

Latent space modeling of parametric and time-dependent PDEs using neural ODEs

Longhi, Alessandro; Lathouwers, Danny; Perkó, Zoltán

DOI

[10.1016/j.cma.2025.118394](https://doi.org/10.1016/j.cma.2025.118394)

Publication date

2026

Document Version

Final published version

Published in

Computer Methods in Applied Mechanics and Engineering

Citation (APA)

Longhi, A., Lathouwers, D., & Perkó, Z. (2026). Latent space modeling of parametric and time-dependent PDEs using neural ODEs. *Computer Methods in Applied Mechanics and Engineering*, 448, Article 118394. <https://doi.org/10.1016/j.cma.2025.118394>

Important note

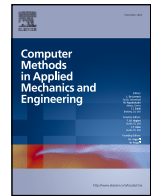
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright


Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy

Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.



Latent space modeling of parametric and time-dependent PDEs using neural ODEs

Alessandro Longhi ^{*}, Danny Lathouwers, Zoltán Perkö

Department of Radiation Science and Technology, TU Delft University of Technology, Mekelweg 15, Delft, 2629JB, the Netherlands

ARTICLE INFO

Keywords:

Partial differential equations
Surrogate modeling
Dimensionality reduction
Neural ordinary differential equations

ABSTRACT

Partial Differential Equations (PDEs) are central to science and engineering. Since solving them is computationally expensive, a lot of effort has been put into approximating their solution operator via both traditional and recently increasingly Deep Learning (DL) techniques. In this paper, we propose an autoregressive and data-driven method using the analogy with classical numerical solvers for time-dependent, parametric and (typically) nonlinear PDEs. We present how Dimensionality Reduction (DR) can be coupled with Neural Ordinary Differential Equations (NODEs) in order to learn the solution operator of arbitrary PDEs accounting both for (continuous) time and parameter dependency. The idea of our work is that it is possible to map the high-fidelity (i.e., high-dimensional) PDE solution space into a reduced (low-dimensional) space, which subsequently exhibits dynamics governed by a (latent) Ordinary Differential Equation (ODE). Solving this (easier) ODE in the reduced space allows avoiding solving the PDE in the high-dimensional solution space, thus decreasing the computational burden for repeated calculations for e.g., uncertainty quantification or design optimization purposes. The main outcome of this work is the importance of exploiting DR as opposed to the recent trend of building large and complex architectures: we show that by leveraging DR we can deliver not only more accurate predictions, but also a considerably lighter and faster DL model compared to existing methodologies.

1. Introduction

Physical simulations are crucial to all areas of physics and engineering, such as fluid dynamics, nuclear physics, climate science, etc. Although a lot of work has been done in constructing robust and quick numerical Partial Differential Equations (PDE) solvers [1], traditional solvers such as finite element methods are still computationally expensive when the system is complex. This is a problem especially when repeated evaluations of a model for different initial conditions and parameters are needed, which is typical in sensitivity analysis, design optimization or uncertainty quantification studies [2,3]. To overcome such time limitations, decades of extensive research has been put into building so called *surrogate models*, i.e., faster to evaluate but accurate enough approximations of the original complex model describing the physical system of interest.

The first studies on surrogate modeling fall under the umbrella of *Reduced Order Modeling* (ROM) [4] methods, with the pioneering work on Proper Orthogonal Decomposition (POD) by Lumley in 1967 [5]. The main assumption of ROM is that a system determined by N (potentially infinite) degrees of freedom (full space) can instead be projected into a lower dimensional space of dimension n (reduced space), hence its evolution can be calculated by solving a much smaller system of $n \ll N$ equations. A common way of

^{*} Corresponding author.

E-mail address: a.longhi@tudelft.nl (A. Longhi).

proceeding, under the name of reduce basis methods, is by assuming that the solution field $s(x, t)$ of a PDE can be approximated as: $s(x, t) \approx \sum_{k=1}^n a_k(t) V_k(x)$ with $a_k(t)$ being time-dependent coefficients and $V_k(x)$ being independent variable x dependent functions, the latter constituting an orthonormal basis (the reduced basis). Once the optimal basis is found the system is completely described by the n coefficients $a_k(t)$. The same concept of Dimensionality Reduction (DR) is known in the Deep Learning (DL) field under the name of *manifold hypothesis* [6–9], analogously stating that high-dimensional data typically lie in low dimensional manifolds (due to correlations, symmetries, noise in data, etc.). In DL jargon, this reduced space is usually named *latent space*.

Several recent works explore the potential of DL for surrogate modeling, both following the ideas of traditional ROM approaches and proposing new paradigms. A non exhaustive list of methods that integrate DL techniques with ROM concepts is provided in Fresca and Manzoni [10], Bhattacharya et al. [11], Lee and Carlberg [12], Solera-Rico et al. [13], Li et al. [14]. Among these works, DL is used to approximate the mapping between full space and reduced space, to determine the coefficients of the reduced basis and/or to map initial states of the PDE into the PDE solution $s(x, t)$. Lusch et al. [15] implements concepts from Koopman Operator theory [16] for dynamical models, where the linearity of the Koopman Operator is exploited to advance in time the dynamical fields in a reduced space. The Sparse Identification of Nonlinear Dynamics (SINDy) is proposed in Brunton et al. [17], where the reduced vectors are assumed to follow a dynamics governed by a library of functions determined a priori.

Recently, *Neural Operators* (NO) [18,19], i.e., DL models whose objective is the *approximation of operators* instead of functions - contrary to what is typical in DL - have found applications in surrogate modeling tasks. As in the case of PDEs we deal with a mapping between infinite-dimensional functional spaces (from the space of initial and boundary conditions to the solution space of the PDE), the approximated operator is called the *solution operator* of the PDE. The (chronologically) first works on Neural Operators are the DeepONets [20] and the Fourier Neural Operator [21]. Since these publications, the literature on NO has flourished, with many theoretical and empirical studies [18,22–28]. In some related works Graph Deep Learning has been used for surrogate modeling to generalize to different geometries [29–32]. Beside DR, our model also leverages Neural ODEs [33] (NODEs), which are a class of Neural Networks (NNs) where the state of the system $h(t) \in \mathbb{R}^D$ behaves according to $\frac{dh(t)}{dt} = f_\theta(h(t), t)$, with f_θ being parametrized by a NN. NODEs present the advantage of modeling the dynamics of $h(t)$ continuously in time.

1.1. Related works

Among the large literature on Neural Operators and methods at the intersection between DL and ROM, there are four sets of papers closest to our work:

1. papers that use AutoEncoders to map the PDE solution space into a reduced space but do not model the latent dynamics continuously (as we do by using a Neural ODE), like [34–36];
2. papers that use AutoEncoders to map the PDE solution space into a reduced space and model the latent dynamics through a Neural ODE like [37,38,38–41];
3. papers based on the Latent Space Dynamics Identification framework, such as He et al. [42], Bonneville et al. [43], Park et al. [44], Anderson et al. [45];
4. papers that build surrogate models of parametric and time-dependent PDEs using architectures with no use of dimensionality reduction and Neural ODEs like [46,47]. In these works the different methodologies are built on Neural Fields, Transformers, Neural Operators and/or Graph Neural Networks.

1.2. Contributions

In this work we propose an autoregressive DL-based method for solving **parametrized, time-dependent** and (typically) **nonlinear** PDEs exploiting *dimensionality reduction* and *Neural ODEs*. Our novel contributions, especially compared to the papers in set 2 of Section 1.1, are the following:

- We construct a model that allows for the variation of both the PDE's **parameters** and **initial conditions**. We do so by defining two mappings parametrized by 2 NNs: a close-to bijective mapping between the **full** (high-fidelity) PDE solution space and the **latent** (low-fidelity) space via an AutoEncoder (AE) and a mapping from the latent vector at time t_i to the next latent vector at time t_{i+1} modeled by a (latent) NODE.
- Training on a **given** Δt we show that our model can generalize at testing time to **finer** time steps $\Delta t' < \Delta t$. To this regard we also study the connection between the order of the Runge–Kutta solver used to solve the NODE in latent space and the time generalization capabilities of the model. Additionally, in Section 2.6, we introduce a term of the loss function which enhance time generalization.
- We show a simple but effective strategy to train this model combining a **Teacher Forcing** type of training with an approach which takes into account the **Autoregressive** nature of this model at testing time.
- We achieve computational speed up compared to standard numerical PDE solvers thanks to 3 factors: doing inference at a Δt **higher** than what is usually required by standard numerical solvers, solving an **ODE** instead of a PDE and advancing in time in a **low** dimensional space instead of the full original space.
- We test our methodology on a series of PDE benchmarks (using [48] among others) and show that thanks to DR, our model is (at least 2 times) **lighter** and (at least 2 times) **faster** than current Deep Learning based State of the Art (SOTA) methods.

The distinction of our work compared to the papers in set 3 of Section 1.1 is as follows:

- *Greater flexibility in modeling latent dynamics:* [42–44] model the low dimensional dynamics through an ODE by using variants of the SINDy [17] algorithm, which needs an a priori selected library of *candidate functions* to construct the source the ODE (right-hand side function f (6) later). We instead use a NN to approximate f , requiring no a priori knowledge about the functional form of the source and allowing for greater flexibility. Anderson et al. [45] also uses a NN to approximate f , but uses only Teacher-Forcing losses in training. As we explain 2.5 however, adding Autoregressive loss terms to the training can greatly improve performance.
- *Better handling of parameter variations:* our model allows for the (simultaneous) variation of both the initial condition of the PDE and of general parameters (like boundary condition and PDE parameters) by delegating the treatment of the initial condition directly to the AutoEncoder and the treatment of the parameters to the NODE (as input to the source function f allowing learned non-linear dependencies). In contrast, in He et al. [42], Bonneville et al. [43], Park et al. [44] a different f_i is obtained for each parameter instance i , necessitating interpolation of the different f_i values at inference time and limiting the dependency of f on the parameters to the complexity of the prescribed interpolation scheme. Only in Anderson et al. [45] are the parameters an input to f , similar to our work.
- *Proven simultaneous generalization to initial conditions and parameters:* our extensive experiments demonstrates how our method can achieve generalization when initial conditions and the PDE parameters are simultaneously varied, contrary to all works of He et al. [42], Bonneville et al. [43], Park et al. [44], Anderson et al. [45], which never vary them at the same time. In addition, we also compare our model to several recently published methods featuring Neural Fields, Transformers, Neural Operators and Graph Neural Networks, which are not considered in He et al. [42], Bonneville et al. [43], Park et al. [44], Anderson et al. [45];
- *Detailed study of time generalization:* we perform several experiments to study our methods ability to generalize in time, contrary to all the papers of He et al. [42], Bonneville et al. [43], Park et al. [44], Anderson et al. [45].

2. Methodology

2.1. Mathematical background

Let \mathcal{V} , S and S^0 be the functional spaces of the boundary condition functions, PDE solution functions and the initial condition functions of a given PDE, respectively. These functions are required to satisfy some properties, such that:

$$\begin{cases} \mathcal{V} = \{v|v : \partial D_x \times D_t \rightarrow \mathbb{R}^m; \|v_i\|_2 < \infty \forall i \in [1, \dots, m] ; v \in C^2\}, \\ S = \{s|s : D_x \times D_t \rightarrow \mathbb{R}^m; \|s_i\|_2 < \infty \forall i \in [1, \dots, m] ; s \in C^2\}, \\ S^0 \subseteq S^t = \{s(x, t = \tilde{t}|\mu)|s(x, t = \tilde{t}|\mu) : D_x \rightarrow \mathbb{R}^m, \forall \tilde{t} \in D_t, \mu \in D_\mu\}, \end{cases} \quad (1)$$

where S^t is the set of all possible states, $D_x \subseteq \mathbb{R}^n$ is the domain of independent variables \mathbf{x} , $D_t \subseteq \mathbb{R}^+$ is the temporal domain and $\partial D_x \subseteq \mathbb{R}^n$ the boundary of D_x and $D_\mu \subseteq \mathbb{R}^z$ is a domain for the vector $\mu = (\mu_1, \mu_2, \dots, \mu_z)$, containing information about the PDE parameters, the geometry of the problem and whatever quantity defines the physical system. We are interested in solving general PDEs of the form of:

$$\begin{cases} \hat{\mathcal{N}}(s(\mathbf{x}, t|\mu), \mathbf{x}, t, \mu) = g(\mathbf{x}, t, \mu) \\ s(\tilde{\mathbf{x}}, t|\mu) = v(\tilde{\mathbf{x}}, t, \mu) \\ s(\mathbf{x}, t = 0|\mu) = s^0(\mathbf{x}, \mu), \end{cases} \quad (2)$$

where $\hat{\mathcal{N}}(s(\mathbf{x}, t|\mu), \mathbf{x}, t, \mu)$ is a (typically) nonlinear integro-differential operator, $g(\mathbf{x}, t, \mu)$ is the forcing term, $s \in S$ is the PDE solution, $v \in \mathcal{V}$ and $s^0 \in S^0$ are the boundary and initial conditions, $\mathbf{x} \in D_x$, $\mu \in D_\mu$, $\tilde{\mathbf{x}} \in \partial D_x$, $t \in D_t$. The different elements of the system of Eq. (2) have either an explicit dependency on μ , signaled by (\cdot, μ) , or an implicit dependency, signaled by $(\cdot|\mu)$. Although (2) describes a very general PDE system and our method description addresses this fully general case, in the experiments shown in Section 3 we fix $v(\tilde{\mathbf{x}}, t, \mu)$, i.e., it is not an input to the NN, we choose $g(\mathbf{x}, t, \mu) = 0$ and $\hat{\mathcal{N}} = \hat{\mathcal{N}}(s(\mathbf{x}, t|\mu), \mathbf{x}, \mu)$, i.e., without explicit t dependence.

In the context of surrogate modeling for parametric PDEs, one usually approximates by means of a NN either the *Solution Operator* \hat{S} (global approach) or the *Evolution Operator* \hat{H} (autoregressive approach), where

$$\hat{S} : S^0 \times D_t \times D_\mu \rightarrow S \quad \text{and} \quad \hat{H} : S^t \times D_{\Delta t} \times D_\mu \rightarrow S^t, \quad (3)$$

with $D_{\Delta t} \subseteq \mathbb{R}^+$. When approximating \hat{S} , the NN is given as input (s^0, t, μ) to output $s(\mathbf{x}, t|\mu)$, while when approximating \hat{H} , the NN is given as input $(s(\mathbf{x}, t = \tilde{t}|\mu), \Delta t, \mu)$ to output $s(\mathbf{x}, \tilde{t} + \Delta t|\mu)$. While the former can approximate the solution s at any point in time t with just one call of the solver, the latter requires advancing iteratively in time by predicting the solution at the next time step from the solution at the previous time step as input, starting from s^0 . Although the global approach has a (potential) advantage in terms of computational speed, we propose an autoregressive method for the following reasons:

- Most PDEs represent causal physical phenomena, hence their solution evolution at time t only depends on the system state at t . Therefore, as it is done in classical numerical solvers, only the solution s at time t is necessary for the prediction of s at time $t + \Delta t$. This fact is not respected by global approaches.
- Global approaches require in general a high number of NN weights, as a mapping for arbitrary t is required, contrary to autoregressive methods, as the state $s(t)$ carries more information than s^0 to predict $s(t + \Delta t)$.

While the two approaches are clearly different from a theoretical perspective, from a purely *architectural* point of view they are very similar, as an architecture conceived as global can always be used autoregressively and vice-versa. It is primarily the training strategy that determines whether a global or autoregressive logic is followed.

In what follows we show how to approximate with a NN the (latent) Evolution Operator that governs the dynamics of the **reduced** space to which the full space S is mapped.

2.2. Discretization

In order to work with solution functions s computationally it is necessary to discretize the independent variable (typically spatial at least), temporal and parametric domains. We thus define: $\mathbf{X} = \{x_k | x_k \in D_{\mathbf{x}}, x_k = (x_k^1, \dots, x_k^n), k = 0, \dots, N_{\mathbf{x}}\}$ as the set of points inside the domain of independent variables; $\partial\mathbf{X} = \{\tilde{x}_k | \tilde{x}_k \in \partial D_{\mathbf{x}}, \tilde{x}_k = (\tilde{x}_k^1, \dots, \tilde{x}_k^n), k = 0, \dots, N_{\tilde{\mathbf{x}}}\}$ as the set of points on the boundary of the independent variables of the domain $D_{\mathbf{x}}$; $\mathcal{X} = \mathbf{X} \cup \partial\mathbf{X}$, $\mathbf{M} = \{\boldsymbol{\mu} \in D_{\boldsymbol{\mu}}, \boldsymbol{\mu} = (\mu_0, \mu_1, \dots, \mu_z)\}$ as the set of parameter points $\boldsymbol{\mu}$; $\mathbf{T} = \{t | t \in D_t, t = (t_0, t_1, \dots, t_F)\}$ as the set of discrete points in time. We also define the *solution* and the *initial condition* sets, as the sets of functions living in S and S^0 discretized on \mathcal{X} and \mathbf{T} : $S_r = \{s_r(\mathbf{x}, t | \boldsymbol{\mu}) | \mathbf{x} \in \mathcal{X}, t \in \mathbf{T}, \boldsymbol{\mu} \in \mathbf{M}\} \subset \mathbb{R}^{|\mathcal{X}| \times n \times m}$ and $S_r^0 = \{s_r^0(\mathbf{x}, \boldsymbol{\mu}) | \mathbf{x} \in \mathcal{X}, \boldsymbol{\mu} \in \mathbf{M}\} \subset S_r$, where r is a subscript that indicates a discretized representation of s . Obviously, $s_r(\mathbf{x}, t_0 | \boldsymbol{\mu}) = s_r^0(\mathbf{x}, \boldsymbol{\mu})$. We will signal the implicit parameter dependency of s_r using the notation $s_r(\mathbf{x}, t | \boldsymbol{\mu})$. In principle we have $s_r(\mathbf{x}, t | \boldsymbol{\mu}, s_r^0)$, but for notational ease we will drop the implicit dependence on s_r^0 . Although we are using a finite difference approach for discretization, our methodology is fully general to other discretization schemes too (finite volumes, finite elements, etc.)

2.3. Reduced space and (latent) neural ODEs

We want to build a method that at inference time maps the initial condition s^0 into its reduced representation and then evolves it in time (according to the PDE parameters) autoregressively. The **first** building block of our methodology is the mapping between the full and the reduced space by an AutoEncoder. Let \mathcal{E} be the *reduced (latent) set*

$$\mathcal{E} = \{\varepsilon(t | \boldsymbol{\mu}) | \varepsilon(t | \boldsymbol{\mu}) = (\varepsilon_1(t | \boldsymbol{\mu}), \dots, \varepsilon_\lambda(t | \boldsymbol{\mu})), t \in D_t, \boldsymbol{\mu} \in D_{\boldsymbol{\mu}}\} \subset \mathbb{R}^\lambda, \quad (4)$$

with $\lambda \ll |\mathcal{X}| \cdot nm$ being the dimension of the latent space. Each time-dependent vector $\varepsilon(t | \boldsymbol{\mu}) \in \mathcal{E}$ has a one-to-one correspondence with a given solution function $s \in S$ (implicitly depending on the parameter $\boldsymbol{\mu}$), so that by computing the dynamics of $\varepsilon(t | \boldsymbol{\mu})$ we can reconstruct the original trajectory of $s(\mathbf{x}, t | \boldsymbol{\mu})$. Each dimension $\varepsilon_i(t)$ is an *intrinsic representation* of the corresponding function s and embodies the correlations, symmetries and fundamental information about the original object s (for a deeper understanding of the nature and the desiderata of a latent representation, see Eastwood and Williams [49], Higgins et al. [50]). Although we will work with discretized functions belonging to S_r , each vector $\varepsilon(t | \boldsymbol{\mu})$ is in principle associated with the original **continuous** function belonging to S (i.e., $\varepsilon(t | \boldsymbol{\mu})$ should be independent of the discretization of S).

The mathematical operators mapping S to \mathcal{E} and viceversa are the *Encoder* φ and the *Decoder* ψ , such that:

$$\varphi : S \rightarrow \mathcal{E} \quad \text{and} \quad \psi : \mathcal{E} \rightarrow S, \quad (5)$$

with $\varphi \circ \psi = \psi \circ \varphi = \mathbb{1}$, together forming the AutoEncoder. We approximate φ and ψ by two NNs, respectively $\varphi_\theta : S_r \rightarrow \mathcal{E}$ and $\psi_\theta : \mathcal{E} \rightarrow S_r$. The **second** building block concerns the dynamics of the vectors ε belonging to the reduced set \mathcal{E} . We assume that the temporal dynamics of \mathcal{E} follows an ODE:

$$\frac{d}{dt} \varepsilon(t | \boldsymbol{\mu}) = f(\varepsilon(t | \boldsymbol{\mu}), \boldsymbol{\mu}), \quad f \in F : \mathcal{E} \times D_{\boldsymbol{\mu}} \rightarrow \mathcal{E}, \quad (6)$$

where $\boldsymbol{\mu} \in \mathbf{M}$ is the vector of PDE parameters. f does not depend *explicitly* on t since the PDEs we work with do not have explicit time dependence, making the dynamics of \mathcal{E} an *autonomous system*. If instead $\hat{\mathcal{N}}$ or $g(\mathbf{x}, t, \boldsymbol{\mu})$ had an explicit dependence on t , we would have $f = f(\varepsilon(t | \boldsymbol{\mu}), \boldsymbol{\mu}, t)$ and would treat t in the model simply as an additional dimension of $\boldsymbol{\mu}$. We can now define the *Processor*

$$\pi : \mathcal{E} \times F \times D_{\boldsymbol{\mu}} \times D_{\Delta t} \rightarrow \mathcal{E}, \quad (7)$$

as the mathematical operator that advances the latent vector $\varepsilon(t | \boldsymbol{\mu})$ in time according to Eq. (6):

$$\pi(\varepsilon(t_i | \boldsymbol{\mu}), f, \boldsymbol{\mu}, \Delta t_{i+1,i}) = \varepsilon(t_i | \boldsymbol{\mu}) + \int_{t_i}^{t_{i+1}} f(\varepsilon(t | \boldsymbol{\mu}), \boldsymbol{\mu}) dt, \quad (8)$$

with $\Delta t_{i+1,i} = t_{i+1} - t_i$ and the $\boldsymbol{\mu}$ dependency being controlled by f . Clearly, $\pi(\varepsilon(t_i | \boldsymbol{\mu}), f, \boldsymbol{\mu}, \Delta t_{i+1,i}) = \varepsilon(t_{i+1} | \boldsymbol{\mu})$. In summary, φ and ψ describe the mapping between the full order and reduced order representation of the system, while π describes the *dynamics* of the system. For notational convenience, we will **drop** the dependence of π on f .

We now define f_θ as a NN which approximates f and π_θ as the discrete approximation of π which advances in time the vectors belonging to \mathcal{E} by solving the integral of Eq. (8), using known integration schemes (see in Appendix A):

$$\pi_\theta(\varepsilon(t_i | \boldsymbol{\mu}), \boldsymbol{\mu}, \Delta t_{i+1,i}) = \text{ODESolve}(\varepsilon(t_i | \boldsymbol{\mu}), \boldsymbol{\mu}, \Delta t_{i+1,i}), \quad (9)$$

as it is done in Neural ODEs (NODEs) [33,51]. Hence, by approximating f_θ , we approximate the time derivative of $\varepsilon(t | \boldsymbol{\mu})$ and not $\varepsilon(t | \boldsymbol{\mu})$ itself. For example, in the case of the explicit Euler scheme [52]:

$$\pi_\theta(\varepsilon(t_i | \boldsymbol{\mu}), \boldsymbol{\mu}, \Delta t_{i+1,i}) = \varepsilon(t_i | \boldsymbol{\mu}) + \Delta t_{i+1,i} f_\theta(\varepsilon(t_i | \boldsymbol{\mu}), \boldsymbol{\mu}). \quad (10)$$

The Processor π is the equivalent of the Evolution Operator \hat{H} of Eq. (3) but acting on the reduced space \mathcal{E} of **discrete** intrinsic representations: as such π does not need to be equipped with the notion of (spatial) discretization invariance as in the case of Neural Operators.

2.4. Training of the model

The model we defined in Section 2.3 requires the optimization of **two** training processes which we consider **coupled**: the training of the AE which regulates the mappings between S_r and \mathcal{E} and the training of π_θ which regulates the latent dynamics described by Eq. (6). The latter can be approached by combining a *Teacher Forcing (TF)* and an *Autoregressive (AR)* strategy. We thus define:

$$\mathcal{L}_{1,i} = \frac{\|s_r(\mathbf{x}, t_i | \boldsymbol{\mu}) - \psi_\theta \circ \varphi_\theta(s_r(\mathbf{x}, t_i | \boldsymbol{\mu}))\|_2}{\|s_r(\mathbf{x}, t_i | \boldsymbol{\mu})\|_2} \quad (11)$$

as the term which governs the AE training. By introducing

$$\begin{cases} \varepsilon_i^\mu = \varphi_\theta(s(\mathbf{x}, t_i | \boldsymbol{\mu})), \\ \varepsilon_i^{\mu,k} = \pi_\theta(\cdot, \boldsymbol{\mu}, \Delta t_{i,i-1}) \circ \dots \circ \pi_\theta(\varepsilon_{i-k}^\mu, \boldsymbol{\mu}, \Delta t_{i-k+1,i-k}), \end{cases} \quad (12)$$

we define the two terms which govern the latent dynamics:

$$\begin{cases} \mathcal{L}_{2,i}^{T,k_1} = \frac{\|\varepsilon_i^\mu - \varepsilon_i^{\mu,k_1}\|_2}{\|\varepsilon_i^\mu\|_2}, \\ \mathcal{L}_{2,i}^{A,k_2} = \frac{\|\varepsilon_i^\mu - \varepsilon_i^{\mu,i}\|_2}{\|\varepsilon_i^\mu\|_2}, \end{cases} \quad (13)$$

where T identifies the TF approach and A the AR one. The term $\mathcal{L}_{2,i}^{T,k_1}$ (TF), penalizes the difference between the expected latent vector ε_i^μ and the predicted latent vector ε_i^{μ,k_1} obtained by applying autoregressively π_θ to $\varepsilon_{i-k_1}^\mu$, which comes from encoding the **true field** $s_r(\mathbf{x}, t_{i-k_1} | \boldsymbol{\mu})$ (hence the name *Teacher-Forcing*, as the **true** input k_1 steps earlier, is fed into the NN). Using TF when training autoregressive models is known to cause potential *distribution shift* [29], representing a problem at inference time: as depicted in Fig. 1, at testing time the input of π_θ is the previous output of π_θ starting from ε_0^μ , contrary to what $\mathcal{L}_{2,i}^{T,k_1}$ penalizes (unless $k_1 = F$, i.e., the full length of the time series). To avoid this mismatch between training and inference, we introduced $\mathcal{L}_{2,i}^{A,k_2}$ (AR), which penalizes the difference between the expected latent vector ε_i^μ and the predicted latent vector $\varepsilon_i^{\mu,i} = \pi_\theta(\cdot, \boldsymbol{\mu}, \Delta t_{i,i-1}) \circ \dots \circ \pi_\theta(\varepsilon_0^\mu, \boldsymbol{\mu}, \Delta t_{1,0})$ obtained by repeated application of π_θ starting from the encoded representation of the initial condition s_r^0 , as it is done at testing time. k_2 denotes the number of steps in time from which the **gradients of the backpropagation algorithm** flow, i.e., the predicted latent vector at time t_i is obtained by encoding the initial condition s_r^0 and fully evolving it autoregressively (by applying π_θ i times), but the gradients of the backpropagation algorithm **flow only** from the predicted latent vector at time t_{i-k_2} up to t_i . It follows that $\mathcal{L}_{2,i}^{T,k_1}$ and $\mathcal{L}_{2,i}^{A,k_2}$ are computed in the same way only if $k_1 = k_2 = F$. By truncating the gradients flow at time t_{i-k} , we are implementing a form of Truncated Backpropagation Through Time (TBPTT) as it is usually done for gradients stability purposes when training Recurrent

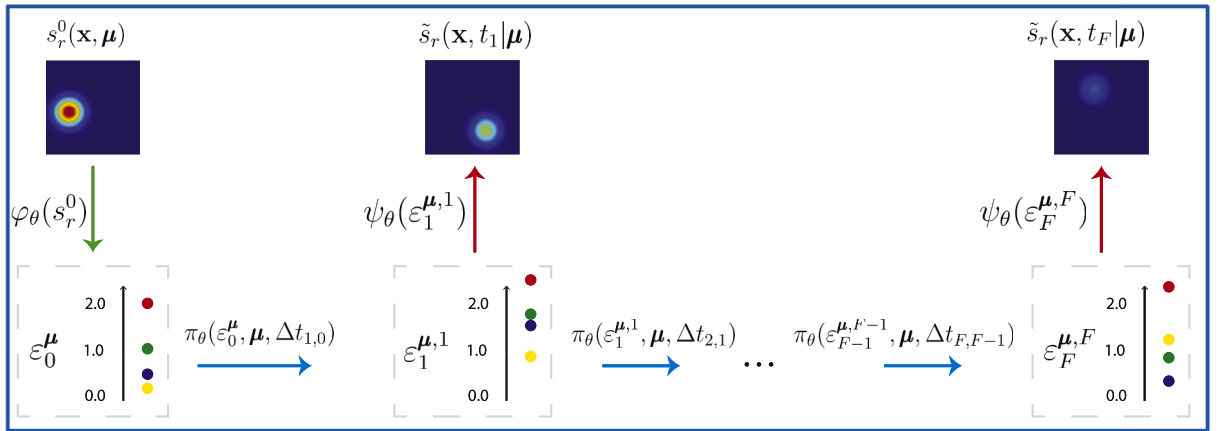


Fig. 1. Workings of our proposed method at **testing** time. The initial condition s_r^0 is mapped through the Encoder φ_θ into its latent representation ε_0^μ . Subsequently the vector ε_0^μ is advanced in time autoregressively by repeated evaluation of the processor π_θ , conditioned to the vector of parameters $\boldsymbol{\mu}$ and to the size of the temporal jump $\Delta t_{i+i,j}$. The Decoder ψ_θ is used to map back each predicted latent vector $\varepsilon_i^{\mu,i}$ into the corresponding field $\tilde{s}_r(\mathbf{x}, t_i | \boldsymbol{\mu})$. Notice that φ_θ is applied only to the initial condition s_r^0 . The colored dots represent the λ different values of the λ -dimensional vector ε for a given moment in time.

Neural Networks (RNNs) [53]. Fig. 1 shows a summary of the method at testing time, where the predicted solution is computed as $\tilde{s}_r(\mathbf{x}, t_i | \boldsymbol{\mu}) = \psi_\theta \circ \pi_\theta(\cdot, \boldsymbol{\mu}, \Delta t_{i,i-1}) \circ \dots \circ \pi_\theta(\cdot, \boldsymbol{\mu}, \Delta t_{1,0}) \circ \varphi_\theta(s_r^0)$.

2.5. Combining teacher forcing with autoregressive

How do we combine AR and TF strategies in practice? We start by considering the loss $\mathcal{L}_{2,i}^{T,1}$, the simplest form of TF strategy with the advantage of being computationally efficient and stable, but at the cost of making the training agnostic to the autoregressive nature of the model at testing time and not addressing the *accumulation of errors* which is typical of autoregressive models. $\mathcal{L}_{2,i}^{A,k_2}$ instead, regardless of the chosen k_2 , already reflects during training the autoregressive modality used at testing; it has however the disadvantage of being computationally more demanding (for $k_2 > 1$) and more difficult to train the larger k_2 is. If $k_1 > 1$, $\mathcal{L}_{2,i}^{T,k_1}$ introduces a certain degree of 'autoregressiveness' as well, although the latent vector at time t_{i-k_1} is still provided by the true solution. Among the several possible training strategies, here we list the two we used, with $\mathcal{L}_{2,i} = \beta \mathcal{L}_{2,i}^{T,k_1} + \gamma \mathcal{L}_{2,i}^{A,k_2}$, where β and γ weigh the importance of the terms:

1. set $\beta = 1$, $\gamma = 0$ and $k_1 = 1$, using only the TF term.
2. set $\beta = 1$, $\gamma = 1$, $k_1 = 1$ and dynamically increase k_2 during the training, starting with $k_2 = 1$. This strategy has the advantage of taking into account the AR term gradually during the training.

Our experiments have shown that although for some systems strategy 1 is enough, more complex datasets require using strategy 2, mainly due to two separate behaviors in our observations. First, that in the **early stages** of the training, $\mathcal{L}_{1,i}$, $\mathcal{L}_{2,i}^{T,1}$ and $\mathcal{L}_{2,i}^{A,1}$ play the important role of building a latent space whose dynamics is described by Eq. (6); and second, that in the **later stages** of the training, with k_2 becoming larger (and the computed gradients more complex), the autoregressive nature of the model is increasingly taken into account, with π_θ becoming more robust to the accumulation of errors.

2.6. Generalization in the time domain

As shown in Fig. A.10 we expect our models to be trained on a given set of time-steps, but we want them to generalize to time-steps not seen during the training phase (such as intermediate times). For this reason, we introduce a **last term** of the loss function as:

$$\begin{cases} \mathcal{L}_{3,i} = \frac{\|\epsilon_i^\mu - \tilde{\epsilon}_i^\mu\|_2}{\|\epsilon_i^\mu\|_2} \\ \tilde{\epsilon}_i^\mu = \pi_\theta(\cdot, (\Delta t_{i,i-1} - \Delta t_{m,i})) \circ \pi_\theta(\epsilon_{i-1}^\mu, \boldsymbol{\mu}, \Delta t_{m,i-1}), \end{cases} \quad (14)$$

where $\Delta t_{m,i-1} \in [0, \Delta t_{i,i-1}]$ is a randomly sampled intermediate time step and $i-1 < m < i$. In Appendix A.1 we further detail $\mathcal{L}_{3,i}$. In some cases we also found it beneficial to add a *regularization* term \mathcal{L}_{rg} to the latent vectors, such as $\mathcal{L}_{rg} = \lambda_{rg} \sum_{i=0}^F \|\epsilon_i^\mu\|_1 / \lambda$, with $\lambda_{rg} \in \mathbb{R}^+$.

Thus, during model training for a given $s_r^0(\mathbf{x}, \boldsymbol{\mu})$, the gradients are computed based on the final loss function of:

$$\begin{aligned} \mathcal{L}_{tr} &= \frac{1}{F} \sum_{i=0}^F \alpha \mathcal{L}_{1,i} + \frac{1}{F} \sum_{i=1}^F \left[\beta \mathcal{L}_{2,i}^{T,k_1} + \gamma \mathcal{L}_{2,i}^{A,k_2} + \delta \mathcal{L}_{3,i} \right] + \mathcal{L}_{rg} \\ &= \alpha \mathcal{L}_1 + \beta \mathcal{L}_2^{T,k_1} + \gamma \mathcal{L}_2^{A,k_2} + \delta \mathcal{L}_3 + \mathcal{L}_{rg}, \end{aligned} \quad (15)$$

where k_1 and k_2 depend on the chosen strategy of Section 2.5 and α , β , γ and δ weigh the importance of each term. We use \mathcal{L}_{tr} for training and \mathcal{L}_{vl} for validation:

$$\mathcal{L}_{vl} = \mathcal{L}_{tr} + \sum_{i=1}^F \frac{\|s_r(\mathbf{x}, t_i | \boldsymbol{\mu}) - \tilde{s}_r(\mathbf{x}, t_i | \boldsymbol{\mu})\|_2}{\|s_r(\mathbf{x}, t_i | \boldsymbol{\mu})\|_2}. \quad (16)$$

Fig. 2 visualizes our training methodology.

3. Results

In this section we compare our method with a series of SOTA methods from Hagnberger et al. [46] and [48]. The datasets we use for comparison are taken from Takamoto et al. [48]. A complete description of the PDEs can be found Appendix F. In Appendix C we list all the training and hyperparameter details and in Appendix D the methods used for comparison. We use as metrics the total error nRMSE, the parametric total error nRMSE($\boldsymbol{\mu}$), the temporal total error nRMSE(t), the parametric neural ODE error NODE-nRMSE($\boldsymbol{\mu}$) and the parametric AutoEncoder error AE-nRMSE($\boldsymbol{\mu}$) defined in Eqs. (F.7), (F.8), (F.11), (F.9) and (F.10).

3.1. PDEs with fixed parameter

In this Section we are going to apply our method to the 1D Advection Eq. (F.1) ($\zeta = 0.1$), to the 1D Burgers' Eq. (F.3) ($\nu = 0.001$) and to the 2D Shallow-Water (SW) Eq. (F.4). In Tables 1 and 3 we compare our results to the ones obtained (on the same dataset), in Takamoto et al. [48] and Hagnberger et al. [46]. In Table 1 we show that our proposed model achieves a **lower nRMSE** compared to

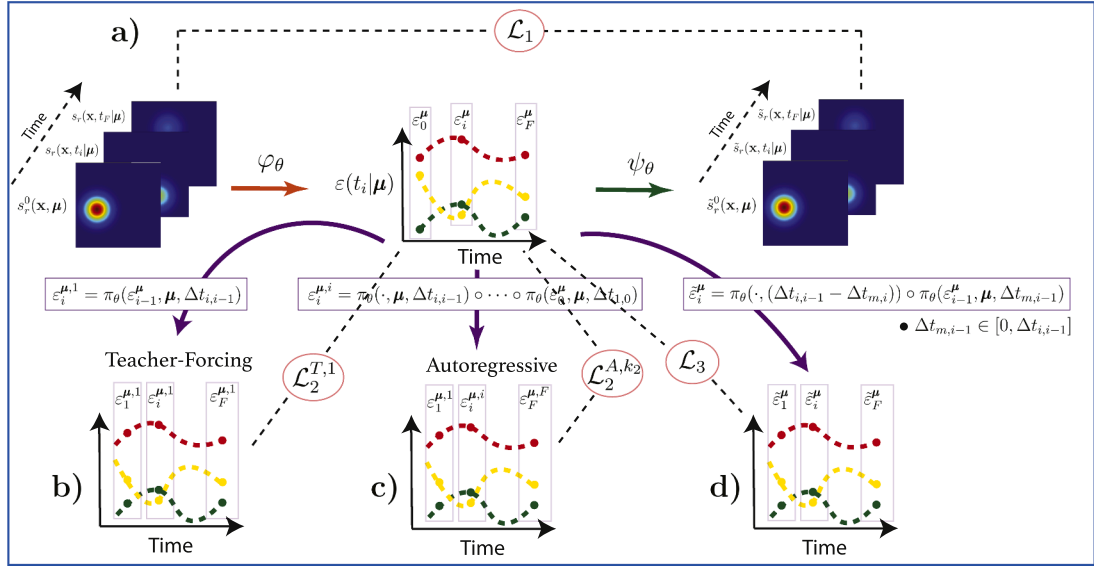


Fig. 2. A representation of the **training** procedure. **a)** The time series of fields $s_r(x, t_i | \mu)$, with $i \in \{0, F\}$, is processed by the Encoder φ_θ and the corresponding latent vectors ε_i^μ are obtained; these are subsequently mapped back to the full space by means of the Decoder ψ_θ which generates the time series of reconstructed fields $\tilde{s}_r(x, t_i | \mu)$, allowing for the computation of \mathcal{L}_1 . **b)** The Processor π_θ receives as input the sequence of latent vectors ε_i^μ with $i \in \{0, F-1\}$ and predicts the latent vectors ε_{i+1}^μ with $i \in \{1, F\}$. $\mathcal{L}_2^{T,1}$, where T stands for *Teacher-Forcing*, is thus computed with inputs ε_i^μ and ε_{i+1}^μ . **c)** The Processor π_θ is applied autoregressively to the initial latent vector ε_0^μ and the whole time series of vectors ε_i^μ is reconstructed with $i \in \{1, F\}$; \mathcal{L}_2^{A,k_2} , where A stands for *Autoregressive*, is thus computed with inputs ε_i^μ and ε_{i+1}^μ . **d)** The Processor π_θ takes as input the sequence of latent vectors ε_i^μ with $i \in \{0, F-1\}$ and outputs for each ε_i^μ an intermediate vector $\varepsilon_m^{\mu,1}$ with a time-step $\Delta t_{m,i-1}$ randomly sampled from $[0, \Delta t_{i,i-1}]$. Last, π_θ advances in time each $\varepsilon_m^{\mu,1}$ with a time-step of $\Delta t_{i,i-1} - \Delta t_{m,i}$ to get the predicted vectors $\tilde{\varepsilon}_i^\mu$; \mathcal{L}_3 is thus computed with inputs ε_i^μ and $\tilde{\varepsilon}_i^\mu$.

Table 1

nRMSE on test dataset for fixed μ and varying s^0 for the 1D Advection and Burgers datasets. The column with \dagger refers to testing with the Δt of the training, while the one with \star with a smaller Δt . Cells are empty when comparison was not found in literature.

PDE	Model	\dagger nRMSE, $\Delta t = 0.05$ s	\star nRMSE, $\Delta t = 0.01$ s
1D Advection	(Ours)	0.0066	0.0066
	FNO	0.0190	0.0258
	MP-PDE	0.0195	
	UNet	0.0079	
	CORAL	0.0198	0.9656
	Galerkin	0.0621	
	OFormer	0.0118	
	VCNeF	0.0165	0.0165
	VCNeF-R	0.0113	
	(Ours)	0.0373	0.0399
1D Burgers	FNO	0.0987	0.1154
	MP-PDE	0.3046	
	UNet	0.0566	
	CORAL	0.2221	0.6186
	Galerkin	0.1651	
	OFormer	0.1035	
	VCNeF	0.0824	0.0831
	VCNeF-R	0.0784	

Table 2

The total error nRMSE on test dataset for fixed μ and varying s^0 for the 2D Shallow-Water dataset.

PDE	Model	nRMSE, $\Delta t = 0.01$ s
2D SW	(Ours)	0.0025
	FNO	0.0044
	U-Net	0.0830
	PINN	0.0170

Table 3

nRMSE on test dataset for fixed μ and varying s^0 for the 2D Shallow-Water dataset. The columns with \dagger refer to testing with the Δt of the training, while the one with \star with a smaller Δt . Our model is trained on $\Delta t = 0.05$ s, while the others on $\Delta t = 0.01$ s.

PDE	Model	\dagger nRMSE, $\Delta t = 0.05$ s	\dagger nRMSE, $\Delta t = 0.01$ s	\star nRMSE, $\Delta t = 0.01$ s
2D SW	(Ours)	0.0028		0.0032
	FNO		0.0044	
	U-Net		0.0830	
	PINN		0.0170	

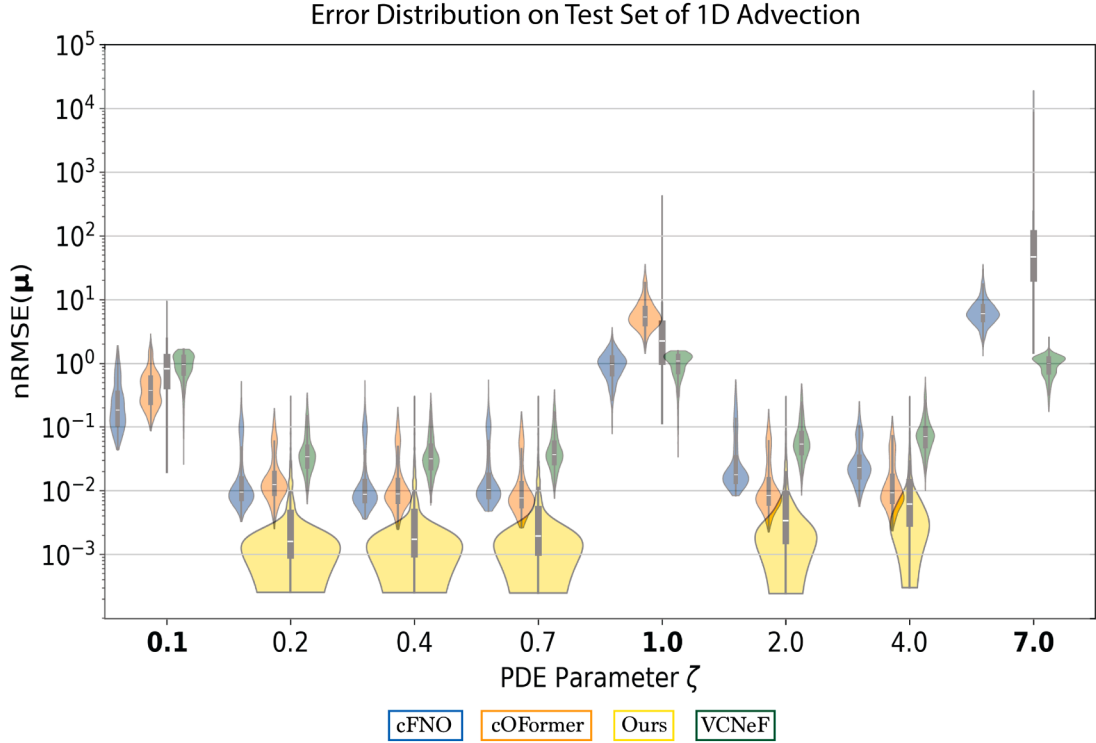


Fig. 3. Distribution of the $\text{nRMSE}(\mu)$ across the test sample for the parametric 1D Advection. Regular font on the x axes refers to training parameter values, while bold ones to testing parameters (but in both cases testing initial conditions). We compare our methodology (yellow) with other published methods (taken from Hagnberger et al. [46]). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

a series of common methods used in Scientific Machine Learning for the 3 cases. Furthermore, we observe that our model achieves a better **generalization in time** than the other methods in the Burgers' and Advection cases, meaning that we got little to none increase of the nRMSE when going from testing on the training $\Delta t = 0.05$ s to testing on a smaller $\Delta t = 0.01$ s. For the SW dataset, in Table 2 we show how our model performs when trained and tested on $\Delta t = 0.01$ s. In addition, in Table 3 we show that even if our model is trained with $\Delta t = 0.05$ s while the others with $\Delta t = 0.01$ s, we still get a lower nRMSE when testing on $\Delta t = 0.01$ s (thus the comparison on the same number of time-steps is only between our method in the column with \star and the other methods in the columns with \dagger). For the experiments in this section we used Strategy 1 of Section 2.5, as using $\mathcal{L}_{2,i}^{T,1}$ alone was sufficient to reach acceptable results.

3.2. PDEs with varying parameters

In this section we experiment with 3 datasets where we both vary the initial conditions and the PDE parameters: 1D Advection Eq. (F.1), the 1D Burgers' Eq. (F.3) and the 2D Molenkamp Test (F.5). In all three cases we use Strategy 2 of Section 2.5, since only using $\mathcal{L}_{2,i}^{T,1}$ optimized correctly \mathcal{L}_{tr} but resulted in a larger value of \mathcal{L}_{vl} . We start with $k_2 = 1$ and we increase it by 1 every 30 epochs until the maximum length of the time series is reached. We thus define $p(k_2)$ as the number of epochs needed to increase k_2 by 1. To make the training more stable, we introduce gradually the $\gamma \mathcal{L}_{2,i}^{A,k_2}$ term by starting with a $\gamma = \gamma_0 < 1$ and increasing it every epoch by an amount of γ_0 until $\gamma = 1$. In Figs. 3 and 4 we show a comparison of our methodology (yellow) with the cFNO, cOFormer and VCNeF

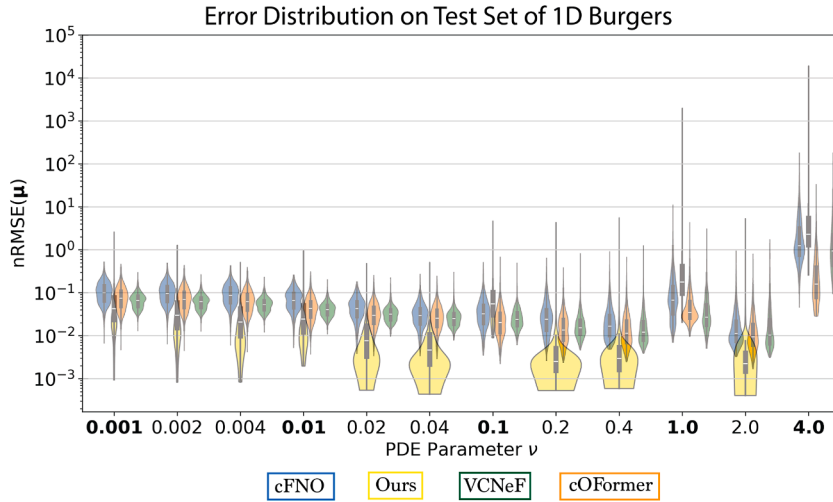


Fig. 4. Distribution of the $nRMSE(\mu)$ across the test sample for the parametric 1D Burgers'. Regular font on x axes refers to training parameters, while bold ones to testing parameters (but in both cases testing initial conditions). We compare our methodology (yellow) with other published methods (taken from Hagnberger et al. [46]). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

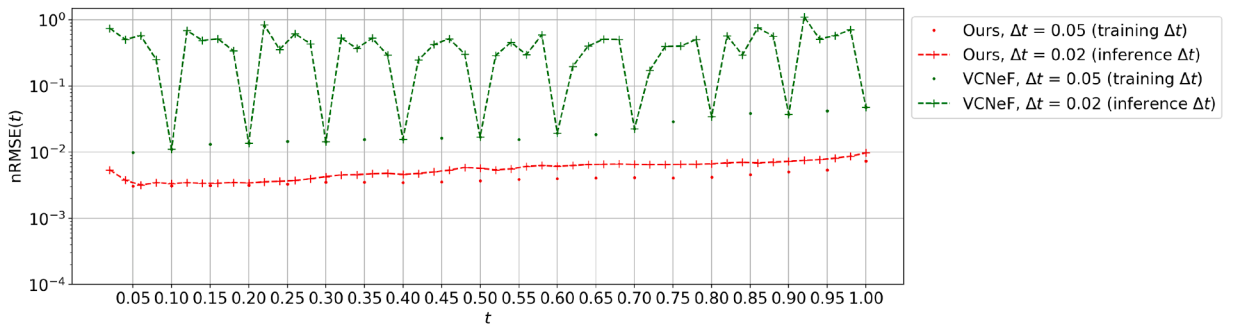


Fig. 5. Comparison of the temporal error $nRMSE(t)$ between our model (red) and the VCNeF (green) for the test dataset of the Molenkamp application. We study the difference between applying at inference the same Δt as used for the training ($\Delta t = 0.05$ s) and a smaller one $\Delta t = 0.02$ s. The $nRMSE(t)$ of our model only slightly increases when decreasing the Δt , while VCNeF struggles with inference at intermediate time-steps. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

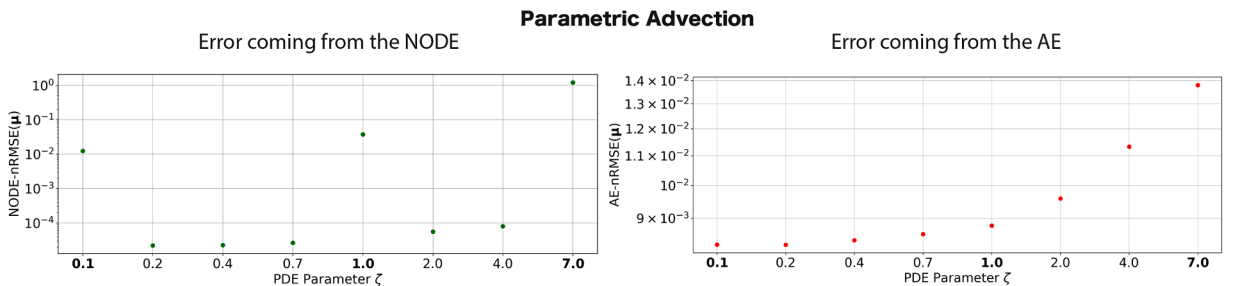


Fig. 6. Comparison of the parametric Neural ODE error and the parametric AutoEncoder error for the parametric Advection problem. **Left figure:** the NODE-nRMSE on the latent vectors predicted by the NODE is computed on the test set. The error is larger on the testing parameters (bold ones), signaling a struggle of the NODE to correctly reconstruct the dynamics of unseen parameters, both in interpolation and in extrapolation. **Right figure:** the AE-nRMSE on the solution fields is computed by applying consecutively the Encoder and the Decoder on the test set. While the error is increasing with increasing velocity ζ , the AutoEncoder does not struggle with testing parameters.

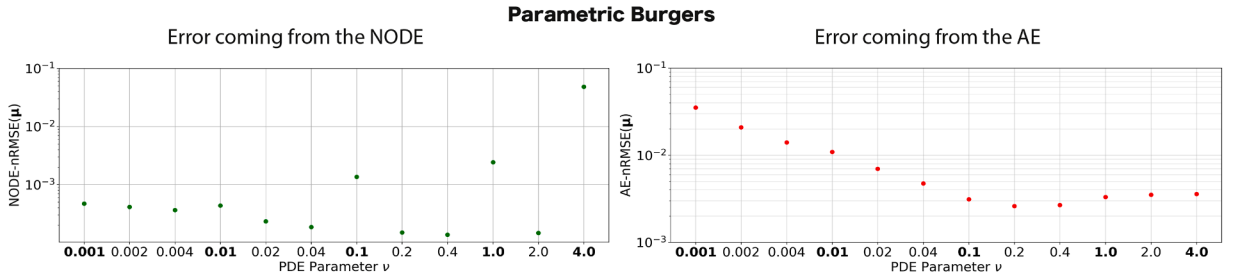


Fig. 7. Comparison of the parametric Neural ODE error and the parametric AutoEncoder error for the parametric Burgers' problem. **Left figure:** the NODE-nRMSE on the latent vectors predicted by the NODE is computed on the test set. The error is larger on the testing parameters (bold ones) for larger parameters. For smaller testing parameters like $\nu = 0.001, 0.01$ there is no noticeable increase in error. **Right figure:** the AE-nRMSE on the solution fields is computed by applying consecutively the Encoder and the Decoder on the test set. There is no correlation between the error and ν being used or not in the training.

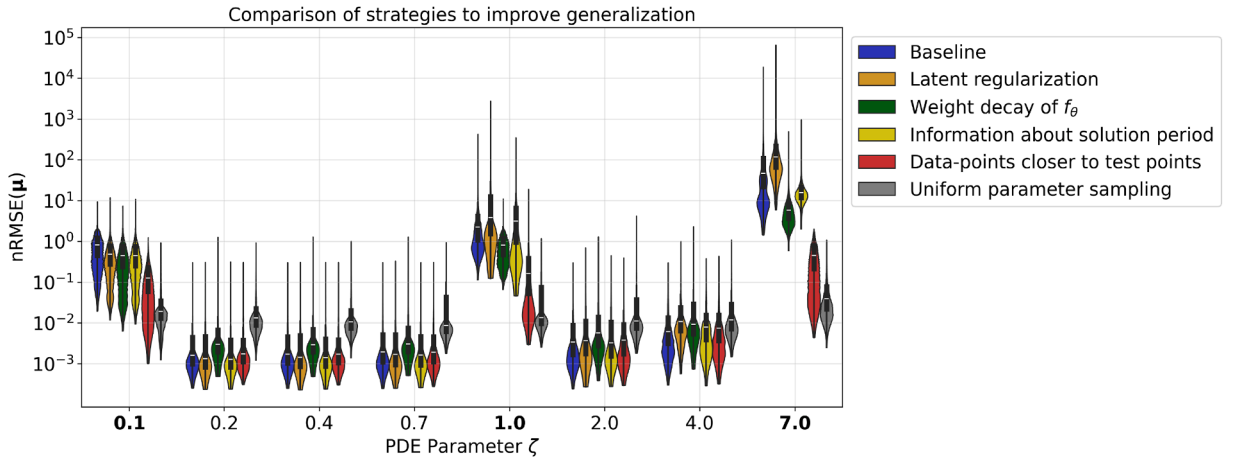


Fig. 8. Performance comparison of 5 training strategies to the 'Baseline' of Fig. 3 to overcome poor parameter generalization (in interpolation and extrapolation) of the NODE for the parametric Advection dataset. Only strategies that use more data during the training close to $\zeta = 0.1, 1.0, 7.0$ ('Data-points closer to test points' and 'Uniform parameter sampling') improve the prediction at test point (bold characters).

from Hagnberger et al. [46]. In both cases, we show the distribution across the test samples of training (regular font) and testing (bold font) parameters when using **testing initial conditions**. From Fig. 3, we see that our model has a lower median than the compared methods on the training velocities ζ , while it struggles with testing parameters, similar to cFNO, cOFormer and VCNeF too. This is likely a dataset issue yielding insufficient generalization, with 0.1 and 7.0 both being outside the training range and 1.0 possessing a dynamic not easily interpolated by f_θ with the information coming from the points 0.7 and 2.0. Similarly for the Burgers' case in Fig. 4, the median of the nRMSE given by our model is lower than the compared methods for all parameters ν except $\nu = 1.0$ and $\nu = 4.0$. In this case our model - similar to the ones used for comparison - is able to generalize better than in the Advection example to test parameters, as in the case of $\nu = 0.001$ and $\nu = 0.01$. Given the discrepancy in the ability of the models to generalize to different testing parameters, more accurate strategies for adaptively selecting parameter points for training should be researched. In Figs. 3 and 4, although we show also training parameters, the corresponding initial conditions **belong to the testing set**, thus we are extrapolating on the initial conditions even when the parameters are the ones used during training. In Fig. 5 we compare our method with VCNeF on the Molenkamp test when testing on $\Delta t = 0.05$ s (same as the one in training) and when $\Delta t = 0.02$ s: our method achieves a lower nRMSE and is able to generalize to intermediate time points better than VCNeF. Noticeably, with the Molenkamp test we use a latent space of dimension $\lambda = 5$, which is equal to the actual number of degrees of freedom of the PDE solution (Eq. (F.5)). In Appendix C.4, we compare the number of parameters and the inference speed of the methods used and we show that our proposed method is **lighter** and **faster** at inference.

3.3. Discussion

Although in Tables 1–3 and Figs. 3–5 we have shown that our method achieves a comparable or lower nRMSE when compared to other methods, there are noticeable issues that must be addressed. In particular, Figs. 3 and 4 signal problems in generalization to unseen parameters both in interpolation (within training range) and extrapolation (outside training range). In order to properly analyze such model failures we first need to **decouple two error sources**: the error coming from the AE and the error coming

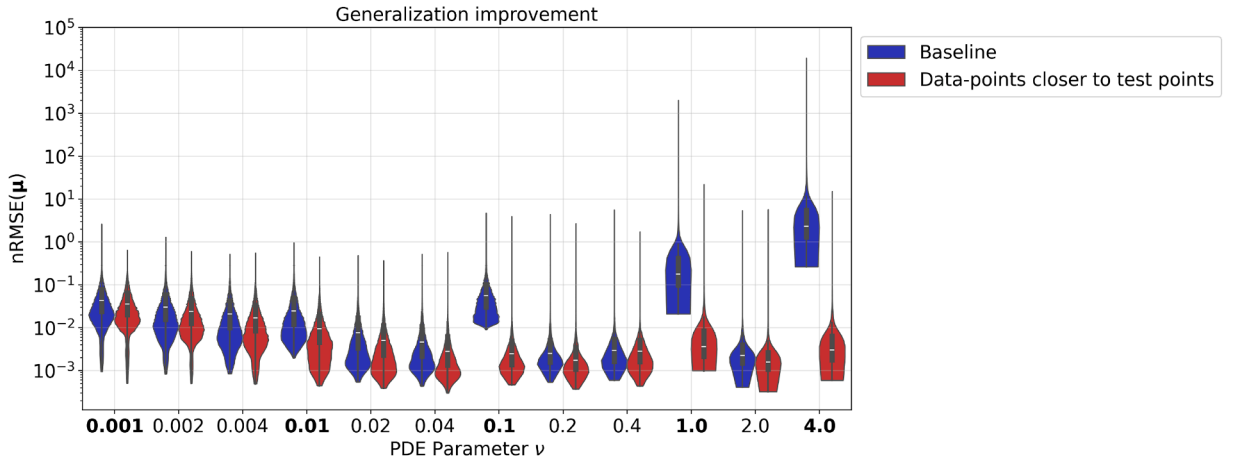


Fig. 9. Adding more training data-points closer to $\nu = 0.1, 1.0, 4.0$ at training improves the generalization to test parameters (bold characters) for the parametric Burgers dataset of Fig. 4.

from the NODE. In Fig. 6 we show on the left the parametric Neural ODE error $\text{NODE-nRMSE}(\mu)$ (Eq. (F.9)) at inference computed between the latent vectors predicted by the NODE and the ones computed by the Encoder on the testing set: the error is larger on the testing parameters (bold characters). Conversely, on the right of Fig. 6 we show the $\text{AE-nRMSE}(\mu)$ (Eq. (F.10)) computed when applying the Encoder and then the Decoder on the test set: the error is not larger for testing parameters, while there is a correlation with the magnitude of the velocity ζ . We conduct an analogous study on the parametric Burgers' case in Fig. 7: again, the error on unseen parameters is coming from a poor prediction of the NODE. Interestingly, in the left plot of Fig. 7 we see that the error on the unseen parameters is considerably larger for values of $\nu \geq 0.1$: because for values smaller than 0.1 the sampling is finer, this suggests that a **finer sampling** of the parameter would improve the generalization. From this decoupling experiment we conclude that the generalization issues (both in extrapolation and interpolation) of our model are coming from a **low generalization power of the NODE**. Furthermore, generalization in *interpolation* and *extrapolation* are equally affected by the sampling: in Fig. 4 our model can extrapolate at $\nu = 0.001$ but fails to do so at $\nu = 4.0$. However, the distance from the point $\nu = 4.0$ to the closest point included in the training set is 2.0, while the distance of the closest point to $\nu = 0.001$ belonging to the training set is 0.001. Similarly, the point $\nu = 0.01$ is well interpolated, while $\nu = 1.0$ is not (but again, they are at different distances from their closest point belonging to the training set.)

Focusing firstly on the parametric Advection dataset, we test 5 approaches to overcome the generalization issue: *latent regularization*, *weight decay of the latent space evolution ODE source term f_θ* , *information about solution period* of the Advection equation through positional encoding of time and *data-points closer to test points* $\zeta = 0.1, 1.0, 7.0$ (more details are given in Appendix C.1) and *uniform parameter sampling*. In Fig. 8, we compare the 5 approaches: only using training data closer to $\zeta = 0.1, 1.0, 7.0$ reduces the test error, although we see that for 'Data-points closer to test points', there is still a significant gap in the $\text{nRMSE}(\mu)$ between testing parameters and training parameters. Interestingly, when 'Uniform parameter sampling is used', we notice a stabilization of the $\text{nRMSE}(\mu)$ across parameters at the expense of the $\text{nRMSE}(\mu)$ for training parameters which increases: this signals a potential overfitting regime in the baseline, where the model is hyper-optimized for the training parameters.

In Fig. 9 we perform a sampling closer to the test points for the parametric Burgers' case as well (more details are provided in Appendix C.1). For this dataset we see that adding training points closer to the test points reduces the $\text{nRMSE}(\mu)$ on test parameters much more than in the Advection case.

In conclusion, the experiments performed in this Section highlighted two elements: the generalization error (both in interpolation and extrapolation) is caused by the NODE and intelligent data sampling must be adopted in order to overcome this issue. Furthermore, we see that generalization is much easier for the Burgers' case rather than the Advection case: this may signal a struggle of our method to generalize well in the case of transport-like phenomena, potentially due to the *spectral bias phenomena* addressed in Anderson et al. [45], i.e., the tendency of NN architectures to approximate better low frequency signals. We see from the right plot of Fig. 6 that the AutoEncoder indeed approximates better low frequency signals (the period of the solution is given by $0.5/\zeta$); however we do not see such phenomenon on the left plot of Fig. 6, where the error at $\zeta = 0.1$ is much larger than the error at $\zeta = 4.0$. Such observations, together with the findings of Fig. 8, indicate that the sampling is more of an issue than the spectral bias.

3.4. Ablation studies

In Appendix E.1 we conduct ablation studies regarding how the choice of the ODE solver and $\mathcal{L}_{3,i}$ impact the capabilities of the method and the generalization in time. In Appendix E.2 we show sensitivity studies concerning γ_0 and $p(k_2)$ for the Autoregressive training. In Appendix E.3 we experiment with decoupling the AE from the NODE during training.

4. Conclusions

In this work we showed how **Dimensionality Reduction** and **Neural ODEs** can be coupled to construct a surrogate model of time-dependent and parametric PDEs. Our model inherits from these two paradigms two important features which are desiderata when building DL models that substitute standard numerical solvers, i.e., **fast computational inference** and **continuity in time**. The former is achieved thanks to the low dimensionality of the reduced space \mathcal{E} , while the latter by the definition of the latent dynamics through the ODE of Eq. (6). In Section 3 we showed that our methodology **surpasses in accuracy** several state of the art methods on different benchmarks used in the Scientific Machine Learning field. In addition, our model requires **significantly less NN's weights** (thus less memory) and is **computationally faster at inference** compared to other published methodologies (Tables C.8 and C.9); for these reasons relying on *dimensionality reduction* as opposed to *large and overparametrized* architectures is going to be key in the future for building fast and memory efficient surrogate models of complex physical systems.

5. Limitations and future research directions

The main limitation of our method is the use of CNNs for Encoding and Decoding, which hinders its applicability to non-uniform meshes and makes it necessary to re-train models if inference needs to be done on grid points not used at training; future research will explore using Neural Operators for the construction of the Encoder and the Decoder. Another aspect which should be improved concerns the definition of an efficient sampling strategy as done in He et al. [42], Bonneville et al. [43] to determine which PDE parameters should be used during training in order to be able to overcome the poor generalization to new ones for some datasets, as evident from 3.2 and 3.3. Finally, while leaving the construction of \mathcal{E} and definition of the function f_θ general gives flexibility to the fitting of the training dataset, researching into the *interpretability* of both \mathcal{E} and f_θ can at the same time improve our understanding of NNs and build more accurate surrogate models: for example in Cha and Thiyagalingam [54] a method is proposed to disentangle the latent space dimensions in order to obtain the true generative factors of the high-dimensional images (in the surrogate modeling case those would be the generative factors of the PDE solution); especially interesting would be combining interpretability with physical constraints, as it is done in Park et al. [44], where the latent dynamics is forced to respect the first and second laws of thermodynamics.

Source code

Source code is available at https://github.com/Aleartulon/AE_NODE.

CRediT authorship contribution statement

Alessandro Longhi: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Danny Lathouwers:** Supervision, Resources, Project administration, Methodology, Funding acquisition, Formal analysis, Conceptualization; **Zoltán Perkó:** Conceptualization, Methodology, Formal analysis, Resources, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Data availability

Most data used are publicly available (links are given in the paper). The ones not available can be made so upon request.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Alessandro Longhi reports financial support was provided by European Union. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Funded by the European Union under the grant agreement no. 101059682. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission-Euratom. Neither the European Union nor the granting authority can be held responsible for them. This work has been developed within ASSAS (Artificial intelligence for Simulation of Severe AccidentS), a Horizon Europe funded project targeting the development of nuclear severe accident simulators.

Appendix A. Runge–Kutta schemes

Runge–Kutta methods [52] are a family of numerical methods for the solution of Ordinary Differential Equations (ODEs). They belong to the category of *one-step* methods, as such they do not use any information from previous time steps. Given $t_i \in \mathbf{T}$, a standard explicit Runge–Kutta method would solve Eq. (6) as:

$$\varepsilon(t_{i+1}|\boldsymbol{\mu}) = \varepsilon(t_i|\boldsymbol{\mu}) + \Delta t_{i+1,i} \sum_{j=1}^q h_j b_j, \quad (\text{A.1})$$

where q is called the *stage* of the Runge–Kutta approximation, $\Delta t_{i+1,i} = t_{i+1} - t_i$ and:

$$\begin{aligned} b_1 &= f(\varepsilon(t_i|\boldsymbol{\mu}), t_i, \boldsymbol{\mu}), \\ b_2 &= f(\varepsilon(t_i|\boldsymbol{\mu}) + (a_{2,1} b_1) \Delta t_{i+1,i}, t_i + c_2 \Delta t_{i+1,i}, \boldsymbol{\mu}), \\ &\vdots \\ b_k &= f(\varepsilon(t_i|\boldsymbol{\mu}) + \sum_{l=1}^{k-1} a_{k,l} b_l \Delta t_{i+1,i}, t_i + c_k \Delta t_{i+1,i}, \boldsymbol{\mu}). \end{aligned}$$

The matrix composed by a_{ij} is known as Runge–Kutta matrix, h_j are the weights and c_j are the nodes, with their values given by the Butcher tableau [55].

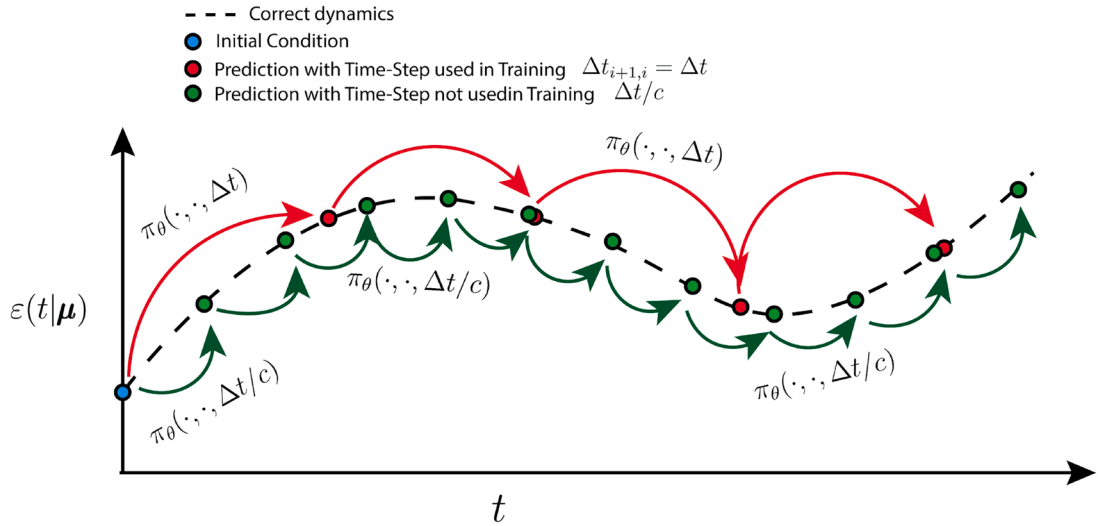


Fig. A.10. Time evolution of a one dimensional $\varepsilon(t|\boldsymbol{\mu})$ is shown (dot line). The red points indicate the steps in time used during the training, at intervals of $\Delta t_{i+1,i}$, and the green points show the points in time that can be predicted at testing time at distance of $\Delta t_{i+1,i}/\alpha$, where $\alpha \in [1, \infty)$. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

A.1. The effect of the stage of RK on time generalization

In this work we use a fixed Runge–Kutta time stepper, and to go from the state i to the state $i + 1$ we do not step through intermediate states. We defined the Processor of Eq. (7) with a $\Delta t_{i+1,i}$ dependency as we want to perform inference even at temporal discretizations finer than the one used at training. Although this task may look trivial as π_θ directly takes $\Delta t_{i+1,i}$ as input, it raises the following issue when solving the ODE. Let us consider a processor π_θ which evolves in time the latent vector $\varepsilon(t|\boldsymbol{\mu})$ using an Euler integration scheme (from here on we omit the $\boldsymbol{\mu}$ dependency for ease of reading):

$$\varepsilon(t_{i+1}) = \varepsilon(t_i) + \Delta t_{i+1,i} f_\theta(\varepsilon(t_i)), \quad (\text{A.2})$$

and let us define a moment in time t_m such that $t_i < t_m < t_{i+1}$. We want f_θ to satisfy the following conditions:

$$\begin{cases} \varepsilon(t_{i+1}) = \varepsilon(t_i) + \Delta t_{i+1,i} f_\theta(\varepsilon(t_i)), \\ \varepsilon(t_{i+1}) = \varepsilon(t_m) + (\Delta t_{i+1,i} - \Delta t_{m,i}) f_\theta(\varepsilon(t_m)), \end{cases} \quad (\text{A.3})$$

where $\varepsilon(t_m) = \varepsilon(t_i) + \Delta t_{m,i} f_\theta(\varepsilon(t_i))$. By taking the difference of the two Equations of (A.3) we get that

$$f_\theta(\varepsilon(t_i)) = f_\theta(\varepsilon(t_m)), \quad (\text{A.4})$$

i.e., when we use the Euler scheme f_θ must be a constant if we want π_θ to be coherent with its predictions at the variation of $\Delta t_{i+1,i}$. Notice that a constant f_θ would imply a linear time dependence of $\varepsilon(t)$ (the dotline of Fig. A.10 would be a line), meaning that the construction of \mathcal{E} would be subjected to a **strong constraint**, thus limiting the expressiveness of the AE. For a RK method of order 2 instead:

$$\varepsilon(t_{i+1}) = \varepsilon(t_i) + \Delta t_{i+1,i} f_\theta \left(\varepsilon(t_i) + \frac{1}{2} \Delta t_{i+1,i} f_\theta(\varepsilon(t_i)) \right), \quad (\text{A.5})$$

where f_θ is evaluated at $\Delta t_{i+1,i}/2$. We want f_θ to respect the following system:

$$\begin{cases} \varepsilon(t_{i+1}|\boldsymbol{\mu}) = \varepsilon(t_i|\boldsymbol{\mu}) + \Delta t_{i+1,i} f_\theta(\varepsilon(t_i^{i+1}|\boldsymbol{\mu})), \\ \varepsilon(t_{i+1}|\boldsymbol{\mu}) = \varepsilon(t_i|\boldsymbol{\mu}) + \Delta t_{m,i} f_\theta(\varepsilon(t_i^m|\boldsymbol{\mu})) + (\Delta t_{i+1,i} - \Delta t_m) f_\theta(\varepsilon(t_m^{i+1}|\boldsymbol{\mu})), \end{cases} \quad (\text{A.6})$$

where $t_i < t_m < t_{i+1}$, $t_k^j = \frac{t_j+t_k}{2}$ and $\varepsilon(t_k^j|\boldsymbol{\mu}) = \varepsilon(t_k|\boldsymbol{\mu}) + \frac{\Delta t_{j,k}}{2} f_\theta(\varepsilon(t_k|\boldsymbol{\mu}))$. By taking the difference of System (A.6) we get:

$$\Delta t_{i+1,i} f_\theta(\varepsilon(t_i^{i+1}|\boldsymbol{\mu})) = \Delta t_{m,i} f_\theta(\varepsilon(t_i^m|\boldsymbol{\mu})) + (\Delta t_{i+1,i} - \Delta t_m) f_\theta(\varepsilon(t_m^{i+1}|\boldsymbol{\mu})), \quad (\text{A.7})$$

which, by going back to full notation, results in the following Equation:

$$\begin{aligned} \Delta t_{i+1,i} f_\theta \left[\varepsilon(t_i|\boldsymbol{\mu}) + \frac{\Delta t_{i+1,i}}{2} f_\theta(\varepsilon(t_i|\boldsymbol{\mu})) \right] &= \Delta t_{m,i} f_\theta \left[\varepsilon(t_i|\boldsymbol{\mu}) + \frac{\Delta t_{m,i}}{2} f_\theta(\varepsilon(t_i|\boldsymbol{\mu})) \right] + \\ &+ (\Delta t_{i+1,i} - \Delta t_m) f_\theta \left[\varepsilon(t_i|\boldsymbol{\mu}) + \frac{\Delta t_{m,i}}{2} f_\theta(\varepsilon(t_i|\boldsymbol{\mu})) + \frac{\Delta t_{i+1,m}}{2} f_\theta \left[\varepsilon(t_i|\boldsymbol{\mu}) + \frac{\Delta t_{m,i}}{2} f_\theta(\varepsilon(t_i|\boldsymbol{\mu})) \right] \right], \end{aligned} \quad (\text{A.8})$$

where $\varepsilon(t_i|\boldsymbol{\mu}) + \frac{\Delta t_{m,i}}{2} f_\theta(\varepsilon(t_i|\boldsymbol{\mu})) = \varepsilon(t_m|\boldsymbol{\mu})$. The constraint to which f_θ is now subjected allows for a more complex form of f_θ which in turns results in a reduced space \mathcal{E} **more capable of adapting** to the complexity of the original space S . It follows that, if we require f_θ to generalize to a variable $\Delta t_{i+1,i}$, the higher the stage of the Runge–Kutta scheme used, the more complex f_θ can be and the more complex the reduced space \mathcal{E} can be.

Appendix B. Architecture details

As detailed in Section 2.3, our model is made up of three components: an Encoder φ_θ , a Processor π_θ and a Decoder ψ_θ .

Table B.4

The structure of the Encoder φ_θ layer by layer, for a 2D case. m is the dimension of the solution field $s(\mathbf{x}, t|\boldsymbol{\mu})$, N is the size of the spatial discretization of the field in the x and in the y axis, $Fe = [Fe_1, \dots, Fe_{L-2}]$ is the vector of convolutional filters, $Ke = [Ke_1, \dots, Ke_{L-2}]$ is the vector of kernels, $j \in \{2, L-2\}$ and λ is the latent dimension. The Flat layer takes all the features coming from the last Convolutional layer and flattens them in a 1D vector. This example is easily reduced to the 1D case.

Layer Number	Input size	Output size	Filters	Kernel	Stride
1 (Convolutional)	$[m, N, N]$	$[Fe_1, N, N]$	Fe_1	$[Ke_1, Ke_1]$	$[1, 1]$
2 (Convolutional)	$[Fe_1, N, N]$	$[Fe_2, \frac{N}{2}, \frac{N}{2}]$	Fe_2	$[Ke_2, Ke_2]$	$[2, 2]$
\vdots					
j (Convolutional)	$[Fe_{j-1}, \frac{N}{2^{j-2}}, \frac{N}{2^{j-2}}]$	$[Fe_j, \frac{N}{2^{j-1}}, \frac{N}{2^{j-1}}]$	Fe_j	$[Ke_j, Ke_j]$	$[2, 2]$
\vdots					
$L-1$ (Flat layer)					
L (Linear)	$[Fe_{L-2} \times \frac{N}{2^{L-3}} \times \frac{N}{2^{L-3}}]$	$[\lambda]$			

Encoder. We build φ_θ as a series of Convolutional layers [56] followed by a final Linear layer as in the 2D example of Table B.4.

The first layer has stride 1 to do a preprocessing of the fields and the subsequent layers up to the Flat layer halve each spatial dimension by 2. We use as activation function after each Convolutional layer the GELU function [57] and we **do not** use any activation function after the final Linear layer to not constrain the values of the latent space. We experimented with Batch Normalization [58] and Layer Normalization layers [59] between the Convolutional layers and the GELU function but we did not notice any improvements in the results. The weights of all the layers are initialized with the Kaiming (uniform) initialization [60]. Notice that we are not using any Pooling layer [61] to reduce the dimensionality but only strided Convolutions, as Pooling layers would enforce translational invariance which is not always a desired property.

Processor. Inside the Processor π_θ , in practice only the function f_θ which approximates f of Eq. (6) is parametrized by a NN with $f_\theta(\varepsilon(t|\boldsymbol{\mu}), \boldsymbol{\mu})$ being a function of both $\varepsilon(t|\boldsymbol{\mu})$ and $\boldsymbol{\mu}$. We experimented with the parameter dependency in two ways:

- $\boldsymbol{\mu}$ is simply concatenated to the reduced vector $\varepsilon(t|\boldsymbol{\mu})$. In this case $f_\theta : \mathbb{R}^{\lambda+z} \rightarrow \mathbb{R}^\lambda$, where z is the dimensionality of $\boldsymbol{\mu}$ and λ the latent dimension.

Table B.5The structure of the Decoder ψ_θ layer by layer, for a 2D case. 'T.' stands for 'Transposed'.

Layer Number	Input size	Output size	Filters	Kernel	Stride
1 (Linear)	$[\lambda]$	$[Fd_1 \times \frac{N}{2^{L-3}} \times \frac{N}{2^{L-3}}]$			
2 (Reshape layer)					
\vdots					
j (T. Convolutional)	$[Fd_{j-2}, \frac{N}{2^{L-j}}, \frac{N}{2^{L-j}}]$	$[Fd_{j-1}, \frac{N}{2^{L-j-1}}, \frac{N}{2^{L-j-1}}]$	Fd_{j-1}	$[Kd_{j-1}, Kd_{j-1}]$	$[2, 2]$
\vdots					
$L-1$ (T. Convolutional)	$[Fd_{L-3}, \frac{N}{2}, \frac{N}{2}]$	$[Fd_{L-2}, N, N]$	Fd_{L-2}	$[Kd_{L-3}, Kd_{L-3}]$	$[2, 2]$
L (T. Convolutional)	$[Fd_{L-2}, N, N]$	$[m, N, N]$	$[m]$	$[Kd_{L-2}, Kd_{L-2}]$	$[1, 1]$

- The vector $\varepsilon(t|\mu)$ is conditioned to μ through a FiLM layer [62]. This means defining the function $\alpha : \mathbb{R}^z \rightarrow \mathbb{R}^\lambda$ and the function $\tau : \mathbb{R}^z \rightarrow \mathbb{R}^\lambda$. The input to f_θ will thus be $\alpha(\mu) \odot \varepsilon(t|\mu) + \tau(\mu)$, where \odot is the Hadamard product. α and τ are chosen to be simple Linear layers.

In both cases f_θ is built as a sequence of Linear layers followed by the GELU activation function. Importantly the activation function is not used after the last Linear layer as this is a regression problem.

Decoder. We build ψ_θ as a Linear layer followed by a series of Transposed Convolutional layers [56] as shown in Table B.5. The initial Linear layer and the last Transposed Convolutional layer are **not** followed by an activation function while the other Transposed Convolutional layers are followed by a GELU function. We do not use any activation function for the Linear layer for symmetry with the Encoder, while for the last layer of the Decoder because this is a regression task. After the Reshape layer each Transposed (T.) Convolutional layer doubles the dimensionality in both x and y dimensions until the layer number $L-1$. The last layer does not increase the dimensionality of the input (stride = 1) and is just used to go to the final dimensionality m of the solution field s . $Fd = [Fd_1, \dots, Fd_{L-2}]$ is the vector of convolutional filters, $Ke = [Kd_1, \dots, Kd_{L-2}]$ is the vector of kernels, $j \in \{3, L-1\}$.

B.1. Normalization of the inputs

In order to facilitate the training process we normalize the inputs, as standard Deep Learning practice. We use a max-min normalization both for the input fields s and for the parameters μ . We do not normalize $\Delta t_{i+1,i}$. By max-min normalization, we mean the following: given an input y , we transform it accordingly to $y \rightarrow \frac{y - \min(D_y)}{\max(D_y) - \min(D_y)}$, where $\max(D_y)$ and $\min(D_y)$ are computed over the training datasets D_y to which y belongs. In our experiments s is a scalar field so we only compute one tuple $(\max(D_s), \min(D_s))$. In the case of μ instead, since each parameter μ_i can belong to a different scale, we compute z tuples $(\max(D_{\mu_i}), \min(D_{\mu_i}))$ with $i \in \{1, z\}$. We normalize accordingly the parameters for all the datasets while the input fields for all the datasets but the Burgers' Equation and the parametric Advection.

Appendix C. Training and hyperparameter details

In all the experiments we use the Adam optimizer [63] and we stop the training if the validation loss has not decreased for 200 epochs. We use an Exponential Learning Rate Scheduler, with a decay parameter γ_{lr} . We set 5000 as the maximum number of epochs. In all the experiments, unless otherwise specified, we used $q = 4$ as the stage of the RK algorithm. We use \mathcal{L}_{tr} for training and \mathcal{L}_{vl} for validation, defined as

$$\mathcal{L}_{vl} = \mathcal{L}_{tr} + \sum_{i=1}^F \frac{\|s_r(\mathbf{x}, t_i|\mu) - \tilde{s}_r(\mathbf{x}, t_i|\mu)\|_2}{\|s_r(\mathbf{x}, t_i|\mu)\|_2}, \quad (\text{C.1})$$

with $s_r(\mathbf{x}, t_i|\mu)$ being the ground truth solution at time t_i and parameters μ .

In Table C.6 we detail the training-validation-test splits and the hyperparameters of our model for the experiments of Section 3.1.

In Table C.7 we detail the training-validation-test splits and the hyperparameters of our model for the experiments of Section 3.2.

C.1. Strategies for the improvement of parameter generalization

In Section 3.3 we experiment with possible methods to improve the generalization abilities of our model to testing parameters. Fig. 8 compares the performance of the following five strategies to that of the baseline in Fig. 3 for the parametric Advection dataset:

1. **Latent regularization:** we increase the value of λ_{rg} (set to 0 in the baseline) to check whether the generalization benefits from suppressing unimportant degrees of freedom of the latent space.
2. **Weight decay of latent space evolution ODE source term f_θ :** we set the 'weight decay' parameter of the Adam optimizer to $1e-6$ (lower would largely increase the nRMSE) for the NN weights of f_θ to prevent overfitting it to the training data.
3. **Information about solution period:** we exploit the knowledge that the PDE solution $s(\mathbf{x}, t|\zeta)$ has a period $T_\zeta = \frac{0.5}{\zeta}$ by augmenting the vector of parameters $\mu = \zeta$ that is given as input to f_θ with the vector $\left[\sin\left(\frac{2\pi}{0.5/\zeta}t\right), \cos\left(\frac{2\pi}{0.5/\zeta}t\right)\right]$, informing f_θ with the explicit periodicity of the solution.

Table C.6

Training and hyperparameter details for the PDEs with fixed parameters. Training, validation and test rows signal which time series are taken from the datasets, i.e., 1–8000 means between the 1st and the 8000th series were taken. Initial learning rate (LR) and architecture details are included, together with regularization details.

Parameter	1D Advection	1D Burgers	2D Shallow-Water
Training	1–8000	1–8000	1–800
Validation	8001–9000	8001–9000	801–900
Test	9001–10000	9001–10000	901–1000
Initial LR	0.001	0.0014	0.001
γ_{lr}	0.997	0.999	0.999
Batch Size	16	32	16
F_e	[8,16,32,64,64,64,64]	[8,16,32,32,32,32,32]	[8,32,32,32,32,32,32]
F_d	[64,64,64,64,32,16,1]	[32,32,32,32,32,16,1,1]	[32,32,32,32,32,16,1,1]
K_e	[5,5,5,5,5,5]	[5,5,3,3,3,3,3]	[5,5,3,3,3,3,3]
K_d	[6,6,6,6,6,5]	[4,4,4,4,4,4,3]	[4,4,4,4,4,4,3,3]
f_θ layer no.	2	4	2
f_θ neuron no.	50	200	50
λ	30	30	20
λ_{rg}	0.0	0.001	0.001

Table C.7

Training and hyperparameter details for the PDEs with varying parameters. Train and Test parameter (par.) rows signal which PDE parameter values were selected. Train, Validation (Val.) and Test data rows show which time series are taken from the datasets, i.e., 1–8000 means between the 1st and the 8000th series were taken, while for Molenkamp the number of uniformly sampled parameter values are given. Initial learning rate (LR) and architecture details are included, together with regularization details. The Molenkamp problem had GELU activation functions in the last layer of the Encoder and the first layer of the Decoder.

Parameter	1D Advection	1D Burgers	2D Molenkamp
PDE par.	ζ	v	$\lambda_1, \dots, \lambda_5$
Train par.	0.2, 0.4, 0.7, 2.0, 4.0	0.002, 0.004, 0.02, 0.04, 0.2, 0.4, 2.0	Uniform sampling
Test par.	0.1, 1.0, 7.0	0.001, 0.01, 0.1, 1.0, 4.0	Uniform sampling
Train data	1–8000/ ζ	1–8000	5000
Val. data	8000–9000/ ζ	8000–9000	200
Test data	9000–10000/ ζ	9000–10000	100
LR	0.0018	0.0018	0.0015
γ_{lr}	0.995	0.995	0.995
Batch Size	64	124	16
F_e	[8,16,32,32,32,32,32]	[8,32,32,32,32,32,32]	[8,16,32,32,32,32,32]
F_d	[32,32,32,32,32,16,1]	[32,32,32,32,32,16,1,1]	[32,32,32,32,32,16,1,1]
K_e	[5,5,3,3,3,3,3]	[5,5,3,3,3,3,3]	[5,5,3,3,3,3,3]
K_d	[4,4,4,4,4,3]	[4,4,4,4,4,4,3]	[4,4,4,4,4,4,3,3]
f_θ Layers	4×200	4×200	2×100
λ	30	30	5
λ_{rg}	0.0	0.0	0.0
γ_0	1/500	1/1000	1/500

4. *Data-points closer to test points*: we add points $\zeta \in \{0.05, 1.05, 7.05\}$ to the training data set. Each added ζ has the same 8000 initial conditions for training and 1000 conditions for validation as the baseline, as explained in [Appendix C](#).
5. *Uniform parameter sampling*: we use $\zeta \in [0.5, 7.05]$ values sampled uniformly with steps of 0.05 as training values. We exclude from the training set $\zeta \in \{0.1, 1.0, 7.0\}$. The training initial conditions (per parameter ζ) are the first initial conditions from the 1st to the 250th used by the baseline and the initial conditions for validation are the ones from the 250th to the 300th used by the baseline ([Appendix C](#)). So we use **less** initial conditions per ζ compared to the baseline case.

[Fig. 9](#) compares the performance the "Data-points closer to test points" strategy to that of the baseline in [Fig. 4](#)) for the Burgers' case. We add points $v \in \{0.11, 1.1, 4.1\}$ to the training data set. Each added v has the same 8000 initial conditions for training and 1000 conditions for validation as the baseline, as explained in [Appendix C](#).

C.2. Training instabilities

\mathcal{L}_2^{T,k_1} and especially \mathcal{L}_2^{A,k_2} can involve complex gradients. During the training, this can sometimes lead the NN to be stuck in the trivial minimum for $\mathcal{L}_{2,i}^T$ and $\mathcal{L}_{2,i}^A$ which consists in φ_θ and π_θ returning a constant output. Based on our experiments some datasets are particularly sensitive to this problem, while others are not affected by it, and the following measures help avoiding the trivial solution:

Table C.8

Comparison of the inference time. The dataset used is the Burger's one with $\nu = 0.001$. We do not consider the time spent for sending the batches from the CPU to the GPU; the time measured is the time taken to do inference on the whole testing dataset of 1000 initial conditions with a batchsize of 64. Inference time shows the mean and standard deviation, which we computed for our model by doing inference 100 times. Values of models used for comparisons are taken from Hagnberger et al. [46].

Time resolution	Model	Inference time [ms]
41	Ours	466.73 \pm 68.38
	FNO	917.77 \pm 2.51
	VCNeF	2244.04 \pm 6.65
	Galerkin	2415.99 \pm 54.56
	VCNeF s.	4853.17 \pm 75.29
	OFormer	6025.75 \pm 12.75
81	Ours	932.43 \pm 136.588
	FNO	1912.19 \pm 56.03
	VCNeF	4422.65 \pm 4.11
	Galerkin	4940.80 \pm 89.44
	VCNeF s.	9701.80 \pm 84.48
	OFormer	12081.98 \pm 19.39
121	Ours	1440.67 \pm 250.29
	FNO	2808.04 \pm 82.22
	VCNeF	6606.41 \pm 3.0
	Galerkin	7908.18 \pm 96.52
	VCNeF s.	14577.00 \pm 112.83
	OFormer	17965.47 \pm 14.19
161	Ours	1846.729 \pm 270.72
	FNO	3733.10 \pm 62.94
	VCNeF	6084.04 \pm 9.37
	Galerkin	10295.78 \pm 116.50
	VCNeF s.	19449.80 \pm 113.73
	OFormer	24108.24 \pm 6.45
201	Ours	2389.07 \pm 386.02
	FNO	4614.21 \pm 97.52
	VCNeF	7584.48 \pm 1.86
	Galerkin	13151.47 \pm 93.95
	VCNeF s.	24252.38 \pm 101.41
	OFormer	29986.81 \pm 6.35
240	Ours	2773.019 \pm 406.31
	FNO	5572.07 \pm 109.23
	VCNeF	8935.28 \pm 7.08
	Galerkin	15600.60 \pm 262.51
	VCNeF s.	29063.89 \pm 79.58
	OFormer	35900.51 \pm 6.71

- Removing the biases from the Encoder.
- Using (Batch/Layer) Normalization layers in the Encoder (not necessarily after each convolution).
- Careful tuning of the learning rate (lowering the learning rate or increasing the batch size).
- Warm up of the learning rate.
- Turning off the more complex $\mathcal{L}_{2,i}^{T,k_1}$ and $\mathcal{L}_{2,i}^{A,k_2}$ terms for the initial epochs (e.g., during the warm up of the learning rate) by setting β and γ to zero, if the instabilities come mostly from these terms. This allows for an initial construction of \mathcal{E} with simpler constraints.

C.3. Hardware details

For training we use either an NVIDIA A40 40 GB or an NVIDIA A100 80GB PCIe depending on availabilities.

C.4. Model size and inference speed

In Table C.9 we show the number of NNs weights associated with our model and the models used for comparison from Hagnberger et al. [46]. In Table C.8 we report the inference time for the Burgers' dataset with $\nu = 0.001$. We do not consider the time spent for sending the batches from the CPU to the GPU; the time measured is the time taken to do inference on the whole testing dataset of

Table C.9

Model size of the different architectures. In the first line of the 1D Advection case we show the number of weights of our model for $\zeta = 0.01/\zeta$ varying.

Model	Model size (# NNs weights)		
	Advection	Burgers'	Molenkamp
Ours	188,549 / 214,461	216,657	166,825
Galerkin T.	530,305	530,305	–
FNO	549,569	549,569	–
U-Net	557,137	557,137	–
MP-PDE	614,929	614,929	–
OFormer	660,814	660,814	–
VCNeF	793,825	793,825	1,594,005

1000 initial conditions with a batchsize of 64. We use an NVIDIA A100 80GB PCIe to conduct the inference test. Inference time of other methods is from Hagnberger et al. [46] where they use an NVIDIA A100-SXM4 80GB GPU.

C.5. Training budget

This section summarizes the available information about the training budgets for both our method and the ones used for comparison. For the Molenkamp test we trained the method of comparison ourselves. For all other experiments, the information we have about the trainings of comparison methods comes from Hagnberger et al. [46] as we directly compare to the results within.

C.5.0.1. Molenkamp test. Our method was trained for 1325 epochs in a total time of 13.5 h, and training stopped because the validation loss had not decreased for 200 epochs. The comparison VCNeF method of Hagnberger et al. [46] was trained for 500 epochs for a total time of 24.4 h, using the 'One Cycle Scheduler' [64], with maximum learning rate of 0.2, initial and final division factors of 0.003 and 0.0001, respectively, with each epoch taking approximately 176 s. Both trainings used the same GPU (NVIDIA A100 80GB PCIe).

Not parametric-1D Burgers. Our model has been trained for 4336 epochs, each of which taking approximately 6.1 s on a **single** NVIDIA A100 80GB PCIe, for a total of 7.3 h. The comparison VCNeF training took 18 s per epoch and was trained for 500 epochs in **parallel** on 4 NVIDIA A100-SXM4 80GB GPUs, for a total of 2.5 h. Appendix E.3 of Hagnberger et al. [46] gives no additional information about training times; only stating that all methods have been trained for 500 epochs, except for MP-PDE using 20 epochs.

Not parametric-1D advection. Our model has been trained for 1724 epochs, each taking approximately 10 s on a **single** NVIDIA A100 80GB PCIe, for a total of 4.8 h. Comparison methods have been trained for 500 epochs, except for MP-PDE using 20 epochs, without any additional information available.

Parametric 1D Burgers. Our model has been trained for 1330 epochs on a **single** NVIDIA A100 80GB PCIe, for a total of 25.1 h. Comparison methods have been trained for 500 epochs, except for MP-PDE using 20 epochs, without any additional information available.

Parametric 1D advection. Our model has been trained for 1607 epochs on a **single** NVIDIA A100 80GB PCIe, for a total of 48 h. Comparison methods have been trained for 500 epochs, except for MP-PDE using 20 epochs, without any additional information available.

Shallow water equations. Our model has been trained for 1366 epochs on a **single** NVIDIA A100 80GB PCIe, for a total of 3.3 h.

Appendix D. Methods used for comparison

In Section 3 we compare our model to the following methods:

Fourier neural operator (FNO). Li et al. [21]: it is a particular case of a *Neural Operator*, i.e., a class of models which approximate operators and that can thus perform mapping from infinite-dimensional spaces to infinite-dimensional spaces. The name comes from the assumption that the Kernel of the operator layer is a convolution of two functions, which makes it possible to exploit the Fast Fourier Transform under particular circumstances.

cFNO. Takamoto et al. [47]: it is an adaptation of the FNO methodology which allows to add the PDE parameters as input.

Message passing neural PDE solver (MP-PDE). Brandstetter et al. [29]: it leverages Graph Neural Networks (GNNs) for building surrogate models of PDEs. All the components are based on neural message passing which representationally contain classical methods such as finite volumes, Weighted Essentially Non-Oscillatory (WENO) schemes and finite differences.

U-Net. Ronneberger et al. [65]: it is a method based on an Encoder-Decoder architecture with skip connections between the down-sampling and upsampling procedures. Originally it emerged for image segmentation tasks and but has since been applied to the field of PDE solving as well [27].

Coordinate-based model for operator learning (CORAL). Serrano et al. [66]: it is a method which leverages Neural Fields [67] for the solution of PDEs on general geometries and general time discretizations.

Galerkin transformer (Galerkin). Cao [68]: it is a Neural Operator based on the self-attention mechanism from Transformers with a novel layer normalization scheme mimicking the Petrov–Galerkin projection.

Operator transformer (OFormer). Li et al. [69]: it is a Neural Operator which leverages the fact that the self-attention layer of Transformers is a special case of an Operator Layer (as shown in Kovachki et al. [18]) to build a PDE solver.

cOFormer. It is an adaptation of the OFormer architecture which allows for the query of PDE parameters as inputs, following what is done in Takamoto et al. [48].

Vectorized conditional neural fields (VCNeF). Hagnberger et al. [46]: it is a transformer based model which leverages neural fields to represent the solution of a PDE at any spatial point continuously in time. For the Molenkamp test we implemented the VCNeF method from the Git-Hub repository of Hagnberger et al. [46], using the same 'One Cycle Scheduler' [64] with maximum learning rate at 0.2, initial division factor 0.003 and final division factor 0.0001 for training. We employ 1000 epochs, a batch size of 40, an embedding size of 96, 1 transformer layer with 8 heads and 6 modulation blocks.

Physics-informed neural networks (PINN). Raissi et al. [70]: it is a class of methods which uses the physical knowledge of the system (in this case the PDE) to improve the approximate solution of the PDE by the NN (via e.g., penalizing the PDE residual too in the loss).

The above methods have been implemented in Hagnberger et al. [46] (for the 1D Advection and 1D Burgers results) and [48] (for the 2D Shallow Water results) and we used the reported values to compare our results to in Section 3.

Appendix E. Ablation studies

E.1. The role of the ODE solver and of \mathcal{L}_3 in time generalization

In Fig. E.11a we show the effect of the stage q of the RK algorithm used to solve Eq. (6) for the Burgers' dataset with $\nu = 0.001$. We see that by increasing q not only the nRMSE(t) (from Eq. (F.11)) is lowered, but also the gap between the trajectory of $\Delta t = 0.05$ (used during training) and $\Delta t = 0.01$ is decreased, i.e., the larger the q the better the generalization in time during inference. This is particularly clear when looking at the nRMSE(t) of $q = 3$ and $q = 4$, since for $\Delta t = 0.05$ they are almost the same, while for $\Delta t = 0.01$ it is noticeably lower when $q = 4$. In Fig. E.11b we do the same experiment with the Advection dataset: here only for $q = 1$ there is a big gap between the prediction at $\Delta t = 0.05$ and $\Delta t = 0.01$.

In Fig. E.11c once again we show the same pattern for the Molenkamp dataset: increasing the value of q results in a better capability of the model to generalize in time during inference by taking a smaller Δt .

In Fig. E.11d we show a comparison of the nRMSE(t) on the Burgers' dataset with $\nu = 0.001$ between using the full loss \mathcal{L}_{lr} and switching off \mathcal{L}_3 by setting $\delta = 0$: while for $\Delta t = 0.05$ (the one used at training) the two curves are comparable, for $\Delta t = 0.01$ a huge gap is present. This result is in line with the reasoning that \mathcal{L}_3 helps the model to generalize in time, as explained Section 2.6.

E.2. Impact of γ_0 and $p(k_2)$ in the autoregressive strategy

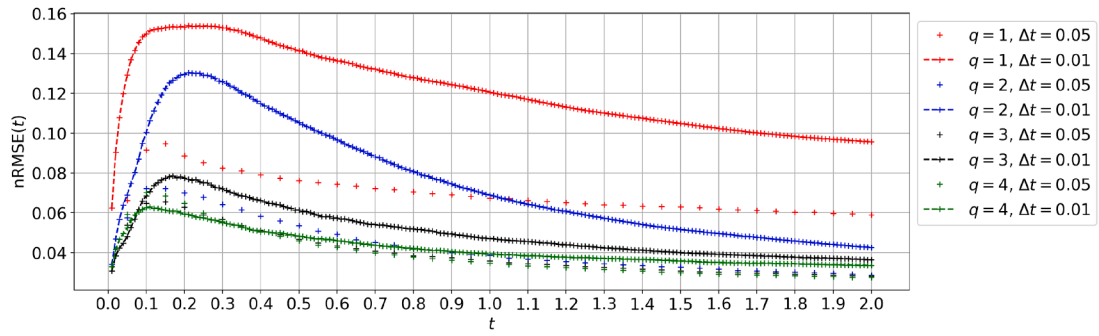
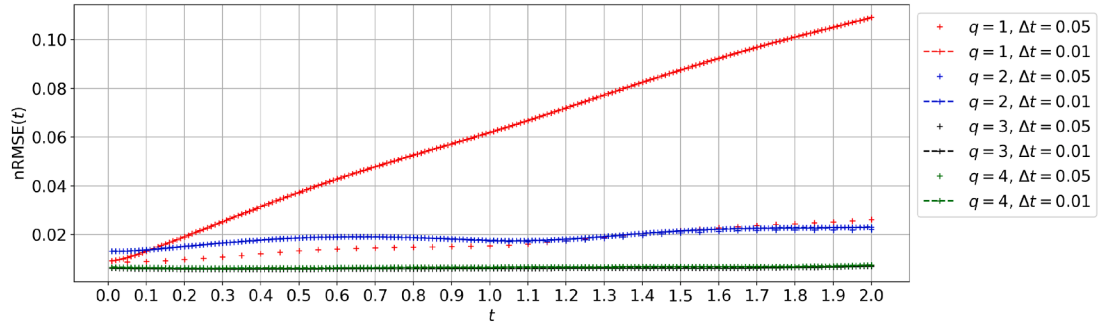
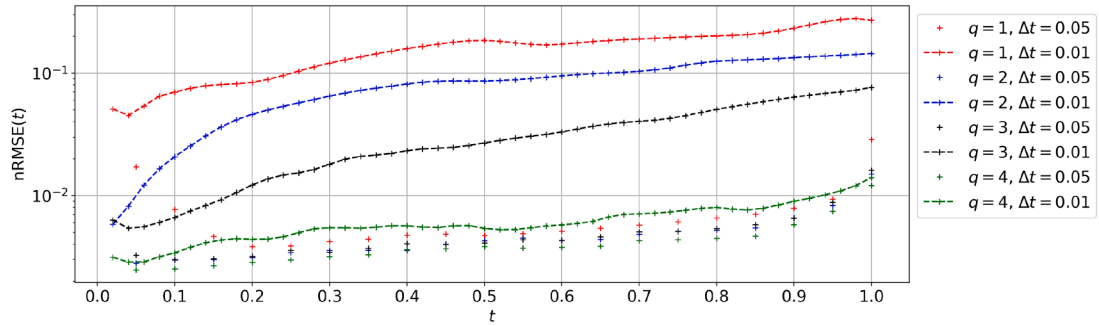
The autoregressive term $\gamma \mathcal{L}_{2,i}^{A,k_2}$ during the training is multiplied by a scalar γ , where initially $\gamma = \gamma_0 < 1$ and is increased every epoch by an amount of γ_0 until $\gamma = 1$. In Fig. E.12 we vary the value of γ_0 from 0 (Teacher Forcing) to 0.002. We also experimented with higher values ($\gamma_0 = 0.01, 0.1, 1$) but resulted in NaN errors in the training. We see a trend where the higher the value of γ_0 the lower the error; however it comes with a more unstable training process which may require careful hyperparameters tuning.

Furthermore, in Fig. E.13 we experiment with varying the value of $p(k_2)$ on the parametric Burgers dataset, where $p(k_2)$ is the amount of epochs needed for k_2 to be increased by 1: no significant difference is present in the three experiments.

E.3. Coupling of AE and NODE

In Section 2 we stated that the training of the AutoEncoder was coupled with the training of the Neural ODE. This choice was rooted in the assumption that training the encoder φ_θ and the decoder ψ_θ together with f_θ pushes the NNs's weights towards a minimum such that the latent space \mathcal{E} allows for an easier modeling of the latent dynamics through f_θ , since \mathcal{E} and f_θ are built at the same time. In this section we give some empirical results backing up such decision.

Firstly, how different is the latent space found when the training is coupled from when it is not coupled? In Fig. E.14 we show the Time evolution for a given initial condition and for a given parameter instance from the test set of the corresponding latent vector ϵ ,

(a) Burgers' dataset with $\nu = 0.001$.(b) Advection dataset with $\zeta = 0.1$.

(c) Molenkamp dataset.

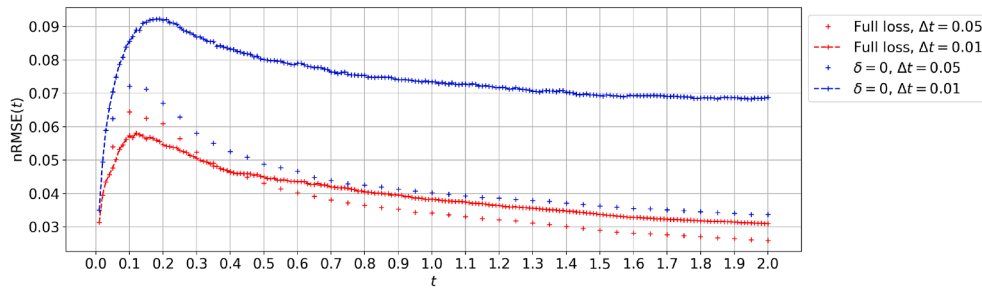
(d) Burgers' dataset ($\nu = 0.001$), with and without ($\delta = 0$) \mathcal{L}_3 . Training time-step is $\Delta t = 0.05$.

Fig. E.11. nRMSE(t) when varying the stage q of the RK algorithm to solve the ODE of Eq. (6) for the Burgers' (a), Advection (b) and Molenkamp datasets (c) and when using \mathcal{L}_3 in the training (d). In (a)-(c) the same q is used at training and inference. Increasing q improves the predictions when using the same Δt as during training ($\Delta t = 0.01$) and also yields better generalization in time. In (d) the presence of \mathcal{L}_3 at training (red curves) improves the generalization in time. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

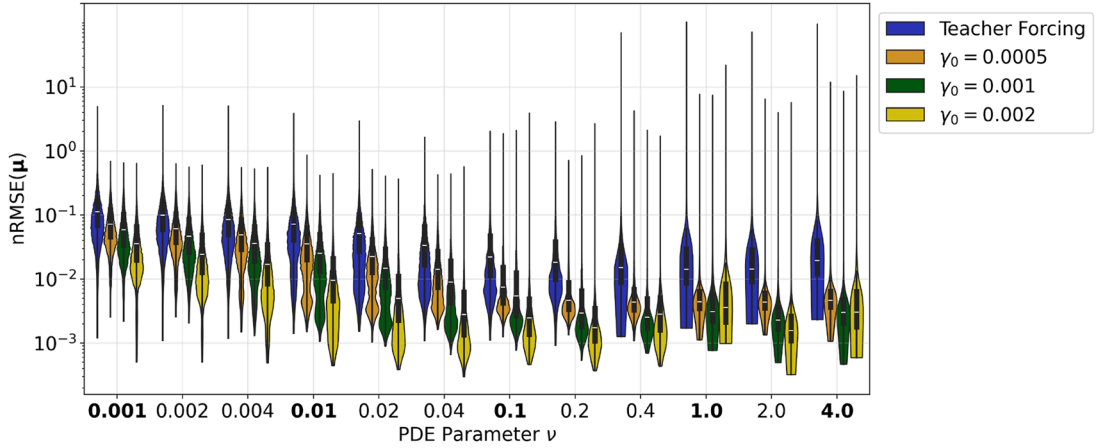


Fig. E.12. We show the impact of the scheduling chosen for the autoregressive term with the variation of γ_0 . Teacher Forcing is equivalent to $\gamma_0 = 0$.

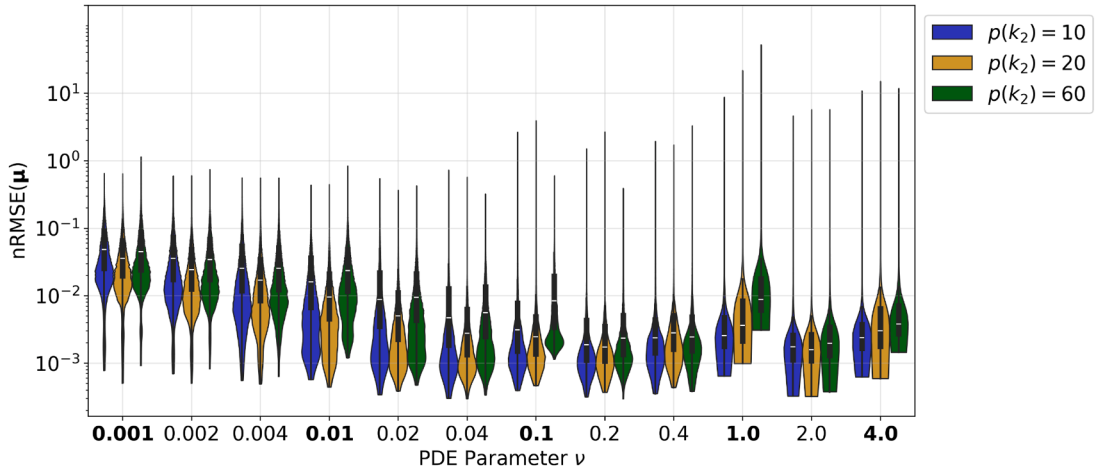


Fig. E.13. We show the impact of varying the term $p(k_2)$. No noticeable difference is present at its variation.

both for the parametric Advection and Burgers case. In both cases the most evident difference between coupling and not is the scale of the variation of each latent dimension: it is larger for the not coupled models and very small for the coupled one, as shown in Fig. E.15. Finally, in Fig. E.17 we compare the $\text{nRMSE}(\mu)$ distribution of both the parametric Advection and parametric Burgers case when the system is trained coupled and when not: the error is smaller when the training is performed coupled.

We thus observed 3 phenomena:

- the error of the AutoEncoder is smaller when the training is not coupled (Fig. E.16);
- the shape of the latent space is not dramatically different in the two cases (Figs. E.14 and E.15). The main distinction is observed in the scale of the variation of each dimension of the latent vectors over time, which is considerably smaller in the coupled case;
- the approximation error of the PDE solution is smaller when the training is coupled (Fig. E.17).

It is difficult by looking at Fig. E.14 to justify why it is true that the latent space found in the coupled case is more easily approximated by the NODE, we only report what we observed experimentally. We conclude by reporting that training the two processes decoupled results in a much more stable training process, which did not require the use of the tricks documented in Appendix C.2 as in the coupled case.

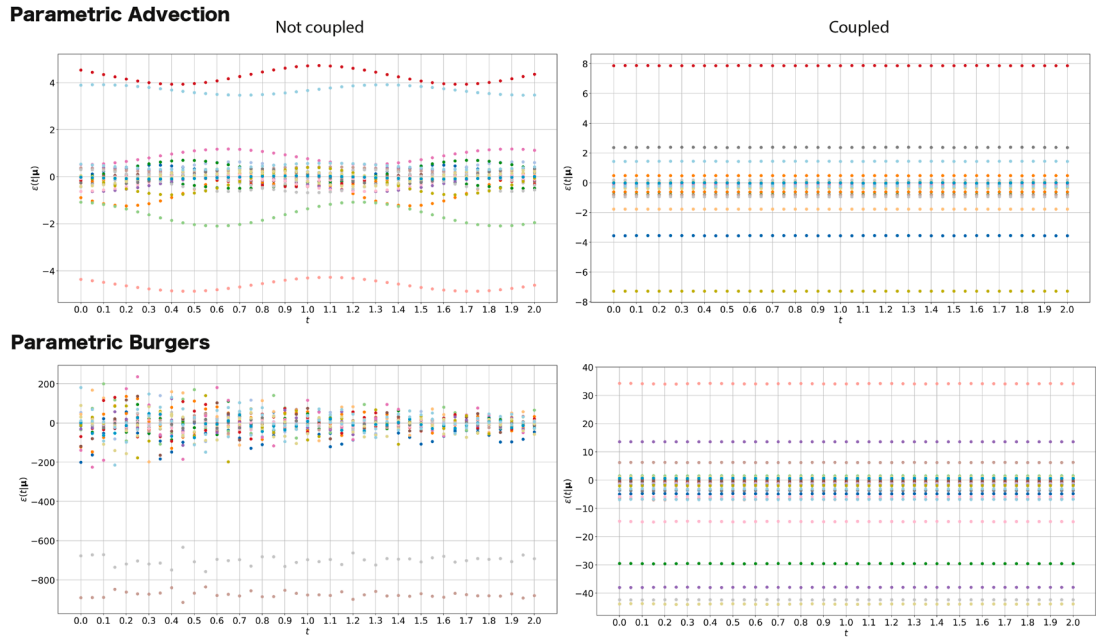


Fig. E.14. Time evolution of the latent vector $\varepsilon(t|\mu)$ for a given initial condition and parameter instance (ζ for Advection and v for Burgers). Each color corresponds to a different dimension of the latent vector, 30 in total. We compare the time evolution for both the parametric Advection (top) and the parametric Burgers (bottom) cases, between the two training strategies: the Autoencoder training coupled to the latent space dynamics optimization (right) and not (left).

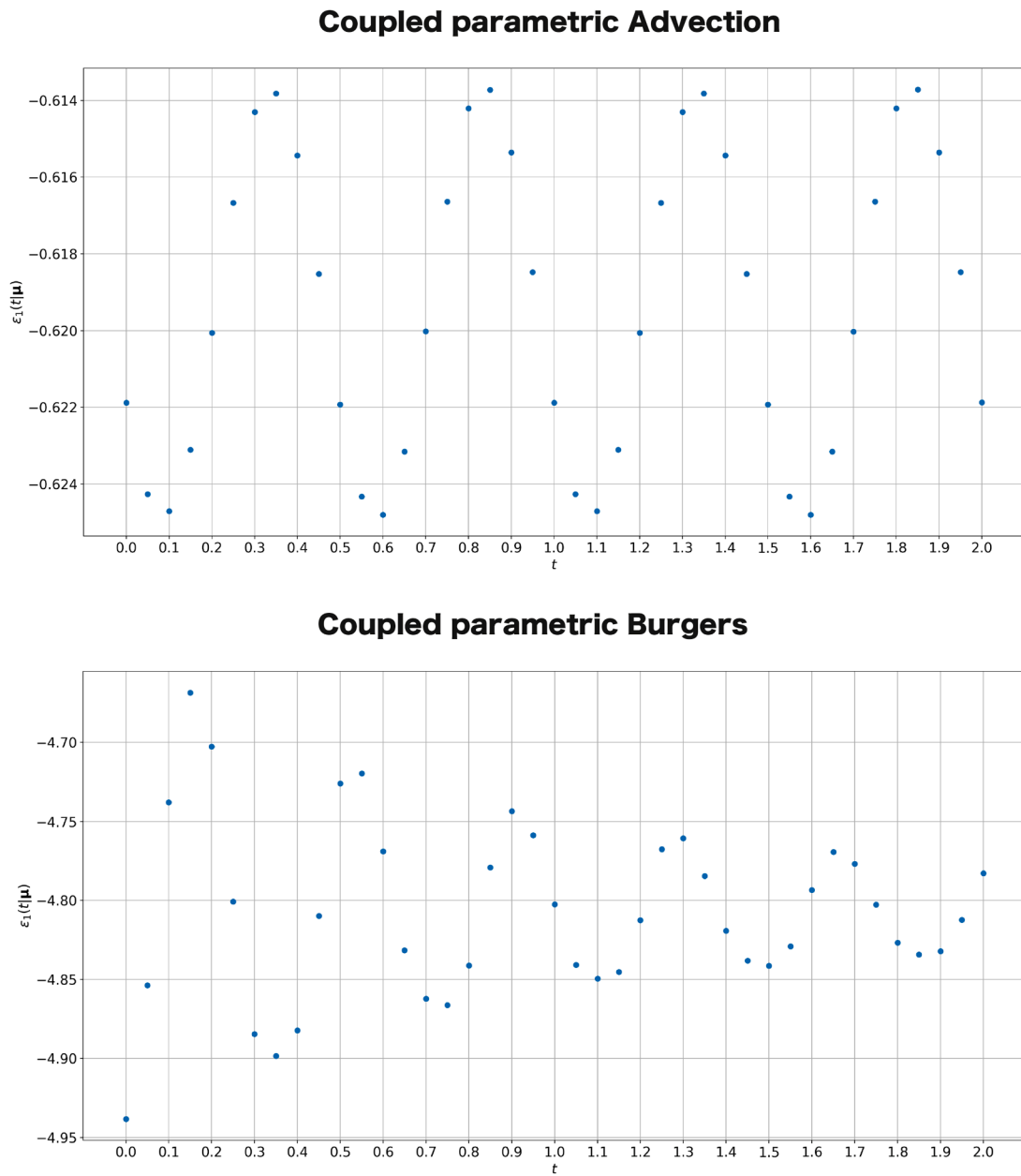


Fig. E.15. Time evolution of a chosen latent variable from the latent vector $\varepsilon(t|\mu)$ for a given initial condition and parameter instance (ζ for Advection and ν for Burgers). Although at smaller scales than in the coupled case, the latent variables show similar variations in time.

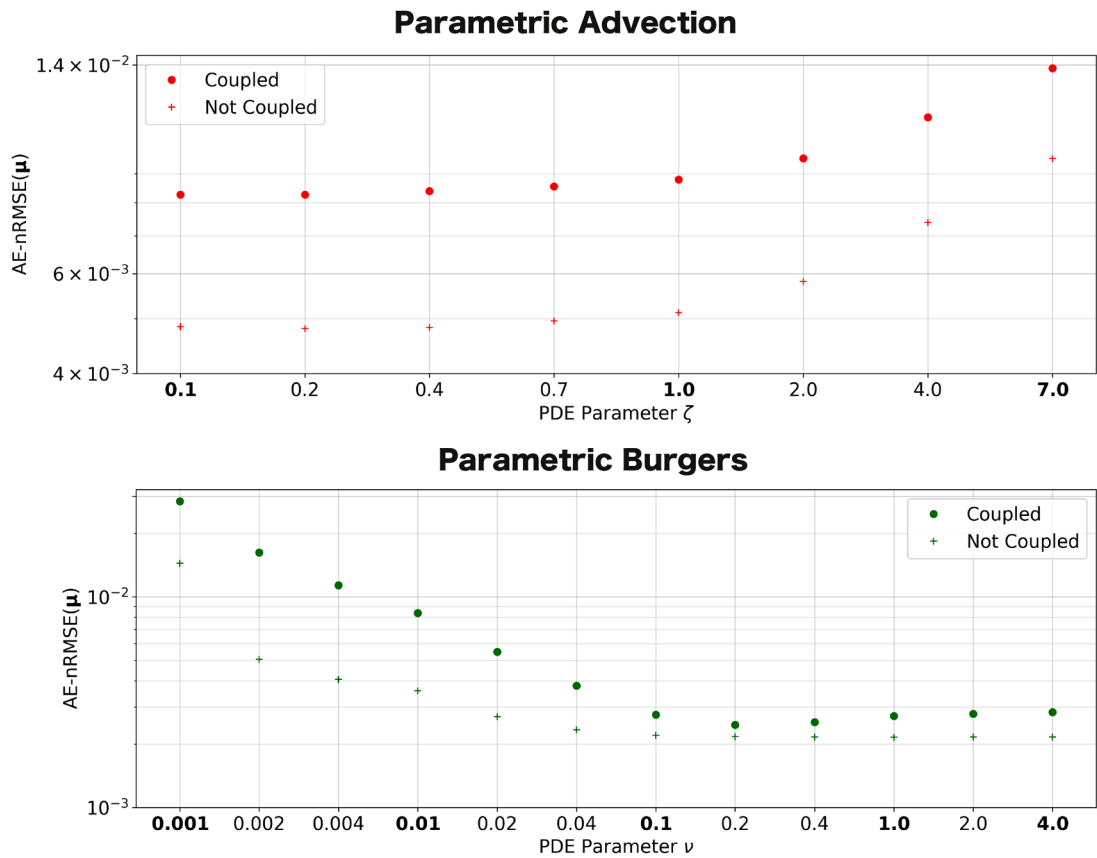
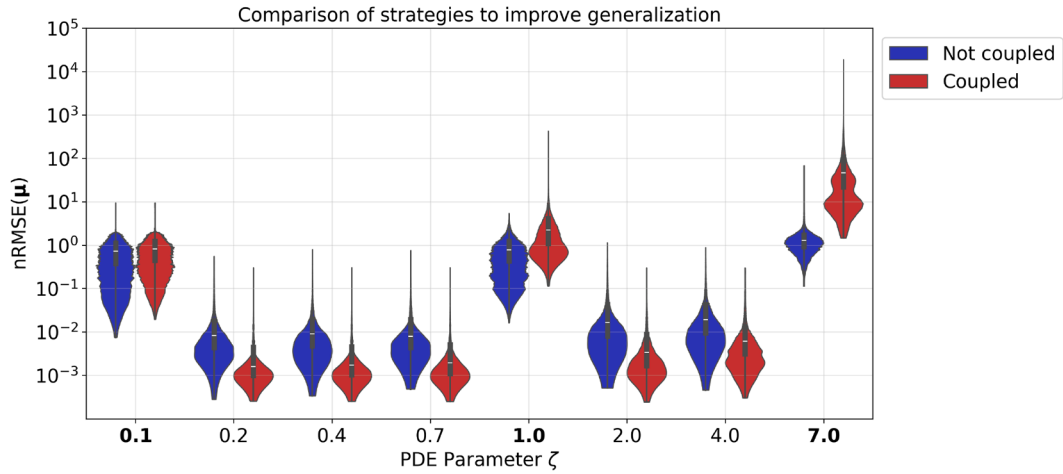


Fig. E.16. Comparison of the AE-nRMSE(μ) when encoding and decoding the test set, both for the parametric Advection (top) and the parametric Burgers datasets (bottom). In both cases, the AE-nRMSE(μ) is lower when the AutoEncoder is trained independently from learning the latent space dynamics.

Parametric Advection



Parametric Burgers

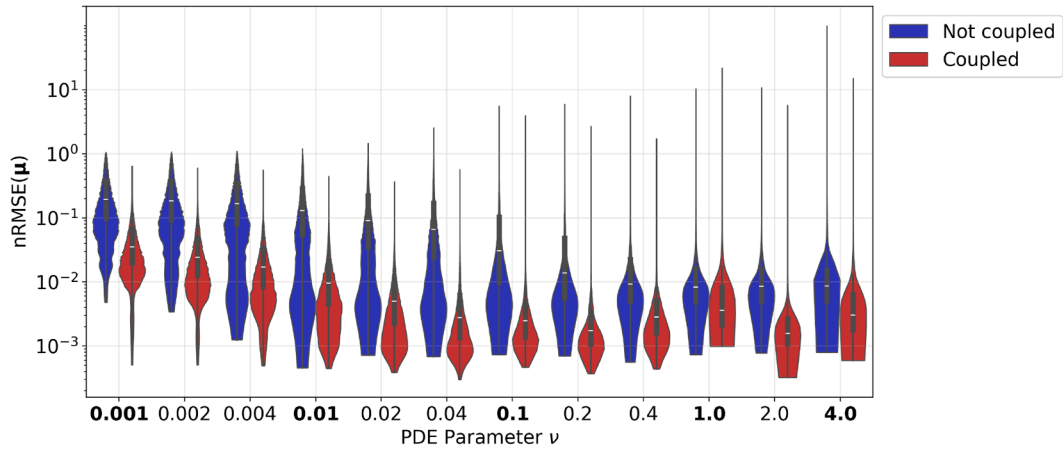


Fig. E.17. Comparison of the $\text{nRMSE}(\mu)$ with (red) and without (blue) coupling the training of the AutoEncoder and the Neural ODE, both for the parametric Advection (top) and the Burgers advection datasets (bottom). Bold characters signal testing parameter values not included in the training set. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Appendix F. Datasets

Unless stated otherwise, the solutions of the PDEs described in this section used in our model training come from Takamoto et al. [48].

F.1. 1D advection equation

The 1D Advection Equation is a linear PDE which transports the initial condition with a constant velocity ζ :

$$\begin{cases} \partial_t s(\mathbf{x}, t|\mu) + \zeta \partial_x s(\mathbf{x}, t|\mu) = 0, & x \in (0, 1), t \in (0, 2] \\ s(\mathbf{x}, 0|\mu) = s^0(\mathbf{x}, \mu), & x \in (0, 1). \end{cases} \quad (\text{F.1})$$

Periodic boundary conditions are considered (i.e., $s(0, t|\boldsymbol{\mu}) = s(1, t|\boldsymbol{\mu})$) and as initial condition a super-position of sinusoidal waves is used:

$$s^0(\mathbf{x}, \boldsymbol{\mu}) = \sum_{i=1, \dots, N} A_i \sin(k_i x + \phi_i), \quad (\text{F.2})$$

where $k_i = 2\pi n_i / L_x$ with n_i being random integers, L_x is the domain size, A_i are random numbers from the interval $[0, 1]$ and ϕ_i are the phases chosen randomly in $(0, 2\pi)$. We use 256 equidistant spatial points x in the interval $[0, 1]$ and for training 41 uniform timesteps in the interval $[0, 2]$.

F.2. 1D Burgers' equation

The Burgers's equation is a non-linear PDE used in various modeling tasks such as fluid dynamics and traffic flows:

$$\begin{cases} \partial_t s(\mathbf{x}, t|\boldsymbol{\mu}) + \partial_x (s^2(\mathbf{x}, t|\boldsymbol{\mu})/2) - \nu/\pi \partial_{xx} s(\mathbf{x}, t|\boldsymbol{\mu}) & x \in (0, 1), t \in (0, 2] \\ s(\mathbf{x}, 0|\boldsymbol{\mu}) = s^0(\mathbf{x}, \boldsymbol{\mu}), & x \in (0, 1), \end{cases} \quad (\text{F.3})$$

where ν is the diffusion coefficient. The initial conditions and the boundary conditions are the same as in Section F.1. We use 256 equidistant spatial points in the interval $[0, 1]$ and for training 41 uniform timesteps in the interval $[0, 2]$.

F.3. 2D shallow water equations

The 2D Shallow Water Equations are a system of hyperbolic PDEs derived from the Navier Stokes equations and describe the flow of fluids, primarily water, in situations where the horizontal dimensions (length and width) are much larger than the vertical dimension (depth):

$$\begin{aligned} \partial_t h + \partial_x hu + \partial_y hv &= 0, \\ \partial_t hu + \partial_x \left(u^2 h + \frac{1}{2} g_r h^2 \right) + \partial_y uvh &= -g_r h \partial_x b, \\ \partial_t hv + \partial_x \left(v^2 h + \frac{1}{2} g_r h^2 \right) + \partial_y uvh &= -g_r h \partial_y b, \end{aligned} \quad (\text{F.4})$$

where u, v are the horizontal and vertical velocities, h is the water depth and b is a spatially varying bathymetry. g_r is the gravitational acceleration. We use 128×128 equidistant spatial points (x, y) in the interval $[-1, 1] \times [-1, 1]$ and for training 21 uniform timesteps in the interval $[0, 1]$, while the compared methods use 101 uniform timesteps in the interval $[0, 1]$.

F.4. 2D molenkamp test

The Molenkamp test is a two dimensional advection problem, whose exact solution is given by a Gaussian function which is transported trough a circular path without modifying its shape. Here we add a reaction term which makes the Gaussian shape decay over time:

$$\begin{aligned} \partial_t q(x, y, t) + u \partial_x q(x, y, t) + v \partial_y q(x, y, t) + \lambda_3 q(x, y, t) &= 0 \\ q(x, y, 0) = \lambda_1 0.01 \lambda_2 h(x, y, 0)^2, \quad h(x, y, 0) &= \sqrt{(x - \lambda_4 + \frac{1}{2})^2 + (y - \lambda_5)^2}, \end{aligned} \quad (\text{F.5})$$

with $u = -2\pi y$ and $v = 2\pi x$ and $(x, y) \in [-1, 1]$. For this problem an exact solution exists:

$$\begin{aligned} q(x, y, t) &= \lambda_1 0.01 \lambda_2 h(x, y, t)^2 \exp^{-\lambda_3 t}, \\ h(x, y, t) &= \sqrt{(x - \lambda_4 + \frac{1}{2} \cos(2\pi t))^2 + (y - \lambda_5 - \frac{1}{2} \sin(2\pi t))^2}. \end{aligned} \quad (\text{F.6})$$

The PDE depends on 5 parameters $\lambda_1, \dots, \lambda_5$, which control the magnitude of the initial Gaussian, the size of the cloud, the speed of decay, and the initial coordinates x and y . The ranges of the parameters are taken from Alsayyari et al. [71]: $\lambda_1 \in [1, 20]$, $\lambda_2 \in [2, 4]$, $\lambda_3 \in [1, 5]$, $\lambda_4 \in [-0.1, 0.1]$, $\lambda_5 \in [-0.1, 0.1]$. We use 128×128 equidistant spatial points (x, y) in the interval $[-1, 1] \times [-1, 1]$ and for training 21 uniform timesteps in the interval $[0, 1]$.

F.5. Test error metrics

We use as testing metrics the Normalized-Root-Mean-Squared-Error, defined in different ways according to which quantities are averaged over:

$$\text{nRMSE} = \frac{1}{N_u N_{\boldsymbol{\mu}} F} \sum_{i=1}^{N_u} \sum_{p=1}^{N_{\boldsymbol{\mu}}} \sum_{j=1}^F \frac{\|s_r(\mathbf{x}, t_j | \boldsymbol{\mu}_p, s_{r,i}^0) - \tilde{s}_r(\mathbf{x}, t_j | \boldsymbol{\mu}_p, s_{r,i}^0)\|_2}{\|s_r(\mathbf{x}, t_j | \boldsymbol{\mu}_p, s_{r,i}^0)\|_2}, \quad (\text{F.7})$$

$$\text{nRMSE}(\boldsymbol{\mu}) = \frac{1}{N_u F} \sum_{i=1}^{N_u} \sum_{j=1}^F \frac{\|s_r(\mathbf{x}, t_j | \boldsymbol{\mu}, s_{r,i}^0) - \tilde{s}_r(\mathbf{x}, t_j | \boldsymbol{\mu}, s_{r,i}^0)\|_2}{\|s_r(\mathbf{x}, t_j | \boldsymbol{\mu}, s_{r,i}^0)\|_2}, \quad (\text{F.8})$$

$$\text{NODE-nRMSE}(\boldsymbol{\mu}) = \frac{1}{N_u F} \sum_{i=1}^{N_u} \sum_{j=1}^F \frac{\|\varphi_\theta \circ s_r(\mathbf{x}, t_j | \boldsymbol{\mu}, s_{r,i}^0) - \tilde{\varepsilon}(t_j | \boldsymbol{\mu}, s_{r,i}^0)\|_2}{\|\varphi_\theta \circ s_r(\mathbf{x}, t_j | \boldsymbol{\mu}, s_{r,i}^0)\|_2}, \quad (\text{F.9})$$

$$\text{AE-nRMSE}(\boldsymbol{\mu}) = \frac{1}{N_u F} \sum_{i=1}^{N_u} \sum_{j=1}^F \frac{\|s_r(\mathbf{x}, t_j | \boldsymbol{\mu}, s_{r,i}^0) - \psi_\theta \circ \varphi_\theta \circ s_r(\mathbf{x}, t_j | \boldsymbol{\mu}, s_{r,i}^0)\|_2}{\|s_r(\mathbf{x}, t_j | \boldsymbol{\mu}, s_{r,i}^0)\|_2}, \quad (\text{F.10})$$

$$\text{nRMSE}(t) = \frac{1}{N_u N_\boldsymbol{\mu}} \sum_{i=1}^{N_u} \sum_{p=1}^{N_\boldsymbol{\mu}} \frac{\|s_r(\mathbf{x}, t | \boldsymbol{\mu}_p, s_{r,i}^0) - \tilde{s}_r(\mathbf{x}, t | \boldsymbol{\mu}_p, s_{r,i}^0)\|_2}{\|s_r(\mathbf{x}, t | \boldsymbol{\mu}_p, s_{r,i}^0)\|_2}, \quad (\text{F.11})$$

where N_u , $N_\boldsymbol{\mu}$ and F are the number of initial conditions, parameter instances and time steps used at testing, respectively, and $s_{r,i}^0$ stands for the i th initial condition. $\tilde{\varepsilon}(t_j | \boldsymbol{\mu}_p, s_{r,i}^0) = \pi_\theta(\cdot, \boldsymbol{\mu}_p, \Delta t_{j,j-1}) \circ \dots \circ \pi_\theta(\cdot, \boldsymbol{\mu}_p, \Delta t_{1,0}) \circ \varphi_\theta(s_{r,i}^0)$ is the latent vector predicted by the NODE at time t_j during inference. Thus:

- nRMSE is the total error;
- nRMSE($\boldsymbol{\mu}$) is the parametric total error, i.e., total error for any parameter instance $\boldsymbol{\mu}$;
- NODE-nRMSE($\boldsymbol{\mu}$) is the parametric Neural ODE error, i.e., the total error for any parameter instance $\boldsymbol{\mu}$ coming from the NODE prediction;
- AE-nRMSE($\boldsymbol{\mu}$) is the parametric AutoEncoder error, i.e., the total error per for any parameter instance $\boldsymbol{\mu}$ coming purely from the autoencoder;
- nRMSE(t) is the temporal total error, i.e., the total error for any time step t .

We also define the *relative error* $e_r(\mathbf{x}, t)$ as a more spatially meaningful error measure between the predicted field $\tilde{s}_r(\mathbf{x}, t | \boldsymbol{\mu})$ and the ground truth field $s_r(\mathbf{x}, t | \boldsymbol{\mu})$:

$$e_r(\mathbf{x}, t) = \frac{|s_r(\mathbf{x}, t | \boldsymbol{\mu}) - \tilde{s}_r(\mathbf{x}, t | \boldsymbol{\mu})|}{\|s_r(\mathbf{x}, t | \boldsymbol{\mu})\|_2}, \quad (\text{F.12})$$

where the numerator is the point-wise absolute value of the difference between $\tilde{s}_r(\mathbf{x}, t | \boldsymbol{\mu})$ and $s_r(\mathbf{x}, t | \boldsymbol{\mu})$ (hence it has the same dimensionality as $s_r(\mathbf{x}, t | \boldsymbol{\mu})$), while the denominator is a scalar.

Appendix G. Additional images

Fig. G.18 shows our model's performance in the Molenkamp test for 4 different parameter values $\boldsymbol{\mu}$ listed in Table G.10. Fig. G.19 displays the predictions of our model on the Shallow-Water test case for 4 different initial conditions $s_{r,1}^0$, $s_{r,2}^0$, $s_{r,3}^0$ and $s_{r,4}^0$. Fig. G.20 shows our model's performance on the 1D Advection test case for 2 different initial conditions and 4 different velocities: $\zeta = 0.4$, $\zeta = 0.7$, $\zeta = 2.0$ and $\zeta = 4.0$. Finally, Fig. G.21 displays the prediction for the Burgers case with $\nu = 0.001$ for two different initial conditions.

Table G.10
The 4 different parameter vectors used in the Molenkamp test.

	$\boldsymbol{\mu}_1$	$\boldsymbol{\mu}_2$	$\boldsymbol{\mu}_3$	$\boldsymbol{\mu}_4$
λ_1	2.452	19.8578	11.7423	16.8555
λ_2	2.373	2.5791	3.9285	3.4449
λ_3	2.791	1.9388	2.5638	2.6506
λ_4	0.053	0.0959	0.0384	0.0502
λ_5	0.0125	-0.0857	0.0200	0.0423

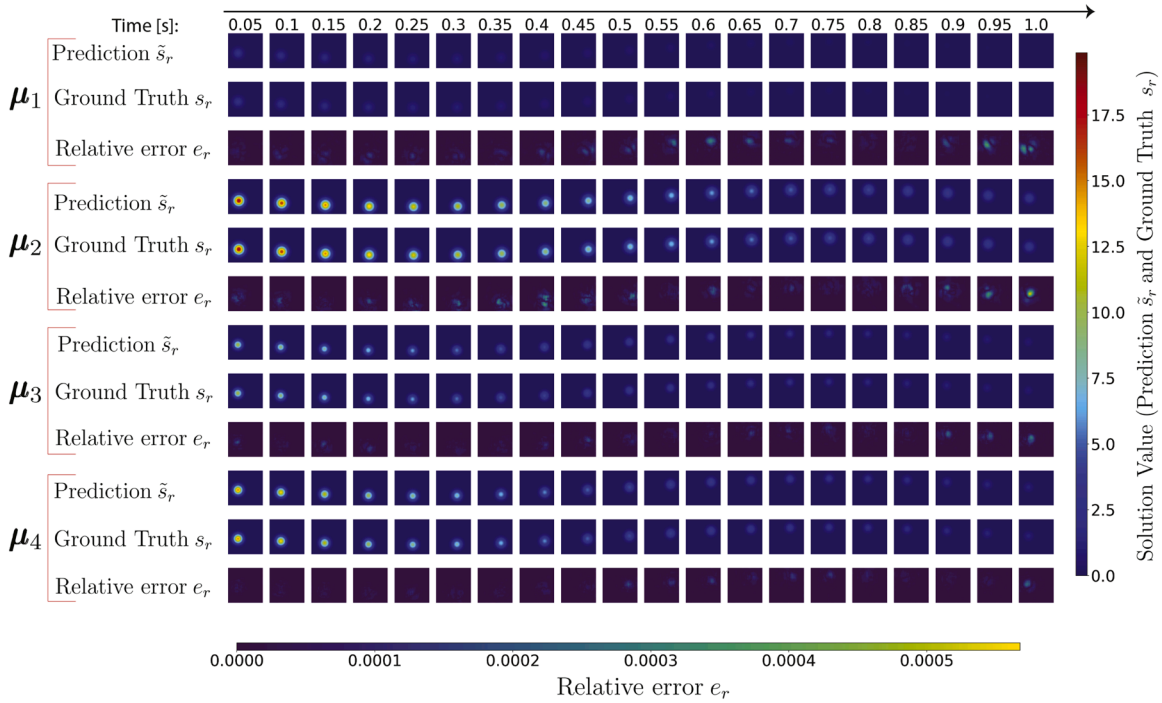


Fig. G.18. Model predictions over time on the Molenkamp test dataset for 4 different parameter combinations μ_1 , μ_2 , μ_3 and μ_4 . The vertical colorbar refers to the prediction $s_r(\mathbf{x}, t|\mu)$ and ground truth $\tilde{s}_r(\mathbf{x}, t|\mu)$ fields (top and middle rows for each parameter vector), while the horizontal one to the relative error e_r of Eq. (F.12) (bottom rows for each parameter vector). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

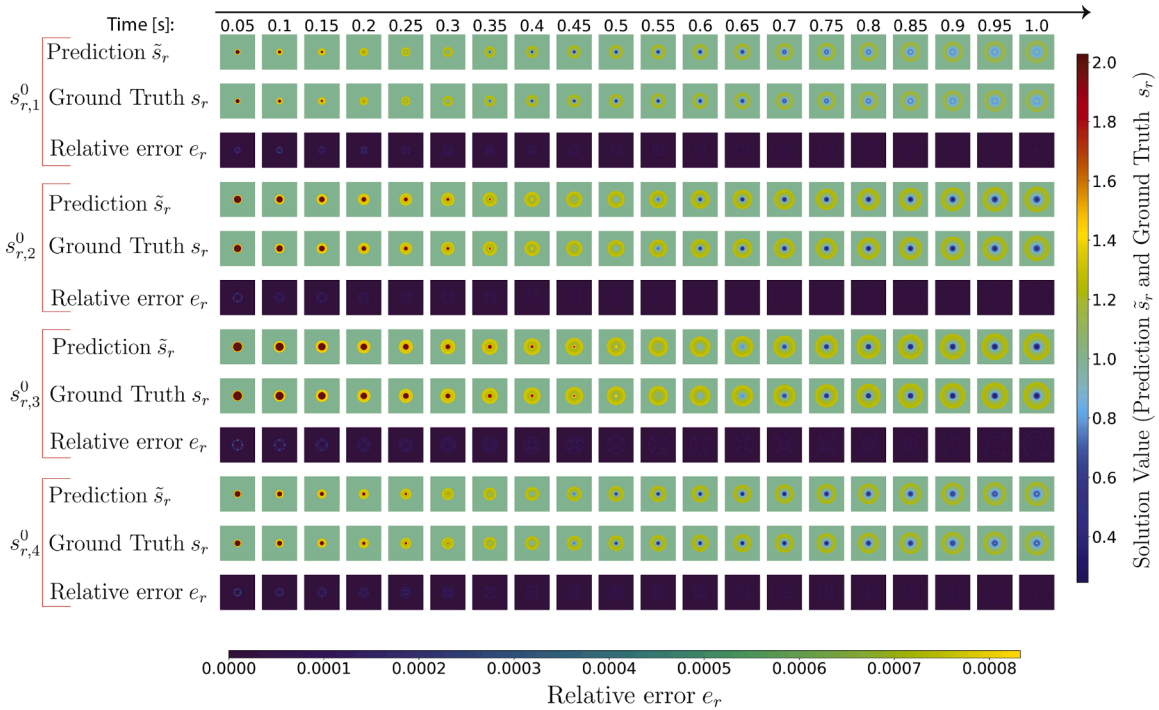


Fig. G.19. Model predictions over time on the Shallow-Water test dataset for 4 different initial conditions $s_{r,1}^0$, $s_{r,2}^0$, $s_{r,3}^0$ and $s_{r,4}^0$. The vertical colorbar refers to the prediction $\tilde{s}_r(\mathbf{x}, t|\mu)$ and ground truth $s_r(\mathbf{x}, t|\mu)$ fields (top and middle rows for each initial condition), while the horizontal one to the relative error e_r of Eq. (F.12) (bottom rows for each initial condition). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

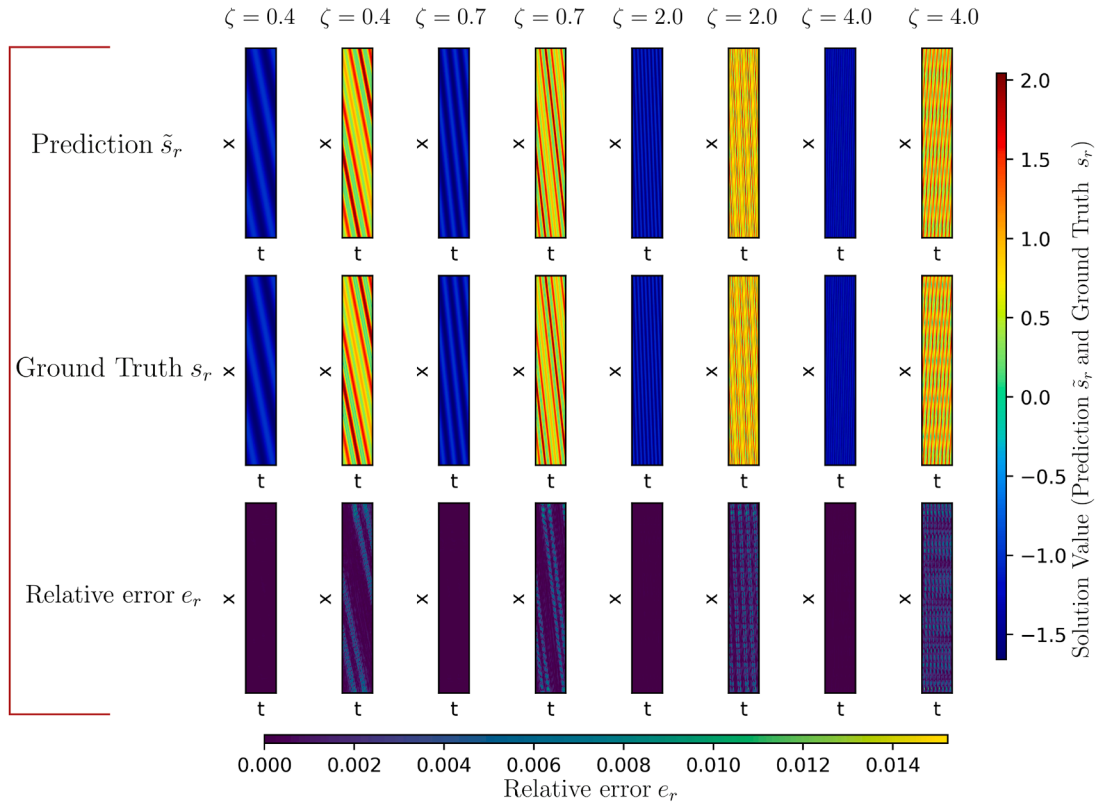


Fig. G.20. Predictions of our model on the parametric Advection dataset for 2 different initial conditions (odd and even columns) and 4 different velocities: $\zeta = 0.4$, $\zeta = 0.7$, $\zeta = 2.0$ and $\zeta = 4.0$. Each plot is a heat map with time t on the horizontal axis and space x on the vertical axis. The vertical colorbar refers to the prediction $\tilde{s}_r(x, t|\mu)$ (top) and the ground truth $s_r(x, t|\mu)$ (middle) fields, while the horizontal one refers to the relative error e_r of Eq. (F.12) (bottom row). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

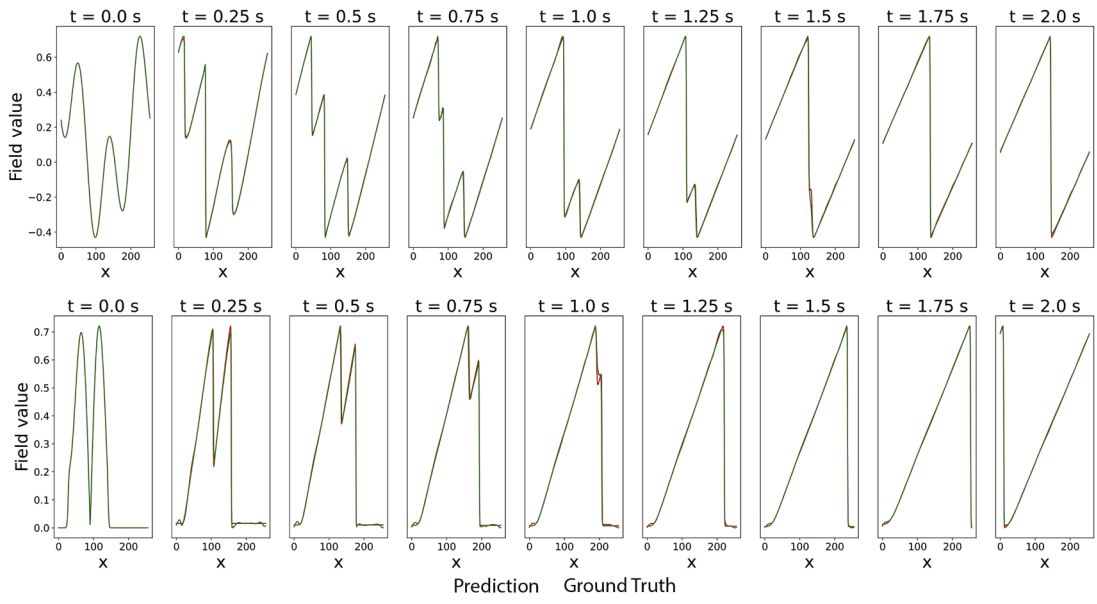


Fig. G.21. Predictions of our model on the Burgers' dataset with $\nu = 0.001$. The two rows correspond to two different initial conditions.

References

- [1] A. Quarteroni, R. Sacco, F. Saleri, *Numerical Mathematics (Texts in Applied Mathematics)*, Springer-Verlag, Berlin, Heidelberg, 2006.
- [2] R. Iman, J. Helton, An investigation of uncertainty and sensitivity analysis techniques for computer-models, *Risk Anal.* 8 (2006) 71–90. <https://doi.org/10.1111/j.1539-6924.1988.tb01155.x>
- [3] Z. Perkó, L. Gilli, D. Lathouwers, J.L. Kloosterman, Grid and basis adaptive polynomial chaos techniques for sensitivity and uncertainty analysis, *J. Comput. Phys.* 260 (2014) 54–84. <https://doi.org/10.1016/j.jcp.2013.12.025>
- [4] A. Quarteroni, G. Rozza, *Reduced Order Methods for Modeling and Computational Reduction*, Springer International Publishing, 2014. <https://doi.org/10.1007/978-3-319-02090-7>
- [5] J.L. Lumley, The structure of inhomogeneous turbulent flows, *Atmos. Turbul. Wave Propag.* (1967) 166–178. <https://cir.nii.ac.jp/crid/1573387449825294592>
- [6] C. Fefferman, S. Mitter, H. Narayanan, Testing the manifold hypothesis, *J. Am. Math. Soc.* 29 (4) (2016) 983–1049. <https://doi.org/10.1090/jams/852>
- [7] S. Goldt, M. Mézard, F. Krzakala, L. Zdeborová, Modeling the influence of data structure on learning in neural networks: the hidden manifold model, *Phys. Rev. X* 10 (4) (2020) 041044.
- [8] T. Cohen, M. Welling, Learning the irreducible representations of commutative lie groups, in: *International Conference on Machine Learning*, PMLR, 2014, pp. 1755–1763.
- [9] I. Higgins, D. Amos, D. Pfau, S. Racaniere, L. Matthey, D. Rezende, A. Lerchner, Towards a definition of disentangled representations, *arXiv preprint arXiv:1812.02230* (2018).
- [10] S. Fresca, A. Manzoni, POD-DL-ROM: Enhancing deep learning-based reduced order models for nonlinear parametrized PDEs by proper orthogonal decomposition, *Comput. Methods Appl. Mech. Eng.* 388 (2022) 114181. <https://www.sciencedirect.com/science/article/pii/S0045782521005120>. <https://doi.org/https://doi.org/10.1016/j.cma.2021.114181>
- [11] K. Bhattacharya, B. Hosseini, N.B. Kovachki, A.M. Stuart, Model reduction and neural networks for parametric PDEs, *SMAI J. Comput. Math.* 7 (2021) 121–157.
- [12] K. Lee, K.T. Carlberg, Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders, *J. Comput. Phys.* 404 (2020) 108973. <https://doi.org/10.1016/j.jcp.2019.108973>
- [13] A. Solera-Rico, C. Sanmiguel Vila, M. Gómez-López, Y. Wang, A. Almashjary, S.T.M. Dawson, R. Vinuesa, Variational autoencoders and transformers for reduced-order modelling of fluid flows, *Nat. Commun.* 15 (1) (2024). <https://doi.org/10.1038/s41467-024-45578-4>
- [14] Z. Li, S. Patil, F. Ogoke, D. Shu, W. Zhen, M. Schneider, J.R. Buchanan, A. Barati Farimani, Latent neural PDE solver: a reduced-order modeling framework for partial differential equations, *J. Comput. Phys.* 524 (2025) 113705. <https://doi.org/10.1016/j.jcp.2024.113705>
- [15] B. Lusch, J.N. Kutz, S.L. Brunton, Deep learning for universal linear embeddings of nonlinear dynamics, *Nat. Commun.* 9 (1) (2018). <https://doi.org/10.1038/s41467-018-07210-0>
- [16] J. Nathan Kutz, J.L. Proctor, S.L. Brunton, Applied Koopman theory for partial differential equations and data-driven modeling of spatio-temporal systems, *Complexity* 2018 (1) (2018). <https://doi.org/10.1155/2018/6010634>
- [17] S.L. Brunton, J.L. Proctor, J.N. Kutz, Discovering governing equations from data by sparse identification of nonlinear dynamical systems, *Proc. Natl. Acad. Sci.* 113 (15) (2016) 3932–3937. <https://doi.org/10.1073/pnas.1517384113>
- [18] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, A. Anandkumar, Neural operator: learning maps between function spaces with applications to PDEs, *J. Mach. Learn. Res.* 24 (89) (2023) 1–97. <http://jmlr.org/papers/v24/21-1524.html>
- [19] F. Bartolucci, E. de Bezenac, B. Raonic, R. Molinaro, S. Mishra, R. Alaifari, Representation equivalent neural operators: a framework for alias-free operator learning, in: *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. <https://openreview.net/forum?id=7LSEkvEGCM>
- [20] L. Lu, P. Jin, G. Pang, Z. Zhang, G.E. Karniadakis, Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators, *Nat. Mach. Intell.* 3 (3) (2021) 218–229. <https://doi.org/10.1038/s42256-021-00302-5>
- [21] Z. Li, N.B. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Fourier neural operator for parametric partial differential equations, in: *National Conference on Learning Representations*, 2021. <https://openreview.net/forum?id=c8P9NQVtmnO>
- [22] B. Raonic, R. Molinaro, T. De Ryck, T. Rohner, F. Bartolucci, R. Alaifari, S. Mishra, E. de Bézenac, Convolutional neural operators for robust and accurate learning of PDEs, *Adv. Neural Inf. Process. Syst.* 36 (2023) 77187–77200.
- [23] L. Lu, X. Meng, S. Cai, Z. Mao, S. Goswami, Z. Zhang, G.E. Karniadakis, A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data, *Comput. Methods Appl. Mech. Eng.* 393 (2022) 114778. <https://doi.org/10.1016/j.cma.2022.114778>
- [24] Z. Hao, Z. Wang, H. Su, C. Ying, Y. Dong, S. Liu, Z. Cheng, J. Song, J. Zhu, Gnot: a general neural operator transformer for operator learning, in: *International Conference on Machine Learning*, PMLR, 2023, pp. 12556–12569.
- [25] G. Kissas, J.H. Seidman, L.F. Guilhoto, V.M. Preciado, G.J. Pappas, P. Perdikaris, Learning operators with coupled attention, *J. Mach. Learn. Res.* 23 (215) (2022) 1–63.
- [26] Z. Li, N. Kovachki, C. Choy, B. Li, J. Kossaifi, S. Otta, M.A. Nabian, M. Stadler, C. Hundt, K. Azizzadenesheli, et al., Geometry-informed neural operator for large-scale 3d pdes, *Adv. Neural Inf. Process. Syst.* 36 (2024).
- [27] J.K. Gupta, J. Brandstetter, Towards multi-spatiotemporal-scale generalized pde modeling, *arXiv preprint arXiv:2209.15616* (2022).
- [28] P. Jin, S. Meng, L. Lu, MIONet: Learning multiple-input operators via tensor product, *SIAM J. Sci. Comput.* 44 (6) (2022) A3490–A3514.
- [29] J. Brandstetter, D.E. Worrall, M. Welling, Message passing neural PDE solvers, in: *International Conference on Learning Representations*, 2022. <https://openreview.net/forum?id=vSix3HPYKSU>
- [30] F. Pichi, B. Moya, J.S. Hesthaven, A graph convolutional autoencoder approach to model order reduction for parametrized PDEs, *J. Comput. Phys.* 501 (2024) 112762. <https://doi.org/10.1016/j.jcp.2024.112762>
- [31] N.R. Franco, S. Fresca, F. Tombari, A. Manzoni, Deep learning-based surrogate models for parametrized PDEs: handling geometric variability through graph neural networks, *Chaos* 33 (12) (2023). <https://doi.org/10.1063/5.0170101>
- [32] L. Equer, T.K. Rusch, S. Mishra, Multi-scale message passing neural pde solvers, *arXiv preprint arXiv:2302.03580* (2023).
- [33] R.T.Q. Chen, Y. Rubanova, J. Bettencourt, D.K. Duvenaud, Neural ordinary differential equations, *Adv. Neural Inf. Process. Syst.* 31 (2018).
- [34] S. Wiewel, M. Becher, N. Thuerey, Latent space physics: towards learning the temporal evolution of fluid flow, *Comput. Graph. Forum* 38 (2) (2019) 71–82. <https://doi.org/10.1111/cgf.13620>
- [35] P.Y. Chen, J. Xiang, D.H. Cho, Y. Chang, G.A. Pershing, H.T. Maia, M.M. Chiaramonte, K. Carlberg, E. Grinspun, CROM: continuous reduced-order modeling of PDEs using implicit neural representations, *International Conference on Learning Representations* (2023).
- [36] T. Wang, C. Wang, Latent neural operator for solving forward and inverse PDE problems, in: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. <https://openreview.net/forum?id=VLw8ZyKfcm>
- [37] D.M. Knigge, D. Wessels, R. Valperga, S. Papa, J.-J. Sonke, E.J. Bekkers, S. Gavves, Space-time continuous PDE forecasting using equivariant neural fields, in: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. <https://openreview.net/forum?id=wN5AgPD0J0>
- [38] Y. Yin, M. Kirchmeyer, J.-Y. Franceschi, A. Rakotomamonjy, p. gallinari, Continuous PDE dynamics forecasting with implicit neural representations, in: *The Eleventh International Conference on Learning Representations*, 2023. <https://openreview.net/forum?id=B73niNjbPs>
- [39] X. Xie, S. Mowlavi, M. Benosman, Smooth and Sparse Latent Dynamics in Operator Learning with Jerk Regularization, *arXiv preprint arXiv:2402.15636* (2024).
- [40] Q. Zhuang, J.M. Lorenzi, H.-J. Bungartz, D. Hartmann, Model order reduction based on Runge–Kutta neural networks, *Data-Centric Eng.* 2 (2021). <https://doi.org/10.1017/dce.2021.15>
- [41] T. Wen, K. Lee, Y. Choi, Reduced-order modeling for parameterized PDEs via implicit neural representations, *NeurIPS 2023 Workshop: Machine Learning and the Physical Sciences* (2023).
- [42] X. He, Y. Choi, W.D. Fries, J.L. Belof, J.-S. Chen, gLaSDI: parametric physics-informed greedy latent space dynamics identification, *J. Comput. Phys.* 489 (2023) 112267. <https://doi.org/10.1016/j.jcp.2023.112267>

- [43] C. Bonneville, Y. Choi, D. Ghosh, J.L. Belof, GPLaSDI: Gaussian process-based interpretable latent space dynamics identification through deep autoencoder, *Comput. Methods Appl. Mech. Eng.* 418 (2024) 116535. <https://doi.org/10.1016/j.cma.2023.116535>
- [44] J.S.R. Park, S.W. Cheung, Y. Choi, Y. Shin, tLaSDI: thermodynamics-informed latent space dynamics identification, *Comput. Methods Appl. Mech. Eng.* 429 (2024) 117144. <https://doi.org/10.1016/j.cma.2024.117144>
- [45] W. Anderson, K. Chung, Y. Choi, mLaSDI: Multi-stage latent space dynamics identification, *arXiv preprint arXiv:2506.09207* (2025).
- [46] J. Hagnberger, M. Kalimuthu, D. Muekamp, M. Niepert, Vectorized conditional neural fields: a framework for solving time-dependent parametric partial differential equations, in: R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, F. Berkenkamp (Eds.), *Proceedings of the 41st International Conference on Machine Learning*, 235 of *Proceedings of Machine Learning Research*, PMLR, 2024, pp. 17189–17223. <https://proceedings.mlr.press/v235/hagnberger24a.html>.
- [47] M. Takamoto, F. Alesiani, M. Niepert, Learning neural pde solvers with parameter-guided channel attention, in: *International Conference on Machine Learning*, PMLR, 2023, pp. 33448–33467.
- [48] M. Takamoto, T. Praditua, R. Leiteritz, D. MacKinlay, F. Alesiani, D. Pflüger, M. Niepert, Pdebench: an extensive benchmark for scientific machine learning, *Adv. Neural Inf. Process. Syst.* 35 (2022) 1596–1611.
- [49] C. Eastwood, C.K.I. Williams, A framework for the quantitative evaluation of disentangled representations, in: *International Conference on Learning Representations*, 2018. <https://api.semanticscholar.org/CorpusID:19571619>.
- [50] I. Higgins, D. Amos, D. Pfau, S. Racanière, L. Matthey, D.J. Rezende, A. Lerchner, Towards a Definition of Disentangled Representations, *abs/1812.02230* (2018). <https://api.semanticscholar.org/CorpusID:54447715>.
- [51] P. Kidger, On Neural Differential Equations, *abs/2202.02435* (2022). <https://api.semanticscholar.org/CorpusID:246634262>.
- [52] U.M. Ascher, L.R. Petzold, *Computer methods for ordinary differential equations and differential-algebraic equations*, 1998. <https://api.semanticscholar.org/CorpusID:32366732>.
- [53] C. Aicher, N.J. Foti, E.B. Fox, Adaptively truncating backpropagation through time to control gradient bias, in: R.P. Adams, V. Gogate (Eds.), *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, 115 of *Proceedings of Machine Learning Research*, PMLR, 2020, pp. 799–808. <https://proceedings.mlr.press/v115/aicher20a.html>.
- [54] J. Cha, J. Thiyagalingam, Orthogonality-enforced latent space in autoencoders: an approach to learning disentangled representations, in: A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, J. Scarlett (Eds.), *Proceedings of the 40th International Conference on Machine Learning*, 202 of *Proceedings of Machine Learning Research*, PMLR, 2023, pp. 3913–3948. <https://proceedings.mlr.press/v202/cha23b.html>.
- [55] J.C. Butcher, Coefficients for the study of Runge–Kutta integration processes, *J. Aust. Math. Soc.* 3 (2) (1963) 185–201. <https://doi.org/10.1017/S1446788700027932>
- [56] V. Dumoulin, F. Visin, A guide to convolution arithmetic for deep learning, 2016, <https://arxiv.org/abs/1603.07285>. <https://doi.org/10.48550/ARXIV.1603.07285>
- [57] D. Hendrycks, K. Gimpel, Gaussian error linear units (gelus), *arXiv preprint arXiv:1606.08415* (2016).
- [58] S. Ioffe, C. Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift, in: *Proceedings of the 32nd International Conference on Machine Learning - Volume 37, ICML'15, JMLR.org*, 2015, p. 448–456.
- [59] J.L. Ba, Layer normalization, *arXiv preprint arXiv:1607.06450* (2016).
- [60] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: surpassing human-level performance on imagenet classification, in: *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.
- [61] H. Gholamalizadeh, H. Khosravi, Pooling methods in deep neural networks, a review, *arXiv preprint arXiv:2009.07485* (2020).
- [62] E. Perez, F. Strub, H. De Vries, V. Dumoulin, A. Courville, Film: visual reasoning with a general conditioning layer, in: *Proceedings of the AAAI Conference on Artificial Intelligence*, 32, 2018.
- [63] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980* (2014).
- [64] L.N. Smith, N. Topin, Super-convergence: very fast training of neural networks using large learning rates, in: *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, 11006, SPIE, 2019, pp. 369–386.
- [65] O. Ronneberger, P. Fischer, T. Brox, U-net: convolutional networks for biomedical image segmentation, in: *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5–9, 2015, Proceedings, Part III 18*, Springer, 2015, pp. 234–241.
- [66] L. Serrano, L.L. Boudec, A.K. Koupai, T.X. Wang, Y. Yin, J.-N. Vittaut, P. Gallinari, Operator learning with neural fields: tackling PDEs on general geometries, in: *NeurIPS*, 2023.
- [67] Y. Xie, T. Takikawa, S. Saito, O. Litany, S. Yan, N. Khan, F. Tombari, J. Tompkin, V. Sitzmann, S. Sridhar, Neural fields in visual computing and beyond, in: *Computer Graphics Forum*, 41, Wiley Online Library, 2022, pp. 641–676.
- [68] S. Cao, Choose a transformer: Fourier or Galerkin, in: A. Beygelzimer, Y. Dauphin, P. Liang, J.W. Vaughan (Eds.), *Advances in Neural Information Processing Systems*, 2021. <https://openreview.net/forum?id=ssohLcmn4-r>.
- [69] Z. Li, K. Meidani, A.B. Farimani, Transformer for partial differential equations' operator learning, *Trans. Mach. Learn. Res.* (2023). , <https://openreview.net/forum?id=EPPqt3uERT>.
- [70] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, *J. Comput. Phys.* 378 (2019) 686–707. <https://doi.org/10.1016/j.jcp.2018.10.045>
- [71] F. Alsayyari, Z. Perkó, M. Tiberge, J.L. Kloosterman, D. Lathouwers, A fully adaptive nonintrusive reduced-order modelling approach for parametrized time-dependent problems, *Comput. Methods Appl. Mech. Eng.* 373 (2021) 113483. <https://doi.org/10.1016/j.cma.2020.113483>