref. [36]. The solution can be computed exactly using the methods discussed in Ref. [74]. The final profile is shown in Figure 5.8 along with the exact solution. The NNC2P scheme is able to approximate the discontinuity and coincides closely with the Kastaun scheme. The absolute deviations on the pressure of both schemes is shown in Figure 5.9. The accuracy of the NN around the center of the domain is comparable with the errors that are obtained with Kastaun's scheme. However, at the borders of the domain, the error of the NN seems to dominate the accuracy of the NNC2P method. Indeed, the errors obtained at the edges of the domain are similar to the average absolute deviation of the NN, as reported in Table 5.2. Kastaun's scheme therefore delivers a superior accuracy compared to the NNC2P method for this more challenging simulation.

Besides comparing the accuracy of the schemes, we also compared the timing of both methods. However, we delay the discussion of this timing test to Section 5.3.2, where we also include tabulated look-up methods in the comparison.

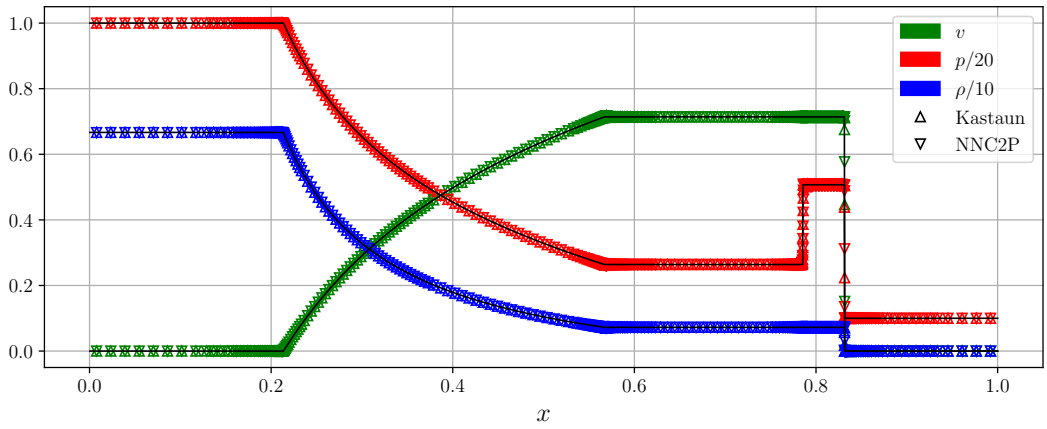

Figure 5.8: Solution of the shocktube problem at $t = 0.4$ with $N = 528$.

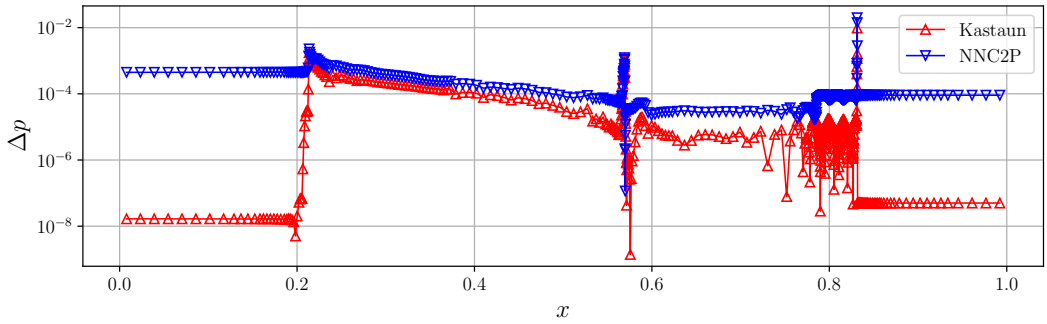

Figure 5.9: Absolute errors of the Kastaun and NNC2P schemes in the shocktube problem.

5.3 NNC2P with tabulated EOS

The tests performed in the previous section merely demonstrate the feasibility of replacing the C2P with an NN, using an analytic EOS for simplicity. However, as discussed in Section 3.2.2, realistic simulations make use of tabulated EOS. Therefore, we extend our analysis and consider tabulated EOS in this section.

To compare the performance of an NNC2P scheme in the context of tabulated EOS, we perform the following experiment. We simulate the smooth sine wave and shocktube problems with the ideal-fluid EOS, but infer the EOS from a mock table of values rather than its analytic expression. Therefore, we can rely on the same NN, since the physics does not change compared to the set-up of the previous section. This enables us to directly compare the efficiency of the NNC2P scheme to the methods used to evaluate tabulated EOS, which are more costly than evaluating an analytic EOS.

5.3.1 Generating the mock tabulated EOS

The mock table is constructed as follows. First, recall that the output values of the EOS tables are functions of the density ρ, the temperature T and the electron fraction Y_e. The electron fraction is not relevant for ideal fluids, as it is only relevant when microphysical effects are taken into account. Hence, Y_e is a dummy variable for our purposes. The temperature can be inferred from the specific internal energy density through the thermodynamic relation

$$\varepsilon = \frac{1}{\Gamma - 1} N_A k_B T \,, \tag{5.13}$$

where $N_A = 6.022 \times 10^{23}$ is Avogadro's number and k_B is the Boltzmann constant. Due to the units employed in `Gmunu`, we set $k_B = 1$ and express temperature in units of megaelectron volts (MeV). As before, we set $\Gamma = 5/3$. Since ε is contained in the range $(0, 2.02)$ for the GRHD simulations considered here, the temperature takes values in the range

$$T \in \left(0, 2.24 \times 10^{-24}\right) . \tag{5.14}$$

The EOS tables use grids of the input variables, where ρ and T are tabulated on a logarithmic grid (in base 10) and Y_e with a linear grid. To construct the table, we therefore choose a certain size for these grids, denoted by n_ρ, n_T and n_{Y_e}. The input variables are therefore constrained to the range determined by Eq. (5.1), which results in the domain

$$\log \rho \in (-307.653, 1.004) \tag{5.15a}$$

$$\log T \in (-307.653, -23.650) \tag{5.15b}$$

$$Y_e \in (0, 1) \,, \tag{5.15c}$$

where the lower bound of $\log \rho$ and $\log T$ is determined by machine precision. Afterwards, we determine ε, p and c_s^2 through the known analytic relations, *i.e.* Eq. (3.7), Eq. (3.9) and Eq. (5.13).

Microphysical EOS have 19 columns in total, of which only three (ε, p, c_s^2) were just discussed. The remaining columns provide thermodynamic relations (entropy, enthalpy,...), compositions of chemical species and chemical potentials. Each column has shape (n_ρ, n_T, n_{Y_e}). To fully replicate the microphysical EOS, we hence provide dummy values for the other columns. All variables are put in an HDF5 file and we write a custom function in `Gmunu` that is able to process our mock EOS table and infer from it during simulations.

We choose the size of the EOS table based on common sizes of EOS tables, which we infer from the EOS tables provided by Ref. [26]. We consider the SLy4 EOS as example. This table has $(n_\rho, n_T, n_{Y_e}) = (391, 163, 66)$ such that each individual column of the EOS table contains 4 206 378 entries. The mock EOS table that we generate hence has a size of 834 MB.

We check our implementation of the mock EOS table in Figure 5.10. For the smooth sine wave, a comparison between the evolved variables obtained with the analytic EOS and the tabulated EOS shows that the errors introduced by the

tabulated methods is negligible compared to the errors reported in Figure 5.6. For the shocktube, it is more instructive to consider the accuracy of the obtained pressure for the mock tabulated EOS. We find that the results are comparable to those obtained with Kastaun's scheme using the analytic EOS. Therefore, we are guaranteed that using the tabulated form of the EOS has a negligible impact on accuracy for the simulations that we consider here.

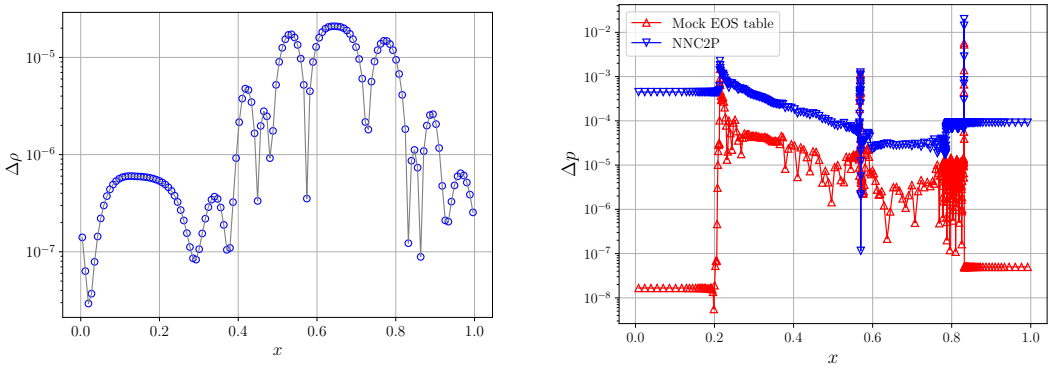

Figure 5.10: Performance of the Kastaun scheme with a mock table for the ideal-fluid EOS. *Left*: Smooth sine wave problem. *Right*: Shocktube problem.

5.3.2 Timing measurements

After establishing the accuracy of the NNC2P method compared to existing algorithms, we measure the speed[2] of the methods discussed above for the smooth sine wave and shocktube problems. The results are shown in Table 5.3.[3] We measure the efficiency of the methods in two ways. First, we measure the amount of iterations that are completed each second (its/s) during the simulation. However, this is a rather unreliable and noisy measure, which moreover depends on the size of the grid, which changes during shocktube simulations since `Gmunu` uses an adaptable mesh refinement technique. Therefore, we also consider the time-to-completion (TTC), *i.e.* the total integration time[4] required to finish the simulation, since this is in line with our goal of speeding up the simulations. As such, the TTC measure is the most informative and robust measure out of the two. The timing measurements reported below are averaged over several repetitions.

The timings of the methods in the simulations considered so far are shown in Table 5.3. We find that the NNC2P scheme is around 50 times slower than the Kastaun scheme making use of the analytic expression for the EOS in the smooth

[2]We run the code using an Intel(R) Core(TM) i5-8250U CPU, with 4 cores at 1.60GHz and 8 GB RAM. However, simulations run on a single core to have reliable timing measurements.

[3]For simplicity, we benchmark all methods on the first 1 000 iterations of the shocktube problem to reduce the influence coming from the adaptable mesh refinement.

[4]This excludes initialization of `Gmunu` and any I/O process, such as reading the EOS tables. For large-scale simulations, reading the EOS tables is expected to take up a negligible fraction of the total simulation time.

sine problem, whereas it is around 20 times slower at solving the shocktube problem. When the Kastaun scheme instead relies on the mock tabulated version of the EOS, the NNC2P scheme is around 4 to 5 times slower than the existing methods.

Table 5.3: Timing measurements of the Kastaun scheme, using an analytic or tabulated EOS, and NNC2P scheme in GRHD simulations.

	Sine wave		Shocktube	
	Speed (its/s)	TTC (s)	Speed (its/s)	TTC (s)
Analytic EOS	2561.80 ± 24.17	0.10 ± 0.001	615.36 ± 19.11	1.77 ± 0.05
Tabulated EOS	218.50 ± 35.34	1.24 ± 0.19	133.00 ± 40.22	7.63 ± 0.11
NNC2P	52.62 ± 1.84	5.22 ± 0.27	40.20 ± 7.19	36.31 ± 0.68

5.4 NNEOS

Besides replacing the entire C2P transformation with a deep learning component, we can also consider optimizing another central algorithmic step. As discussed in Chapter 3, tabulated EOS can be costly to evaluate and a source of numerical errors. Therefore, in this section, we explore whether it is possible to improve the evaluation of the EOS with an ML model, again taking NNs as test case.

5.4.1 Data preprocessing

As discussed in detail in Section 3.2.2, an EOS table provides a mapping from three input variables, $\log \rho$, $\log T$ and Y_e, to a collection of physical quantities of interest. Here, we restrict ourselves to three columns which are most important for simulations: the pressure p, the specific internal energy density ε and the speed of sound c_s^2, leading to the architecture shown in Figure 5.1.

As a test case, we work with the SLy4 tabulated EOS provided by Ref. [26]. The input features of this table take values in the range

$$\log \rho \in (3.02399601, 16.02399601) \tag{5.16a}$$
$$\log T \in (-3, 2.4) \tag{5.16b}$$
$$Y_e \in (0.005, 0.655)\,. \tag{5.16c}$$

We manually remove negative c_s^2 values from the EOS table, as they are unphysical. Afterwards, we take the log values (in base 10) of the c_s^2 column such that its range is comparable to the range of the p and ε columns. Afterwards, the output labels lie within the range

$$\log p \in (19.02463658, 33.15270695) \tag{5.17a}$$
$$\log \varepsilon \in (17.42022108, 33.74498022) \tag{5.17b}$$
$$\log c_s^2 \in (16.57442573, 21.18058849)\,. \tag{5.17c}$$

Furthermore, we convert the 3D format of the table into rows of examples to train the NN. Since the input variables have different ranges, we normalize the features of the training data by subtracting the mean and dividing by the standard deviation of each variable. Since the EOS table provides over 4 million examples, we train the NN on 5% of the entire dataset obtained through random sampling and determine the generalization of the network after training.

5.4.2 Architecture design

To design the NNEOS architecture, we focus our attention on smaller, and hence faster, architectures. Given the observation from the previous section that NNs are costly to evaluate in our simulations, we consider whether an architecture can be designed that provides a balance between speed and accuracy for simulations in `Gmunu`. Therefore, we adopt the following strategy to design our NN. Before training, we study which sizes of the NN are able to achieve a speed-up in the Fortran code. Once the appropriate size is determined, we train the architecture and assess its accuracy in simulations.

For simplicity, we restrict ourselves to architectures with one hidden layer or two hidden layers containing an equal amount of hidden neurons and investigate their efficiency at predicting the EOS values as a function of the hidden layer size. We perform a grid search over the size of the hidden layers and the activation functions, which are taken to be either sigmoid or ReLU. We expect the ReLU activation function to be more efficient to compute than the sigmoid, resulting in faster NNs. Each architecture is implemented and tested in Fortran directly after randomly initializing the weights without training the network, since we only want to estimate the speed of the NNs. The inference efficiency is measured in an artificial test case, where the `Gmunu` environment is initialized with a single gridpoint. We measure the time it takes for each NNEOS architecture to predict 1 000 000 random input data sampled from the range of the EOS table. The timings are compared to those of the look-up procedure implemented in `Gmunu`, which uses the trilinear interpolation code given in Ref. [75] and discussed in Appendix B. We report the average prediction time with one standard deviation for each architecture, averaged over 10 repetitions, in Figure 5.11. The gray shaded region shows the timing results of the interpolation method, again with one standard deviation. As expected, the ReLU activation is faster than the sigmoid activation function. We notice that only very small architectures are able to provide a faster replacement of the interpolation method.

5.4.3 Performance in simulations

To keep a balance between speed and accuracy, we will consider the performance of an architecture with two hidden layers of size 20 with ReLU activation functions in a simulation.

We train the NNEOS architecture with a similar set-up as the NNC2P architecture until convergence, which is determined by reaching a learning rate smaller than 10^{-8}.

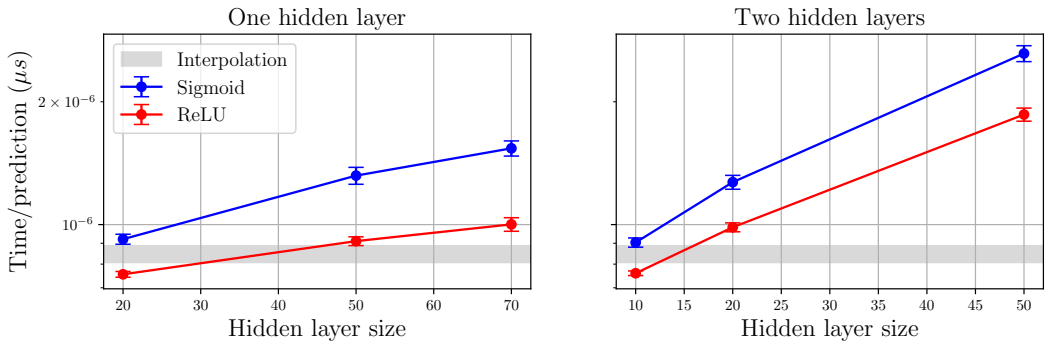

Figure 5.11: Timing results of the NNEOS architectures implemented in Fortran compared to the interpolation method.

The errors are shown in Table 5.4. The errors between the training set and the entire table are comparable, except for the error on the speed of sound in ℓ_∞ norm which is overall quite high. However, this can be partially attributed to the fact that the tabulated speed of sound values are highly irregular or unphysical in some regions of the parameter space.

Table 5.4: Errors of an NNEOS architecture with two hidden layers containing 20 hidden neurons and ReLU activation functions. The NN is trained on a subset containing 5% of the SLy4 EOS table.

		$\Delta \log p$	$\Delta \log \varepsilon$	$\Delta \log c_s^2$
Training dataset	ℓ_1	2.17×10^{-2}	3.40×10^{-2}	4.51×10^{-2}
	ℓ_2	8.38×10^{-4}	1.72×10^{-3}	4.72×10^{-3}
	ℓ_∞	1.82×10^{-1}	3.66×10^{-1}	1.67×10^{-0}
Entire table	ℓ_1	2.17×10^{-2}	3.40×10^{-2}	4.52×10^{-2}
	ℓ_2	8.41×10^{-4}	1.72×10^{-3}	4.83×10^{-3}
	ℓ_∞	1.92×10^{-1}	3.73×10^{-1}	5.2×10^{-0}

After training, we export the trained model to `Gmunu` to assess its performance in simulations. We compare the results of a simulation of a spherically symmetric, 1D neutron star simulation using both the tabulated EOS with look-up procedures and its NNEOS replacement. Since the error on the pressure values between the training dataset and the entire EOS table are comparable, we consider a minimal substitution in the test simulation. That is, only the pressure values computed by the NN are used during the simulation, and we rely on the look-up procedure to compute the energy and speed of sound. The evolved values of the density and specific internal energy density, converted to cgs units, are shown in Figure 5.12. The introduction of the NNEOS architecture clearly leads to numerical artefacts in the obtained solution.

Therefore, the architecture that we have considered does not seem capable enough to approximate the EOS to a sufficient accuracy. Since our architecture was chosen based on its efficiency, we conclude that an NNEOS replacement of microphysical EOS is unlikely to accelerate GRMHD solvers.

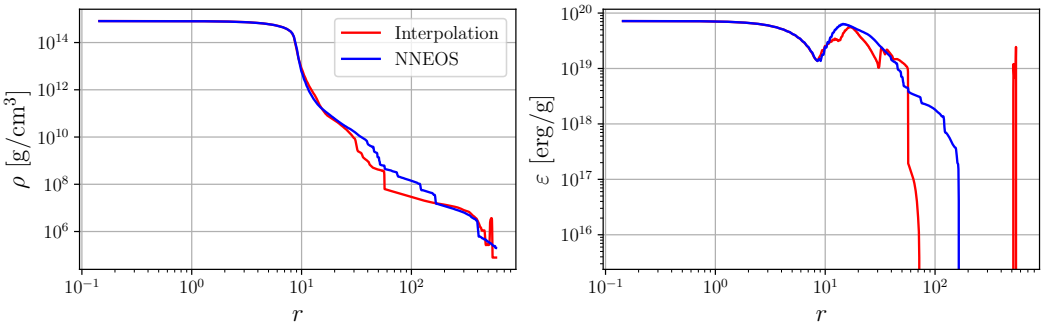

Figure 5.12: Comparison between solutions for a 1D neutron star simulation obtained with the interpolation and NNEOS methods.

5.5 Hybrid methods

Finally, we consider a hybrid approach that unifies Kastaun's C2P scheme with an NN. Our aim is to find a framework that is able to combine the best of both worlds. The Kastaun scheme has advantages over an NN, such as guaranteed accuracy and robustness. However, to assure that the solution will be found, the scheme searches for the root of its master function in a relatively large interval. Hence, the rootfinding methods involve multiple iterations which can be costly to compute. We can leverage ML to narrow this search space to reduce the amount of rootfinding iterations and EOS calls to speed up the C2P conversion.

First, we apply this idea in the smooth sine wave and shocktube simulations. However, major improvements of the hybrid scheme are mainly expected in GRMHD simulations. As discussed in more detail in Section B.3, Kastaun's C2P scheme in GRMHD involves a nested rootfinding procedure. A first rootfinding method, usually an NR method, determines a root μ_+, which determines the range $(0, \mu_+]$ in which a second rootfinding procedure, usually Brent's method, searches for the root μ from which the primitive variables can be reconstructed. Properly restricting this domain is important. For instance, the root may not be contained in a domain that is too small, causing the C2P to fail. On the other hand, Kastaun's master function can contain kinks outside of the domain $(0, \mu_+]$, which also might cause the scheme to fail to converge.

Our goal is to implement a hybrid approach which bypasses the first NR rootfinding procedure by training an NN to predict an appropriate range to look for the root μ that is provided to the second rootfinding procedure. Therefore, the architecture we consider is quite similar to the NNC2P architecture and is therefore trained in a similar way. The key difference is that the NN outputs a prediction of the root μ

instead of the pressure p. The value of μ is determined by

$$\mu = \frac{1}{Wh}\,, \tag{5.18}$$

where $W(v^i)$ is the Lorentz factor and $h(\rho, \varepsilon)$ is the enthalpy. Therefore, μ can be computed if the primitive variables are known, such that we can use the same data generation technique considered for the NNC2P method. While the NNs compute an estimate of the root μ, Brent's method takes an interval (μ_-, μ_+) as a starting guess. Therefore, we have to convert our point estimate into an interval. Here, we artificially determine this interval by specifying a fixed width w, such that the NN predicts the range

$$(\hat{\mu} - w, \hat{\mu} + w)\,, \tag{5.19}$$

where $\hat{\mu}$ is the prediction of the NN. We take the width to be either 0.1 or 0.01. In our tests performed below, this ensures that the roots are contained in our predicted intervals. Since $w = 0.1$ creates a relatively wide interval as initial guess, this reflects a situation in which the NN is uncertain about its estimate. The latter width $w = 0.01$ provides a smaller interval, reflecting the scenario where the NN is confident in its prediction. While this implementation is currently quite ad-hoc, we envision future extensions that determine the range in a more sophisticated manner, taking the uncertainty of the trained NN into account. Moreover, this ad-hoc procedure allows us to study the influence of the width of the interval on the performance of the method in a controlled manner.

The architecture we consider for this proof of concept consists of two hidden layers, each containing 20 neurons and using ReLU activation functions. We have deliberately chosen to work with small networks, since the idea is to leverage the speed of the NNs and rely on the Kastaun scheme to improve the accuracy to the desired level. Since Kastaun's master function has a unique root, we are assured that our scheme does not alter the end result of the simulation as long as we ensure that the roots are bracketed, *i.e.*, the width of the predicted interval is chosen sufficiently large enough. Therefore, the accuracy of the C2P scheme is not compromised by the NN prediction.

5.5.1 GRHD simulations

First, we implement a hybrid approach for the C2P scheme in GRHD and apply it to the smooth sine wave and the shocktube problems of GRHD. As mentioned, we generate the training data using the same principles from Section 5.2. We sample primitive variables, from which the conservative variables and μ can be computed. We refer readers to Section 5.2 for a thorough discussion on our training procedure. Since our main intent is to apply the NN assist to GRMHD simulations, the GRHD tests merely serve as verification of the validity of our implementation. We fix the width parameter w of the interval to 0.01. The average number of iterations of Brent's method for both the standard and the hybrid implementation of Kastaun's C2P scheme is shown in Table 5.5. Using the analytic ideal-fluid EOS, we obtain, on average, 1886 its/s and a TTC of around 0.164s, which is slightly slower than the

standard Kastaun scheme. While slightly reducing the average number of iterations required to converge, computing the output of the network is likely more costly than a single iteration of the rootfinding procedure in this case.

Table 5.5: Average number of iterations of Brent's rootfinding method in GRHD.

	Standard	Hybrid
Sine wave	5.55 ± 0.51	4.55 ± 0.49
Shocktube	3.80 ± 1.98	3.17 ± 1.58

5.5.2 GRMHD simulations

Moving on to GRMHD simulations for the first time in this thesis, we first point out key differences that arise from including the electromagnetic contributions in the simulations. First, all simulations take place in 3D. As a result, the NNs take S_y, S_z as input values on top of S_x. Moreover, the magnetic field itself is an independent variable as well, such that we gain three more input variables. Since we will consider a test simulation in flat space-time, the metric γ_{ij} can be neglected for our purposes.

Concretely, we consider a simple test case involving an Alfvén wave on a 1D computational domain $[0, L]$ [37, 76]. The magnetic field is uniform and given by

$$B_x = B_0\,, \quad B_y = A_0 B_0 \cos(kx)\,, \quad B_z = A_0 B_0 \sin(kx)\,, \tag{5.20}$$

where $k = 2\pi/L$ is the wave vector. The initial condition furthermore uses

$$\rho = 1\,, \quad p = 0.5\,, \tag{5.21}$$

$$v_x = 0\,, \quad v_y = -v_A A_0 \cos(kx)\,, \quad v_z = -v_A A_0 \sin(kx)\,. \tag{5.22}$$

where the speed of the Alfvén wave v_A is a complicated expression and is given by Eq. (118) of Ref. [37]. We fix $A_0 = 1$, $B_0 = 1$ and consider a domain with $L = 1$. The detailed set-up of the integration is given by Ref. [37]. The domain is subdivided into $N = 128$ grid points and the equations are integrated for one period, *i.e.* until $t = 2$.

We again rely on the ideal-fluid EOS. As before, we use the analytic expression as well as the mock tabulated version to evaluate the EOS. As mentioned, the standard formulation of Kastaun's scheme involves a nested rootfinding procedure. The first procedure, the NR scheme, takes 18 iterations to converge at each time step and for each grid point. The initial condition for B_z of the Alfvén wave and its value at $t = 2$, obtained with the standard Kastaun scheme, are shown in Figure 5.13.

Data generation and training

Compared to the GRHD test simulations, we adapt our data generation technique. That is, rather than generating training data by relying on the GRMHD equations and the P2C transformation, we run the simulation and save the relevant input and output values. After the simulation, we train an NN on this dataset. This

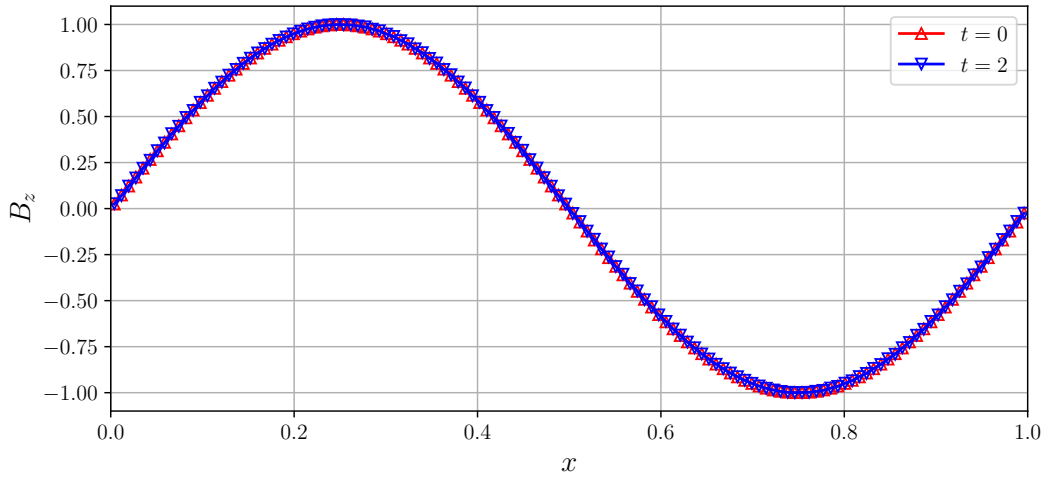

Figure 5.13: Solution of the Alfvén wave using the standard Kastaun scheme.

technique has several advantages over the one previously used in the GRHD tests. First, it is simpler in formulation and execution. Since we have to keep track of eight independent variables in the GRMHD simulations, training on a dataset uniformly sampled from an appropriate range requires more training data and becomes more costly. We can train an NN more efficiently for a specific simulation by training it on samples generated by that simulation. Second, generating samples from a uniform prior requires us to specify the range to sample from for each independent variable. While this can easily be achieved for well-known and small-scale test simulations, this becomes infeasible for large-scale and realistic simulations. Third, we envision future applications which improve their performance concurrently with running simulations through online learning. These online learners have to learn a mapping directly from data generated by simulations. Therefore, we investigate a first step towards such learners by training NNs in an offline manner on data generated from simulations. Specifically, we run the Alfvén wave simulation during which the conservative variables, the magnetic field components and the solutions of the root of the Kastaun scheme are saved to an external file. Afterwards, we train an NN to approximate a mapping from the conservative variables to the root.

Since the goal of the NN assist is to speed up the overall time-to-completion of the Kastaun scheme, we have deliberately chosen to work with relatively small (and hence, efficient) NNs. Therefore, we investigate simple architectures with two hidden layers each containing 20 neurons and using sigmoid or ReLU activation functions.

Analytic EOS

First, we consider the performance of the NN assist method in the Alfvén wave test simulation using the analytic expression for the ideal-fluid EOS. The results are shown in Table 5.6. All values reported here and below are obtained by averaging over 3 repetitions. Besides circumventing the NR rootfinding procedure, the NN assist is

also able to reduce the amount of iterations of the second rootfinding procedure. We notice that the amount of iterations saved depends on the uncertainty of the NN, with the smaller width saving an extra iteration. Moreover, the NN is around 13% faster with the ReLU activation function. Comparing the TTC between the standard implementation and the fastest hybrid implementation, we achieve a speed-up of 28%.

Table 5.6: Timing results for the NN assist, using an analytic EOS.

Method	Activation	Width	Rootfinding its		Speed (its/s)	TTC (s)
			NR	Brent		
Standard	–	–	18	6.66 ± 0.47	1085.42 ± 51.50	4.57 ± 0.08
Hybrid	Sigmoid	0.1	–	5.66 ± 0.47	1178.42 ± 41.85	4.19 ± 0.13
Hybrid	Sigmoid	0.01	–	4.67 ± 0.47	1200.00 ± 97.10	4.05 ± 0.09
Hybrid	ReLU	0.1	–	5.66 ± 0.47	1333.67 ± 121.04	3.70 ± 0.17
Hybrid	ReLU	0.01	–	4.66 ± 0.47	1363.78 ± 208.55	3.56 ± 0.15

Tabulated EOS

Next, we consider the Alfvén simulation where we use the mock tabulated ideal-fluid EOS discussed in Section 5.3 instead of the analytic expression. The results are shown in Table 5.7. We notice that the speed advantage of the ReLU function is less pronounced, likely since the tabulated methods cause the C2P schemes to overall become less efficient. Nevertheless, the NN assist offers a potential speed-up of up to 25% in this case.

Table 5.7: Timing results for the NN assist, using the tabulated version of the EOS.

Method	Activation	Width	Rootfinding its		Speed (its/s)	TTC (s)
			NR	Brent		
Standard	–	–	18	5.58 ± 1.69	207.57 ± 16.69	23.48 ± 0.54
Hybrid	Sigmoid	0.1	–	5.56 ± 1.65	227.71 ± 95.10	20.83 ± 0.09
Hybrid	Sigmoid	0.01	–	5.46 ± 1.59	252.22 ± 14.90	19.06 ± 0.16
Hybrid	ReLU	0.1	–	5.56 ± 1.64	227.29 ± 22.99	20.79 ± 0.22
Hybrid	ReLU	0.01	–	5.46 ± 1.59	254.33 ± 21.34	18.84 ± 0.19

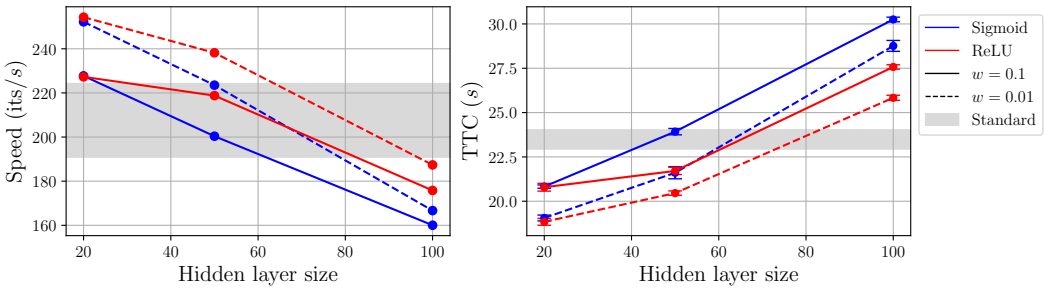

Figure 5.14: Speed of the NN assist method as a function of the size of the NN. The architectures considered have two hidden layers of equal size.

While the above results are obtained with an NN with two hidden layers, each having 20 hidden neurons, we also considered the impact of changing the size of the hidden layers on the speed of the NN assist compared to the C2P using interpolation routines. The results of this analysis are shown in Figure 5.14.[5] We notice that for increasing size of the networks, we again find that networks with a ReLU activation function are faster than those using a sigmoid. Most likely, the efficiency of the C2P conversions is mainly impacted by computing the output of the network rather than the rootfinding iterations for larger architectures. Moreover, we notice that across all architecture sizes, the hybrid method is faster if the width of the predicted interval is smaller, which agrees with our earlier observations. From the analysis, we conclude that networks containing more hidden neurons are still able to achieve a speed-up of the simulations, although this is a case-dependent issue and naturally depends on the simulation considered. Therefore, future work has to investigate the results of our methods in more realistic simulations.

[5] For clarity, we only show the errorbars for the TTC, as the amount of iterations per second is generally a noisy measure of speed.

Chapter 6

Discussion and outlook

After presenting our results in the previous chapter, we provide a critical reflection on the use of ML algorithms for the C2P conversion. Finally, we conclude this thesis by providing remarks on future implementations as well as possible extensions of our work, taking other ML approaches into consideration.

6.1 Discussion

As our work directly extends that of Ref. [23], we first provide a comparison between our results and those of Ref. [23]. Afterwards, we compare the three methods studied in the previous chapter and reflect on their applicability to more realistic simulations in future extensions.

6.1.1 Comparison with earlier work

The work presented in Sections 5.2 – 5.4 provided an extension of the work of Ref. [23]. Therefore, we provide a thorough and critical comparison between the methodologies and results of both works.

NNC2P

Our results reported in Section 5.2 and Section 5.3 show discrepancies with those reported in Ref. [23] which performed a similar analysis. In Table 2 of Ref. [23], the authors show that an NNC2P surrogate model can yield a speed-up with a factor of around 22 for an architecture with 600 and 200 neurons in the first, respectively second hidden layer compared to interpolations of tabulated EOS. Our results indicated that, instead, our NNC2P model was around 4 to 5 times slower, despite having only 504 and 127 hidden neurons, hence only having slightly over half as much parameters. This discrepancy is likely the result due to three key differences between our methods and those of Ref. [23], which we now discuss in more detail.

A first important difference is the programming language. Ref. [23] used a hydrodynamics solver programmed in Python, whereas `Gmunu` is written in Fortran. This change can already lead to different levels of optimization of central operations

used in the algorithms, such as matrix multiplications used to perform inference on the NNs. This can already influence a comparison between different techniques. As a small demonstration of this claim, we investigate the speed of an NN compared to an optimized interpolation method in both Fortran and Python. In Figure 5.11, we discussed the efficiency of NNs compared to the interpolation methods used in `Gmunu` for the tabulated EOS. We observed that an NN with a single hidden layer of size 50 and ReLU activation functions is slightly slower than the interpolation method. Here, we perform the same analysis in Python. To make a fair comparison, we implement the interpolation routine used by `Gmunu`, which can be found in Refs. [75, 77] in Python and optimize it with Numpy.[1] We compare the timing of this interpolation routine to that of our NN, which is evaluated using a method that mimics the evaluation performed in `Gmunu` rather than relying on the built-in methods of PyTorch. That is, we extract the parameters of the NN as vectors and matrices and hard-code the computational steps that the NN performs in a function. The results are shown in Figure 6.1. Besides the clear, overall speed advantage of Fortran, we notice that the most efficient method differs between the two programming languages. While this comparison only considers a specific application, it demonstrates that a comparison of the efficiency of methods depends on the programming language considered. Therefore, it is likely that the Kastaun C2P scheme with rootfinding methods is executed more efficiently in Fortran than Python, which could result in the observed discrepancy between our results and those of Ref. [23].

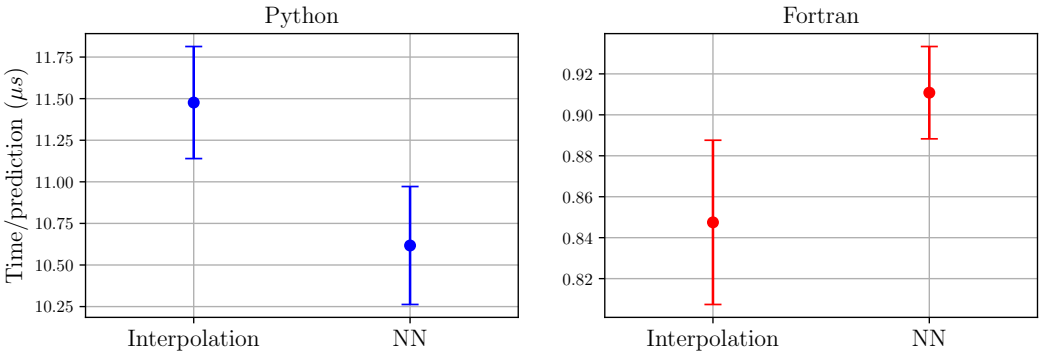

Figure 6.1: Comparison between interpolation methods and NNEOS in Python and Fortran.

Second, the rootfinding C2P scheme that we considered differs from that used in the analysis of Ref. [23]. The authors apply an NR scheme with master function

$$f(p) = p_{\text{ideal}}(\rho, \varepsilon) - p\,, \tag{6.1}$$

with p_{ideal} representing the ideal-fluid EOS given by Eq. (3.7). Our scheme, on the other hand, is that of Ref. [27] which uses a different master function and rootfinding method, which could affect performance as well. Third, our comparison between the NNC2P and interpolation methods for tabulated EOS (*i.e.*, Section 5.3) uses

[1] Notably, these interpolation methods tend to be 20, respectively 25 times faster than SciPy's `RegularGridInterpolator`, respectively `interpn`, methods.

a slightly different experimental set-up. In Ref. [23], the authors constructed a tabulated EOS with mock values from the ideal-fluid EOS, as we have done here as well. However, their EOS table has shape $(n_\rho, n_T, n_{Y_e}) = (500, 500, 500)$ whereas our table has shape $(391, 163, 66)$. Therefore, each column of the mock EOS table of Ref. [23] has 125 000 000 entries. Assuming that Ref. [23] only tabulated the quantities of relevance in the GRHD test simulations considered (*i.e.*, $\log\rho$, $\log T$, Y_e, $\log\varepsilon$, $\log p$ and c_s^2), the table already requires more than 2 GB. Our table, on the other hand, has only 4 206 378 entries for each column and takes around 830 MB of memory while tabulating values for all 19 columns that are provided by a typical EOS obtained from Ref. [26]. In particular, tabulating 500 values for the electron fraction Y_e seems excessive and hardly occurs in actual EOS tables. Therefore, it appears that the EOS table used for the analysis of Ref. [23] is larger than those used by realistic simulations, which could potentially further slow down the tabulated methods and bias the comparison.

In the end, the accumulation of these affects likely results in our observation that an NN as surrogate model for the entire C2P conversion is slower for an optimized solver written in Fortran, such as `Gmunu`. Since most HPC codes are written in Fortran, we believe that the experiments reported in this work are a more realistic reflection of the actual performance of such NN surrogate models for future extensions.

NNEOS

Besides proposing the NNC2P algorithm, Ref. [23] also proposed the NNEOS algorithm, where an NN approximates the quantities determined by the EOS. As a proof of concept, only the ideal-fluid EOS was studied in Ref. [23]. Hence, the authors use an architecture which takes ρ, ε as input and returns p as output. A second architecture additionally outputs derivatives of the pressure, *i.e.* $\partial p/\partial\rho$ and $\partial p/\partial\varepsilon$, as these are used in the computation of the speed of sound c_s^2 as shown by Eq. (3.9). Evaluating the EOS with an NNEOS model, the authors reported a speed-up of a factor 3 to 7 for these two architectures compared to standard interpolation methods.

This analysis is again biased and leads to an optimistic conclusion which is unrepresentative for realistic simulations. First of all, we again have to take into account that the mock table for the ideal-fluid EOS used in Ref. [23] does not contain all the dependent variables of the EOS and uses an unrealistic size. Furthermore, by only considering the ideal-fluid EOS as proof of concept, the analysis ignores three aspects that will affect the conclusions drawn from it. First, as discussed in Chapter 3, the EOS tables take $\log\rho$, $\log T$ and Y_e as input rather than ρ and ε which already alters the architecture that should be considered. Second, an NNEOS architecture that has to replace a tabulated EOS should output all dependent variables that the EOS provides which includes compositions, chemical potentials and thermodynamic relations besides the pressure and its derivatives. Moreover, the authors did not investigate the possibility of using automatic differentiation to predict the derivatives of the pressure, which seems a promising and more efficient way of obtaining the required derivatives. Third, the mapping that the NNEOS has to approximate becomes more complicated when considering a tabulated EOS. In the ideal-fluid EOS,

there exists a simple, analytic relationship between the input and output neurons, such that an NN can easily learn the mapping to a high degree of accuracy. This is not the case for a tabulated EOS, which implies that an NN must have a higher capacity to learn the mapping to the same level of accuracy. Therefore, a realistic replacement of microphysical EOS tables should consider deeper and wider networks, which will significantly impact the performance of the NNEOS replacement.

The analysis presented in this work addressed two of these points. First, we had a more representative estimate of the accuracy that can be attained with an NNEOS replacement by training on data from a microphysical EOS, taking the SLy4 EOS as example. Second, we considered an architecture that uses the correct input variables. However, we did not yet include all possible output variables in our experiments. Our observations seem to indicate that accelerating a Fortran solver, such as `Gmunu`, is only possible with small architectures which turn out to be incapable of approximating the EOS to a desirable degree of accuracy. We believe that our experiments provide a more realistic setting for future extensions of this approach. Therefore, our results seem to indicate that this proposed method is unable to scale to realistic simulations.

6.1.2 Critical reflection on the proposed algorithms

Continuining the discussion of the previous section, we critically examine the ML algorithms studied in this work.

In Section 3.3, we provided a few criteria taken from Ref. [28] to evaluate C2P schemes, namely speed, accuracy and robustness. A scheme based on rootfinding procedures comes with guaranteed accuracy, since these algorithms use a tolerance parameter that stops the rootfinding procedure when a desired accuracy is achieved. Additionally, the Kastaun scheme is formulated in terms of a master function that guarantees that the solution is unique. Moreover, the scheme is able to detect if the evolved variables are physically invalid and applies corrections accordingly to enforce a valid solution. Therefore, the Kastaun scheme provides a scheme with high accuracy and robustness.

When using an ML model to replace a C2P scheme, we therefore have to consider the impact of this replacement on these three criteria. ML models can potentially be faster than existing methods due to their ability to learn from examples. Most ML models are also flexible in design such that their capacity can be changed in order to achieve a desired accuracy. This flexibility leads to a trade-off between speed and accuracy that has to be studied experimentally. However, the most significant risk associated with the replacement of rootfinding schemes with ML models lies in the potential loss of robustness. In the NNC2P approach, the NN will be unable to detect unphysical input and will still predict values for the primitive variables. Moreover, if an input is provided from a region of the parameter space in which the ML model has not been trained, we risk to make an inaccurate prediction that propagates throughout the remainder of the simulation. This is demonstrated for our pruned NNC2P network. In Figure 5.4, we showed the reconstruction error of the network for values sampled from its training domain, which showed that the

model was accurate on this domain. However, in Figure 6.2, we show the same reconstruction error for a larger domain. We clearly notice that the NN is unable to generalize well outside of the domain it is trained on, indicating that the method is not robust. While this is obviously expected, as the network was not trained on values from these regions, it shows that a feasible NNC2P model can only provide robust predictions for input values inside of its training domain, which sets a limitation on the applicability of the model. This is certainly undesirable as it might make a simulation fail, wasting computational effort and having to resort to standard algorithms again. Therefore, a faster C2P scheme without a guarantee of robustness might turn out to be computationally more intensive in the end. Hence, ML models that wish to replace the C2P should not sacrifice robustness.

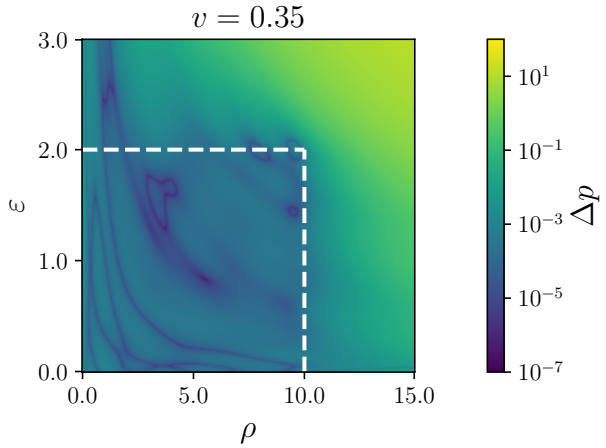

Figure 6.2: Reconstruction error of NNC2P outside of its training domain, indicated by the white dashed lines.

Given this problem, we therefore proposed a hybrid scheme that integrates an NN within the Kastaun scheme. Such a scheme takes a prediction, computed efficiently by an ML model, and provides it to the rootfinding method of the Kastaun scheme which improves the initial guess to the desired accuracy. An additional advantage of this method is that we can sacrifice accuracy of the ML model in favour of speed, since the rootfinding procedures guarantee that the end result is nevertheless accurate. We have demonstrated that such a hybrid scheme can lead to a C2P scheme which is around 25% faster for GRMHD simulations using interpolations on tabulated EOS. Moreover, our tests are performed in a Fortran solver which implies that it provides a realistic estimate of the potential of the method for more realistic simulations.

Finally, an important critical remark has to be made regarding the fact that ML models have to be trained on data. This leads to two implications for future applications of ML surrogate models in GRMHD simulations, not necessarily restricted to the methods proposed here. First, the dependence on data implies that ML models are EOS dependent. Since different microphysical EOS predict different values for the dependent variables, a training dataset for an ML model considers a specific EOS. As a result, if one wishes to use an ML surrogate model in simulations

adopting different EOS, a different realization of the ML model has to be trained for each EOS. This is another argument in favour of smaller models. Second, since the C2P training data has to be generated from the P2C transformation, one has to specify the range from which the training data has to be sampled. Whereas this was possible within the simple test cases considered in this work, this becomes infeasible for realistic simulations. Moreover, realistic simulations can cover a vast region of the parameter space, such that we have to train on a large dataset which introduces additional computational costs. Simulations likely only cover specific regions in the parameter space, since the values of the evolved variables are correlated over time. Training from data sampled from a uniform prior distribution is therefore not the most efficient.

Our test case of the hybrid method tried to partially alleviate this problem by changing the data generation technique. Instead of sampling training data from a uniform distribution over the parameter space, we applied a distribution determined by the simulations. Here, we have considered an extreme case where the training data is sampled directly from the simulation itself. This data generation technique avoids the need to determine the range of interest beforehand at the expense of having to run a simulation to obtain the training data. As a result, the training data is limited in range such that the NN can be trained efficiently, especially combined with the fact that we use smaller architectures and rely on the Kastaun scheme for accuracy. However, this likely results in a model with worse generalization such that alternatives have to be explored that find a balance between a uniform prior and training on simulation data.

6.2 Future work

After a critical reflection of the methods we have proposed and explored in this work, we propose a few directions in which our work can be extended.

A first point is to improve the implementation of NNs in Fortran. We have made several attempts already to optimize the Fortran implementation. First, we have integrated the `roseNNa` library, which we introduced in Section 4.3, within `Gmunu`. After comparing the timing between our hard-coded implementation and `roseNNa`, we did not observe any noticeable difference. Besides, we have also changed the matrix multiplication method used in our implementation. We have considered the use of the BLAS package which offers optimized linear algebra operations [78]. Here as well, we did not observe significant improvements. For future work, it would be interesting to consider the results after integrating a more sophisticated library within `Gmunu`, such as `SmartSim` discussed in Section 4.3.

Furthermore, we wish to apply our hybrid scheme to more advanced and challenging simulations. Such simulations offer new challenges for any ML surrogate model since a wider range of the input variables is covered. Moreover, we gain additional input nodes for the NN if we consider simulations with dynamical space-times and in higher dimensions. Using a microphysical EOS adds further complexity to the training dataset. It is interesting to check whether the scheme is still able to achieve

an acceleration in more realistic scenarios.

Given the discussion of the previous sections, we believe that extensions of our work should center around hybrid schemes that provide a robust method. We can start by improving our proposed scheme by training an NN that outputs a distribution rather than a single value to predict an initial bracket for the rootfinding methods in a more sophisticated manner. NNs trained in a Bayesian setting are one possibility to achieve this.

It can also be interesting to consider different training procedures. In the previous section, we highlighted the drawbacks from training on uniformly sampled data and training on data saved during a simulation. One could potentially investigate other interesting schemes to sample training data that provides a trade-off between these two methods. More interestingly, we could investigate the potential of using online learning, where the ML model is trained concurrently with the simulation. This provides another motivation to consider the integration of `SmartSim`, as it offers the ability to perform online learning of ML models concurrently with running a Fortran simulation.

Furthermore, we could take inspiration from physics-inspired neural networks (PINNs), which have already been used in the field of fluid mechanics [79]. A PINN is trained with a loss function that takes errors violating known physical constraints into account. Most PINNs are used as surrogate models for partial differential equations, where the traditional loss function consists of the errors on the differential equation, the boundary condition, the initial condition and simulation predictions and measurements. While the C2P is not a differential equation, future work can borrow ideas from the field of PINNs and add physical prior knowledge to the loss function to improve the generalizability of the model outside of its training domain.

Future work can also invent new schemes that combine the robustness of existing C2P schemes in a more efficient manner than proposed here. Furthermore, we have not yet explored alternative ML models that are potentially better suited for our applications. For instance, we have briefly considered the use of recurrent neural networks that iteratively improve their predictions until converging towards a more accurate result. However, training these networks is much more challenging. Moreover, to achieve a high accuracy, we need many hidden neurons that imply that the method likely becomes computationally more expensive. Furthermore, this again implies that we lose the robustness of the Kastaun scheme. Another interesting option is to consider graph neural networks (GNNs) [80]. GNNs are a specific class of NN architectures used to represent graphs. Hence, GNNs are suitable for structural data with explicit, known relational structure. In our case, we could model each physical variable of interest as a node in the graph and encode known dependencies, such as equations that hold among a set of variables, as edges within the graph. Going beyond deep learning, there is a plethora of other ML techniques to be considered for our application. However, we remark that a major challenge in relying on other ML techniques is that the models must easily be exported to the Fortran language, which was feasible for NNs but might be harder to achieve for other ML models. However, this problem can potentially be circumvented by the `SmartSim` library.

Furthermore, we remark that the work presented here only considered simulations

in GRHD and ideal GRMHD. Future extensions ideally have to go beyond the ideal MHD limit and moreover investigate the potential application of ML surrogate models for the C2P conversion in simulations that, *e.g.*, also take the neutrino transport into account.

It can also be interesting to delve deeper into the idea of replacing the tabulated EOS with ML models and its applications beyond the C2P problem. While we have demonstrated that NN replacements are likely slower than existing interpolation methods, ML models can offer other advantages. As mentioned before, tabulated EOS are expensive in memory, with a standard EOS table easily taking more than 800 MB of memory, with tables of higher resolution going up to 2 GB. Therefore, we can consider compression techniques that reduce the amount of memory required to use tabulated EOS. An NN is one option, as we only have to load the weights and bias parameters which are small in size even for larger and accurate networks. However, other ML models may be more suitable for this application. For instance, we can consider the use of support vector machines (SVM) for this problem. SVMs can be applied to regression problems and have an advantage over NNs by being built on top of the mathematically well-founded statistical learning theory. SVMs are linear models that rely on the well-known kernel trick to be able to learn non-linear mappings as well. Like the look-up procedures currently used for tabulated EOS, SVMs are non-parametric models, meaning that their structure depends on the provided training data. However, the look-up procedures, which are a form of locally weighted or nearest-neighbours regression, have to retain all the provided training data in order to make new predictions. SVMs, on the other hand, lead to sparse models, where only a subset of the original training data (known as the support vectors) is used for making predictions. As such, SVMs are attractive since they combine the advantages of parametric and non-parametric models. Hence, an approach relying on SVMs could potentially require less memory and still offer a similar accuracy.

6.3 Conclusion

In this thesis, we have investigated the potential of leveraging machine learning (ML) in order to speed up numerical relativistic hydrodynamics solvers to tackle challenges in the field of gravitational wave astrophysics. Specifically, we have proposed three methods to optimize the conservative-to-primitive (C2P) inversion, a central but computationally expensive algorithmic step. First, we investigated the possibility of replacing the entire C2P transformation with a neural network (NN), which we referred to as NNC2P. We have successfully integrated this NN in the `Gmunu`, a numerical solver written in the Fortran programming language. However, we have demonstrated that the NNC2P method is slower than existing C2P schemes. Specifically, schemes that use an analytic equation of state (EOS) are up to 50 times faster than the NN, while schemes that rely on a tabulated EOS, which are generally slower to evaluate, are still 4 to 5 times faster than the NN. Moreover, we presented arguments that disfavour this technique, the most significant being the fact that the

method is not guaranteed to be robust.

Second, we have investigated a method to replace the evaluation of the EOS with an NN, which we called NNEOS, for its potential to provide faster evaluation methods than the look-up procedures that are currently used for tabulated EOS. We showed that it is infeasible to accelerate existing Fortran codes with this method without sacrificing accuracy. However, future work can still consider extensions where the EOS is replaced by ML models that can improve the robustness of evaluations with an EOS table while requiring less memory.

Finally, we have proposed a novel, hybrid scheme that tries to optimally combine the currently used schemes with the benefits of ML by learning to provide accurate, initial estimates for the rootfinding methods used in existing schemes. We have shown that this method offers a potential speed-up of around 25% in GRMHD schemes, without compromising accuracy or robustness. Moreover, we have shown that our scheme is able to learn directly from data provided by simulations, such that our method can be further refined and improved by training the NNs in an online setting. Therefore, this scheme improves upon state-of-the-art C2P schemes and seems a promising method to explore in more detail in future work.

Appendices

Appendix A

GRMHD equations

In Chapter 3, we introduced the GRMHD evolution equations, written in the conservative formalism:

$$\frac{\partial \boldsymbol{\mathcal{C}}}{\partial t} + \boldsymbol{\nabla} \boldsymbol{F}(\boldsymbol{\mathcal{C}}) = \boldsymbol{S}\,, \tag{A.1}$$

Here, we give further details on the precise nature of these equations for the ideal GRMHD case and the relation between conservative and primitive variables [28]. The conserved variables are

$$\boldsymbol{\mathcal{C}} = (D, S_i, \tau, B^i, DY_e)\,, \tag{A.2}$$

which contain the conserved variables that we introduced in GRHD as well as the magnetic field B^i and the electron fraction Y_e such that the present discussion holds for GRMHD simulations making use of microphysical EOS. The primitive variables are

$$\boldsymbol{\mathcal{P}} = (\rho, v^i, \varepsilon, B^i, Y_e)\,. \tag{A.3}$$

As in the GRHD case, there exists a P2C transformation, given by

$$D = \rho W \tag{A.4}$$

$$S_i = (\rho h + b^2) W^2 v_i - \alpha b^0 b_i \tag{A.5}$$

$$\tau = (\rho h + b^2) W^2 - \left(p + \frac{b^2}{2}\right) - (\alpha b^0)^2 - D\,, \tag{A.6}$$

where $h = 1 + \varepsilon + p/\rho$ is the enthalpy, $W = \left(1 - v^2\right)^{-1/2}$ is the Lorentz factor and α is the lapse function. The vector b^μ is a derived quantity, for which the most important relations are

$$\alpha b^0 = W B^i v_i \tag{A.7}$$

$$b_i = \frac{B_i}{W} + \alpha b^0 v_i \tag{A.8}$$

$$b^2 = \frac{1}{W^2}\left(B^2 + (\alpha b^0)^2\right)\,. \tag{A.9}$$

Appendix B

Algorithms and C2P schemes used in Gmunu

B.1 Rootfinding algorithms

In numerical analysis, a rootfinding method is a numerical method to determine the zeroes, also called roots, of a continuous function $f(x)$. We will give a discussion for one-dimensional functions only. Here, we discuss two algorithms that are used in the C2P conversion in `Gmunu`, based on Ref. [34].

B.1.1 Brent's method

Brent's method, also called the Van Wijngaarden-Dekker-Brent method, is an example of a bracketing method. These methods iteratively determine smaller intervals $[a, b]$, called brackets, that contain a root. An interval $[a, b]$ is a *bracket* if $f(a)f(b) < 0$, (*i.e.*, the function f has opposite signs at the endpoints of the interval) such that the intermediate value theorem guarantees that $[a, b]$ contains a root of the function f. In that case, we say that the root is bracketed. A first way of iteratively reducing the brackets is by the bisection method, where the interval is divided into two smaller intervals of equal size $[a, c]$ and $[c, b]$, with $c = (a + b)/2$ and determining which of the two is a bracket. Another option is the regula falsi method, closely related to the secant method, which instead uses

$$c = \frac{af(b) - bf(a)}{f(b) - f(a)}\,. \tag{B.1}$$

The secant and regula falsi methods are generally faster than the bisection method and can achieve superlinear convergence. However, the bisection method is guaranteed to converge. Brent's method is able to combine the superior convergence speed of the secant method with the sureness of the bisection method by using a combination of root bracketing, bisection and inverse quadratic interpolation. The latter uses three prior points to fit an inverse quadratic function (with x being a quadratic function of y) to determine the next estimate of the root. Additionally, Brent's method has

contingency plans in case the root is no longer bracketed after an update. If these three points are $(a, f(a))$, $(b, f(b))$ and $(c, f(c))$, then the interpolation formula is

$$x = \frac{(y-f(a))(y-f(b))c}{(f(c)-f(a))(f(c)-f(b))} + \frac{(y-f(b))(y-f(c))a}{(f(a)-f(b))(f(a)-f(c))} + \frac{(y-f(c))(y-f(a))b}{(f(b)-f(c))(f(b)-f(a))} \tag{B.2}$$

The next estimate for the root is determined by setting $x = y(0)$, which gives

$$x = b + P/Q\,, \tag{B.3}$$

where P and Q are determined by R, S, T, defined as

$$R = \frac{f(b)}{f(c)}\,, \quad S = \frac{f(b)}{f(a)}\,, \quad T = \frac{f(a)}{f(c)}\,. \tag{B.4}$$

through the formulae

$$P = S[T(R-T)(c-b) - (1-R)(b-a)]\,, \tag{B.5}$$

$$Q = (T-1)(R-1)(S-1)\,. \tag{B.6}$$

Brent's method is the recommended choice of method for a general one-dimensional function of which only the values and not its derivative or functional form are known.

B.1.2 Newton-Raphson

A well-known rootfinding algorithm is the Newton-Raphson (NR) algorithm. This method requires that the derivative of the function $\mathrm{d}f$ is known. When the derivative can be evaluated, the NR method extends the tangent to the function evaluated at the current estimate for the root, and uses the point where the tangent crosses the horizontal axis as next estimate. Given a current estimate x_i, the NR determines the next estimate as

$$x_{i+1} = x_i - \frac{f(x_i)}{\mathrm{d}f(x_i)}\,. \tag{B.7}$$

The NR algorithm converges quadratically. This very strong convergence property makes NR the method of choice for any function whose derivative can be evaluated efficiently and whose derivative is continuous and non-zero in the neighbourhood of a root. It is not advised to rely on the NR scheme in case the derivative has to be approximated numerically, as this will decrease the order of convergence of the method. Therefore, the NR method with numerical derivatives is always dominated by Brent's method for one-dimensional functions.

B.2 Trilinear interpolation

Microphysical EOS use a table of values provided at rectilinear grids. Therefore, numerical techniques have to be used to obtain the EOS values at arbitrary points

within this grid. Since each dependent variable of the EOS depends on three input values (namely, $\log\rho$, $\log T$ and Y_e), relativistic hydrodynamics codes usually rely on trilinear interpolation [81]. To simplify notation, we will present the trilinear interpolation method for input values x, y and z and an output variable p.

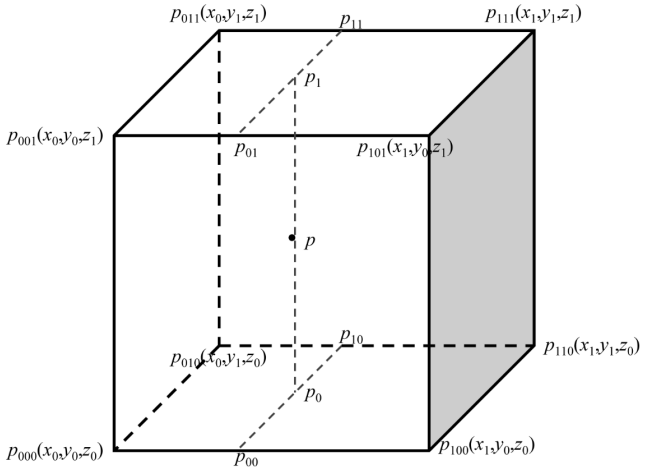

Figure B.1: Eight-point stencil used in trilinear interpolation. Figure taken from Ref. [81]

When a new input point (x, y, z) is provided, for which the value p has to be determined, trilinear interpolation performs a weighted regression using the eight-point stencil around the input point, as shown in Figure B.1. The general formula for trilinear interpolation is therefore

$$p(x, yz) = c_0 + c_1\Delta x + c_2\Delta y + c_3\Delta z + c_4\Delta x\Delta y + c_5\Delta y\Delta z + c_6\Delta x\Delta z + c_7\Delta x\Delta y\Delta z\,, \tag{B.8}$$

where Δx, Δy and Δz represent the separation between the provided input point and the starting point p_{000} as shown in Figure B.1, *viz.*

$$\Delta x = \frac{x - x_0}{\mathrm{d}x}\,, \Delta y = \frac{y - y_0}{\mathrm{d}y}\,, \Delta z = \frac{z - z_0}{\mathrm{d}z}\,. \tag{B.9}$$

Here, x_i, y_i and z_i determine the gridpoints of the table and $\mathrm{d}x$, $\mathrm{d}y$ and $\mathrm{d}z$ refer to the separation between the gridpoints in each direction, *e.g.* $\mathrm{d}x = x_{i+1} - x_i$. The coefficients c_i are determined by

$$c_0 = p_{000} \tag{B.10a}$$
$$c_1 = p_{100} - p_{000} \tag{B.10b}$$
$$c_2 = p_{010} - p_{000} \tag{B.10c}$$
$$c_3 = p_{001} - p_{000} \tag{B.10d}$$
$$c_4 = p_{110} - p_{010} - p_{100} + p_{000} \tag{B.10e}$$
$$c_5 = p_{011} - p_{001} - p_{010} + p_{000} \tag{B.10f}$$
$$c_6 = p_{101} - p_{001} - p_{100} + p_{000} \tag{B.10g}$$
$$c_7 = p_{111} - p_{011} - p_{101} - p_{110} + p_{100} + p_{001} + p_{010} - p_{000} \tag{B.10h}$$

where we denote the eight-point stencil gridpoints as defined in Figure B.1. The Fortran code of such a trilinear interpolation method can be found in Refs. [75, 77].

B.3 Kastaun's C2P scheme

The C2P conversion currently adopted in Gmunu is based on that of Ref. [27] which applies to both GRHD as well as GRMHD simulations. The procedure is summarized in Ref. [37]. We specify the full routine for the GRMHD case. Readers are referred to Appendix A for more details on the GRMHD equations. First, the following rescaled auxiliary variables are computed:

$$q = \frac{\tau}{D}\,, \quad r_i = \frac{S_i}{D} \quad \mathcal{B}^i = \frac{B^i}{\sqrt{D}}\,. \tag{B.11}$$

Next, we decompose the vectors along the directions determined by the magnetic field:

$$r^i_{||} = \frac{b^l r_l}{b^2} b^i\,, \quad r^i_{\perp} = r^i - r^i_{||}\,. \tag{B.12}$$

From this, we compute the following auxiliary quantity

$$r^2 = r^i r_i\,, \quad \mathcal{B}^2 = \mathcal{B}^i \mathcal{B}_i\,, \quad \mathcal{B}^2 r^2_{\perp} = \mathcal{B}^2 r^2 - (r^l \mathcal{B}_l)^2\,. \tag{B.13}$$

As mentioned, solving the C2P in GRMHD involves a nested rootfinding procedure. The first rootfinding procedure uses the master function $f_a(\mu)$

$$f_a(\mu) = \mu\sqrt{h_0^2 + \bar{r}^2(\mu)} - 1\,, \tag{B.14}$$

where we have defined the auxiliary functions

$$\chi(\mu) = 1/(1 + \mu\mathcal{B}^2)\,, \tag{B.15a}$$

$$\bar{r}^2(\mu) = r^2\chi^2(\mu) + \mu\chi(\mu)\,(1 + \chi(\mu))\left(r^l\mathcal{B}_l\right)^2\,, \tag{B.15b}$$

and h_0 is a lower bound for the relativistic enthalpy across the entire validity region of the EOS employed. Since the derivative of the function f_a can be expressed analytically, an NR method determines the root μ_+ of the function in the interval $(0, h_0^{-1}]$. Afterwards, a second rootfinding procedure takes the master function

$$f(\mu) = \mu - \frac{1}{\hat{\nu} + \mu\bar{r}^2(\mu)}\,, \tag{B.16}$$

where we again introduce new auxiliary quantities

$$\hat{\nu}(\mu) = \max\left(\nu_A(\mu), \nu_B(\mu)\right) , \tag{B.17a}$$

$$\hat{\nu}_A(\mu) = \left(1 + \hat{a}(\mu)\right) \frac{1 + \hat{\varepsilon}(\mu)}{\hat{W}(\mu)} , \tag{B.17b}$$

$$\hat{\nu}_B(\mu) = \left(1 + \hat{a}(\mu)\right) \left(1 + \bar{q}(\mu) - \mu \bar{r}^2(\mu)\right) , \tag{B.17c}$$

$$\hat{p}(\mu) = p_{\mathrm{EOS}}\left(\hat{\rho}(\mu), \hat{\varepsilon}(\mu)\right) , \tag{B.17d}$$

$$\hat{a}(\mu) = \frac{\hat{p}(\mu)}{\hat{\rho}(\mu)\left(1 + \hat{\varepsilon}(\mu)\right)} , \tag{B.17e}$$

$$\hat{\rho}(\mu) = \frac{D}{\hat{W}(\mu)} , \tag{B.17f}$$

$$\hat{\varepsilon}(\mu) = \hat{W}(\mu) \left(\bar{q}(\mu) - \mu \bar{r}^2(\mu)\right) + \hat{v}^2(\mu) \frac{\hat{W}^2(\mu)}{1 + \hat{W}(\mu)} , \tag{B.17g}$$

$$\hat{v}^2(\mu) = \min\left(\mu^2 \bar{r}^2(\mu), v_0^2\right) , \tag{B.17h}$$

$$\hat{W}(\mu) = \frac{1}{\sqrt{1 - \hat{v}^2(\mu)}} , \tag{B.17i}$$

$$\bar{q}(\mu) = q - \tfrac{1}{2}\mathcal{B}^2 - \tfrac{1}{2}\mu^2 \chi^2(\mu) \left(\mathcal{B}^2 r_\perp^2\right) , \tag{B.17j}$$

$$v_0^2 = \frac{r^2}{h_0^2 + r^2} . \tag{B.17k}$$

Note that, as indicated, the pressure has to be inferred from the EOS. In case microphysical EOS are used, p_{EOS} additionally depends on temperature T and electron fraction Y_e. In that case, computing $\hat{p}(\mu)$ is a costly step, since one has to transform the evolved specific internal energy density ε to the corresponding temperature T, which involves another rootfinding procedure using the tabulated EOS. Each call to the EOS moreover performs a trilinear interpolation, such that a single iteration of Brent's method can become quite costly in case tabulated EOS are used.

Appendix C

Neural network implementation in Fortran

```
1  module mod_grhd_phys_neuralnet
2
3    !> Module that replaces the C2P conversion in GRHD with a neural network
4    use mod_physics
5    use mod_grhd_phys_parameters
6
7    implicit none
8
9    private
10
11   !> Specify the architecture (by default, we use two hidden layers)
12   !> The values hard-coded here are for the pruned neural network
13   integer, parameter :: INPUT_SIZE = 3
14   integer, parameter :: HIDDEN_SIZE_1 = 504
15   integer, parameter :: HIDDEN_SIZE_2 = 127
16   integer, parameter :: OUTPUT_SIZE = 1
17   logical, parameter :: USE_SIGMOID = .true. ! True for sigmoid, false for
         ReLU
18
19   !> Define type that stores (information on) the weights and biases of the
         neural network
20   type nn_table_t
21      double precision, dimension(:,:), allocatable :: weight0 ! Weights
            first hidden layer
22      double precision, dimension(:,:), allocatable :: bias0 ! Bias first
            hidden layer
23      double precision, dimension(:,:), allocatable :: weight2 ! Weights
            second hidden layer
24      double precision, dimension(:,:), allocatable :: bias2 ! Bias second
            hidden layer
25      double precision, dimension(:,:), allocatable :: weight4 ! Weights
            output layer
```

```
26      double precision, dimension(:,:), allocatable :: bias4 ! Bias output
            layer
27    end type nn_table_t
28
29    type(nn_table_t) :: nn_tab
30
31    ! Public methods
32    public :: grhd_phys_neuralnet_init
33    public :: nn_predict
34
35  contains
36
37    subroutine grhd_phys_neuralnet_init(filepath)
38      !> Subroutine to initialize the module. Weights and biases should be in
            the same directory (filepath) and have the names shown below
39      character(*), intent(in) :: filepath
40      character(len=256) :: fname = ''
41
42      !> Initialize the neural network weights and biases here by reading the
            CSV files
43      fname = trim(adjustl(filepath))//"/weight0_flat.csv"
44      call read_matrix(fname, HIDDEN_SIZE_1, INPUT_SIZE, nn_tab%weight0)
45      fname = trim(adjustl(filepath))//"/bias0_flat.csv"
46      call read_matrix(fname, HIDDEN_SIZE_1, 1, nn_tab%bias0)
47
48      fname = trim(adjustl(filepath))//"/weight2_flat.csv"
49      call read_matrix(fname, HIDDEN_SIZE_2, HIDDEN_SIZE_1, nn_tab%weight2)
50      fname = trim(adjustl(filepath))//"/bias2_flat.csv"
51      call read_matrix(fname, HIDDEN_SIZE_2, 1, nn_tab%bias2)
52
53      fname = trim(adjustl(filepath))//"/weight4_flat.csv"
54      call read_matrix(fname, OUTPUT_SIZE, HIDDEN_SIZE_2, nn_tab%weight4)
55      fname = trim(adjustl(filepath))//"/bias4_flat.csv"
56      call read_matrix(fname, OUTPUT_SIZE, 1, nn_tab%bias4)
57
58    end subroutine grhd_phys_neuralnet_init
59
60    subroutine read_matrix(fname, nrows, ncols, matrix)
61      !> Subroutine which reads the values from a CSV, loads them into a
            matrix, and stores them into the NN type
62      implicit none
63
64      ! Declare the variables
65      character(len=256), intent(in) :: fname
66      integer, intent(in) :: nrows, ncols
67      double precision, dimension(:, :), allocatable, intent(out) :: matrix
68
69      ! Local variables for processing CSV files
70      integer :: iflag, nlines
71      integer :: i, j
```

```
72    double precision, dimension(nrows*ncols) :: values
73    ! To reshape in the correct shape after reading flattened arrays:
74    integer, dimension (1:2) :: order2 = (/ 2, 1 /)
75
76    ! Allocate memory for matrix
77    allocate(matrix(nrows, ncols))
78
79    ! Open the file for reading
80    open(unit=666, file=fname, status='old')
81    do i = 1, nrows*ncols
82      read(666,*,iostat=iflag) values(i)
83      if (iflag/=0) exit
84    end do
85    ! Reading is over, close the file
86    close(unit = 666)
87
88    matrix = reshape(values, (/ nrows, ncols /), order=order2)
89
90  end subroutine read_matrix
91
92  subroutine nn_predict(D, S, tau, p)
93    !> Make a prediction with the NN: Given the conserved variables D, S,
          tau, returns the pressure p as computed by the neural network
94    implicit none
95
96    !> Input and output of the NN
97    double precision, intent(in) :: D ! Conservative energy density
98    double precision, intent(in) :: S ! Conservative momentum density
99    double precision, intent(in) :: tau ! Conservative energy density
          relative to D
100   double precision, intent(out) :: p ! Pressure
101   double precision, dimension(3) :: x ! Input for the neural net as a
          vector (D, S, tau)
102   integer :: i, j
103
104   x(1) = D
105   x(2) = S
106   x(3) = tau
107   !> Call to make the computations:
108   call nn_compute(x, p, nn_tab)
109
110 end subroutine nn_predict
111
112 subroutine relu(x, relu_values)
113   !> ReLU activation function
114   implicit none
115
116   double precision, dimension(:), intent(in) :: x
117   double precision, intent(out) :: relu_values(size(x))
118
```

```
119     relu_values = max(0.0d0, x)
120
121   end subroutine relu
122
123   subroutine sigmoid(x, sigmoid_values)
124     !> Sigmoid activation function
125     implicit none
126
127     integer, dimension(1) :: s ! Shape of the input array x
128     integer :: i
129     double precision, dimension(:), intent(in) :: x
130     double precision, dimension(:), intent(out) :: sigmoid_values(size(x))
131
132     ! Get shape of array
133     s = shape(x)
134
135     ! Fill array with sigmoid values
136     do i = 1, s(1)
137       sigmoid_values(i) = 1.0d0 / (1.0d0 + dexp(-x(i)))
138     end do
139
140   end subroutine sigmoid
141
142   subroutine nn_compute(x, p, nn_tab_in)
143   ! Computations that the NN performs
144     implicit none
145
146     double precision, intent(in) :: x(INPUT_SIZE) ! Input of the NN
147     double precision, intent(out) :: p ! Pressure as return value (scalar)
148     type(nn_table_t), intent(in) :: nn_tab_in ! Neural network parameters
149
150     double precision :: xx(HIDDEN_SIZE_1) ! intermediate result, after first
          hidden layer computaiton
151     double precision :: yy(HIDDEN_SIZE_1) ! intermediate result first hidden
          layer after activation function
152     double precision :: xxx(HIDDEN_SIZE_2) ! intermediate result, after
         first second layer computaiton
153     double precision :: yyy(HIDDEN_SIZE_2) ! intermediate result second
         hidden layer after activation function
154     double precision :: y(OUTPUT_SIZE) ! Output NN as array
155
156     ! Do the calculation:
157     xx = matmul(nn_tab_in%weight0, x) + nn_tab_in%bias0(:,1)
158     if (USE_SIGMOID) then
159       call sigmoid(xx, yy)
160     else
161       call relu(xx, yy)
162     end if
163     xxx = matmul(nn_tab_in%weight2, yy) + nn_tab_in%bias2(:,1)
164     if (USE_SIGMOID) then
```

```
165       call sigmoid(xxx, yyy)
166     else
167       call relu(xxx, yyy)
168     end if
169     y = matmul(nn_tab_in%weight4, yyy) + nn_tab_in%bias4(:,1)
170 
171     ! Get the end result as a scalar, not an array
172     p = y(1)
173   end subroutine nn_compute
174 
175 
176 end module mod_grhd_phys_neuralnet
```

Bibliography

[1] J. B. Hartle. *Gravity: an introduction to Einstein's general relativity*. 2003.

[2] S. M. Carroll. *Spacetime and geometry*. Cambridge University Press, 2019.

[3] J. A. Wheeler and K. Ford. *Geons, black holes and quantum foam: a life in physics*. 2000.

[4] M. Maggiore. *Gravitational waves: Volume 1: Theory and experiments*. OUP Oxford, 2007.

[5] T. G. F. Li. *Extracting physics from gravitational waves: testing the strong-field dynamics of general relativity and inferring the large-scale structure of the Universe*. Springer, 2015.

[6] *LIGO - A Gravitational-Wave Interferometer*. `https://www.ligo.caltech.edu/page/ligo-gw-interferometer`. Accessed: 2023-05-15.

[7] *Virgo*. `https://www.virgo-gw.eu/`. Accessed: 2023-05-15.

[8] *KAGRA Large-scale Cryogenic Graviational wave Telescope Project*. `https://gwcenter.icrr.u-tokyo.ac.jp/en/`. Accessed: 2023-05-15.

[9] R. Hulse and H. Taylor. "Discovery of a Pulsar in a Close Binary System." In: *Bulletin of the American Astronomical Society*. Vol. 6. 1974, p. 453.

[10] 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`.

[11] gwastro. *PyCBC-Tutorials: Learn how to use PyCBC to analyze gravitational-wave data and do parameter inference*. 2022. URL: `https://github.com/gwastro/PyCBC-Tutorials`.

[12] R. Abbott et al. "Observation of gravitational waves from two neutron star–black hole coalescences". In: *The Astrophysical journal letters* 915.1 (2021), p. L5.

[13] B. P. Abbott et al. "GW170817: observation of gravitational waves from a binary neutron star inspiral". In: *Physical review letters* 119.16 (2017), p. 161101.

[14] W. Hartley. "Multi-messenger Observations of a Binary Neutron Star Merger". In: *The Astrophysical Journal Letters* 848.2 (2017), p. L12.

[15] G. Dálya et al. *Constraining Supernova Physics through Gravitational-Wave Observations*. 2023. arXiv: `2302.11480 [astro-ph.HE]`.

[16] E. Abdikamalov, G. Pagliaroli, and D. Radice. "Gravitational Waves from Core-Collapse Supernovae". In: *Handbook of Gravitational Wave Astronomy*. Springer Singapore, 2021, pp. 1–37. DOI: 10.1007/978-981-15-4702-7_21-1.

[17] T. H. Pang et al. "From spacetime to nucleus: Probing nuclear physics and testing general relativity". PhD thesis. Utrecht University, 2022.

[18] J. M. Lattimer. "The nuclear equation of state and neutron star masses". In: *Annual Review of Nuclear and Particle Science* 62 (2012), pp. 485–515.

[19] A. d. S. Schneider, L. F. Roberts, and C. D. Ott. "Open-source nuclear equation of state framework based on the liquid-drop model with Skyrme interaction". In: *Physical Review C* 96.6 (2017), p. 065802.

[20] L. Rezzolla and O. Zanotti. *Relativistic hydrodynamics*. Oxford University Press, 2013.

[21] D. Radice. "Advanced numerical approaches in the dynamics of relativistic flows". PhD thesis. 2013.

[22] M. Alcubierre. *Introduction to 3+ 1 numerical relativity*. Vol. 140. OUP Oxford, 2008.

[23] T. Dieselhorst et al. "Machine learning for conservative-to-primitive in relativistic hydrodynamics". In: *Symmetry* 13.11 (2021), p. 2157.

[24] E. O'Connor and C. D. Ott. "A new open-source code for spherically symmetric stellar collapse to neutron stars and black holes". In: *Classical and Quantum Gravity* 27.11 (2010), p. 114103.

[25] A. d. S. Schneider, L. F. Roberts, and C. D. Ott. "Open-source nuclear equation of state framework based on the liquid-drop model with Skyrme interaction". In: *Physical Review C* 96.6 (2017), p. 065802.

[26] A. da Silva Schneider, L. F. Roberts, and C. D. Ott. *SRO EOS*. https://stellarcollapse.org/SROEOS.html. Accessed: 2023-05-19.

[27] W. Kastaun, J. V. Kalinani, and R. Ciolfi. "Robust recovery of primitive variables in relativistic ideal magnetohydrodynamics". In: *Physical Review D* 103.2 (2021), p. 023018.

[28] D. M. Siegel et al. "Recovery schemes for primitive variables in general-relativistic magnetohydrodynamics". In: *The Astrophysical Journal* 859.1 (2018), p. 71.

[29] S. C. Noble et al. "Primitive variable solvers for conservative general relativistic magnetohydrodynamics". In: *The Astrophysical Journal* 641.1 (2006), p. 626.

[30] P. Cerdá-Durán et al. "A new general relativistic magnetohydrodynamics code for dynamical spacetimes". In: *Astronomy & Astrophysics* 492.3 (2008), pp. 937–953.

[31] D. Neilsen et al. "Magnetized neutron stars with realistic equations of state and neutrino cooling". In: *Physical Review D* 89.10 (2014), p. 104029.

[32] C. Palenzuela et al. “Effects of the microphysical equation of state in the mergers of magnetized neutron stars with neutrino cooling”. In: *Physical Review D* 92.4 (2015), p. 044045.

[33] W. I. Newman and N. D. Hamlin. “Primitive variable determination in conservative relativistic magnetohydrodynamic simulations”. In: *SIAM Journal on Scientific Computing* 36.4 (2014), B661–B683.

[34] W. H. Press et al. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[35] P. Mösta et al. “`GRHydro`: a new open-source general-relativistic magnetohydrodynamics code for the Einstein toolkit”. In: *Classical and Quantum Gravity* 31.1 (2013), p. 015005.

[36] P. C.-K. Cheong, L.-M. Lin, and T. G. F. Li. “`Gmunu`: toward multigrid based Einstein field equations solver for general-relativistic hydrodynamics simulations”. In: *Classical and Quantum Gravity* 37.14 (2020), p. 145015.

[37] P. C.-K. Cheong et al. “`Gmunu`: paralleled, grid-adaptive, general-relativistic magnetohydrodynamics in curvilinear geometries in dynamical space–times”. In: *Monthly Notices of the Royal Astronomical Society* 508.2 (2021), pp. 2279–2301.

[38] P. C.-K. Cheong et al. “An extension of `Gmunu`: general-relativistic resistive magnetohydrodynamics based on staggered-meshed constrained transport with elliptic cleaning”. In: *The Astrophysical Journal Supplement Series* 261.2 (2022), p. 22.

[39] P. C.-K. Cheong et al. *General-relativistic radiation transport scheme in `Gmunu` I: Implementation of two-moment based multi-frequency radiative transfer and code tests*. 2023. arXiv: `2303.03261 [astro-ph.IM]`.

[40] S. J. Chapman. *Fortran 90/95 for scientists and engineers*. McGraw-Hill Higher Education, 2004.

[41] H. H.-Y. Ng et al. “Gravitational-wave Asteroseismology with f-modes from Neutron Star Binaries at the Merger Phase”. In: *The Astrophysical Journal* 915.2 (2021), p. 108.

[42] A. K. L. Yip, P. C.-K. Cheong, and T. G. F. Li. *General-relativistic simulations of the formation of a magnetized hybrid star*. 2023. arXiv: `2303.16820 [astro-ph.HE]`.

[43] M. Y. Leung et al. “Oscillations of highly magnetized non-rotating neutron stars”. In: *Communications Physics* 5.1 (2022), p. 334.

[44] H. Blockeel. “Machine learning and inductive inference”. In: *Uitgeverij Acco* (2010).

[45] S. J. Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[46] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[47] T. M. Mitchell et al. *Machine learning*. Vol. 1. McGraw-hill New York, 2007.

[48] M. Corporation. *Bing Chat*. 2023.

[49] T. Elsken, J. H. Metzen, and F. Hutter. "Neural architecture search: A survey". In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 1997–2017.

[50] D. Blalock et al. "What is the state of neural network pruning?" In: *Proceedings of machine learning and systems* 2 (2020), pp. 129–146.

[51] Y. LeCun, J. Denker, and S. Solla. "Optimal brain damage". In: *Advances in neural information processing systems* 2 (1989).

[52] B. Hassibi and D. Stork. "Second order derivatives for network pruning: Optimal brain surgeon". In: *Advances in neural information processing systems* 5 (1992).

[53] A. Paszke et al. "PyTorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[54] M. Chantry et al. "Opportunities and challenges for machine learning in weather and climate modelling: hard, medium and soft AI". In: *Philosophical Transactions of the Royal Society A* 379.2194 (2021), p. 20200083.

[55] L. J. Kedward et al. "The state of Fortran". In: *Computing in Science & Engineering* 24.2 (2022), pp. 63–72.

[56] J. Yin, F. Wang, and M. Shankar. "Strategies for Integrating Deep Learning Surrogate Models with HPC Simulation Applications". In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2022, pp. 01–10.

[57] F. D. Group. *Fortran and Neural Networks*. `https://fortran-lang.discourse.group/t/fortran-and-neural-networks/1268/2`. Accessed: 2023-05-21.

[58] M. Chantry et al. "Machine learning emulation of gravity wave drag in numerical weather forecasting". In: *Journal of Advances in Modeling Earth Systems* 13.7 (2021), e2021MS002477.

[59] Z. I. Espinosa et al. "Machine learning gravity wave parameterization generalizes to capture the QBO and response to increased CO2". In: *Geophysical Research Letters* 49.8 (2022), e2022GL098174.

[60] E. Rabel et al. *`Forpy`*. `https://github.com/ylikx/forpy`.

[61] P. Contributors. *TorchScript*. `https://pytorch.org/docs/stable/jit.html`. Accessed: 2023-05-19. 2023.

[62] J. Reed and M. Suo. *Introduction to TorchScript*. `https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html`. Accessed: 2023-05-19. 2022.

[63] PyTorch. *Loading a TorchScript Model in C*. `https://pytorch.org/tutorials/advanced/cpp_export.html`. Accessed: 2023-05-19.

[64] ajaybati and S. Bryngelson. *`roseNNa`*. `https://github.com/comp-physics/roseNNa`. Accessed: 2023-05-21. 2021. DOI: `10.5281/zenodo.7334627`.

[65] D. Alexeev. *pytorch-fortran*. https://github.com/alexeedm/pytorch-fortran. Accessed: 2023-05-21. 2021.

[66] J. Ott. *FKB*. https://github.com/scientific-computing/FKB. Accessed: 2023-05-21. 2021.

[67] D. Alexeev. *neural-fortran*. https://github.com/modern-fortran/neural-fortran. Accessed: 2023-05-21. 2021.

[68] PyTorch. *ONNX*. https://pytorch.org/docs/stable/onnx.html. Accessed: 2023-05-21.

[69] J. Ott et al. *A Fortran-Keras Deep Learning Bridge for Scientific Computing*. 2020. arXiv: 2004.10652 [cs.LG].

[70] M. Curcic. "A parallel Fortran framework for neural networks and deep learning". In: *Acm sigplan fortran forum*. Vol. 38. 1. ACM New York, NY, USA. 2019, pp. 4–21.

[71] S. Partee et al. "Using machine learning at scale in numerical simulations with SmartSim: An application to ocean climate modeling". In: *Journal of Computational Science* 62 (2022), p. 101707.

[72] *A Server for Machine and Deep Learning Models - RedisAI*. https://oss.redis.com/redisai/intro/. Accessed: 23-05-2023.

[73] M. Kurz et al. "Deep reinforcement learning for computational fluid dynamics on HPC systems". In: *Journal of Computational Science* 65 (2022), p. 101884.

[74] J. M. Martí and E. Müller. "Numerical hydrodynamics in special relativity". In: *Living Reviews in Relativity* 6 (2003), pp. 1–100.

[75] J. Teunissen. *lookup_table_fortran: Linear lookup table implemented in modern Fortran*. https://github.com/jannisteunissen/lookup_table_fortran. Accessed: 2023-05-29.

[76] L. Del Zanna et al. "ECHO: a Eulerian conservative high-order scheme for general relativistic magnetohydrodynamics and magnetodynamics". In: *Astronomy & Astrophysics* 473.1 (2007), pp. 11–30.

[77] E. O'Connor. *GR1D: General Relativistic, Spherically Symmetry, Neutrino Transport Code for Stellar Collapse*. https://github.com/evanoconnor/GR1D. Accessed: 2023-06-03.

[78] *BLAS (Basic Linear Algebra Subprograms)*. https://netlib.org/blas/. Accessed: 2023-06-03.

[79] S. Cai et al. "Physics-informed neural networks (PINNs) for fluid mechanics: A review". In: *Acta Mechanica Sinica* 37.12 (2021), pp. 1727–1738.

[80] J. Zhou et al. "Graph neural networks: A review of methods and applications". In: *AI open* 1 (2020), pp. 57–81.

[81] H. R. Kang. "Three-Dimensional Lookup Table with Interpolation". In: *Computational Color Technology*. SPIE press, 2006, pp. 151–159.

Index