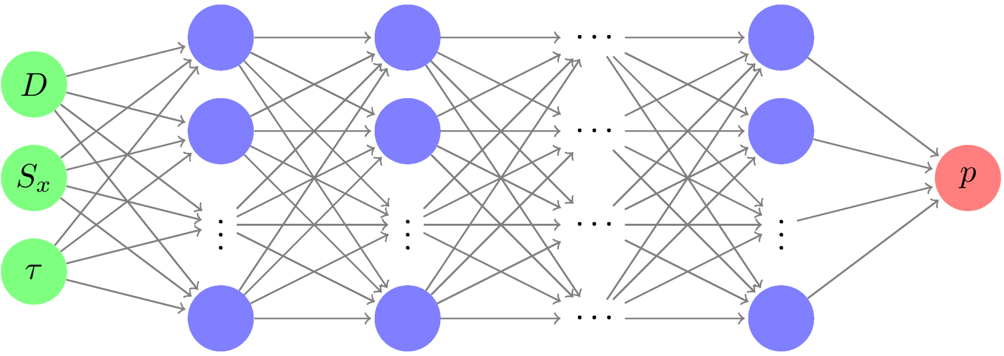
Improving the Performance of Conservative-to-Primitive Inversion in Relativistic Hydrodynamics Using Artificial Neural Networks

A Practical Demonstration of Computation Time Reduction in GRMHD simulations

K.S. Alders

Student number: 13196634

June 23, 2023



Thesis Bachelor Project Physics and Astronomy, size 15 EC

Supervisors: Dr. P. Moesta, S. Shankar
Second examiner: Prof. dr. S.B. Markoff

University of Amsterdam
Anton Pannekoek Institute for Astronomy

Abstract

?abstractname?

Contents

1 Introduction 1

2 Theoretical background 2

2.1 Numerical methods for evolution . . . 2

2.1.1 GraM-X evaluation times for evolution steps . . . 5

2.1.2 Current Implementation of Conservative-to-Primitive Inversion . . . 6

2.2 Relativistic fluid dynamics . . . 6

2.2.1 Special relativistic hydrodynamics . . . 7

2.2.2 Relation between conserved and primitive variables in SRHD . . . 8

2.2.3 General relativistic magnetohydrodynamics . . . 9

2.2.4 Relation between conserved and primitive variables in GRMHD . . . 11

2.3 Artificial neural networks theory . . . 12

2.3.1 Activation functions . . . 14

2.3.2 Loss functions . . . 16

2.3.3 Metrics . . . 17

3 Methodology 18

3.1 Research objective . . . 18

3.2 Neural Network architecture . . . 20

3.2.1 Architecture for the SRHD models . . . 20

3.2.2 Architecture for the GRMHD models . . . 21

3.3 Program structure and flow . . . 22

3.4 Data input to the neural network models . . . 26

3.4.1 Data Specifications . . . 26

3.4.2 Sampling of variables . . . 28

3.4.3 Validity of samples . . . 29

3.4.4 Equation of State . . . 29

3.4.5 Primitive-to-Conservative Transformation . . . 30

3.4.6 Generation of Labels . . . 30

3.4.7 Importing data . . . 30

3.5 Hyperparameter optimization . . . 31

3.5.1 Hyperparameter optimization for SRHD models . . . 31

3.5.2 Hyperparameter optimization for GRMHD models . . . 34

3.6 Training and Evaluation of Artificial Neural Network Models . 36

4 Results **37**
4.1 Models for SRHD . . . 37
4.1.1 Hyperparameters for the SRHD models . . . 38
4.1.2 Settings and runtime values . . . 39
4.2 Models for GRMHD . . . 44
4.2.1 Hyperparameters for NNGR1 and NNGR2 . . . 44
4.2.2 Settings and runtime values . . . 45
4.2.3 Comparison between the performance of the GRMHD models to that of the root-finders . . . 50

5 Discussions **52**
5.1 Discussion on the integration of the GRMHD models into GRaM-X . . . 52
5.2 Discussions on NN accuracy . . . 53
5.2.1 On the differences in metric errors with those of Dieselhorst et al. . . . 53
5.2.2 On the poor accuracy of the NNGR models . . . 54
5.3 Discussion on the performance of the NNs compared to the performance of the root finders . . . 56

6 Conclusion **57**

7 Recommendations **58**

A Details of training and evaluation in ANNs **63**
A.1 Training Process . . . 63
A.2 Evaluation Process . . . 66

B Subparameter search space for SRHD and GRMHD models **68**

C Installation procedure **69**

D Extended discussions **69**
D.1 Discussions on NN hyperparameters . . . 69
D.1.1 On the use of the ReLU output activation in the SRHD models . . . 69
D.1.2 On the unconventionality of using Sigmoid hidden activation in the SRHD models . . . 70
D.1.3 On why lower errors are obtained for the NNSR3 and NNSR4 models then for the NNSR1 and NNSR2 models 70
D.1.4 On the value of the `min_lr` subparameter for the SRHD models . . . 71

D.1.5 On the number of layers for the GRMHD models . . . 71
D.1.6 On the similarities between the hyperparameters of the GRMHD models 72
D.2 More discussions on NN accuracy 72
D.2.1 On the test errors being lower than the train errors . . 72
D.2.2 On the spikes that appear in the test error for the SRHD models 73
D.3 More discussions on neural network performance 73
D.3.1 On the increase in evaluation times between the SRHD and the GRMHD models 73
D.3.2 On the accuracy of the root-finding implementation of `Con2prim Interior` 75

List of Tables

1 Comparative runtime of GraM-X simulation evolution steps at different grid resolutions (n_{cells} values of 32, 64, 128, and 256). Each row represents a different evolution step, including the cumulative `GraM-X Iteration`. The table further highlights the computational bottlenecks, notably the `Fluxes` and `Con2prim Interior` steps. Data derived from [1]. 5

2 Breakdown of the data used in the ANN models: total samples, distribution into training, validation, and test sets, and the extent of hyperparameter optimization trials and training epochs. The variations in sample numbers and trials reflect the progressive complexity and the resource allocation in model development. 27

3 Intervals of uniform and log-uniform sampling for variables ANN models. For the SRHD models, rest-mass density (ρ), specific internal energy (ϵ), and velocity in the x-direction (v_x) are sampled. For the GRMHD models, in addition to these variables, velocities in the y and z-directions, magnetic field components, and the components of the metric tensor are sampled. Log-uniform sampling is used only for the specific internal energy (ϵ) in the GRMHD models. 28

4 Hyperparameter search spaces for different SRHD models. The table compares the ranges of hyperparameters considered during optimization for the NNSR3 model and the NNSR4 and NNSR5 models. The hyperparameters include architectural components such as number of hidden layers and units per layer, activation functions, and training settings such as loss function, optimizer, initial learning rate, batch size, number of epochs, and learning rate scheduler. 31

5 Hyperparameter search space for the GRMHD models with ranges and options for various aspects such as the number of layers, units per layer, activation functions, loss function, optimizer, initial learning rate, batch size, number of epochs, scheduler, and dropout rate. These parameters provide a broad starting point for optimization, subject to fine-tuning using Optuna's TPE sampler and pruning algorithms. 35

6 A comprehensive comparison of hyperparameters utilized in all for ANNs for con2prim in SRHD, including Dieselhorst et al.'s models NNC2PS, NNC2PL, and our newly developed models NNSR1-4. Hyperparameters including the number of hidden layers and units, activation functions for both hidden and output layers, loss functions, optimizers, learning rates, batch sizes, learning rate schedulers, and specific parameters for these schedulers are presented. 38

7 Summary of settings, performance metrics and evaluation times for all developed ANNs. The tabulated settings are the total, training, validation, and testing samples, the number of hyperparameter optimization trials, the number of epochs, the number of trainable parameters, the L_1 and L_∞ error rates and the average and best evaluation times. 40

8 Comparison of hyperparameters for the GRMHD models. The table details the number of hidden layers, number of hidden units per layer, the activation functions, loss function, optimizer, learning rate, batch size, learning rate scheduler, dropout rate, and specific parameters for the PReLU activation function and the learning rate schedulers for each model. 44

9 Comparison of average and best evaluation times of all developed models to the `Con2prim Interior` root-finding procedure as currently implemented in GRaM-X for different grid sizes (n_{cells}). The table entries depict fractions of `Con2prim Interior` times over respective model times. Higher values denote faster model performance. All fractions are rounded to three significant figures. Evaluation times for `Con2prim Interior` were obtained from Appendix C of [1]. 51

10 Subparameter search spaces for SRHD models. 68

11 Subparameters search space for GRMHD models 69

List of Figures

1 Flowchart depicting the sequence of operations in the numerical methods as used in [2]. The initial setup and subsequent evolution scheme, including steps from setting initial conditions to computing the net flux, are presented. The loop represents repeated evaluation of the RHS function in the Method of Lines. 3

2 Two-dimensional representation of a three-dimensional cell grid focusing on the central cell. The figure visualizes the process of defining primitive variables $\mathbf{P}$ at cell center, reconstructing them at cell faces ($\mathbf{P}_{\mathrm{rec}}$), and calculating the fluxes $\mathbf{F}$ across cell boundaries. 4

3 Plots of various activation functions used in the hyperparameter search spaces for the ANNs. The GELU function is approximated here as $0.5x(1 + \tanh(1.702x))$, its exact form is $0.5x(1 + \tanh\Big(\sqrt{2/\pi}(x + 0.044715x^3)\Big))$. 15

4 Illustration of the neural network architecture for the SRHD models. The network is a deep, feedforward, fully-connected structure. The input layer (green) receives three features: conserved density D, conserved momentum in the x-direction S_x, and energy density τ. An unspecified number of hidden layers (blue) interconnects all neurons from one layer to the next, passing through the activation functions (not shown). The output layer (red) consists of a single neuron, representing the primitive pressure p. The ellipsis between layers and neurons denotes an arbitrary number of hidden layers and neurons, respectively. These are part of the hyperparameters to be determined through optimization processes. 20

5 Illustration of the neural network architecture for the NNGR models. This architecture, compared to the NNSR, has expanded its input layer (green) to include 14 features, representing conserved density D, three-dimensional conserved momentum $\vec{S}$, energy density τ, three-dimensional conserved magnetic field $\vec{\mathscr{B}}$, and three-metric γ_{ij}. The hidden layers (blue), potentially larger in number and size due to the increased complexity of the GR case, feature dropout neurons (gray), which are randomly omitted during training to prevent overfitting. The output layer (red) consists of a single neuron, outputting the product of enthalpy and the Lorentz factor hW. The ellipsis denotes an arbitrary number of hidden layers, whose actual count and neuron count per layer are determined through optimization processes. 22

6 Flowchart illustrating the initial stages of the program flow for the con2prim ANNs in both SRHD and GRMHD. This includes initializing the Python script, loading data from CSV or generating new data, splitting the data into different sets, and creating the neural network model. 23

7 Flowchart illustrating the latter stages of the program flow. This includes possible hyperparameter tuning, combining the training and validation sets, training and evaluating the model, and porting the model to a C++ environment. 24

8 Schematic diagram depicting the iterative process of training and evaluation in a machine learning model. The cyclic flow signifies the repeated adjustment of model parameters for N epochs, which is aimed at minimizing a defined loss function and evaluating the model's performance on unseen data. . . . 37

9 L_1 and L_∞ norm error plots per epoch for the NNSR1 model during training (blue) and testing (red) for 400 epochs. The L_1 and L_∞ norm errors decrease over the epochs, where from about 200 epochs onward they stay roughly constant. 41

10 L_1 and L_∞ norm error plots per epoch for the NNSR2 model during training (blue) and testing (red) for 400 epochs. As with the NNSR1 model, the L_1 and L_∞ norm errors decrease over the epochs, where from about 200 epochs onward they stay roughly constant. 42

11 L_1 and L_∞ norm error plots per epoch for the NNSR3 model during training (blue) and testing (red) for 400 epochs. Like with the NNSR1 and NNSR2 models, the L_1 and L_∞ norm errors decrease over the epochs, where from about 200 epochs onward they stay roughly constant. 42

12 L_1 and L_∞ norm error plots per epoch for the NNSR4 model during training (blue) and testing (red) for 400 epochs. Like with the NNSR1–3 models, the L_1 and L_∞ norm errors decrease over the epochs, where from about 200 epochs onward they stay roughtly constant. The curve of the NNSR4 model looks different due to the use of the Cosine Annealing learning rate scheduler as opposed to the ReduceLROnPlateau scheduler used by NNSR1–3 43

13 Training and testing error metrics for the NNGR1 model. The left plot illustrates the L_1 norm errors, and the right plot illustrates the L_∞ norm errors. Both errors are plotted as a function of the number of epochs trained, ranging from 0 to 500. 46

14 Training and testing error metrics for the NNGR2 model. The left plot illustrates the L_1 norm errors, and the right plot illustrates the L_∞ norm errors. Both errors are plotted as a function of the number of epochs trained, ranging from 0 to 500. 47

15 Optimization history plots for the GRMHD models. The left plot corresponds to NNGR1 and the right to NNGR2. Blue dots represent the trial-wise objective values, while the orange line marks the best objective value obtained over the course of the trials. 48

16 Relative importance of different hyperparameters for the NNGR1 model, represented by their objective value fractions. Hyperparameters include the optimizer, number of hidden layers, dropout rate, scheduler, hidden activation function, batch size, number of epochs, loss function, size of the first hidden layer, learning rate, and the output activation function. 49

17 Relative importance of different hyperparameters for the NNGR2 model, represented by their objective value fractions. Hyperparameters include the optimizer, batch size, hidden activation function, number of epochs, size of the first hidden layer, learning rate, dropout rate, scheduler, loss function, and the output activation function. 50

18 Detailed flowchart of the training and evaluation processes of the ANNs. The solid lines illustrate mandatory steps such as forward propagation, loss computation, backward propagation, parameter updates, and network evaluation for the training process, and forward propagation, loss computation, and metrics calculation for the evaluation process. The dashed lines represent optional steps like scheduler updates and pruning for hyperparameter optimization. Each process is iterated over each batch of data. 64

1 Introduction

General Relativistic Magneto-Hydrodynamics (GRMHD) simulations are indispensable in the study of astrophysical phenomena such as core-collapse supernovae and binary neutron star mergers. Over the past decade, these simulations have been employed to make predictions of gravitational waves resulting from compact-object mergers, to determine the amount and composition of ejected materials during these events, and to identify the remnants that are left behind [2]. Such simulations demand complex calculations, requiring both speed and accuracy to deliver meaningful results. GRaM-X [2], a state-of-the-art code that supersedes the older GRHydro [3], is used for executing these simulations.

Built on the Cactus computational framework [4], an environment for high-performance computing for numerical simulations, and part of the Einstein Toolkit [5], a collection of software components and tools for simulating and analyzing astrophysical phenomena in the context of general relativity, GRaM-X incorporates enhancements such as the utilization of Graphics Processing Units (GPUs) to markedly boost the computational efficiency of the simulations.

The process of these simulations begins with a specific configuration, such as an iron core approximately 3000 km in diameter for a core-collapse supernova. The simulation then progresses, capturing the evolution of the system over time, including critical features such as the formation of jets.

The part of the code that is of our interest is its grid-based magnetohydrodynamics. The evolution of the system in grid-based MHD involves a set of variables, which can be classified as primitive or conserved variables. Primitive variables, such as density ρ, velocity $\mathbf{v}$, energy density ϵ, pressure p, and the magnetic field $\mathbf{B}$, are intuitive and directly related to the state of the system. The conserved variables on the other hand include D for conserved density, S^i for conserved momentum, τ for conserved energy density, and $\mathscr{B}^k$ for the magnetic field. These variables are linked through the Valencia formulation [6], which allows for the evolution of the simulated system through these conserved variables.

Our project focuses on one specific aspect of the GRam-X code: the conversion from conserved variables to primitive variables. This step, although crucial, is non-trivial and computationally expensive, contributing significantly to the total computation time. It currently employs Newton-Rhapson's method in 3D, which, while effective, can encounter local relative minima and hence lead to errors.

To overcome these limitations, our aim is to implement an artificial neural network that can carry out the conservative-to-primitive inversion more efficiently. This approach has the potential to reduce computation time, lower the chances of numerical errors, and, ultimately, improve the accuracy of the simulations.

2 Theoretical background

2.1 Numerical methods for evolution

In this section, we provide a summary of the numerical methods as used in [2] to evolve a simulated astrophysical system. This section will provide the necessary background to understand the context in which the conservative-to-primitive inversion, the focus of our project, takes part in.

The Valencia formulation considers eight coupled hyperbolic partial differential equations for an eight-element state vector, $\mathbf{U}$ consisting of the conserved variables. To calculate the fluxes, $\mathbf{F}$, primitive variables are needed, which are represented by the vector $\mathbf{P}$. Figure 2 provides a visual representation of this concept, where $\mathbf{P}$ is initially known at the cell center.

The numerical methods used in the Valencia formulation follow a sequence of steps, which are depicted in Figure 1. The flowchart illustrates the initial setup and the subsequent evolution scheme.

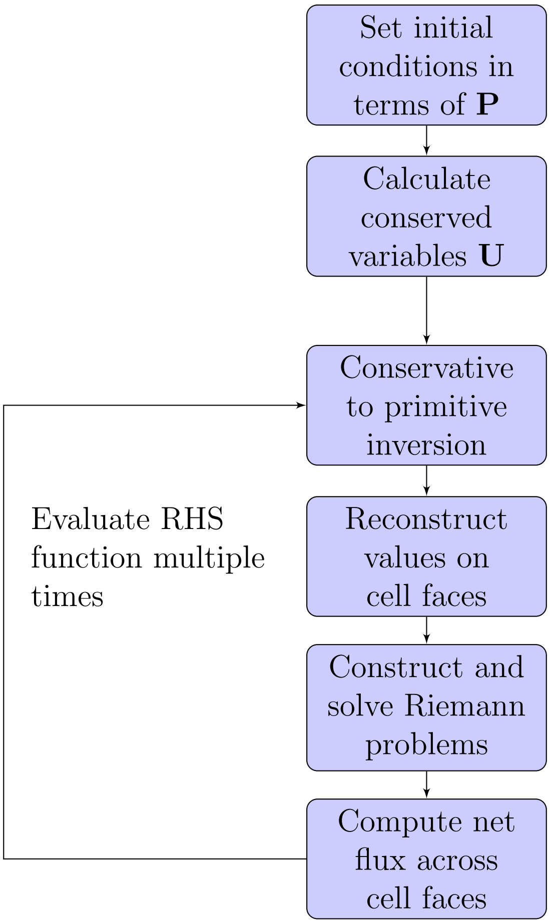

Set initial conditions in terms of $\mathbf{P}$
Calculate conserved variables $\mathbf{U}$
Conservative to primitive inversion
Reconstruct values on cell faces
Construct and solve Riemann problems
Compute net flux across cell faces
Evaluate RHS function multiple times

Figure 1: Flowchart depicting the sequence of operations in the numerical methods as used in [2]. The initial setup and subsequent evolution scheme, including steps from setting initial conditions to computing the net flux, are presented. The loop represents repeated evaluation of the RHS function in the Method of Lines.

The initial setup involves two steps. Firstly, the initial conditions are established in terms of the primitive variables, $\mathbf{P}$. As depicted in Figure 2, these primitive variables, which include quantities like the fluid density, velocities, internal energy, and magnetic fields, are known at the cell centers. This corresponds to the topmost block in Figure 1. Next, the conserved variables $\mathbf{U}$ are computed from these initial conditions. This step is known as the primitive-to-conservative step and is straightforward, since $\mathbf{U}\left(\mathbf{P}\right)$ is closed, as we shall see mathematically in Section 2.2.2 and Section 2.2.4. The evolution of $\mathbf{U}$ is governed by hyperbolic partial differential equations, which naturally arise when describing wave propagation and transport phenomena [7].

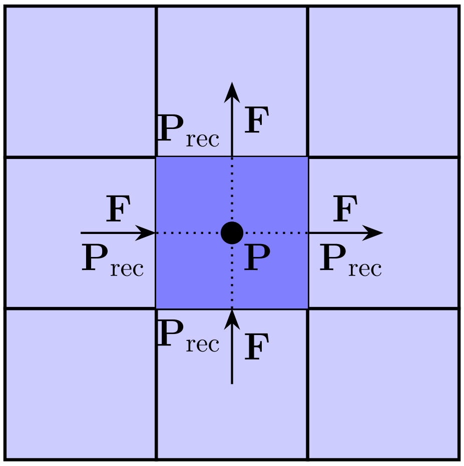

$\mathbf{P}_{\text{rec}}$
$\mathbf{F}$
$\mathbf{F}$
$\mathbf{P}_{\text{rec}}$
$\mathbf{P}$
$\mathbf{F}$
$\mathbf{P}_{\text{rec}}$
$\mathbf{P}_{\text{rec}}$
$\mathbf{F}$

Figure 2: Two-dimensional representation of a three-dimensional cell grid focusing on the central cell. The figure visualizes the process of defining primitive variables $\mathbf{P}$ at cell center, reconstructing them at cell faces ($\mathbf{P}_{\text{rec}}$), and calculating the fluxes $\mathbf{F}$ across cell boundaries.

The evolution scheme is based on the Method of Lines [8]. The Method of Lines involves a loop that iteratively computes the right-hand side (RHS) function to evaluate the time evolution of the state vector $\mathbf{U}$. In Figure 1, this loop starts from "Conservative to primitive inversion" and circles back to it. The RHS function, which is evaluated within this loop, consists of spatial derivatives representing the change in conserved variables due to fluxes across cell faces. It is important to note here that the change in the conserved quantities within a cell over time is driven by these fluxes, making the computation of fluxes across cell boundaries crucial for the system's evolution.

Within this loop, the first step is to determine the primitive variable vector $\mathbf{P}$ from the conserved variable vector $\mathbf{U}$. This is a challenging step as it entails solving a multi-dimensional nonlinear equation for each grid cell. It also involves evaluating the equation of state, possibly by interpolating in a nuclear equation of state table.

The second step within the loop, illustrated in Figure 2, is the reconstruction of the primitive variables, denoted as $\mathbf{P}_{\text{rec}}$, on the cell faces. This is a crucial step where the primitive variables known at the cell centers are used to compute the corresponding quantities at the cell faces. This reconstruction at cell faces enables the computation of fluxes across the cell boundaries.The spacetime metric, which is known at cell vertices, is also linearly interpolated to cell faces in this step.

Subsequently, Riemann problems are constructed and solved using a Harten-Lax-van Leer-Einfeldt (HLLE) solver [9, 10].The Riemann solver computes the numerical fluxes at the cell interfaces.

Finally, the solutions of the Riemann problems are used to compute the net flux across all the cell faces, which is crucial for defining the flux term of the RHS. In Figure 2, the computed fluxes across the cell faces are represented by the arrows labeled **F**. This completes one iteration of the Method of Lines.

The Method of Lines loop iterates multiple times, as indicated by the label "Evaluate RHS function multiple times" in Figure 1, allowing the evolution of the system from time t to $t + \Delta t$.

2.1.1 GraM-X evaluation times for evolution steps

Table 1 presents the runtime of various evolution steps at different grid resolutions, i.e., for n_{cells} values of 32, 64, 128, and 256.

Table 1: Comparative runtime of GraM-X simulation evolution steps at different grid resolutions (n_{cells} values of 32, 64, 128, and 256). Each row represents a different evolution step, including the cumulative `GraM-X Iteration`. The table further highlights the computational bottlenecks, notably the `Fluxes` and `Con2prim Interior` steps. Data derived from [1].

Evolution step	$n_{\text{cells}} = 32$ (ms)	$n_{\text{cells}} = 64$ (ms)	$n_{\text{cells}} = 128$ (ms)	$n_{\text{cells}} = 256$ (ms)
Con2prim Interior	0.497	2.0533	8.803143	36.224613
Fluxes	1.559	6.975	54.303	434.776
Source	0.172	0.742	5.115	40.165
Update	0.081	0.299	2.132	16.957
Tmunu	0.238	1.138	8.231	65.409
GraM-X Iteration	36.7	157.4	1008.4	10702.6

The rows of this table align with the evolution steps discussed in Section 2.1. The `Con2prim Interior` step represents the conservative-to-primitive inversion. This step's specifics, particularly within the context of the GraM-X implementation, will be elaborated on in Section 2.2.4. The `Fluxes` and `Source` rows correspond to the calculation of fluxes and source terms, respectively. The mathematical expressions for these calculations are presented in Equations (26) and (27), both of which will be reviewed in depth in Section2.2.4. The `Update` step involves solving the Riemann problem, while `Tmunu` corresponds to the calculation of the energy-momentum tensor, the equation for which is given by Equation (18). Finally, the `GraM-X Iteration` row represents a complete iteration of the steps mentioned above. This is a cumulative step that inherently incorporates the runtime of all the previous individual steps. As can be observed from the table, the calculation of fluxes `Fluxes` consumes the majority of runtime in every grid configuration, marking it as a bottleneck in these simulations in terms of runtime. However, as these fluxes are calculated from first principles (as shown in Equation (26)), there is little room for optimization in this part of the process. On the other hand, the conservative-to-primitive inversion (`Con2prim Interior`) is the second major bottleneck, and does present an opportunity for performance enhancement, and this potential improvement is the primary focus of our project.

2.1.2 Current Implementation of Conservative-to-Primitive Inversion

Theoretically, the current implementation of conservative-to-primitive inversion in GRaM-X is a five-dimensional root-finding problem; however, the inversion of magnetic field components is straightforward, since they differ by a numerical factor only, reducing the problem's dimensionality to three [2].

GRaM-X incorporates two methods for the conservative-to-primitive conversion: the 3D Newton-Raphson method (3D-NR) [11] and Newman & Hamlin's method (Newman's method) [12]. In the 3D-NR method, which is a widely known iterative technique for finding the roots or zeros of a real-valued function [11], the system of equations is restructured into three equations with three unknowns. In contrast, Newman's method is essentially a one-dimensional method that iteratively solves for fluid pressure. This method reduces the system to a cubic polynomial [12].

In practice, 3D-NR is employed as the primary method for the conservative-to-primitive conversion within GRaM-X. If 3D-NR fails to converge, Newman's method is used as a fallback. If both methods fail to converge, a bisection method in temperature is employed as it is assured to converge, albeit at a higher computational cost.

We will next take a deeper look at the mathematics of the conservative-to-primitive conversion step, for which we first need to briefly look at fluid dynamics in the relativistic regime.

2.2 Relativistic fluid dynamics

In this section, we go over the mathematics involved in conservative to primitive conversion. We consider first the special relativistic case and then the general relativistic case.

2.2.1 Special relativistic hydrodynamics

The Valencia formulation [6] is a widely used formulation [13] in which the relativistic conservation laws of hydrodynamics are stated in the form

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}^i}{\partial x^i} = \mathbf{S} \ , \tag{1}$$

with $i = 1, 2, 3$. Here, the state vector $\mathbf{U}$ is defined as

$$\mathbf{U} = (D, S^1, S^2, S^3, \tau) \ ; \tag{2}$$

it consists of the conserved quantities rest-mass density D, the three components of momentum S^i and the energy density relative to the mass energy density τ; all measured in the laboratory frame. $\mathbf{F}^i$ are flux vectors given by

$$\mathbf{F}^i = \left(Dv^i, S^1v^i + p\delta^{1i}, S^2v^i + p\delta^{2i}, S^3v^i + p\delta^{3i}, S^i - Dv^i\right) \ , \tag{3}$$

where p is the primitive pressure of the fluid, v^i is the primitive 3-velocity of the fluid and δ^{ij} is the Kronecker delta.The flux vectors describe how the conserved quantities are transported across a given surface in the fluid. $\partial\mathbf{U}/\partial t$ gives the time rate of change of the conserved quantities, and $\partial\mathbf{F}^i/\partial x^i$ gives the spatial divergence of the flux vectors. If the latter term is positive, it means that there is a net outflow of the conserved quantity from a small volume, leading to a decrease in the quantity within the volume. Conversely, if this term is negative, it means that there is a net inflow of the quantity, leading to an increase within the volume. $\mathbf{S}$ on the RHS of (1) represents a source term, by which we mean any source of e.g. matter, energy and momentum not already accounted for by the terms on the LHS of (1). In the astrophysical context of fluid dynamics, we can think of the sources as the gravitational pull of an astrophysical object, nuclear reactions inside the core, chemical changes in the composition, etc. The meaning of equation (1) as a whole is readily conveyed when solved for the time rate of change of the conserved quantity; we then see that the equation states that the change in the conserved quantity in a small volume of fluid is equal to the net flux of the quantity across the boundary of the volume plus any sources or sinks of the quantity within the volume. This type of formulation of conservation laws is fundamental in physics, and the reader can compare it to e.g. the divergence theorem in multi-variable calculus or to its application in electrodynamics, Gauss's law.

Equation (1) is derived from the formulation of the conservation laws of a relativistic fluid in tensor notation

$$\nabla_\mu J^\mu = 0 \ , \tag{4}$$

$$\nabla_\mu T^{\mu\nu} = 0 \ , \tag{5}$$

, where $(\mu, \nu = 0, 1, 2, 3)$, J^μ the 4-current $J^\mu \equiv \rho u^\mu$, ∇_μ denotes the covariant derivative with respect to coordinate x^μ and with the Einstein summation convention implied for repeated indices. Equation (4) is known as the continuity equation and equation (5) is the conservation of energy-momentum. Together these give five conservation laws in SRHD. ρ gives the primitive proper rest-mass density of the fluid and $T^{\mu\nu}$ is the energy-momentum tensor. In astrophysical context, the fluid is often taken to be a perfect fluid, and in that case the energy-momentum tensor is given by

$$T^{\mu\nu} = \rho h u^\mu u^\nu + p g^{\mu\nu} \ , \tag{6}$$

with $g^{\mu\nu}$ the metric tensor, p the primitive pressure as before, u^μ the fluid's 4-velocity and h the specific enthalpy given by

$$h = 1 + \epsilon + \frac{p}{\rho} \tag{7}$$

, where ϵ is the primitive specific internal energy. A detailed derivation of (1) requires a mathematical background in general relativity and tensor calculus, and is out of the scope of this project; however, the interested reader is referred to [6].

2.2.2 Relation between conserved and primitive variables in SRHD

The conservative variables (2) are related to the primitive variables that we encountered in Section 2.2.1, i.e., gathered into one vector, they are related to

$$\mathbf{P} = \left(\rho, v^i, \epsilon, p\right) \tag{8}$$

through the equations [14]

$$\begin{aligned} D &= \rho W \ , && (9) \\ S^i &= \rho h W^2 v^i \quad i = 1, 2, 3 \ , && (10) \\ \tau &= \rho h W^2 - p - D \ . && (11) \end{aligned}$$

with h the specific enthalpy as defined in (7) and W the Lorentz factor between two reference frames

$$W = \frac{1}{\sqrt{1 - v^i v_i}} \ . \tag{12}$$

These equations are directly used in the code to generate the input to the neural networks.

Given an equation of state (EoS) assumed to be of the form

$$p = p(\rho, \epsilon) \ , \tag{13}$$

the system of equations (1) is closed. However, the inversion $\mathbf{P}(\mathbf{U})$, which is what we refer to as con2prim, cannot be written in closed form, and numerical approximation is necessary. The reason that we need to do conservative-to-primtive inversion is as follows. Where the conservative variables are the quantities that are natural to use to evolve the system, e.g. two merging neutron stars or a supernova explosion, by means of equation (1), the primitive variables are needed to apply boundary conditions to the system, to compute the source terms for the next time step and to have an intuitive understanding of the evolution of the system.

Equation (9) to (11) with W defined as in (12) and h defined as in (7) are used directly in first part of our project for the calculation of the input to the NN for the case of SRHD. The primitive variables ρ, v^i and ϵ are sampled on appropriate intervals, and an analytic EOS is used for pressure, the latter which coincidentally are also used as the labels for supervised learning of the NNs. These topics are quantitatively discussed in Section 3.

2.2.3 General relativistic magnetohydrodynamics

In our project we set up two types of artificial neural network models, models for SRHD and models for GRMHD. Equation (9) is directly used in the NNs for SRHD. It is now of interest to us to see how these equations change and how the formulation altogether changes in the case of GRMHD.

In ideal GRMHD, the conservation laws (4) and (5) still hold, but four additional equations are added for the conservation of the magnetic flux and an extra term is added to $T^{\mu\nu}$ due to the inclusion of the magnetic field. The equations for the conservation of magnetic flux are known as the source-free Maxwell's equations and are given by

$$\nabla_v {}^*F^{\mu v} = 0 \ , \tag{14}$$

with ${}^*F^{\mu v}$ the Hodge dual of the Faraday tensor $F^{\mu\nu}$, the latter also known as the electromagnetic field tensor.

Equation 6 gives the energy-momentum tensor in the case of SRHD. The inclusion of the magnetic field in GRMHD has as a result that the energy-momentum tensor now consists of a hydrodynamic component, which we label $T^{\mu\nu}_{\mathrm{H}}$ and which is coincident with equation (6), as well as an electromagnetic contribution, labeled $T^{\mu\nu}_{\mathrm{EM}}$ [2]

$$T^{\mu\nu}_{\mathrm{H}} := \rho h u^\mu u^\nu + p g^{\mu\nu} \ , \tag{15}$$

$$T^{\mu\nu}_{\mathrm{EM}} := F^{\mu\lambda} F^{\nu}_{\lambda\,\lambda} - \frac{1}{4} g^{\mu\nu} F^{\lambda\kappa} F_{\lambda\kappa} = b^2 u^\mu u^\nu - b^\mu b^\nu + \frac{b^2}{2} g^{\mu\nu} \ ; \tag{16}$$

here b^μ is the magnetic 4-vector defined as

$$b^\mu = u_\nu {}^* F^{\mu\nu} \ ; \tag{17}$$

it represents the magnetic field as seen in the rest frame of the fluid. The full energy-momentum tensor is therefore given by

$$T^{\mu\nu} = \left(\rho + \rho\epsilon + p + b^2\right) u^\mu u^\nu + \left(p + \frac{b^2}{2}\right) g^{\mu\nu} - b^\mu b^\nu \ , \tag{18}$$

or defining new quantities

$$p^* := p + \frac{b^2}{2} \tag{19}$$

and

$$h^* := 1 + \epsilon + \frac{(p + b^2)}{\rho} \tag{20}$$

it is equal to

$$T^{\mu\nu} = \rho h^* u^\mu u^v + p^* g^{\mu v} - b^\mu b^v \ . \tag{21}$$

We highlighted the definitions (19) and (20) because they will be used in the equations for conservative-to-primitive inversion as shown in Section **??**.

It is noteworthy that, although the form of the equations of the five conservation laws (4) and (5) stays the same in the generalization from SRHD to GRMHD, the underlying physics changes as the metric $g^{\mu\nu}$ is no longer necessarily flat due to the influence of gravity. The full set of nine conservation laws in GRMHD is given by

$$\nabla_\mu J^\mu = 0, \quad \nabla_\mu T^{\mu v} = 0, \quad \nabla_v {}^* F^{\mu v} = 0 \ . \tag{22}$$

Likewise the MHD evolution equations (1)

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}^i}{\partial x^i} = \mathbf{S} \ , \qquad [1]$$

still hold in form, but the underlying physics is now different due to the influence of gravity and the inclusion of the magnetic field. We discuss how the relation between the conservative and primitive variables has changed in Section 2.2.4.

2.2.4 Relation between conserved and primitive variables in GRMHD

In GRMHD, the vector of conserved variables is redefined to be

$$\mathbf{U} = \left(D, S_j, \tau, \mathscr{B}^k\right) \tag{23}$$

i.e. it consists of conserved density, momentum, and energy density as before, but now it includes the conserved magnetic field, defined to be

$$\mathscr{B}^k = \sqrt{\gamma} B^k \ , \tag{24}$$

where B^i is the spatial magnetic field and γ is the determinant of the three-metric γ_{ij},

$$\gamma = \det \gamma_{ij} = \gamma_{xx}\gamma_{yy}\gamma_{zz} + 2\gamma_{xy}\gamma_{xz}\gamma_{yz} - \gamma_{xx}\gamma_{yz}^2 - \gamma_{yy}\gamma_{xz}^2 - \gamma_{zz}\gamma_{xy}^2 \ . \tag{25}$$

γ_{ij} is one of the standard Arnowitt-Deser-Misner (ADM) variables. The ADM formulation is a formulation of general relativity that is suitable for numerical solutions. We will encounter two other ADM variables, the lapse α and the shift β, shortly.

The expressions of the flux vectors $\mathbf{F}^i$ and the source terms $\mathbf{S}$ in GRMHD are respectively

$$\mathbf{F}^i = \alpha \times \begin{pmatrix} D\tilde{v}^i \\ S_j\tilde{v}^i + \sqrt{\gamma}p^*\delta^i_j - b_j\mathscr{B}^i/W \\ \tau\tilde{v}^i + \sqrt{\gamma}p^*v^i - \alpha b^0\mathscr{B}^i/W \\ \mathscr{B}^k\tilde{v}^i - \mathscr{B}^i\tilde{v}^k \end{pmatrix}, \tag{26}$$

$$\mathbf{S} = \alpha\sqrt{\gamma} \times \begin{pmatrix} 0 \\ T^{\mu v}\left(\frac{\partial g_{vj}}{\partial x^\mu} - \Gamma^\lambda_{\mu\nu}g_{\lambda j}\right) \\ \alpha\left(T^{\mu 0}\frac{\partial \ln\alpha}{\partial x^\mu} - T^{\mu v}\Gamma^0_{\mu v}\right) \\ \vec{0} \end{pmatrix}, \tag{27}$$

with α the ADM lapse variable, $\tilde{v}^i = v^i - \beta^i/\alpha$, β the ADM shift variable, δ^i_j the Kronecker delta, $\Gamma^\lambda_{\mu\nu}$, the Christoffel symbols, variables in general relativity that provide a measure of how much the basis vectors change as you move along a curve in a manifold, and with all other quantities as defined in this section.

The equations (24) add three additional equations to the relations (9) to (11) that we already had. These previous equations are also modified to include the effects of gravity and the magnetic field, and the full set of conservative-to-primitive inversion equations in GRMHD is given by

$$
\begin{aligned}
D &= \sqrt{\gamma}\rho W \ , && (28)\\
S_j &= \sqrt{\gamma}\left(\rho h^* W^2 v_j - \alpha b^0 b_j\right) \ , && (29)\\
\tau &= \sqrt{\gamma}\left(\rho h^* W^2 - p^* - (\alpha b^0)^2\right) - D \ , && (30)\\
\mathscr{B}^k &= \sqrt{\gamma} B^k \ , && [24]
\end{aligned}
$$

In the data generation code, the variables $\rho, \epsilon, v^x, v^y, v^z, B^x, B^y, B^z$ and all the unique components of γ_{ij}, i.e. $\gamma_{xx}, \gamma_{xy}, \gamma_{xz}, \gamma_{yy}, \gamma_{yz}, \gamma_{zz}$. The appropriate sampling intervals are discussed in Section 3.

These topics cover the theoretical background on fluid dynamics that is important for understanding the context in which the calculations of Con2Prim are performed. Many excellent resources exist on fluid dynamics, and the reader who would like a deeper dive into the theory can consider e.g. [15, 16, 17] among many others.

2.3 Artificial neural networks theory

In this section, we provide the relevant background of machine learning in the context of artificial neural networks.

Artificial Neural Networks (ANNs) have emerged as a fundamental cornerstone of Machine Learning and, more broadly, Artificial Intelligence (AI) [18]. Stemming from an effort to replicate the information processing capabilities of biological neural networks, ANNs aim to capture and model complex patterns within data [19]. ANNs' ability to learn from data makes them particularly adept at handling a diverse range of tasks, from classification to regression problems, as in the case of our Special Relativistic Hydrodynamics (SRHD) and General Relativistic Magnetohydrodynamics (GRMHD) neural network (NN) models.

The overarching field that ANNs belong to is known as Machine Learning (ML), which can be thought of as a subset of AI where computers learn to perform tasks without being explicitly programmed [20].

Deep Learning (DL), a subfield of ML, involves the construction and training of neural networks with multiple layers, or "depth". These layers allow the network to learn more abstract features from the input data. ANNs with multiple layers are known as deep neural networks [18].

A typical ANN architecture is composed of interconnected layers of artificial neurons or nodes. Each layer consists of multiple nodes, which perform a weighted sum of their inputs, followed by the application of an activation function. These activation functions, such as the sigmoid or Rectified Linear

Unit (ReLU) functions, introduce non-linearity into the model, enabling it to learn and represent more complex relationships in the data. A more detailed discussion of activation functions as we consider them in the hyperparameter search space used in our work can be found in Section 2.3.1.

The layers between the input and output layers are called hidden layers. The number of hidden layers and the number of neurons in these layers are examples of what we call hyperparameters. These are variables that define the network structure and dictate how the network is trained .

The process of hyperparameter optimization involves searching for the set of hyperparameters that gives the best performance on the validation set. This search is often conducted within a predefined hyperparameter search space, which denotes the range or set of values each hyperparameter can take. This search space can be defined based on prior knowledge or through empirical testing.For a more detailed description of hyperparameters and a comprehensive listing of the hyperparameter search space used in our models, we refer the reader to Section 3.5.1 and Section 3.5.2.

At its core, the learning process within an ANN is fundamentally an optimization problem, with the aim to minimize a predefined loss function. The specifics of the loss functions used in our study are also detaile in Section 3.5.1 and Section 3.5.2. This optimization is often accomplished through a technique known as Stochastic Gradient Descent (SGD), or through one of its variants. These algorithms iteratively adjust the model's parameters in the direction that reduces the loss function the most [21].

Among various learning paradigms, we focus on supervised learning, where the system learns a function mapping inputs to outputs based on example input-output pairs [22].

Training an ANN involves two key processes: forward propagation and backpropagation. During forward propagation, the input data is passed through the network, and an output prediction is generated [23]. Then, in backpropagation, the network's error, as determined by the loss function, is propagated backwards, and the network's weights are updated according to the calculated gradients [23].

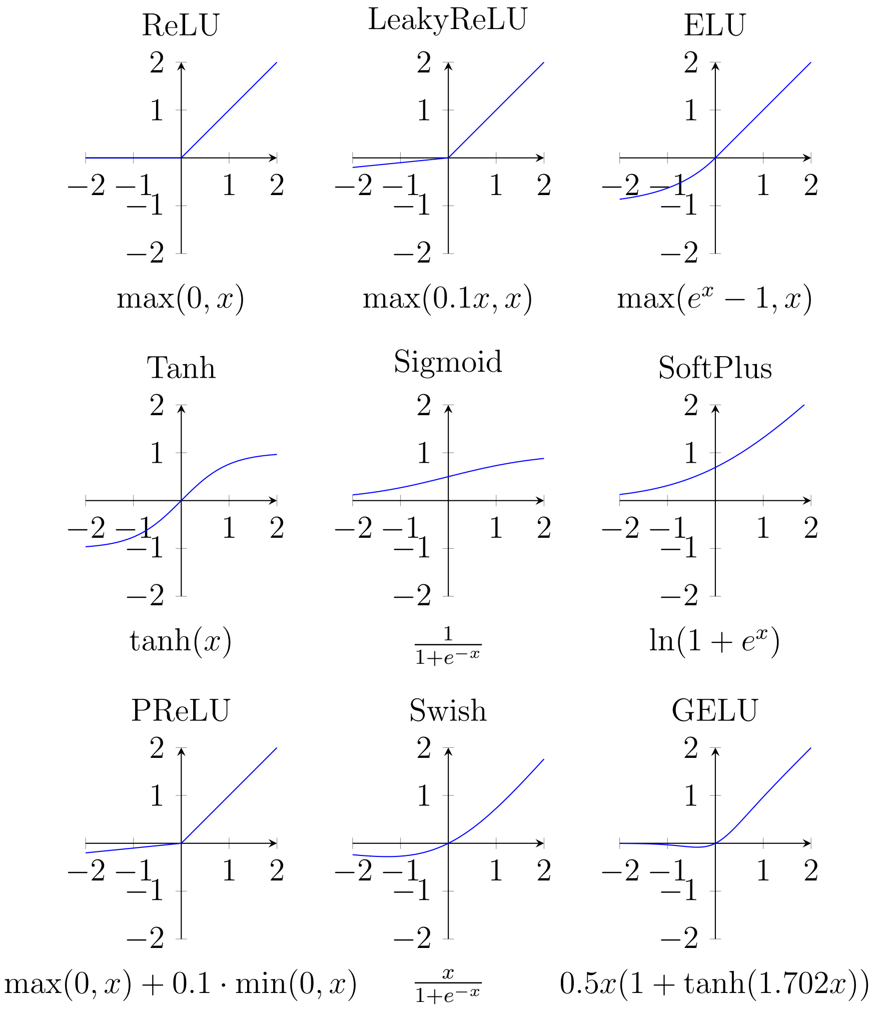
To assess the performance of ANN models, datasets are split into training and test sets, as well as usually into an additional set called the validation set. The training set is used for learning the model parameters, while the test set provides an unbiased evaluation of the final model. The validation set aids in hyperparameter tuning and early stopping to prevent overfitting. Concretely, the performance of ANNs is evaluated using metrics, which provide a quantitative measure of the model's predictive capabilities.

In summary, ANNs provide a robust method for learning complex patterns within data. ANNs are powerful tools capable of modelling complex nonlinear relationships, which makes them an excellent choice for tackling the challenge of conservative-to-primitive inversion in special relativistic hydrodynamics and general relativistic magnetohydrodynamics. The journey from initializing a network to evaluating its performance encompasses several important steps and considerations, each contributing to the success of the modeling process. In the subsequent sections, we delve into more details regarding the activation functions, loss functions, and metrics used in our study.

2.3.1 Activation functions

In the domain of machine learning, particularly in neural networks, activation functions play a pivotal role. These functions introduce non-linearity into the model, enabling it to learn and model complex patterns in data that a linear model cannot. In a feedforward neural network, an activation function takes the output from a neuron (or a layer of neurons), which is a weighted sum of its input, and transforms it into a non-linear form. This output is then passed onto the next layer in the network. Without activation functions, neural networks would be limited to linear transformations of the input data. This restriction would make them similar to single-layer perceptrons, a type of linear classifier that forms the building block of more complex networks but can only separate data that is linearly separable (like points that can be separated by a straight line in two-dimensional space). With activation functions, networks can learn from more complex, non-linear data and solve more intricate tasks.

Various activation functions are utilized in the field, each having their own characteristics and are suitable for different kinds of tasks. We illustrate some of these commonly used activation functions in Figure 2.3.1; these are the activation functions that we use in the hyperparameter search spaces of our models as also discussed in Section 3.5.1 and Section 3.5.2.


ReLU

$\max(0, x)$

LeakyReLU

$\max(0.1x, x)$

ELU

$\max(e^x - 1, x)$

Tanh

$\tanh(x)$

Sigmoid

$\frac{1}{1+e^{-x}}$

SoftPlus

$\ln(1 + e^x)$

PReLU

$\max(0, x) + 0.1 \cdot \min(0, x)$

Swish

$\frac{x}{1+e^{-x}}$

GELU

$0.5x(1 + \tanh(1.702x))$

Figure 3: Plots of various activation functions used in the hyperparameter search spaces for the ANNs. The GELU function is approximated here as $0.5x(1 + \tanh(1.702x))$, its exact form is $0.5x(1 + \tanh\left(\sqrt{2/\pi}(x + 0.044715x^3)\right))$.

ReLU, or Rectified Linear Unit, represented as $\max(0, x)$, is a widely employed activation function in deep learning models due to its simplicity and efficiency [24]. However, it has a disadvantage known as the dying ReLU problem where neurons can sometimes get stuck in the negative phase and cease to learn.

LeakyReLU functions were proposed as a modification of ReLU that attempt to solve the dying ReLU problem by allowing small negative values when $x < 0$, with the slope being a hyperparameter [25]. PReLU, or Parametric ReLU, is a further refinement where the slope for $x < 0$ is not a fixed hyperparameter, but a parameter that can be learned by the model [26].

The ELU, or Exponential Linear Unit, function is another variation of ReLU that mitigates the dying ReLU problem by having a non-zero gradient for $x < 0$ [27].

The Tanh and Sigmoid functions were traditionally used in the earlier days of neural networks. While Sigmoid outputs a value between 0 and 1 and is particularly useful in the output layer of binary classification problems, Tanh outputs a value between -1 and 1, leading to zero-centered outputs which can help the model converge faster.

The SoftPlus function [28], as a smooth approximation of ReLU, offers benefits in optimization problems, where its everywhere-differentiability can provide more stable convergence during training and mitigate the 'dying ReLU' problem by maintaining non-zero gradients.

The SoftPlus function is a smooth approximation of ReLU and can be beneficial in certain contexts. These include when we are dealing with optimization problems where the derivative of the activation function plays a critical role. In such cases, the non-differentiability of the ReLU at zero can cause issues. The SoftPlus function, being smooth and differentiable everywhere, does not have this problem. This can result in more stable convergence during the training process of the neural network [28].

Swish [29] and GELU (Gaussian Error Linear Units) [30] are more recently proposed functions that have shown superior performance in various tasks due to their ability to model more complex patterns.

2.3.2 Loss functions

The feedforward neural network's hyperparameter tuning process involved the consideration of multiple loss functions. These loss functions serve as performance metrics for the model, quantifying the discrepancy between the model's predictions and the true values. The following sections detail the loss functions considered.

Mean Squared Error (MSE)

The Mean Squared Error (MSE) quantifies the average squared difference between the true output y_i and the predicted output $\hat{y}_i$ over N samples. It is mathematically expressed as

$$L_{\mathrm{MSE}}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \ . \tag{31}$$

In this equation, y_i corresponds to the true values, $\hat{y}_i$ represents the model's predictions, and N is the total number of samples in the dataset.

Mean Absolue Error (MAE)

Unlike MSE, the Mean Absolute Error (MAE) measures the average absolute difference between the actual and predicted values. The advantage of this approach is that it does not overly penalize large errors, as MSE does. The expression for MAE is given by

$$L_{\text{MAE}}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i| \ . \tag{32}$$

Huber Loss

The Huber loss function, denoted as $L_{\text{Huber}}(y, \hat{y})$, is a blend of MSE for smaller errors and MAE for larger ones. It introduces a hyperparameter δ which serves as a threshold defining the transition between the MSE and MAE regimes. It is less sensitive to outliers in the data than the MSE, thus preventing the model from being excessively influenced by them. It is defined as

$$L_{\text{Huber}}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq \delta \\ \delta |y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} , \tag{33}$$

where δ is a hyperparameter that controls the transition point between the squared loss (for smaller errors) and the absolute loss (for larger errors).

2.3.3 Metrics

In the training and evaluation loop of the feedforward neural network, we consider the L_1 and the L_∞ norms. We introduce first what is the meaning of Lp norm in general

Lp Norm

The general Lp norm is a metric that can be used in the training and evaluation loop. It is computed as the pth root of the sum of the pth power of the absolute relative errors between the predicted and true values. Mathematically, it is given by

$$Lp_{\text{norm}} = \left(\frac{1}{N} \sum_{i=1}^{N} \left| \frac{y_i - \hat{y}_i}{y_i} \right|^p \right)^{\frac{1}{p}} , \tag{34}$$

where p is a positive real number.

L_1 **Norm**

The L_1 norm, also known as the Mean Absolute Error (MAE), is computed as the mean of the relative errors between the predicted and true values. Mathematically, it is given by

$$L1_{\text{norm}} = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right| , \tag{35}$$

where y_i are the true values, $\hat{y}_i$ are the predicted values, and N is the number of samples.

L_∞ **Norm**

The L_∞ norm, also known as the Maximum Absolute Error, is computed as the maximum of the relative errors between the predicted and true values. Mathematically, it is given by

$$L_{\infty\text{norm}} = \max_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right| . \tag{36}$$

These topics cover the theoretical background on machine learning that is important for understanding the NN models that we have implemented for Con2Prim. Many excellent resources exist on machine learning and deep learning in particular. The reader who would like a deeper dive into the theory can consider e.g. [18, 31, 22] among many others. There are also hyperparameters besides the activation and loss functions, such as optimizers, learning rates and schedulers, which will be discussed to some amount of depth in Section 3.5.

3 Methodology

3.1 Research objective

In the project, we work towards a neural network implementation that approximates con2prim inversion in GRMHD. Specifically, we want to write a code in Python, using the open-source machine learning framework PyTorch [32], that implements a fully-connected deep feedforward NN that can be trained to approximate con2prim in GRMHD. Once trained, the NN will be ported to C++ source code.To the end of creating such a NN implementation, we will create multiple NN models.

The project is split into two parts. The first part of the project will be focused on the special relativistic case. In this part, we will focus our attention on the neural network as presented in Dieselhorst et al.'s Section 2 and their Appendix A of the paper Machine Learning for Conservative-to-Primitive in Relativistic Hydrodynamics [33]. In this part, we will write a NN from scratch, initially using the same hyperparameters as in the paper, and then using our own hyperparameters, both with the aim to get to the same order of accuracy as the published errors on their metrics. By aiming to achieve the same order of accuracy we mean that we aim to achieved errors on the metrics that are less than a factor ten times those of Dieselhorst et al.

This first part of the project furthermore serves as a playground in which we can get familiar with the PyTorch framework, try different network architectures, different hyperparameters, different numbers of data samples and experiment with other settings, as well as that it forms the basic NN architecture from which we can implement the GR case, which will be the focus of the second part of the project.

The second part of the project focuses on the more general case of GRMHD. Our primary objective is to demonstrate the viability of an ANN for con2prim in GRMHD. We define viability in this context as demonstrating potential in two critical areas. First, we aim to show that an ANN can reduce the computational time of con2prim by at least an order of magnitude, given the available computational resources, expertise in machine learning, and time to write the implementation. The performance improvement will be specifically benchmarked on the Nvidia RTX A6000 GPU. Second, we aim to illustrate that the ANN can be integrated into GRaM-X, once it has been ported to C++ source code, again given the available resources and expertise.

To clarify, demonstrating potential in this context refers to showcasing the ability to fulfill the aforementioned criteria. While the prospect of reduction in computational time is relatively straightforward, the potential for integration into GRaM-X hinges on our capacity to translate the Python-based ANN into C++ and the compatibility of this translated model with the existing GRaM-X framework.

In the light of these definitions, our research objective can be restated more explicitly: To demonstrate that a fully-connected deep feedforward ANN can be designed, implemented, and trained to approximate con2prim inversion in GRMHD with potential to reduce the computation time by at least an order of magnitude on the Nvidia RTX A6000 GPU and demonstrating the potential to integrate this ANN into the GRaM-X framework, given the resources and expertise at our disposal.

It is worth noting that while the focus of this project is predominantly on the evaluation time of the NN models, we also strive to maintain and improve accuracy where possible. Accuracy is, after all, the language in which the training of ANN is articulated, as the training of ANN aims to reduce the errors in the loss function and the metrics. Therefore, a natural part of our project will be to keep the errors as low as possible for both the SRHD and GRMHD models.

3.2 Neural Network architecture

3.2.1 Architecture for the SRHD models

The network architecture for the NNSRX models can be visualized as in Figure 4.

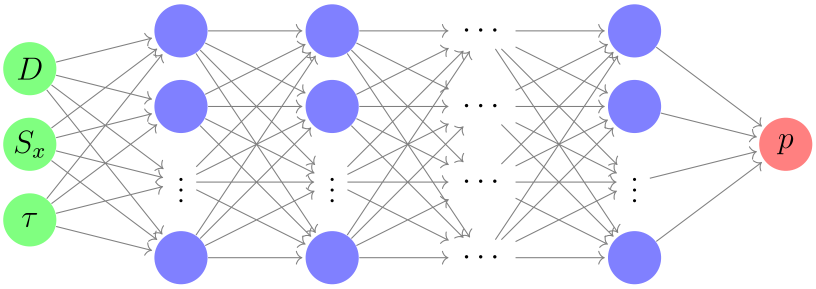


Figure 4: Illustration of the neural network architecture for the SRHD models. The network is a deep, feedforward, fully-connected structure. The input layer (green) receives three features: conserved density D, conserved momentum in the x-direction S_x, and energy density τ. An unspecified number of hidden layers (blue) interconnects all neurons from one layer to the next, passing through the activation functions (not shown). The output layer (red) consists of a single neuron, representing the primitive pressure p. The ellipsis between layers and neurons denotes an arbitrary number of hidden layers and neurons, respectively. These are part of the hyperparameters to be determined through optimization processes.

The figure illustrates a deep, fully-connected neural network, showing in green an input layer with three input features, the conserved density D, the conserved momentum in the x-direction S_x, and the energy density τ, and output layer in red with one output feature, the primitive pressure p. The hidden layers of the network are visualized in blue. All neurons in a given layer are interconnected to every neuron in the subsequent layer, symbolizing the network's learned weights. The arrows show the information flow from the input layer to the output layer. The ellipsis between the layers signifies a yet undetermined number of hidden layers, and likewise, the ellipsis between the neurons signifies a still unknown number of neurons. These are two examples of hyperparameters that will be determined by the hyperparameter optimization, discussed in Section 3.5.1 and Section 3.5.2. The network has an output layer with a single neuron, the primitive variable p, representing the target prediction.

3.2.2 Architecture for the GRMHD models

The network architecture for the GRMHD models is visualized in Figure 5. Compared to the architecture for the SRHD models, the number of inputs has increased from 3 to 14; the GRMHD models count the conserved density D, the conserved three-momentum vector $\vec{S}$, the energy density τ, the conserved magnetic field vector $\vec{\mathscr{B}}$ and the three-metric γ_{ij} as their input features. The vectors are three-dimensional and thus contain three components each, and the three-metric has six unique components: γ_{xx}, γ_{xy}, γ_{xz}, γ_{yy}, γ_{yz}, and γ_{zz}; together with the scalar inputs, this gives 14 inputs. There is still one output, but it is no longer pressure; rather, it is the quantity hW, also defined as x in Siegel et al. [34], which is the product of the enthalpy and the Lorentz factor, quantities we discussed in Section 2.2.

Due to the added complexity of the GR case, the NN will likely have a greater number of hidden layers and a greater number of units per layer. Additionally, dropout layers are implemented. The addition of dropout layers is a so-called regularization technique that helps in mitigating the tendency of a NN to overfit [35], a tendency signifying a network's excessive adaption to training data. The technique works by omitting, that is dropping out, a random number of neurons from the network, which we visualize in Figure 5 with gray disconnected neurons. The technique makes the NN less reliant on specific neurons, thereby reducing the tendency to overfit [36]. We will discuss our application of dropout regularization quantitatively in Section 3.5.2.

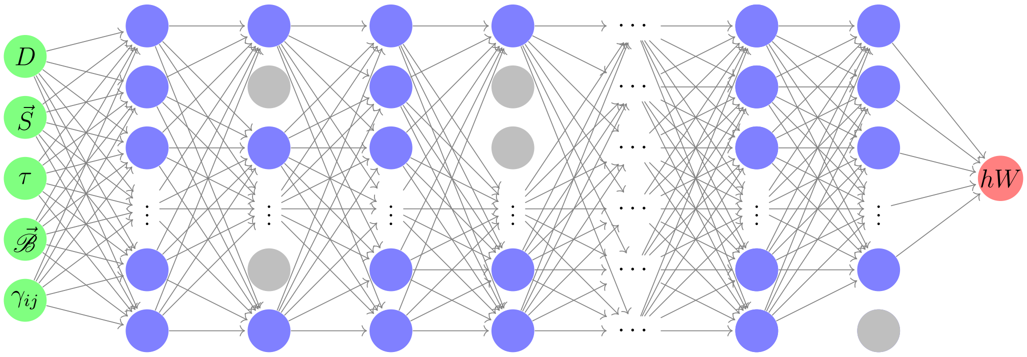


Figure 5: Illustration of the neural network architecture for the NNGR models. This architecture, compared to the NNSR, has expanded its input layer (green) to include 14 features, representing conserved density D, three-dimensional conserved momentum $\vec{S}$, energy density τ, three-dimensional conserved magnetic field $\vec{\mathscr{B}}$, and three-metric γ_{ij}. The hidden layers (blue), potentially larger in number and size due to the increased complexity of the GR case, feature dropout neurons (gray), which are randomly omitted during training to prevent overfitting. The output layer (red) consists of a single neuron, outputting the product of enthalpy and the Lorentz factor hW. The ellipsis denotes an arbitrary number of hidden layers, whose actual count and neuron count per layer are determined through optimization processes.

3.3 Program structure and flow

The high-level program flow for creating the NN models for con2prim in SRHD and GRMHD is visualized in a flowchart, split into two figures, Figure 6 and Figure 7.

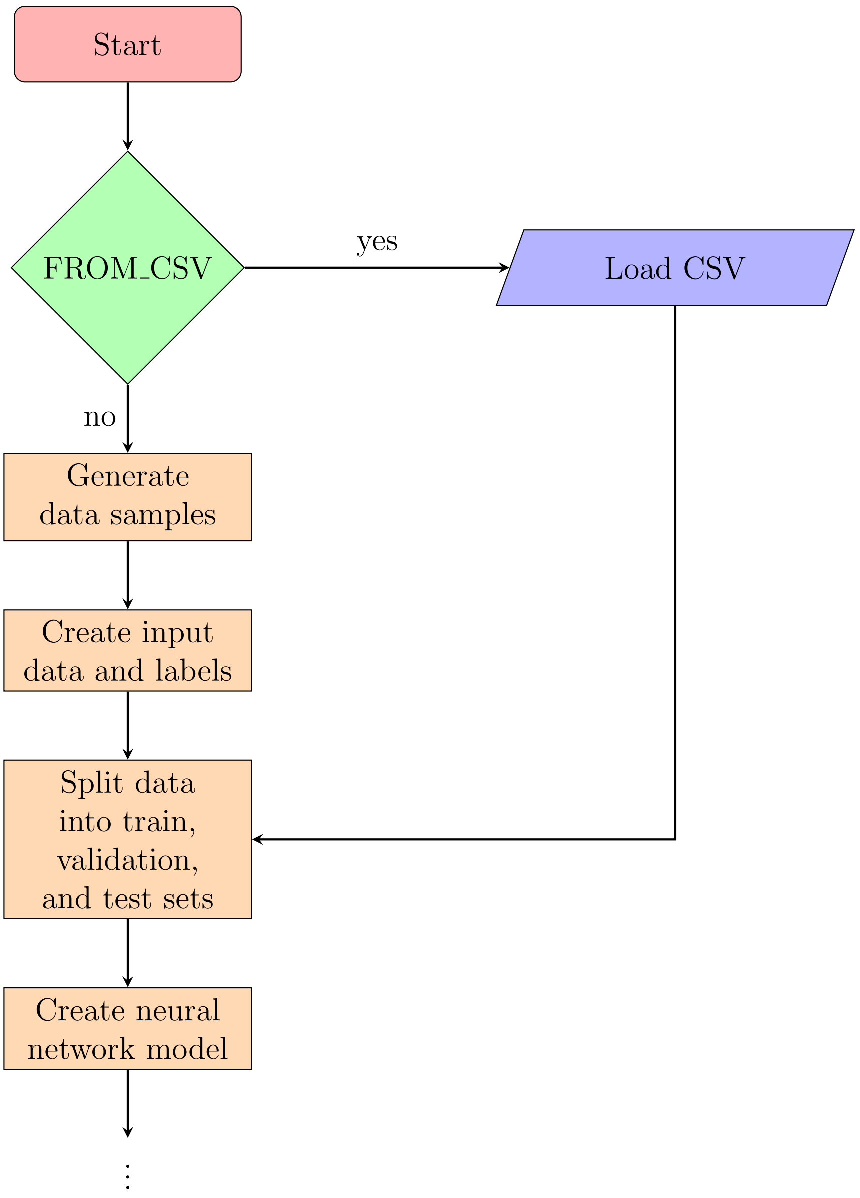

Start
FROM_CSV
yes
Load CSV
no
Generate data samples
Create input data and labels
Split data into train, validation, and test sets
Create neural network model
⋮

Figure 6: Flowchart illustrating the initial stages of the program flow for the con2prim ANNs in both SRHD and GRMHD. This includes initializing the Python script, loading data from CSV or generating new data, splitting the data into different sets, and creating the neural network model.

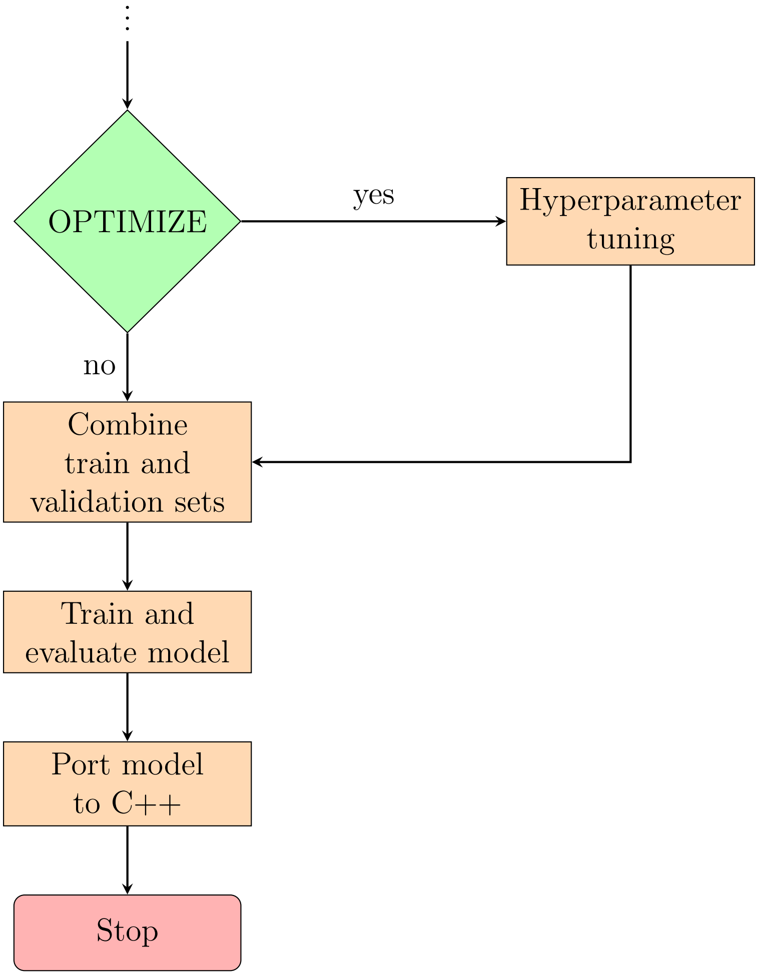


Figure 7: Flowchart illustrating the latter stages of the program flow. This includes possible hyperparameter tuning, combining the training and validation sets, training and evaluating the model, and porting the model to a C++ environment.

The first node, `Start`, signifies the initiation of the Python script. At this stage, the program checks for the availability of a GPU and sets the device to CUDA if it is available. This allows computations to be offloaded to the GPU, significantly improving performance when training the neural network models.

The second node, `FROM_CSV`, is a decision node. If this flag is set, the program loads data from CSV files as described in Section 3.4.7. If not set, the data and labels are generated and processed following the procedure outlined in the subsections of Section 3.4.

Following data loading or generation, the data is split into training, validation, and test sets, as illustrated by the node labeled `Split data into train, validation and test sets`.

Subsequently, the neural network model is created according to the specifications defined for each model. As detailed in Section 3.2, for the SRHD model, the model is built as a simple feedforward neural network with linear layers and specified activation functions.

Moving on to the second part of the flowchart, Figure 7, if the `OPTIMIZE` flag is set, if the `OPTIMIZE` flag is set, the program proceeds to hyperparameter tuning. Hyperparmeter tuning or hyperparameter optimization is done using the Optuna library [37]. Optuna automates the process of finding the best hyperparameters to optimize the performance of a machine learning model. It works by defining an objective function, which the library seeks to minimize by iteratively adjusting and testing the model's hyperparameters.

In our case, the objective function is defined such that the L_1-norm of the validation set is minimized. To efficiently sample the hyperparameters, we employ Optuna's Tree-structured Parzen Estimator (TPE) sampler. The TPE sampler uses a Bayesian approach, improving the sampling process by basing it on the results of past trials [38].

Moreover, we also utilize Optuna's MedianPruner, which is a type of pruner that terminates trials whose performance is comparatively poor at certain stages of the training. The decision to terminate is made based on whether the trial's performance is below the median of completed trials at the same step [37].

The hyperparameter optimization process continues for a number of trials specified in Table 2 of Section 3.4.1.

Whether hyperparameter optimization was performed or not, the training and validation sets are then combined into one training set in the `Combine train and validation sets` node. This is done to use all available data for training, ensuring the model has learned as much as possible before final evaluation on the test set.

In the `Train and evaluate model` node, the model is trained and evaluated either with optimized hyperparameters if the `OPTIMIZE` flag was set, or with predefined hyperparameters otherwise, as explained in detail in Section **??**.

The trained model is then converted into a form that can be run in a C++ environment using `torch.jit.trace` in the `Port model to C++` node, and then the Python code terminates. The model is ported to a C++ environment using `torch.jit.trace` [39]. The `torch.jit.trace` function is a part of the PyTorch's Just-In-Time (JIT) compilation capability that creates a trace of the model. It works by running the model once and recording the operations performed during the forward pass. For the specific procedure of importing trained models into C++ source code, we refer the reader to the Appendix on the installation of the code.

3.4 Data input to the neural network models

For the development of our artificial neural network models, we generated a significant amount of data in accordance with specific conditions, as described in the following sections.

We had considered Z-score optimization to preprocess the data, but initial trials using it did not improve network performance for the SRHD models. Therefore, in the subsequent project we did not employ any forms of data preprocessing for either the SRHD or the GRMHD models.

3.4.1 Data Specifications

The data for each model is specified according to the quantities detailed in Table 2. This table outlines the total number of samples used, the number of training samples, the number of validation samples, and the number of test samples for each model. Furthermore, it provides information about the number of hyperparameter optimization trials conducted for each model, and the number of training epochs performed after optimization.

	NNSR1	**NNSR2**	**NNSR3**	**NNSR4**	**NNGR1**	**NNGR2**
Total Samples	90k	90k	90k	90k	100k	100k
Training Samples	80k	80k	80k	80k	80k	80k
Validation Samples	–	–	–	–	15k	15k
Test Samples	10k	10k	10k	10k	15k	15k
Number of Trials	–	–	250	250	500	672
Number of Epochs	400	400	400	400	500	500

Table 2: Breakdown of the data used in the ANN models: total samples, distribution into training, validation, and test sets, and the extent of hyperparameter optimization trials and training epochs. The variations in sample numbers and trials reflect the progressive complexity and the resource allocation in model development.

In the SRHD models, each model utilizes a total of 90,000 samples, with 80,000 of these designated for training and the remaining 10,000 used for testing, following [33]. Dieselhorst et al. do not discuss the rationale for choosing 80k samples for training and 10k samples for testing, but simple speculation would suggest that they made this choice based on what experimentally worked best, as they did for settings such as the number of hidden layers, the hidden layer sizes and the choice of activation functions. We do not employ validation sets for the SRHD models. For the NNSR3 and NNSR4 models, a total of 250 optimization trials are performed to tune the hyperparameters.

Moving to the GRMHD models, we increase the total samples to 100,000, with the same number, 80,000, used for training.We experimented with increasing the sample size to a size of 1000k to see if it is beneficial to hyperparameter optimization, but we did not notice any improvements over using hyperparameter optimization with a sample size of 100k, and, together with the significant increase in computational time that it took to optimize with ten times the amount of data, we decided to keep the sample size at 100k. In contrast to the SRHD models, the NNGR1 model utilize a validation set of 15,000 samples, with the remaining 15,000 set aside for testing. Moreover, the number of optimization trials is doubled to 500 for NNGR1 and set to

NNGR2 for 672, the latter of which was a practical limitation of the setup rather than a logically reasoned value.

The current numbers of optimization trials used (Table 2) are pragmatically set rather than logically reasoned; we ran the SRHD models on Google Colab, and 250 was used for the number of trails to obtain optimal hyperparameter results within a few hours. For the GRMHD models, we had access to the local workstation, on which we could run for longer, so we set the number of trials to 500.

These specifications show a progression in complexity and resources utilized, with a greater emphasis on hyperparameter optimization and validation in the more advanced GRMHD models.

3.4.2 Sampling of variables

We performed variable sampling within specified intervals for each model. The intervals for these samplings are detailed in Table 3.

Variable	**NNSR Interval**	**NNGR Interval**
ρ	[0, 10.1]	[0, 2]
ϵ	[0, 2.02]	[1e-2, 2000], (log$_{10}$)
v_x	[0, 0.721]	[0, 0.999]
v_y	–	[0, 0.999]
v_z	–	[0, 0.999]
B_x	–	[-10, 10]
B_y	–	[-10, 10]
B_z	–	[-10, 10]
γ_{xx}	–	[0.9, 1.1]
γ_{xy}	–	[0, 0.1]
γ_{xz}	–	[0, 0.1]
γ_{yy}	–	[0.9, 1.1]
γ_{yz}	–	[0, 0.1]
γ_{zz}	–	[0.9, 1.1]

Table 3: Intervals of uniform and log-uniform sampling for variables ANN models. For the SRHD models, rest-mass density (ρ), specific internal energy (ϵ), and velocity in the x-direction (v_x) are sampled. For the GRMHD models, in addition to these variables, velocities in the y and z-directions, magnetic field components, and the components of the metric tensor are sampled. Log-uniform sampling is used only for the specific internal energy (ϵ) in the GRMHD models.

In the SRHD models, the rest-mass density ρ is sampled uniformly from the range $[0, 10.1]$. Similarly, the velocity in the x-direction v_x is sampled uniformly between 0 and 0.721, and the specific internal energy ϵ is sampled within the interval $[0, 2.02]$.

Conversely, for the GRMHD models, the rest-mass density ρ is uniformly sampled from a more constrained interval $[0, 2]$. The velocities in the x, y, and z directions are each uniformly sampled from $[0, 0.999]$. The specific internal energy, ϵ, is sampled log-uniformly between $1e-2$ and 2000.

Moreover, the GRMHD models introduce additional variables which are not used by the SRHD models. These include the magnetic field components B_x, B_y, and B_z, each uniformly sampled from $[-10, 10]$. Furthermore, the components of the metric tensor γ_{xx}, γ_{xy}, γ_{xz}, γ_{yy}, γ_{yz}, and γ_{zz} are uniformly sampled within $[0.9, 1.1]$ for the diagonal elements, and within $[0, 0.1]$ for the off-diagonal elements.

3.4.3 Validity of samples

We impose three conditions on the samples to ensure their validity:

1. If $v^{x2} + v^{y2} + v^{z2} \geq 1$, the sample is rejected.
2. If $WT < 0$, the sample is rejected.
3. If $\sqrt{\gamma} < 0$, the sample is rejected.

If *none* of these conditions are met, the sample is *accepted*. The first condition imposes a limit on the speed of the fluid such that it is less than the speed of light. The second condition prevents a division by zero in the calculation of the conserved magnetic field, as detailed in the code snippet provided in the appendix. The third condition avoids the evaluation of an imaginary square root. We re-sample until the number of accepted samples matches the number specified in Table 2.

3.4.4 Equation of State

We assume an analytical EoS for both the SRHD and GRMHD models. The form of the EoS is based on the general equation presented in equation (13) from Section 2.2.2.

The analytical equation used to compute the pressure p is given by:

$$p = (\Gamma - 1)\rho\epsilon \ , \tag{37}$$

where Γ is the adiabatic index, set to $5/3$, ρ is the primitive rest-mass density, and ϵ is the primitive specific internal energy. This EoS is used throughout the calculations in the SRHD and GRMHD models.

3.4.5 Primitive-to-Conservative Transformation

The conserved variables are the inputs to the neural networks. In order to compute these conserved variables, the models make use of the equations of fluid dynamics and the EoS to transform the sampled primitive variables into conserved variables. Specifically, the SRHD models use equations (9), (10), and (11) from Section (2.2.1), while the GRHMD models use equations (28), (29), (30), and (24) from Section 2.2.4. Once the conserved variables are computed, they are stacked into a single PyTorch tensor for input into the models.

3.4.6 Generation of Labels

The labels for the models are generated from the computed primitive variables. For the SRHD models, as the output is the primitive variable pressure p, the labels are generated by evaluating equation

$$p = \frac{2}{3}\rho\epsilon \; ; \tag{37}$$

the analytical EoS described in Section 3.4.4.

For the GRHMD models, the labels are generated differently. Firstly, the primitive variable pressure is obtained by evaluating the same EoS as specified by equation (37). Subsequently, the quantity hW is computed using equations (12) and (7) from Section 2.2.1. The computed hW become the labels for the GRMHD models. The labels for the GRMHD models are thus generated using the following equation

$$hW = \left(1 + \epsilon + \frac{p}{\rho}\right) \frac{1}{\sqrt{1 - (v^x)^2 - (v^y)^2 - (v^z)^2}} \,, \tag{38}$$

where h is the specific enthalpy, W is the Lorentz factor, ϵ is the specific internal energy, p is the pressure, ρ is the rest-mass density, and v^x, v^y, and v^z are the velocities in the x, y, and z directions respectively.

3.4.7 Importing data

Alternatively to generating the data, we provide the option to import the data directly from CSV files. This requires six separate files that correspond to different parts of the dataset: the training input, the validation input, and the test input, along with their respective labels, the training labels, the validation labels, and the test labels. This arrangement allows for flexibility, accommodating cases where the data already exists in an appropriate format.

3.5 Hyperparameter optimization

3.5.1 Hyperparameter optimization for SRHD models

The search spaces for the hyperparameters of the SRHD models is given in Table 4. We experiment with different search spaces for NNSR3 compared to that of the NNSR4 and NNSR5 models. For the NNSR4 and NNSR5 models we also experiment with subparameters for some of the optimizers and some of the schedulers; these subparameter search spaces are given in Appendix B.

Hyperparameter	**Search space for NNSR3**	**Search space for NNSR4 and NNSR5**
Number of hidden layers (n_{layers})	$1 \leq n_{\text{layers}} \leq 3$	$1 \leq n_{\text{layers}} \leq 5$
Number of units per layer (n_{units})	$16 \leq n_{\text{units}} \leq 256$ for each layer	$16 \leq n_{\text{units}} \leq 1048$ for each layer
Hidden activation	ReLU, LeakyReLU, ELU, Tanh, Sigmoid	ReLU, LeakyReLU, ELU, Tanh, Sigmoid
Output activation	Linear, ReLU	ReLU, Softplus
Loss function	MSE, MAE, Huber, LogCosh	MSE, MAE, Huber
Optimizer	Adam, SGD, RMSprop, Adagrad	Adam, SGD, RMSprop, Adagrad
Learning rate (η)	$1 \times 10^{-4} \leq \eta \leq 1 \times 10^{-2}$ (log-uniform)	$1 \times 10^{-4} \leq \eta \leq 1 \times 10^{-2}$ (log-uniform)
Batch size	$32 \leq$ Batch size ≤ 256	$16 \leq$ Batch size ≤ 512
Number of epochs (n_{epochs})	$50 \leq n_{\text{epochs}} \leq 100$	$50 \leq n_{\text{epochs}} \leq 200$
Scheduler	None, CosineAnnealingLR, ReduceLROnPlateau, StepLR, ExponentialLR	None, CosineAnnealingLR, ReduceLROnPlateau, StepLR, ExponentialLR

Table 4: Hyperparameter search spaces for different SRHD models. The table compares the ranges of hyperparameters considered during optimization for the NNSR3 model and the NNSR4 and NNSR5 models. The hyperparameters include architectural components such as number of hidden layers and units per layer, activation functions, and training settings such as loss function, optimizer, initial learning rate, batch size, number of epochs, and learning rate scheduler.

We now discuss the grounds for these hyperparameter search spaces. It is noteworthy that we are experimenting with all the settings, and we thereby choose the search spaces broadly, so to consider many combinations of settings. Because of this, some search spaces are non-optimal.

Number of hidden layers

The number of hidden layers in a neural network architecture can be related to the complexity of the target problem, as more hidden layers can offer greater capacity for capturing intricate relationships within the data [18]. However, it is important to consider the trade-off between model complexity and overfitting, which requires an adequate quantity of training data to support the complexity of the model [40]. 1 to 3 hidden layers are considered for NNSR3 as a simple extension to Dieselhorst et al.'s consideration of 2 to 3 hidden layers to experiment with 1 hidden layer as well. 1 to 5 hidden layers were chosen for NNSR4 and NNSR5 both as an extension to this space and as a balance between the two aforementioned factors of modelling complex data and overfitting.

Number of units per hidden layer

In NNSR3 we start out small with an interval of 16 to 256 for the number of units per hidden layers, coming from the idea of setting up an interval around the hidden size of 200 used in Dieselhorst et al. We then extend to a search space of 16 to 1048. This range provides a balance between underfitting and overfitting, while also ensuring the model has enough representation power to capture the complex, non-linear relationships in the con2prim problem.

Activation functions

Dieselhorst et al. considered the activation functions of tanh, Sigmoid and ReLU for the hidden activation and the non-linearity applied to the output. To broaden the space explored for the hidden layers, we consider in addition to these functions the LeakyReLU and ELU. For the output nonlinearity, in NNSR3 we consider both no output nonlinearity being applied (i.e. the `linear` option) and ReLU, the latter being what Dieselhorst et al. found to work best. For NNSR4 and NNSR5 we consider SoftPlus as another possibility. There is no reason not to consider no output nonlinearity for NNSR4 and NNSR5 as we do for NNSR3; this was an inadvertent omission in the setup.

Loss function

The choice of the loss functions of MSE, MAE, Huber and LogCosh for NNSR3 comes forth from having a selection of loss functions such that the model is able to learn well from the data, handle outliers and reduce large errors. In NNSR4 and NNSR5 we removed the LogCosh loss function as it did not provide a particular advantage to include it.

Optimizer

Just as with the choice of loss function, we want to have a wide space of optimizers to try out in order to make sure that the model is able to choose what is most effective in solving the con2prim problem. SGD and its variants, Adam, RMSProp, Adagrad are some of the most common choices in deep learning for the choice of optimizers [18].

Learning rate

The learning rate is a crucial parameter for the NN model [18]. A learning rate too large can cause unstable learning, with the loss function fluctuating significantly or even diverging. In contrast, a learning rate too small can lead to very slow convergence or the optimization process getting stuck in less optimal solutions. With a learning rate range of 10^{-4} to 10^{-2} we hope to strike a balance between these extremes, offering stable learning while maintaining an adequate pace of convergence. With this range, we let Optuna's algorithm further decide on what is an appropriate learning rate given the other hyperparameters; most importantly, given the scheduler, which we will discuss last of the hyperparameters.

Batch size

The literature considers both small batch sizes, in the range of 2 to 32, as well as large batch sizes, in the range of 1000s [41]. For NNSR3, we are choosing a middle ground, exploring batch sizes from 32 to 256, while for NNSR4 and NNSR5 we explore batch sizes in the range of 16 to 515.

Number of epochs

The choices to have the number of epochs be between 50 and 100 for the NNSR3 search space and between 50 and 200 for the NNSR4 and NNSR5 search spaces have been pragmatic ones in terms of the computational time it takes to run high-epoch trials; although it is worth mentioning that the ability of Optuna to prune unpromising trials allows for a higher bound for the number of epochs than would otherwise be practical. The increase in number of epochs for the NNSR3 and NNSR4 model is an effort to capture the possibly higher complexity of a model that comes forth from the hyperparameter tuning.

Scheduler

For the scheduler we also choose a wide landscape to provide the possibility adjust the learning rate in different ways.

The settings *None* means having no scheduler, which in turn means that the set learning rate η stays constant over all epochs of a trial or training. The CosineAnnealingLR scheduler follows a cyclical approach, that of a cosine curve, and periodically *anneals* the learning rate from a higher value down to a minimum, and resets it to the initial value upon reaching the minimum, thereby initiating a new cycle [42]. The cyclical behavior of the scheduler is clearly visualized in the training curves of NNSR4 in Figure 12 Section 4. Schedulers like CosineAnnealingLR aid in avoiding local minima because of this dynamical adjustment of the learning rate[43]. The ReduceLROnPlateau scheduler gets its name from adjusting the learning rate scheduler when the metric against epochs curves shows a plataeu; e.g. when the metric has stopped improving for a set amount and after a set number of epochs determined by the subparameters. The behavior is especially clearly visualized in the training curves of NNSR3 in Figure 11 of Section 4. For the specific search space intervals for the subparameter, please consult Table 10 of Appendix B. The StepLR scheduler reduces the learning rate at fixed time intervals, unlike the CosineAnnealing scheduler and the ReduceLROnPlateau scheduler, which both adjust the learning rate based on the performance of the model. Lastly, the ExponentialLR scheduler applies an exponential decay to the learning rate after each epoch, allowing the model to make larger updates during the initial training phase and progressively smaller, more refined updates in the later stages.

3.5.2 Hyperparameter optimization for GRMHD models

The hyperparameter search space we provide for the GRMHD models is shown in Table 5. Finding a more finely tuned search space than shown is out of the scope of the project, as it requires more time to experiment and a greater dive into the theory of deep learning. The search space as it is chosen here should be seen as a rough and broad starting point, leveraging Optuna's TPE sampler and pruning algorithms, rather than having a carefully narrowed-down search space. Nonetheless, we briefly go over the choice of hyperparameters search spaces for these models.

Hyperparameter	**Search space**
Number of hidden layers (n_{layers})	$1 \leq n_{\text{layers}} \leq 5$
Number of units per hidden layer (n_{units})	$16 \leq n_{\text{units}} \leq 4096$ for each layer
Hidden activation function	ReLU, LeakyReLU, ELU, PReLU, Swish, GELU, SoftPlus
Output activation function	Linear
Loss function	MSE, MAE, Huber
Optimizer	Adam, SGD, RMSprop, Adagrad
Learning rate (η)	$1 \times 10^{-4} \leq \eta \leq 1 \times 10^{-2}$ (log-uniform)
Batch size	32, 64, 128, 256, 512
Number of epochs (n_{epochs})	$50 \leq n_{\text{epochs}} \leq 150$
Scheduler	CosineAnnealingLR, ReduceLROn-Plateau, StepLR
Dropout rate (p_{dropout})	$0.0 \leq p_{\text{dropout}} \leq 0.5$

Table 5: Hyperparameter search space for the GRMHD models with ranges and options for various aspects such as the number of layers, units per layer, activation functions, loss function, optimizer, initial learning rate, batch size, number of epochs, scheduler, and dropout rate. These parameters provide a broad starting point for optimization, subject to fine-tuning using Optuna's TPE sampler and pruning algorithms.

Number of hidden layers

The GRMHD models have increased complexity due to the change in underlying equations and the additional inputs compared to the SRHD models. We experiment with 1 to 5 hidden layers. It is possible to go beyond this range, but it requires a more extensive dive into the theory, as deeper networks are harder to optimize and are more prone to overfitting [18].

Number of units per layer

We set a broad range of 16 to 4096 units for each hidden layer as it is not a priori clear what is a number of units fits the model's complexity; rather, it is an empirical question, which we let Optuna's algorithm decide further.

Activation functions and output nonlinearity

We add PReLU, Swish, GELU and SoftPlus as additional options to explore. As the output is no longer exclusively non-positive, we do not apply a non-linearity to the output.

Loss function and optimizer
The same search space of loss functions and search space of optimizers as for the SRHD models is used.

Learning rate
We keep the log-uniform range of the SRHD models.

Batch size
The literature shows that batch sizes of 32 to 512 are most effective for SGD and its variants [44]. Larger batch sizes can leverage the GPU's ability to do its computations in parallel, but hurt the model's ability to generalize [45], we therefore stay with the range of 32 to 512.

Number of epochs
Initial runs with the search space for the SRHD models suggested to us that having the number of epochs be between 50 and 150 is sufficient for Optuna to determine the goodness of the combination of hyperparameters in question.

Scheduler
The same schedulers as for the SRHD models are considered.

Dropout
A selection between 0 and 50% of the neurons in each hidden layer will be disconnected from the neural network. According to the seminal paper on dropout by Srivastava et al., the optimal value for most task is a dropout rate of 50% of the neurons [40]. We let Optuna explore the space of anywhere between 0 and 50% to see if there are advantageous combinations below 50% as well.

3.6 Training and Evaluation of Artificial Neural Network Models

This section presents the methodology for the training and evaluation of the NN models.

The training and evaluation procedure, as illustrated in Figure 8, is a standard approach in the field of machine learning. This procedure represents a cyclic adjustment of the model parameters with an objective to minimize a defined loss function, and to evaluate the performance on unseen data, thereby ensuring the model's ability to generalize beyond the training set.

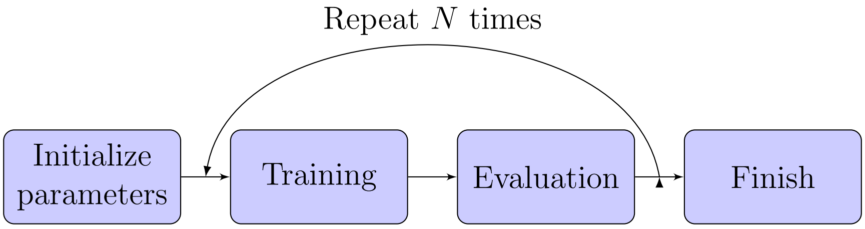


Figure 8: Schematic diagram depicting the iterative process of training and evaluation in a machine learning model. The cyclic flow signifies the repeated adjustment of model parameters for N epochs, which is aimed at minimizing a defined loss function and evaluating the model's performance on unseen data.

Training commences with the loading of training and test samples using `torch.Dataloader`, and the initialization of lists to hold train and test losses.

Following this setup, the models undergo the iterative process of training and evaluation. The training phase is centered around adjusting the weights of the neural network with the objective of minimizing the discrepancy between the predicted output and the actual output. Simultaneously, the evaluation phase is focused on assessing the model's performance on the test set without any parameter adjustment [18].Upon the completion of the training and evaluation processes, the models return train and test losses, along with the corresponding metrics—the L_1 and L_∞ norms—at the end of each epoch. An epoch concludes when the entire dataset has been passed forward and backward through the neural network once. The entire procedure from initialization to evaluation is repeated N times, where N is a predetermined number of epochs for each model as specified in Table 2. This repetition is captured in Figure 8, which showcases the looping path between evaluation and training.

An in-depth exploration of the mechanisms employed in the training and evaluation processes of ANNs can be found in Appendix A.1 and Appendix A.2, correspondingly.

4 Results

This section presents the results of our project to approximate con2prim using ANNs.

4.1 Models for SRHD

We commence by presenting the results for the SRHD models.

4.1.1 Hyperparameters for the SRHD models

In our study, we created four different artificial neural network models for con2prim n SRHD and compare these with existing models by Dieselhorst et al. The hyperparameters of these models are presented in Table 6.

Table 6: A comprehensive comparison of hyperparameters utilized in all for ANNs for con2prim in SRHD, including Dieselhorst et al.'s models NNC2PS, NNC2PL, and our newly developed models NNSR1-4. Hyperparameters including the number of hidden layers and units, activation functions for both hidden and output layers, loss functions, optimizers, learning rates, batch sizes, learning rate schedulers, and specific parameters for these schedulers are presented.

Hyperparameter	NNC2PS & NNSR1	NNC2PL & NNSR2	NNSR3	NNSR4
Number of hidden layers	2	2	3	5
Number of hidden units	600, 200	900, 300	555, 458, 115	617, 858, 720, 989, 613
Hidden activation	Sigmoid	Sigmoid	ReLU	ReLU
Output activation	ReLU	ReLU	ReLU	ReLU
Loss function	MSE	MSE	Huber	MSE
Optimizer	Adam	Adam	RMSprop	Adagrad
Learn. Rate	6×10^{-3}	6×10^{-3}	1.23×10^{-4}	1.69×10^{-3}
Batch Size	32	32	49	16
LR Scheduler	Red. Plat.	Red. Plat.	Red. Plat.	Cos. Ann.
CosineAnnealingLR *T_max*	–	–	–	8
ReduceLROnPlateau factor	0.5	0.5	1.897×10^{-1}	–
ReduceLROnPlateau patience	5	5	11	–
ReduceLROnPlateau threshold	5×10^{-4}	5×10^{-4}	1.720×10^{-3}	–
ReduceLROnPlateau min_lr	1×10^{-6}	1×10^{-6}	0	−−

Dieselhorst et al.'s, NNC2PS model comprises of 2 layers, with the number of units in each layer being 600 and 200 respectively. For the hidden activation function Sigmoid is used, while the output activation used is ReLU. Furthermore, an MSE loss function is used in this model, Adam is used as the optimizer, and the initial learning rate is set to 6×10^{-3}. The model trains with a batch size of 32 and employs the ReduceLROnPlateau learning rate scheduler with a factor of 0.5, a patience of 5, and a threshold of 5×10^{-4}. The first model we developed, NNSR1, parallels Dieselhorst et al.'s NNC2PS in terms of its hyperparameters, where in addition we set the `min_lr` subparameter of the ReduceLROnPlateau scheduler, to 1×10^{-6}; the value set by Dieselhorst et al. was not specified. Dieselhorst et al's NNCPL and our NNSR2 parallel NNC2PS, except for the larger sizes of 900 and 300 used for the number of hidden units of the two layers, and in NNSR2 we too set `min_lr`.

The NNSR3 model introduces some variations in the hyperparameters. It consists of 3 layers with 555, 458, and 115 units respectively. This model uses the ReLU activation function for both the hidden and output layers, and employs the Huber loss function. The optimizer used here is RMSprop, with a considerably reduced learning rate of 1.23×10^{-4} compared to the previous models. The batch size is increased to 49, while the learning rate scheduler remains to be ReduceLROnPlateau. The parameters of the scheduler change, with a factor of 1.897×10^{-1}, a patience of 11, a threshold of 1.720×10^{-3}, and a minimum learning rate of 0.

Finally, the NNSR4 model has a more complex architecture with 5 layers, comprising 617, 858, 720, 989, and 613 units. Like NNSR3, it uses the ReLU activation function for the hidden and output layers, and employs the MSE loss function. The Adagrad optimizer is employed here, with a learning rate of 1.69×10^{-3}. This model trains with a smaller batch size of 16 and switches to the Cosine Annealing learning rate scheduler with a T_max of 8.

4.1.2 Settings and runtime values

Additional settings and results alongside the hyperparameters are presented in Table 7. The SRHD models, designated as **NNSR1** through **NNSR4**, were trained and evaluated with the same set of a total of 90,000 samples each – 80,000 for training and 10,000 for testing. No validation sets were utilized for any of these models.

	NNSR1	**NNSR2**	**NNSR3**	**NNSR4**	**NNGR1**	**NNGR2**
Total samples	90k	90k	90k	90k	100k	100k
Train samples	80k	80k	80k	80k	80k	80k
Val. samples	–	–	–	–	15k	15k
Test samples	10k	10k	10k	10k	15k	15k
Trials	–	–	250	250	500	672
Epochs	400	400	400	400	500	500
Params	100k	300k	300k	2500k	5000k	2500k
Test L_1 error	1.12×10^{-2}	7.24×10^{-3}	2.22×10^{-3}	1.97×10^{-3}	1.23×10^{0}	2.07×10^{0}
Test L_∞ error	2.37×10^{-1}	1.92×10^{-1}	5.44×10^{-2}	2.29×10^{-2}	7.48×10^{1}	2.95×10^{2}
Average eval. time (µs)	88.3	72.7	361	245	550	581
Best eval. time (µs)	18.4	20.5	24.6	64.5	404	368

Table 7: Summary of settings, performance metrics and evaluation times for all developed ANNs. The tabulated settings are the total, training, validation, and testing samples, the number of hyperparameter optimization trials, the number of epochs, the number of trainable parameters, the L_1 and L_∞ error rates and the average and best evaluation times.

The architecture and complexity of the SRHD models were varied. The simplest model, NNSR1, has 122,801 trainable parameters. The complexity gradually increases, with NNSR2 having 274,201 parameters, NNSR3 having 309,769 parameters, and the most complex model, NNSR4, having 2,471,745 parameters.

NNSR1 and NNSR2 were modeled after the NNC2PS and NNC2PL models of Dieselhorst et al.[33], respectively, hence no hyperparameter optimization was performed. For NNSR3 and NNSR4, 250 hyperparameter optimization trials were performed. All models were trained for 400 epochs.

The models' performance was measured using L_1 and L_∞ norm errors.

The L_1 and L_∞ norm errors were 1.12×10^{-2} and 2.37×10^{-1}, respectively for NNSR1, 7.24×10^{-3} and 1.92×10^{-1} respectively for NNSR2, 2.22×10^{-3} and 5.44×10^{-2} respectively for NNSR3 and NNSR4 1.97×10^{-3} and 2.29×10^{-2} respectively for NNSR4. Comapring the errors of the models to those of the NNC2PS and NNC2PL models of Dieselhorst et al., we find that the L_1 and L_∞ norm errors for NNSR1 were 29.2 and 29.1 times larger than those for NNC2PS respectively. In a similar manner, the errors were 20.0 and 20.7 times larger for NNSR2 than those of NNC2PL. Comparing our remaining models also compared to NNC2PS, as this was Dieselhorst et al.'s better model, the L_1 and L_∞ norm errors for NNSR3 were 5.78 and 6.68 times larger respectively, while they were 5.13 and 2.81 times larger for NNSR4.

Plots of the training and testing of the SRHD models are presented in Figure 10, Figure **??**, Figure 11 and Figure 12. Each of these figures provides two plots for the model in question, in which the left plot depicts the L_1 norm train and test errors, whereas the right plot outlines the L_∞ norm train and test errors. In all plots, the horizontal axis gives the number of epochs and ranges from 0 to 400, while the y-axis represents the respective metric, varying from 10^{-3} to 10^2.

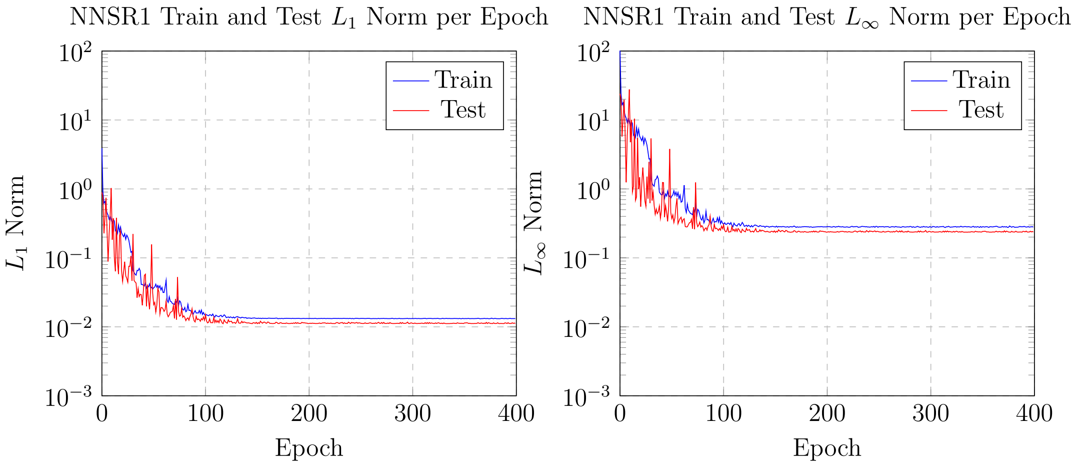


Figure 9: L_1 and L_∞ norm error plots per epoch for the NNSR1 model during training (blue) and testing (red) for 400 epochs. The L_1 and L_∞ norm errors decrease over the epochs, where from about 200 epochs onward they stay roughly constant.

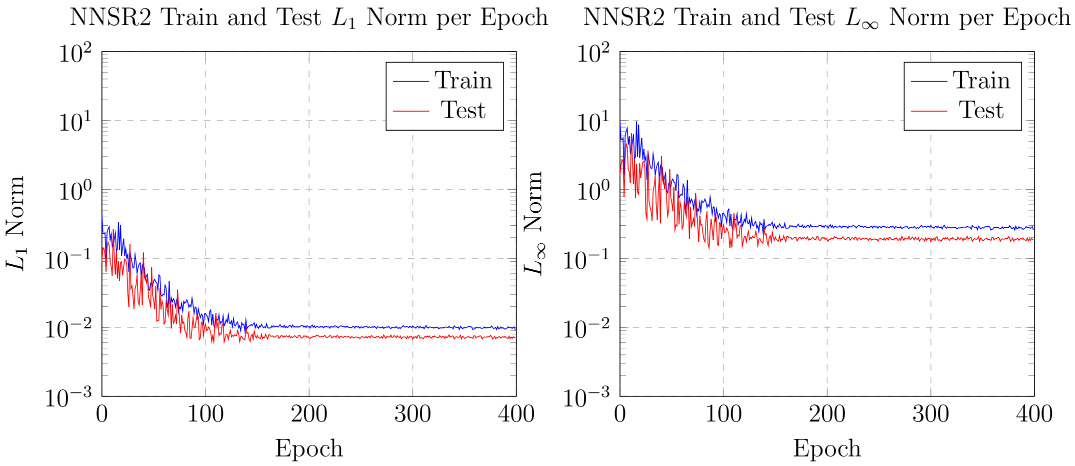


Figure 10: L_1 and L_∞ norm error plots per epoch for the NNSR2 model during training (blue) and testing (red) for 400 epochs. As with the NNSR1 model, the L_1 and L_∞ norm errors decrease over the epochs, where from about 200 epochs onward they stay roughly constant.

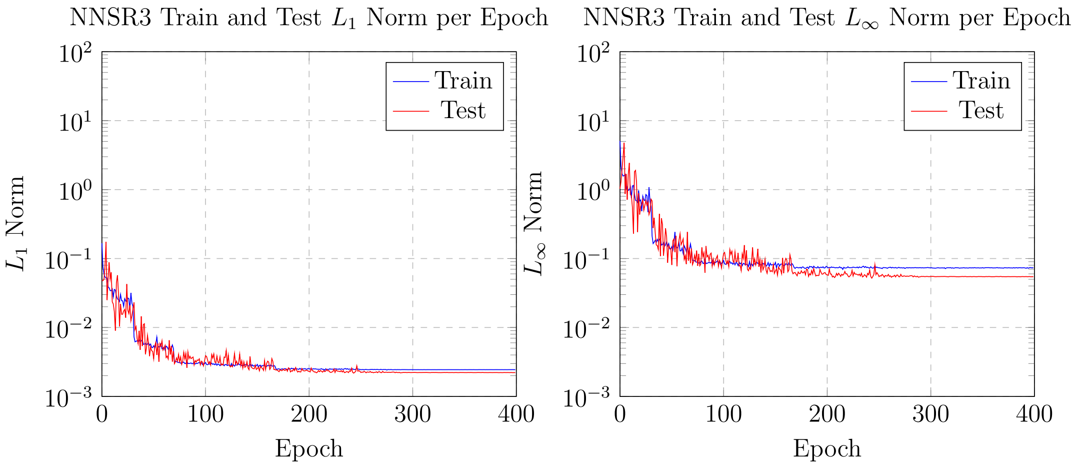


Figure 11: L_1 and L_∞ norm error plots per epoch for the NNSR3 model during training (blue) and testing (red) for 400 epochs. Like with the NNSR1 and NNSR2 models, the L_1 and L_∞ norm errors decrease over the epochs, where from about 200 epochs onward they stay roughly constant.

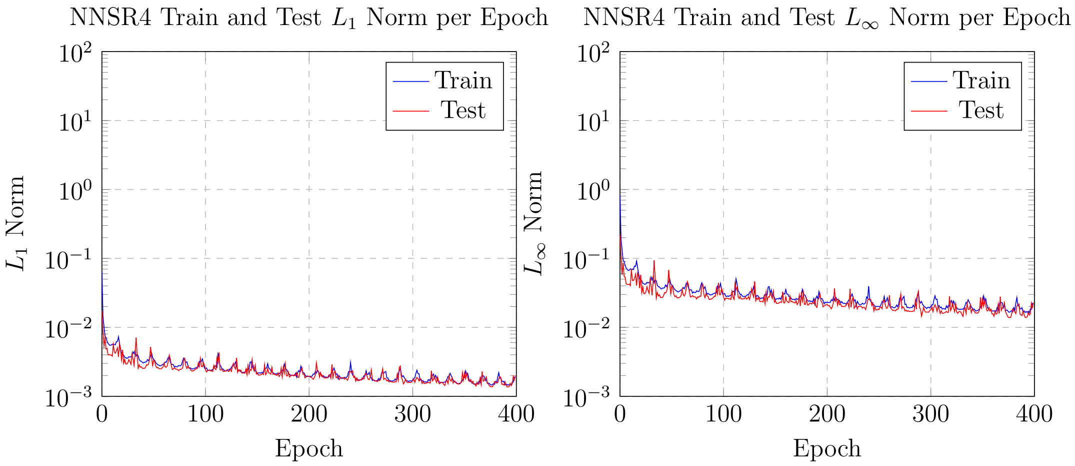


Figure 12: L_1 and L_∞ norm error plots per epoch for the NNSR4 model during training (blue) and testing (red) for 400 epochs. Like with the NNSR1–3 models, the L_1 and L_∞ norm errors decrease over the epochs, where from about 200 epochs onward they stay roughly constant. The curve of the NNSR4 model looks different due to the use of the Cosine Annealing learning rate scheduler as opposed to the ReduceLROnPlateau scheduler used by NNSR1–3

In terms of the overall trend, it is evident that as the number of epochs increases, there is a decrease in both the L_1 and L_∞ errors for all NNSR models.

The test errors remain below the training errors for all models except for occasional spikes where they overshoot in the first 200 epochs.

The evaluation times for the models were also measured. The average evaluation times for NNSR1, NNSR2, NNSR3, and NNSR4 were 88.3, 72.7, 361, and 245 microseconds, respectively. The best evaluation times were 18.4, 20.5, 24.6, and 64.5 microseconds, respectively. Evidently, NNSR2 model achieved the lowest average evaluation time, and NNSR1 yielded the lowest minimal evaluation time. These times were computed in Python on a workstation with an Nvidia RTX A6000 GPU[**?**] and using the `torch.cuda.Event` functionality from PyTorch[46]. The average evaluation times where computed by evaluating the NN ten times in a row and taking the average time. `torch.cuda.Event` enables accurate time measurements in CUDA environment, offering more precise GPU performance metrics than Python's `time` module.

4.2 Models for GRMHD

We now present the results for our GRMHD models.

4.2.1 Hyperparameters for NNGR1 and NNGR2

In Table 8, we provide a comprehensive comparison of the hyperparameters used for our two GRMHD models, NNGR1 and NNGR2.

Table 8: Comparison of hyperparameters for the GRMHD models. The table details the number of hidden layers, number of hidden units per layer, the activation functions, loss function, optimizer, learning rate, batch size, learning rate scheduler, dropout rate, and specific parameters for the PReLU activation function and the learning rate schedulers for each model.

	NNGR1	NNGR2
Hidden layers	5	5
Hidden units	216, 2666, 1459, 485, 103	900, 113, 1440, 478, 3328
Hidden activation	PReLU	PReLU
Output activation	Linear	Linear
Loss function	MSE	MSE
Optimizer	Adagrad	Adagrad
Learn. rate	1.37×10^{-4}	1.11×10^{-4}
Batch size	512	512
LR scheduler	StepLR	ReduceLROnPlateau
Dropout rate	2.86×10^{-1}	4.67×10^{-1}
PReLU init	1.15×10^{-1}	1.75×10^{-1}
StepLR step size	7	–
StepLR gamma	4.33×10^{-1}	–
Red Plat. factor	–	1.27×10^{-1}
Red Plat. patience	–	5
Red Plat. threshold	–	6.89×10^{-3}

Both models come with 5 hidden layers, and with PReLU as the activation function for the hidden layers. The sizes of the hidden layers for NNGR1 are $[216, 2666, 1459, 485, 103]$ and for NNGR2 they are $[900, 113, 1440, 478, 3328]$. No output activation is used for either model, as the quantity hW that is being output by the networks can be any real number. Both use MSE as the loss function, and Adagrad for the optimizer. The initial learning rates used are 1.37×10^{-4} for NNGR1 and 1.11×10^{-4} for NNGR2. The batch size is common between the two models and is set to 512 for both models.

To prevent overfitting, dropout with rates of 2.86×10^{-1} for NNGR1 and 4.67×10^{-1} for NNGR2 are used. The initial value for the PReLU activation function is set to 1.15×10^{-1} and 1.75×10^{-1} for NNGR1 and NNGR2 respectively. While both models employ learning rate scheduling, they use different strategies. NNGR1 uses StepLR with a step size of 7 and gamma subparameter of 4.33×10^{-1}, while NNGR2 uses the ReduceLROnPlateau scheduler with a factor of 1.27×10^{-1}, patience of 5, and threshold of 6.89×10^{-3}.

4.2.2 Settings and runtime values

We now consider settings besides the hyperparameters as well as the performance of the two GRMHD models.

The GRMHD models have the same structure for training and validation; they use a total of 100,000 samples, with 80,000 for training, 15,000 for validation, and 15,000 for testing as per Table 7. Hyperparameter optimization was conducted over 500 trials for each model, with the network trained for a total of 500 epochs in each case.

When considering the number of trainable parameters, NNGR1 is configured with a total of 5,231,178 parameters, while NNGR2 has about half of this amount, namely 2,565,713 parameters.

In terms of performance, we note that the L_1 norm and the L_∞ norm errors of NNGR1 are a factor of 110 and 31.6 larger, respectively, than the worst errors of the SRHD models, i.e. than those of NNSR1. For NNGR2, these factors grow even more, to 185 and 1250 respectively. Regardless of these larger errors, of the GRMHD models, NNGR1 achieves a lower L_1 norm error, while NNGR2 has a lower L_∞ norm error.

As for the SRHD models, the evaluation times for these models were also computed using the Nvidia RTX A6000 GPU and `torch.cuda.Event` for the time measurement. In terms of absolute values, the average evaluation times for NNGR1 and NNGR2 were found to be 550μs and 581μs respectively. In comparison, NNGR1's average evaluation time was found to be a factor of 1.52 longer than the longest average time for the SRHD models, attributed to NNSR3, while NNGR2 increased this factor slightly to 1.61. When considering the best evaluation times, the absolute times for NNGR1 and NNGR2 are 404μs and 368μs respectively. In relative terms, NNGR1 is slower by a factor of 6.26 when compared to NNSR4's longest best evaluation time, and this factor slightly reduces to 5.71 for NNGR2. When considering only the GRMHD models, NNGR1 achieves the lowest average evaluation time, and NNGR2 secures the fastest minimum evaluation time.

The plots of the L_1 and L_∞ metrics for the GRMHD models are presented in Figure 13 and Figure 14. Again the plot on the left gives the L_1 norm train errors in blue and the test errors in red, while the plot on the right gives those for the L_∞ norm. On the vertical axis, the L_1 norm ranges from 10^{-1} to 10^1, while the L_∞ norm ranges from 10^{-1} to 10^3. The horizontal axis is common between the pairs of plots and shows the number of epochs trained, ranging from 0 to 500. Notably, both NNGR models exhibit test errors consistently below the train errors, with no overshooting evident as seen in the previously described NNSR models. However, both models demonstrate an increase in metric error with errors of order 10^0 for the L_1 norm and 10^2 for the L_∞ norm, with the test errors showing of the NNGR2 models showing an increase over the number of epochs, opposite to what is desired.

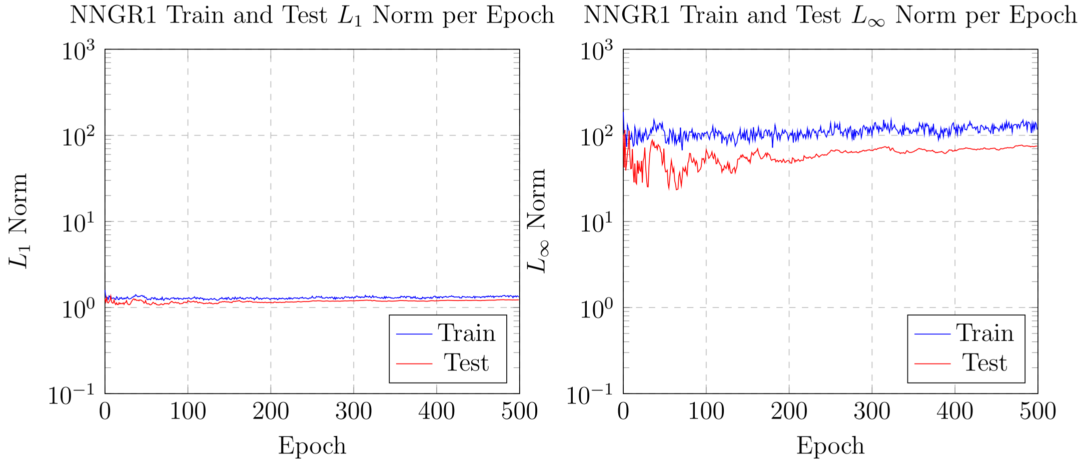


Figure 13: Training and testing error metrics for the NNGR1 model. The left plot illustrates the L_1 norm errors, and the right plot illustrates the L_∞ norm errors. Both errors are plotted as a function of the number of epochs trained, ranging from 0 to 500.

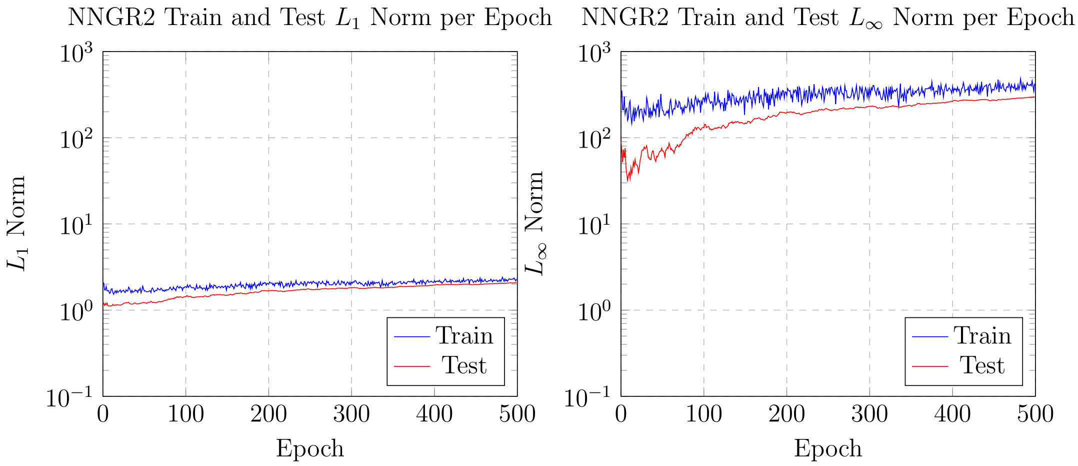


Figure 14: Training and testing error metrics for the NNGR2 model. The left plot illustrates the L_1 norm errors, and the right plot illustrates the L_∞ norm errors. Both errors are plotted as a function of the number of epochs trained, ranging from 0 to 500.

The optimization histories for the GRMHD models are depicted in Figure 15. Each plot in the figure, one for NNGR1 on the left and one for NNGR2 on the right, shows the trial-wise objective values as blue dots and the best objective value obtained, represented by an orange line, over the number of trials. There is a clear trend of decreasing objective value as the number of trials increases, suggesting successful optimization in the model training process.

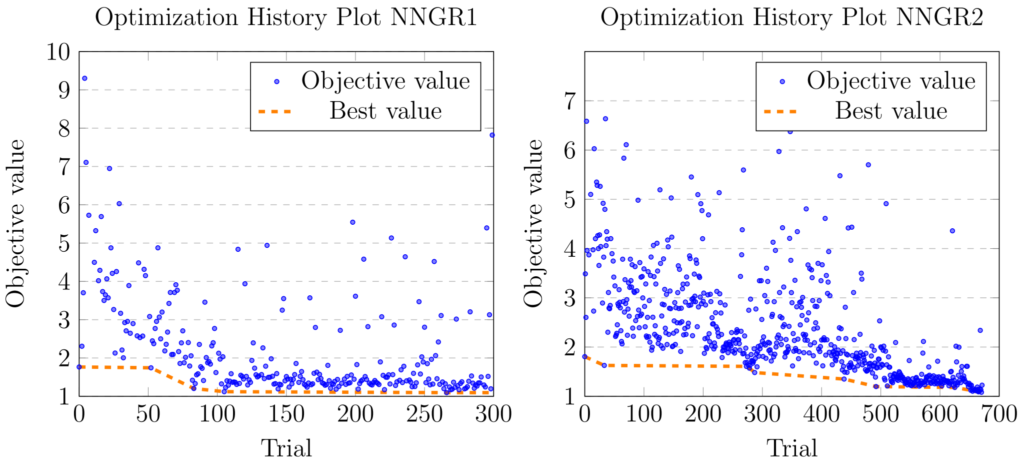


Figure 15: Optimization history plots for the GRMHD models. The left plot corresponds to NNGR1 and the right to NNGR2. Blue dots represent the trial-wise objective values, while the orange line marks the best objective value obtained over the course of the trials.

Finally, the importance of various hyperparameters for each model are illustrated in Figures 16 and 17. For NNGR1, the plot demonstrates the importance of the optimizer used, followed by the number of hidden layers used, with respective objective value fractions of 0.43 and 0.18. Subsequently, it shows the importance of the dropout rate used (0.11), scheduler used (0.10), hidden activation function used (0.07), batch size used (0.04), number of epochs used (0.04), loss function used (0.02), size of the first hidden layer used (0.01), learning rate used (< 0.01), and finally the output activation function used (< 0.01). For NNGR2, the plot similarly starts with the optimizer used (0.67), followed by the batch size used (0.10), hidden activation function used (0.05), number of epochs used (0.05), size of the first hidden layer used (0.03), learning rate used (0.02), dropout rate used (0.02), scheduler used (0.02), loss function used (< 0.01), and finally the output activation function used (< 0.01).

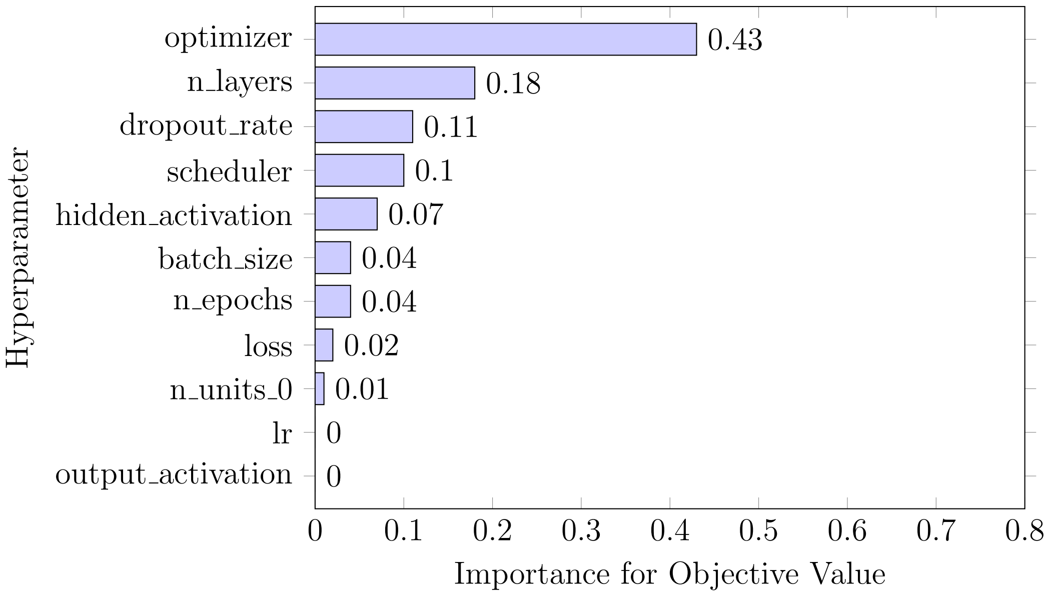


Figure 16: Relative importance of different hyperparameters for the NNGR1 model, represented by their objective value fractions. Hyperparameters include the optimizer, number of hidden layers, dropout rate, scheduler, hidden activation function, batch size, number of epochs, loss function, size of the first hidden layer, learning rate, and the output activation function.

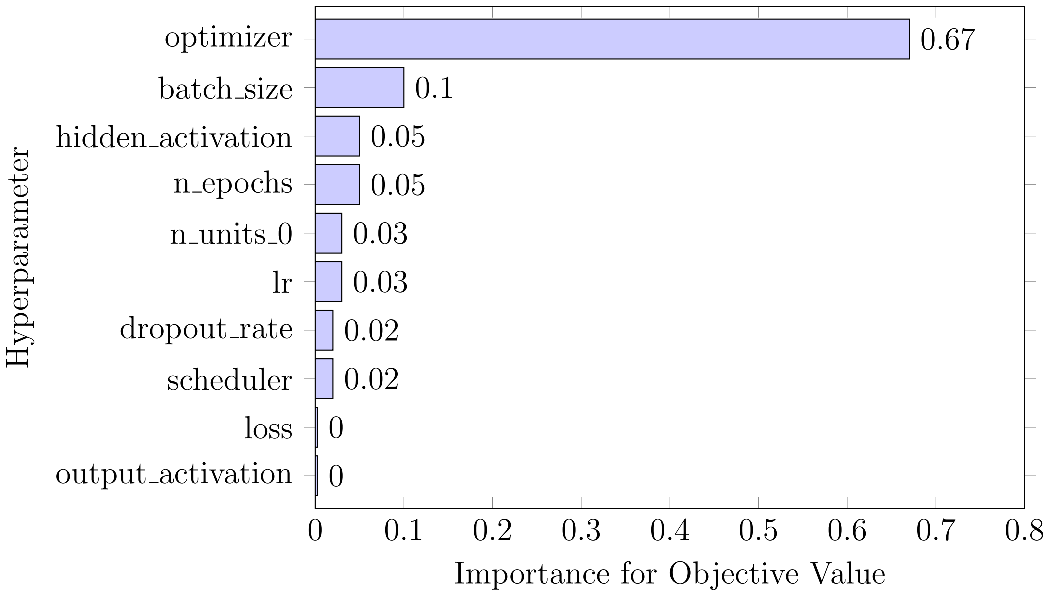


Figure 17: Relative importance of different hyperparameters for the NNGR2 model, represented by their objective value fractions. Hyperparameters include the optimizer, batch size, hidden activation function, number of epochs, size of the first hidden layer, learning rate, dropout rate, scheduler, loss function, and the output activation function.

4.2.3 Comparison between the performance of the GRMHD models to that of the root-finders

In Table 9, we present a comparison between the average and best execution times of the ANNs and the standard performance times observed in the Con2prim Interior approach for various n_{cells} values as they are implemented in GRaM-X. Con2prim Interior refers to the conventional method of conservative-to-primitive inversion implemented using root-finders, as also described in Section 2.1. The performance times in the table are expressed as fractions, where the Con2prim Interior times form the numerator and the evaluation times of the models form the denominator. This provides a means to gauge the performance of our models relative to the traditional method. As with our ANN models, the Con2prim Interior computational times have been measured on the Nvidia RTX A6000 GPU.

It is important to highlight that the NN evaluation times represent single calls to the NN, i.e. evaluations of single cells, while the times reported for the GraM-X models encompass the evaluation of the entire grid at once—i.e.

for a grid of 32, 64, 128, or 256 cells, simultaneously.

	$n_{\text{cells}} = 32$	$n_{\text{cells}} = 64$	$n_{\text{cells}} = 128$	$n_{\text{cells}} = 256$
Average fraction NNSR1	5.63	23.3	99.7	410
Best fraction NNSR1	27.0	112	478	1960
Average fraction NNSR2	6.84	28.2	121	498
Best fraction NNSR2	24.2	100	429	1760
Average fraction NNSR3	1.38	5.69	24.4	100
Best fraction NNSR3	20.2	83.5	358	1470
Average fraction NNSR4	2.03	8.38	35.9	148
Best fraction NNSR4	7.71	31.8	136	562
Average fraction NNGR1	0.904	3.73	16.0	65.9
Best fraction NNGR1	1.23	5.08	21.8	89.7
Average fraction NNGR2	0.856	3.53	15.2	62.3
Best fraction NNGR2	1.35	5.58	23.9	98.4

Table 9: Comparison of average and best evaluation times of all developed models to the `Con2prim Interior` root-finding procedure as currently implemented in GRaM-X for different grid sizes (n_{cells}). The table entries depict fractions of `Con2prim Interior` times over respective model times. Higher values denote faster model performance. All fractions are rounded to three significant figures. Evaluation times for `Con2prim Interior` were obtained from Appendix C of [1].

When we scrutinize the table, we see a diverse range in the fractions of average and fractions of best evaluation times across all models. For the fractions for the SRHD models, we observe a range from a low value of 1.38 times faster evaluation as seen in the NNSR3 model when $n_{\text{cells}} = 32$, to a maximum of 498 faster evaluation recorded for the NNSR2 model with $n_{\text{cells}} = 256$. The improvement for the GRMHD models are typically less as expected, with the fractions of the average evaluation times varying from a minimum of 0.856 for NNGR2 with $n_{\text{cells}} = 32$, to a maximum of 65.9 for NNGR1 with $n_{\text{cells}} = 256$. When we consider the best evaluation times, the range for all models extends from a low value of 7.71 faster evaluation, found in the NNSR4 model when $n_{\text{cells}} = 32$, to a high value of 1960 faster evaluation, observed for the NNSR1 model with $n_{\text{cells}} = 256$. The GRMHD models display a much narrower range, with 1.23 for NNGR1 and $n_{\text{cells}} = 32$ to 98.4 with NNGR2 and $n_{\text{cells}} = 256$.

5 Discussions

In this section, we provide main discussion points based on the obtained results as found in Section 4. We extend some of these discussions, and provide points of discussion that are of lesser importance, in Appendix D.

5.1 Discussion on the integration of the GRMHD models into GRaM-X

At an infrastructural level, the integration of the NNs into GRaM-X or another GRMHD computational environment is just a matter of exporting the models from Python to C++ (this we discussed in 3.3 and discuss in C). However, two modifications still need to be made to exported models at the level of the data generation or data import in order to make them work in such an environment. These are

1. Support for tabulated EoS.

2. Support for more than one output.

A tabulated EoS refers to an EoS where the relationship between dependent variable, pressure in our case, and its dependencies, e.g. the density and the specific internal energy, are pre-computed and stored in a table, instead of being calculated in real-time during the simulation. The inclusion of Item 3 thereby greatly improves computational efficiency and is required for computational environments such as GRaM-X to run efficiently.

In the case of GRaM-X, Item 2 is a modification that is required because hW alone, the output of the NNGR models (see Figure 5) does not resolve all the quantities that are required to do the con2prim step of the evolution (see Section 2.1 for context) without using root-finding algorithms; the con2prim step also requires the temperature T of the fluid, which is still calculated using a root-finding algorithm if it is not provided as an output from the NN. In the current implementation of GRaM-X, the modification to the models would be to provide the variables (h, W, T) as the output, i.e. we would modify the model to have not one but three outputs. However, networks with more than one output requires additional machine learning theory to be studied and implemented, such as the theory and implementation of multi-task learning, which is the predicting of multiple outcomes simultaneously. The training of the network also becomes more complex, as there would be multiple loss functions that need to be considered, one for each output, and thereby a more complex training algorithm would be required. It was for these reasons that the implementation of Item 2 was not carried out in our project.

5.2 Discussions on NN accuracy

5.2.1 On the differences in metric errors with those of Dieselhorst et al.

In Section 4.1.2 we compared the values of the metrics to those of Dieselhorst et al., and found that for the SRHD models, the errors remained to be larger than what they had obtained. Why this would be is an especially interesting question for the NNSR1 and NNSR2 models, which use the same hyper-parameters as their respective models. We consider the following possible causes and discuss them

1. Different subparameters for the optimizer or for the scheduler were used.
2. The preprocessing of the data was different.
3. A different EoS was used.
4. There are errors in the source code.

As to Item 1, Dieselhorst et al. list four of the subparameters of the scheduler; these we also use in our NNSR1 and NNSR2 models as illustrated by Table 6 and Table ??. It is possible that a subparameter was used that was undocumented. The same holds for the subparameters used for the scheduler; which subparameters were used for the scheduler was not listed at all, indicating that either these were undocumented settings or the default subparameters were used. We used the default subparameters in the NNSR1 and NNSR2 models. It is not clear how much impact specifically a difference in these subparameters can have, but in general the impact of subparameters on the NN's performance is an active field of research.

Regarding Item 2, it is possible that the data was processed differently before it was fed into the NN. One option for the preprocessing of data is to normalize the data. As discussed in Section 3.4.1, we did not do this, as it increased our errors instead of reducing them. There are a number of other preprocessing steps that we did not try out and from which a difference in the errors could result; these are e.g. to have used standardized data and to handle outliers. It is not clear from the paper of Dieselhorst et al. whether such preprocessing steps have been taken out for the data fed into their NNC2PS and NNC2PL networks.

On Item 3, it is possible that Dieselhorst et al. did not use the analytic EoS that we used (equation (37)); it was not clear from our reading of Dieselhorst et al. whether this same analytical EoS was used in their generated data, or whether a tabulated EoS was used instead.

Lastly, although it has been carefully checked for, it is possible that an unspecified mistake or multiple mistakes have been made in our source code. We did access the source code of Dieselhorst et al., and we tried to obtain their results purely from the results that they had listed; this we we did to get acquainted with implementing neural network models and as a preparation to writing the proper source code for the GRMHD models.

5.2.2 On the poor accuracy of the NNGR models

In Section 4.2.2 we compared the errors of the GRMHD models to the errors of the SRHD models (Table 7), and saw that they are one to three orders of magnitude worse. Moreover, the plots (Figure 13, Figure 14) show that the error is not improving over the number of epochs, but rather stays roughly constant, and in the case of the test errors for the NNGR2 model, even increases. We next discuss possible causes as to why these errors are so bad

1. **The training data is inadequate**
   When there is not enough training data, it is possible that the network does not have enough examples to learn from. However, in Section 3.4.1, we already mentioned how our initial trials using ten times the number of training samples did not improve the errors. As to poor

test performance specifically, it is also possible that the dataset is imbalanced or lacks diversity, which could make the model struggle to generalize. A possibility of imbalance or lack of diversity in the data we could conceive of being the case in our data as follows. The input to the network consists of conserved variables, which are computed according to equations (28), (29), (30), and (24) (see Section 3.4.5 for context) to compute these equations, primitive variables are sampled uniformly (Section 3.4.2), the latter which in turn are filtered to keep only samples for which the three conditions for the validity of the samples are met (Section 3.4.3). It is possible that through this filtering process, more samples are selected in a narrower subrange of the interval. This indeed holds true for the speed of the fluid as the sampling process tends to favor scenarios where the velocities in the x-, y-, and z-directions are significantly less than the speed of light. This is because, as one or two velocity components approach the speed of light, the remaining variable(s) must reside within a progressively narrower range to satisfy the conditions.

2. **The data needs to be preprocessed**
   For a regression problem like we have for our models, normalization or standardization of the features could help improve the performance of the model, as it can prevent the dominance of features with larger magnitudes over features with smaller magnitudes and improve the stability of training. As to the latter point, features that vary widely in magnitude can cause numerical instability during the training of the model. There are multiple orders of magnitude of variation in the input data and some of the data is heavily skewed or has extreme outliers, so the models could benefit from such preprocessing steps as normalization and standardization.

   Unless the data was particularly ill-suited for the GRMHD models, we do not expect data preprocessing on its own to reduce the errors 2 or 3 orders of magnitude to at least get them on the scale of the errors of the SRHD models; rather, it is the combination of data preprocessing with the other discussed items that could result in a substantial decrease in errors.

3. **The network architecture is poor and the hyperparameter search space needs to be narrowed-down.**
   It is likely that the neural network architectures that resulted from the hyperparameter optimization (i.e. all the hyperparameters of Table **??**) are not suitable to solve the problem of predicting hW given the 14 in-

puts (Section 3.2.2). This is likely because the neural hyperparameter search space (Table 5) is very broad, and although it is possible that even with a broad search space one obtains optimal hyperparameters, it would require lots trials to be run, which in turn requires lots of computational time. Evidently, the current number of trials run (Table 2) is not enough for the hyperparameter optimizer to find optimal hyperparameters, but, alongside running for more trials, which is not disadvantageous so long as one has the computational resources, it is a good idea to narrow down the search space, increasing the probability to obtaing optimal hyperparameters.

5.3 Discussion on the performance of the NNs compared to the performance of the root finders

We would like to discuss now the performance of the NNs compared to the performance of the root finders as they are currently implemented in GRaM-X. As was already noted in the results, a crucial point is that the evaluation times in GRaM-X, as they are found in Table 1 and as the denominator of the fractional values of Table 9, are evaluation times of the entire grid of cells, and not of a single cell. The evaluation times reported for the NNs, in contrast, correspond to the latter, single-cell evaluations. The results of Table 9 should therefore be taken with a fare degree of skepticism. Even though these results show promising indications of the superiority of NNs in terms of speed, the discrepancies in evaluation measures make it impossible to draw definite conclusions at this point. To decide conclusively what is the performance benefit of the NNs over the root finders, we have to fully port the NNs to GRaM-X. Although the models can already be ported to C++ source, they cannot yet be ported to GRaM-X to make this comparison, due to Item 1 and Item 2 of Discussion 5.1.

From the Discussion 5.1, an important point relevant to the performances of the NNs folows, namely that the complexity of the NNs for GRMHD can increase even further due to the requirement of multiple outputs as specified by Item 2. On the other hand, we had discussed in Discussion D.1.5 also that the complexity of a properly trained GRMHD model with lower errors may well be lower than that of NNGR1 and NNGR2. It is for these reasons too then that the comparison of Table 9 should not be taken as conclusive. Still, we think that the comparisons show the potential of an implementation of the Con2Prim step using an ANN to change the efficiency of the code by multiple orders of magnitude, and that project is worthy of being finished up with an implementation into the GRaM-X code once the GRMHD models have been trained with a better search space so that they have optimal parameters.

6 Conclusion

In this research project, our primary objectives were twofold: firstly, to explore the con2prim in SRHD using an ANN, and secondly, to extend this investigation to a more general case, that of GRMHD. The aim was not merely academic; we sought to practically demonstrate that a fully-connected deep feedforward ANN could reduce the computation time of con2prim in GRMHD by at least an order of magnitude, benchmarked on the Nvidia RTX A6000 GPU, and that it provides potential for integration into the GRaM-X framework.

In the first part of the project, we faced significant challenges in matching the metric errors of the SRHD models of Dieselhorst et al. when utilizing identical hyperparameters. However, this challenge was mitigated when we explored different hyperparameters, achieving errors errors of the same order of magnitude as those of Dieselhorst et al., demonstrating our success in learning how to effectively tune the network. More importantly, this initial exploration served as an invaluable foundation for designing and implementing the more complex GRMHD models.

For the GRMHD models, despite the errors not being as low as we might have desired, we successfully demonstrated a substantial reduction in computation time for con2prim, achieving up to a 100 times faster evaluation time compared to the current implementation in GRaM-X that uses root finders. Moreover, we reasoned that even faster evaluation times could be achieved as the GRMHD models can potentially be reduced in complexity with optimized hyperparameters. This signifies a major step towards the practical application of ANNs in this field. In addition, we demonstrated that the Python-based ANN models could be ported to C++ source code, which holds promise for the future integration of these models into the GRaM-X framework.

In light of these achievements, there are several prospects for future work that have opened up. The first is to integrate the GRMHD NNs into GRaM-X, thereby contributing to the reduction of computational time for con2prim inversion in real-world applications of GRMHD simulations. Secondly, there is potential for improving the accuracy of the GRMHD models, for instance, by narrowing-down the search space for the hyperparameters. Lastly, further research could focus on quantifying the relationship between the hyperparameters and the performance of the NN, which could provide insights into enhancing the performance of the NN and the overall effectiveness of this approach.

This project has not only yielded promising results but also underscored the potential of machine learning methodologies in the field of relativistic hydrodynamics. We look forward to future advancements and applications of these techniques in GRMHD, paving the way for more efficient computations and significant progress in the field, and thereby paving the way for a greater understanding of the universe.

7 Recommendations

A list of recommendations for further research into the project of implementing con2prim in GRMHD is readily compiled from the conclusion of the project, the discussions that we held and the extended discussions on topics, the latter which can be found in Appendix **??**. We keep the list simple and concise; for detailed and quantitative aspects, please review the sections and items referred to.

1. Integrate NN models into GRaM-X (see Section 5.1)
    (a) Add support for tabulated EoSs.
    (b) Add support for multiple outputs architecture.
        i. Make the necessary adjustments to the hyperparameter search space for the new architecture (e.g. Section 5.3)
2. Improve the accuracy of the GRMHD models (Section 5.2.2)
    (a) Narrow-down the hyperparameter search space (e.g. Section D.1.6)
        i. Experiment with the number of layers (Section **??**)
        ii. Experiment with the learning rate. (Item 3.5.1 of Section 3.5.1)
        iii. Experiment with more epochs per trial (e.g. Item **??** and Item 3.5.2 of Section 3.5).
        iv. Narrow-down the search spaces for the hyperparameters with the highest importance value for the objective values (Figures 16, 17 of Section 4.2.2)
        v. Experiment with all other hyperparameter search spaces (Section 3.5.2)
    (b) Run the hyperparameter optimizer with more trials (see Item 3 of Section 5.2.2)
    (c) Experiment with the input data

   i. Experiment again with increasing the input data (Item 1 of Section 5.2.2)
   ii. Evaluate the diversity of the data (Item 1 of Section 5.2.2)

(d) Apply data preprocessing techniques (Section 2) such as

   i. Normalization
   ii. Standardization
   iii. Outlier management

(e) Investigate into the test errors being consistently below the train errors (Section D.2.1)

   i. Potential data leakage
   ii. Potential underfitting, e.g. excessive dropout regularization

3. Improve the performance of the GRMHD models (Section 5.3)

   (a) Carry out Item 1 and Item 2 of this section.
   (b) Compare the accuracy of the GRMHD models to that of the root-finders (Section D.3.2)
   (c) Quantify the performance in terms of FLOPs rather than computational time (Section D.3.1).
   (d) Obtain quantitative relationships between hyperparameters and NN performance (Section D.3.1)

      i. Perform hardware specific benchmarking of the model performance D.3.1.

Acknowledgements

References

[1] Y. de Graaf, Gpu optimization of grmhd code grhydrox, Master's thesis, University of Amsterdam (2023).

[2] S. Shankar, *et al.*, Gram-x: A new gpu-accelerated dynamical spacetime grmhd code for exascale computing with the einstein toolkit, *arXiv preprint arXiv:2210.17509* (2022).

[3] P. Mösta, *et al.*, GRHydro: a new open-source general-relativistic magnetohydrodynamics code for the einstein toolkit, *Classical and Quantum Gravity* **31**, 015005 (2013).

[4] T. Goodale, *et al.*, The cactus framework and toolkit: Design and applications, *Vector and Parallel Processing – VECPAR 2002, 5th International Conference, Lecture Notes in Computer Science* **2400**, 197 (2002).

[5] F. Löffler, *et al.*, *2012 International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE, 2011), pp. 1–11.

[6] L. Antón, *et al.*, Numerical 3+ 1 general relativistic magnetohydrodynamics: a local characteristic approach, *The Astrophysical Journal* **637**, 296 (2006).

[7] R. Courant, D. Hilbert, *Methods of mathematical physics*, vol. 2 (Interscience Publishers, Inc., New York, N.Y., 1962).

[8] J. R. Shewchuk, An introduction to the conjugate gradient method without the agonizing pain, *Tech. rep.* (1996).

[9] A. Harten, P. D. Lax, B. van Leer, B. Einfeldt, Upstream differencing scheme for hyperbolic conservation laws, *SIAM Journal on Numerical Analysis* **21**, 1 (1983).

[10] E. F. Toro, *Riemann solvers and numerical methods for fluid dynamics: a practical introduction* (Springer Science & Business Media, 2009).

[11] S. C. Noble, C. F. Gammie, J. C. McKinney, L. Del Zanna, Primitive variable solvers for conservative general relativistic magnetohydrodynamics, *The Astrophysical Journal* **641**, 626 (2006).

[12] W. I. Newman, N. D. Hamlin, Primitive variable determination in conservative relativistic magnetohydrodynamic simulations, *SIAM Journal on Scientific Computing* **36**, B661 (2014).

[13] G. Servignat, J. Novak, I. Cordero-Carrión, A new formulation of general-relativistic hydrodynamic equations using primitive variables, *Classical and Quantum Gravity* **40**, 105002 (2023).

[14] J. M. Martí, E. Müller, Numerical hydrodynamics in special relativity, *Living Reviews in Relativity* **6**, 1 (2003).

[15] L. D. Landau, E. M. Lifshitz, *Fluid Mechanics: Landau and Lifshitz: Course of Theoretical Physics, Volume 6*, vol. 6 (Elsevier, 2013).

[16] C. K. Batchelor, G. K. Batchelor, *An introduction to fluid dynamics* (Cambridge university press, 1967).

[17] T. E. Faber, *Fluid dynamics for physicists* (Cambridge university press, 1995).

[18] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, 2016).

[19] W. S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, *The bulletin of mathematical biophysics* **5**, 115 (1943).

[20] T. Mitchell, *Machine Learning* (McGraw-Hill, 1997).

[21] L. Bottou, *Proceedings of COMPSTAT'2010* (Springer, 2010), pp. 177–186.

[22] K. P. Murphy, *Machine learning: a probabilistic perspective* (MIT press, 2012).

[23] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors, *nature* **323**, 533 (1986).

[24] V. Nair, G. E. Hinton, *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010), pp. 807–814.

[25] A. L. Maas, A. Y. Hannun, A. Y. Ng, *Proc. ICML* (2013), vol. 30, p. 3.

[26] K. He, X. Zhang, S. Ren, J. Sun, *Proceedings of the IEEE international conference on computer vision* (2015), pp. 1026–1034.

[27] D.-A. Clevert, T. Unterthiner, S. Hochreiter, Fast and accurate deep network learning by exponential linear units (elus), *arXiv preprint arXiv:1511.07289* (2015).

[28] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, R. Garcia, *Advances in neural information processing systems* (2001), pp. 472–478.

[29] P. Ramachandran, B. Zoph, Q. V. Le, Searching for activation functions, *arXiv preprint arXiv:1710.05941* (2017).

[30] D. Hendrycks, K. Gimpel, Bridging nonlinearities and stochastic regularizers with gaussian error linear units, *arXiv preprint arXiv:1606.08415* (2016).

[31] C. M. Bishop, *Pattern Recognition and Machine Learning* (Springer, 2006).

[32] Pytorch.

[33] T. Dieselhorst, W. Cook, S. Bernuzzi, D. Radice, Machine learning for conservative-to-primitive in relativistic hydrodynamics (2021).

[34] D. M. Siegel, P. Mösta, D. Desai, S. Wu, Recovery schemes for primitive variables in general-relativistic magnetohydrodynamics, *The Astrophysical Journal* **859**, 71 (2018).

[35] A. Labach, H. Salehinejad, S. Valaee, Survey of dropout methods for deep neural networks, *arXiv preprint arXiv:1906.11586* (2019).

[36] S. Cai, *et al.*, Effective and efficient dropout for deep convolutional neural networks, *arXiv preprint arXiv:1904.03392* (2019).

[37] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (ACM, 2019), pp. 2623–2631.

[38] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, *Advances in neural information processing systems* (2011), pp. 2546–2554.

[39] A. Paszke, *et al.*, *Advances in Neural Information Processing Systems* (2019), pp. 8026–8037.

[40] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, *The Journal of Machine Learning Research* (JMLR. org, 2014), vol. 15, pp. 1929–1958.

[41] D. Masters, C. Luschi, Revisiting small batch training for deep neural networks, *arXiv preprint arXiv:1804.07612* (2018).

[42] L. N. Smith, Cyclical learning rates for training neural networks, *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)* pp. 464–472 (2017).

[43] R. Ge, F. Huang, C. Jin, Y. Yuan, *Conference on Learning Theory* (2015), pp. 797–842.

[44] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, P. T. P. Tang, On large-batch training for deep learning: Generalization gap and sharp minima, *arXiv preprint arXiv:1609.04836* (2016).

[45] Y. LeCun, L. Bottou, G. B. Orr, K.-R. Müller, *et al.*, Neural networks: Tricks of the trade, *Springer Lecture Notes in Computer Sciences* **1524**, 6 (1998).

[46] PyTorch, torch.cuda — pytorch 1.0.0 documentation, https://pytorch.org/docs/stable/cuda.html (2023).

A Details of training and evaluation in ANNs

A.1 Training Process

The processes involved in training and evaluating our feedforward artificial neural networks for con2prim are detailed in Figure A.1. As delineated by the figure, the training process, characterized by a series of computational steps, precedes the evaluation process. Notably, these procedures are repeated iteratively for each batch of data, as illustrated by the arrows.

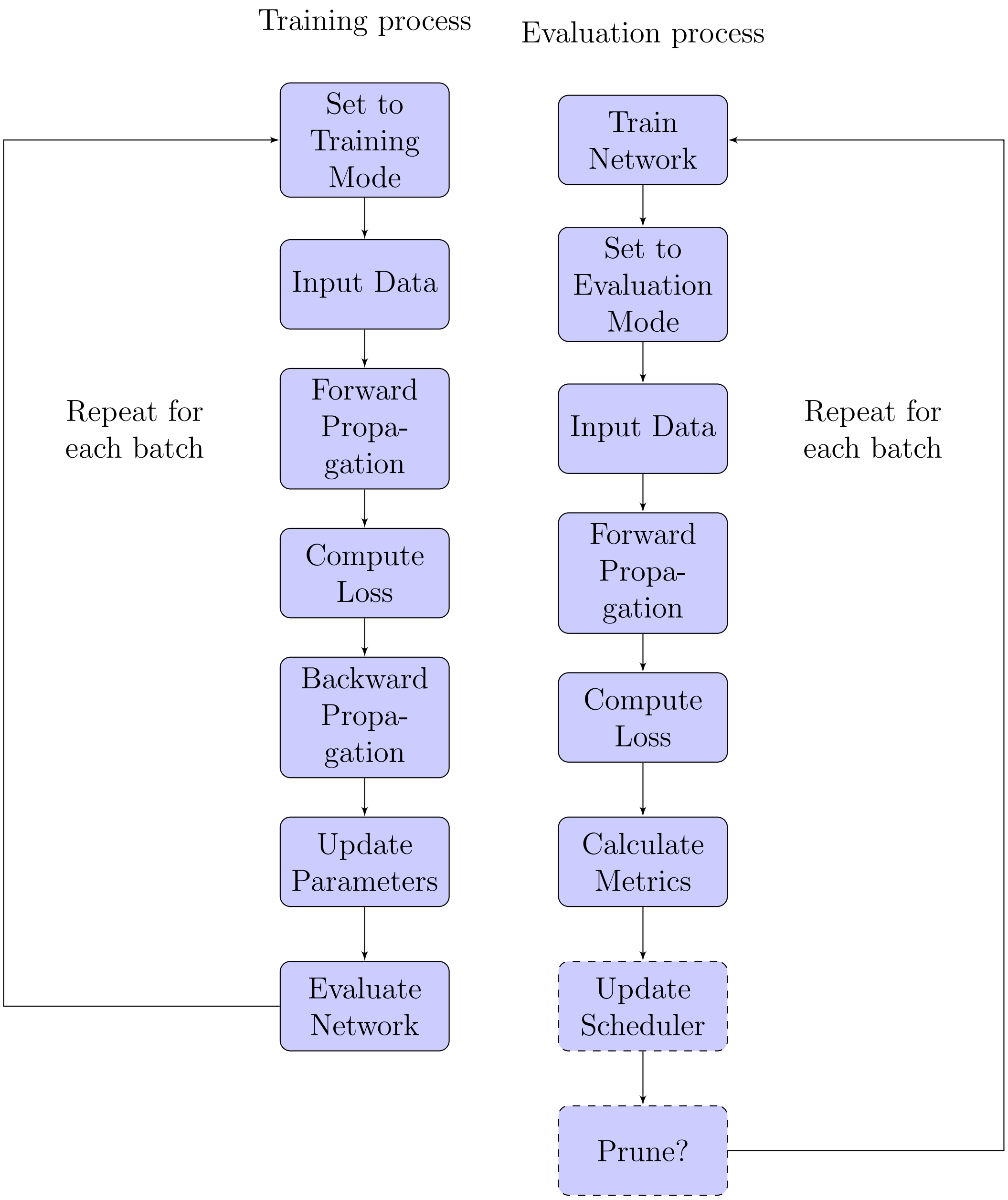


Figure 18: Detailed flowchart of the training and evaluation processes of the ANNs. The solid lines illustrate mandatory steps such as forward propagation, loss computation, backward propagation, parameter updates, and network evaluation for the training process, and forward propagation, loss computation, and metrics calculation for the evaluation process. The dashed lines represent optional steps like scheduler updates and pruning for hyperparameter optimization. Each process is iterated over each batch of data.

The training procedure commences with the network being set to `Training Mode`, enabling the network to learn and adjust its internal parameters such as weights and biases. Setting the network to training mode allows dropout layers to randomly nullify some activations to prevent overfitting, and batch normalization layers to normalize the outputs using the batch's mean and variance. Only the former is relevant to us, as dropout layers are used in our GRMHD models, while we do not perform batch normalization in any of our models. The subsequent `Input Data` node signifies the transfer of the batch to the device, such as a GPU for efficient computation. At this point, gradients from previous operations are reset to prevent accumulation across multiple forward and backward propagations.

In the step of the `Forward propagation` node, our model ingests the input data and carries out a series of computations. The main aim of these computations is to transform the input data into a form that allows the model to learn a mapping between the input and output data. The forward propagation process starts from the input layer and propagates through the hidden layers up to the output layer of the neural network. Each neuron in a layer receives an input from the neurons in the previous layer, performs a weighted sum of these inputs (the weights being the parameters we are trying to learn), applies an activation function to this sum, and passes this output to the neurons in the next layer. This process effectively transforms the initial inputs, through a series of mathematical operations governed by the model's current parameters, into a prediction output [18]. This output is the model's current prediction for the given inputs based on its current parameters. The purpose of this prediction is to compare it to the actual output during the Compute Loss stage, which allows us to see how well the model is performing and thus update its parameters. To illustrate this further, for a given set of parameters, we would expect the model's prediction to be close to the actual output for a good model. On the other hand, for a model that is not well-trained, the prediction could be quite far from the actual output. This comparison during the Compute Loss stage forms the basis for learning by updating the model's parameters to improve its prediction capability.

In the `Compute Loss` step of our training process, we evaluate the performance of our model on the batch of data processed in the forward propagation stage. This performance is evaluated using a loss function and a set of metrics. The loss function used in this context is either the mean squared error (MSE), mean absolute error (MAE), or Huber loss function, as described in the corresponding equations (equations (31), (32) or (33)) of Section 2.3.2. Firstly, to calculate the loss, we pass the batch of predicted outputs from the forward propagation and the corresponding actual outputs (.e. the labels) into the chosen loss function. The loss function quantifies the discrepancy between the model's predictions and the actual outputs, effectively giving us a measure of the current error our model is making on the batch of data. This loss value guides the subsequent updating of the model's parameters. In addition to the loss, we also compute two norm-based metrics: the L_1 and L_∞ norms. These metrics offer additional perspectives on the performance of the model. By computing the loss and these metrics, we are able to assess the performance of the model on the current batch of data, and this assessment guides the subsequent backpropagation and optimization stages where the model's parameters are updated.

In the `Back-propagation` step of our training process, we apply the backpropagation algorithm to update the parameters of our model. This step is integral to the learning capability of our model and primarily relies on the principles of the chain rule from calculus to compute gradients of the loss function with respect to the model's parameters. At a high level, backpropagation involves computing the gradient (or derivative) of the loss function with respect to the parameters of the model. In essence, this gradient represents the direction in the parameter space where the loss function increases most rapidly. Thus, by moving the parameters in the opposite direction, we aim to minimize the loss function and thereby improve the predictive performance of the model. Backpropagation operates in a reverse manner, starting from the output layer and moving towards the input layer, which explains the term *backward*. For each layer, the algorithm computes the partial derivative of the loss with respect to the parameters of the layer (weights and biases) and then proceeds to the preceding layer [23]. This operation is efficiently performed through automatic differentiation, a feature provided by modern deep learning frameworks. The result of the backpropagation process is a set of gradients, one for each parameter in the model. These gradients are used in the subsequent `Update Parameters` step to adjust the parameters in a way that reduces the loss.

In the `Update Parameters` step, we modify the model's weights and biases in a way that minimizes the computed loss. The `Evaluate Network` node then signifies the commencement of the evaluation process.

A.2 Evaluation Process

We continue with the analysis of the right chart of Figure A.1.

During hyperparameter optimization, evaluation is performed on the validation set. Otherwise, during regular training, the evaluation process is conducted using the test set. The `Train Network` node in the flowchart indicates that the preceding training process has been completed for a batch.

Setting the network to `Evaluation Mode` makes it ready to test its performance on unseen data. The mode change is crucial because some layers of the neural network, such as dropout and batch normalization layers, again, the former which is relevant to us, have different behaviors in training and evaluation modes.

In the `Input Data` node for the evaluation phase, we perform a similar task as in the training phase by transferring the data batch to the device for computation. However, unlike in the training phase, we do this operation under a different computational environment that eliminates the need for tracking gradients. Specifically, `torch.no_grad` is used to suspend the tracking of gradients for computations. This is done because during the evaluation phase, we are not interested in updating the model's parameters. We are instead assessing the model's performance on unseen data, so the gradients (which quantify how much a small change in the input will change the output) are not needed and thus not calculated. By not calculating the gradients in this process, we optimize memory usage and improve computational efficiency [18]. Tracking of gradients is disabled for any of the subsequent actions in the evaluation process as well.

During the `Forward Propagation` node of the evaluation process, we execute the forward pass on the input data batch through our model. However, the focus here differs from that of the training phase. In the training phase, the forward pass is instrumental in both loss calculation and subsequent gradient-based parameter update. It is a step towards optimizing our model to better capture the patterns in the training data. In contrast, the forward propagation during the evaluation phase does not contribute to model optimization but is utilized to evaluate the model's performance on unseen data. In this phase, we pass the data batch through the model, making use of the learned parameters without modifying them. The objective is to generate model predictions on the data batch, which are later compared with the actual targets to calculate the loss and metrics. This comparison, achieved in the next step of the evaluation phase, provides us with quantitative measures of our model's performance.

Hence, the subsequent `Compute Loss` node involves the computation of the loss and metrics between the network's predictions and the true output values. The `Calculate Metrics` node denotes the point where often additional metrics, such as accuracy, precision, recall, or area under the ROC curve, are calculated, but we did not compute such additional metrics.

When we are optimizing hyperparameters, the next step is to update the learning rate scheduler in `Update Scheduler` based on the computed validation L_1 norm. This step is omitted when we are training without hyperparameter optimization. Likewise, the `Prune?` node is a step that only happens during hyperparameter optimization, in which Optuna checks whether the current trial should be stopped because it has unpromising results.

B Subparameter search space for SRHD and GRMHD models

Table 10 lists the subparameter search space used for the NNSR4 and NNSR5 models.

Subparameter	**Search space for NNSR4 and NNSR5**
SGD weight decay	$1\times10^{-5} \leq$ Weight decay $\leq 1\times10^{-2}$ (log-uniform)
SGD momentum	$0 \leq$ Momentum ≤ 0.99
Adam weight decay	$1\times10^{-5} \leq$ Weight decay $\leq 1\times10^{-2}$ (log-uniform)
Adam beta1	$0.5 \leq$ beta1 ≤ 0.99
Adam beta2	$0.9 \leq$ beta2 ≤ 0.999
StepLR step size	$5 \leq$ Step size ≤ 15
StepLR gamma	$0.05 \leq \gamma \leq 0.5$
ExponentialLR gamma	$0.8 \leq \gamma \leq 0.99$
CosineAnnealingLR T_max	$5 \leq T_{\max} \leq 15$
ReduceLROnPlateau factor	$0.1 \leq$ Factor ≤ 0.9
ReduceLROnPlateau patience	$5 \leq$ Patience ≤ 15
ReduceLROnPlateau threshold	$1\times10^{-4} \leq$ Threshold $\leq 1\times10^{-2}$ (log-uniform)

Table 10: Subparameter search spaces for SRHD models.

Table **??** lists the subparameter search space used for the GRMHD models.

Subparameter	**Search space for GRMHD models**
SGD weight decay	$1 \times 10^{-5} \leq$ Weight decay $\leq 1 \times 10^{-2}$ (log-uniform)
SGD momentum	$0 \leq$ Momentum ≤ 0.99
Adam weight decay	$1 \times 10^{-5} \leq$ Weight decay $\leq 1 \times 10^{-2}$ (log-uniform)
Adam beta1	$0.9 \leq$ beta1 ≤ 0.999
Adam beta2	$0.999 \leq$ beta2 ≤ 0.9999
StepLR step size	$5 \leq$ Step size ≤ 15
StepLR gamma	$0.1 \leq \gamma \leq 0.5$
ExponentialLR gamma	$0.8 \leq \gamma \leq 0.99$
CosineAnnealingLR t_max_fraction	Depends on n_epochs
CosineAnnealingLR eta_min	$1 \times 10^{-7} \leq$ eta_min $\leq 1 \times 10^{-2}$ (log-uniform)
ReduceLROnPlateau factor	$0.1 \leq$ Factor ≤ 0.5
ReduceLROnPlateau patience	$5 \leq$ Patience ≤ 10
ReduceLROnPlateau threshold	$1 \times 10^{-4} \leq$ Threshold $\leq 1 \times 10^{-2}$ (log-uniform)
LeakyReLU negative_slope	$0.01 \leq$ negative_slope ≤ 0.3
PReLU init	$0.1 \leq$ init ≤ 0.3
Softplus beta	$0.5 \leq$ beta ≤ 1.5

Table 11: Subparameters search space for GRMHD models

C Installation procedure

D Extended discussions

D.1 Discussions on NN hyperparameters

D.1.1 On the use of the ReLU output activation in the SRHD models

We can question if it is required for the NNSR models to use ReLU output nonlinearity to obtain non-negative values only for the output for those models, i.e. the primitive pressure. For is it not the case that if the model has been trained well, it should output non-negative values only regardless of whether the output nonlinearity is ReLU or linear? This is indeed the case, but the use of ReLU as the output nonlinearity has several advantages.

It could be that the model is not well calibrated due to problems like that of overfitting and lack of regularization (i.e. the NNSR models do not use either weight decay or dropout layers), or if the model complexity is not appropriate for the data. In these cases, the use of ReLU as output nonlinearity would ensure that predictions never go below zero. Another advantage is that ReLU output nonlinearity can help prevent the problem of exploding gradients. Exploding gradients are a problem where the gradient becomes too large, which can lead to unstable learning updates and divergence in the learning process, which typically happens through repeated multiplication by values greater than 1 in the process of backpropagation. ReLU can help with this problem because its derivative is either 0, for negative inputs or 1 for positive inputs, meaning that during backpropagation the gradients are not scaled up by the activation function but either set to zero or passed through as they are. A potential problem of using ReLU output nonlinearity is the problem of dead neurons, the problem that if a neuron's weights get updated such that the weighted sum of the neuron's input is negative, the output of the neuron will be zero, and if this condition persists through many iterations, that neuron essentially stops learning—the neuron dies. Dead neurons can slow down learning or lead to suboptimal solutions. Another downside is that no weight updates will occur when the input to the ReLU is negative, which could slow down learning.

D.1.2 On the unconventionality of using Sigmoid hidden activation in the SRHD models

The use of the Sigmoid activation function (Figure 2.3.1) as used by NNSR1 and NNSR2 is uncommon because it suffers from a number of problems, such as vanishing gradient, a problem where the gradient becomes very small during the backpropagation process, which slows down learning, and the problem of saturation of neurons, in which neurons go into a state in which they are unlikely to change or adapt, affecting the model's ability to learn, when the input is too positive or too negative. Again, the use of Sigmoid hidden activation turned out to work best for Dieselhorst et al., but we abstained from its use in the NNSR3, NNSR4 models, obtaining better errors using ReLU hidden activation(Table 6, Table 6).

D.1.3 On why lower errors are obtained for the NNSR3 and NNSR4 models then for the NNSR1 and NNSR2 models

One can wonder why it is that we obtain lower errors on the SRHD models using hyperparameters different from those of Dieseldorst et al.; who found optimal parameters for the SRHD con2prim task. The answer to this question is that there can be multiple optimal sets of optimal parameters. Although theoretically one strives to find the global minimum of the loss function used, in practice only local minima are achieved by models, because the loss function for deep learning models is often has high-dimensionality, and besides, one often wants to find a minimum that is good enough, given the resources. Given the latter, it is therefore not strange that our NNSR3 and NNSR4 models can achieve lower errors than those found for NNSR1 and NSNR2 too.

D.1.4 On the value of the `min_lr` subparameter for the SRHD models

The `min_lr` subparameter (Table 6, Table **??**) is a subparameter that is not optimized (cf. Table 10); there is no particular reason why it has not been. The value of 0 is the default, and the value of 1×10^{-6} for NNSR1 and NNSR2 was chosen as an educated guess that the learning rate should not ever go beyond this value. For the GRMHD models we just decided to use the default value of 0, and so it is not included in the table of the models (Table 8).

D.1.5 On the number of layers for the GRMHD models

We see that both GRMHD models have a number of 5 hidden layers (Table 8). This could indicate that it may be instructive to expand the search space (Table 5) beyond 5 as the upper bound for the the number of of hidden layers. On the other hand, by the universal approximation theorem, any regression problem like that of con2prim in GRMHD can be solved by a neural network with one hidden layer in theory [**?**], and it is still instructive to explore the space of low hidden layers sizes, as this decreases the evaluation time of the NN. Despite the badness of the errors we can see from the optimization histories (Figure 15) that there is improvement in lowering the objective values over the hyperparameter optimization trials. However, as the best value line in orange shows, the improvements stagnates, and this suggests that it could be beneficial to modify the hyperparameter search space as discussed.

Finally, the hyperparameter importances (Figure 16, Figure 17) also give an indication as to what is a good direction to go into for narrowing-down the search space.

D.1.6 On the similarities between the hyperparameters of the GRMHD models

The similarities between the hyperparameters of the GRMHD models is noteworthy (Table 8). That the number of hidden layers, the hidden activation, the loss function, the optimizer and the batch size (recall that the output activation was set to be always linear; Table **??**) are the same could indicate that these are hyperparameters that are optimal, but such a statement cannot be justified using just two models; for one, the models used the same search space, and if the search space was suboptimal to begin with, then both models simply use a common set of suboptimal hyperparameters.

D.2 More discussions on NN accuracy

D.2.1 On the test errors being lower than the train errors

We saw in Section 4.1.2 and Section 4.2.2 that the test errors remain below the training errors for all models except for occasional spikes where they overshoot in the NNSR models. The overshooting phenomenon underscores the complexity of the learning process and the potential for sudden increases in model errors, possibly due to shifts in the parameter space during training.

It is unusual for the test error to always be below the train error, because because the model is trained to minimize the error on the training set. Note the difference of this with using the test metric to evaluate the performance of the network; see Section 3.6 for the details. We expected the test errors to be below the train errors for at the NNSR1 and NNSR2 models, as Dieselhorst et al. explicitly states that this is what they had for the NNC2PS and NNC2PL models. Incidentally, they also give the test errors always being below the train errors as a justification that overfitting did not occur in the their tests, and that the implementation of regularization methods like dropout would not improve the training [33]. This provided the rationale for us to not use regularization methods for any of the NNSR models. Dieselhorst et al. do not provide a reasoning as to why it is valid that their test errors are always below their train errors.

Setting aside the phenomena of the errors for the NNSR models, for the NNGR models there is no intrinsic reason for the test error to always be below the train errors, and it could indicate a number of potential issues. It could be a data leakage issue, where some information from the test set is inadvertently being used during training. It could also be that the model is underfitting the training data, meaning it is not complex enough to capture the patterns in the training data. Finally it could mean that due to a high regularization term, i.e. dropout rate or weight decay, the model might underperform on the training set but perform well on the test set. Since we did not set any weight decay with the Adagrad optimizer used for the GRMHD models (see Table 11 for the subparameter search space of the GRMHD models), the final potential issue could only be due to the dropout rates of 0.286 and 0.467 (Table 8) being too high.

D.2.2 On the spikes that appear in the test error for the SRHD models

We noted in Section 4.1.2 that in the plots of SRHD models (Figures 10, **??**, 11), the test errors are always below the train errors except for the occasional spikes in the about the 200 epochs; after which the spikes disappear. It is of interest to ask the question: what causes these early spikes? The following (non-mutually exclusive) causes can be reasoned

1. The initial learning rate is high
2. The scheduler modifies the learning rate at non-ideal times
3. The batch size is high

The causes are reasoned as follows. As to Item 1, A high learning rate can cause drastic updates to the weights, which can sometimes result in temporary high errors. This is especially the case if the model is still in the early stages of training where the weights have not yet started to converge. As to Item 2, The learning rate schedulers reduce the learning rate based on certain conditions (conditions which we discuss in Section 3.5.1), if the learning rate is reduced too quickly or at an inappropriate time, it could cause temporary spikes in the test error. As to Item 3, small batch sizes can lead to noisy gradient estimates, which can result in oscillations in the training process. It is because of these reasons or a combinations of them that we expect the test error to occasionally spike above the train error.

D.3 More discussions on neural network performance

D.3.1 On the increase in evaluation times between the SRHD and the GRMHD models

Table **??** shows an increase in evaluation times for the GRMHD NNs compared to the SRHD NNs. The minimum factor difference in average evaluation time between the SRHD and the GRMHD models is about 1.5, and the maximum factor difference in evaluation time is about 8. Likewise, the minimum factor difference in best evaluation time is about 6, while the maximum factor difference in evalution time is about 22. The pivotal question is: what increases the evaluation time between these models? The following are determinants that are associated with increases in evaluation time for our problem

1. The number of trainable parameters
   (a) The number of hidden layers
   (b) The number of units per layer
      i. The number of inputs
      ii. The number of units per hidden layer
2. The activation function used
   (a) The activation function's subparameters.

To discuss why these determinants increase the evaluation times, it is more convenient to talk in terms of floating-point operations (FLOPs). In a feed-forward neural network, the basic operations include matrix multiplications, additions, and the application of activation functions. These operations are performed for each layer from input to output during the forward pass, and each operation is counted towards the total FLOPs. The number of these operations increases as the size of the matrices increase, which in turn is determined by the number of units (parameters) in each layer. An increase in FLOPs results in an increase in evaluation time, and so an increase in the number of trainable parameters results in an increase in the evaluation time. It also follows that we can see FLOPs as a measure of the complexity of the model.

As to the activation functions, more complex mathematical functions require more FLOPs, and thereby increase the evaluation time. Furthermore, subparameters of the activation function, such as PReLU's `init` parameter, can add to the number of FLOPs during evaluation because they are able to introduce additional training parameters.

It is hard to quantify exactly the relations between these determinants and the increases in evaluation timeThe relationships depend on the hardware and its capabilities, e.g. the GPU's ability to execute operations in parallel. Hence, knowing how much each of the aforementioned determinants increases the evaluation times would require hardware specific benchmarking.

D.3.2 On the accuracy of the root-finding implementation of `Con2prim Interior`

The results lack a comparison between the errors that we obtained for the models (Table **??**) and those of Con2prim Interior, the root-finding implementation as it is currently found in GRaM-X (see Section 2.1). At the time of writing, there is no source available that lists the errors in the Con2prim step as implemented in GRaM-X that we can use for comparison. However, this was not a problem to our project, as we were interested in the the temporal advantage of the an ANN implementation of con2prim rather than the advantage in accuracy (Section 3.1).