Optimising Neural Fractional Differential Equations for Performance and Efficiency

Bernd Zimmering [1], Cecília Coelho [2] Oliver Niggemann [1]
[1] *Institute for Artificial Intelligence (HSU-AI), Helmut-Schmidt-University Hamburg*
[2] *Centre of Mathematics (CMAT), University of Minho*
BERND.ZIMMERING@HSU-HH.DE, CMARTINS@CMAT.UMINHO.PT, OLIVER.NIGGEMANN@HSU-HH.DE

Editors: Cecília Coelho, Bernd Zimmering, M. Fernanda P. Costa, Luís L. Ferrás, Oliver Niggemann

Abstract

Neural Ordinary Differential Equations (NODEs) are well-established architectures that fit an ODE, modelled by a neural network (NN), to data, effectively modelling complex dynamical systems. Recently, Neural Fractional Differential Equations (NFDEs) were proposed, inspired by NODEs, to incorporate non-integer order differential equations, capturing memory effects and long-range dependencies. In this work, we present an optimised implementation of the NFDE solver, achieving up to 570 times faster computations and up to 79 times higher accuracy. Additionally, the solver supports efficient multidimensional computations and batch processing. Furthermore, we enhance the experimental design to ensure a fair comparison of NODEs and NFDEs by implementing rigorous hyperparameter tuning and using consistent numerical methods. Our results demonstrate that for systems exhibiting fractional dynamics, NFDEs significantly outperform NODEs, particularly in extrapolation tasks on unseen time horizons. Although NODEs can learn fractional dynamics when time is included as a feature to the NN, they encounter difficulties in extrapolation due to reliance on explicit time dependence. The code is available at https://github.com/zimmer-ing/neural-fde.

Keywords: Neural Networks, Cyber-Physical Systems, Neural Ordinary Differential Equation, Neural Fractional Differential Equation, Benchmark

1. Introduction

In various scientific and engineering disciplines, it is fundamental to model the behaviour of processes over time. This is crucial in fields such as epidemiology (Grassly and Fraser, 2008), economics (Mircea et al., 2014), and materials science (Zhang et al., 2016), among others.

While traditional approaches often use integer order differential equations, such as Ordinary Differential Equations (ODEs), to model systems' dynamics, these might not always be the best choice. Non-integer order differential equations, also known as Fractional Differential Equations (FDEs), have gained increasing attention in recent years due to their ability to capture complex behaviours and memory effects in various systems (Akgül and Khoshnaw, 2020). Unlike integer-order derivatives, they can describe phenomena that exhibit exponential relations, long-range interactions, and anomalous diffusion, which are often observed in fields such as physics (Abdou, 2017; Uchaikin, 2003), engineering (Goodwine, 2014; Ge et al., 2015), and finance (Ma et al., 2019; Aljethi and Kılıçman, 2023).

While differential equations are powerful mathematical modelling tools, the process of designing and adjusting these models is time-consuming and difficult, relying heavily on

field experts. Recently, neural networks (NNs) have demonstrated their ability to model systems from data, addressing these problems and becoming a go-to solution for a wide range of fields (Faroughi et al., 2022; Pavlidis et al., 2006; Grebner et al., 2021). Given the importance of differential equations to science and engineering, Chen et al. (2018) were the first to propose a NN that models ODEs from data, called Neural Ordinary Differential Equations (NODEs). This innovation revolutionised the NN paradigm, as NODEs use ODE solvers that integrate over the time axis, providing a continuous representation. This allows NODEs to handle irregularly sampled data and make predictions across the entire time domain, independent of the sampling rate. The popularity of NODEs has inspired several works proposing NNs that model other types of differential equations (Zhu et al., 2021; Liu et al., 2019), and the exploration of continuous representations like Neural Laplace (Holt et al., 2022) and Neural Flows (Biloš et al., 2021), which are based on mathematical principles also often used in engineering (Zimmering and Niggemann, 2024). Furthermore, NODEs are now a well-established architecture with several applications that can be found in the literature (Meleshkova et al., 2021; Sorourifar et al., 2023; Bräm et al., 2024; Bonnaffé et al., 2021).

One of the most recent proposals in this context, are Neural Fractional Differential Equations (NFDEs) (Coelho et al., 2024), an architecture that leverages FDEs' memory and long-range dependency capabilities to improve modelling performance. Since NFDEs are in their infancy and have demonstrated theoretical and experimental potential, this paper proposes further investigation into their capabilities.

Previous work by Coelho et al. (2024) highlighted the promise of NFDEs but also revealed several areas for improvement. The experiments were conducted using standard hyperparameters and different solver types for NODE and NFDE, which might not provide a fair comparison. Additionally, the solver used was not batch-capable, supported only one-dimensional data, and was relatively slow. To fully understand and optimise these algorithms, a more efficient solver and a comprehensive empirical methodology are necessary. This approach ensures fair comparisons and reduces the potential for author bias.

Systems with memory effects, where past states influence current dynamics, are effectively captured by NFDEs due to their ability to model long-term dependencies without requiring time as an explicit input. This makes NFDEs particularly well-suited for systems with fractional dynamics, which involve complex temporal behaviour. Neural ODEs, in principle, could also capture fractional dynamics if time is included as a feature to the neural network. However, it remains an open question whether this ability depends on the explicit availability of time as a feature, and whether it introduces limitations, especially in tasks like extrapolation, where generalisation to unseen time horizons may suffer. This motivates our investigation into how these architectures handle time dependencies in systems with fractional dynamics.

To address these gaps, we propose the following research questions:

- **RQ1:** Is there a difference in performance between NODE and NFDE when incorporating time as a feature to the NN?
- **RQ2:** Can the performance and computational cost of NFDE benefit from a faster, batch-capable numerical method implementation?

- **RQ3:** How can we ensure a fair and accurate performance comparison between NFDE and NODE, using comprehensive methods such as hyperparameter tuning?

These research questions guide our investigation, aiming to enhance the understanding and application of NFDEs in modelling complex systems. Our study includes the development of an improved numerical method, a thorough analysis of hyperparameter impact, and a detailed performance comparison between NODEs and NFDEs.[1]

This paper is structured as follows: In Section 2 a brief motivation and introduction to FDEs is given followed by an overview of NODEs and NFDEs. Section 3 details the implementation improvements of the solver presented in Coelho et al. (2024) and presents our method to fairly compare NFDE and NODE. Section 4 presents the experimental details and numerical results for the systems in study. The paper ends with the conclusions and future work in Section 5. Additionally, a comprehensive appendix provides extensive results and further details on the experiments conducted.

2. Background and Related work

2.1. Transitioning from ODEs to FDEs

To motivate the transition from integer-order to non-integer-order differential equations, consider the example of exponential growth. In the case of integer-order derivatives, the growth rate of a function is directly proportional to its value at a given time $t \in \mathbb{R}^+$:

$$\frac{d^n y(t)}{dt^n} = \lambda y(t), \quad y(0) = y_0 \tag{1}$$

where $n \in \mathbb{N}^+$ is the derivative order and $\lambda \in \mathbb{R}$ is the growth rate. The solution to this differential equation for $n = 1$ leads to the well-known exponential function, which exhibits a constant growth rate and a memory-less property:

$$y(t) = y_0 e^{\lambda t} \tag{2}$$

where $y_0 \in \mathbb{R}$ is known as the initial state. However, in real-world scenarios, growth rates may not be constant and depend on the history of the system.

By introducing non-integer order derivatives, one can capture more complex growth patterns that exhibit both power-law dependencies and memory effects. The non-integer order derivative of a growth rate function is then given by:

$${}_0^C D_t^\alpha y(t) = \lambda y(t), \quad y(0) = y_0 \tag{3}$$

where α is the non-integer order of the FDE the definition of fractional derivative according to Caputo is ${}_0^C D_t^\alpha f(t) = \frac{1}{\Gamma(\lceil\alpha\rceil - \alpha)} \int_0^t (t-\tau)^{\lceil\alpha\rceil - \alpha - 1} f^{(\lceil\alpha\rceil)}(\tau) d\tau$.[2] Here $\lceil\cdot\rceil$ is the ceiling function and $\Gamma(n) = (n-1)!$ is the gamma function (Diethelm, 2010). This can model growth rates that gradually change over time, allowing for a more realistic representation of various phenomena such as population dynamics, tumour growth, and financial market trends.

1. All implementations can be found at https://github.com/zimmer-ing/Neural-FDE.
2. Please note that for $0 < \alpha < 1$ the equation reduces to ${}_0^C D_t^\alpha y(t) = \frac{1}{\Gamma(1-\alpha)} \int_0^t (t-\tau)^{-\alpha} \left[\frac{d}{d\tau} f(\tau)\right] d\tau$

One important advantage of FDEs is their ability to describe sub-exponential growth, referring to systems that show growth rates that are slower than exponential but still exhibiting power-law dependencies:

$$ {}_0^C D_t^\alpha y(t) = \lambda y(t), \text{ with } \alpha \in (0,1). \tag{4} $$

The solution of (4) is:

$$ y(t) = y_0 E_\alpha(\lambda t^\alpha), \tag{5} $$

where $E_\alpha(z)$ is the Mittag-Leffler function, which is a generalisation of the well-known exponential function, $E_1(z) = e^z$ (Diethelm, 2010).

2.2. Neural Approaches to Learning ODEs and FDEs

NODEs (Chen et al., 2018) and NFDEs (Coelho et al., 2024) are sophisticated frameworks designed to model complex dynamical systems by leveraging NNs to approximate differential equations from data. Both methodologies share a fundamental conceptual foundation, using NNs to model system dynamics and employing numerical solvers to integrate these dynamics over time.

NODEs approximate the right-hand side of an ODE with a NN:

$$ \frac{d\boldsymbol{y}(t)}{dt} = \boldsymbol{f_\theta}(\boldsymbol{y}(t), t), \quad \text{with} \quad \boldsymbol{y}(t_0) = \boldsymbol{y}_0, \tag{6} $$

where $\boldsymbol{f_\theta} : \mathbb{R}^{m+1} \to \mathbb{R}^m$ is a NN parameterised by $\boldsymbol{\theta} \in \mathbb{R}^p$, and $\boldsymbol{y}_0 \in \mathbb{R}^m$ is the initial condition (Chen et al., 2018). Here $m \in \mathbb{N}^+$ represents the number of features and $p \in \mathbb{N}^+$ number of (trainable) parameters. The solution to the initial value problem (IVP) for NODEs can be expressed as:

$$ \boldsymbol{y}(t) = \boldsymbol{y}(t-\epsilon) + \int_{t-\epsilon}^{t} \boldsymbol{f_\theta}(\boldsymbol{y}(\tau), \tau) d\tau, \tag{7} $$

where $(t-\epsilon)$, with $\epsilon \in \mathbb{R}^+$, is an arbitrary time step back in the trajectory's past, used to compute the current value $\boldsymbol{y}(t)$. The integral is approximated by an ODE solver, which is implemented in `pytorch` and supports backpropagation (Chen, 2018). Equation (7) illustrates that future states depend solely on the immediate past state $\boldsymbol{y}(t-\epsilon)$, such as the initial value $\boldsymbol{y}_0$ or the solution from a previous instant in the numerical procedure.

NFDEs extend the NODE framework to FDEs, enabling the capture of more intricate and history-dependent dynamics. For NFDEs, the IVP is equivalent to the Volterra integral equation, particularly when $\alpha < 1$ (Diethelm et al., 2002):

$$ \boldsymbol{y}(t) = \boldsymbol{y}(0) + \frac{1}{\Gamma(\alpha)} \int_0^t (t-\tau)^{\alpha-1} \boldsymbol{f_\theta}(\boldsymbol{y}(\tau), \tau) \mathrm{d}\tau, \tag{8} $$

where $\Gamma(\alpha)$ denotes the gamma function, and $\boldsymbol{f_\theta}$ is the same trainable NN as in (6). Unlike Equation (7), the numerical solver for NFDEs must compute the entire integral from 0 to t for each $\boldsymbol{y}(t)$, reflecting the fractional, history-dependent nature of the system.

Additionally, NFDE is capable of learning the α value from data, thus using a second NN with parameters $\boldsymbol{\phi}$, $\alpha_{\boldsymbol{\phi}}$. However, in this work the α value will instead be optimised using hyperparameter search and its value fixed during the training process of NFDE.

3. Methods

3.1. Advanced Techniques for NFDE Solving

Coelho et al. (2024) introduced an implementation of the Predictor-Corrector (PC) method by Diethelm et al. (2002), which generalizes the Adams–Bashforth–Moulton algorithm (Hairer et al., 1993; Hairer and Wanner, 1996) known for solving first-order ODE problems. This implementation supports backpropagation, enabling the modelling of an FDE using a neural network (NN) to fit the data. While effective, their implementation can be further optimised in terms of computational efficiency and scalability. Specifically, operations such as the summations in the predictor and corrector steps were computed explicitly in loops, and batch processing was not fully leveraged, limiting the ability to exploit parallel computation available in frameworks like `pytorch`. Additionally, the NN evaluations were recomputed multiple times without reusing intermediate results, leading to increased computational cost. Our approach addresses these limitations by precomputing as much as possible, using `pytorch`'s parallelisation capabilities, supporting batch processing, and minimising redundant evaluations of the neural network.

The PC method follows a multi-step approach, where it first calculates an approximation of the next value $\boldsymbol{y}_{\text{pred}} \in \mathbb{R}^m$(predictor step) and then uses it to compute the next value of the solution $\boldsymbol{y}_j \in \mathbb{R}^m$ (corrector step). Diethelm et al. (2002) applied this method to numerically solve Equation (8), resulting in the predictor equation (9) and the corrector equation (10):

$$\boldsymbol{y}_{\text{pred}} = \boldsymbol{y}_0 + \frac{h^\alpha}{\Gamma(\alpha+1)} \sum_{k=0}^{j-1} b_{j-k} \boldsymbol{f}(\boldsymbol{y}_k, kh) \tag{9}$$

$$\begin{aligned} \boldsymbol{y}_j = \boldsymbol{y}_0 + \frac{h^\alpha}{\Gamma(\alpha+2)} \Big(& \boldsymbol{f}(\boldsymbol{y}_{\text{pred}}, jh) + \left((j-1)^{\alpha+1} - (j-1-\alpha)j^\alpha\right) \boldsymbol{f}(\boldsymbol{y}_0, 0) \\ & + \sum_{k=1}^{j-1} a_{j-k} \boldsymbol{f}(\boldsymbol{y}_k, kh) \Big) \end{aligned} \tag{10}$$

Here, $\boldsymbol{f}(\boldsymbol{y}, t) : \mathbb{R}^{m+1} \to \mathbb{R}^m$ represents the FDE (i.e. the NN in our case), $h \in \mathbb{R}^+$ denotes the fixed step size, and b_{j-k} and a_{j-k} are components from the vectors $\boldsymbol{b} \in \mathbb{R}^j$ and $\boldsymbol{a} \in \mathbb{R}^j$, respectively, which are calculated based on α (refer to Equations 12 and 14 in Diethelm et al. (2002)). In these equations, k is the index of the time-step, and $\boldsymbol{y}_k$ represents the solution at the k-th time-step. The index $j \in \mathbb{N}^+$ refers to the current time-step, and the sums run over $k = 0, 1, \ldots, j-1$, leveraging the historical values up to the current time $j \cdot h$.

In machine learning contexts, directly implementing these equations can be computationally intensive. Evaluating $\boldsymbol{f}(\boldsymbol{y}_k, kh)$ is resource-demanding, and as j increases, the number of terms in the sums of Equations (9) and (10) grows linearly with j. Without optimisation, this leads to redundant computations of $\boldsymbol{f}(\boldsymbol{y}_k, kh)$ for the same k across different time-steps j, since these values are needed repeatedly. Additionally, the computational graph used for automatic differentiation must retain all operations, increasing memory usage and slowing down performance (Baydin et al., 2017).

To address this inefficiency, we optimise the implementation by caching the evaluations of $\boldsymbol{f}(\boldsymbol{y}_k, kh)$ for each k. Once $\boldsymbol{f}(\boldsymbol{y}_k, kh)$ is computed at time-step k, it is stored and reused

in all subsequent time-steps $j > k$. Consequently, for each new time-step j, we only need to compute two new evaluations of the NN: one for $\boldsymbol{f}(\boldsymbol{y}_k, kh)$ with $k = j - 1$ in the predictor step (9), the corrector step (10), and one for $\boldsymbol{f}(\boldsymbol{y}_p, jh)$ in the corrector step (10). The term $\boldsymbol{f}(\boldsymbol{y}_0, 0)$ can be precomputed at the beginning. By storing and reusing these function evaluations, we avoid redundant computations, significantly reducing the computational cost. Additionally, we leverage `pytorch`'s parallelisation capabilities and support batch processing, which enhances performance by using the parallel processing power of GPUs. Our full pseudocode can be found in Appendix A.

3.2. Enhancing Experimental Design

The experiments conducted by Coelho et al. (2024) demonstrated the superior properties of NFDE for modelling sub-exponential growth. These experiments included three independent training sessions with different random seeds. However, manual selection of hyperparameters – such as the number of layers and neurons, activation functions, and learning rates – can introduce bias and be a tedious "trial and error" process. Extensive research underscores that hyperparameter choices critically impact algorithm performance (Krauß, 2022). Furthermore, empirical studies on machine learning algorithms should meet standard requirements, including a rigorous tuning methodology, such as grid search, random search, or Bayesian optimisation (Sculley et al., 2018; Vranješ and Niggemann, 2024).

In addition to the standard hyperparameters, the choice of the numerical solver plays a critical role in NODE performance (Chu et al., 2024). In classical ODE simulations, the choice of the ODE solver can significantly affect performance (Städter et al., 2021). This holds true for NODEs as well, since the solver influences the dynamical properties that the NN can learn (Westny et al., 2023). Coelho et al. (2024) used the default solver from `torchdiffeq`, the Dormand–Prince 5 (DOPRI5), which is a widely used single-step, adaptive solver suitable for non-stiff ODEs. In contrast, the NFDE solver is based on the Adams–Bashforth–Moulton method, a fixed-step, multi-step solver that achieves higher efficiency by using results from previous time steps in its calculations. To ensure a fair comparison, we also employ the Adams–Bashforth–Moulton solver for NODE in this work.

To mitigate bias and ensure a robust comparison between NODE and NFDE, we conduct 1000 hyperparameter trials per run using the Tree-Parzen-Estimator (TPE) (Ozaki et al., 2020) in a multidimensional search space. TPE was selected for its efficiency in handling high-dimensional optimisation (Valsecchi et al., 2021). We perform multiple runs, each with a distinct seed assigned to both the training and hyperparameter tuning processes. This setup, combined with 1000 hyperparameter trials per run, ensures thorough exploration of the search space and robust model evaluation.

Each set of hyperparameters is evaluated using a validation dataset, ensuring that the test dataset is not involved in either training or hyperparameter optimisation. This practice is crucial to avoid overfitting and to provide an unbiased assessment of model performance. Specifically, we calculate the mean and standard deviation of the performance metrics across different runs, providing insights into the stability and reliability of the models under slightly different hyperparameter sets.

4. Numerical Experiments and Results

4.1. Evaluating Solver Efficiency and Accuracy

Since the numerical method implementation in Coelho et al. (2024), referred to as original, was improved with the main goal of reducing computational cost, this optimised version was validated by comparison with the FDE ${}_0^C D_t^\alpha y(t) + y(t) = 0$, $t \in (0, T]$, $\alpha \in (0, 1)$, with initial condition $y_0 = 1$, and analytical solution given by $y(t) = E_{\alpha,1}(-t^\alpha)$, where $E_{\alpha,\gamma}(z) = \sum_{k=0}^{\infty} \frac{z^k}{\Gamma(\alpha k+\gamma)}, \alpha, \gamma > 0$ is the Mittag-Leffler function with two parameters (Liu et al., 2018). We used an `python` implementation of the Mittag-Leffler function that is based on Garrappa (2015).

To validate and compare the performance and computational costs of original and the herein proposed optimised version, $\alpha = 0.6$ and four different step sizes, $\Delta t \in \{1, 0.1, 0.01, 0.05\}$, over the time interval $[0, 10]$. Our choice of α is typically considered low for FDEs and indicates a *strong* fractional behaviour, thereby challenging the accuracy of the solver.

To ensure high precision, we utilise the `float64` data type for all computations. We use a batch size of one, to fairly compare the solvers as the original implementation would have to process a batch of samples sequentially and would thus be much slower. For each numerical method implementation five independent runs [3] were done to measure elapsed time in seconds, memory consumption in megabytes (MB) [4] and the difference between numerical and analytical solution (Error). The mean results are shown in Table 1.

Experiments were conducted on an Intel(R) Core(TM) i9-10900KF CPU, featuring 10 cores and 20 threads, with a base clock speed of 3.70 GHz and a maximum turbo frequency of 5.30 GHz. This processor supports both 32-bit and 64-bit operation modes.

Table 1: Mean performance of original and optimised solver over five runs.

Δt	Time (s)		Memory (MB)		Error	
	Original	Optimised	Original	Optimised	Original	Optimised
0.005	185.934	0.325	15811	15678	1.28e-08	1.62e-10
0.010	46.166	0.157	4305	3588	5.77e-08	1.27e-09
0.100	0.489	0.016	506	500	9.91e-06	1.03e-06
1.000	0.008	0.003	461	460	6.43e-03	1.03e-03

The results in Table 1 show substantial improvements with the optimised solver. It is approximately 570 times faster on average, with the most significant speedup observed at $\Delta t = 0.005$ (from 185.934 seconds to 0.325 seconds). Memory usage is slightly reduced. Importantly, the optimised solver is also more accurate, with error reductions up to a factor of 79 for $\Delta t = 0.005$. These results underscore the efficacy of the optimised solver in reducing computational costs while enhancing precision. Detailed results including the standard deviation can be found in Table 4 in Appendix C.1.

3. The execution order of the solvers, ODE and FDE, was chosen randomly within each run. Before every single run, `sleep` is called for one second to minimise side effects caused by other background tasks.
4. Memory consumption is measured separately from runtime performance as memory measurement heavily influences the runtime.

4.2. Empirical Comparison of NODE and NFDE

To compare the performance of NODE and NFDE, we apply the methodology described in Section 3.2 for a sub-exponential growth system described by

$$ {}_0^C D_t^\alpha y(t) = -\frac{y(t)}{2}, \quad y(0) = y_0, \quad y_0 \sim \mathcal{U}(a, b) \tag{11} $$

with $\alpha = 0.8$, and analyse the architectures' behaviour on the two tasks: reconstruction and extrapolation.

For the **reconstruction task**, 25 trajectories with distinct initial conditions are sampled independently for the training, validation, and test sets. The initial conditions for the training set are sampled from the uniform distribution $y_0 \sim \mathcal{U}(-1, 1)$, and the time horizon is fixed at $t \in [0, 20]$ with a time step of $dt = 0.1$. For the validation set, the initial conditions are sampled from $y_0 \sim \mathcal{U}(-1.5, 1.5)$, and for the test set, they are sampled from $y_0 \sim \mathcal{U}(-2, 2)$. This setup tests the models', ability to generalise to new initial conditions while keeping the time horizon consistent across datasets.

For the **extrapolation task**, 25 trajectories with distinct initial conditions are sampled randomly from the same uniform distribution $y_0 \sim \mathcal{U}(-1, 1)$ for the training, validation, and test sets. However, the difference lies in the time horizons used. The training set uses a time range $t \in [0, 1.5]$, the validation set uses $t \in [0, 3]$, and the test set uses $t \in [0, 10]$, all with a time step of $dt = 0.1$. This task evaluates the models', ability to predict over longer time horizons, while the initial conditions remain consistent within each dataset.

The datasets were generated using the *CaputoLSchema* FDE solver from the `python` library `brainpy` (Wang et al., 2023), with fixed seeds (42, 43, 44) for the train, validation, and test sets, respectively. These seeds ensure consistent dataset generation across evaluations, but differ between sets to provide distinct samples from the same distribution ranges. It is important to note that these dataset seeds are separate from the random seeds used for hyperparameter tuning, as described in Section 3.2. In both tasks, the solver always incorporates time t, while the NN is trained either with t as an explicit input (i.e. $\mathbf{f}_\theta(y(t), t)$) or using only the state $y(t)$ (i.e. $\mathbf{f}_\theta(y(t))$).

For each architecture and each task, 1000 hyperparameter tuning trials were conducted using TPE. The hyperparameters include the learning rate, number of layers (up to 4), number of hidden neurons (up to 100), and $\alpha \in [0.5, 1]$ (for NFDE). For further details on the search space, refer to Appendix B. Additionally, an exponential learning rate scheduler and the Adam optimiser (Kingma and Ba, 2014) were employed, with all samples processed in a single batch (*i.e.*, full-batch training). Performance was evaluated by Mean Squared Error (MSE) and, to ensure robustness, four independent runs with distinct seeds were performed for each architecture. Each run includes a hyperparameter search, where the train and validation data set was used to evaluate the different hyperparameter sets, found by the TPE. The final evaluation was performed on a distinct test set. Table 2 shows the mean and standard deviation on the test set. The results in Table 2 show that for the extrapolation task, the NFDE model performs significantly better without the time feature, achieving much lower MSE. A similar trend is observed for the NODE model, where the absence of the time feature also results in improved performance. For the reconstruction task, the NFDE model again shows better performance without the time feature, while the NODE model performs better with the time feature. These results highlight the crucial

Table 2: MSE over four seeds, after 100 Epochs, for the test set over four runs.

	Extrapolation	
	with Time	without Time
NFDE	1.69e-04 ± (9.17e-05)	**3.34e-05 ± (1.02e-05)**
NODE	3.47e-02 ± (3.53e-02)	2.90e-03 ± (8.65e-05)
	Reconstruction	
	with Time	without Time
NFDE	1.07e-04 ± (1.11e-04)	**4.40e-06 ± (1.90e-06)**
NODE	1.86e-04 ± (6.42e-05)	4.04e-04 ± (1.03e-04)

role the time feature plays in model accuracy, demonstrating that its impact can vary significantly depending on the model and task at hand. More results, including box plots, plots of the behaviour over epochs for all data sets can be found in Appendix C.2.

5. Conclusion

This work systematically addressed critical research questions comparing NODEs and NFDEs for modeling dynamical systems. The key findings are summarized below:

Time Feature Impact (RQ1): Our results demonstrate that NFDEs perform significantly better in both extrapolation and reconstruction tasks when time is excluded as an input feature to the neural network for systems with fractional behaviour. While NODEs can theoretically model fractional behaviour by learning time-varying dynamics, they often rely on time as an explicit feature in the neural network. This dependency on time introduces challenges in extrapolation tasks, as the model's predictions become sensitive to the time variable and struggle to generalise beyond the observed time range. In contrast, NFDEs excel due to their ability to capture long-range dependencies without requiring time as an explicit feature, making them well-suited for systems with fractional dynamics. Therefore, the superior performance of NFDEs in extrapolation applies specifically to systems with fractional behaviour and is not a general rule. The solver, in both cases, continues to operate using time.

Solver Efficiency and Accuracy (RQ2): We introduced an optimised solver for NFDEs that is approximately 570 times faster and up to 79 times more accurate than the original implementation. These improvements significantly enhance the feasibility of NFDEs for large-scale, high-precision applications, making them a practical tool for real-world systems governed by fractional dynamics.

Fair Comparison via Hyperparameter Tuning (RQ3): A rigorous hyperparameter search, combined with multiple independent runs, confirmed that NFDEs consistently outperform NODEs for systems exhibiting fractional behaviour, especially in extrapolation tasks. This highlights the importance of proper hyperparameter tuning for ensuring fair comparisons and optimal performance.

In conclusion, NFDEs provide a robust, efficient, and accurate approach for modelling systems with fractional dynamics, especially in tasks requiring extrapolation. Future work will focus on further improving the computational efficiency of NFDEs, particularly in terms of memory usage, and exploring their broader applicability.

Acknowledgments

C. Coelho acknowledges the funding by Fundação para a Ciência e Tecnologia - FCT (Portuguese Foundation for Science and Technology) through CMAT projects UIDB/00013/2020 and UIDP/00013/2020, the funding by FCT and Google Cloud partnership through projects CPCA-IAC/AV/589164/2023 and CPCA-IAC/AF/589140/2023. C. Coelho is also funded by FCT through the scholarship with reference 2021.05201.BD.

We extend our gratitude to OpenAI's ChatGPT for enhancing the readability of this manuscript and to GitHub Copilot for improving the structure and quality of our code. The contributions and final responsibility for the content, findings, and conclusions of this work lie solely with the authors.

References

Mohamed Aly Abdou. An analytical method for space–time fractional nonlinear differential equations arising in plasma physics. *Journal of Ocean Engineering and Science*, 2(4):288–292, 2017.

Ali Akgül and SarbazH A Khoshnaw. Application of fractional derivative on non-linear biochemical reaction models. *International Journal of Intelligent Networks*, 1:52–58, 2020.

Reem Abdullah Aljethi and Adem Kılıçman. Analysis of fractional differential equation and its application to realistic data. *Chaos, Solitons & Fractals*, 171:113446, 2023.

Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, jan 2017. ISSN 1532-4435.

Marin Biloš, Johanna Sommer, Syama Sundar Rangapuram, Tim Januschowski, and Stephan Günnemann. Neural Flows: Efficient Alternative to Neural ODEs. In *Neural Information Processing Systems*. January 2021. URL https://arxiv.org/pdf/2110.13040.

Willem Bonnaffé, Ben C Sheldon, and Tim Coulson. Neural ordinary differential equations for ecological and evolutionary time-series analysis. *Methods in Ecology and Evolution*, 12(7):1301–1315, 2021.

Dominic Stefan Bräm, Uri Nahum, Johannes Schropp, Marc Pfister, and Gilbert Koch. Low-dimensional neural odes and their application in pharmacokinetics. *Journal of Pharmacokinetics and Pharmacodynamics*, 51(2):123–140, 2024.

Ricky T. Q. Chen. torchdiffeq, 2018. URL https://github.com/rtqichen/torchdiffeq.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

Haoyu Chu, Shikui Wei, Qiming Lu, and Yao Zhao. Improving neural ordinary differential equations via knowledge distillation. *IET Computer Vision*, 18(2):304–314,

March 2024. ISSN 1751-9632, 1751-9640. doi: 10.1049/cvi2.12248. URL https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/cvi2.12248.

Cecília Coelho, M Fernanda P Costa, and Luís L. Ferrás. Neural fractional differential equations. *arXiv preprint arXiv:2403.02737*, 2024.

Kai Diethelm. *The Analysis of Fractional Differential Equations: An Application-Oriented Exposition Using Differential Operators of Caputo Type*, volume 2004 of *Lecture Notes in Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14573-5 978-3-642-14574-2. doi: 10.1007/978-3-642-14574-2. URL https://link.springer.com/10.1007/978-3-642-14574-2.

Kai Diethelm, Neville J. Ford, and Alan D. Freed. A Predictor-Corrector Approach for the Numerical Solution of Fractional Differential Equations. *Nonlinear Dynamics*, 29 (1/4):3–22, 2002. ISSN 0924090X. doi: 10.1023/A:1016592219341. URL http://link.springer.com/10.1023/A:1016592219341.

Salah A Faroughi, Ana I Roriz, and Célio Fernandes. A meta-model to predict the drag coefficient of a particle translating in viscoelastic fluids: a machine learning approach. *Polymers*, 14(3):430, 2022.

Roberto Garrappa. Numerical evaluation of two and three parameter mittag-leffler functions. *SIAM Journal on Numerical Analysis*, 53(3):1350–1369, 2015. doi: 10.1137/140971191. URL https://doi.org/10.1137/140971191.

Fudong Ge, YangQuan Chen, and Chunhai Kou. Cyber-physical systems as general distributed parameter systems: three types of fractional order models and emerging research opportunities. *IEEE/CAA Journal of Automatica Sinica*, 2(4):353–357, 2015.

Bill Goodwine. Modeling a multi-robot system with fractional-order differential equations. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1763–1768. IEEE, 2014.

Nicholas C Grassly and Christophe Fraser. Mathematical models of infectious disease transmission. *Nature Reviews Microbiology*, 6(6):477–487, 2008.

Christoph Grebner, Hans Matter, Daniel Kofink, Jan Wenzel, Friedemann Schmidt, and Gerhard Hessler. Application of deep neural network models in drug discovery programs. *ChemMedChem*, 16(24):3772–3786, 2021.

Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. ISBN 978-3-642-05220-0 978-3-642-05221-7. doi: 10.1007/978-3-642-05221-7. URL http://link.springer.com/10.1007/978-3-642-05221-7.

Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 8 of *Springer Series in Computational Mathematics*. Springer, Berlin, 2., rev. ed. edition, January 1993. ISBN 978-3-540-56670-0. doi: 10.1007/978-3-540-78862-1.

Samuel I Holt, Zhaozhi Qian, and Mihaela van der Schaar. Neural laplace: Learning diverse classes of differential equations in the laplace domain. In *International Conference on Machine Learning*, pages 8811–8832. PMLR, 2022.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Jonathan Krauß. *Optimizing Hyperparameters for Machine Learning Algorithms in Production*. PhD thesis, Apprimus Wissenschaftsverlag, Aachen, 2022. URL https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=6952102.

Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. Neural sde: Stabilizing neural ode networks with stochastic noise. *arXiv preprint arXiv:1906.02355*, 2019.

Yanzhi Liu, Jason Roberts, and Yubin Yan. A note on finite difference methods for nonlinear fractional differential equations with non-uniform meshes. *International Journal of Computer Mathematics*, 95(6-7):1151–1169, July 2018. ISSN 0020-7160, 1029-0265. doi: 10.1080/00207160.2017.1381691. URL https://www.tandfonline.com/doi/full/10.1080/00207160.2017.1381691.

Wangrong Ma, Maozhu Jin, Yifeng Liu, and Xiaobo Xu. Empirical analysis of fractional differential equations model for relationship between enterprise management and financial performance. *Chaos, Solitons & Fractals*, 125:17–23, 2019.

Zoya Meleshkova, Sergei Evgenievich Ivanov, and Lubov Ivanova. Application of neural ode with embedded hybrid method for robotic manipulator control. *Procedia Computer Science*, 193:314–324, 2021.

Iulian Mircea, Mihaela Covrig, and Radu Serban. Some mathematical models for longevity risk in the annuity market and pension funds. *Procedia Economics and Finance*, 15: 115–122, 2014.

Yoshihiko Ozaki, Yuki Tanigaki, Shuhei Watanabe, and Masaki Onishi. Multiobjective Tree-Structured Parzen Estimator for Computationally Expensive Optimization Problems. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, pages 533–541. Association for Computing Machinery, New York, NY, USA, January 2020. ISBN 978-1-4503-7128-5. doi: 10.1145/3377930.3389817.

Nicos G Pavlidis, Vassilis P Plagianakos, Dimitris K Tasoulis, and Michael N Vrahatis. Financial forecasting through unsupervised clustering and neural networks. *Operational Research*, 6:103–127, 2006.

D. Sculley, Jasper Snoek, Alexander B. Wiltschko, and Ali Rahimi. Winner's curse? on pace, progress, and empirical rigor. In *International Conference on Learning Representations*, 2018. URL https://api.semanticscholar.org/CorpusID:59548991.

Farshud Sorourifar, You Peng, Ivan Castillo, Linh Bui, Juan Venegas, and Joel A Paulson. Physics-enhanced neural ordinary differential equations: Application to industrial chemical reaction systems. *Industrial & Engineering Chemistry Research*, 62(38):15563–15577, 2023.

Philipp Städter, Yannik Schälte, Leonard Schmiester, Jan Hasenauer, and Paul L. Stapor. Benchmarking of numerical integration methods for ODE models of biological systems. *Scientific Reports*, 11(1):2696, January 2021. ISSN 2045-2322. doi: 10.1038/s41598-021-82196-2. URL https://www.nature.com/articles/s41598-021-82196-2.

VV Uchaikin. Relaxation processes and fractional differential equations. *International Journal of Theoretical Physics*, 42:121–134, 2003.

Cecile Valsecchi, Viviana Consonni, Roberto Todeschini, Marco Emilio Orlandi, Fabio Gosetti, and Davide Ballabio. Parsimonious Optimization of Multitask Neural Network Hyperparameters. *Molecules*, 26(23):7254, November 2021. ISSN 1420-3049. doi: 10.3390/molecules26237254. URL https://www.mdpi.com/1420-3049/26/23/7254.

Daniel Vranješ and Oliver Niggemann. Design Principles for Falsifiable, Replicable and Reproducible Empirical ML Research, 2024. URL https://arxiv.org/abs/2405.18077.

Chaoming Wang, Tianqiu Zhang, Xiaoyu Chen, Sichao He, Shangyang Li, and Si Wu. BrainPy, a flexible, integrative, efficient, and extensible framework for general-purpose brain dynamics programming. *eLife*, 12:e86365, December 2023. ISSN 2050-084X. doi: 10.7554/eLife.86365. URL https://elifesciences.org/articles/86365.

Theodor Westny, Arman Mohammadi, Daniel Jung, and Erik Frisk. Stability-Informed Initialization of Neural Ordinary Differential Equations, 2023. URL https://arxiv.org/abs/2311.15890.

P Zhang, X Xiao, and ZW Ma. A review of the composite phase change materials: Fabrication, characterization, mathematical modeling and application to performance enhancement. *Applied Energy*, 165:472–510, 2016.

Qunxi Zhu, Yao Guo, and Wei Lin. Neural delay differential equations. *arXiv preprint arXiv:2102.10801*, 2021.

Bernd Zimmering and Oliver Niggemann. Integrating continuous-time neural networks in engineering: Bridging machine learning and dynamical system modeling. In *Proceedings of the Conference on Machine Learning for Cyber-Physical Systems (ML4CPS)*. Helmut-Schmidt-Universität Hamburg, January 2024. doi: 10.24405/15313. URL https://openhsu.ub.hsu-hh.de/handle/10.24405/15313.

Appendix A. Pseudocode of the Optimised FDE Solver

Algorithm 1 Optimised Predictor-Corrector implementation from Diethelm et al. (2002) in `python` and `pytorch`. Bold symbols indicate tensors (batch, time, features). Blue indicates calls to the NN f.

function FDE_Integrator(f, $\boldsymbol{t}$, $\boldsymbol{y_0}$, α, h)
  Input: Function f, time points $\boldsymbol{t}$, initial condition $\boldsymbol{y_0}$, fractional order α, step-size h
  Output: Approximate solution $\boldsymbol{y}$ at time points $\boldsymbol{t}$
  $N \leftarrow$ number of time steps based on t and h
  $\boldsymbol{y_{int}} \leftarrow$ tensor of shape (batch_size, $N+1$, dim_y) ▷ Internal memory for solution
  $\boldsymbol{t_{int}} \leftarrow$ tensor of shape (batch_size, $N+1$, 1) ▷ Internal memory for time
  $\boldsymbol{f_{mem}} \leftarrow$ tensor of shape (batch_size, $N+1$, dim_y) ▷ Memory for previous evaluations of $f()$
  $\boldsymbol{y_{int}}[:, 0, :] \leftarrow \boldsymbol{y_0}$ ▷ Add initial condition to results as it is not computed in the main loop
  $\boldsymbol{k} \leftarrow$ range from 1 to N ▷ Indices for coefficients $\boldsymbol{a}$ and $\boldsymbol{b}$
  $\boldsymbol{b} \leftarrow ((\boldsymbol{k}^{\alpha}) - (\boldsymbol{k}-1)^{\alpha})$ ▷ Precompute coefficient b from Equation (9)
  $\boldsymbol{a} \leftarrow ((\boldsymbol{k}+1)^{\alpha+1} - 2\boldsymbol{k}^{\alpha+1} + (\boldsymbol{k}-1)^{\alpha+1})$ ▷ and a from Equation (10)
  $\Gamma_{\alpha 1} \leftarrow \gamma(\alpha+1)$ ▷ Gamma terms for Equation (9)
  $\Gamma_{\alpha 2} \leftarrow \gamma(\alpha+2)$ ▷ Gamma terms for Equation (10)
  $\boldsymbol{f_0} \leftarrow f(\boldsymbol{y_0}, t[:, 0, :])$ ▷ Precompute $\boldsymbol{f_0}$ for the corrector (Equation (10))
  $\boldsymbol{f_{mem}}[:, 0, :] \leftarrow \boldsymbol{f_0}$ ▷ and store results
  $\boldsymbol{y_{new}} \leftarrow \boldsymbol{y_0}$ ▷ Initialise last solution with the initial value
  for $j = 1$ to $N+1$ **do** ▷ Main loop of solver
    $t_{act} \leftarrow j \cdot h$ ▷ Calculate the current time
    $\boldsymbol{t_{int}}[:, j] \leftarrow t_{act}$ ▷ and store it
    $\boldsymbol{f_{mem}}[:, j, :] \leftarrow f(\boldsymbol{y_{new}}, t_{act})$ ▷ Compute and store the value of f at the current step
    $\boldsymbol{k} \leftarrow \boldsymbol{kn}[:j]$ ▷ Get indices of all previous values
    $\boldsymbol{b_{jk}} \leftarrow \boldsymbol{b}[j - \boldsymbol{k}]$ ▷ Retrieve necessary values for $\boldsymbol{b}$ in reverse order
    $\boldsymbol{y_p} \leftarrow \boldsymbol{y_0} + \left(\frac{h^{\alpha}}{\Gamma_{\alpha 1}}\right) \sum(\boldsymbol{b_{jk}} \cdot \boldsymbol{f_{mem}}[:, :j, :])$ ▷ Compute the predictor step (Equation (9)) using torch's parallel computation for the sum term
    $\boldsymbol{a_{jk}} \leftarrow \boldsymbol{a}[(j - \boldsymbol{k})[1:]]$ ▷ Retrieve necessary values for $\boldsymbol{a}$ in reverse order
    $\boldsymbol{y_{new}} \leftarrow \boldsymbol{y_0} + \left(\frac{h^{\alpha}}{\Gamma_{\alpha 2}}\right) \cdot \Big(f(\boldsymbol{y_p}, t_{act}) + ((j-1)^{\alpha+1} - (j-1-\alpha)j^{\alpha}) \cdot \boldsymbol{f_0} + \sum \boldsymbol{a_{jk}} \cdot \boldsymbol{f_{mem}}[:, 1:j, :]\Big)$ ▷ Compute the corrector step (Equation (10)) using torch's parallel computation for the sum term
    $\boldsymbol{y_{int}}[:, j, :] \leftarrow \boldsymbol{y_{new}}$ ▷ Store the new value for the solution
  end for
  return LinearInterpolation($\boldsymbol{y_{int}}$, $\boldsymbol{t_{int}}$, $\boldsymbol{t}$)
end function

Algorithm 1 continued

function LinearInterpolation($\boldsymbol{y_{internal}}, \boldsymbol{t_{internal}}, \boldsymbol{t}$)
    $\boldsymbol{t_{internal}}$ is a fixed, evenly spaced time grid
    $\boldsymbol{t}$ contains arbitrary, potentially uneven time points
    Find indices of $\boldsymbol{t_{internal}}$ just before ($\boldsymbol{t_0}$) and after ($\boldsymbol{t_1}$) for each $\boldsymbol{t}$
    Gather corresponding $\boldsymbol{y_{internal}}$ values at $\boldsymbol{t_0}$ and $\boldsymbol{t_1}$
    $\boldsymbol{y_{out}} = \boldsymbol{y_0} + (\boldsymbol{y_1} - \boldsymbol{y_0}) \cdot \dfrac{(\boldsymbol{t} - \boldsymbol{t_0})}{(\boldsymbol{t_1} - \boldsymbol{t_0})}$
    return $\boldsymbol{y_{out}}$
end function

Appendix B. Hyperparameter Space Details

Table 3: Hyperparameter ranges for NODE and NFDE models.

Hyperparameter	**NODE**	**NFDE**
Learning Rate	$[1 \times 10^{-5}, 0.5]$	$[1 \times 10^{-5}, 0.5]$
Gamma Scheduler	$[0.999, 1.0]$	$[0.999, 1.0]$
Hidden Size	$[10, 100]$	$[10, 100]$
Number of Hidden Layers	$[1, 4]$	$[1, 4]$
Activation Function	{relu, leaky_relu, elu, none}	{relu, leaky_relu, elu, none}
Alpha (Non-integer Order)	-	$[0.5, 1.0]$

Appendix C. Additional Results

C.1. Detailed Results for Numerical Method Implementations Comparison

Table 4: Performance comparison between original and optimised solver including standard deviation from over five runs for each value.

Δt	Original Solver	Optimised Solver
	Time (s)	
0.005	1.86e+02 (2.51e+00)	3.25e-01 (4.79e-03)
0.01	4.62e+01 (9.28e-01)	1.57e-01 (2.05e-03)
0.1	4.89e-01 (1.21e-02)	1.61e-02 (1.42e-04)
1.0	8.06e-03 (1.34e-03)	2.51e-03 (4.69e-04)
	Memory (MB)	
0.005	1.58e+04 (1.29e+03)	1.57e+04 (7.43e+02)
0.01	4.31e+03 (3.24e+02)	3.59e+03 (1.72e+03)
0.1	5.07e+02 (7.06e+00)	5.00e+02 (2.16e+01)
1.0	4.61e+02 (6.99e-02)	4.61e+02 (8.39e-01)
	Error	
0.005	1.28e-08 (0.00e+00)	1.62e-10 (0.00e+00)
0.01	5.77e-08 (0.00e+00)	1.27e-09 (0.00e+00)
0.1	9.91e-06 (0.00e+00)	1.03e-06 (0.00e+00)
1.0	6.43e-03 (0.00e+00)	1.03e-03 (0.00e+00)

C.2. More results of the empirical comparison for NODE and NFDE.

Table 5: Mean MSE, after 100 Epochs, for the training set over four runs.

	Extrapolation	
	with Time	without Time
NFDE	1.90e-06 ± (4.84e-07)	2.00e-06 ± (2.27e-07)
NODE	3.70e-06 ± (3.10e-06)	7.56e-05 ± (1.30e-06)
	Reconstruction	
	with Time	without Time
NFDE	1.13e-04 ± (1.17e-04)	6.40e-06 ± (4.40e-06)
NODE	1.87e-04 ± (6.99e-05)	4.01e-04 ± (8.76e-05)

Table 6: Mean MSE, after 100 Epochs, for the validation set over four runs.

	Extrapolation	
	with Time	without Time
NFDE	4.20e-06 ± (1.40e-06)	2.30e-06 ± (6.43e-07)
NODE	2.50e-04 ± (2.89e-04)	6.67e-04 ± (2.23e-05)
	Reconstruction	
	with Time	without Time
NFDE	1.24e-04 ± (1.39e-04)	6.60e-06 ± (4.40e-06)
NODE	2.17e-04 ± (8.21e-05)	4.38e-04 ± (1.21e-04)

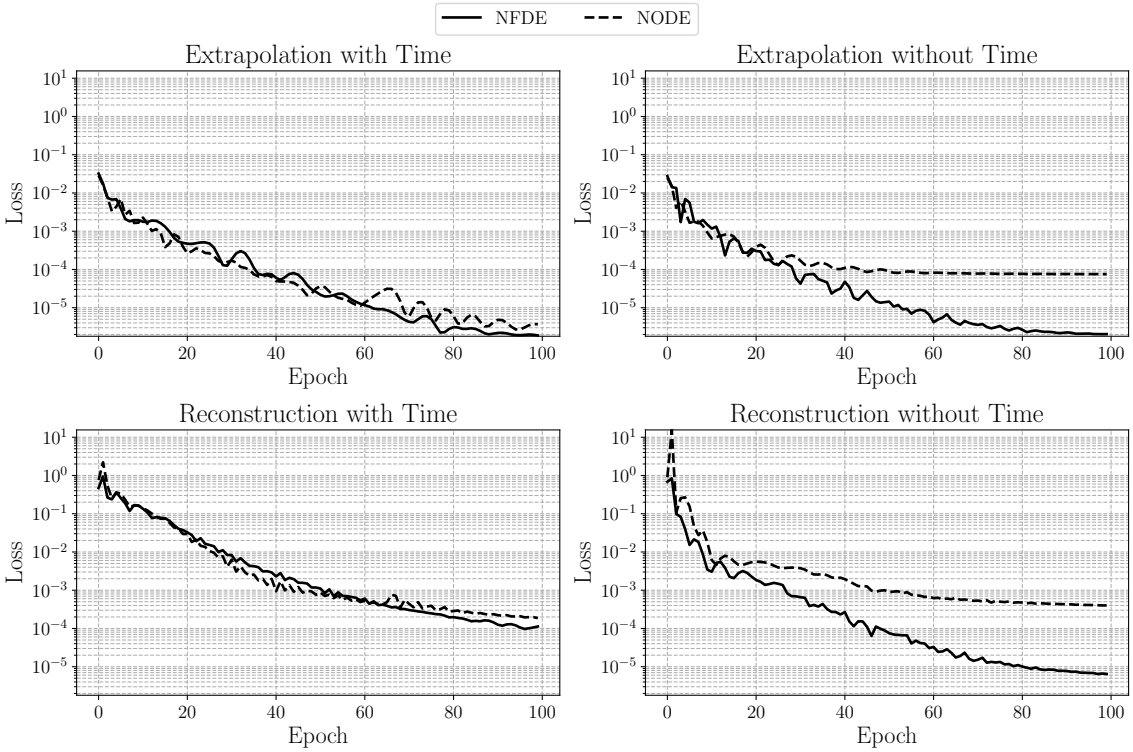

Figure 1: Mean MSE over four seeds on the **training set** over 100 epochs for the reconstruction and extrapolation tasks.

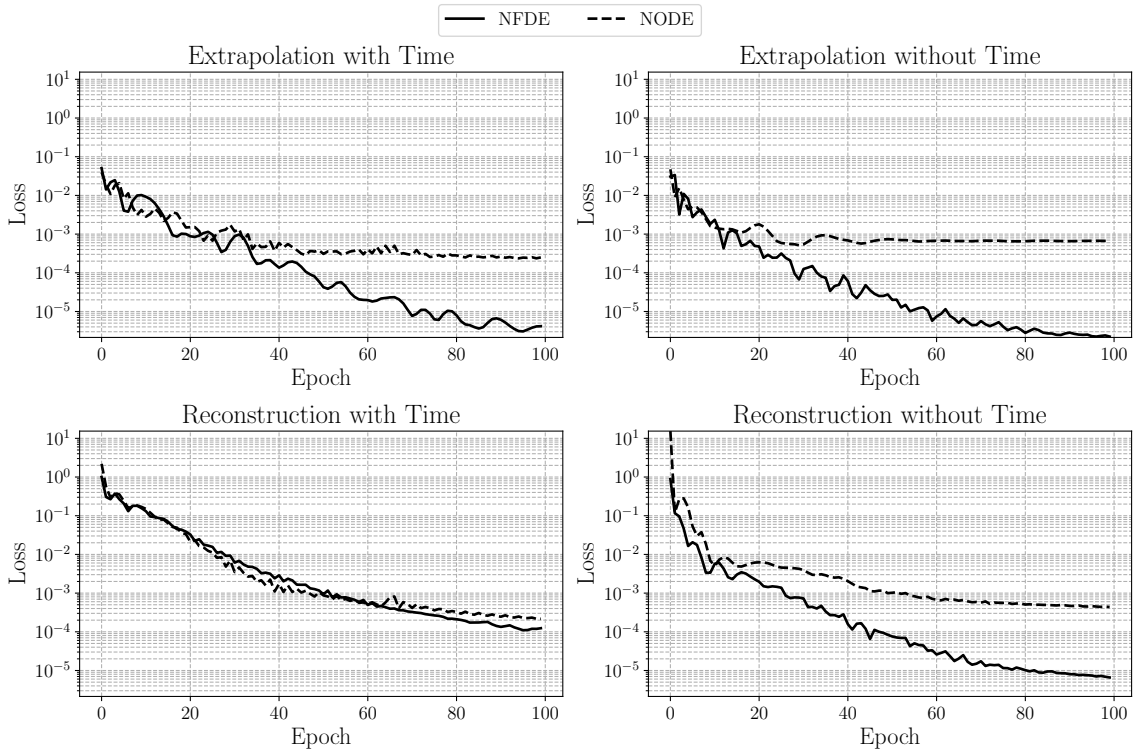

Figure 2: Mean MSE over four seeds on the **validation set** over 100 epochs for the reconstruction and extrapolation tasks.

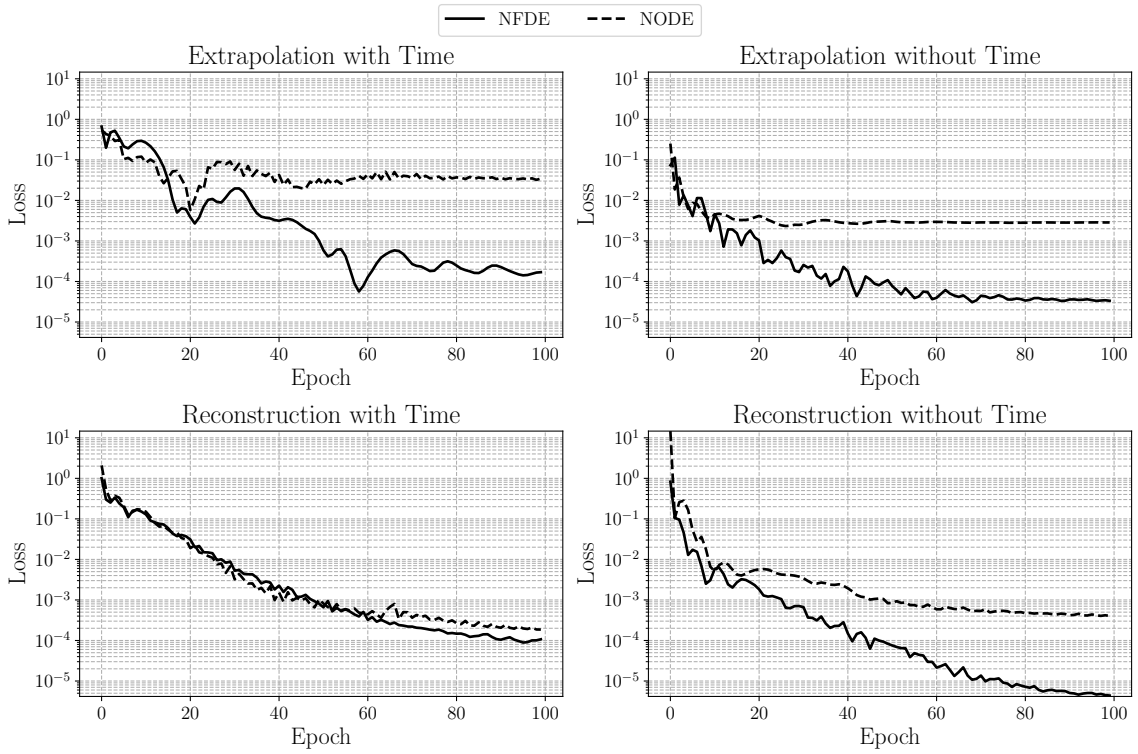

Figure 3: Mean MSE over four seeds on the **test set** over 100 epochs for the reconstruction and extrapolation tasks.

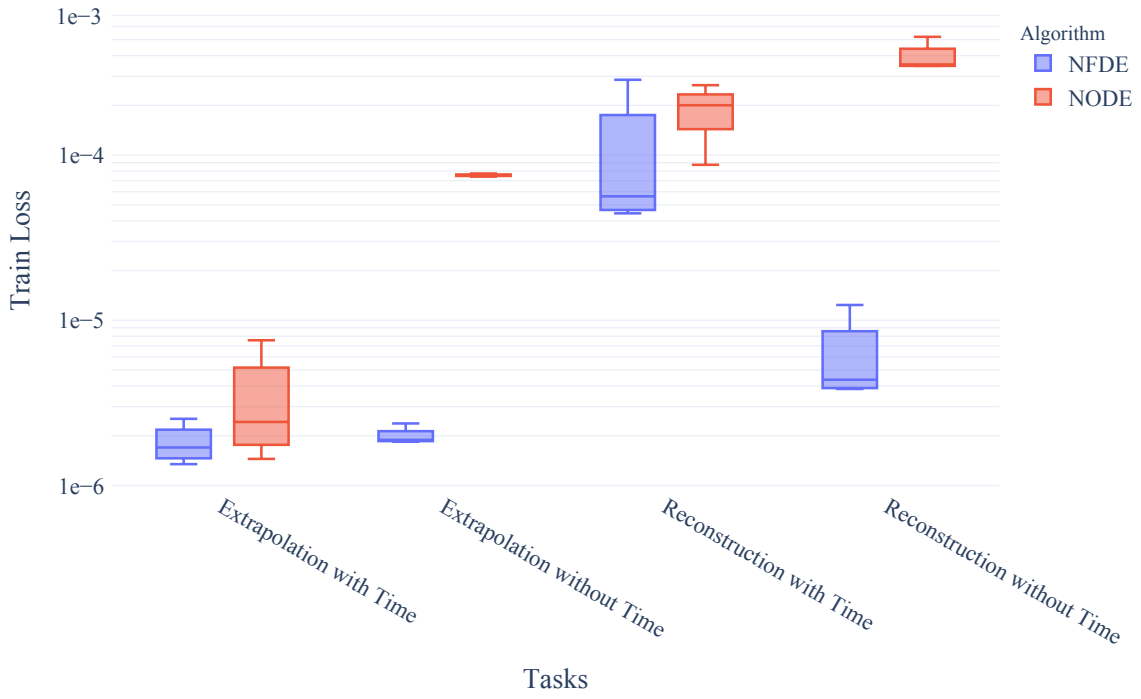

Figure 4: Box plot of MSE for the **training set** after 100 epochs.

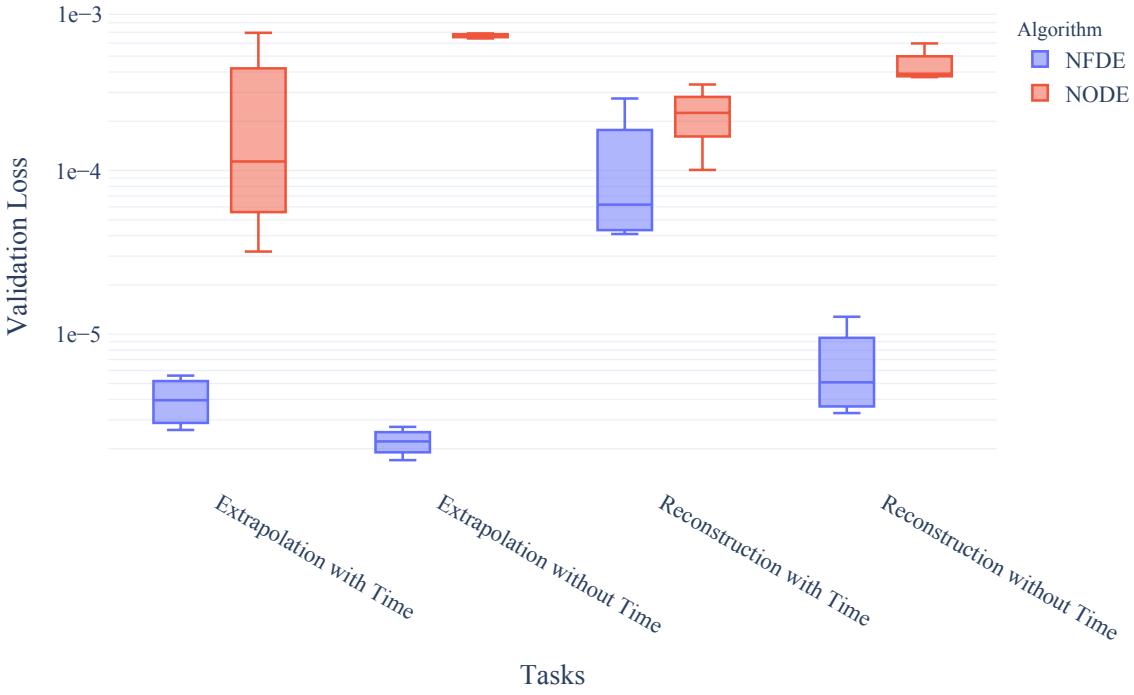

Figure 5: Box plot of MSE for the **validation set** after 100 epochs.

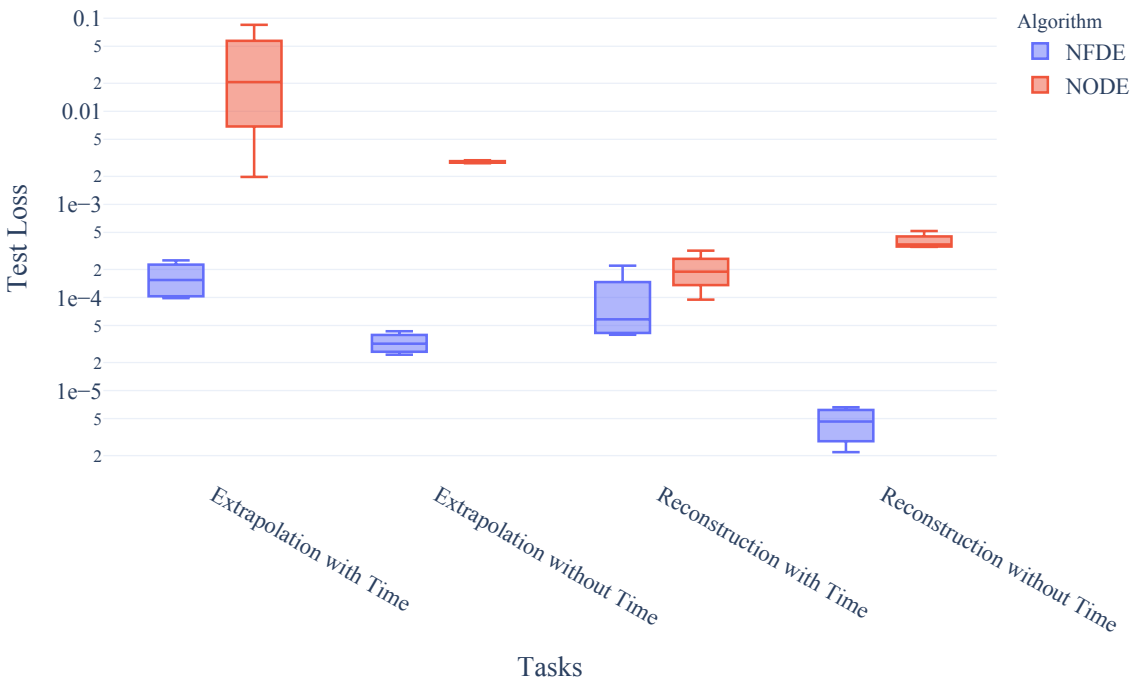

Figure 6: Box plot of MSE for the **test set** after 100 epochs.