Journal of Computational Science 53 (2021) 101408

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Reduced order modeling for parameterized time-dependent PDEs using spatially and memory aware deep learning

Nikolaj T. Mücke [a,*], Sander M. Bohté [a], Cornelis W. Oosterlee [a,b]

[a] CWI, Science Park 123, 1098 XG Amsterdam, Netherlands
[b] Mathematical Institute, Utrecht University, Netherlands

ARTICLE INFO

Keywords:
Parameterized PDEs
Spatio-temporal dynamics
Reduced order modeling
Deep learning

ABSTRACT

We present a novel reduced order model (ROM) approach for parameterized time-dependent PDEs based on modern learning. The ROM is suitable for multi-query problems and is nonintrusive. It is divided into two distinct stages: a nonlinear dimensionality reduction stage that handles the spatially distributed degrees of freedom based on convolutional autoencoders, and a parameterized time-stepping stage based on memory aware neural networks (NNs), specifically causal convolutional and long short-term memory NNs. Strategies to ensure generalization and stability are discussed. To show the variety of problems the ROM can handle, the methodology is demonstrated on the advection equation, and the flow past a cylinder problem modeled by the incompressible Navier–Stokes equations.

1. Introduction

Simulations based on first-principles models often form an essential element for understanding, designing, and optimizing problems in, for example, physics, engineering, chemistry, and economics. However, with an increasing complexity of the mathematical models under consideration, it is not always possible to achieve the desired fidelity of such simulations in a satisfactory time frame. This is especially the case when dealing with multi-query and/or real time problems as encountered in uncertainty quantification and model predictive control, where the computational model is typically parameterized.

There are several approaches to reduce the computation time bottleneck. The arguably most common ones include high-performance computations [1], high-order discretizations [2], iterative and/or multigrid methods [3,4], and reduced order modeling (ROM) [5]. High-performance computing may be costly; the improvements due to high-order discretization strongly depend on the smoothness of solutions at hand, and iterative methods are highly dependent on being able to identify suitable preconditioners. Furthermore, these approaches may suffer from the curse of dimensionality. ROM, a relatively recent research area, is an interesting alternative to the other approaches.

The ROM solution process is generally divided into two distinct stages [5]: a so-called “offline stage”, in which the reduced model is derived, and an “online stage”, where the reduced model is utilized and solved. Popular choices for the two stages are the proper orthogonal decomposition (POD) model definition, combined with a (Galerkin) projection procedure in the online stage [5,6]. Whereas this combination has shown important successes, it has also been shown that the POD and projection approaches perform worse in certain settings, such as for advection-dominated or nonlinear problems. Furthermore, projection-based methods are intrusive, as they require access to the underlying high-fidelity model. Nowadays, it is a reasonable assumption that an industrial model is not directly accessable, and therefore non-intrusive approaches, i.e. approaches that are only based on a series of snapshots of solutions, are increasingly interesting alternatives.

Machine learning has recently gained the attention from the scientific computing community due to great successes of artificial intelligence in various settings. Specifically artificial neural networks (ANNs), often simply denoted neural networks (NNs), have shown remarkable results in tasks such as image analysis and speech recognition. Much of the success has been boosted further by the availability of open source software frameworks, such as PyTorch [7] and Tensorflow [8], which have made implementation and training possible without expert knowledge and the availability of computation accelerating hardware, such as GPUs, has made training of very large models feasible. These recent advances have accelerated research in especially deep learning, i. e. multilayered NNs, which was not possible few years ago, resulting in many NN architectures specialized in certain tasks, such as time series

* Corresponding author.
E-mail addresses: nikolaj.mucke@cwi.nl (N.T. Mücke), S.M.Bohte@cwi.nl (S.M. Bohté), c.w.oosterlee@uu.nl (C.W. Oosterlee).

https://doi.org/10.1016/j.jocs.2021.101408
Received 6 July 2020; Received in revised form 20 May 2021; Accepted 10 June 2021
Available online 21 June 2021

forecasting and dimensionality reduction.

NNs have gained traction within the mathematics, numerical analysis, and engineering communities either as a replacement or as a supplement to conventional function approximation methods. For an overview of articles, prospects, and future challenges, see e.g. [9–11]. In this paper, we will combine ROM and machine learning in both the offline and the online stages to showcase the potential of using these technologies on conventional problems from scientific computing.

Important work has already been done on the topic of NN-based ROM. For example, the authors in [12–14] have used proper orthogonal decomposition (POD) for dimensionality reduction and data-driven methods to map the parameters to the reduced basis coefficients. However, none of these approaches considers time-dependent problems.

Examples of approaches that utilize POD and also deal with time are found in [15–17]. A difference with our method is that these approaches do not compute the unsteady states by means of time stepping, but rather consider time an extra parameter. Hence, it is not possible to advance a solution in time from an arbitrary point on the trajectory. The above mentioned approaches are based on a linear dimensionality reduction scheme in the form of the POD.

In [18] a CAE is utilized for a nonlinear dimensionality reduction while the time stepping is done, intrusively, using multistep methods on the reduced model, derived from a Galerkin projection procedure. Due to the Galerkin projection of the high-fidelity model, this approach requires access to the underlying model. In [19], a CAE is also used for model reduction and an LSTM is used for time stepping of the reduced state, but with the problem parameters kept fixed. The paper [18] also considers CAEs for dimensionality reduction and a dense feedforward neural network (DFFNN) to map the parameters but without any time stepping procedure. Closest our work is [20], where a CAE is employed to reduce the dimension and a causal convolutional neural network (CCNN) to encode previous reduced states. The CCNN and the DFFNN are trained independently of each other. Stability of the methodology is not discussed in that paper and neither are comparisons with alternative regression techniques.

In our work, we present a non-intrusive framework, based on deep learning, for computing parameterized spatio-temporal dynamics. The resulting reduced order model is divided into two distinct stages: Firstly, a dimensionality reduction stage based on convolutional autoencoders (CAEs), and secondly a memory-aware NN stage for parameterized time stepping. This methodology utilizes the effectiveness of CAEs as nonlinear dimensionality reduction techniques for spatially distributed data. To discuss the advantages of using CAEs, we make a comparison to the widely used linear counterpart, POD. Specifically we show that POD is a special case of an autoencoder. Furthermore, we present a flexible neural network structure for time stepping that takes into consideration previous states as well as parameters. The framework is quite general and allows for various types of neural network architectures, hence allowing state-of-the-art techniques that fit the problem at hand. We present and compare two modern time series forecasting architectures, long short-term memory (LSTM) networks [21], and causal convolutional neural networks (CCNNs) [22]. Furthermore, we present and discuss a series of approaches to ensure stability and generalization of the time stepping network. The scheme presented in this paper is compared to similar, but different, schemes, like Gaussian processes with POD.

To the best of our knowledge, there is no other work on deep learning-based ROM that is non-intrusive, uses CAEs for dimensionality reduction, has memory-aware and parameterized time stepping, and discusses practical approaches to ensure stability and generalization. The result is a flexible offline-online scheme that works for various physical phenomena and can easily be modified according to the specific problem at hand. This makes the presented approach suitable for multi-query problems.

The structure of the present paper is as follows. In Section 2 we present parameterized time-dependent PDEs. In Section 3 we discuss dimensionality reduction. Furthermore, we discuss how convolutional autoencoders are used for nonlinear dimensionality reduction. In Section 4 we present the parameterized memory-aware time stepping neural network. In Section 5 we showcase the performance on two test problems: a linear advection equation and a flow past a cylinder modeled by the incompressible Navier–Stokes equations.

2. Parameterized time-dependent PDEs

The model under consideration is of the form

$$\partial_t u(t,x;\mu) = F(t,x,u;\mu), \quad u(0,x;\mu) = u_0(x;\mu), \tag{1}$$

where F is a (nonlinear) differential operator, $u : \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{N_p} \to \mathbb{R}$ or $u : \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{N_p} \to \mathbb{R}^d$, $t \in [0, T]$, and $x \in \mathbb{R}^d$. Eq. (1) is a very general parameterized PDE. μ is to be considered a vector of parameters on which the solution depends. These parameters could be diffusion rate, Reynolds number, parameterize an initial or boundary condition, etc. For technical reasons, the parameter space $\mathcal{P}$ is chosen to be a compact subspace of $\mathbb{R}^{N_p}$ [5].

Spatially discretizing (1), using finite elements, finite volumes, finite differences [23], gives the following finite-dimensional dynamical system,

$$\partial_t u_h(t,\mu) = F_h(t, u_h(t,\mu);\mu), \quad u_h(0,\mu) = u_h^0(\mu), \tag{2}$$

h defines the granularity of the discretization, i.e. grid size, number of elements, etc. We will not go into details regarding these discretizations and it should be assumed that the discretized system is stable and converges to the exact solution when granularity is refined. $u_h(t,\mu) \in \mathbb{R}^{N_h}$ will be referred to as the high-fidelity or full-order solution.

The manifold of high-fidelity solutions, parameterized by time and the parameters, is called the spatial discrete solution manifold,

$$M_h = \{u_h(t,\mu) \,|\, \mu \in \mathcal{P},\, t \in [0,T]\} \subset \mathbb{R}^{N_h}, \tag{3}$$

Our goal is to approximate this manifold.

Defining a time discretization, $\{t_0, t_1, \ldots, t_{N_t}\}$, $t_n = n\delta t$, and using a time stepping scheme gives us the time discrete approximation of (2):

$$u_h^{n+1}(\mu) = F_{h,\delta t}(u_h^n;\mu), \tag{4}$$

where $u^n(\mu) = u(t_n, \mu)$. We will refer to $u_h^n(\mu)$ as the state at time step n. Note that the discrete time evolution map is not necessarily restricted to only depend on the last state, but can take in several previous states, as is done in, e.g. multistep methods, or it could depend on the current state as in implicit methods. We can now define the time-discrete high-fidelity solution manifold:

$$M_{h,\delta t} = \left\{u_h^n(\mu) | \mu \in \mathcal{P},\, n = 0, \ldots N_t\right\} \subset \mathbb{R}^{N_h}. \tag{5}$$

The subscripts h and δt refer to the chosen spatial and time discretizations, respectively. $M_{h,\delta t}$ can be seen as the set of discrete state trajectories parameterized by the set of parameters.

In general, N_h will be very large, which makes advancing the state with (4) for many time steps time consuming. This is especially the case when dealing with high-dimensional domains and multiphysics problems. It is indeed a problem when dealing with multi-query problems such as uncertainty quantification and data assimilation or when real-time solutions are of importance as in real-time control settings and digital twins.

3. Dimensionality reduction

The fundamental idea of dimensionality reduction is that the minimal number of variables necessary to represent the state, also called the intrinsic dimension, of the dynamical system is low compared to the dimension of the high-fidelity model. However, identifying an optimal low-dimensional representation is, in general, not a trivial task. In this section we will give a brief overview of linear dimensionality reduction, particularly, the well-known proper orthogonal decomposition (POD). Then, from the linear outset, we will describe the more general case of nonlinear dimensionality reduction.

In general, for both linear and nonlinear dimensionality reduction, we assume that a state, $u_h^n(\mu) \in \mathbb{R}^{N_h}$, can be approximated,

$$u_h^n(\mu) \approx \Phi(u_h^n) = \Phi_{\text{dec}} \circ \Phi_{\text{enc}}(u_h^n(\mu)), \tag{6}$$

where $\Phi_{\text{enc}}(u_h) \in \mathbb{R}^{N_l}$ with $N_l \ll N_h$. Φ_{enc} is referred to as the *encoder* and Φ_{dec} the *decoder*. The encoder transforms the high-dimensional input to a *latent space* of low dimension and the decoder transforms the latent variable back to the high-fidelity space. The latent space is often denoted the *reduced trial manifold*. The state at time step n in the latent space is denoted $u_l^n(\mu) = \Phi_{\text{enc}}(u_h^n(\mu))$, and will be referred to as the latent state.

Ideally, Φ reconstructs the input perfectly for any given parameters and time step. However, that is, in general, not possible. The precision of the reconstruction is heavily dependent on the dimension of the latent space, as this determines the amount of compression applied. One computes Φ by choosing a latent dimension, N_l, and then solving the minimization problem,

$$\Phi^* = \underset{\Phi}{\text{arg}min}\sqrt{\int_{\mu\in P}\left[\sum_{n=0}^{N_t} ||u_h^n(\mu) - \Phi(u_h^n(\mu))||_2^2\right]d\mu}, \tag{7}$$

where $||\cdot||_2$ denotes the l^2-norm. Theoretically, the reconstruction error should decrease when N_l is increased until the intrinsic dimension of the problem is reached. From thereon, increasing the dimension of the latent space should have very little effect on the reconstruction error.

There are many ways of solving (7) [5]. In this paper we focus on a data-driven approach, sometimes referred to as the *method of snapshots*. A snapshot is a high-fidelity solution for a given parameter realization at a certain time. The idea of this approach is to make N_{train} samples from the parameter space and then compute a series of $N_T + 1$ snapshots, i.e. trajectories, per parameter sample,

$$M_{N_{\text{train}},h,\delta t} = \left\{u_h^0(\mu_1),\dots,u_h^{N_t}(\mu_1),u_h^0(\mu_2),\dots,u_h^{N_t}(\mu_2),\dots,u_h^0(\mu_{N_{\text{train}}}),\dots,u_h^{N_t}(\mu_{N_{\text{train}}})\right\} \tag{8}$$

Then (7) is rewritten into an empirical minimization problem:

$$\Phi^* = \underset{\Phi}{\text{arg}min}\sqrt{\sum_{i=1}^{N_{\text{train}}}\sum_{n=0}^{N_t} ||u_h^n(\mu_i) - \Phi(u_h^n(\mu_i))||_2^2}. \tag{9}$$

The idea is that sampling a finite number of discrete trajectories a sufficient number of times yields a good enough representation of the time-discrete high-fidelity solution manifold. It should be noted that computing (8) is potentially very expensive and even infeasible in some cases.

When a reduction scheme is computed, one can then compute the parameterized trajectories in the latent space, by

$$u_l^{n+1}(\mu) = F_{l,\delta t}(u_l^n;\mu), \quad u_l^0(\mu) = \Phi_{\text{enc}}\left(u_h^0(\mu)\right), \tag{10}$$

from which the trajectories in the high-fidelity space can be recovered by $u_h^n(\mu) = \Phi_{\text{dec}}\left(u_l^n(\mu)\right)$. $F_{l,\delta t}$ can be derived in many ways and much time and effort have been put into deriving optimal latent dynamics.

3.1. Linear dimensionality reduction

In linear dimensionality reduction the strategy is to find a reduced linear trial manifold of low dimension. Since the sought manifold is linear it can be written as the column space, Col(V) of some matrix, $V \in \mathbb{R}^{N_h \times N_l}$. The column space is the space spanned by the columns of the matrix V. From the orthogonal projection theorem, it can be shown that the optimal projection onto a latent linear space is given by

$$u_h \approx \mathrm{V}\mathrm{V}^T u_h. \tag{11}$$

Hence, this is a special case of (6) where

$$\Phi = \mathrm{V}\mathrm{V}^T, \quad \Phi_{\text{enc}} = V^T, \quad \Phi_{\text{dec}} = V. \tag{12}$$

This simplification reduces (9) to

$$V^* = \underset{V}{\text{arg}min}\sqrt{\sum_{i=1}^{N_{\text{train}}}\sum_{n=0}^{N_t} ||u_h^n(\mu_i) - \mathrm{V}\mathrm{V}^T u_h^n(\mu_i)||_2^2}, \tag{13}$$

often accompanied by the constraint that the columns of V are orthogonal, $V^T V = 0$. It can be shown that (13) has an exact solution [5]. By collecting the snapshots in a *snapshot matrix*,

$$S = \left[u_h^0(\mu_1)|\dots|u_h^{N_t}(\mu_1)|\dots|u_h^0(\mu_{N_{\text{train}}})|\dots|u_h^{N_t}(\mu_{N_{\text{train}}})\right], \tag{14}$$

one can show that the optimal $V \in \mathbb{R}^{N_h \times N_l}$ is given by the first N_l *left singular vectors*. The left singular vectors are computed through the singular value decomposition (SVD),

$$S = U\Sigma Z^T, \tag{15}$$

where U is a matrix whose columns are the left singular vectors, Z is a matrix whose columns are the right singular vectors, and Σ is a diagonal matrix with the singular values on the diagonal. V is then chosen to be the first N_l columns of U. This method of obtaining V is the *proper orthogonal decomposition* (POD) [5], also denoted *principal component analysis* (PCA) [24].

To obtain $F_{l,\delta t}$ a Petrov Galerkin projection is often performed, which yields

$$F_{l,\delta t}(u_l^n;\mu) = W^T F_{h,\delta t}(\mathrm{V}u_l^n;\mu). \tag{16}$$

When $W = V$ it is denoted the Galerkin projection. This approach is *intrusive* which means that direct access to the model, $F_{h,\delta t}$, is required. Furthermore, in the online phase a transformation between the latent space and the high-fidelity space must be performed in each time step in order to be able to evaluate $F_{h,\delta t}(\mathrm{V}u_l^n;\mu)$, which slows down the computations. Various methods to circumvent that problem, such the discrete empirical interpolation methods [5], already exist. In recent years there are also many studies exploring approximating $F_{l,\delta t}$ with neural networks [25,17].

While there are many advantages of a linear reduction scheme, such as the explicit solution to (13), there are, indeed, disadvantages as well. A significant problem is the restriction to a linear trial manifold. The optimal trial manifold, i.e. the trial manifold of the intrinsic dimension, is rarely linear. Especially for advection-dominated and nonlinear problems, it has been shown that a linear reduced approximation does not necessarily lead to significant speed-ups.

3.2. Nonlinear dimensionality reduction

The extension from linear to nonlinear dimensionality reduction comes naturally and addresses several of the drawbacks of linear dimensionality reduction. The fundamental difference is that we remove the constraint that the latent space has to be a linear manifold. Due to this generalization, we cannot write the projection operator as the matrix product VV^T anymore, but instead we must use the general form in (6), where Φ_{enc} and Φ_{dec} can be any type of nonlinear functions. This gives rise to a major difference in solving (9), since no general exact solution exists and therefore (9) will be solved numerically.

Even though extra approximation steps have to be introduced in the nonlinear case, the potential gains will, in some cases, outweigh this hurdle. This is due to the fact that with a nonlinear reduction scheme it is theoretically possible to reduce the high-fidelity space down to its

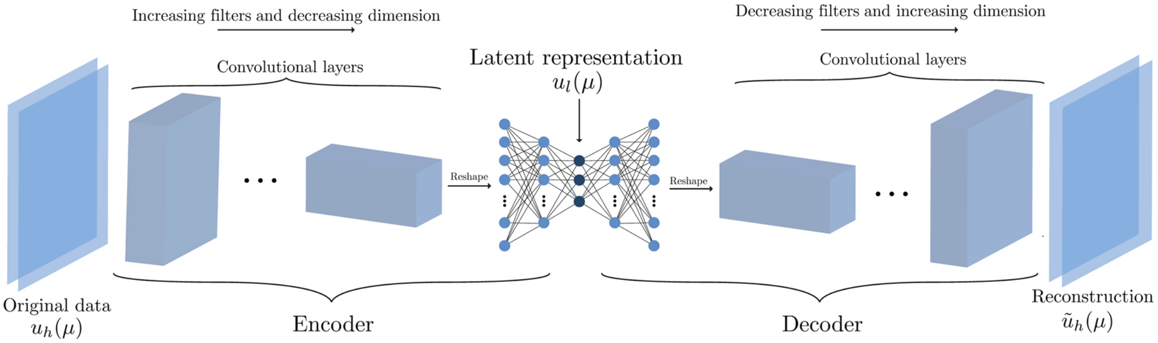


Fig. 1. Illustration of a convolutional autoencoder.

intrinsic dimension, N_P+1. However, this depends on the choice of Φ and the minimization scheme.

A common method for nonlinear dimensionality reduction from the machine learning communities is, among others, kernel PCA. Here, the nonlinear manifold is embedded into a linear space, often of higher dimension, using a predefined nonlinear mapping, $\psi:\mathbb{R}^{N_h}\to\mathbb{R}^{N_k}$, $N_k>N_h$. From thereon, a linear PCA is performed on the high-dimensional linear data. In order to speed up computations the so-called kernel trick is typically invoked. Utilizing that the nonlinear embedding induces a kernel, $K=k(\psi(x),\ \psi(y))=\psi(x)^T\psi(y)$, one can compute the low-dimensional basis without explicitly transforming the data and perform PCA in the high-dimensional space. For more details see [26].

This approach works well in many cases but suffers from one crucial downside: Choosing the nonlinear mapping, ψ, or the kernel, K, is far from trivial. There exist no clear guiding principles that work across several cases.

3.2.1. Autoencoders

To overcome the problems of other nonlinear dimensionality reduction methods, such as kernel PCA and DEIM, we present autoencoders (AEs). AEs are a type of NN. For a brief introduction to NNs and the terminology used in this paper, see Appendix A. In the context of dimensionality reduction one can interpret an AE as a kernel PCA where the kernel is learned during the training process. Thus, one circumvents the problem of choosing a suitable kernel. Note that this interpretation is merely presented in order to give an intuition of AEs in context of other methods. To further explain the connections between AEs and PCA, it is worth noting that a single hidden layer AE with linear activation functions is equivalent to PCA. A single hidden layered AE without bias terms can be written as

$$\Phi(u_h^n(\mu);\theta)=T_2\circ T_1(x)=W_2W_1u_h^n(\mu), \tag{17}$$

where $\theta=\{W_1,W_2\}$, $T_1:\mathbb{R}^{N_h}\to\mathbb{R}^{N_l}$, and $T_2:\mathbb{R}^{N_l}\to\mathbb{R}^{N_h}$ are linear maps and W_1 and W_2 are matrices. Typically, the mean squared error is chosen as the loss function for AEs, which gives the following minimization problem for the single hidden layer AE:

$$arg\,\underset{W_1,W_2}{min}\frac{1}{N_{\text{train}}N_t}\sum_{i=1}^{N_{\text{train}}}\sum_{n=0}^{N_t}\left|\left|u_h^n(\mu_i)-W_2W_1u_h^n(\mu_i)\right|\right|^2. \tag{18}$$

Hence, training a single layer AE is equivalent to solving the PCA minimization problem, eq. (13), without the orthogonality constraint. Conclusively, PCA, Eq. (11), can be considered a special case of an AE.

By dividing the AE into the encoder and decoder parts and allowing an arbitrary number of layers and nonlinear activation functions, it is easier to understand the similarities to linear dimensionality reduction and why AEs have the potential to perform significantly better. Consider the encoder part with a linear activation in the final layer,

$$\Phi_{\text{enc}}(u_h^n(\mu);\theta)=T_L^{\text{enc}}\circ\underbrace{\sigma_{L-1}^{\text{enc}}\circ T_{L-1}^{\text{enc}}\ldots\circ\sigma_1^{\text{enc}}\circ T_1^{\text{enc}}(u_h^n(\mu))}_{\psi_{\text{enc}}(u_h^n(\mu);\theta)}=W_L^{\text{enc}}\psi_{\text{enc}}(u_h^n(\mu);\theta)=z, \tag{19}$$

where $\psi_{\text{enc}}(x;\theta)=(\psi_{\text{enc}}^1(x;\theta),\ldots,\psi_{\text{enc}}^{N_e}(x;\theta))\in\mathbb{R}^{N_e}$ and $W_L^{\text{enc}}\in\mathbb{R}^{N_e\times N_l}$. For convenience we ignore bias terms. We see that this corresponds to a nonlinear embedding onto $\mathbb{R}^{N_e}$ and then a projection onto the space spanned by the vectors $\psi_{\text{enc}}^1,\ldots,\psi_{\text{enc}}^{N_e}$. This is similar to the idea behind kernel PCA. The difference is that in the AE framework we adjust the nonlinear embedding in the training instead of defining it beforehand. The decoder part is similarly written as

$$\Phi_{\text{dec}}(z;\theta)=T_L^{\text{dec}}\circ\underbrace{\sigma_{L-1}^{\text{dec}}\circ T_{L-1}^{\text{dec}}\ldots\circ\sigma_1^{\text{dec}}\circ T_1^{\text{dec}}(z)}_{\psi_{\text{dec}}(z)}=W_L^{\text{dec}}\psi_{\text{dec}}(z;\theta)=\widetilde{u}_h^n(\mu), \tag{20}$$

where $W_L^{\text{dec}}\in\mathbb{R}^{N_d\times N_h}$.

Note that this is merely a brief discussion of the topic of AEs aiming to give an intuitive understanding. For more details see [27].

3.2.2. Convolutional autoencoders

Convolutional autoencoders (CAEs) are a special type of AEs utilizing convolutional layers instead of dense layers. A brief introduction to convolutional neural networks (CNNs) can be found in Appendix A. It can be shown that dense and convolutional neural networks are equivalent regarding approximation rates [28], which means that theoretical approximation results for dense NNs translate directly to CNNs. For practical purposes, however, convolutional layers are often to be preferred due to especially the following two properties:

- *Local connections*, which utilizes that spatial nodes close to each other are highly correlated.
- *Shared weights*, which in practice makes the affine transformations very sparse and enables location invariant feature detection.

An additional advantage is that it is straightforward to handle multiple spatially distributed states. These occur in coupled PDEs such as the Navier-Stokes equations where one is dealing with both the x-, y- and z-components of the velocity field as well as the pressure field. In the framework of CAEs, these can all be included by interpreting them as different channels. This enables the possibility of including multiple spatial states without increasing the number of weights in the neural network significantly. The connection between PDEs and CNNs has already been made, see e.g. [29].

In Fig. 1 one sees an illustration of a CAE. The encoding consists of a series of convolutional layers with an increasing number of filters and decreasing dimension, effectively down sampling the number of degrees of freedom, followed by dense layers. Similarly, the decoding consists of a series of dense layers followed by a series of deconvolutional layers with a decreasing number of filters and increasing dimension, effectively up sampling. The down sampling is often performed by utilizing pooling

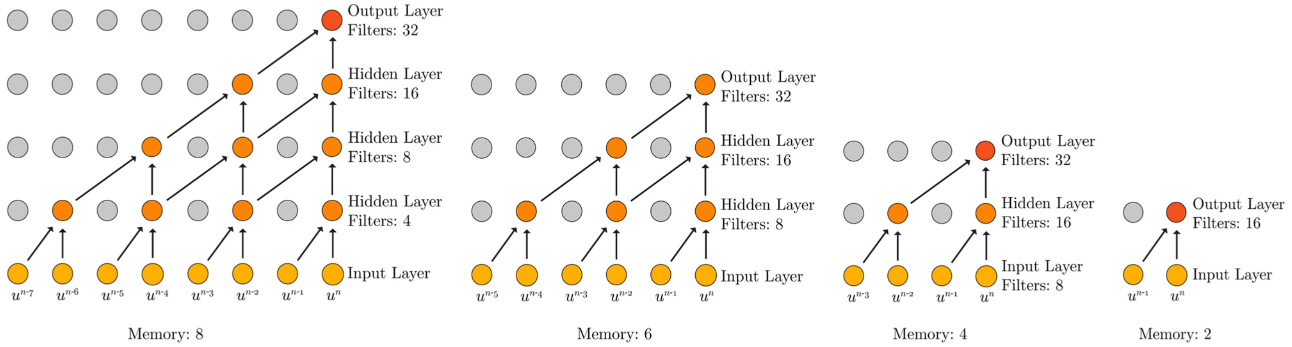


Fig. 2. CCNN network architectures for varying memory.

layers or strides larger than one.

It is worth noting that computing the decoder, Φ_{dec}, of a CAE in the training phase is effectively solving an inverse problem. Inverse problems are, in general, ill-posed and therefore require regularization. L^2-regularization, often referred to as weight decay, is frequently used, and results in the following minimization problem to solve:

$$\underset{\theta}{argmin}\frac{1}{N_{\text{train}}N_t}\sum_{i=1}^{N_{\text{train}}}\sum_{n=0}^{N_t}\left|\left|u_h^n(\mu_i)-\Phi(u_h^n(\mu_i);\theta)\right|\right|^2+\alpha||\theta||_2^2, \tag{21}$$

where α is a hyperparameter to be tuned. Besides ensuring well-posedness the term also ensures generalization.

4. Approximating parameterized time evolution using neural networks

In the previous section we presented the general framework for nonlinear dimensionality reduction and showcased how convolutional autoencoders fit into this framework. In this section we explain how neural networks will be utilized for approximating the dynamics in the latent space. For a brief review of the relevant types of neural networks, see again Appendix A. Neural networks have already shown to be able to approximate dynamical systems [20,30,31].

We aim to approximate the dynamics in the latent space non-intrusively by a function, $\Psi \approx F_{l,\delta t}$:

$$u^{n+1} = \Psi(u^n), \tag{22}$$

where Ψ is a neural network. The approximated latent states will be denoted, $\widetilde{u}_l^n(\mu)$, to distinguish from the encoded high-fidelity state, $u_l^n(\mu) = \Phi_{\text{enc}}(u_h^n(\mu))$. Thereby, we aim to achieve:

$$\begin{aligned}&\{\widetilde{u}_l^0(\mu_1),\dots,\widetilde{u}_l^{N_t}(\mu_1),\dots,\widetilde{u}_l^0(\mu_{N_{\text{train}}}),\dots,\widetilde{u}_l^{N_t}(\mu_{N_{\text{train}}})\}\\ &\quad\approx\{u_l^0(\mu_1),\dots,u_l^{N_t}(\mu_1),\dots,u_l^0(\mu_{N_{\text{train}}}),\dots,u_l^{N_t}(\mu_{N_{\text{train}}})\}\end{aligned} \tag{23}$$

4.0.1. *Taking larger steps*

Using high-fidelity methods for time stepping often includes some restrictions on the step size in order for the scheme to be stable. An example is the Courant–Friedrichs–Lewy (CFL) condition for advection-dominated problems [32]. With our strategy, where we aim to learn a neural network representation of the time evolution map, there is no immediate connection between step size and stability. Therefore, in order to speed up online computations, the neural network can be trained to learn to take steps of size $s\delta t$. Hence, $\Psi \approx F_{l,s\delta t}$.

In the offline phase, the high-fidelity trajectories are still computed with step size δt, to ensure stability, but only every sth step is used for training the NN:

$$\{u_h^0(\mu),u_h^1(\mu),u_h^2(\mu)\dots,u_h^{N_t}(\mu)\}\mapsto\{u_h^0(\mu),u_h^s(\mu),u_h^{2s}(\mu)\dots,u_h^{N_t}(\mu)\} \tag{24}$$

In general, $u_h^n(\mu)$ and $u_h^{n+1}(\mu)$, are highly correlated, which means that we gain very little extra information by using both in the training of the NN. Therefore, it makes sense only use every sth step to save memory and speed up the training. Hoever, the number s must be chosen according to various factors, like the requested detail of the dynamics in the online phase. It should further be kept in mind that larger s results in a more complicated map to learn, and thus complicates the training.

For simplicity we will use the notation $\{u_h^0(\mu),u_h^1(\mu),u_h^2(\mu)\dots,u_h^{N_t}(\mu)\}$ when referring to the trajectory used for training the neural network.

4.0.2. *Approximating the state vs. residual*

At first glance, it makes sense to train a neural network to approximate u_l^{n+1} directly given u^n. However, it is shown in [25,33] that learning the residual instead of the next state often improves the accuracy. Hence, we consider the case

$$u_l^{n+1} = \Psi(u^n) = u_l^n + R(u_l^n), \tag{25}$$

where R is approximated by a neural network. This practically makes Ψ what is often referred as a residual neural network.

4.0.3. *Incorporating memory*

In [25,19] the potential benefits of not only using the present state but also incorporating several previous time steps for the future predictions were shown. Therefore, we now consider

$$u_l^{n+1} = \Psi(u_l^n, u^{n-1},\dots,u_l^{n-\xi}) = u_l^n + R(u_l^n,u_l^{n-1},\dots,u_l^{n-\xi}), \tag{26}$$

where ξ is the number of previous states included as input into the residual computation by the NN. The idea of incorporating several previous timesteps can loosely be compared to linear multistep methods where the order of approximation can be increased by using several previous steps [32]. In contrast to linear multistep methods, NNs incorporate the previous time steps in a nonlinear fashion.

We consider two different types of networks in this paper: LSTM, and the CCNN. The two types of neural networks take varying computational time to train, have varying numbers of parameters, and vary in regards to how they interpret memory. In the appendix, there is a short

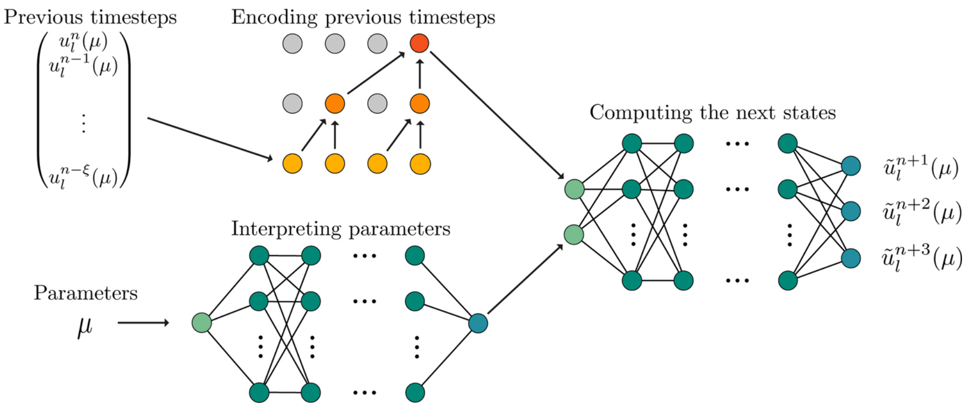
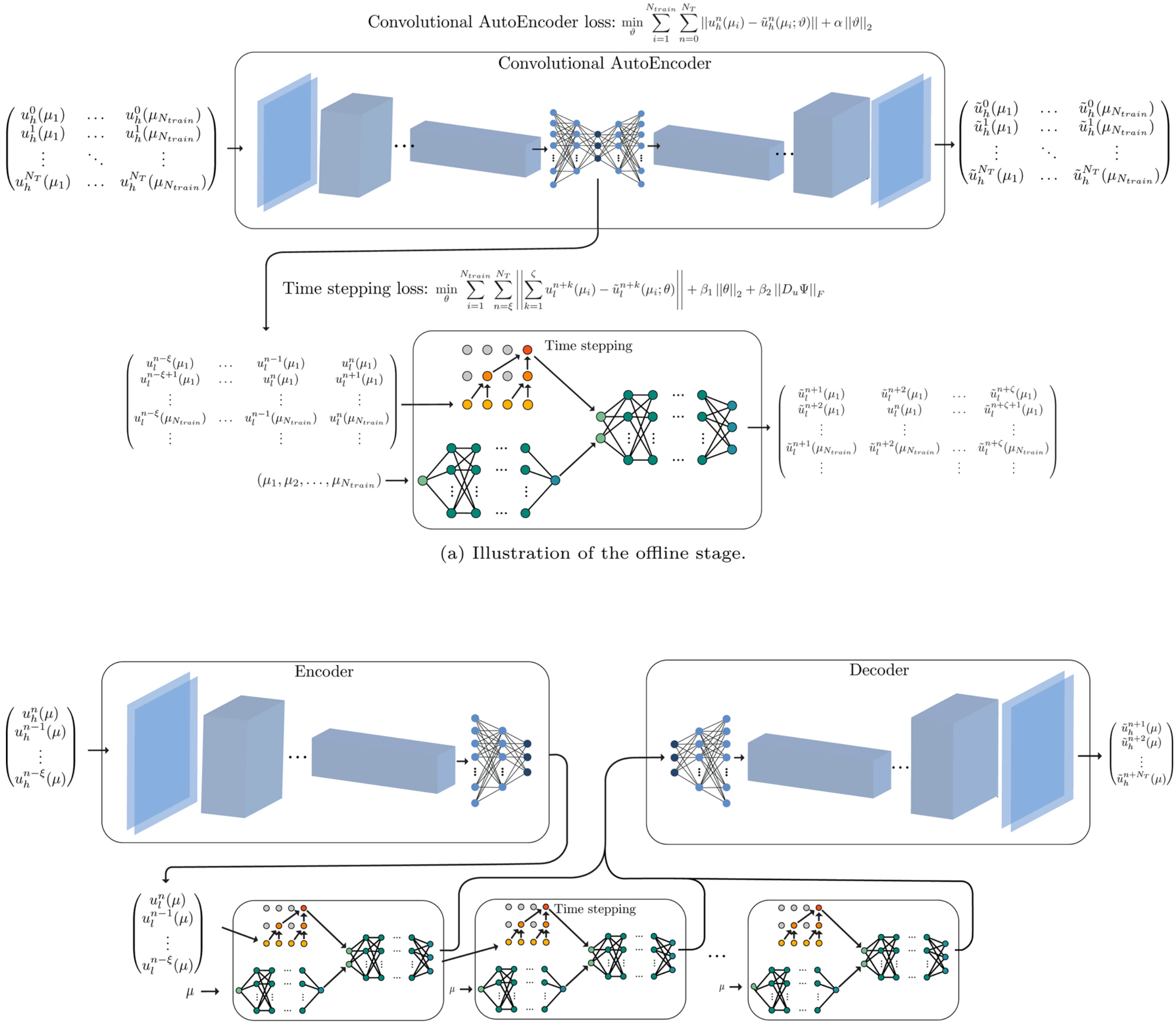


Fig. 3. Illustration of the parallel neural network structure. The “Encoding previous timesteps” part is visualized using the CCNN, but it should be noted that an LSTM network (or any suitable time series encoder) could be put in its place.


Convolutional AutoEncoder loss: $\min_{\vartheta} \sum_{i=1}^{N_{train}} \sum_{n=0}^{N_T} ||u_h^n(\mu_i) - \tilde{u}_h^n(\mu_i;\vartheta)|| + \alpha\,||\vartheta||_2$
Convolutional AutoEncoder
$u_h^0(\mu_1) \dots u_h^0(\mu_{N_{train}})$
$u_h^{N_T}(\mu_1) \dots u_h^{N_T}(\mu_{N_{train}})$
$\tilde{u}_h^0(\mu_1) \dots \tilde{u}_h^0(\mu_{N_{train}})$
$\tilde{u}_h^{N_T}(\mu_1) \dots \tilde{u}_h^{N_T}(\mu_{N_{train}})$
Time stepping loss: $\min_{\theta} \sum_{i=1}^{N_{train}} \sum_{n=\xi}^{N_T} \left|\left| \sum_{k=1}^{\zeta} u_l^{n+k}(\mu_i) - \tilde{u}_l^{n+k}(\mu_i;\theta) \right|\right| + \beta_1\,||\theta||_2 + \beta_2\,||D_u\Psi||_F$
Time stepping
$(\mu_1, \mu_2, \dots, \mu_{N_{train}})$
$u_l^{n-\xi}(\mu_1) \dots u_l^{n-1}(\mu_1)\ u_l^n(\mu_1)$
$\tilde{u}_l^{n+1}(\mu_1)\ \tilde{u}_l^{n+2}(\mu_1) \dots \tilde{u}_l^{n+\zeta}(\mu_1)$
Encoder
Decoder
$u_h^n(\mu)$
$u_h^{n-\xi}(\mu)$
$\tilde{u}_h^{n+1}(\mu)$
$\tilde{u}_h^{n+N_T}(\mu)$
$u_l^n(\mu)$
$u_l^{n-\xi}(\mu)$
μ

(a) Illustration of the offline stage.

(b) Illustration of the online stage.

Fig. 4. (a) Illustration of the offline stage. (b) Illustration of the online stage.

description of the two types. Regarding the CCNNs, there are different ways to include memory. In this paper, we have chosen to include memory in shape of adding more layers, see Fig. 2, for examples for $\xi = 8$, $\xi = 6$, $\xi = 4$, and $\xi = 2$.

4.0.4. *Parameterized dynamics*

We aim to simulate parameterized trajectories of the latent dynamics. Hence, we need to incorporate the parameters as input to the residual computation, resulting in a map

$$\Psi : \mathbb{R}^{\xi N_l + N_P} \rightarrow \mathbb{R}^{\zeta N_l}, \quad u^{n+1}(\mu) = \Psi(u_l^n(\mu), u^{n-1}(\mu), \dots, u_l^{n-\xi}(\mu), \mu). \tag{27}$$

For now, we consider constant parameters, but it should be possible to incorporate time-dependent parameters. For this reason, the parameters do not need to be part of the memory aware section of the network. We propose a parallel architecture consisting of two branches combining into one: One branch interpreting the last ξ states and one branch processing the parameters. The two branches then connect and provide one final prediction for the residual. Having a single neural network incorporating the previous states and the parameters enables simultaneous training of the two branches. See Fig. 3 for an illustration of the network structure. This ensures that the learned latent features from both branches are optimal with respect to predicting the next state. This is in contrast to what is done in [20], where the memory and the parameters are incorporated into two completely separate networks.

For the parameter branch we simply make use of a dense FFNN. There is no immediate reason to believe that more complicated architectures are necessary, since we are neither dealing with time-dependent nor high-dimensional or continuously spatially varying input. Note, however, that there is no reason to believe that this methodology will not work if the FF network in the parameter branch is replaced with a memory aware network in more advanced settings.

The training of the full time-evolution network is done by minimizing the loss function

$$L(u_l, \mu; \theta) = \frac{1}{N_{\text{train}} N_t} \sum_{i=1}^{N_{\text{train}}} \sum_{n=\xi}^{N_T} \left| \left| u^{n+1}(\mu_i) - \Psi(u_l^n(\mu_i), \dots, u_l^{n-\xi}(\mu_i), \mu_i; \theta) \right| \right|_2^2, \tag{28}$$

with respect to the NN parameters θ.

Whereas the individual techniques described may be well-known, we here show how these techniques can be integrated to achieve better performance and accuracy.

4.0.5. *Imposing stability and generalization*

It is well-known that NNs do not necessarily generalize well beyond the training data without some kind of regularization. Combining that with the general risk of having instability in discrete dynamical systems makes it crucial to address these problems during the training.

The arguably most common technique is to add L^1- or L^2-regularization to the loss function. Furthermore, specifically for dynamical systems, it has been shown in [31] and [34] that regularizing the eigenvalues of the Jacobian of the dynamics with respect to the state variable, $D_u\Psi$, does improve long term predictions. In short, this is related to linear and Lyapunov stability analysis of dynamical systems, that are related to sensitivity to initial conditions. Hence, we propose adding the term $||D_u\Psi||_2$, which is the matrix 2-norm, i.e. the spectral radius of the Jacobian of Ψ, to the loss function. In practice, by utilizing the relation

$$||D_u\Psi||_2 \leq ||D_u\Psi||_F, \tag{29}$$

we instead add the computationally much cheaper Frobenius norm.

It can empirically be shown that the long term predictions are significantly better if the network takes several steps at a time instead of a single one. Hence, we modify the output of the NN to

$$R\left(u_l^n(\mu), \dots, u_l^{n-\xi}(\mu), \mu; \theta\right) = [R_1, R_2, \dots, R_\zeta]^T, \tag{30}$$

which gives future predictions,

$$\begin{bmatrix} u_l^{n+1}(\mu) \\ \vdots \\ u_l^{n+\zeta}(\mu) \end{bmatrix} = \begin{bmatrix} \Psi_1\left(u_l^n(\mu), \dots, u_l^{n-\xi}(\mu), \mu; \theta\right) \\ \vdots \\ \Psi_\zeta\left(u_l^n(\mu), \dots, u_l^{n-\xi}(\mu), \mu; \theta\right) \end{bmatrix} = u_l^n(\mu) + \begin{bmatrix} R_1\left(u_l^n(\mu), \dots, u_l^{n-\xi}(\mu), \mu; \theta\right) \\ \vdots \\ R_\zeta\left(u^n(\mu), \dots, u_l^{n-\xi}(\mu), \mu; \theta\right) \end{bmatrix} \tag{31}$$

Empirically we see that this modification keeps the prediction from exploding for longer time and it reduces spurious oscillations.

The resulting loss function for the dynamics NN is given by:

$$\begin{aligned} L(u,\mu;\theta) \; &= \frac{1}{N_{\text{train}} N_t} \sum_{i=1}^{N_{\text{train}}} \sum_{n=\xi}^{N_T} \left|\left| \sum_{k=1}^{\zeta} \left[u_l^{n+k}(\mu_i) - \Psi_k\left(u_l^n(\mu_i), \dots, u_l^{n-\xi}(\mu_i), \mu_i; \theta\right) \right] \right|\right|_2^2 \\ &\quad + \underbrace{\beta_1 ||\theta||_2^2}_{\text{Weight decay}} + \underbrace{\beta_2 ||D_u R||_F}_{\text{Jacobian regularization}} \quad , \end{aligned} \tag{32}$$

4.1. *The complete scheme*

Putting the components together, we have a scheme subdivided into two parts that are trained independently: The CAE, and the time evolution. The whole process is divided into an online phase and an offline phase.

In the offline phase the CAE is trained on a series of high-fidelity snapshots in order to identify a nonlinear reduced trial manifold. Then, the CAE is used to reduce the high-fidelity snapshots to the latent space. The latent space trajectories are used to train the time evolution NN. The training of the two neural networks is visualized in Fig. 4a and outlined in Algorithm 1. Note that in Steps 3 and 5, where the autoencoder and the time evolution network, respectively, are being trained, the considerations mentioned in A have to be included, like early-stopping, multiple-initialization, choice of optimizer, etc. In Algorithm 2, an algorithm to automatically choose the latent dimension, number of training trajectories, memory, and future steps per iteration is presented. Note that this is a basic approach to tune the network. More advanced methods such as Bayesian optimization or reinforcement learning could be utilized here. Furthermore, it is worth noting that we can, assuming no time constraints, generate as many training samples as necessary.

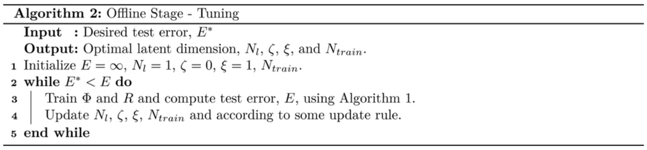
In the online phase the first ξ time steps of the state, computed with a high-fidelity scheme for a given parameter realization μ, are projected onto the latent space using the encoder part of the CAE. From there, the time evolution NN computes the parameterized latent space trajectories iteratively. The latent space trajectories are then transformed to the high-fidelity space using the decoder of the CAE. The online stage is visualized in Fig. 4b and described in pseudo code in Algorithm 3.

Algorithm 1. Offline stage – training

$$\{u_h^0(\mu_1),\dots,u_h^{N_t}(\mu_1),\dots,u_h^0(\mu_{N_\text{train}}),\dots,u_h^{N_t}(\mu_{N_\text{train}})\}$$

```
Input: $N_l$, $\zeta$, $\xi$, $N_{train}$.
Output: Trained CAE, $\Phi$, and Time Evolution Network, $R$, and test error, E.
Sample $N_{train}$ parameter samples from the parameter space.
Generate high-fidelity trajectories,
    $\{u_h^0(\mu_1),\dots,u_h^{N_t}(\mu_1),\dots,u_h^0(\mu_{N_\text{train}}),\dots,u_h^{N_t}(\mu_{N_\text{train}})\}$
Train CAE, $\Phi = \Phi_{dec} \circ \Phi_{enc}$, with latent space dimension $N_l$, by minimizing
    $\underset{\theta}{argmin}\frac{1}{N_\text{train}N_t}\sum_{i=1}^{N_\text{train}}\sum_{n=0}^{N_t}||u_n(\mu_i)-\Phi(u_n(\mu_i);\theta)||^2.$    (33)
Encode high-fidelity trajectories to get latent state space trajectories
    $\{\Phi_\text{enc}(u_h^0(\mu_1)),\dots,\Phi_\text{enc}(u_h^{N_t}(\mu_1)),\dots,\Phi_\text{enc}(u_h^0(\mu_{N_\text{train}})),\dots,\Phi_\text{enc}(u_h^{N_t}(\mu_{N_\text{train}}))\}$
Train time evolution network, $R$, to take the last $\zeta$ states and output the residuals for the next $\xi$ states, by minimizing
    $\frac{1}{N_\text{train}N_t}\sum_{i=1}^{N_\text{train}}\sum_{n=\xi}^{N_T}\left|\left|\sum_{k=1}^{\zeta}[u_l^{n+k}(\mu_i)-\Psi_k(u_l^n(\mu_i),\dots,u_l^{n-\xi}(\mu_i),\mu_i;\theta)]\right|\right|_2^2$
    $+\beta_1||\theta||_2^2+\beta_2||D_uR||_F,$
Estimate error on a test set.
```

Algorithm 2. Offline stage – tuning



```
Algorithm 2: Offline Stage - Tuning
Input  : Desired test error, $E^*$
Output: Optimal latent dimension, $N_l$, $\zeta$, $\xi$, and $N_{train}$.
Initialize $E = \infty$, $N_l = 1$, $\zeta = 0$, $\xi = 1$, $N_{train}$.
while $E^* < E$ do
    Train $\Phi$ and $R$ and compute test error, $E$, using Algorithm 1.
    Update $N_l$, $\zeta$, $\xi$, $N_{train}$ and according to some update rule.
end while
```

Algorithm 3. Online stage

$$(u_l^0(\mu),\dots,u_l^\xi(\mu)) = (\Phi_\text{enc}(u_h^0(\mu)),\dots,\Phi_\text{enc}(u_h^\xi(\mu)))$$

```
Input: $\Phi_{dec}$, $R$, $\mu$, $u_h^0(\mu),\dots,u_h^\xi(\mu)$
Output: Approximated trajectory in high-fidelity space.
Encode the initial $\xi$ high-fidelity states,
    $(u_l^0(\mu),\dots,u_l^\xi(\mu)) = (\Phi_\text{enc}(u_h^0(\mu)),\dots,\Phi_\text{enc}(u_h^\xi(\mu)))$
Compute approximated latent trajectory by iterating,
    $(\widetilde{u}_l^n(\mu),\dots,\widetilde{u}_l^{n+\zeta}(\mu)) = \widetilde{u}_l^n(\mu)+R(\widetilde{u}^n(\mu),\dots,\widetilde{u}^{n-\xi}(\mu),\mu;\theta),$
until desired end time has been reached.
Decode approximated latent space trajectories to high-fidelity space:
    $\{\widetilde{u}_h^0(\mu),\dots,\widetilde{u}_h^{N_t}(\mu)\} = \{\Phi_\text{dec}(\widetilde{u}_l^0(\mu)),\dots,\Phi_\text{dec}(\widetilde{u}_l^{N_t}(\mu))\}$
```

5. Results

The aim of this section is to showcase how well our frameworks perform for different parameterized PDE problems. Furthermore, we show how the various approaches, regularizations, and parameters affect the performance.

To assess the performance measure, the error on N_{test} test trajectories for parameter values, $\{\mu_1,\dots,\mu_{N_\text{test}}\}$, that the NNs have not seen in the training phase is evaluated. We measure the mean relative error (MRE) at every time step and take the mean over multiple runs of the test cases:

$$\text{MRE}(u_h^n(\mu_i),\widetilde{u}_h^n(\mu_i)) = \frac{1}{N_\text{test}}\sum_{i=1}^{N_\text{test}}\frac{||u_h^n(\mu_i)-\widetilde{u}_h^n(\mu_i)||_2^2}{||u_h^n(\mu_i)||_2^2}, \tag{34}$$

where

$$||u_h^n(\mu_i)||_2^2 = (u_h^n(\mu_i))^T u_h^n(\mu_i). \tag{35}$$

Besides the MRE, we also analyze the standard error:

$$\text{Standard Error} = \frac{\sigma}{\sqrt{N_\text{test}}}, \tag{36}$$

where σ is the variance of the MRE. With this measure, we can assess if the trained NN performs similarly on all the test data, i.e. we empirically show robustness and generalization.

For comparison, we show how the error evolves in time using various regression approaches for time stepping, together with the CAE and POD. We do not compare our method to intrusive approaches, such as POD-Galerkin, as we assume here that the PDE model is not available. The approaches dealing with time as a parameter, instead of performing time stepping are inherently different and are therefore also not considered here. Instead, we compare another regression technique, decision tree regression (DTR), as in [14] (see [24] for details on DTR). K-nearest-neighbor regression and Gaussian process regression (GPR) were also tested. K-nearest-neighbor regression performed very similar to DTR and GPR was infeasible to train due to the many training samples needed in the time stepping training. The implementation of DTR was done with the Python package Scikit-learn [35].

Regarding hyperparameters, we considered the number of layers, number of neurons, regularization parameters, learning rate, batch size, and memory. Due to the high-dimensional hyperparameter space, we used Gaussian process minimization a quick and approximate way to tune the hyperparameters.

5.1. Neural network setup

All neural networks are implemented in Tensorflow 2.0 [8] in Python. The training is performed in the Google Colab framework on NVIDIA Tesla P100 GPUs.

The neural network architecture configurations for the CAE can be found in Appendix B

For the CCNN memory encoding, the layers are organized as shown in Figure 2. For the LSTM we work with network architectures of 3–5 layers with 16–64 neurons in each LSTM layer.

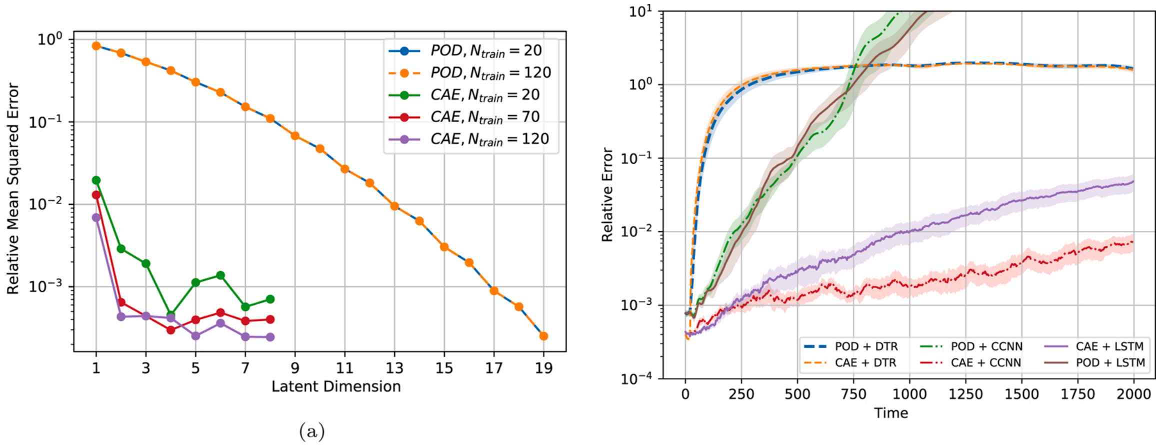
Furthermore, before the LSTM or CCNN layers every previous state is passed through a dense layer with 16 neurons. In TensorFlow 2.0 this type of layer is denoted `TimeDistributed`.

For the parameter encoding, the neural network is a 3 layer deep network with 16 neurons in each layer.

For the final prediction, we utilize a 3–5 layer deep NN with 32 neurons in each layer.

Remark. We only present results on dimensionality reduction using convolutional autoencoders and compare them to POD. It should be noted that dense autoencoders were also tested and showed significantly worse results.

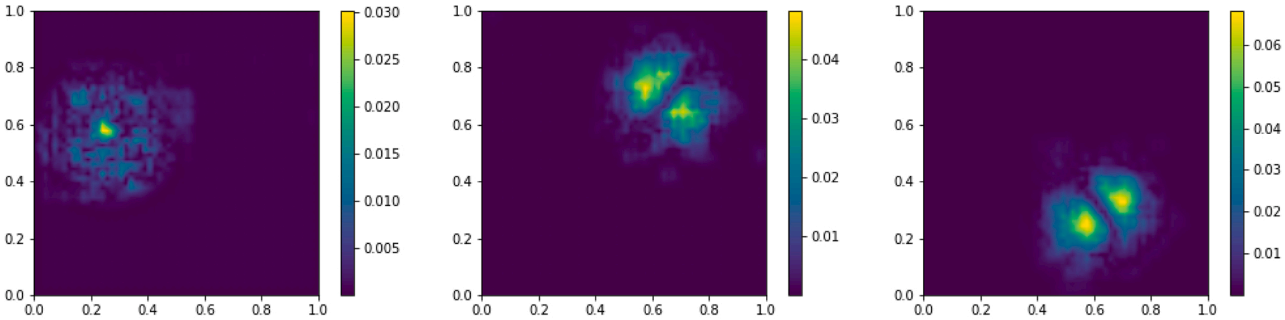
Moreover, we only consider LSTMs, and CCNNs for the time



(a)

(b)

Fig. 5. (a) Comparison of convergence of the time averaged MRE of the reconstruction using CAE and POD for the advection equation. (b) Average test errors computed for the linear advection equation for various combinations of POD, CAE, and regression methods.



(a) $t = 0$

(b) $t = 7.5$

(c) $t = 15$

Fig. 6. Pointwise absolute error between the high-fidelity solution and the neural network prediction (CCNN) for the linear advection equation with velocity, $\mu_1 = 1.4161$, and initial angle, $\mu_2 = 2.8744$.

stepping. We also studied other achitectures, such as neural ODEs [36], gated recurrent units (GRUs), and simple recursive neural networks. However, we chose to not include those results. Neural ODEs performed significantly worse and the training took much longer time. GRUs performed similarly to LSTMs and simple recursive neural networks performed slightly worse.

5.2. Linear advection equation

We first consider a linear advection equation on the domain $\Omega = [0, 1]^2$:

$$\partial_t u(\mu) + b \cdot \nabla u(\mu) = 0, \quad \text{in} \quad \Omega, \tag{37a}$$

$$u(\mu) = 0 \quad \text{on} \quad \Gamma, \tag{37b}$$

where $\Gamma = \partial\Omega$,

$$b = \mu_1 \begin{pmatrix} -y - \dfrac{1}{2} \\ x - \dfrac{1}{2} \end{pmatrix}, \tag{37c}$$

with initial condition

$$u_0(\mu) = exp\left(\frac{1}{2}\left[\frac{(x - x_0)^2}{0.005} + \frac{(y - y_0)^2}{0.005}\right]\right), \tag{38}$$

where

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \frac{1}{4}\begin{pmatrix} cos(\mu_2) \\ sin(\mu_2) \end{pmatrix} + \frac{1}{2}. \tag{39}$$

This problem models a Gaussian curve being advected with velocity μ_1 in a circle with origin at $[\frac{1}{2}, \frac{1}{2}]$ and radius $\frac{1}{4}$, starting at the position given by the angle μ_2. This problem is parameterized by two parameters, $\mu = (\mu_1, \mu_2) \in [0.5, 1.5] \times [0, 2\pi]$. The first parameter, the velocity, is directly affecting the phase of the dynamics, while the other, μ_2, is only dictating the initial placement of the Gaussian curve. Hence, we are dealing with a 2-dimensional parameter space, while the dynamics are only parameterized by a single parameter.

The high-fidelity snapshots are computed on a 60 × 60 grid using the discontinuous Galerkin method with linear Lagrange elements, resulting in a second-order convergence scheme that suits advection dominated problems well. The high-fidelity model consists of 21,600 degrees of freedom. For the implementation we used the FEniCS library in Python [37]. The time stepping is done using the Crank-Nicolson scheme with time steps of size 0.0075 for 2000 steps, resulting in a time interval, $t \in [0, 15]$. The training of the neural networks is done using every 4th time step, $s = 4$, meaning the model is trained to take steps of size 0.03. We are using 15 trajectories for the parameters not included in the training set for testing. The parameter μ is sampled uniformly in the domain $[0.5, 1.5] \times [0, 2\pi]$ for the training data.

In Appendix C, various figures showcasing how each parameter affects the accuracy and stability are presented. From these plots, we infer

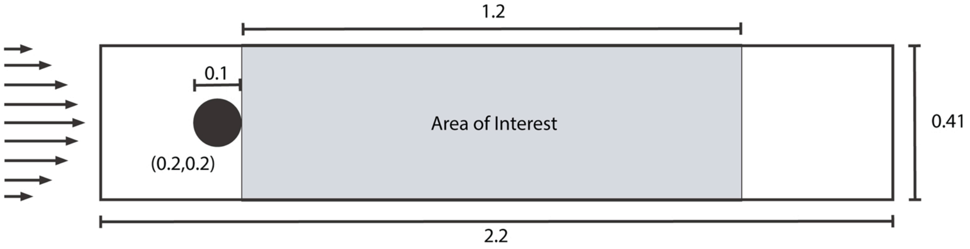


Fig. 7. The setting for the flow past cylinder problem.

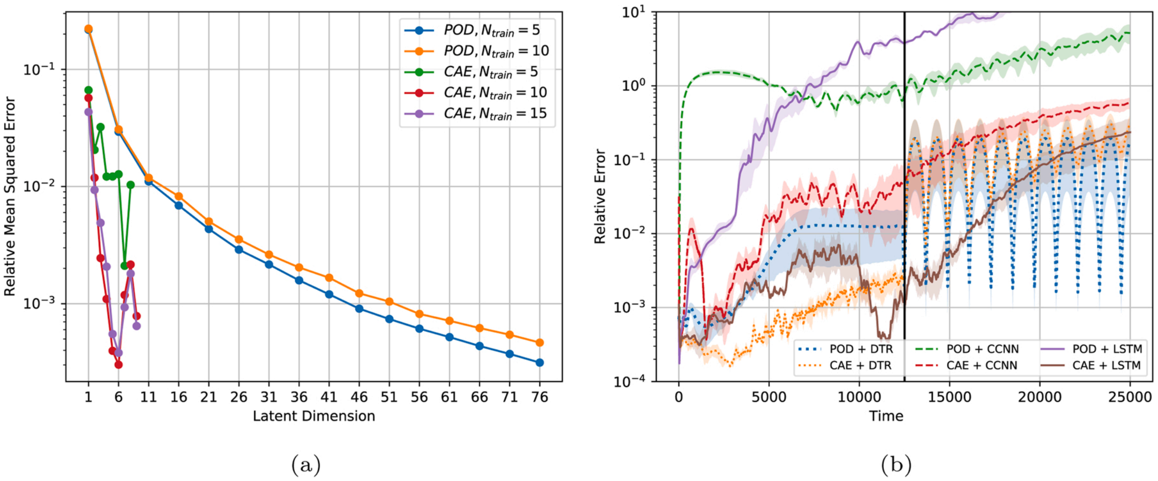


Fig. 10. (a) CAE and POD convergence as well as (b) average errors with standard error for the flow past cylinder problem for various Reynolds numbers computed with various combinations of POD, CAE, and regression methods. The vertical black line signifies the end of the training horizon. The CAE based solutions are computed with a latent dimension of 6 and the POD based with a latent dimension of 76.

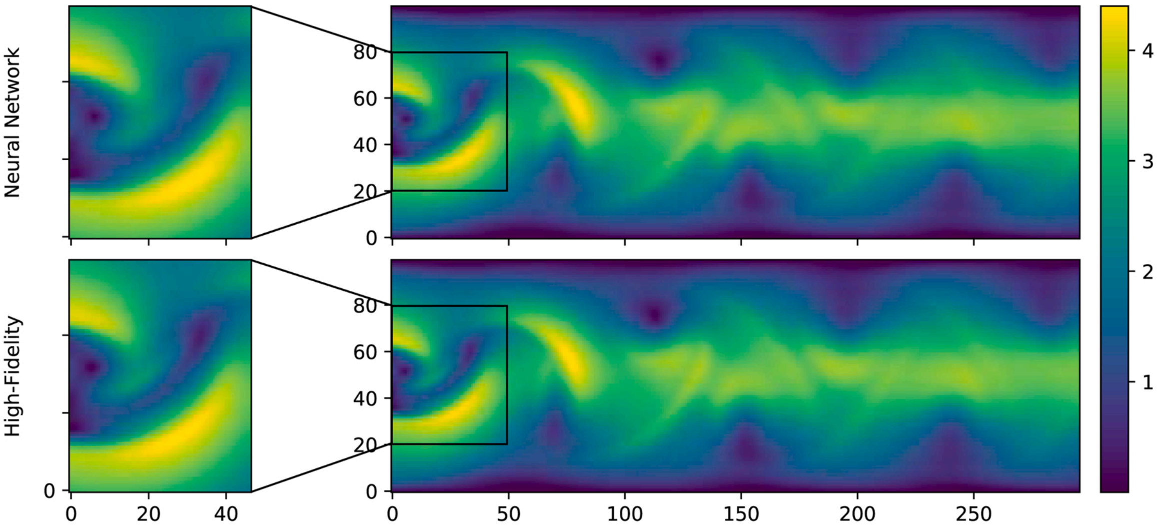
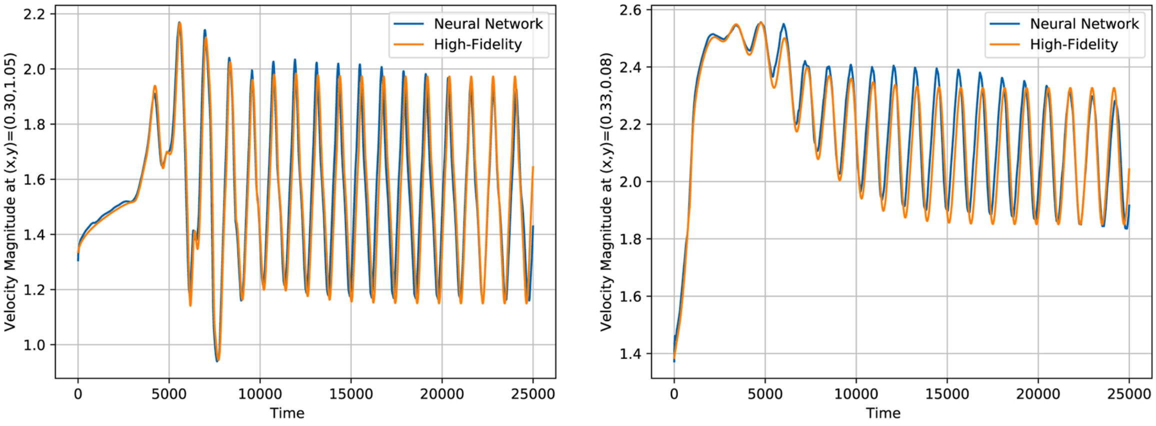


Fig. 8. Comparison of the velocity at $t = 25$ for $Re = 192$ using CAE+LSTM (top) and the high-fidelity method (bottom).

that the NN configuration that performs the best employs $\xi = 6$, $\beta_1 = 10^{-9}$, $\beta_2 = 10^{-6}$, and computes the residual rather than the state directly. Furthermore, the training is performed with 120 training trajectories.

In Fig. 5a, we see a significant improvement by using the CAE compared to the POD approach. Using a latent dimension of 2, which is also the intrinsic dimension of the solution manifold, the CAE reconstructs the high-fidelity solution with an MRE between 10^{-3} and 10^{-4}. To achieve the same accuracy using the POD method, one needs a latent dimension of at least 17. This supports the previous claim that POD does, in general, not perform well on advection dominated problems. Furthermore, it is shown that the accuracy improves with the amount of training data until the point where more data becomes redundant. Specifically, one sees that using 70 trajectories or 120 trajectories is very similar in performance. Note that the POD method does not improve with the amount of training data.

In Fig. 5b we compare the mean error at each time step of our method with the alternative approaches. A latent space of dimension 18 is used



(a) $(x, y) = (1.5, 0.30)$

(b) $(x, y) = (0.08, 0.33)$

Fig. 9. A comparison of the velocity magnitude for the flow past cylinder problem at two distinct points in space for $Re = 192$ using CAE+LSTM.

Table 1
Time averaged error for the CAE+DTR and CAE+LSTM for various number of training trajectories.

N_{train}	20	50	90
CAE+DTR	1.75×10^{-1}	1.41×10^{-1}	6.41×10^{-2}
CAE+LSTM	4.45×10^{-1}	1.11×10^{-1}	3.92×10^{-2}

Table 3
Online computation time in seconds for each of the time stepping regression techniques together with CAE and POD respectively for each test problem.

Test problem	High-fidelity	CAE			POD		
		CCNN	LSTM	DTR	CCNN	LSTM	DTR
Advection equation	532.22	2.08	4.63	0.03	1.78	3.41	0.02
Flow past cylinder	17251.31	7.56	9.74	3.13	5.14	6.57	1.45

Table 4
Offline computation time, i.e. NN training time, in seconds for the CCNN and LSTM using GPUs. Furthermore we show the time it took to generate the training trajectories. Note that the generation of training trajectories is not necessary in cases where the data already exists. Furthermore, we have omitted to show the training time for the non-NN regression methods as the training time negligible.

Test problem	CCNN	LSTM	Generation of trajectories
Advection equation	396.67	622.28	50,763.13 (120 trajectories)
Flow past cylinder	300.39	985.54	931,570.74 (90 trajectories)

for the methods using POD to achieve the same accuracy as for the CAE. All regression approaches are trained with 120 trajectories. Clearly, none of the alternative approaches capture the dynamics accurately. Using the POD basis for the NN time stepping performs significantly worse than the proposed methodology. The time stepping map becomes truly high-dimensional and thereby much harder to approximate.

By looking at the pointwise error between the high-fidelity and the NN solutions in Fig. 6, it is clear that the NN approximation introduces a small phase error. This error could possibly be corrected in a post-processing step.

5.3. 2D nonlinear equation – flow past cylinder

We consider the incompressible Navier–Stokes Equations, governing transitional flow, resulting in a complicated flow pattern. The Equations are given by:

$$\partial_t u(\mathrm{Re}) + (u(\mathrm{Re})\cdot\nabla)u(\mathrm{Re}) - \nabla p(\mathrm{Re}) = \frac{1}{\mathrm{Re}}\Delta u(\mathrm{Re}), \text{in} \quad \Omega, \tag{40a}$$

$$\nabla\cdot u(\mathrm{Re}) = 0, \text{in} \quad \Omega, \tag{40b}$$

$$u(\mathrm{Re}) = 0, \text{on} \quad \Gamma, \tag{40c}$$

$$(u_x(\mathrm{Re}), u_y(\mathrm{Re})) = (1, 0) \quad \text{for} \quad y = 0, \tag{40d}$$

with initial conditions $u(Re) = 0$ and $p(Re) = 0$. Consider a channel with a cylinder with an inflow at the left side and outflow on the right side, see Fig. 7 for a visualization of the setting. In the figure, we have also marked the area of interest, as we are interested in the complex flow pattern in the area immediately behind the cylinder. Note that the present methodology can be employed in a specific subdomain of interest. It is not necessary to work on the whole computational domain, as opposed to most high-fidelity methods.

The inflow profile is given by:

$$u(0, y, t) = \left(1.5\frac{4y(0.41 - y)}{0.41^2}, 0\right) \tag{41}$$

On the walls and the cylinder no slip conditions are prescribed.

We parameterize the problem by the Reynolds number and consider values in the interval $Re \in [120, 200]$. For Reynolds numbers in this interval, the flow exhibits very interesting nonlinear behavior, such as Karman vortex streets. We compute the velocity as well as the pressure

Table 2
Time averaged error for each of the time stepping regression techniques together with CAE and POD respectively for each test problem. The results are for $Re \in [120, 200]$ for the flow past cylinder problem.

	CAE			POD		
Test Problem	CCNN	LSTM	DTR	CCNN	LSTM	DTR
Advection equation	1.54×10^{-3}	5.02×10^{-3}	1.53×10^{0}	2.02×10^{2}	2.84×10^{1}	1.50×10^{0}
Flow past cylinder	1.64×10^{-1}	3.92×10^{-2}	6.41×10^{-2}	–	27.389	5.23×10^{-2}

field, meaning we include all relevant physics in the methodology.

The high-fidelity problem is solved here using the finite element method with Taylor–Hood elements and the second-order incremental pressure correction scheme for time stepping [38], implemented in FEniCS. We use 128 elements resulting in 73,768 degrees of freedom. The solution is then evaluated at a 300 × 100 uniform grid.

For the training trajectories, we solve for $t \in [0, 2.5]$ and for the test trajectories we compute with $t \in [0, 5]$. This means that we also test how our method performs beyond the training horizon. The high-fidelity model stepsize is 0.0002, resulting in 12,500 time steps for the training trajectories and 25,000 steps for the test trajectories. For the training of the reduced model, we only use every 5th step resulting in 2500 time steps, which also means that we use 5000 time steps for the test trajectories.

In Fig. 10a we compare the CAE with the POD method for dimensionality reduction for various number of training trajectories. Using the CAE, one achieves accuracy of approximately 10^{-4} with a latent dimension of 6. To achieve the same accuracy with POD one needs a latent space of dimension 76. Furthermore, it is clear that the CAE performs better with more training data until a certain point. However, it is apparent that the error increases when the latent dimension is increased. This phenomena is a result of overfitting or insufficient training, such as convergence to a local minimum.

As for the advection equation, we compare the accuracy with DTR. In Fig. 10b we see that CAE+DTR performs well within the interval of the training, but fails to give anything meaningful beyond it. On the other side, CAE+LSTM performs significantly better beyond the training horizon. The CAE+CCNN performs consistently one error of magnitude worse than CAE+LSTM.

In Fig. 8 we see the velocity magnitude at $t = 25$ in the area of interest. Visually, there is close to no difference between the two, suggesting that the CAE+LSTM approach is able to capture the complicated flow patterns beyond the training time interval. This is further shown in Fig. 9 where we see the velocity magnitude at two specific points in space. It is clear that there is a small dispersion error as well as small errors in magnitude. However, the general flow pattern is approximated well.

Lastly, we compare how the number of training trajectories affect the accuracy for the CAE+DTR and CAE+LSTM in Table 1. As expected, the test error decreases with the number of training trajectories. It is further apparent that the CAE+LSTM approach decreases faster, suggesting that this method benefits, to a higher degree, from more data.

Remark – larger reynolds number intervals. Above, we considered the case with the Reynolds numbers $Re \in [120, 200]$. For larger ranges of Reynolds numbers the flow in the wake of the cylinder varies more. Specifically, the flow regimes are known to have the following characteristics [39]: $0 < Re < 5$: Steady without a wake; $4 < Re < 40$: Steady symmetric separation; $30 < Re < 90$: Laminar unstable wake; $80 < Re < 300$: Von Karman vortex street; $150 < Re < 1.3 \cdot 10^5$: Vortex street with (turbulent) instabilities. Note that the intervals are overlapping as the exact boundaries between two regimes are unclear.

The neural networks need to have a greater approximation ability in order to capture the dynamics in such different flow regimes. This would require a very large neural network with a large training data set. This can be circumvented by using, instead of a single CAE plus time stepping NN, a conditional approach. By dividing the parameter space into N_I intervals and constructing N_I CAEs with time stepping NNs to be trained on each interval, we made the data fitting an easier task. The offline procedure then involved training N_I CAEs and time stepping NNs and the online procedure included an initial step determining the regime in which the given parameters lie.

The division of the parameter space can be done by utilizing knowledge of the bifurcation diagram. Here, we saw that the NNs are not sensitive to the specific choice of intervals. In this experiment, we divided the parameter space into the following three intervals: $I_1 = [1, 10]$, $I_2 = [10, 65]$, and $I_3 = [65, 120]$. Hence, we tested whether the NNs could approximate the flow across different regimes.

We chose to use the same hyperparameter setting for all N_I networks. The CAEs were trained on 10 trajectories for I_1 and 20 trajectories for I_2 and I_3. The time stepping NNs were trained on 13, 72, and 77 trajectories for I_1, I_2, and I_3, respectively. Testing of the NNs in each interval was done with three trajectories with Reynolds numbers uniformly distributed in the relevant interval.

We compared results for CAE and LSTM with results computed with CAE and DTR. Regarding the convergence of the CAEs for the three Reynolds number intervals, a latent dimension of 4 showed satisfactory results. We found essentially the same results for the three flow regimes, as those presented in the previous subsection for I_4. The CAEs significantly outperfomed POD and LSTMs resulted in a better time stepping scheme that could approximate the flow beyond the training horizon while DTRs failed.

In conclusion, dividing the parameter space into intervals and train NNs in each regime was a feasible solution to the problem of large parameter intervals.

5.4. Computation time and accuracy

We have showed and discussed performance regarding relative error for the two test cases. The results for the two cases, using the CCNN, LSTM, and DTR combined with CAE and POD, are summarized in Table 2, where the time averaged error is shown. The results for the for the flow past cylinder case are computed for I_4.We considered the case with the Reynolds numbers

As mentioned in the introduction, the aim is to be able to compute solutions fast in the online stage. In Table 3 the high-fidelity as well as the NN, and DTR online times in seconds are shown. In the online stage there has not been used any form of parallelization. Therefore, it should be noted that significant speed ups for both the high-fidelity and the regression time stepping approaches could be achieved with a greater effort on this matter. The NN online time and the high-fidelity computation time is computed on an AMD Ryzen 9 3950X CPU. We observe that DTR is computationally faster, due to the much simpler model. However, it was shown above, that DTR was not able to approximate the dynamics well.

Comparing the CAE+LSTM computation with the high-fidelity computation time we see speed-ups of around 115 times for the advection equation and 1770 times for the flow past cylinder test case. Hence, we see significant speed-ups.

In Table 4 the offline time is shown, divided into NN training time and the time it took to generate the training trajectories. In cases where the training trajectories come from collected data the simulation step is unnecessary, and hence the training time alone is the relevant number. For the training we used an Nvidia GeForce RTX 3090 GPU. Compared to the online stage it makes a massive difference to use a GPU instead of a CPU due to the heavy computations associated with backpropagation. We have chosen to only show the GPU training time. It is clear that the most time consuming part is generating the training trajectories.

6. Conclusion

We presented a novel deep learning approach to non-intrusive reduced order modeling for parameterized time-dependent PDEs using CAEs for dimensionality reduction and CCNNs and LSTMs combined with FFNNs for time evolution. This approach was demonstrated on two test cases and was shown to perform well in the online phase, showcasing the potential of using deep learning based ROMs for different physical phenomena.

Regarding dimensionality reduction, a discussion and comparison of linear and nonlinear methods was presented with POD and CAEs as the focal points. The discussion focused on why a nonlinear approach has the potential to outperform a linear approach.

For time stepping, the general idea was to encode the previous states and the parameters separately in parallel and then combine the encoded data to make a final prediction using an FFNN. The two encoding NNs, as well as the final prediction NN, constitute a single network, meaning everything is trained simultaneously. This ensures that both the memory and parameters are encoded in relation to one another. Furthermore, various methods to ensure generalization, stability, and precision were discussed and tested.

For the advection equation the CAE+CCNN approach performed very well with errors below 10^{-2} for all time steps while the CAE+LSTM performed similarly but slightly worse. Interestingly, the alternative approaches, using POD instead of CAE and DTR for time stepping, failed to approximate the dynamics in any meaningful way. Furthermore, as expected the CAE reached much better precision than POD for dimensionality reduction with much fewer dimensions in the latent space.

Secondly, a more involved problem, flow past a cylinder, was also studied. Here, we were dealing with multiple vector fields and complicated nonlinear patterns. Furthermore, we tested how the methodology performs beyond the training horizon. We saw that the CAE+LSTM approach showed errors below 10^{-2} within the training horizon, and a slow increase in error beyond the horizon. However, the increase in error is primarily due to small phase errors, meaning the overall structure of the flow still resembles the high-fidelity flow. When using DTR, either with POD or the CAE, the approximations completely failed beyond the training horizon, suggesting more complicated models are needed to actually learn the time stepping map.

Lastly, we discussed an approach to deal with large parameter intervals that give rise to highly varying flow regimes. Namely, training a CAE and time stepping NN on subdomains and use the NNs corresponding to the subdomain it was trained on. We conclude that for the flow past cylinder case this approach is successful.

In summary, the contributions in this work include a nonlinear dimensionality reduction scheme using convolutional autoencoders, a novel parallel neural network architecture for parameterized time stepping using CCNNs and LSTMs, and a discussion on different approaches to achieve stability and generalization for neural network-based time stepping. It is furthermore worth mentioning that framework presented allows for flexibility in shape of replacing certain elements with alternatives. E.g. one could replace the CCNN or LSTM with another choice if needed.

In the future the methodology will be tested on more advanced PDE problems. By advanced problems, we are both referring to increasing nonlinearity, higher dimensions, and multi-query problems such as uncertainty quantification, model predictive control, and data assimilation. Especially, data assimilation seems like a promising direction, since incorporating data could rectify the phase errors.

Besides considering other use cases one could work on improving the NN architecture and training by, e.g. incorporating the physics in the training [31,39], and use reinforcement learning [40] to ensure effective snapshot generation. Furthermore, with the amount of hyperparameters (ξ, β_1, β_2, number of layers and neurons, etc.) the task of hyperparameter tuning is not trivial and could potentially be solved more effectively using alternative approaches.

Conflict of interest

None declared.

Declaration of Competing Interest

The authors report no declarations of interest.

Acknowledgements

This work is supported by the Dutch National Science Foundation NWO under the grant number 629.002.213, which is a cooperative projects with IISC Bangalore and Shell Research as project partners. The authors furthermore acknowledge fruitful discussions with Dr. B. Sanderse. Lastly, the authors thank Peter Piontek for proofreading.

Appendix A. Artificial neural networks

A.1 Feedforward neural networks

The arguably most common ANN architecture is the feedforward neural network (FNN). An FNN can be considered a function, $G : \mathbb{R}^{N_i} \rightarrow \mathbb{R}^{N_o}$, consisting of a series of affine transformations, T_i, followed by an element-wise (nonlinear) activation function, σ_i:

$$G(x;\theta) = \sigma_L \circ T_L \circ \cdots \circ \sigma_1 \circ T_1(x). \tag{A.1}$$

The combination of an afine transformation followed by the activation is called a neuron. The afine transformation can be written as $T_i(x) = W_i x + b_i$, where $W \in \mathbb{R}^{M_i \times M_{i-1}}$ and $b \in \mathbb{R}^{M_i}$. We call W_i the weight matrix, b_i the bias vector, and M_i the number of neurons in layer i, and L the number of layers. (A.1) is conveniently visualized as a network of neurons. We will refer to the set of parameters as $\theta = \{W_1, b_1, \ldots, W_L, b_L\}$.

In supervised learning one tries to approximate a function by an ANN, typically done by minimizing the empirical risk w.r.t. the parameters θ:

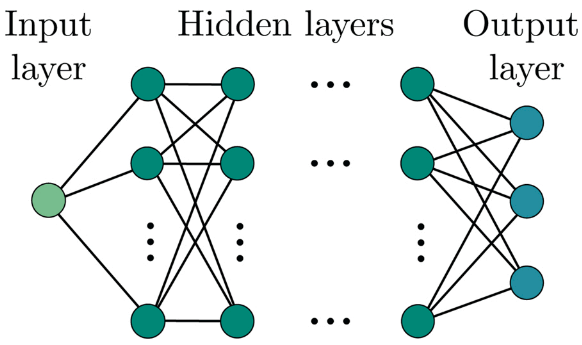


Fig. A.11. Visualization of a feedforward densely connected neural network.

$$\theta^* = \underset{\theta}{\operatorname{arg}min}\mathbb{E}_{(x,y)\sim\mathcal{P}_{\text{data}}}[\mathcal{L}(G(x;\theta),y)], \tag{A.2}$$

where $\mathcal{P}_{\text{data}}$ is the distribution generating the data and $\mathcal{L}$ is a chosen loss function measuring the discrepancy between the predicted output and the target. For regression type problems the mean squared error (MSE) is the most common choice. However, especially for physics-informed machine learning the physics is often incorporated in the loss function in shape of extra term [39,31].

Computing (A.2), i.e. training the ANN, is mostly done using stochastic gradient descent (SGD) or a variants such as ADAM [41].

A.2 Convolutional neural networks

Convolutional neural networks (CNNs) gained attention due their great performance in image recognition. The general idea is to utilize local properties of the data instead of only considering global properties. This is done by having local connections and shared weights in the neural networks. These properties are not only great for detecting patterns in data but it also makes it possible to do computations on very high-dimensional data.

A convolutional layer is effectively a feature map where each unit in the layer is connected to a local patch of the previous layer through a filter bank and an activation function. A feature map at layer l is a tensor, $H^l \in \mathbb{R}^{N^l_{\text{chan}}\times N^l_1\times N^l_2}$, where $H^l_{i,j,k}$ is a unit at channel i, row j, and column k. The filter bank at layer l is a 4-dimensional tensor, $F^l \in \mathbb{R}^{N^l_{\text{filter}}\times N^{l-1}_{\text{chan}}\times k_1\times k_2}$, where $F^l_{i,j,m,n}$ connects a unit in channel i of the output and channel j of the input with m and n being the offset of rows and columns respectively. N^l_{filter} denotes the number of filters in the feature bank in layer l and k_1 and k_2 denotes the kernel size. The convolution operation between a feature map and a filter bank is given by

$$H^l_{i,j,k} = \sigma_l\left(\sum_{r=1}^{N^{l-1}_{\text{chan}}}\sum_{m=1}^{N^{l-1}_1}\sum_{n=1}^{N^{l-1}_2} H^{l-1}_{r,(j-1)s+m,(k-1)s+n}F^l_{i,r,m,k} + B^l_{i,j,k}\right), \tag{A.3}$$

where $B^l_{i,j,k}$ is a bias term and σ_l is an activation function applied element-wise. s denotes the stride and effectively downsamples the feature map between layers. The filters, $F^l_{i,r,m,k}$, and biases, $B^l_{i,j,k}$, are the learnable parameters while the kernel sizes, k_1, k_2, the stride, s, and the number of filters, N^l_{filter}, are chosen. Often these are subject case specific objectives or hyperparameter optimization.

A.3 Causal convolutional neural networks

As the name suggests, causal convolutional neural networks (CCNNS) are related to convolutional neural networks. CCNNs are sometimes referred to as temporal convolutional neural networks, but in this paper we use the term CCNN.

CCNNs are used for encoding time series data with the purpose of forecasting or classification. The general idea is to use 1-dimensional convolutions on time series data. In the multivariate case the multiple dimensions are interpreted as channels. The term causal refers to the fact that the filter banks are only convolved with the current and previous time steps, thus establishing a causal relationship between the past the future.

A.4 Recurrent neural networks and long short-term memory

A recurrent neural network (RNN) is an alternative to CCNNs for interpreting time series data. The general idea is to process and retain information from previous time steps in an efficient manner. In this paper, we solely focus on a specific RNN called long short-term memory (LSTM) [21]. For an input consisting of several previous time steps, x^n, an LSTM layer consists of four components [19]: An input gate:

$$i^{n+1} = \sigma(W_i x^n + b_i), \tag{A.4}$$

a forget gate:

$$f^{n+1} = \sigma\big(W_f x^n + b_f\big), \tag{A.5}$$

an output gate:

$$o^{n+1} = \sigma(W_o x^n + b_o), \tag{A.6}$$

and a cell state

$$c^{n+1} = i \odot c^n + i^n \odot tanh(W_c x^n + b_c) \tag{A.7}$$

The prediction is then given by

$$x^{n+1} = o^n \odot tanh(c^n). \tag{A.8}$$

W_i, b_i, W_f, b_f, W_o, b_o, W_c, b_c are the trainable weight matrices and bias vectors, and $\odot$ is the Hadamard product. Ideally, the input gate identifies what information to be passed to the from the cell state, the forget what to be dropped, and the output gate decides what to be passed to the final prediction.

Appendix B. Convolutional autoencoder configurations

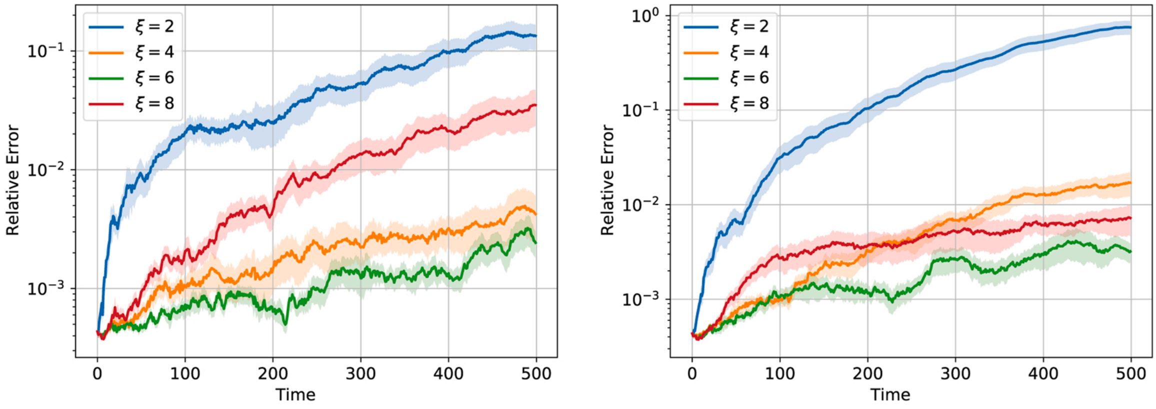
Table B.5
Convolutional autoencoder configuration for the advection equation. Note that the decoder is the inverse of the encoder.

Encoder			
Convolutional Layers	Kernels	Filter size	Stride
Convolutional layer 1	4	5 × 5	2 × 2
Batchnormalization 1			
Convolutional layer 2	8	5 × 5	2 × 2
Batchnormalization 2			
Convolutional layer 3	16	5 × 5	2 × 2
Batchnormalization 3			
Convolutional layer 4	32	5 × 5	2 × 2
Batchnormalization 4			
Flatten			
Dense layers			Neurons
Dense layer 1			N_l

Table B.6
Convolutional autoencoder configuration for the flow past cylinder problem. Note that the decoder is the inverse of the encoder.

Encoder			
Convolutional layers	Kernels	Filter size	Stride
Convolutional layer 1	8	5 × 5	2 × 2
Batchnormalization 1			
Convolutional layer 2	16	5 × 5	2 × 2
Batchnormalization 2			
Convolutional layer 3	32	5 × 5	2 × 2
Batchnormalization 3			
Convolutional layer 4	64	5 × 5	2 × 2
Batchnormalization 4			
Convolutional layer 5	128	5 × 5	2 × 2
Batchnormalization 5			
Flatten			
Dense layers			Neurons
Dense layer 1			493
Dense layer 2			247
Dense layer 3			N_l

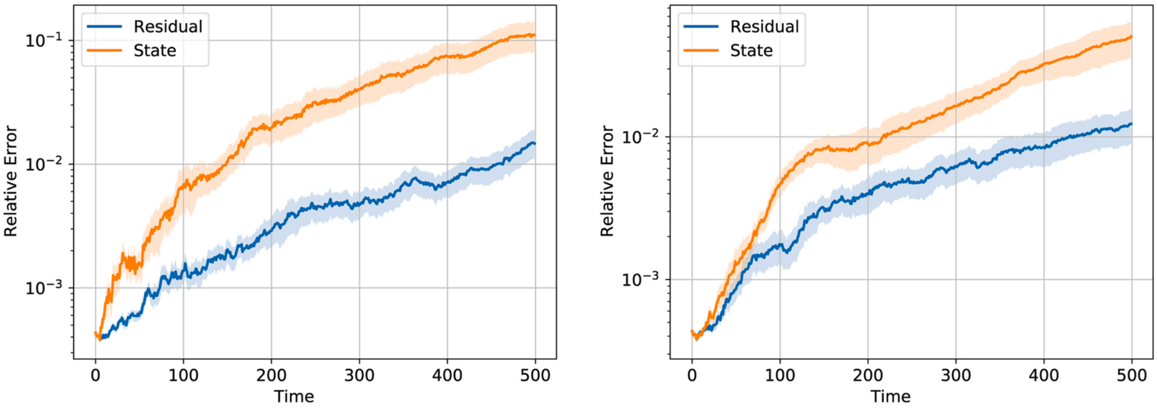
Appendix C. Linear advection



(a) CCNN.

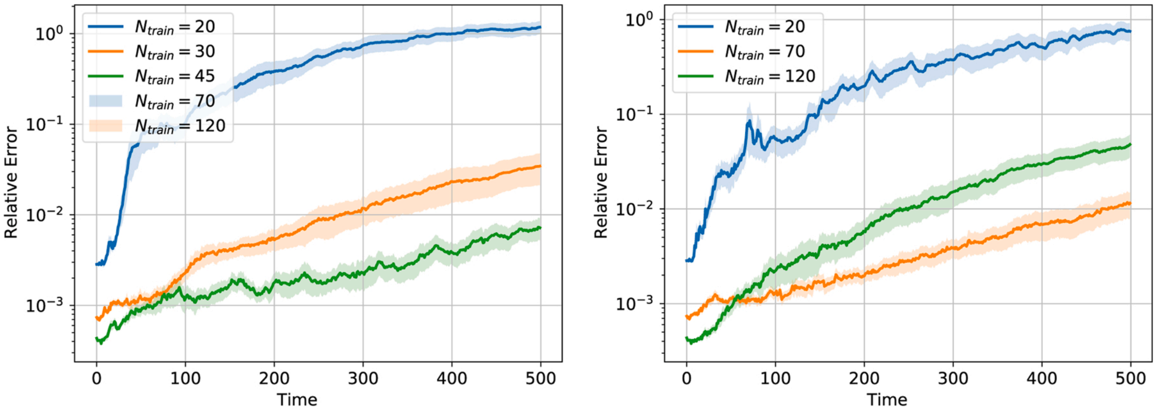
(b) LSTM.

Fig. C.12. Comparison of CCNN and LSTM in relative error for each time step in high-fidelity space for the linear advection equation for varying memory, ξ. The error for each time step is a computed average over 15 test cases with the standard error.



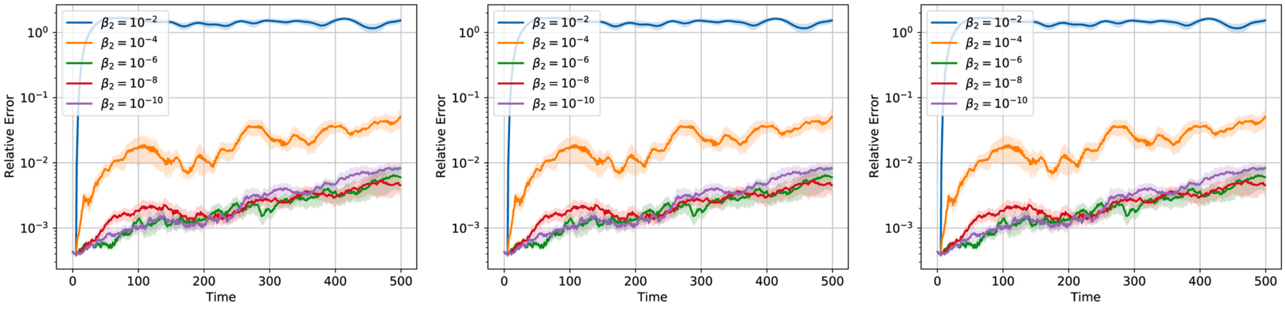
(a) CCNN. (b) LSTM.

Fig. C.13. Comparison of computing the next step directly and the residual for the CCNN and LSTM. The figures show relative error for each time step in high-fidelity space for the linear advection equation.



(a) CCNN. (b) LSTM.

Fig. C.14. Comparison of CCNN and LSTM in relative error for each time step in high-fidelity space for the linear advection equation for number of training samples, N_{train}.



(a) $\beta_1 = 10^{-3}$. (b) $\beta_1 = 10^{-6}$. (c) $\beta_1 = 10^{-9}$.

Fig. C.15. Impact of the two regularization terms, weight decay, β_1, and Jacobian, β_2 for the linear advection equation. The average relative error in high-fidelity space over 15 test trajectories for each time step is shown. Each figure shows the error for a constant β_1 and varying β_2.

References

[1] G. Hager, G. Wellein, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2010.

[2] D.A. Kopriva, Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers, Springer Science & Business Media, 2009.

[3] Y. Saad, Iterative Methods for Sparse Linear Systems, vol. 82, SIAM, 2003.

[4] U. Trottenberg, C.W. Oosterlee, A. Schuller, Multigrid, Elsevier, 2000.

[5] A. Quarteroni, A. Manzoni, F. Negri, Reduced Basis Methods for Partial Differential Equations: An Introduction, vol. 92, Springer, 2015.

[6] J.S. Hesthaven, G. Rozza, B. Stamm, et al., Certified Reduced Basis Methods for Parametrized Partial Differential Equations, vol. 590, Springer, 2016.

[7] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch, 2017.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: a system for large-scale machine learning,

12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16) (2016) 265–283.

[9] S. Lee, N. Baker, Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence, Tech. Rep., USDOE Office of Science (SC), United States, 2018.

[10] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, et al., Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence, Tech. Rep., USDOE Office of Science (SC), Washington, DC, United States, 2019.

[11] S. Brunton, B. Noack, P. Koumoutsakos, Machine Learning for Fluid Mechanics, 2019. arXiv:1905.11075.

[12] M. Guo, J.S. Hesthaven, Reduced order modeling for nonlinear structural analysis using Gaussian process regression, Comput. Methods Appl. Mech. Eng. 341 (2018) 807–826.

[13] J.S. Hesthaven, S. Ubbiali, Non-intrusive reduced order modeling of nonlinear problems using neural networks, J. Comput. Phys. 363 (2018) 55–78.

[14] R. Swischuk, L. Mainini, B. Peherstorfer, K. Willcox, Projection-based model reduction: formulations for physics-based machine learning, Comput. Fluids 179 (2019) 704–717.

[15] Q. Wang, J.S. Hesthaven, D. Ray, Non-intrusive reduced order modeling of unsteady flows using artificial neural networks with application to a combustion problem, J. Comput. Phys. 384 (2019) 289–307.

[16] C. Audouze, F. De Vuyst, P.B. Nair, Nonintrusive reduced-order modeling of parametrized time-dependent partial differential equations, Numer. Methods Partial Diff. Equ. 29 (5) (2013) 1587–1628.

[17] N.T. Mücke, L.H. Christiansen, A.P. Karup-Engsig, J.B. Jørgensen, Reduced Order Modeling for Nonlinear PDE-Constrained Optimization Using Neural Networks, 2019. arXiv:1904.06965.

[18] K. Lee, K.T. Carlberg, Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders, J. Comput. Phys. 404 (2020) 108973.

[19] F.J. Gonzalez, M. Balajewicz, Deep Convolutional Recurrent Autoencoders for Learning Low-Dimensional Feature Dynamics of Fluid Systems, 2018. arXiv: 1808.01346.

[20] J. Xu, K. Duraisamy, Multi-Level Convolutional Autoencoder Networks for Parametric Prediction of Spatio-Temporal Dynamics, 2019. arXiv:1912.11114.

[21] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.

[22] A.v.d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuoglu, Wavenet: A Generative Model for Raw Audio, 2016. arXiv:1609.03499.

[23] A. Quarteroni, S. Quarteroni, Numerical Models for Differential Problems, vol. 2, Springer, 2009.

[24] J. Friedman, T. Hastie, R. Tibshirani, The Elements of Statistical Learning, vol. 1, Springer Series in statistics New York, 2001.

[25] S. Pawar, S. Rahman, H. Vaddireddy, O. San, A. Rasheed, P. Vedula, A deep learning enabler for nonintrusive reduced order modeling of fluid flows, Phys. Fluids 31 (8) (2019) 085101.

[26] R. Vidal, Y. Ma, S. Sastry, Generalized principal component analysis (gpca), IEEE Trans. Pattern Anal. Mach. Intell. 27 (12) (2005) 1945–1959.

[27] S. Yu, J.C. Principe, Understanding autoencoders with information theoretic concepts, Neural Netw. 117 (2019) 104–123.

[28] P. Petersen, F. Voigtlaender, Equivalence of approximation by convolutional neural networks and fully-connected networks, Proc. Am. Math. Soc. 148 (4) (2020) 1567–1581.

[29] L. Ruthotto, E. Haber, Deep neural networks motivated by partial differential equations, J. Math. Imaging Vision (2019) 1–13.

[30] S. Fresca, L. Dede, A. Manzoni, A Comprehensive Deep Learning-Based Approach to Reduced Order Modeling of Nonlinear Time-Dependent Parametrized PDEs, 2020. arXiv:2001.04001.

[31] N.B. Erichson, M. Muehlebach, M.W. Mahoney, Physics-Informed Autoencoders for Lyapunov-Stable Fluid Flow Prediction, 2019. arXiv:1905.10866.

[32] R.J. LeVeque, Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems, vol. 98, SIAM, 2007.

[33] C. Gin, B. Lusch, S.L. Brunton, J.N. Kutz, Deep Learning Models for Global Coordinate Transformations That Linearize PDEs, 2019. arXiv:1911.02710.

[34] S. Pan, K. Duraisamy, Long-time predictive modeling of nonlinear dynamical systems using neural networks, Complexity 2018 (2018).

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[36] T.Q. Chen, Y. Rubanova, J. Bettencourt, D.K. Duvenaud, Neural ordinary differential equations, Advances in Neural Information Processing Systems (2018) 6571–6583.

[37] A. Logg, K.-A. Mardal, G. Wells, Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book, vol. 84, Springer Science & Business Media, 2012.

[38] K. Goda, A multistep technique with implicit difference schemes for calculating two-or three-dimensional cavity flows, J. Comput. Phys. 30 (1) (1979) 76–95.

[39] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics Informed Deep Learning (Part I): Data-Driven Solutions of Nonlinear Partial Differential Equations, 2017. arXiv: 1711.10561.

[40] M. Sugiyama, Statistical Reinforcement Learning: Modern Machine Learning Approaches, CRC Press, 2015.

[41] D.P. Kingma, J. Ba, Adam: A Method for Stochastic Optimization, 2014. arXiv: 1412.6980.

Nikolaj T. Mücke is a Ph.D. candidate at CWI in Amsterdam. He has masters' degree in Mathematical Modeling and Computation from the Technical University of Denmark (DTU). His research deals with fluid mechanics, reduced order modeling, and physics informed machine learning.

Sander M. Bohté is a senior researcher and PI in the CWI Machine Learning group, and also a full professor of Cognitive Computational Neuroscience at the University of Amsterdam and a full professor of Bio-Inspired Neural Networks at the Rijksuniversiteit Groningen, The Netherlands. His research bridges the field of neuroscience with applications thereof as advanced neural networks. His work has been pioneering in the development of advanced and efficient spiking neural networks. Recent work has also developed biologically plausible deep learning and deep reinforcement learning models for cognition, causal convolutional neural networks, and efficient spiking recurrent neural networks.

Cornelis Oosterlee is a senior scientist at the CWI, Center for Mathematics & Computer Science, in Amsterdam, and a full professor at the Delft University of Technology, in applied Mathematics. He has written two textbooks, and approximately 150 journal publications. His expertise includes numerical methods, computational mathematics, iterative solution methods, physics inspired neural networks, computational finance and classical engineering applications.