POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Physics-informed machine learning methods for reduced-order modeling

Tesi di Laurea Magistrale in
Ingegneria Aeronautica

Author: **Riccardo Tomada**

Student ID: 966532
Advisor: Prof. Edie Miglio
Academic Year: 2021-22

To my grandmother Irena
Dedykuje mojej babci Irene

Abstract

The need for a fast and accurate solution of parameterized Partial Differential Equations (PDEs) traditionally translates into the development of linear Reduced Order Models (ROMs). These data-driven approaches, such as the Proper Orthogonal Decomposition (POD), are nowadays widespread and embedded in the most famous linear-algebra libraries of any scientific programming language. While their accuracy is in general satisfactory when dealing with linear and steady parameterized PDEs, both their performances and dimensionality reduction capabilities decay in the case of time-dependent and possibly non-linear problems.

To mitigate this issue, novel approaches have been devised in the Deep Learning context. In this work, two completely different strategies are presented. The first one consists of the computation of the parameterized solution in one step by means of Physics Informed Neural Networks (PINNs), which are trained to minimize the PDE residual while satisfying the boundary and initial conditions, for any instance of the parameter inside its domain. The main advantage of this strategy, named PINNs-FOM, compared with traditional PDE solvers lies in the fact that they are completely mesh-free algorithms. On the other hand, compared to classical ROMs, PINNs are of particular interest because they do not require the generation of high-fidelity solutions, called *snapshots*, via computationally expensive Full Order Models (FOMs) solvers.

The second idea consists of the development of a convolutional-autoencoder based non-linear ROM. Despite this approach, named PDNNs-Autoencoder (Projection Driven Neural Network-Autoencoder) still relies on the projection of costly snapshots on a lower dimensional manifold, the feature extractor capabilities of the autoencoder are superior to the linear ROM ones, leading to an improvement in accuracy, especially in the above mentioned cases in which linear ROMs are less effective.

The accuracy and efficiency of these two strategies will be assessed on 4 different parameterized PDEs test problems. The results will be compared with linear ROM based approaches, which in turn rely on Artificial Neural Networks too for the fast online prediction of the solution.

Keywords: ROM, PINN, Surrogate models, Autoencoder

Abstract in lingua italiana

La necessità di calcolare in maniera rapida e accurata la soluzione di equazioni alle derivate parziali (PDEs) parametrizzate si traduce tradizionalmente nello sviluppo di modelli di ordine ridotto di natura lineare. Questi approcci *data-driven*, come la Proper Orthogonal Decomposition (POD), sono oggigiorno diffusi e già implementati all'interno delle più famose librerie di algebra lineare per qualsiasi linguaggio di programmazione scientifico. Mentre il grado di accuratezza che in genere essi raggiungono quando applicati a PDEs di tipo lineare e stazionario è soddisfacente, sia le loro prestazioni che la loro capacità di ridurre la dimensionalità del problema calano una volta che lo stesso sia instazionario o eventualmente non lineare.

Per rispondere all'insorgere di questa problematica, nel campo del Deep Learning sono stati recentemente sviluppati nuovi approcci. Nel presente lavoro vengono presentate due strategie completamente differenti. La prima consiste nel calcolo della soluzione parametrizzata in un unico step mediante l'impiego di Physics Informed Neural Networks (PINNs), le quali vengono allenate allo scopo di minimizzare il residuo della PDE soddisfacendo allo stesso tempo le condizioni al contorno (ed eventualmente iniziali) prescritte, per ogni valore del parametro all'interno del suo dominio. Il principale vantaggio di questa strategia, chiamata PINNs-FOM, rispetto ai solutori di PDEs tradizionali risiede nel fatto che le PINNs sono degli algoritmi completamente mesh-free. Dall'altro lato, comparate con i classici ROMs, le PINNs sono di particolare interesse in quanto non richiedono la generazione di soluzioni high-fidelity, chiamate *snapshots*, particolarmente costose dal punto di vista computazionale.

La seconda idea consiste nello sviluppo di un ROM non lineare basato sull'implementazione di un convolutional autoencoder. Sebbene questo approccio, di seguito chiamato PDNNs-Autoencoder (Projection Driven Neural Network-Autoencoder), si basi ancora sulla proiezione di costosi snapshots sullo spazio in cui cercare le soluzioni candidate, la capacità di questo metodo di estrarre le informazioni più importanti dai dati è superiore rispetto ai ROM di tipo lineare. Per questo motivo i PDNNs-Autoencoder offrono un'accuratezza superiore, nei casi sopracitati in cui i ROM lineari perdono efficacia.

L'accuratezza e l'efficienza di queste due strategie verranno valutate su 4 differenti casi test. I risultati ottenuti saranno confrontati con quelli derivanti da ROM di tipo lineare, ma che impiegano anch'essi reti neurali artificiali per la predizione rapida della soluzione in tempo reale.

Parole chiave: ROM, PINN, Surrogate models, Autoencoder

Contents

Introduction

Physical phenomena can be generally modelled through PDEs derived from natural laws. These PDEs admit a solution in the closed form only in very simple cases. For this reason, a wide number of numerical schemes has been developed starting from the beginning of the last century, such as the Finite Differences Method (FDM) [28], the Finite Elements Method (FEM) [35] and the Finite Volume Method (FVM) [39]. Each of these methods is characterized by its own strengths and weaknesses. To date there is not a numerical method superior to all the others, in fact the choice is driven by their specific peculiarities and how they interact with the problem at hand.

The discretization in time and space leads to the so called Full Order Models (FOMs), which, regardless of the selected numerical method, are in general high-dimensional systems. While these methods are able to achieve high accuracy on a vast set of problems and there is a rigorous mathematical literature supporting them, the high-dimensionality of the FOMs becomes soon an issue as the PDEs are parameterized by one or more coefficients. This is even more evident in the real-time applications context (e.g. real time control), where the solution is required in a short amount of time, or in the multi-query scenarios (e.g. uncertainty quantification, experimental design), where the FOM is solved for many parameters instances.

To alleviate this, some dimensionality reduction techniques have been traditionally devised which fall under the category of the Reduced Order Models (ROMs): they offer an approximation of the high-fidelity solutions at any desired time-parameter instance. Their success relies on the fact that the high-dimensionality of the FOMs is actually artificial, being just a consequence of the underlying numerical scheme chosen. The major hypothesis at the base of the ROMs is given in fact by the observation that even the behaviour of a complex system can be often described by a combination of few dominant modes. If this is the case, the approximation of the solution submanifold leads to a huge computational cost saving, while possibly keeping high levels of accuracy.

To actually achieve a speedup, the ROMs are decoupled into an offline and online stages:

- During the offline stage, a set of high fidelity solutions is collected and the most

computational expensive operations are performed.

- During the online stage the solution is rapidly predicted, in general by fast function evaluation or by solving a low-dimensional linear system. To allow a continuous map between the time-parameters instances and the solution, interpolation methods or Artificial Neural Networks (ANNs) can be exploited. In the present work, all the ROMs considered made use of the latter.

The most widespread dimensionality reduction technique is the Proper Orthogonal Decomposition (POD) [3], [21], also known as Principal Component Analysis (PCA) in other engineering fields. This method exploits the Singular Value Decomposition (SVD), as it will be discussed in section (1.2), to extract a set of reduced basis, which represent the dominant dynamics of the underlying problem.

The main drawback of SVD-based ROMs is the fact that this matrix factorization is essentially a linear technique, therefore the method may suffer especially if the problem at hand is unsteady or advection dominated.

In the present work two new approaches will be investigated, both developed in the deep learning context, with the aim to improve the accuracy of existing linear ROMs. The first one to be introduced is the Projection Driven Neural Networks - Autoencoder (PDNNs-Autoencoder), in which a deep convolutional autoencoder is used as a dimensionality reduction tool in place of the SVD. An autoencoder, being an Artificial Neural Network, allows to map the high-dimensional solution space in a low-rank non-linear manifold, which is expected to perform better than its linear counterpart.

The second approach, namely the Physics Informed Neural Networks - FOM (PINNs-FOM), is completely different from a conceptual point of view. This method consists of an artificial neural network in which the governing equations of the problem at hand are embedded in the network training algorithm. The main interesting aspects of this approach are:

- it does not require the generation of expensive high-fidelity solution snapshots;
- it is a completely mesh-free method;
- it is easy to implement through high-level python libraries.

Both these methods will be assessed on 4 different test problems and their performances compared against SVD-based ROMs.

The thesis is organized as follows:

Chapter 1

In Chapter 1 the ROMs framework is discussed in detail. The SVD factorization, which is at the core of the POD method, will be described and its main drawbacks presented. Finally, an example of SVD-based approach, namely the POD-Galerkin, will be provided.

Chapter 2

Chapter 2 offers all the theoretical backgrounds concerning the world of Artificial Neural Networks. After a brief introduction, the main ingredients of a ANN are introduced: the activation functions, the loss function, the training algorithm and the available optimizers. Moreover, besides the vanilla Feed Forward Neural Network (FFNN) architecture, a whole section is dedicated to the Convolutional Neural Networks (CNNs), which are employed in the developed autoencoder. Finally, the autoencoder architecture itself is introduced.

Chapter 3

In Chapter 3 the focus is on the Physics Informed Neural Networks. The chapter has been designed to answer how the PDEs are embedded in the network training algorithm, explaining the advantages and the drawbacks of this deep learning branch. To support the discussion, a practical demonstration is offered on a problem governed by a non-parameterized PDE. Finally, few advanced techniques are described.

Chapter 4

Chapter 4 comes with the description of the approaches used in this thesis to deal with the fast solution of problems governed by parameterized PDEs.

Chapter 5

In Chapter 5, 4 test problems are presented. For each case, the application of the various strategies is described and the accuracy achieved is reported for comparison. Besides the quantitative analysis of the error, the chapter is enriched with plots concerning the comparison of the approximated solutions with the high-fidelity reference ones.

Chapter 6

Chapter 6 contains an analysis on the results obtained in Chapter 5, highlighting the strengths and the weaknesses of the proposed approaches, in comparison with the SVD-based methods. Some final remarks on possible future developments to enhance the approaches accuracy are reported.

1 | Reduced Order Models

In the present chapter, the basics behind the model order reduction strategies will be introduced. Together with the application of Physics Informed Neural Networks (PINNs) directly on the governing equations, these strategies represent the core of this work. After a short overview on the various techniques and their motivations, the attention will be devoted to the most widespread one, the Proper Orthogonal Decomposition (POD) and its applications for devising a reduced order model system of equations.

1.1. Full Order Models vs Reduced Order Models

In the broad context of engineering applications, many problems require to be modelled through PDEs and systems of PDEs. In general, a large amount of them do not have, to date, any analytical solution. For this reason, numerical methods are widely exploited since the beginning of the previous century. The available approaches to numerically retrieve the PDEs solution can be classified into two main categories:

- numerical computation;
- model reduction.

The numerical computation strategy leads to the so called Full Order Models. These models are characterized by the highest level of accuracy. The solution is derived by discretizing the spatio-temporal domain with a fine mesh, obtaining an artificially high-dimensional system of equations which can be later solved. Common examples are Finite Difference Method (FDM) [28], Finite Elements Method (FEM) [35] and Finite Volume Method (FVM) [39].

The high dimension n of the system, which typically demands both high computational resources and time, is the main drawback of this approach. The problem becomes even more evident once the PDEs are parameterized and the solution is required over a wide range of values in the parameter space: a list of relevant examples includes topology optimization, uncertainty quantification and parameter estimation.

Moreover, while during the final stages of a project a high fidelity solution is desired and thus a FOM represents the best choice, during the preliminary analysis phase a much faster approach is preferable.

To answer this requirement, a number of strategies have been developed which belong to the model order reduction philosophy. In particular, these methods lead to the so called Reduced Order Models (ROMs) and they seek to replace the complex system by a simpler set of equations to improve the computational speed. The idea behind them is based on the observation that even the behaviour of a complex system can be described by a combination of few dominant modes. Thus, a great dimensionality reduction is in general possible. However, by relieving the high resources demand issue, the trade-off of achieving less accurate results is accepted.

Another great advantage of ROMs against FOMs is that their evaluation is fast and therefore they are available for real-time application contexts for which the latter, being computationally expensive, are not suited.

The most widespread category of ROM are Reduced Basis (RB) methods [33]. In general, they are implemented in an offline - online paradigm:

- during the offline stage, a set of high fidelity solutions is collected (snapshots). These may derive from experimental data as well as from FOM samples and are used to extract the Reduced Basis, which represent the dominant dynamics of the underlying problem. The most famous extraction technique is the Proper Orthogonal Decomposition (POD), which exploits the Singular Value Decomposition algorithm (SVD) [4];
- during the online stage the reduced coefficients are computed. The solution is then recovered as a linear combination of the RB functions weighted by the reduced coefficients.

In turn, there are two different approaches to perform the online stage:

- intrusive approach: the FOM is projected onto the reduced space spanned by the RB functions. The most used projection, the Galerkin projection, leverages the RB functions as the test ones (POD-G method);
- non-intrusive approach: the reduced coefficients are determined via interpolation or regression. To this aim, as we will see in the present work, neural networks may be exploited to perform the task.

The main drawback of the traditional intrusive approaches, such as the POD-G, is the

lack of a priori guarantees of stability, accuracy and convergence. Moreover, the eventual presence of a non-linear term may cancel any time gaining since it scales with the FOM, representing a computational bottleneck.

1.2. Proper Orthogonal Decomposition: an overview

The proper orthogonal decomposition (POD) is one of the most used procedures to extract a set of reduced basis. In particular, it exploits the Singular Value Decomposition (SVD) of a matrix representative of the problem dynamics to retrieve a low rank approximation of it. The SVD is attractive for two main reasons:

- its existence is guaranteed for any matrix (unlike the eigendecomposition);
- it is numerically stable.

Before going through the POD procedure used to derive a ROM, in the following subsection an introduction to the SVD algorithm will be presented.

1.2.1. Singular Value Decomposition

Let us consider a matrix $\boldsymbol{A} \in \mathbb{C}^{n \times m}$. This matrix, in the context we are dealing with, consists of a series of experimental data or high fidelity solutions which are sampled uniformly or randomly over the domain and then appropriately reshaped into column vectors $\boldsymbol{a}_k$.

$$\boldsymbol{A} = \begin{bmatrix} | & | & & | & & | \\ \boldsymbol{a}_1 & \boldsymbol{a}_2 & ... & \boldsymbol{a}_k & ... & \boldsymbol{a}_m \\ | & | & & | & & | \end{bmatrix} \tag{1.1}$$

The columns of the matrix may represent therefore the evolution in time of a physical variable or the parameterized solution for a given value assumed by the parameter. For this reason, the columns of $\boldsymbol{A}$ are called *snapshots*.

In general, $n >> m$, since the mesh used for the computation of the FOM solution - or the measurement grid - is usually fine, and the number of snapshots collected cannot be too high (otherwise the use of a ROM is no more time convenient).

The SVD is a unique matrix decomposition which ensures the existence of two unitary matrices $\boldsymbol{U} = \begin{bmatrix} \boldsymbol{u}_1 | ... | \boldsymbol{u}_n \end{bmatrix} \in \mathbb{C}^{n \times n}$ and $\boldsymbol{V} = \begin{bmatrix} \boldsymbol{v}_1 | ... | \boldsymbol{v}_m \end{bmatrix} \in \mathbb{C}^{m \times m}$ with orthonormal columns and a matrix $\boldsymbol{\Sigma} = diag(\sigma_1, ...\sigma_m) \in \mathbb{R}^{n \times m}$ characterized by non-negative real values on

the diagonal and zero off it, such that:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^*, \tag{1.2}$$

with * being the Hermitian transpose operator. Note that:

- $\boldsymbol{u}_i$ are called the *Left Singular Vectors* of $\boldsymbol{A}$;
- $\boldsymbol{v}_j$ are called the *Right Singular Vectors* of $\boldsymbol{A}$;
- σ_k are called the *Singular Values* of $\boldsymbol{A}$ and they are ordered from the largest to the smallest: $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_m$.

Since $n > m$, $\boldsymbol{\Sigma}$ has at most m nonzero entries on the diagonal, and thus can be rewritten as:

$$\boldsymbol{\Sigma} = \begin{bmatrix} \hat{\boldsymbol{\Sigma}} \\ \boldsymbol{0} \end{bmatrix}. \tag{1.3}$$

Therefore, the *economy* SVD can be defined:

$$\boldsymbol{A} = \begin{bmatrix} \hat{\boldsymbol{U}} & \hat{\boldsymbol{U}}^{\perp} \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{\Sigma}} \\ \boldsymbol{0} \end{bmatrix} \boldsymbol{V}^* = \hat{\boldsymbol{U}}\hat{\boldsymbol{\Sigma}}\boldsymbol{V}^*. \tag{1.4}$$

The difference between the full and economy SVD can be graphically appreciated in Figure (1.1).

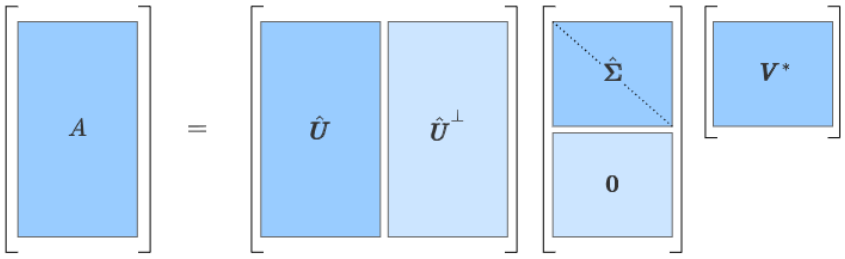
As mentioned above, the SVD factorization is exploited in order to reduce the dimensionality of the problem by extracting a low rank basis matrix from $\boldsymbol{A}$ whose associated linear space is still capable of describing the dynamics of the underlying problem in an accurate way. The Eckart - Young theorem [10] guarantees that the low rank-r approximation given by the SVD is the optimal one in the least squares sense:

$$\arg\min_{\tilde{\boldsymbol{A}}} \left\| \boldsymbol{A} - \tilde{\boldsymbol{A}} \right\|_F = \tilde{\boldsymbol{U}}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{V}}^* \tag{1.5}$$

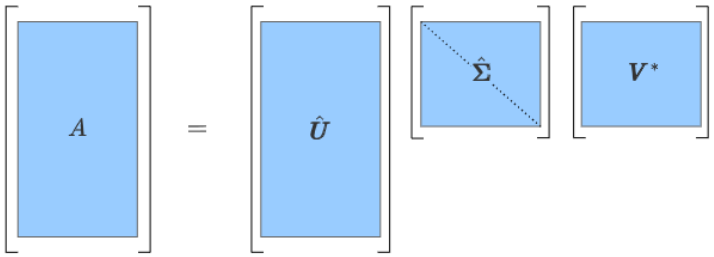
,

where $\tilde{\boldsymbol{U}}$ and $\tilde{\boldsymbol{V}}$ are respectively the first r leading columns of $\boldsymbol{U}$ and the first r leading columns of $\boldsymbol{V}$, and $\| \cdot \|_F$ denotes the Frobenius norm.

From a mathematical point of view, the computation of the SVD of the matrix $\boldsymbol{A}$ is related to an eigenvalue problem involving the correlation matrices $\boldsymbol{A}\boldsymbol{A}^*$ and $\boldsymbol{A}^*\boldsymbol{A}$:



(a) Full SVD



(b) Economy SVD

Figure 1.1: Comparison between the two SVDs.

$$\boldsymbol{A}\boldsymbol{A}^* = \boldsymbol{U}\begin{bmatrix}\hat{\boldsymbol{\Sigma}}\\ \boldsymbol{0}\end{bmatrix}\boldsymbol{V}^*\boldsymbol{V}\begin{bmatrix}\hat{\boldsymbol{\Sigma}} & \boldsymbol{0}\end{bmatrix}\boldsymbol{U}^* = \boldsymbol{U}\begin{bmatrix}\hat{\boldsymbol{\Sigma}}^2 & 0\\ 0 & 0\end{bmatrix}\boldsymbol{U}^*, \tag{1.6a}$$

$$\boldsymbol{A}^*\boldsymbol{A} = \boldsymbol{V}\begin{bmatrix}\hat{\boldsymbol{\Sigma}} & \boldsymbol{0}\end{bmatrix}\boldsymbol{U}^*\boldsymbol{U}\begin{bmatrix}\hat{\boldsymbol{\Sigma}}\\ \boldsymbol{0}\end{bmatrix}\boldsymbol{V}^* = \boldsymbol{V}\hat{\boldsymbol{\Sigma}}^2\boldsymbol{V}^*. \tag{1.6b}$$

In fact, since $\boldsymbol{U}$ and $\boldsymbol{V}$ are unitary matrices, it is possible to conclude that $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{\Sigma}$ are solutions of the following eigenvalue problem:

$$\boldsymbol{A}\boldsymbol{A}^*\boldsymbol{U} = \boldsymbol{U}\begin{bmatrix}\hat{\boldsymbol{\Sigma}}^2 & 0\\ 0 & 0\end{bmatrix}, \tag{1.7a}$$

$$\boldsymbol{A}^*\boldsymbol{A}\boldsymbol{V} = \boldsymbol{V}\hat{\boldsymbol{\Sigma}}^2. \tag{1.7b}$$

From Equations (1.7a) and (1.7b) it is straightforward to conclude that the columns of $\boldsymbol{U}$ are the eigenvectors of the $\boldsymbol{A}\boldsymbol{A}^*$ matrix, the columns of $\boldsymbol{V}$ are the eigenvectors of the $\boldsymbol{A}^*\boldsymbol{A}$ matrix and $\boldsymbol{\sigma}_k$ are the square root of both $\boldsymbol{A}\boldsymbol{A}^*$ and $\boldsymbol{A}^*\boldsymbol{A}$ eigenvalues.

1.2.2. SVD main drawback: data alignment

The most relevant drawback of the SVD is that it often requires a huge data preprocessing - translations, rotations and scaling of the snapshots - before performing it. If the data

are not aligned, in fact, the relevant singular values of the underlying problem increase, meaning that the number of columns of $\boldsymbol{U}$ which must be considered to accurately describe the dynamics becomes high, thus lowering the dimensionality reduction capability of the whole POD procedure.

To mitigate this issue, new non-linear techniques have been proposed in substitution of the SVD: the convolutional autoencoders [11], [22], which will be later discussed in Section (2.4).

1.3. Proper Orthogonal Decomposition - Galerkin projection (POD-G) method

Let us consider a generic parameterized time-dependent system of nonlinear PDEs in the form:

$$\boldsymbol{u}_t = \boldsymbol{N}(\boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_{xx}, ..., x, t, \boldsymbol{\mu}). \tag{1.8}$$

The correspondent discrete FOM obtained after applying a traditional numerical scheme such as the FEM or the FVM reads as:

$$\frac{d\boldsymbol{u}_h(t,\boldsymbol{\mu})}{dt} = \boldsymbol{L}(\boldsymbol{\mu})\boldsymbol{u}_h(t,\boldsymbol{\mu}) + \boldsymbol{N}(\boldsymbol{u}_h(t,\boldsymbol{\mu})) + \boldsymbol{C}(\boldsymbol{\mu}), \tag{1.9}$$

where $\boldsymbol{L}$ and $\boldsymbol{N}$ are respectively a linear and a non-linear operator, $\boldsymbol{C}$ is a constant term and $\boldsymbol{u}_h$ is the discrete full order model solution.

It is possible then to approximate $\boldsymbol{u}_h$ by a Galerkin expansion:

$$\boldsymbol{u}_h(t,\boldsymbol{\mu}) \approx \overline{\boldsymbol{u}} + \tilde{\boldsymbol{U}}\boldsymbol{\alpha}(t,\boldsymbol{\mu}), \tag{1.10}$$

where:

- $\tilde{\boldsymbol{U}}$ is the truncated basis matrix which, in this case, is obtained via the SVD of the snapshots matrix $\boldsymbol{A}$;
- $\overline{\boldsymbol{u}}$ is a constant term, usually taken as the mean of $\boldsymbol{A}$ to avoid the presence of dominant basis;
- $\boldsymbol{\alpha}(t,\boldsymbol{\mu})$ is the vector of the reduced coefficients.

Substituting the Equation (1.10) into System (1.9), a non-linear system for the unknown $\boldsymbol{\alpha}(t, \boldsymbol{\mu})$ can be written. In compact notation it becomes:

$$\tilde{\boldsymbol{U}} \frac{d\boldsymbol{\alpha}(t, \boldsymbol{\mu})}{dt} = \boldsymbol{L}(\boldsymbol{\mu}) \tilde{\boldsymbol{U}} \boldsymbol{\alpha}(t, \boldsymbol{\mu}) + \boldsymbol{N}(\tilde{\boldsymbol{U}} \boldsymbol{\alpha}(t, \boldsymbol{\mu})) + \boldsymbol{C}(\boldsymbol{\mu}). \tag{1.11}$$

The dimensionality reduction is achieved by projecting System (1.11) onto a subspace $\boldsymbol{W} \in \mathbb{R}^{n \times r}$:

$$\boldsymbol{W}^T \tilde{\boldsymbol{U}} \frac{d\boldsymbol{\alpha}(t, \boldsymbol{\mu})}{dt} = \boldsymbol{W}^T \left(\boldsymbol{L}(\boldsymbol{\mu}) \tilde{\boldsymbol{U}} \boldsymbol{\alpha}(t, \boldsymbol{\mu}) + \boldsymbol{N}(\tilde{\boldsymbol{U}} \boldsymbol{\alpha}(t, \boldsymbol{\mu})) + \boldsymbol{C}(\boldsymbol{\mu}) \right). \tag{1.12}$$

In the context of a Galerkin projection, $\boldsymbol{W} = \tilde{\boldsymbol{U}}$ and thus the truncated basis matrix is used for the projection of the ROM into a low rank subspace:

$$\tilde{\boldsymbol{U}}^T \tilde{\boldsymbol{U}} \frac{d\boldsymbol{\alpha}(t, \boldsymbol{\mu})}{dt} = \tilde{\boldsymbol{U}}^T \left(\boldsymbol{L}(\boldsymbol{\mu}) \tilde{\boldsymbol{U}} \boldsymbol{\alpha}(t, \boldsymbol{\mu}) + \boldsymbol{N}(\tilde{\boldsymbol{U}} \boldsymbol{\alpha}(t, \boldsymbol{\mu})) + \boldsymbol{C}(\boldsymbol{\mu}) \right), \tag{1.13}$$

which, exploiting the orthogonality of $\tilde{\boldsymbol{U}}$, becomes:

$$\frac{d\boldsymbol{\alpha}(t, \boldsymbol{\mu})}{dt} = \tilde{\boldsymbol{U}}^T \left(\boldsymbol{L}(\boldsymbol{\mu}) \tilde{\boldsymbol{U}} \boldsymbol{\alpha}(t, \boldsymbol{\mu}) + \boldsymbol{N}(\tilde{\boldsymbol{U}} \boldsymbol{\alpha}(t, \boldsymbol{\mu})) + \boldsymbol{C}(\boldsymbol{\mu}) \right). \tag{1.14}$$

The solution of the ROM is retrieved by solving System (1.14) and substituting the resulting $\boldsymbol{\alpha}(t, \boldsymbol{\mu})$ into Equation (1.10).

If the parameter dependence of the FOM is non-affine, an interpolation technique is required to obtain a continuous map between the time-parameters instances and the corresponding reduced coefficients. In this work ANNs has been employed for this purpose, due to both their fast online evaluation capability and accuracy.

As mentioned before, the computation of the nonlinear term may represent an issue: since it scales with the dimension of the FOM n, it represents a real bottleneck and an obstacle for a true time effective ROM. To deal with it, new techniques has been already widely used such as the gappy POD [15], the discrete empirical interpolation method (DEIM, [5]) and the missing point estimation (MPE, [2]). They allow for a trade-off between the speed up of the whole process and the accuracy of the nonlinear term approximation.

A last issue which requires to be addressed is the need for finding a rule to determine the rank r of the truncated basis matrix. Usually a threshold is defined: the number of modes considered is such that their associated singular values captures are least, for example,

99% or 99.99% of the energy of the original data. As discussed in subsection (1.2.2), in presence of unsteady advection-dominated problems where the snapshots are necessarily misaligned, the rank r required to capture such amount of energy has to be higher than in the other problems.

The summary of the POD-G procedure can be appreciated below, divided respectively into the offline stage (Algorithm (1.1)) and online stage (Algorithm (1.2)).

Algorithm 1.1 POD-G procedure - Offline Stage

1: Generate the set of parameters $\boldsymbol{M} = \{\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_m\}$
2: Solve the FOM for each parameter instance; assemble the snapshots matrix $\boldsymbol{A} = \Big[\boldsymbol{u}_h(\boldsymbol{\mu}_1), ..., \boldsymbol{u}_h(\boldsymbol{\mu}_m)\Big]$
3: Compute the SVD of $\mathbf{A}$; analyze the singular values obtained
4: Truncate the basis matrix $\mathbf{U}$ keeping only the relevant modes
5: Introduce the Galerkin expansion of the FOM solution: $\boldsymbol{u}_h = \overline{\boldsymbol{u}} + \boldsymbol{U}\boldsymbol{\alpha}(t, \boldsymbol{\mu})$
6: Assemble the ROM. For reference, see System (1.11).

Algorithm 1.2 POD-G procedure - Online Stage

1: Project the ROM equations into the basis subspace
2: Solve the ROM system for the desired parameter
3: Compute the ROM solution by inserting the reduced coefficients $\boldsymbol{\alpha}(t, \boldsymbol{\mu})$ just retrieved into the Galerkin expansion formula (1.10)

2 | Artificial Neural Networks

The concept of the Artificial Neural Networks (ANNs, from now on) is surprisingly not new. In 1943, McCullong and Pitts proposed the first Threshold Logic Unit [32], whereas few years later Rosenblatt developed the first Perceptron [36], the precursor of the modern neurons. Both these early attempts to supplement existing algorithms with artificial intelligence were based on the Brain Computational Model, which is characterized by two main elements:

- the synapses, which connect each neuron and allow the exchange of electrical charges between them;
- the neurons, which collect the charges from the synapses. Note that they can have both positive and negative sign. Once a certain threshold is reached, the charges are released to the other neighbor units.

As it will be shown in section (2.2), modern ANNs algorithms are based on the Brain Computational Model too since they are an evolution of these early attempts.

Before introducing the main concepts which lie behind the ANNs success and describing the architectures used in the present work, the following section will provide a short introduction on the Deep Learning field and how this is related with the traditional Machine Learning world.

2.1. Machine Learning vs Deep Learning

Machine Learning is a well established category of research and algorithms focused in finding patterns in data, with the purpose of using those patterns to make predictions.

Given a set of collected data $D = x_1, x_2, x_3, ..., x_N$, Machine Learning algorithms can be classified into three main paradigms:

- Supervised Learning: if a set of desired targets $T = t_1, t_2, t_3, ..., t_N$ is provided through human supervision, the algorithms learn to predict the correct targets given a new set of inputs;

- Unsupervised Learning: the algorithms exploit regularities in D to discover hidden patterns among data without the need of human supervision. Those hidden patterns are then used to make predictions;
- Reinforcement Learning: an agent performs a set of actions $A = a_1, a_2, a_3, ..., a_N$. These actions affect the environment defined by the data and produce a set of rewards $R = r_1, r_2, r_3, ..., r_N$. In this context the agent learns to act in order to maximize the rewards in the long term.

Despite of the learning paradigm considered, Machine Learning algorithms consist of two building blocks:

- an hand-crafted feature extractor, which manipulates the input data to keep only the relevant information;
- the algorithm core, which, given the extracted features, delivers the prediction.

Deep Learning is a subclass of Machine Learning in which, regardless of the paradigm considered, the first block, i.e, the feature extractor, is not hand-crafted. In fact it becomes part of the algorithm core and thus trained in a data-driven way as well. In general, the possibility to tune the feature extractor allows to achieve better performances with respect to the Machine Learning counterpart.

Artificial Neural Networks (ANNs) represents the most famous and nowadays widespread category of Deep Learning algorithms.

2.2. Artificial Neural Networks: an overview

Various obstacles and pitfalls mined the Deep Learning study field since the beginning, leading to the so called *AI winters* multiple times during the last century. The back-propagation algorithm [37], which is at the base of modern ANNs training process, was proposed in 1986, but due to the limited computational resources available, the time was not ripe yet for the spreading of this technology.

The interest on ANNs among the computer scientists, and more generally, among the scientific community has seen a sudden and unprecedented growth at the beginning of the previous decade, mainly for two reasons:

- the increase in the available computational resources, in particular the increase of the GPUs performances, since the ANNs architecture and the backpropagation algorithm make them suitable for parallel computation, easily performed on this kind of hardware;

- the large amount of data sets collected, especially in the field of computer vision, to perform supervised learning tasks. One of the most relevant factors that have contributed to this abundance is the spread of the use of smartphones and in particular the social networks. Nowadays, Internet of Things is playing an important role in autonomous data collection as well.

The architecture of a vanilla Feed Forward Neural Network (FFNN, see Figure (2.1)) is fully described by the number of hidden layers, neurons per layer, activation functions and weight values, and is such that each neuron output depends on all the neurons inside the previous layer.

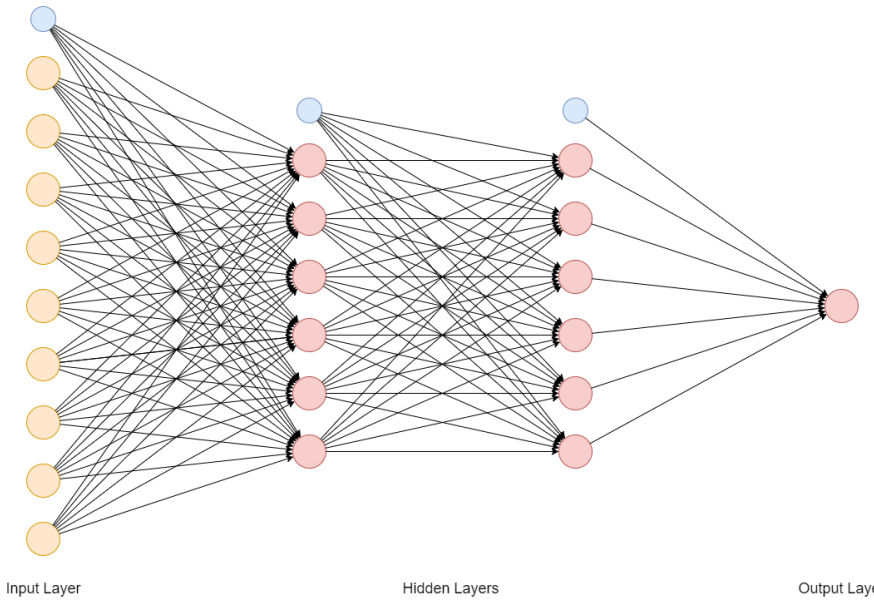


Figure 2.1: Vanilla FFNN. It is characterized by an input layer of 9 neurons (in orange), 2 hidden layers of 6 neurons each and an output layer of a single neuron. The biases are in blue.

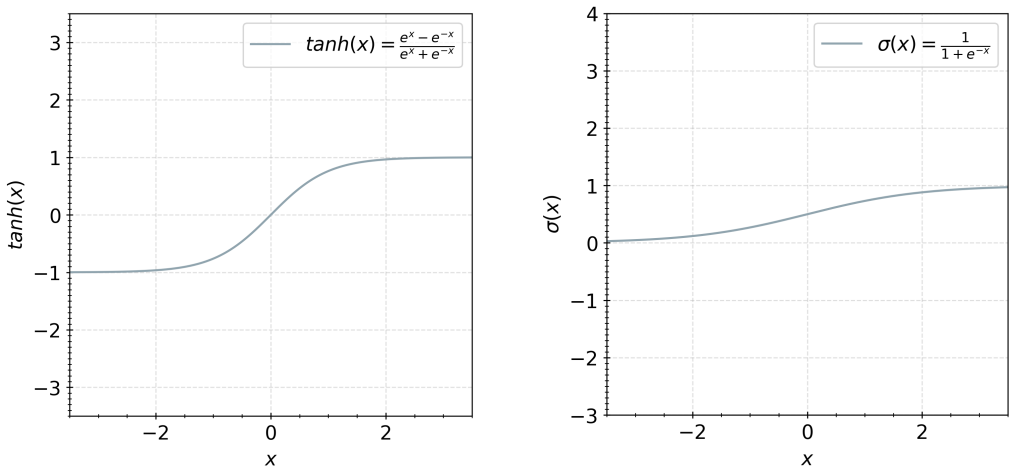
The Universal Approximation Theorem [17] states that, in principle, for a regression task, a single hidden layer FFNN with S shaped activations can approximate any measurable function to any desired degree of accuracy on a compact set. However, this does not guarantee to find a suitable optimizer to compute the optimal weights, and the layer may become unfeasibly large, thus in general adding more hidden layers is a common practice.

Given this premise, the choice of the FFNN hyperparameters is not straightforward and there are no strictly rules to comply with. Moreover, the best architecture to use is often problem dependent: for example, a more complex task may require more hidden layers or neurons per layer, but it is not always the case.

To better clarify all the main ingredients of a FFNN, the remaining part of this section is devoted to offer an overview of them.

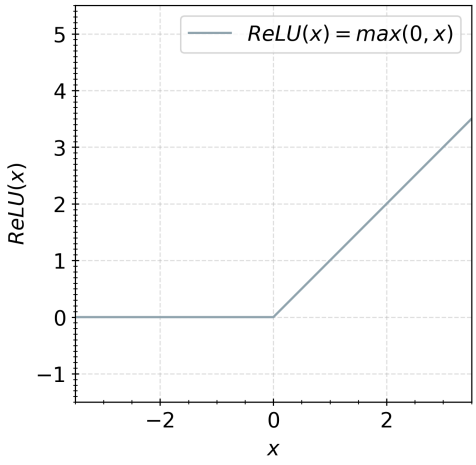
2.2.1. Activation Functions

The activation functions are the core of a neuron, mapping the sum of the weighted inputs into the neuron output. They play an important role in the overall ANNs performance, since they are the main source of non-linearity. The first activation functions employed were the hyperbolic tangent (Figure (2.2a)) and the sigmoid (Figure (2.2b)), and it's clear why: they recall the threshold function of the perceptron. In fact, more precisely, they are the differentiable counterpart respectively of the sign and Heaviside functions.

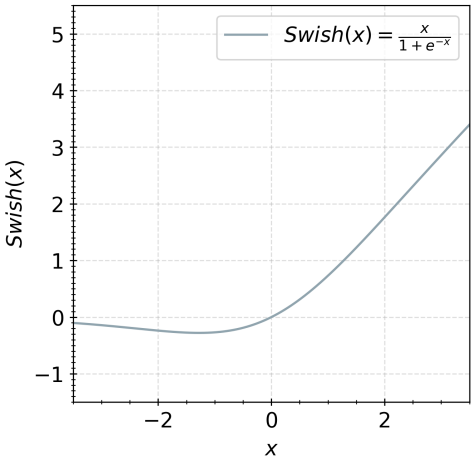


(a) Hyperbolic tangent activation function

(b) Sigmoid activation function



(c) ReLU activation function



(d) Swish activation function

Figure 2.2: Nowadays most widespread activation functions

Today, especially in the computer vision field, the use of the Rectified Linear Unit activation (ReLU, see Figure (2.2c)), along with its variants, is common, since it addresses some issues related to the backpropagation algorithm which will be later discussed in subsection (2.2.4).

For the same reason, other activation functions are being continuously tested and are gaining more and more attention such as the non-monotonic Swish activation function (depicted in Figure (2.2d)). Eventually, all of these presented activation functions can be parameterized by a trainable parameter which adaptively modifies their slope, with the purpose of improving the network performances [19].

To conclude, the linear activation function is used as well. It is commonly employed for example in the output layer of networks performing regression tasks, since in this case the map output has to span the whole $\mathbb{R}$ set.

2.2.2. Loss function

Once the network makes a prediction, it is necessary to quantify the error and possibly find an algorithm that iteratively tries to minimize it. In this subsection the optimal loss function to be minimized for a regression task will be derived.

Let us consider the case of a supervised learning problem where it is required to approximate a target function t having N observations available. Given the output of the neural network $g(\boldsymbol{x}_n|\boldsymbol{w})$, let us suppose that for each observation the target is given by the relationship:

$$t_n = g(\boldsymbol{x}_n|\boldsymbol{w}) + \epsilon_n, \tag{2.1}$$

where $\epsilon_n \sim N(0, \sigma^2)$ is a Gaussian zero mean noise with constant variance σ^2.

It is possible to say then that the target approximately belongs to a Gaussian distribution with mean value $g(\boldsymbol{x}_n|\boldsymbol{w})$ and variance σ^2:

$$t_n \sim N(g(\boldsymbol{x}_n|\boldsymbol{w}), \sigma^2). \tag{2.2}$$

The objective is to determine the set of weights $\boldsymbol{w}$ of the neural network which, for each observation, maximize the probability of the target, i.e., its likelihood under the designed model.

From Equation (2.2) it comes that the target probability given a generic observation reads:

$$p\left(t_n|g(\boldsymbol{x}_n,\boldsymbol{w}),\sigma^2\right)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(t_n-g(\boldsymbol{x}_n|\boldsymbol{w}))^2}{2\sigma^2}}. \tag{2.3}$$

Assuming that the target at each observation is independent and identically separated from the others, the likelihood of the target under the given set of parameters reads as:

$$\begin{aligned} L(\boldsymbol{w}) &= \prod_{n=1}^{N} p\left(t_n|g\left(\boldsymbol{x}_n,\boldsymbol{w}\right),\sigma^2\right) \\ &= \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(t_n-g(\boldsymbol{x}_n|\boldsymbol{w}))^2}{2\sigma^2}}. \end{aligned} \tag{2.4}$$

Now, to convert the products into sums without loosing the point of the discussion, it is possible to take the logarithm of the likelihood:

$$l(\boldsymbol{w})=\sum_{n=1}^{N}\left(ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right)-\frac{1}{2\sigma^2}\left(t_n-g\left(\boldsymbol{x}_n|\boldsymbol{w}\right)\right)^2\right). \tag{2.5}$$

Finally, the weights that maximize the likelihood can be derived as:

$$\begin{aligned} argmax_{\boldsymbol{w}}l(\boldsymbol{w}) &= argmax_{\boldsymbol{w}}\sum_{n=1}^{N}\left(ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right)-\frac{1}{2\sigma^2}\left(t_n-g\left(\boldsymbol{x}_n|\boldsymbol{w}\right)\right)^2\right) \\ &= argmin_{\boldsymbol{w}}\sum_{n=1}^{N}\left(t_n-g\left(\boldsymbol{x}_n|\boldsymbol{w}\right)\right)^2. \end{aligned} \tag{2.6}$$

It is possible to conclude then that the most suitable loss function to be minimized when dealing with a regression task is the sum of squared errors (or better, to avoid numerical instabilities, the mean squared error MSE):

$$MSE=\frac{1}{N}\sum_{n=1}^{N}\left(t_n-g(\boldsymbol{x}_n|\boldsymbol{w})\right)^2. \tag{2.7}$$

Variants of Equation (2.7) are also available, such as the root mean squared error RMSE (see Equation (2.8)) and the mean absolute error MAE (see Equation (2.9)):

$$RMSE=\sqrt{MSE}=\sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(t_n-g(\boldsymbol{x}_n|\boldsymbol{w})\right)^2}, \tag{2.8}$$

$$MAE = \frac{1}{N}\sum_{n=1}^{N} |t_n - g(\boldsymbol{x}_n|\boldsymbol{w})| \,. \tag{2.9}$$

The MAE is less sensitive to the presence of outliers, since the difference between labeled data and predictions is not squared.

Following the same reasoning, but making a different hypothesis on the target distribution, it is possible to determine that for a classification task the best choice for a loss function to be minimized is the Crossentropy CE:

$$CE = -\sum_{n=1}^{N} t_n \cdot log\left(g\left(\boldsymbol{x}_n|\boldsymbol{w}\right)\right). \tag{2.10}$$

2.2.3. Backpropagation algorithm

Once a suitable loss function has been derived, the objective is to determine the set of network weights which minimizes it. This is exactly what happens during the training of the neural network.

To find the minimum of a generic function, it is necessary to compute its partial derivatives and set them to zero:

$$\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} = 0. \tag{2.11}$$

Since analytical solutions of Equation (2.11) are practically never available, the optimal weights are computed through an iterative process:

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^{k} - \eta \left.\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}}\right|_{\boldsymbol{w}^k}, \tag{2.12}$$

where k is the generic iteration and η is the learning rate, which is a tunable hyperparameter.

To avoid being trapped in local minima, some optimizers include also the momentum, i.e., the gradient of the loss function at the $k-1$ iteration, in the formula:

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^{k} - \eta \left.\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}}\right|_{\boldsymbol{w}^k} - \alpha \left.\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}}\right|_{\boldsymbol{w}^{k-1}}, \tag{2.13}$$

where α is a tunable parameter as well, in general lower than η.

The backpropagation is the algorithm through which the gradient of the loss function with respect to all the weights is computed. To understand how it works, let's consider the one hidden layer neural network sketched in Figure (2.3).

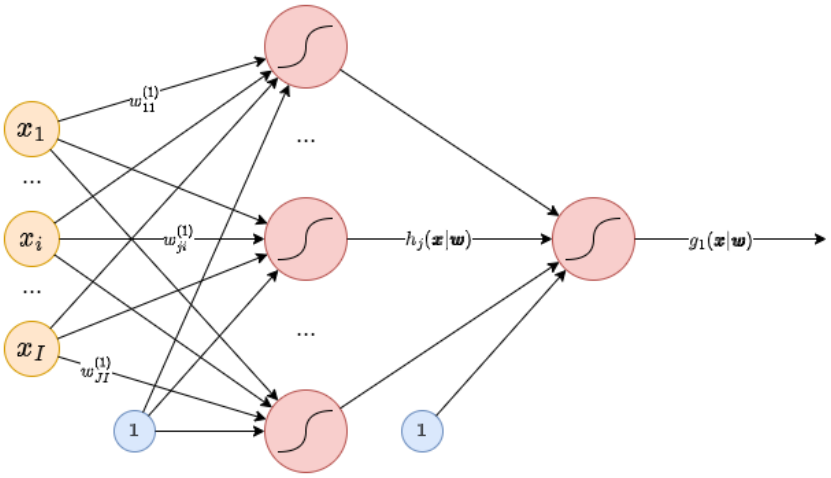


Figure 2.3: Generic one hidden layer neural network

For each layer l, the weights can be stored in a matrix $\boldsymbol{W}^l$, such that the generic entry $w_{ji}^{(l)}$ refers to the weight connecting the $i-th$ node of the layer l to the $j-th$ node of the layer $l+1$.

Since the gradient of E with respect to the weights is a composite function, it can be expressed through the chain rule. For example, the gradient of E with respect to the weights of the hidden layer in Figure (2.3), at each k iteration and considering all the n observations, is:

$$\frac{\partial E(\boldsymbol{w})}{\partial w_{ji}^{(1)}} = -2\sum_{n=1}^{N}\left(t_n - g_1\left(\boldsymbol{x}_n|\boldsymbol{w}\right)\right)\cdot g_1^{'}\left(\boldsymbol{x}_n|\boldsymbol{w}\right)\cdot w_{1j}^{(2)}\cdot h_j^{'}\left(\boldsymbol{x}_n,\boldsymbol{w}\right)\cdot x_{i,n}, \tag{2.14}$$

where each term is the result of the application of the chain rule:

$$\frac{\partial E(\boldsymbol{w})}{\partial w_{ji}^{(1)}} = \frac{\partial E(\boldsymbol{w})}{\partial g_1\left(\boldsymbol{x}_n|\boldsymbol{w}\right)}\cdot\frac{\partial g_1\left(\boldsymbol{x}_n|\boldsymbol{w}\right)}{\partial\left(w_{1j}^{(2)}h_j\left(\boldsymbol{x}_n|\boldsymbol{w}\right)\right)}\cdot\frac{\partial\left(w_{1j}^{(2)}h_j\left(\boldsymbol{x}_n|\boldsymbol{w}\right)\right)}{\partial h_j\left(\boldsymbol{x}_n|\boldsymbol{w}\right)}\cdot\frac{\partial h_j\left(\boldsymbol{x}_n|\boldsymbol{w}\right)}{\partial\left(w_{ji}^{(1)}x_{i,n}\right)}\cdot\frac{\partial\left(w_{ji}^{(1)}x_{i,n}\right)}{\partial w_{ji}^{(1)}}. \tag{2.15}$$

The computation of these terms is really efficient, since:

- $\frac{\partial E(\cdot)}{\partial g_1(\cdot)}$ is directly available once the output of the network $g_1(\cdot)$ is computed;
- $\frac{\partial g_1(\cdot)}{\partial w_{1j}^{(2)} h_j(\cdot)}$ and $\frac{\partial h_j(\cdot)}{\partial w_{ji}^{(1)} x_{i,n}}$ can be computed as soon as $g_1(\cdot)$ and $h_j(\cdot)$ are obtained;
- $\frac{\partial w_{1j}^{(2)} h_j(\cdot)}{\partial h_j(\cdot)}$ and $\frac{\partial w_{ji}^{(1)} x_{i,n}}{\partial w_{ji}^{(1)}}$ are simply respectively the corresponding weights and inputs.

Moreover, the computation of the derivatives of the activations showed in Figure (2.2) are straightforward and can be hard-coded without the need to calculate the gradient each time:

$$\tanh'(\cdot) = 1 - \tanh(\cdot)^2, \tag{2.16}$$

$$\text{sigmoid}'(\cdot) = \text{sigmoid}(\cdot)(1 - \text{sigmoid}(\cdot)), \tag{2.17}$$

$$\text{ReLU}'(\cdot) = \begin{cases} 1, & \text{if} x > 0; \\ 0, & \text{otherwise}, \end{cases} \tag{2.18}$$

$$\text{swish}'(\cdot) = \text{swish}(\cdot) + \text{sigmoid}(\cdot)(1 - \text{swish}(\cdot)), \tag{2.19}$$

and the same can be said for other activations.

Finally, considering the weights between two layers, it is possible to notice that their updating formula require the repetition of the same operations derived from the chain rule: this makes the backpropagation algorithm suitable for the parallel computation, which on a good GPU allows to tremendously speed up the training process.

The backpropagation algorithm, also known as gradient descent, has three variants. To describe them, firstly it is necessary to introduce the concept of epoch: an epoch is one pass through all the data x_n. Depending on how many iterations are performed along an epoch, it is possible to distinguish in:

- batch gradient descent: $\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E(\boldsymbol{x_n}|\boldsymbol{w})}{\partial \boldsymbol{w}}$;
- stochastic gradient descent: $\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} \approx \frac{\partial E_{SGD}(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial E(\boldsymbol{x_n}|\boldsymbol{w})}{\partial \boldsymbol{w}}$;
- mini-batch gradient descent: $\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} \approx \frac{\partial E_{MB}(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{1}{M} \sum_{n=1}^{M \subset N} \frac{\partial E(\boldsymbol{x_n}|\boldsymbol{w})}{\partial \boldsymbol{w}}$.

In the batch gradient descent, the gradients are computed for each observation and then the average over all the batches is taken to update the weights. Thus, one iteration of the updating formula corresponds to a whole epoch: the computational cost for each iteration is high.

In the stochastic gradient descent instead the weights are updated at each observation.

In this way N iterations are performed for each epoch. This reduces dramatically the computational cost at each iteration, but there may be too much variance on the gradients, slowing down the convergence process.

The best compromise between the computational resources required and the variance of the gradients is given by the mini-batch gradient descent. In this case each iteration is performed after having computed the average of the gradients over $M < N$ observations, clustered together in M/N mini-batches.

2.2.4. Backpropagation Algorithm and Activation Functions issues

Now that the Backpropagation Algorithm has been introduced it is possible to discuss about the qualities and the issues concerning the activation functions previously exposed.

The first consideration concerns the chain rule involved in the backpropagation algorithm. When computing the product of many factors, there's the risk of obtaining a vanishing or exploding result, depending on the factors values.

Looking at the sigmoid and hyperbolic tangent functions, the maximum of their derivative is found in the origin and its value is respectively 0.25 and 1. While this guarantees to not incur into an exploding gradient problem, it may become problematic from the vanishing gradient problem point of view. Moreover, except a limited region across the origin, these functions saturate rapidly, leading to extremely low gradients. The issue is more and more relevant as the depth of the network increases.

The ReLU activation function is instead a piece wise linear activation function, and its derivatives values are either 0 or 1, respectively for negative and positive inputs. This characteristic solves the vanishing gradient problem but introduces another one: the dying neuron issue.

If the learning algorithm bring the input of the neuron to be negative for all the data samples, the neuron will always output zero and it will hardly recover from this situation: the learning process will fail or the performances of the network will be limited.

Many variants of ReLU has been proposed to mitigate the dying neuron issue, such as the Leaky ReLU and the ELU, in which the function and its gradient are not identically zero in the negative input plane.

2.2.5. Optimizers

A vanilla optimizer (the Gradient Descent optimizer) is a computational tool which embeds the backpropagation algorithm to update the network weights at each iteration through the Equation (2.12). In subsection (2.2.3) it has already been mentioned an upgraded version (Equation (2.13)) which makes use of the momentum to speed up the convergence to avoid local minima (Momentum optimizer).

Nowadays there are many optimizers available. Starting from Equation (2.12), each of them introduces some changes and eventually improvements in their algorithm. If this wasn't enough, combinations of different optimizers have been developed too. In the following lines the most widespread and famous will be rapidly introduced.

NAG - Nesterov Accelerated Gradient

The Nesterov Accelerated Gradient optimizer is a momentum-based Stochastic Gradient Descent algorithm in which the gradient of the loss function with respect to each weight is computed after the momentum jump. For this reason, it can be split into two steps, with the second being a correction one:

$$\boldsymbol{w}^{k+\frac{1}{2}} = \boldsymbol{w}^{k} - \alpha \left. \frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} \right|_{\boldsymbol{w}^{k-1}},$$

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^{k} - \eta \left. \frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} \right|_{\boldsymbol{w}^{k+\frac{1}{2}}},$$

where α is usually set to 0.9 as in the classic momentum SGD optimizer.

In practice the NAG optimizer has proven to perform slightly better than the vanilla momentum one.

AdaGrad - Adaptive Gradient

Neurons of each layer learn differently. In particular, due to the backpropagation algorithm nature, the gradient magnitudes vary across the layers, with early layers suffering from the vanishing gradient phenomena.

From this consideration it is clear that the use of separate adaptive learning rates for each layer may improve the performances of the training process.

AdaGrad [9] modifies the learning rate η at each time step k and for each parameter w_i

by taking into account the history of the previous gradients with respect to the parameter w:

$$\eta_i^{k+1} = \frac{\eta}{\sqrt{G_{ii}^k + \epsilon}},$$

where G_{ii}^k is the sum of the squared gradients of the loss function with respect to the weight w_i at the previous time steps. η is a constant value defined by the user, ϵ is a small smoothing term.

In this way, smaller gradients are automatically associated with larger learning rates, balancing the weight update across the neurons and thus mitigating the vanishing gradients problem in the early layers. Its main advantage is, at the same time, also its main weakness: the continuous accumulation of the gradients at the denominator causes the latter to grow with the number of the iteration. The learning rate may become too small for the network neurons to keep learning.

RMSProp

RMSProp [38] is an extension of AdaGrad which aims to relieve the issue just mentioned. In particular, with RMSProp the window of accumulation of the past gradients is restricted to a fixed constant size. In such manner the fast and thus dangerous monotonic decrease of the learning rate characterizing the AdaGrad optimizer is avoided. It is curious it was not proposed in an official paper but simply as a part of a Coursera lecture.

AdaDelta

Adadelta [43] is another adaptive learning rate optimizer which can be classified as an extension of AdaGrad and a variant of RMSProp.

The user is not required to define a constant η since the numerator is given by the moving average of the weight updates at previous epochs.

Adam - Adaptive Moment Estimation

Adam [23] is an extension of the RMSProp one which considers both the moving average of the first and second moments of the gradients. The weight updates are performed as:

$$w_i^k = w_i^{k-1} - \eta \frac{\hat{m}^k}{\sqrt{\hat{v}^k} + \epsilon},$$

with:

$$m^k = \beta_1 m^{k-1} + (1-\beta_1)\left.\frac{\partial E}{\partial w_i}\right|_{w_i^{k-1}},$$

$$v^k = \beta_2 v^{k-1} + (1-\beta_2)\left.\left(\frac{\partial E}{\partial w_i}\right)^2\right|_{w_i^{k-1}},$$

$$\hat{m}^k = \frac{m^k}{1-\beta_1^k},$$

$$\hat{v}^k = \frac{v^k}{1-\beta_2^k}.$$

In the original paper the optimal initial learning rate is said to be $\eta = 1e-3$. ϵ is a small number to avoid dealing with divisions by zero. β_1 and β_2 are forgetting parameters, with typical values of 0.9 and 0.99 respectively.

L-BFGS - Limited-Memory Broyden–Fletcher–Goldfarb–Shanno algorithm

The L-BFGS algorithm [29] and its main variant L-BFGS-B, unlike the others mentioned until now, are a second-order optimization algorithm, meaning that they make use of the second-order derivative of the loss function (the Hessian) inside the weights updating rule.

The knowledge of the second order derivative of the loss function is useful to calibrate the direction and the step size of the weight update.

Common second-order optimization algorithms belong to the group of Newton methods. The L-BFGS optimizer, instead, is a Quasi-Newton Method, which means that it computes an approximation of the Hessian matrix (more specifically, its inverse) to speed up the updating process. The approximation is performed starting from the gradient values.

Moreover, the L-BFGS algorithm addresses the problem of having a large number of parameters by computing and storing the approximated inverse Hessian matrix in a simplified way.

2.2.6. Training

When the performance of a neural network is not sufficiently accurate, the reasons behind this behaviour could be essentially three:

- the raw input data are not enough or they are not in a optimal form to be fed to the network;
- underfitting: the network is too shallow. It doesn't have enough parameters to capture the relationship between the input and output variables;
- overfitting: the model fits properly the training data, but it lacks in the capability of generalise to the unseen ones.

If the data are not enough, the network in general will have worse performances. In the computer vision field the data augmentation process has been developed to mitigate this issue. In this context, the input picture at each epoch pass through a series of transformations, such as translation, rotation, cropping, mirroring, noising etc. Each of these transformations are governed by some parameters which vary in a random way epoch after epoch. In this way, in addition of the the original images a second set of pictures is generated, which spatially differs from the original ones, but they represent the same subjects.

It is also always a good practice to preprocess the data. The preprocessing phase usually starts with a feature scaling, possibly after the removal of corrupt data or outliers. Different methods are available, such as:

- min-max normalization. It is the simplest method and re-scales the data into the range $[0, 1]$ (see Equation (2.20)) or in the range $[-1, 1]$ (see Equation (2.21)).

$$x_i^{'} = \frac{x_i - \min(\boldsymbol{x})}{\max(\boldsymbol{x}) - \min(\boldsymbol{x})}, \tag{2.20}$$

$$x_i^{'} = 2 \cdot \frac{x_i - \min(\boldsymbol{x})}{\max(\boldsymbol{x}) - \min(\boldsymbol{x})} - 1. \tag{2.21}$$

- standardization. Data are transformed such that they will be characterized by zero-mean and unit-variance. Once mean $\overline{\boldsymbol{x}}$ and the standard deviation σ are available, the standardization formula is given by:

$$x_i^{'} = \frac{x_i - \overline{\boldsymbol{x}}}{\sigma}. \tag{2.22}$$

Normalization and standardization are data transformations which can be performed also inside the network, just before the non-linearities. They can be interpreted as a data-preprocessing tool embedded in the network in each layer they are added. A typical transformation of this kind is the Batch Normalization.

In practice, each neuron pre-activation is standardized. During training, the mean and standard deviation are computed for each minibatch. At test time instead, the global mean and standard deviation are used (they are estimated from the training running averages).

Concerning the other two main performances issues, it is possible to say that while underfitting can be easily mitigated by increasing the number of parameters of the network, e.g, increasing the number of layers or neurons per layer, the overfitting problem is more tricky to tackle.

In order to visualize it, it is first necessary to obtain a measure of generalization. To this aim, the training data set is split in two sets: the training data and the test data. The training set is used to learn the best model parameters in the model development phase, whereas the test set is used to evaluate the model performances in the pre-production one.

Moreover, the training set could be further split to consider also a small validation set. During training, the observations used are the ones inside the training set, but then the model is tested at each epoch on the validation set to verify its ability to generalise. The overall picture of the various sets is available in Figure (2.4).

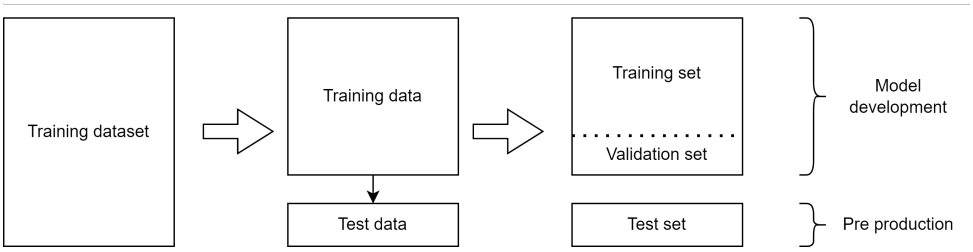


Figure 2.4: Data organization

Thanks to the validation set it is therefore possible to measure the generalisation error and performing model selection: the lower the validation loss, the better the specific model is.

2.2.7. Overfitting prevention

The validation set is a useful tool to check immediately during training if the model is not able to generalise. If this is the case, there is the need for solutions aimed to prevent and mitigate the issue. Let's see some of them:

1. Early stopping. This is the most simple technique. Once the training loss continues to decrease but the validation loss starts showing a plateau or even an increment, it literally stops the network training. It is defined by two parameters: the patience, that is, the delay measured in epochs after which two validation losses are compared, and by a value that represents how much the validation loss should decrease to avoid an early stop.

2. Weight decay. This technique aims at limiting overfitting by means of weight regularization. Regularization is about soft constraining the model weights freedom, based on a-priori assumption. This assumption derives by common practice, and reads as: when a network has small weights, its capability to generalize is increased. This assumption results into the need to modify the loss function by adding a term which takes into account the weights value. The best set of weights become:

$$\hat{\boldsymbol{w}} = argmin_{\boldsymbol{w}} \sum_{n=1}^{N} \left(t_n - g\left(\boldsymbol{x_n}|\boldsymbol{w}\right)\right)^2 + \gamma \sum_{q=1}^{Q} w_q^2, \tag{2.23}$$

   where γ is a tunable hyperparameter which involves the ratio between the variance of the targets and the variance of the weights.

3. Dropout. This technique limits the overfitting by implementing a stochastic regularization of the network. In detail, it randomly switches off a small percentage of neurons at each epoch. In this way, the neurons inside the hidden layer learn to extract the relevant features without relying too much on the others. Each unit is set to zero with $p_j^{(l)}$ probability. For each layer is defined a mask $\boldsymbol{m}^{(l)}$ such that $m_j^{(l)} \sim Be(p_j^l)$, with Be being the Bernoulli distribution.

2.3. Convolutional Neural Networks

An important ANN architecture, which is also employed in the present work, is the Convolutional Neural Network (CNN). The concept of modern CNNs was first presented in [27]. Since 2012, with the success of AlexNet [24] in the ImageNet Large Scale Visual Recognition Challenge, CNNs became the new standard for image recognition tasks, out-

performing all the other existing architectures.

A typical CNN comprises the following layers:

- Convolutional layers
- Activation layers
- Pooling layers
- Dense layers

While the latter have already been described (they are the classical fully connected layers), in the next three subsections some useful information will be provided for the former three.

2.3.1. Convolutional layers

The core of a convolutional layer are the filters (or kernels) whose parameters $\boldsymbol{w}$ need to be learned during the training phase. Each filter is convolved with the input volume (the dimensions can be more than two) to compute a linear combination of all the values in a region. In other words, the filter is slid across the width and height of the input and the dot products between the input and filter are computed at every spatial position (see Figure (2.5)).

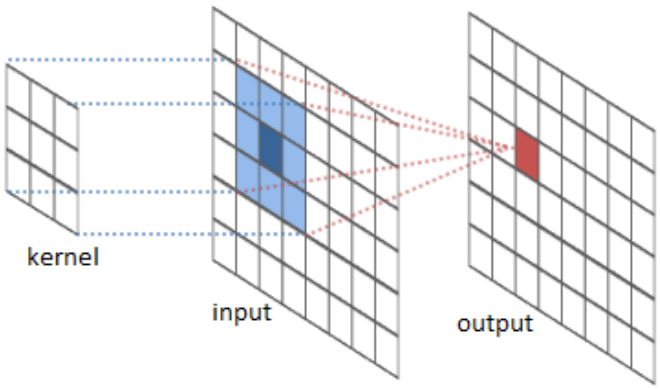


Figure 2.5: Visualization of a CNN filter working principle. Image from [13]

.

Let us consider a filter of size $(2L+1) \times (2L+1)$, centered at the entry (r, c) of a generic two-dimensional input volume $\boldsymbol{I}$. The map of the filter between the input and the output, i.e., the convolution, reads as:

$$(\boldsymbol{I} \otimes \boldsymbol{w}) = b + \sum_{u=-L}^{L} \sum_{v=-L}^{L} \boldsymbol{w}(u,v) \cdot \boldsymbol{I}(r-u, c-v), \tag{2.24}$$

with the addition of a single bias b which is a just trainable threshold to be learned along with the kernel weights.

A graphical visualization of Equation (2.24) can be appreciated in Figure (2.6).

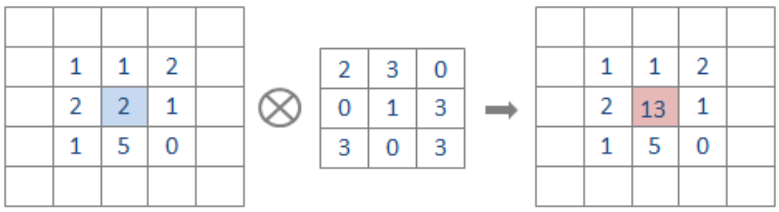


Figure 2.6: Example of the convolution operation. Image from [13]

.

To better extract salient features from the input and thus improving the network performances, the filters used in each convolutional layer can be more than one. The output volume is then obtained by stacking the activation maps of each filter along the depth dimension.

The idea behind the convolutional layers is that the input data shows in general a strong spatial local correlation (for an image, a pixel is more correlated to the nearby pixels than to the others). Given the fact that the discrete correlation formula (plus a constant bias) reads as:

$$(\boldsymbol{I} \otimes \boldsymbol{w}) = b + \sum_{u=-L}^{L} \sum_{v=-L}^{L} \boldsymbol{w}(u,v) \cdot \boldsymbol{I}(r+u, c+v), \tag{2.25}$$

and that the weights $\boldsymbol{w}$ of the filters are trained, the discrete correlation and convolution are equivalent in the neural network context. In fact, if the correlation formula was used (Equation (2.25) instead of Equation (2.24)), the learned filters of the CNN would be equal to the flipped learned filters of a CNN exploiting the convolution operation, leading in the end to the same output.

2.3.2. Activation Layers

The activation layers follow immediately the convolutional ones. They are the main source of non-linearity of the salient features extraction process and they operate element-wise over the input.

For these reasons, in some Deep Learning libraries such as Keras-Tensorflow they can be directly specified inside the convolutional layer.

The most widespread activations in computer vision are the ReLU with its variants such as LeakyReLU and elu. For a more general view over the activation functions, see subsection (2.2.1).

2.3.3. Pooling Layers

The pooling layers reduce the spatial dimensions (height and width) of the volume on which they are applied and they operate independently on each slice of the input.

The most common pooling layers are the MaxPooling and the AveragePooling ones.

MaxPooling is an operation that selects the maximum element from the region specified by the user (see Figure (2.7)). Thus, the output after max-pooling layer is a feature map containing the most prominent features of the previous layer.

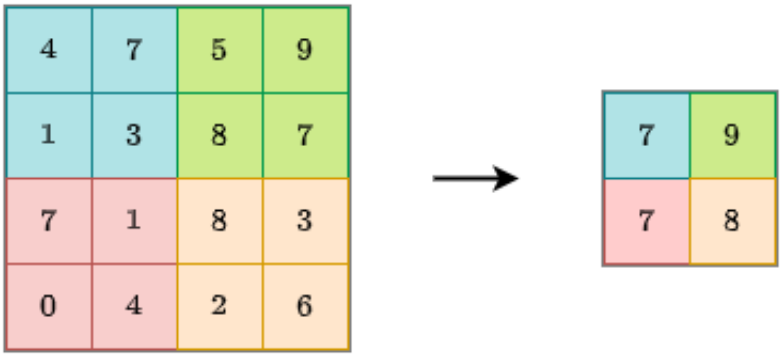


Figure 2.7: Example of the MaxPooling operation.

.

AveragePooling instead computes the average of the entries present in the region covered by the filter (see Figure (2.8)). Therefore, while max pooling returns the most prominent feature in the given area of the feature map, average pooling outputs their average.

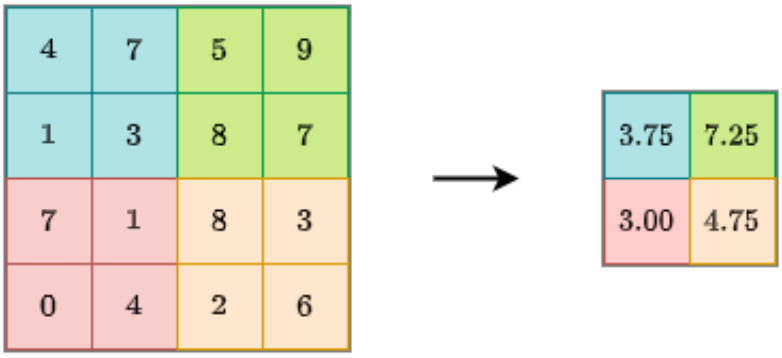


Figure 2.8: Example of the AveragePooling operation.

.

Finally, there exist another pooling operation, the GlobalAveragePooling, which reduces each channel of the volume to a single value, taken as the average of all the feature map entries.

The GlobalAveragePooling operation is usually used at the end of the network for image classification tasks since it has the beneficial property of increasing its robustness to spatial transformation of the input data. In fact, since it takes the average of the feature map entries, the location of the pixels which generated them loose its importance.

2.3.4. Additional technical details

The convolutional layers are characterized by a number of hyperparameters which can be specified by the programmer. The most common ones are:

- *filters*: this parameter specifies the number of filters used in the convolution, and thus the depth of the output. It is a common practice to increase the number of filters used with the depth of the layer to which the operation is applied;
- *kernel size*: it sets the height and width of the 2D convolution window. The most common kernel sizes are 3×3 and 5×5. Greater sizes can be used too, especially in the first shallow layers;
- *strides*: defines the strides of the convolution along the height and width;
- *padding*: to clarify what this hyperparameter does, let us consider a 2D input data of dimension $n \times n$ and a filter of dimension $f \times f$. The output of the convolution will have a shape of $(n - f + 1) \times (n - f + 1)$. Thus, the data shrinks every time a convolution operation is performed. To avoid this, the padding operation consists of simply adding layers of zeros to the edges of the input data.

2.3.5. Relationship between CNNs and FFNNs

Convolution is a linear operation. Therefore, if the input matrix (or image) is unrolled to a vector, the weights of the filters can be seen as the weights of a FFNN with the following characteristics:

- sparse connectivity. The network is not fully connected since the convolution is performed locally;
- weight sharing. The weights are not independent since they come from the same filter.

These two characteristics lead a CNN to be very light from a computational point of view with respect to the fully connected traditional one, allowing a fast training while reducing the risk of overfitting.

Moreover, parameter sharing results in the property of translation equivariance of the convolutional layers: if the input varies, for example after a rigid translation, the output changes accordingly [14].

2.4. Autoencoders

Autoencoders are a class of neural networks used in the field of data reconstruction and dimensionality reduction. The network is trained to learn the identity matrix, i.e., to reconstruct the input (after compressing it in a low-dimension space) and it is made up of two portions: the encoder $E(\boldsymbol{x})$ and the decoder $D(\boldsymbol{x})$.

The structure of the encoder is typically symmetric with respect to the one of the decoder, but in principle there are no constraints.

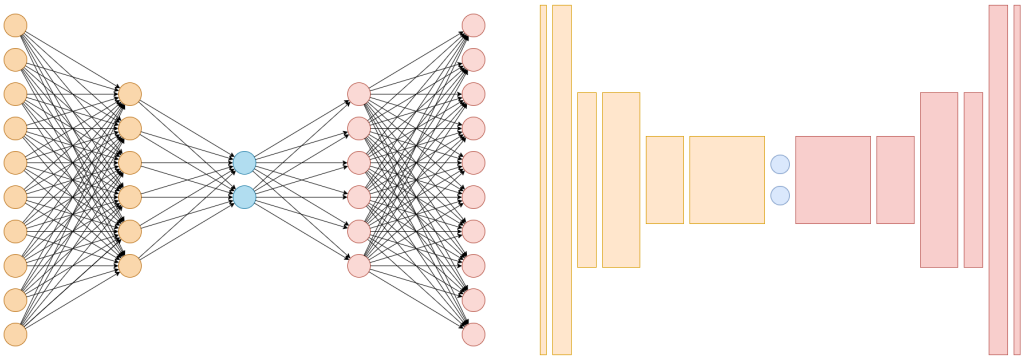
More powerful and non-linear representations can be learned by stacking multiple hidden layers. These networks are called deep autoencoders, and they are the ones used in the present work.

Deep autoencoders can be both FCNNs or CNNs, as it is possible to observe in Figure (2.9). Due to the properties described in subsection (2.3.5), in the present work a CNN deep autoencoder has been employed for the nonlinear dimensionality reduction task.

The reconstruction loss is given by:

$$\mathcal{L}_{AE}(\boldsymbol{w}) = \frac{1}{N} \sum_{\boldsymbol{x} \in \boldsymbol{S}} \| D(E(\boldsymbol{x})) - \boldsymbol{x}) \|_2^2 \,, \tag{2.26}$$

where $\boldsymbol{S}$ is the data set containing N generic input arrays $\boldsymbol{x}$.



(a) FCNN autoencoder (b) CNN autoencoder

Figure 2.9: Examples of autoencoder architectures. In both cases, the encoder is in orange and the decoder in red. The latent representation of the input data is highlighted in blue.

2.4.1. Relationship between autoencoders and SVD

Let us consider an autoencoder with an input dimension n and with a latent space dimension r, such that $E(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^r$ and $D(\boldsymbol{x}) : \mathbb{R}^r \to \mathbb{R}^n$.

If the architecture chosen for the autoencoder is a single hidden layer with linear activation functions, the two map $E(\boldsymbol{x})$ and $D(\boldsymbol{x})$ can be defined as:

$$\begin{aligned} E(\boldsymbol{x}) &= \boldsymbol{W}_E \boldsymbol{x} \\ D(E(\boldsymbol{x})) &= \boldsymbol{W}_D E(\boldsymbol{x}) \end{aligned} \tag{2.27}$$

Where $\boldsymbol{W}_E \in \mathbb{R}^{r \times n}$ and $\boldsymbol{W}_D \in \mathbb{R}^{n \times r}$ are respectively the encoder and the decoder weights matrices, which are comprehensive of the biases. If the encoder and decoder architectures are symmetric, then it is possible to write $\boldsymbol{W} = \boldsymbol{W}_E = \boldsymbol{W}_D^T$. In this specific case, the reconstruction loss is given by:

$$\mathcal{L}_{AE}(\boldsymbol{w}) = \frac{1}{N} \sum_{\boldsymbol{x} \in \boldsymbol{S}} \left\| \boldsymbol{W}\boldsymbol{W}^T \boldsymbol{x} - \boldsymbol{x} \right\|_2^2 \tag{2.28}$$

From Equation (2.28) it is clear that in this case the autoencoder is trained to learn the same subspace as the one spanned by the first r modes coming from the SVD. Note, however, that there are no additional constraints, i.e, $\boldsymbol{W}$ is not required to be an orthogonal matrix and its columns are not required to follow any hierarchical order.

3 | Physics Informed Neural Networks

Physics Informed Neural Networks are a class of ANN central in the present work. In fact, they are the core of the PINNs-FOM approach, allowing the computation of a surrogate of the parameterized PDE solution and its fast real-time evaluation.

Given their importance in this thesis, a whole chapter is dedicated to introducing the reader to the main fascinating concepts and ideas behind this kind of ANNs. After a brief overview in section (3.1), the core of the PINN idea, i.e. its working algorithm, will be presented in section (3.2). Finally, a few advanced techniques will be discussed in section (3.4). In general their aim is to improve the prediction accuracy of the PINNs, but in some cases they may cause a computational overhead which makes them unsuitable.

3.1. Introduction

As written in section (2.2), the success of the ANNs came only after an unprecedented increase in the availability of data. It should be noted though that in many scientific fields, the collection of high fidelity data to perform a supervised learning task might be not feasible both for practical and economical reasons. In a low amount of data context, traditional ANNs are likely to fail to learn or, in the best scenario, to generalize.

Luckily, the physical problems to be modelled usually come with a lot of prior knowledge which could be exploited to drastically reduce the dimensionality of the space in which to find the solutions. For example, this knowledge may take the form of the governing equations, such as Ordinary Differential Equations (ODEs) and Partial Differential Equations (PDEs). Embedding such physical constraints in the network leads to a new category of ANNs: the Physics Informed Neural Networks (PINNs).

PINNs are not a new concept. In fact they were first discussed by Lagaris et al. already in 1998 [25] as a tool to solve physical problems governed by both ODEs and PDEs. Unfortunately time was not yet ripe for them to find some sort of success in the scientific

community, mainly due to the huge amount of resources they required for the available technology.

Recently they were proposed again by Raissi et al. in [34], a paper that has been cited more than 3200 times on which all the renewed interest on the topic is based. Inside their work, Raissi et al. states:

> This simple yet powerful construction allows us to tackle a wide range of problems in computational science and introduces a potentially transformative technology leading to the development of new data-efficient and physics-informed learning machines, new classes of numerical solvers for partial differential equations, as well as new data-driven approaches for model inversion and systems identification.

Let's now analyze which are the major strengths of this new deep learning philosophy:

- no need for mesh generation. This is the first great advantage of the PINNs compared with traditional numerical solvers. Nowadays, in fields such as Computational Fluid Dynamics, the design of the mesh is often the most time consuming phase of the project, since the level of refinement of the mesh as well as the shape of its elements may affect the results in a relevant way;
- flexibility. PINNs can be both used for solve forward and inverse problems with few changes in the code. In the first case the objective is to determine the solution of the problem at hand, while the second case is about model identification, i.e., determine the parameters characterizing the governing equations starting from observed data. Besides, the possibility to easily integrate high fidelity acquired data is also a demonstration of flexibility, if compared with traditional numerical solvers.

As concern the drawbacks, instead:

- novelty. Since PINNs bloomed very recently, there's still a lack of mathematical background supporting the topic. It's still quite cumbersome to check for the best network architecture, which as it will be discussed later it is highly problem-dependent. Moreover, there is very few mathematical theory about convergence estimation;
- less accuracy. In most of the forward problems PINNs performances are still lower than the traditional solvers. However, in the presence of scattered data of the true solution (coming from real time measurements, for example) their performances improve;

- speed. Despite of the great improvements in the recent years, traditional solvers are still faster than PINNs on most of the forward problems. PINNs becomes competitive only when dealing with parameterized PDEs or when the mesh generation is complicated.

The idea behind the PINN algorithm is simple. All the details will be presented in the following section.

3.2. Physics Informed Neural Networks working algorithm

As it was discussed in subsection (2.2.3), traditional ANNs exploit the backpropagation algorithm to compute the derivatives of the loss function with respect to the network weights. In the same way, the backpropagation algorithm can be used to compute other derivatives, i.e. the derivative of the output with respect to the network inputs.

Let us now consider a generic parameterized PDE for the solution $u(\boldsymbol{x})$, with $\boldsymbol{x} = (x_1, x_2, ..., x_d)$ defined over a domain $\Omega \subset \mathbb{R}^d$:

$$f\left(\boldsymbol{x}; u(\boldsymbol{x}); \frac{\partial u}{\partial x_1}, ..., \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 u}{\partial x_1 \partial x_d}; ...; \boldsymbol{\mu}\right) = 0, \tag{3.1}$$

where $\boldsymbol{\mu}$ is the array of the parameters. Moreover, let us consider any generic boundary condition (Dirichlet, Neumann, Robin, or any other kind of boundary condition operator):

$$B(u(\boldsymbol{x}), \boldsymbol{x}) = 0 \quad \text{on } \delta\Omega.$$

The Physics Informed Neural Networks are made up of a traditional neural network architecture. The input layer is fed with the spatio-temporal coordinates array $\boldsymbol{x}$. In the case of a parameterized PDE, the parameters are treated as coordinates along additional domain dimensions. The output layer delivers a surrogate of the solution $\hat{u}(\boldsymbol{x})$ and thus it is an array with the same dimensionality of $u(\boldsymbol{x})$.

In addition to a classical ANN, a PINN presents a backpropagation block in which the derivatives of the output are computed with respect to the input layer variables. In such manner it is possible to force the network to comply with the constraints imposed by both the PDE and its initial & boundary conditions (ICs and BCs). Note that in PINNs the initial boundary conditions for time-dependent problems are treated in the same way as

the boundary conditions, since time is just another coordinate.

To measure the error of the network, the surrogate of the solution and its derivatives with respect to the inputs are evaluated on a number of points, named collocation points. The collocation points can be defined by the user as well as being uniformly or randomly sampled. In general, various techniques can be chosen to perform the random sampling, such as pseudo-random, latin hypercube sampling, Halton sequence, Hammersley sequence or Sobol sequence.

The values obtained are then substituted inside the PDE and BC-IC definitions to retrieve their residuals. The loss function to be minimized is usually constructed as the weighted sum of the L^2-norm of the residuals:

$$\mathcal{L}(\boldsymbol{w};\mathcal{T}) = \alpha_f \mathcal{L}_f(\boldsymbol{w};\mathcal{T}_f) + \alpha_b \mathcal{L}_b(\boldsymbol{w};\mathcal{T}_b), \tag{3.2}$$

where $\boldsymbol{w}$ are the weights of the network and $\mathcal{T}$ is the set of collocation points, divided into the ones inside the domain to test the PDE residual ($\mathcal{T}_f$) and the ones on the boundaries to test the boundary conditions ($\mathcal{T}_b$). Moreover:

$$\begin{aligned}
\mathcal{L}_f(\boldsymbol{w};\mathcal{T}_f) &= \frac{1}{|\mathcal{T}_f|}\sum_{\boldsymbol{x}\in\mathcal{T}_f}\left\| f\left(\boldsymbol{x};\hat{u}(\boldsymbol{x});\frac{\partial\hat{u}}{\partial x_1},...,\frac{\partial\hat{u}}{\partial x_d};\frac{\partial^2\hat{u}}{\partial x_1\partial x_1},...,\frac{\partial^2\hat{u}}{\partial x_1\partial x_d};...;\boldsymbol{\lambda}\right)\right\|_2^2, \\
\mathcal{L}_b(\boldsymbol{w};\mathcal{T}_b) &= \frac{1}{|\mathcal{T}_b|}\sum_{\boldsymbol{x}\in\mathcal{T}_b}\|B(\hat{u}(\boldsymbol{x}),\boldsymbol{x})\|_2^2,
\end{aligned} \tag{3.3}$$

and α_f and α_b are the respective weighting coefficients. As concern the latter, in general they are chosen such that the two terms become of the same order of magnitude. Recently, adaptive self-balancing weighting techniques such as the one described in [40] have been developed.

The final step consists into minimize the Equation (3.2) via an optimizer (such the ones mentioned in subsection (2.2.5)) in order to determine the optimal weights $\boldsymbol{w}^*$ of the network. The most widespread optimizers in the PINNs context are Adam and L-BFGS-B. In many cases the training is initially performed via Adam to "warm-up" the weights and then continued switching to the second-order optimizer L-BFGS, which in general is more efficient. The number of epochs chosen for the training and the number of collocation points are hyperparameters of the process and they are highly problem-dependent.

The complete architecture of a PINN can be visualized in Figure (3.1).

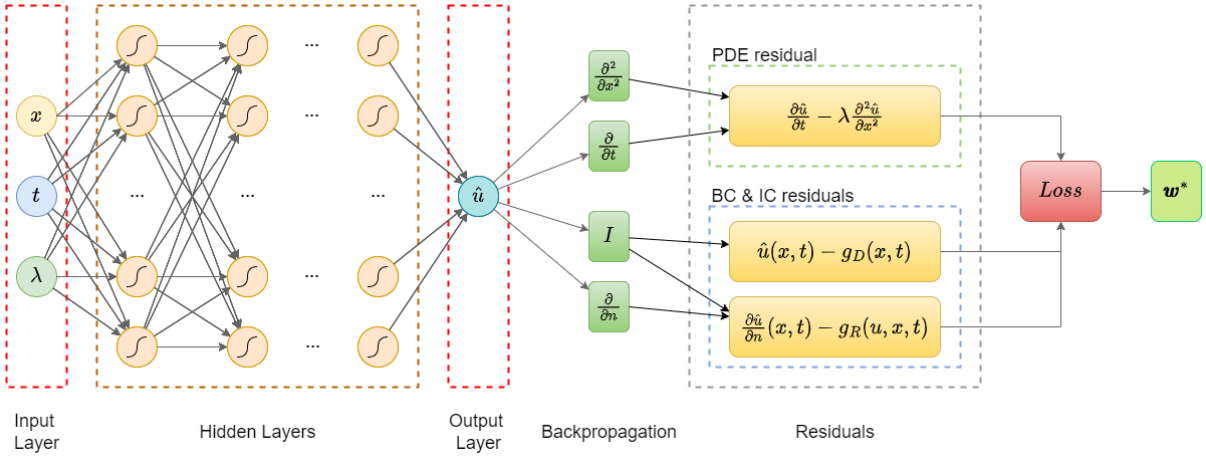


Figure 3.1: Example of a Physics Informed Neural Network Architecture for solving the 1D diffusion PDE $\frac{\partial u}{\partial t} = \lambda \frac{\partial^2 u}{\partial x^2}$ with mixed Dirichlet - Robin BC.

3.3. Practical example: 1D viscous Burgers equation

The viscous Burgers equation is a quasi-linear time-dependent hyperbolic PDE of great importance in the fluid dynamics and gas dynamics context.

In particular, it combines the nonlinear behaviour of the wave motion with a linear diffusion. The presence of the latter guarantees a smooth solution, avoiding the development of discontinuities in the form of shock waves.

Let us consider the following problem setup:

$$\begin{cases} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}, & x \in [-1,1], \quad t \in [0,1], \\ u(-1,t) = u(1,t) = 0, \\ u(x,0) = -sin(\pi x), \end{cases} \tag{3.4}$$

where the viscous burgers equation is constrained by an homogeneous Dirichlet boundary condition and by the initial condition.

The kinematic viscosity is assumed to be $\nu = 0.01$. The reference solution is plotted in Figure (3.2).

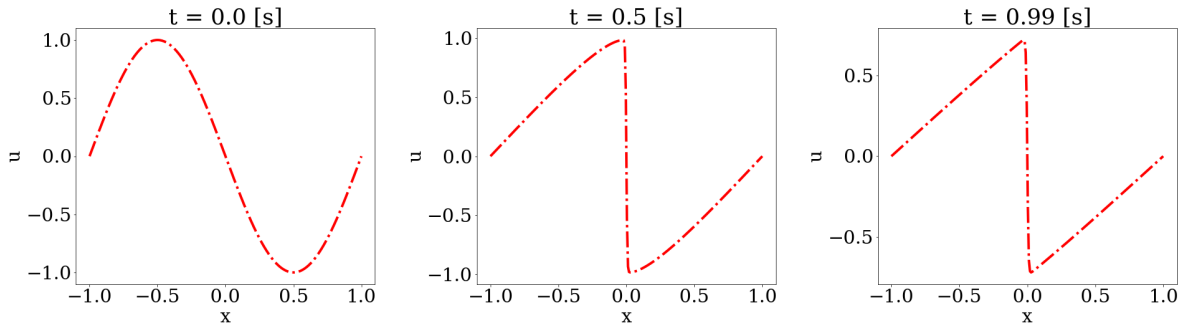


Figure 3.2: Viscous Burgers equation reference solution respectively at times t = 0.0, 0.5 and 0.99 [s].

The PINN solution is obtained by using DeepXDE, a python library for scientific machine learning and physics-informed learning. In the following section more details about it will be presented.

3.3.1. DeepXDE: an overview on the PINN library

DeepXDE has been firstly developed and proposed by Lu et al. in [30]. It consists of a very user-friendly library which allows the user to define the problem in a way it closely resembles the mathematical formulation. It works on top of several of the most widespread deep learning backends, such as:

- Tensorflow 1.x,
- Tensorflow 2.x,
- Pytorch,
- Paddle.
- JAX,

As concern PINNs, it can be used for solving the forward / inverse problems governed by:

- PDEs,
- ODEs,
- IDEs,
- Fractional PDEs,
- Stochastic PDES,

as well as performing function approximation.

The workflow of DeepXDE for solving a classic PDE problem is described in Algorithm 3.1

Algorithm 3.1 DeepXDE workflow

1: Specify the computational domain.
2: Define a function which returns the residual of the PDE
3: Specify the boundary and initial conditions.
4: Combine all the above in a `'data.PDE'` or `'data.TimePDE'` object, depending on weather the problem is time independent or not. Specify how the collocation points are sampled.
5: Build a neural network.
6: Define a model by combining all the information specified with the neural network. Set the optimization hyperparameters to be used such as the learning rate.
7: Train the model.
8: Make predictions and post-process the output to visualize the results.

3.3.2. Results

There are many hyperparameters that affect the network choice and training. Since testing different combinations of all of them is practically impossible, some of them will be considered fixed, whereas the others will vary at each iteration.

Specifically, in this example the hyperparameters that will be left fixed are:

- optimizers: Adam and LBFGS-B are used in sequence for 1.5e04 epochs each. This choice is common among the PINN training techniques and the number of epochs chosen is a good trade-off between loss function minimization and amount of time required;
- learning rate: Adam requires to be initialized with a learning rate lr. In this example a learning rate $lr = 0.001$ has been chosen.

The first hyperparameter to be tested is the number of collocation points on which the PDE residual and boundary conditions are evaluated. For each case a certain number of collocation points are sampled using a pseudo-random generator, as shown for example in Figure (3.3).

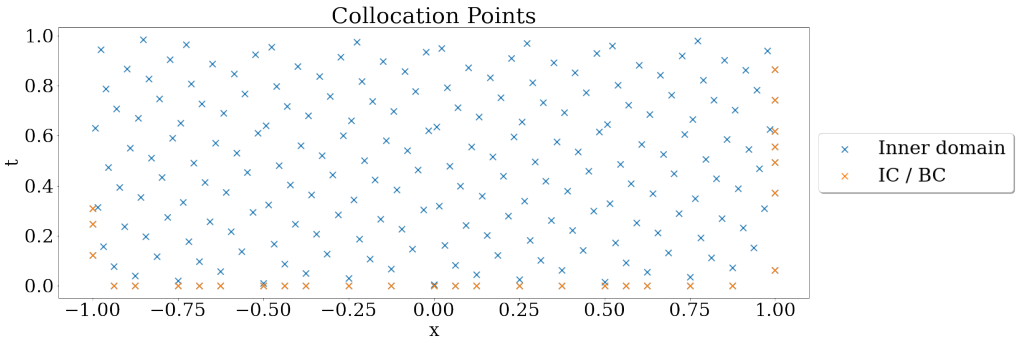


Figure 3.3: Pseudo-random sampled collocation points. In this specific case there are 200 points for the inner domain, 10 on the boundaries and 20 for the initial condition.

For this initial experiment a network of 3 hidden layers with 3 neurons each is employed. All the neurons are characterized by an hyperbolic tangent activation function.

The accuracy of the predictions is evaluated by computing the relative L_2 error between the true solution and the predicted one:

$$e_{l_2} = \frac{\|u_{pred} - u_{true}\|}{\|u_{true}\|}. \tag{3.5}$$

The results of this first analysis are reported in Table (3.1). It is possible to appreciate that for the problem considered, sampling more than 2000 points in the inner domain guarantees to achieve a relative error lower than 3%, even less than 1% with a number of points greater than 2600, which are already very satisfying results.

Given the fact that 3000 samplings in the domain gave the best accuracy, the next analysis has been carried out with that amount of collocation points. In particular, different activation functions has been tested. The results obtained are reported in Table (3.2). The highest relative error is achieved through the ReLU activation function which is known to perform poorly on PINN tasks, due to its characteristics such as the piece-wise constant first derivative and the null second derivative. The little accurate predictions of the ReLU network configuration can be appreciated in Figure (3.4).

Inner domain	BC	IC	e_{l_2}
200	10	20	1.18e-01
400	20	40	1.95e-01
600	30	60	1.30e-01
800	40	80	1.40e-01
1000	50	100	5.23e-02
1200	60	120	8.92e-02
1400	70	140	1.03e-01
1600	80	160	1.14e-01
1800	90	180	1.12e-02
2000	100	200	2.29e-02
2200	110	220	4.85e-03
2400	120	240	1.48e-02
2600	130	260	4.91e-03
2800	140	280	6.56e-03
3000	150	300	3.59e-03

Table 3.1: PINN accuracy vs amount of collocation points

Activation Function	e_{l_2}
tanh	3.59e-03
sigmoid	2.02e-03
relu	6.50e-01
swish	2.47e-01
sin	1.08e-02
LAAF-sigmoid	5.10e-03

Table 3.2: PINN accuracy vs activation functions

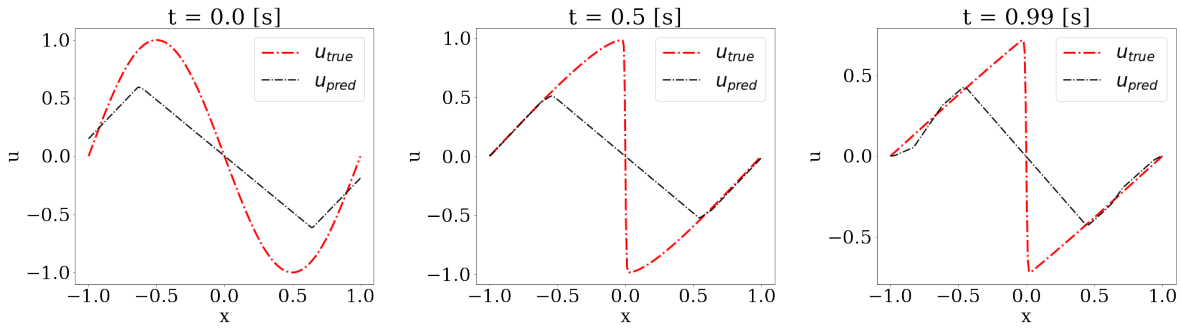


Figure 3.4: Viscous Burgers Equation. Neural network predictions with *relu* activation function vs reference solution.

Apart from the swish activation, which can be seen as the smooth non-monotonic counterpart of the ReLU, all the other activations perform definitely better. The best result is achieved by employing the sigmoid activation function, with a L_2 relative error lower than the one obtained before with the hyperbolic tangent. Keeping this in mind, the last attempt to improve the accuracy of the predictions has been to implement the layer-wise locally adaptive activation function technique proposed in [19] with a scaling factor $n = 10$. As it can be appreciated in Table (3.2), in this specific case this technique did not improve the accuracy of the results.

It is also interesting to have a look at the training process. In Figure (3.5) it is possible to look at the minimization of the PINN loss function through the various epochs, for each activation function. The ReLU case has been omitted because it was too poor.

Analyzing Figure (3.5) allows to derive several interesting conclusions. First of all, it is clear the great benefit of the proposed training strategy. In fact, after the first $1.5e04$ epochs with the Adam optimizer, switching to the L-BFGS-B increased immediately the minimization capabilities of the training process. This is valid for each activation function used, and even more evident for the ones which performed better (sigmoid, tanh and LAAF-sigmoid).

Moreover, it is also clear that there is no a $1:1$ correspondence between the final loss function value and the L_2 relative error, since it is necessary to take into account for the generalisation error of the network to unseen collocation points. In general it is still a good proxy though, since sigmoid, tanh and LAAF-sigmoid, which perform similarly very well, have a final loss function clustered in the same order of magnitude, whereas swish and sine activation functions lead to a much greater final loss, and a high L_2 relative error too.

For this reason, it is also possible to say that, in the case of low amount of time available for the training phase, tanh and especially LAAF-sigmoid are the most suitable activations because their loss functions decay much faster compared with the others.

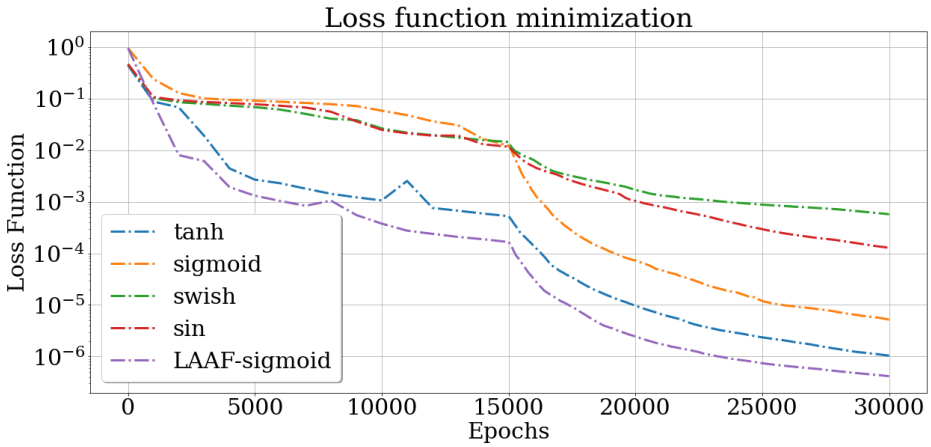


Figure 3.5: Neural network training with different activation functions

The last analysis conducted was about investigating the relationship between the number of hidden layers and their width with the prediction accuracy, in order to check if a deeper network leads to an improvement in the accuracy of the solution. The results have been obtained using 3000 inner domain collocation points and a sigmoid activation function, since those where the optimal choices from the previous tests.

Neurons \ Layers	2	3	4	5	6
20	2.15e-01	5.47e-03	2.94e-03	7.80e-04	3.27e-03
30	5.89e-02	2.02e-03	3.89e-03	1.13e-03	4.93e-04
40	8.21e-02	5.49e-03	4.81e-04	3.86e-03	1.91e-02
50	8.37e-02	4.17e-02	1.37e-03	1.21e-02	6.64e-03
60	4.57e-02	9.75e-04	6.80e-03	7.05e-03	1.41e-01

Table 3.3: L_2 relative error e_{l_2} for different network configutations.

As it can be observed in Table (3.3), a deeper and wider network architecture does not necessarily lead to an increase in accuracy. It is possible to conclude that a network with 2 hidden layers is not enough deep to achieve the desired accuracy, but increasing the

depth is not always a good strategy, as demonstrated by the 6 hidden layers, 60 neurons per layer case which surprisingly lead to one of the most inaccurate result.

The best result has been obtained with the following hyperparameters:

- 3000 collocations points in the inner domain, 150 for the boundary conditions and 300 for the initial condition;
- sigmoid activation function;
- 4 hidden layers with 40 neurons per layer.

Since the number of collocation points was quite small, the network architecture was quite shallow and the PDE didn't require many derivatives to be computed, it took a short time to train the network, just 157$[s]$. For more complex problems, such as in the case of 2D parameterized PDEs, the time may become an important factor as well, and thus the hyperparameters tuning may require to consider also the training time needed for choosing the best combination.

The comparison between the reference solution and the most accurate result achieved is plotted in 3.6. As it can be seen, the two solutions are perfectly overlapped, and the L_2 relative error $e_{l_2} = 4.80e-04$ confirms the high level of precision reached.

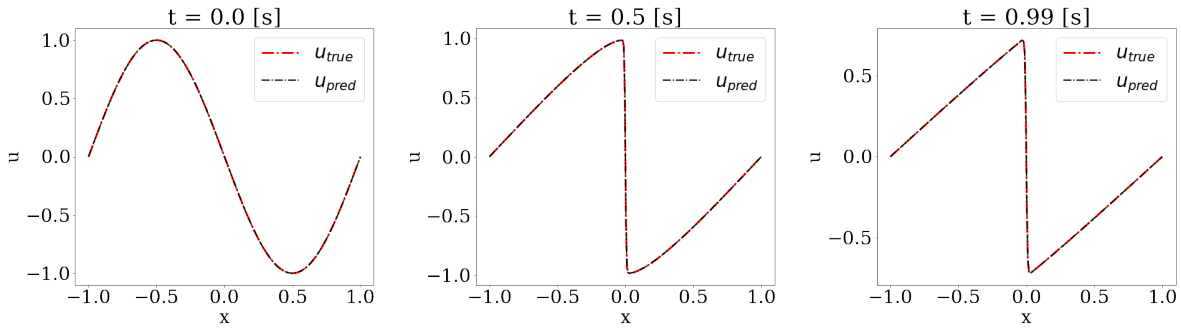


Figure 3.6: Viscous Burgers Equation. Comparison between the most accurate solution and the reference one. Hyperparameters: 3000 inner domain collocation points, *sigmoid* activation functions, 4 hidden layers of 40 neurons each.

3.4. An overview on the PINNs advanced techniques

After the first paper made by Raissi on the topic [34], the community of researchers showed a growing interest in PINNs. Nowadays, tens of new peer-reviewed paper are being published every week. While a part of them concerns the application of these

networks to a specific branch of physics, some other come with new techniques to enhance their performances. In the following subsections the most relevant ones (to the author's knowledge) will be presented. Some of them will be exploited in the present work.

3.4.1. Hard boundary conditions

As discussed in section (3.2), the compliance of the network output with the boundary conditions of the problem is imposed in a soft way and the corresponding loss is added to the PDE residual one.

This strategy may lead to a lack of accuracy in the network output, since it is practically impossible to minimize the loss until it reaches a null value. The area of the domain close to the boundaries, moreover, is of primary relevance for many physical problems (e.g. CFD ones). To avoid any source of error which inevitably would affect the results in the region, it is possible to impose the boundary conditions in the hard way.

In principle, this is valid for all the kind of boundary conditions, but is limited to the Dirichlet and Periodic ones, when feasible.

Imposing the Dirichlet boundary conditions in the hard way is pretty straightforward if the geometry is simple. Given the neural network output $\mathrm{NN}(\boldsymbol{x}, t)$, the boundary condition is imposed in a hard way as:

$$\hat{u}(\boldsymbol{x}, t) = l(\boldsymbol{x}) \cdot \mathrm{NN}(\boldsymbol{x}, t) + g(\boldsymbol{x}), \tag{3.6}$$

where $\hat{u}(\boldsymbol{x}, t)$ is the modified output of the network compliant with the specified boundary condition and $l(\boldsymbol{x})$ can be chosen analytically if the geometry is simple, otherwise an approximation through spline functions may be required.

Let's consider an example, in which the solution $u(\boldsymbol{x}, t)$ is defined over the spatio-temporal domain $\Omega = [0, 1] \times [0, T]$ and the boundary condition is given by:

$$\begin{aligned} u(0, t) &= \gamma_0, \\ u(1, t) &= \gamma_1. \end{aligned} \tag{3.7}$$

The boundary condition prescribed in Equation 3.7 is imposed as:

$$\hat{u}(\boldsymbol{x}, t) = [\boldsymbol{x} \cdot (1 - \boldsymbol{x})] \cdot \mathrm{NN}(\boldsymbol{x}, t) + [\gamma_0 + \boldsymbol{x} \cdot (\gamma_1 - \gamma_0)]. \tag{3.8}$$

Note that $l(\boldsymbol{x})$ has been chosen as $\boldsymbol{x} \cdot (1 - \boldsymbol{x})$, but this is just a possibility. For example, $l(\boldsymbol{x}) = (1 - e^{-\boldsymbol{x}}) \left(1 - e^{(1-\boldsymbol{x})}\right)$ is another valid option.

In the same way it is possible to exploit the a-priori knowledge of the solution behaviour enforcing some additional constraints. For example, it is possible to enforce the neural network prediction to be strictly positive by taking the square of the output value.

3.4.2. Physical activation functions

In order to improve the prediction accuracy of a PINN without attempting to increase its size, it is possible to consider a further physics induction method in the algorithm. More specifically, in [1] the concept of physical activation functions (PAF) is presented. In the mentioned work it is reported that using an activation function which can be derived by the problem at hand may significantly speed up the convergence of the training process, leading to loss values which are smaller than the ones obtained via traditional activation functions described in subsection (2.2.1), even by 1 or 2 orders of magnitude.

The definition of the optimal PAF to be used is problem dependent and must be performed after a careful analysis of the governing PDEs. For example, it can be done by considering the analytical solution to similar PDEs in simple cases when it is available.

3.4.3. Expanding layer

Paper [7] introduces more improvements to the traditional PINNs architecture. The first one to be discussed in this section is the inclusion of an expanding layer which enhance the information given by the spatio-temporal coordinates by realizing the following map:

$$(\boldsymbol{x}, t) \mapsto [pow(\boldsymbol{x}, t), cos(\boldsymbol{x}, t), sin(\boldsymbol{x}, t), (\boldsymbol{x}, t)] \tag{3.9}$$

This map can be stacked twice or even more times to enlarge the number of learnable parameters in the first portion of the network. The transformation may allow to better catch the nonlinear behaviours of the problem at hand.

3.4.4. Attention blocks

A further improvement suggested in [7] is to get rid of the traditional vanilla FFNN and employ a more sophisticated architecture. In this context, the hidden layer of the network are still fully connected, but the importance of each neuron is modulated by an excitation block which implements the self-gating attention mechanism.

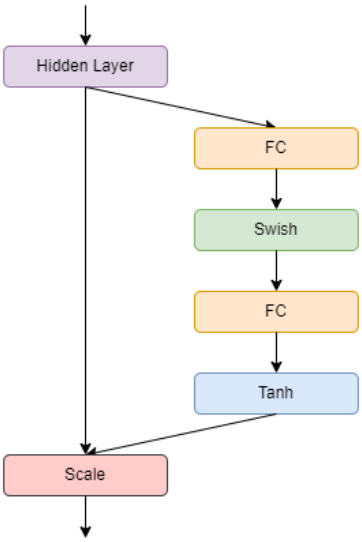


Figure 3.7: Excitation block visualization

The idea behind an excitation block is inspired by the work in [18], where the so called squeeze-and-excitation blocks are first introduced. The squeezing part of the algorithm is ignored since it is was thought for dealing with computer vision tasks, whereas the excitation block is placed between each hidden layer. As it can be observed in Figure (3.7), the excitation block consists of two fully connected layers with nonlinear activation functions (swish and tanh in this case) which get as input the output of the previous hidden layer. The output of the block will be the multiplication between the previous hidden layer values and the channel weights which come from the two fully connected layers in between. The number of neurons per fully connected layer, as well as the type of activation function employed, are an hyperparameter of the attention mechanism.

4 | Strategies and Neural Network architectures adopted

In this chapter we will discuss the details of the strategies and neural networks architectures used to answer the need of a reliable, fast and accurate solution approximation technique suitable for real time applications.

The adopted strategies can be divided into two main branches:

- the first one consists of employing a PINN directly on the governing PDEs, treating the parameter(s) as additional coordinates of the computational domains;
- the second one tackles the problem by first reducing its dimensionality and eventually it makes use of a Neural Network for fast predictions.

The first approach is the simplest and more direct one, since the whole process of deriving a ROM is skipped. Moreover, the usage of the DeepXDE library [30] significantly eases the coding effort from the programmer side.

The construction of a ROM is instead a more complex procedure because its offline stage involves multiple steps:

- coding and solving the FOM over a range of parameter(s) values, in order to obtain the *snapshots* matrix;
- reducing the dimensionality of the FOM. This can be achieved by the use of both linear (SVD, see subsection (1.2.1)) and non-linear (autoencoders, see section (2.4)) tools, as it will be discussed later in detail;
- retrieving the approximation of the high-fidelity solution, possibly through methods that allow a fast online evaluation.

Given these premises, both of these two different approaches have been applied on different test problems in order to evaluate their performances. The results will be later discussed in chapter (5).

4.1. PINNs - FOM

The main drawback of PINNs, to date, is the high amount of time they require for their training phase. In fact, they need to be employed on physical problems governed by *parameterized* ODEs or PDEs to become competitive against the traditional numerical methods. In this way, the time spent is justified by the vast set of solutions which can be instantly predicted by the network.

Setting up the parameterized problem is straightforward. With respect to a generic non-parameterized case, the only task to be performed is to treat the parameter as an additional input coordinate, just as the spatio-temporal ones. The consequent changes in the algorithm are:

- extension of the computational domain by taking into account the amount of parameters. For example, a time independent three-dimensional problem parameterized by one parameter will demand a four-dimensional *hypercube* inside which the residual points are going to be sampled;
- addition of as many input layer neurons as the number of parameters characterizing the governing equation.

Moreover, in general, since the dimensionality of the problem is increased, it is likely that the network width and depth, the number of residual points to be sampled and the training epochs required will be higher than the non-parameterized counterpart.

Apart from these important considerations, the rest of the algorithm is the same as a simple problem like the one described in section (3.3). The objective is still to seek a surrogate of the solution inside the domain of interest, and the way to achieve this result is to minimize the loss defined in Equation (3.2) by a gradient-based optimizer like the ones discussed in subsection (2.2.5).

The neural network architecture employed for this task can be simply a traditional FFNN with problem-dependent hyperparameters to be tuned. The DeepXDE library offers also the ResNet [16] architecture as a built-in option. In any case the user can specify its own custom architecture following the procedure required by the specific backend chosen.

4.2. Projection Driven Neural Networks

The Projection Driven Neural Networks (PDNNs, from now on) is the first algorithm to be analysed which falls into the second category of strategies presented above.

PDNNs were first proposed in [6] and they consists in a purely non-intrusive data-driven ROM approach. In particular, PDNNs are built and trained in order to obtain a continuous map between samples of the parameter space and the corresponding projection coefficients of the ROM.

The offline stage starts with the collection of the FOM problem snapshots $\boldsymbol{u}_h(t, \boldsymbol{\mu})$ sampled at different values of the parameters $\boldsymbol{\mu}$ and time t. In the present work the Finite Element Method (FEM) has been implemented to compute the high-fidelity solution. To this aim, the python library FEniCS [26] has been exploited.

The parameters such as the mesh size and the polynomial degree of the FEM basis are function of the specific problem complexity and peculiarities.

From now on, two different strategies can be adopted in the PDNN context: the linear classical one, which exploits the SVD, and the proposed nonlinear counterpart, which makes use of a deep convolutional autoencoder neural network. Their characteristics and algorithms are both going to be described in the following subsections.

4.2.1. PDNNs - SVD

First of all the collected snapshots are flattened and arranged to form the snapshots matrix $\boldsymbol{A} \in \mathbb{R}^{n \times m}$, where n is the number of collocation points of the high fidelity solution and m is the number of available snapshots. In general, $n >> m$.

Subsequently, the SVD of $\boldsymbol{A}$ is performed to obtain the left singular vectors matrix $\boldsymbol{U}$, the singular values matrix $\boldsymbol{\Sigma}$ and the right singular vectors matrix $\boldsymbol{V}$.

The analysis of $\boldsymbol{\Sigma}$ gives an indication on how many modes r should be kept to capture enough energy of the problem at hand. The matrix $\boldsymbol{U}$ is finally truncated to the first r dominant modes, to obtain $\tilde{\boldsymbol{U}} \in \mathbb{R}^{n \times r}$.

Retrieving the projection coefficients to be used for the network training in the supervised learning context is then straightforward. Recalling Equation (1.10), and given the fact that $\tilde{\boldsymbol{U}}$ is orthogonal, i.e., $\tilde{\boldsymbol{U}}^T \tilde{\boldsymbol{U}} = \boldsymbol{I}$, for each value of $\boldsymbol{\mu}$ and t the projection coefficients $\boldsymbol{\alpha}(t, \boldsymbol{\mu})$ are given by:

$$\boldsymbol{\alpha}(t, \boldsymbol{\mu}) = \tilde{\boldsymbol{U}}^T (\boldsymbol{u}_h(t, \boldsymbol{\mu}) - \overline{\boldsymbol{u}}), \tag{4.1}$$

with $\overline{\boldsymbol{u}}$ being the average of $\boldsymbol{A}$ to avoid the presence of dominant basis.

Finally, the data preprocessing stage can be finalised by assembling the training data

set $\mathcal{D}$ of dimension N_D, associating to each parameter $\boldsymbol{\mu}$ and time t its corresponding projection coefficients array $\boldsymbol{\alpha}(t, \boldsymbol{\mu})$.

The idea of the PDNN is to feed an artificial neural network with the obtained data set, in order to learn the map $\boldsymbol{f} : \mathbb{R}^{N_p+k} \to \mathbb{R}^r$ between each time-parameter instance and the corresponding projection coefficients. Note that k is 1 if the problem at hand is time-dependent, zero otherwise. An ANN tool is chosen for both its accuracy as interpolation method as well as its fast real-time evaluation capability.

The loss function to be minimized is defined as the MSE between the PDNN prediction and the actual projection coefficients:

$$\mathcal{L}_{PDNN}(\boldsymbol{w}; \mathcal{D}) = \frac{1}{N_D} \sum_{(t,\boldsymbol{\mu}),\boldsymbol{\alpha}\in\mathcal{D}} \|\boldsymbol{f}(t, \boldsymbol{\mu}) - \boldsymbol{\alpha}(t, \boldsymbol{\mu})\|_2^2 , \tag{4.2}$$

where $\boldsymbol{w}$ are the network weights to be optimized.

In the present work the PDNN has been built using the DeepXDE library, exploiting its function approximation package.

Once the training is finished, the approximated solution is immediately available for any given time-parameter instance:

$$\hat{\boldsymbol{u}}_h(t, \boldsymbol{\mu}) = \tilde{\boldsymbol{U}} \boldsymbol{f}(t, \boldsymbol{\mu}) + \overline{\boldsymbol{u}} \tag{4.3}$$

,

where $\boldsymbol{f}(t, \boldsymbol{\mu})$ is the output of the PDNN.

4.2.2. PDNNs - Autoencoder

An autoencoder neural network, as discussed in subsection (2.4.1), can be seen as the nonlinear counterpart of the SVD transformation. To briefly sum up the description of its architecture, it is made up by two sub-networks: the encoder $E(\boldsymbol{u}_h(t, \boldsymbol{\mu}))$ and the decoder $D(E(\boldsymbol{u}_h(t, \boldsymbol{\mu})))$. The encoder maps the high-dimensional input into a low-rank r latent representation, whereas the decoder learns the inverse transformation to reconstruct the original data. The training is performed in a semi-supervised paradigm, given the fact that the input of the network coincide with the required output.

Since the dimensionality reduction is performed in a nonlinear context, this approach is expected to perform better than the one relying on the SVD when:

- the snapshots matrix $\boldsymbol{A}$ is characterized by misaligned data or the problem presents non-linearities;
- the reduced trial manifold dimension r is close to 1.

In the present work, the autoencoder has been trained for each test problem with the same amount of snapshots collected for the PDNN-SVD case. The snapshots go through a preprocessing phase in which they are standardized. In this way the training process gains a speed-up and the accuracy of the reconstructed snapshots increases.

The autoencoder architecture is a mixed convolutional - fully connected one. The first layers are two-dimensional and of convolutional type, hence the snapshots are not flattened but they are left as two-dimensional square arrays, if the problem at hand is two-dimensional in space. If the problem is mono-dimensional in space, the snapshots are conveniently reshaped. More details of the chosen autoencoder can be appreciated in Figure (4.1) and in Table (A.1).

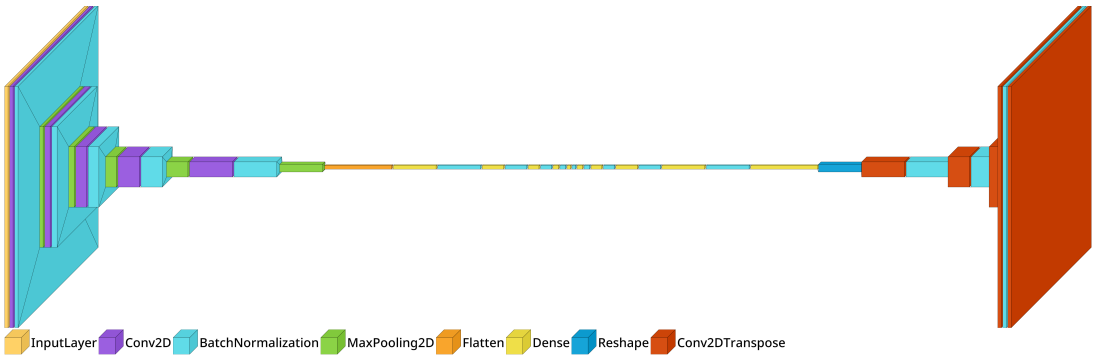


Figure 4.1: Autoencoder architecture details. The plot has been realized via the VisualKeras python library [12]

The hyperparameter trial and error tuning process of the autoencoder required a lot of effort and thus it must be noted that the development of a SVD-based dimensionality reduction method can be certainly faster, especially for beginners in the deep learning field.

The loss function to be minimized during the autoencoder training is given by:

$$\mathcal{L}_{AE}(\boldsymbol{w};\boldsymbol{A}) = \frac{1}{m} \sum_{\boldsymbol{u}_h(t,\boldsymbol{\mu})\in\boldsymbol{A}} \|D(E(\boldsymbol{u}_h(t,\boldsymbol{\mu}))) - \boldsymbol{u}_h(t,\boldsymbol{\mu})\|_2^2 \,, \tag{4.4}$$

where we recall that m corresponds to the amount of available snapshots.

If the amount of snapshots is enough, the collection is split into a training data set and a validation data set through which it is easy to monitor and possibly take actions to mitigate the overfitting issue. In any case, the *weight decay* overfitting prevention technique (see subsection (2.2.7)) is applied to each layer with an activation function. The hyperparameter γ has been fixed to $\gamma = 1e-8$.

Once the autoencoder is successfully trained, the encoder portion is fed with the snapshots corresponding to the sampled time-parameter instances and it outputs their latent representations.

In this way it is possible to construct a dataset $\overline{\mathcal{D}}$ of dimension $N_{\overline{D}}$ for the training of the associated PDNN, which consists of the time-parameters instances along with their corresponding latent representation.

This dataset is finally used to train the PDNN in a supervised learning context in order to learn the map $\boldsymbol{g} : \mathbb{R}^{N_p+k} \to \mathbb{R}^r$ between the time-parameters space and the latent representation one, where again k is 1 if the problem considered is time-dependent, zero otherwise.

The loss to be minimized in order to optimize the PDNN weights is:

$$\mathcal{L}_{PDNN}(\boldsymbol{w}; \overline{\mathcal{D}}) = \frac{1}{N_{\overline{D}}} \sum_{(t,\boldsymbol{\mu}),E(\cdot)\in\overline{\mathcal{D}}} \|\boldsymbol{g}(t,\boldsymbol{\mu}) - E(\boldsymbol{u}_h(t,\boldsymbol{\mu}))\|_2^2 \,. \tag{4.5}$$

In the present work the two processes have been performed sequentially, but it is possible to train the two networks at the same time too by building a loss which takes into account both the reconstruction error of the snapshots and the distance between the two latent representations.

The online stage (see Figure (4.2)) is performed by evaluating the PDNN at the desired time-parameter instance and feeding the corresponding latent representation to the decoder which in turns outputs the ROM solution:

$$\hat{\boldsymbol{u}}_h(t,\mu) = D(\boldsymbol{g}(t,\boldsymbol{\mu})). \tag{4.6}$$

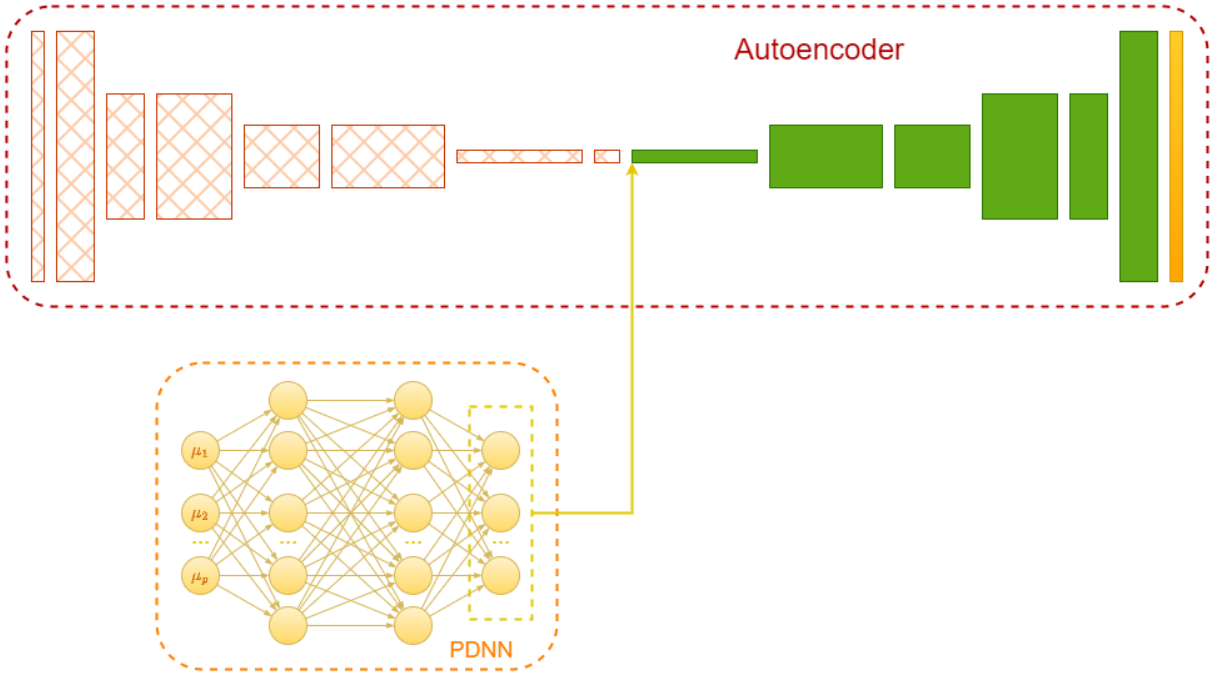


Figure 4.2: PDNN - Autoencoder online stage. Note that the encoder is not being used in this process.

4.3. POD-G-NN

While the previous reduced order model approaches were entirely based on data, this intrusive one relies on physics. In fact, in this strategy, a neural network is trained to learn the continuous interpolation of the reduced coefficients obtained from the POD-G system of equations.

The collection of the high fidelity solution snapshots follows the same procedure as seen for PDNNs. Once the snapshots matrix $\boldsymbol{A}$ is obtained, its SVD is performed and thus the left singular vectors matrix $\boldsymbol{U}$ is retrieved. During this phase, the matrices and arrays coming from the FEM discretization are stored too.

The number r of dominant modes is decided by means of a singular values analysis. Then, the $\boldsymbol{U}$ is truncated to obtain $\tilde{\boldsymbol{U}}$, i.e., the matrix through which the FOM system is projected onto the reduced order space. Recalling the approximation of the high-fidelity solution through the Galerkin expansion (see Equation (1.10)), the POD-G system of equations governing the ROM is assembled (Equation (1.14)) and solved to obtain the reduced coefficients array $\boldsymbol{\alpha}(t, \boldsymbol{\mu})$.

Finally, a neural network is built to map the time-parameter space to the corresponding reduced coefficients. In particular, the neural network map is defined as $\boldsymbol{h} : \mathbb{R}^{N_p+k} \to \mathbb{R}^r$,

where k is 1 in the time-dependent case, 0 otherwise.

The number of input neurons equals the number of parameters characterizing the FOM equation, plus possibly one neuron for the time coordinate in case of a time-dependent problem. The number of output neurons equals the number of unknowns, i.e., the dimension of the reduced coefficients array $\boldsymbol{\alpha}(t, \boldsymbol{\mu})$, which in turn corresponds to the rank r of the matrix $\tilde{\boldsymbol{U}}$.

A data set of collocation points $\mathcal{T}_f$ of dimension N_T is generated, containing the time-parameters instances along with their correspondent reduced coefficients.

The training is performed through a gradient-based optimizer by minimizing the following loss function:

$$\mathcal{L}_{\text{POD-G-NN}}(\boldsymbol{w}; \mathcal{T}) = \frac{1}{N_T} \sum_{(t,\boldsymbol{\mu}),\boldsymbol{\alpha} \in \mathcal{D}} \|\boldsymbol{h}(t, \boldsymbol{\mu}) - \boldsymbol{\alpha}(t, \boldsymbol{\mu})\|_2^2 , \tag{4.7}$$

In the present work the network has been built through the function approximation package of the DeepXDE library and it was just a simple FFNN, but as stated in section (4.1) the library is flexible and the programmer could develop its own custom architecture.

Once the training is completed, the approximated solution $\hat{\boldsymbol{u}}(t, \boldsymbol{\mu})$ is retrieved by plugging, for each value of parameters array $\boldsymbol{\mu}$ and time t, the output of the network, i.e., the approximated reduced coefficients array, into the Galerkin expansion equation:

$$\hat{\boldsymbol{u}}_h(t, \mu) = \tilde{\boldsymbol{U}} h(t, \mu) + \overline{\boldsymbol{u}} \tag{4.8}$$

Note that in this thesis the POD-G-NN method has been developed only for time-independent benchmark problems, but in principle it can be used also for the unsteady ones.

5 | Results and Comparisons

In the present section the reader will be provided with the results obtained by applying the proposed strategies to four parameterized PDE problems, namely the Poisson Equation (5.1), the Advection Diffusion Equation (5.2), the Pure Advection Equation (5.3) and the Viscous Burgers Equation (5.4). To evaluate the performances of the different strategies we rely on the L_2 relative error indicator, defined as:

$$e_{l_2} = \frac{1}{N_{test}} \sum_{i=1}^{N_{test}} \frac{\|\boldsymbol{u}_h(t_i, \boldsymbol{\mu}_i) - \hat{\boldsymbol{u}}(t_i, \boldsymbol{\mu}_i)\|_2}{\|\boldsymbol{u}_h(t_i, \boldsymbol{\mu}_i)\|_2} \tag{5.1}$$

over the test sets built for each case. To guarantee the repeatability of the results, a seed has been fixed for all the operations involving a degree of randomness, such as parameter sampling and neural network training. The code developed for the mentioned tests is freely available at `https://github.com/riccardotomada/Master-Thesis`.

5.1. Test 1: Poisson Equation

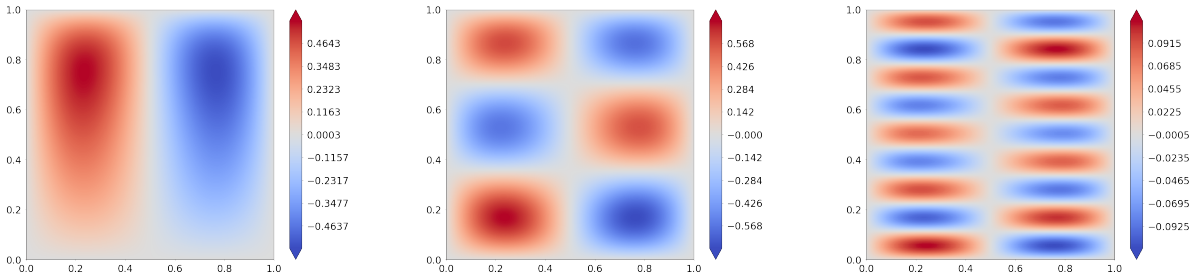
Let us consider the parameterized two-dimensional variable coefficient linear Poisson equation:

$$\begin{cases} -\nabla \cdot (k \nabla u) = f & \text{in } \Omega = (0,1)^2, \\ u = g_D & \text{on } \Gamma_D = \delta\Omega, \end{cases} \tag{5.2}$$

where:

$$\begin{aligned} k &= 1.0 + e^{-10\left((x-0.5)^2+(y-0.5)^2\right)}, \\ f &= 100 \sin(2\pi x) \sin(\mu \pi y), \\ g_D &= 0. \end{aligned} \tag{5.3}$$

System (5.2) has been discretized in space by means of quadratic finite elements, with



(a) (b) (c)

Figure 5.1: Test 1: FEM solution at different instances of the parameter μ. (5.1a) $\boldsymbol{u}_h$ for $\mu = 0.175$, (5.1b) $\boldsymbol{u}_h$ for $\mu = 2.849$, (5.1c) $\boldsymbol{u}_h$ for $\mu = 8.916$.

$N_h = 50 \times 50$ grid points. The parameter space, to which belongs the single μ parameter affecting the source term, is given by $\boldsymbol{M} = [0, 10]$. In Figure (5.1) it is possible to appreciate the behaviour of the FEM solution at different μ samples.

To develop the PDNNs-SVD and the PDNNs-Autoencoder strategies, a number of 80 snapshots ($\boldsymbol{u}_h(\boldsymbol{\mu}_{train})$) have been generated by randomly sampling the parameter with a uniform probability inside $\boldsymbol{M}$.

As concern the POD-G-NN approach, instead, 128 parameter instances have been generated in $\boldsymbol{M}$ again with uniform probability, and the corresponding ROM linear system of equation has been built and solved through a traditional linear algebra solver.

Finally, 10 parameter values have been sampled to later build a test set of high-fidelity benchmark solutions ($\boldsymbol{u}_h(\boldsymbol{\mu}_{test})$) used to evaluate and compare the different approaches through Equation (5.1).

5.1.1. PINNs-FOM

The computational domain has been set up as $D = (0, 0, 0) \times (1, 1, 10)$. The PDE residual collocation points data set $\mathcal{T}_f$ of dimension N_f and the boundary collocation points data set $\mathcal{T}_b$ of dimension N_b have been generated by randomly sampling a number of points respectively in the computational domain and on its boundaries.

The first test that has been performed is the evaluation of the solution accuracy with respect to the number of collocation points used. To this aim, the solution for $\mu = 10$, i.e., the most oscillatory one, has been compared to the correspondent high-fidelity result via Equation (5.1).

The network architecture employed was characterized by the following hyperparameters:

- hidden layers: 5 with 30 neurons each;
- activation functions: hyperbolic tangent (see Figure (2.2a));
- optimizer: Adam (see subsection (2.2.5)) for $1e04$ epochs with a learning rate $lr = 5e-04$ and L-BFGS (see subsection (2.2.5)) for $2e04$ epochs.

Moreover, the network presented a first expanding layer as described in subsection (3.4.3) and the boundary conditions on each edge were enforced in the hard way (more details in subsection (3.4.1)). Since the neural network output was forced to be identically zero on the boundaries, the number of collocation points N_b became irrelevant for the purpose of this analysis.

The results are reported in Table (5.1):

$\mathcal{T}_f$	e_{l_2}	Time [s]
2500	2.32e-01	278
5000	1.64e-01	264
10000	1.34e-01	323
15000	1.71e-01	379
20000	1.05e-01	418

Table 5.1: Test 1: PINNs-FOM L_2 relative error for different number of collocation points

As a trade off between the time needed and the accuracy of the results, the rest of the tests have been performed by using 10000 residual collocation points. In fact, it can be noticed that using 10000 collocation points guarantees the second best result, saving the 22% of the time needed for the 20000 points case.

The aim of the second test was to investigate the accuracy of the network while varying the activation functions employed in each hidden layer neuron. The other hyperparameters have been kept the same as the previous network ones. The accuracy of the model is evaluated again with respect to the high fidelity solution for a value of $\mu = 10$. The results are presented in Table (5.2).

The sine activation function is the one that leads to the lowest L_2 relative error. This is not surprising, since it can be seen as a physical activation function for the specific

Activation Function	e_{l_2}
$tanh$	1.34e-01
$sigmoid$	3.27e-01
$relu$	3.00e+00
$swish$	1.34e-01
sin	4.66e-02
$LAAF - sin$	5.04e-02

Table 5.2: Test 1: PINNs-FOM L_2 relative error for different activation functions

problem, as per [1]. On the contrary, in this case, the network with ReLU activation function fails the training phase, leading to a completely wrong solution.

The purpose of the third and last test was to investigate the relationship between the number and width of the hidden layers with the accuracy of the network output. The activation function chosen, following the previous analysis, was the sine function. The results are presented in Table (5.3).

L / N	3		4		5		6	
	e_{l_2}	Time [s]	e_{l_2}	Time [s]	e_{l_2}	Time [s]	e_{l_2}	Time [s]
20	1.31e-01	181	9.05e-02	213	7.85e-02	240	9.93e-02	265
30	5.45e-02	202	7.63e-02	238	4.66e-02	274	5.85e-02	304
40	2.90e-02	209	5.06e-02	248	4.02e-02	294	8.11e-02	328
50	4.48e-02	234	4.86e-02	279	2.39e-02	421	5.66e-02	582
60	4.79e-02	254	2.32e-02	406	3.81e-02	520	4.69e-02	564
70	4.22e-02	390	2.95e-02	511	3.62e-02	695	4.31e-02	789
80	2.56e-02	416	2.86e-02	533	2.09e-02	693	4.51e-02	795

Table 5.3: Test 1: PINNs-FOM L_2 relative error and training time for different number of hidden layers L and neurons per layer N.

From Table (5.3) it is possible to assess that increasing the number of hidden layers and their width does not necessarily mean that the accuracy of the prediction will improve.

To sum up, after these 3 tests we could conclude that the most effective model was

characterized by:

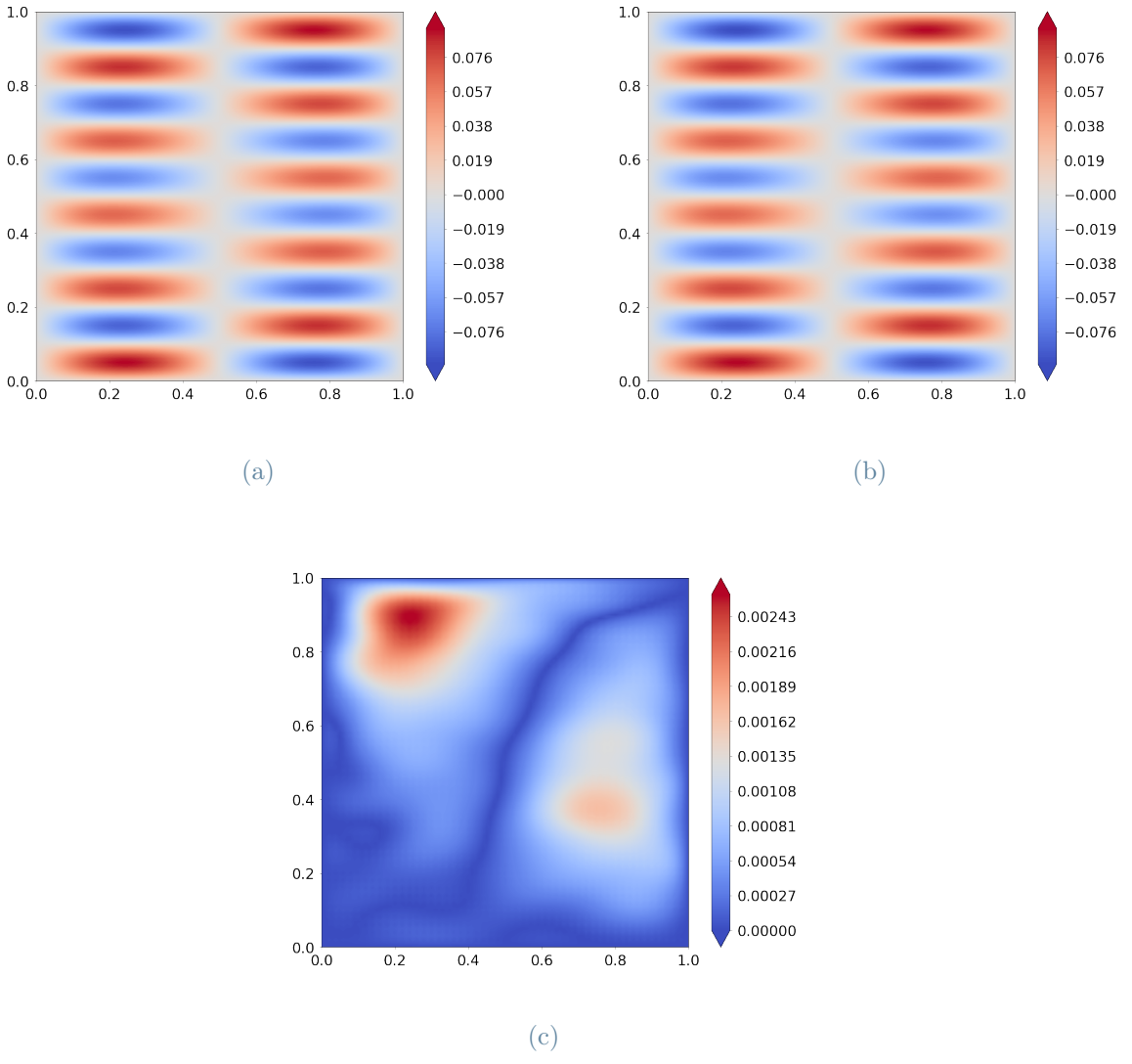
- 10000 residual collocation points;
- hidden layers: 4 with 60 neurons each;
- activation function: sine.

The reason behind this choice of the hidden layers parameters is the trade-off between accuracy (second best accuracy score) and training time (36% lower than the time needed for the best score case, the one with 5 hidden layers and 80 neurons per layer).

The solution for $\mu = 10$ is available in Figure (5.2), together with the FEM solution and the corresponding absolute error.

Finally, this network has been trained one more time and evaluated at 10 different test values of the parameter. The training has been performed for $1e04$ epochs with the Adam optimizer and $3.5e04$ epochs with the L-BFGS. The average of the L_2 relative error has been taken as a benchmark to compare this approach to the others.

The result obtained was $e_{l_2} = 1.32e - 03$. It was expected to be lower than the case for $\mu = 10$ due to both the increasing number of training epochs of the network and the behaviour of the solutions for $\mu < 10$, which necessarily present less oscillations as the number of oscillations in the y direction is controlled by the value of μ.



(a) (b)

(c)

Figure 5.2: Test 1: Comparison of the PINNs-FOM solution with the FEM solution. (5.2a) the FEM solution $\boldsymbol{u}_h$, (5.2b) the PINNs-FOM solution $\hat{\boldsymbol{u}}$, (5.2c) the absolute error $|\boldsymbol{u}_h - \hat{\boldsymbol{u}}|$.

5.1.2. PDNNs-SVD, POD-G-NN & PDNNs-Autoencoder

Problem (5.2) does not represent a remarkably challenging task for linear ROM approaches, such as the PDNNs-SVD and the POD-G-NN. Indeed, by performing the SVD on the snapshot matrix it is found that a linear trial manifold of dimension $r = 10$ is capable to capture more than the 99.99% of the energy of the system. The Singular Value decay is showed in Figure (5.3).

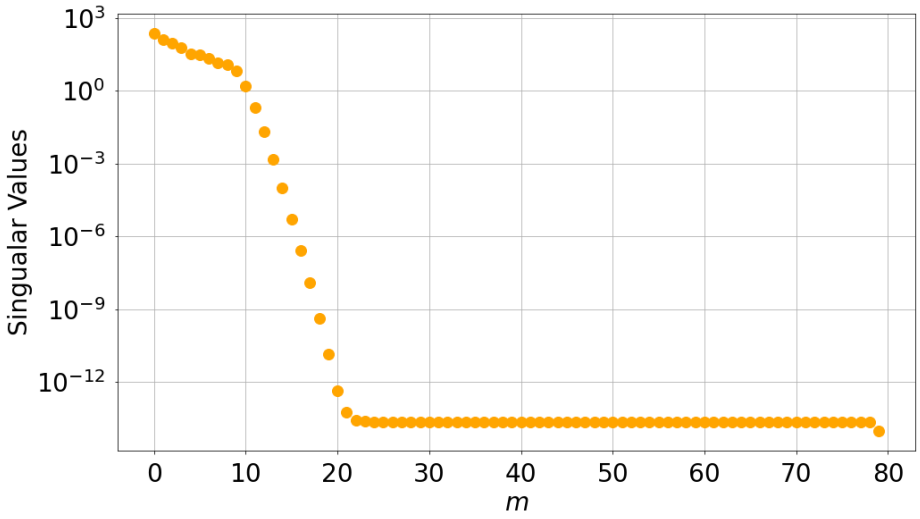


Figure 5.3: Test 1: Singular Value decay.

The configuration of the neural networks for the PDNNs-SVD and POD-G-NN approaches was given by:

- an input layer made up of just a neuron (which feeds the μ instance to the hidden layers);
- 4 hidden layers with 50 neurons each;
- hyperbolic tangent activation function;
- an output layer consisting in a number of neurons equal to the dimension of the linear trial manifold r.

In both cases this architecture has been trained for $2e04$ epochs with the Adam optimizer ($lr = 1e-03$) and subsequently for $2.5e04$ epochs with the L-BFGS one.

The configuration of the autoencoder is showed in Table (A.1). The training has been performed for 1e03 epochs exploiting the Adam optimizer, with an $ad-hoc$ scheduler which gradually reduces the learning rate value.

The corresponding autoencoder PDNN, which maps the parameters to the low rank representation of the associated snapshots, is identical to the network configuration of the other two approaches.

The average L_2 relative error between the three different methods and the FEM solutions, as a function of the dimension r of the corresponding reduced trial manifold is presented in Table (5.4) and graphically in Figure (5.4).

Approach	r						
	2	4	6	8	10	15	30
PDNNs-SVD	8.50e-01	4.74e-01	3.28e-01	2.27e-01	2.57e-02	8.60e-03	1.93e-02
PDNNs-AE	1.79e-02	1.32e-02	1.24e-02	1.26e-02	1.16e-02	1.18e-02	1.45e-02
POD-G-NN	8.57e-01	4.77e-01	3.33e-01	2.39e-01	2.39e-02	7.02e-03	5.55e-03

Table 5.4: Test 1: L_2 relative error e_{l_2} as a function of the reduced trial manifold dimension r for the 3 different ROM approaches.

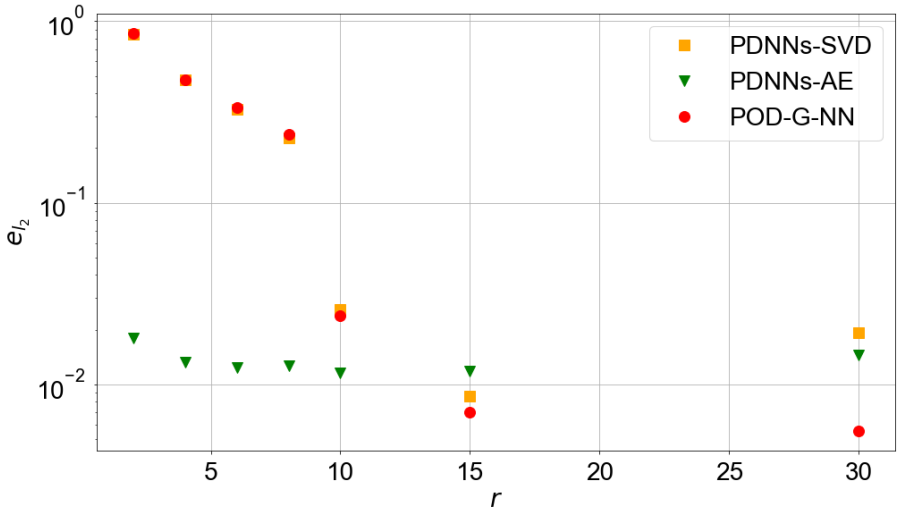
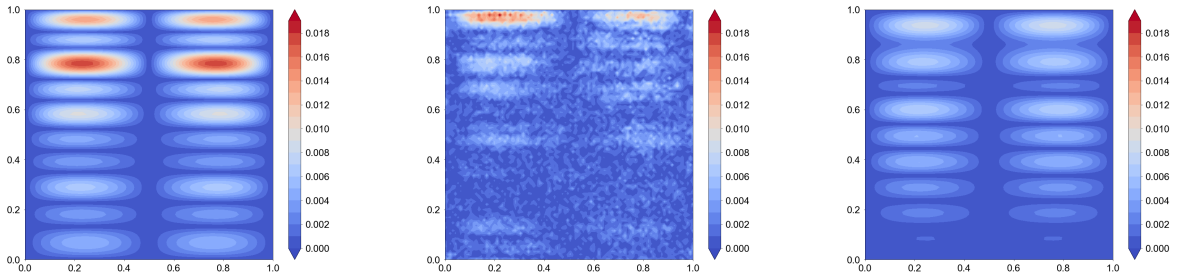


Figure 5.4: Test 1: L_2 relative error e_{l_2} vs reduced trial manifold dimension r: comparison between the ROM approaches.

We observe that for $r < 10$ the PDNNs-Autoencoder approach outperforms the two linear-ROM strategies by one order of magnitude. As r increases, the PDNNs-SVD and the PINNs-ROM methods gain in accuracy, and they achieve better performances than the PDNNs-Autoencoder strategy for $r > 10$. This behaviour was expected because the error of the latter, as r increases, is soon dominated by the Autoencoder reconstruction loss which cannot be improved in any way other than repeating the hyperparameter tuning process for each value of r, which clearly is unfeasible for a metter of time needed to perform it.

The PDNNs-SVD and the POD-G-NN continue to improve as r increases, with the latter giving the best result among all for $r = 30$.



(a) (b) (c)

Figure 5.5: Test 1: Comparison of the ROM approaches absolute error with respect to the FEM solution for $\mu = 10$. (5.5a) PDNNs-SVD for $r = 15$, (5.5b) PDNNs-Autoencoder for $r = 10$, (5.5c) POD-G-NN for $r = 30$.

Finally, in Figure (5.5) the lowest absolute errors for each approach considered are plotted for comparison:

5.2. Test 2: Advection Diffusion Equation

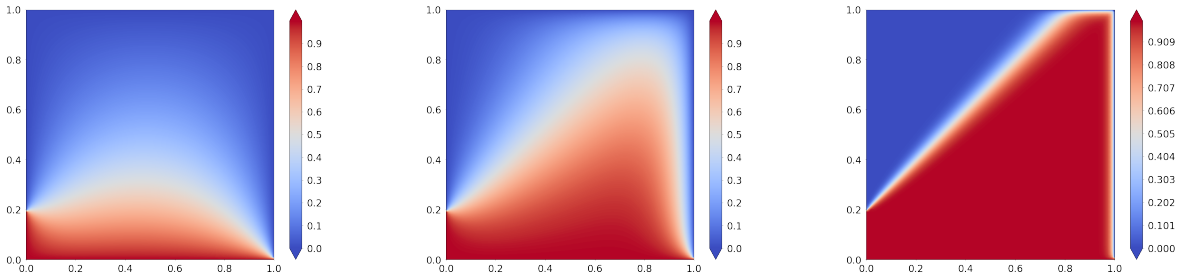
Let us consider now the parameterized two-dimensional steady and linear Advection Diffusion equation defined as:

$$\begin{cases} -\mu\Delta u + \boldsymbol{b}\cdot\nabla u = f & \text{in } \Omega = (0,1)^2, \\ u = g_D & \text{on } \Gamma_D = \delta\Omega, \end{cases} \tag{5.4}$$

where:

$$\begin{aligned} \boldsymbol{b} &= \begin{bmatrix} \cos\left(\frac{\pi}{4}\right) \\ \cos\left(\frac{\pi}{4}\right) \end{bmatrix}, \\ g_D &= \begin{cases} 0 & \text{for } x = 1,\ y = 1 \text{ and } x = 0 \wedge y > 0.2, \\ 1 & \text{for } y = 0 \text{ and } x = 0 \wedge y < 0.2, \end{cases} \\ f &= 0. \end{aligned} \tag{5.5}$$

The parameter space, to which the single μ parameter belongs, is given by $\boldsymbol{M} = [1e-03, 1e01]$. This problem is of particular interest because it shows the development of a thin boundary layer as the value of the viscosity μ decreases, as it can be seen in (5.6). The streamline diffusion stabilization method [20] has been employed to avoid numerical



(a) (b) (c)

Figure 5.6: Test 2: FEM solution at different instances of the parameter μ. (5.6a) $\boldsymbol{u}_h$ for $\mu = 3.681e00$, (5.6b) $\boldsymbol{u}_h$ for $\mu = 6.10e-02$, (5.6c) $\boldsymbol{u}_h$ for $\mu = 1.17e-03$.

instabilities which occur for $\mu \to 0$: this method is in fact capable of damping possible over– and undershootings of the discrete solution near the discontinuities while preserving higher order of convergence in regions where the solution is smooth.

The fact that the boundary conditions are not consistent at point $(1, 0)$ and discontinuous at point $(0, 0.2)$ represents an additional challenge, especially for the PINNs-FOM approach since a neural network is a *continuous* composite function.

To set up the PDNNs-SVD, PDNNs-Autoencoder and POD-G-NN approaches, the same procedure as the one described in section (5.1.2) has been followed. It must be noticed that the sampling of the parameter was not performed with a uniform probability distribution to avoid the generation of mostly $\mathcal{O}(1)$ values, since we were more interested into small numbers which allow the development of the boundary layer.

For this reason, when needed, an auxiliary parameter γ has been randomly sampled in the domain $\Gamma = [-3, 1]$, and then μ has been retrieved as:

$$\mu = 10^{\gamma}. \tag{5.6}$$

As in the previous test problem, 10 parameter values have been finally sampled to build a test set of high-fidelity benchmark solutions $(\boldsymbol{u}_h(\boldsymbol{\mu}_{test}))$ used to evaluate and compare the different approaches through Equation (5.1).

5.2.1. PINNs-FOM

The computational domain has been set up as $D = (0, 0, -3) \times (1, 1, 1)$. The third dimension has been assigned to the auxiliary parameter γ to better capture the solution

variations as μ approaches 0. The PDE residual has been then written keeping in mind the relationship between them written in Equation (5.6).

An hyperparameter optimization has been conducted as described in subsection (5.1.1). The PDE residual collocation points data set $\mathcal{T}_f$ of dimension $N_f = 20000$ and the boundary collocation points data set $\mathcal{T}_b$ of dimension $N_b = 5000$ have been generated by randomly sampling a number of points respectively in the computational domain and on its boundaries.

The optimal network architecture was characterized by the following hyperparameters:

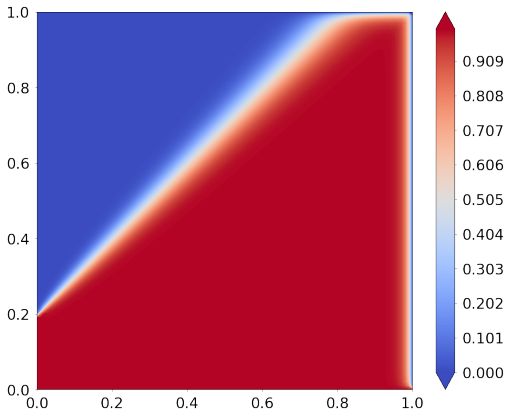
- hidden layers: 6 with 50 neurons each;
- activation functions: swish (see Figure (2.2d));
- optimizer: Adam (see subsection (2.2.5)) for $1e04$ epochs with a learning rate $lr = 1e-03$ and L-BFGS (see subsection (2.2.5)) for $3.5e04$ epochs.

Moreover, the network presented a first expanding layer as described in subsection (3.4.3).

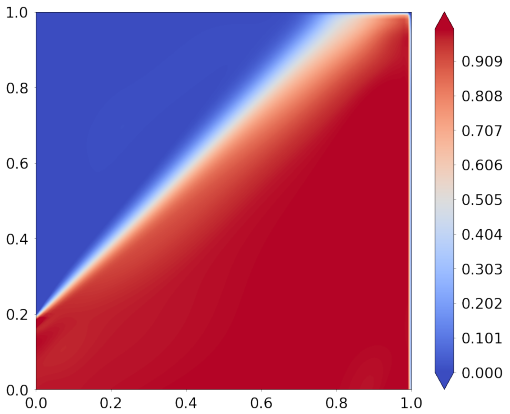
The imposition of the boundary conditions has been performed in a hybrid way. In fact, due to the nature of the Dirichlet boundary conditions to be matched, imposing them in a hard way would not have been straightforward. An attempt to imposing the $u = 0$ b.c. on the right edge was carried out, but it led to a completely off prediction (values of u close to zero in a great portion of the domain). On the contrary, imposing the $u = 0$ b.c. on the top edge led to an improvement in the solution accuracy: the final L_2 relative error on the test set was $e_{l_2} = 3.08e-02$, which is lower than the one obtained with only soft boundary conditions ($e_{l_2} = 4.17e-02$).

A comparison between the high-fidelity FEM solution and the PINNs-FOM one can be appreciated in Figure (5.7). The chosen parameter value to perform the test is $\mu = 1e-03$, the one that leads to the thinnest boundary layer among the ones in the parameter range.

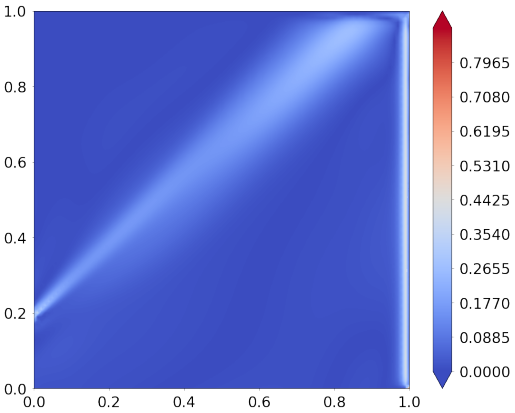
In Figure (5.8) it is possible to see a comparison between the two solution profiles taken for $y = 0.5$. From this plot it is possible to conclude that while the main characteristics of the flow are caught by the PINN, the result is anyway poor from a quantitatively point of view.



(a)



(b)



(c)

Figure 5.7: Test 2: Comparison of the PINNs-FOM solution with the FEM solution. (5.7a) the FEM solution $\boldsymbol{u}_h$, (5.7b) the PINNs-FOM solution $\hat{\boldsymbol{u}}$, (5.7c) the absolute error $|\boldsymbol{u}_h - \hat{\boldsymbol{u}}|$.

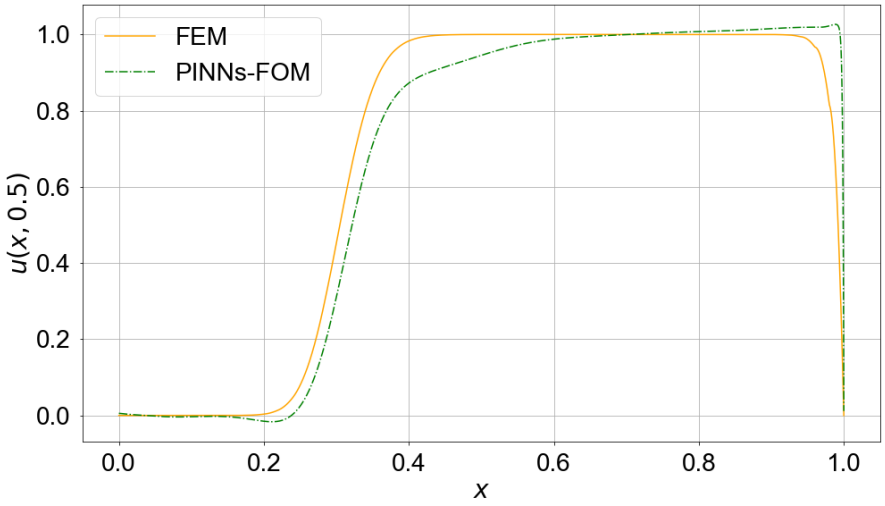


Figure 5.8: Test 2: Velocity profile of the PINNs-FOM solution at $y = 0.5$ compared to the FEM one.

5.2.2. PDNNs-SVD, POD-G-NN & PDNNs-Autoencoder

Problem (5.4) does not represent a challenging task for linear ROMs, as was the case of Problem (5.2). Indeed, by performing the SVD on the snapshot matrix it is found that a linear trial manifold of dimension $r = 6$ is capable to capture more than the 99.99% of the energy of the system, which is even less than what was required in Test 1.

For the sake of completeness, the Singular Values decay is shown in Figure (5.9). The decay is slower compared to the Test 1, but the 6 dominant modes are enough to capture most of the system energy.

The configurations of the networks employed were the same as the ones in Test 1.

The average L_2 relative error between the three different approaches and the FEM solutions, as a function of the dimension r of the corresponding reduced trial manifold is presented in Table (5.5) and graphically in Figure (5.10).

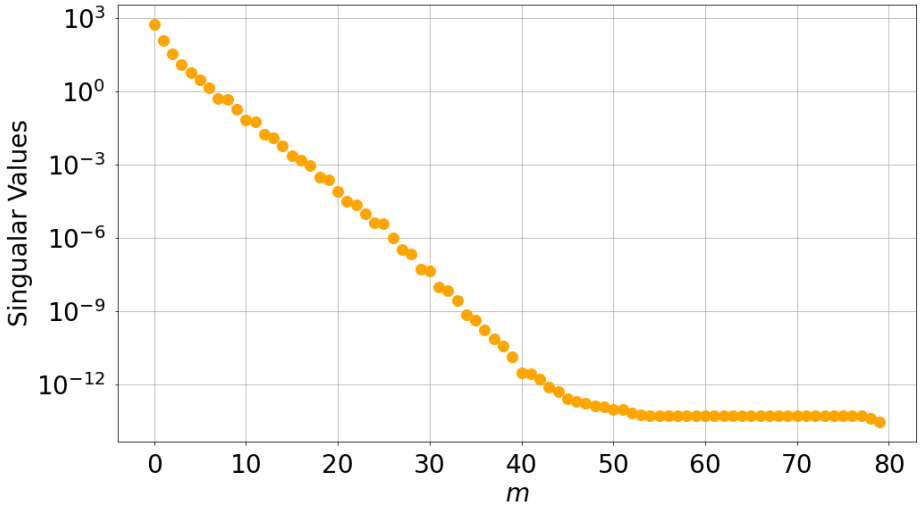
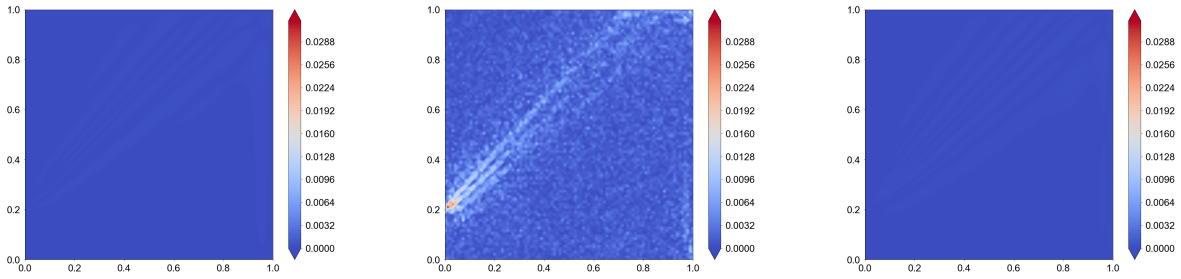


Figure 5.9: Test 2: Singular Values decay.

Approach	r						
	2	4	6	8	10	15	30
PDNNs-SVD	7.34e-02	1.14e-02	2.34e-03	6.82e-04	9.83e-05	3.03e-05	4.86e-05
PDNNs-AE	2.35e-03	2.21e-03	2.41e-03	2.31e-03	2.18e-03	1.99e-03	2.53e-03
POD-G-NN	1.71e-01	2.71e-02	9.53e-03	1.82e-03	8.51e-04	2.14e-05	3.73e-05

Table 5.5: Test 2: L_2 relative error e_{l_2} as a function of the reduced trial manifold dimension r for the 3 different ROM approaches.



(a) (b) (c)

Figure 5.11: Test 2: Comparison of the ROM approaches absolute error with respect to the FEM solution for $\mu = 1e-03$. (5.11a) PDNNs-SVD for $r = 15$, (5.11b) PDNNs-Autoencoder for $r = 15$, (5.11c) POD-G-NN for $r = 15$.

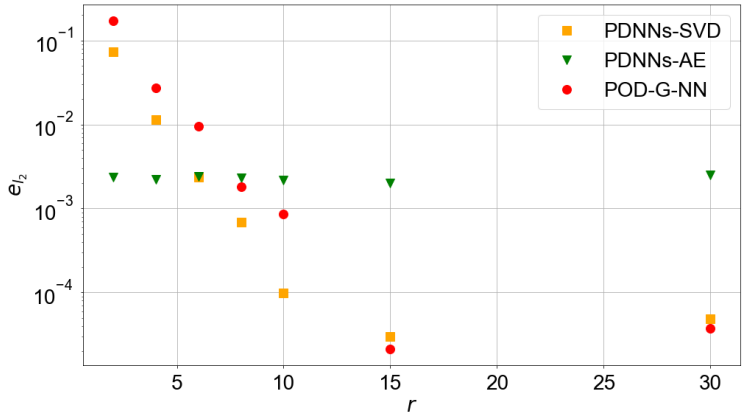


Figure 5.10: Test 2: L_2 relative error e_{l_2} vs reduced trial manifold dimension r: comparison between the ROM approaches.

As discussed in subsection (5.1.2), it is interesting to note that the PDNNs-Autoencoder approach outperforms the others until the reduced trial manifold dimension r is lower than the number of modes capable to capture the 99.99% of the system energy. However, as r increases, the linear ROM methods improve and lead to better results by 2 orders of magnitude. In particular, as r increases in the end the POD-G-NN approach become the most accurate one, like in Test 1.

Finally, in Figure (5.11) the lowest absolute errors for each approach considered are plotted for comparison.

It is possible to appreciate that the PDNNs-SVD and the POD-G-NN approaches led

to a much smaller error than the PDNNs-Autoencoder, for their optimal value of r (the difference is two orders of magnitude).

5.3. Test 3: Pure Advection Equation

Let us consider now the parameterized two-dimensional unsteady and linear Pure Advection equation, the circular transport of a Gaussian perturbation with period 1 [s]:

$$\begin{cases} \dfrac{\partial u}{\partial t} + \nabla \cdot (\boldsymbol{b} u) = f & \text{in } \Omega \times I = (0,1)^2 \times (0,1), \\ u = u_{ex} & \text{on } \Gamma_D = \delta\Omega \times I, \\ u|_{t=0} = u_{ex}(x,y,0) & \text{in } \Omega, \end{cases} \tag{5.7}$$

where:

$$\begin{aligned} &\boldsymbol{b}(x,y) = \begin{bmatrix} 2\pi(y-0.5) \\ 2\pi(0.5-x) \end{bmatrix}, \\ &u_{ex}(x,y,t) = 0.5e^{\left(-\frac{(x-x_c)^2+(y-y_c)^2}{2\sigma^2}\right)}, \\ &x_c(x,y,t) = 0.5 + \frac{\sin(2\pi t)}{4}, \\ &y_c(x,y,t) = 0.5 + \frac{\cos(2\pi t)}{4}, \\ &f = 0. \end{aligned} \tag{5.8}$$

System (5.7) has been discretized in space by means of cubic finite elements, with $N_h = 100 \times 100$ grid points. The time discretization has been characterized by 1001 time steps, and the integration has been performed via the Crank-Nicholson method [8], which, as opposed to the backward Euler, demonstrated to limit the development of an unwanted artificial diffusion. The single σ^2 parameter space is given by $\boldsymbol{M} = [5e-04, 5e-03]$.

In Figure (5.12) it is possible to appreciate the behaviour of the FEM solution at different time instances, while in Figure (5.13) the comparison between the FEM solution for 2 different values of σ^2 at $t = 0$[s].

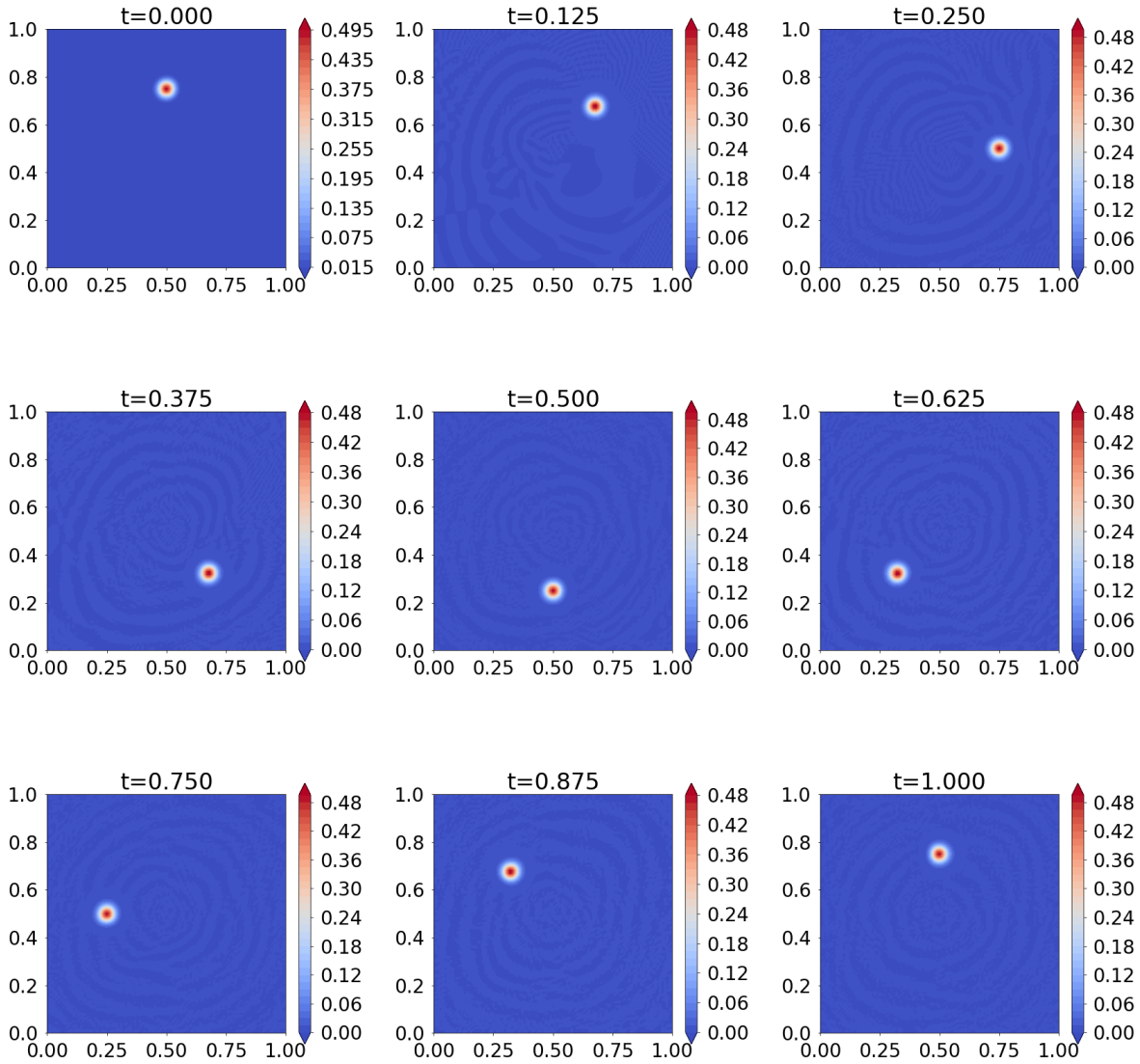
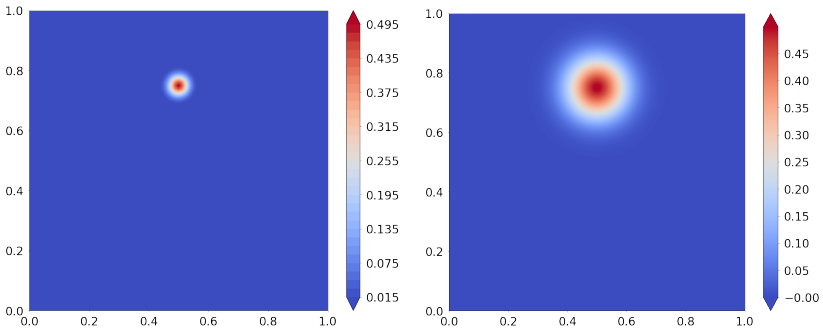


Figure 5.12: Test 3: FEM solution at various time steps for $\sigma^2 = 5e - 03$.



(a) (b)

Figure 5.13: Test 3: FEM solution at different instances of the parameter σ^2 at time $t = 0$[s]. (5.13a) $\boldsymbol{u}_h$ for $\sigma^2 = 5e-04$, (5.13b) $\boldsymbol{u}_h$ for $\sigma^2 = 5e-03$.

This problem is of particular interest because it shows a strong misalignment in the snapshot matrix $\boldsymbol{A}$, leading to a slow decay of its Singular Values. For this reason, as mentioned in subsection (1.2.2), it was expected that the PDNNs-SVD approach could meet some troubles to correctly reconstruct the solution, especially for low dimensions r of the linear trial manifold. At the same time it was expected that the PDNNs-Autoencoder approach, which is not based on the SVD, could mitigate this problem.

To set up the PDNNs-SVD and PDNNs-Autoencoder approaches, the same procedure as the one described in section (5.1.2) has been followed.

Finally, 10 pairs of time-parameter values have been sampled to build a test set of high-fidelity benchmark solutions $(\boldsymbol{u}_h(\boldsymbol{t}_{test}, \boldsymbol{\mu}_{test}))$ used to evaluate and compare the different approaches through Equation (5.1).

5.3.1. PINNs-FOM

The computational domain has been set up as $D = (0, 0, 0.1, 0) \times (1, 1, 1, 1)$. The third dimension has been assigned to the auxiliary parameter δ to avoid dealing with small numbers. The PDE residual has been then written keeping in mind the relationship between them:

$$\sigma^2 = \frac{\delta}{200} \tag{5.9}$$

An hyperparameter optimization has been conducted as described in subsection (5.1.1). The PDE residual collocation points data set $\mathcal{T}_f$ of dimension $N_f = 10000$ and the boundary collocation points data set $\mathcal{T}_b$ of dimension $N_b = 5000$ have been generated by ran-

domly sampling a number of points respectively in the computational domain and on its boundaries. As concern $\mathcal{T}_b$, 2500 points has been allocated to the boundary condition, while the remaining half to the initial condition.

The optimal network architecture was characterized by the following hyperparameters:

- hidden layers: 6 with 60 neurons each;
- activation functions: swish (see Figure (2.2d));
- optimizer: Adam (see subsection (2.2.5)) for $1.5e04$ epochs with a learning rate $lr = 1e-03$ and L-BFGS (see subsection (2.2.5)) for $3.5e04$ epochs.
- loss weights: after a trial-and-error optimization process, it has been found that giving more importance to the initial and boundary conditions leads to better results. Actually, if the weights in Equation (3.2) are left to the default value of 1, the network fails to learn at all. In our tests, a combination of $\alpha_f = 1$ and $\boldsymbol{\alpha}_b = [5, 10]$ led instead to a satisfactory result. Note that the first component of $\boldsymbol{\alpha}_b$ is assigned to the boundary condition, whereas the second one to the initial condition.

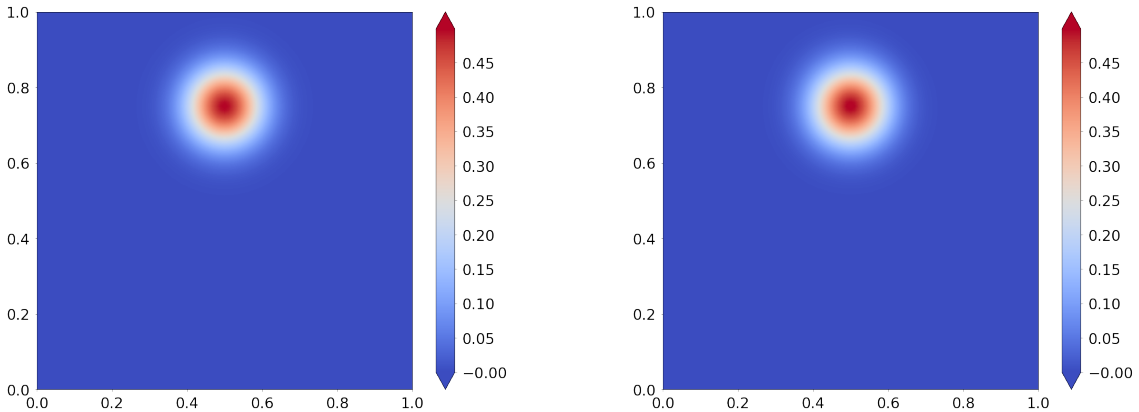
Moreover, the network has a first expanding layer as described in subsection (3.4.3). The imposition of the boundary conditions have been performed in the soft way, otherwise, given the fact that on the boundaries of the domain the function to satisfy is the exact solution, we would have simply forced it in the whole domain, which is not the point of the work.

A comparison between the high-fidelity FEM solution and the PINNs-FOM one can be appreciated in Figure (5.14). The parameter value chosen to conduct the comparison is $\sigma^2 = 5e-03$, the one that leads to the largest gaussian, and the time chosen was the final time, $t = 1.0$ [s].

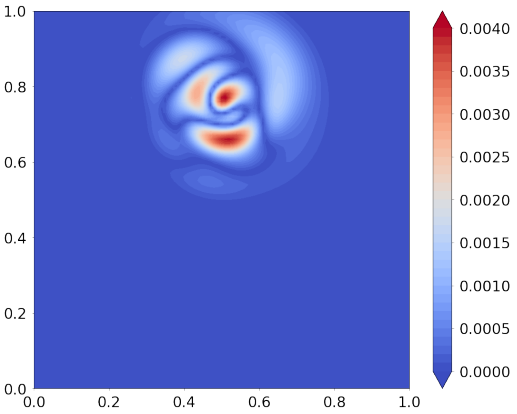
The L_2 relative error achieved was $e_{l_2} = 1.20e-02$.

5.3.2. PDNNs-SVD & PDNNs-Autoencoder

Due to data misalignment, problem (5.7) becomes a challenging task for linear ROMs. By performing the SVD on the snapshots matrix $\boldsymbol{A}$ it is found that 133 modes are needed in order to capture the 99.99% of the system energy. In Figure (5.15) the corresponding Singular Value decay plot is presented.



(a) (b)



(c)

Figure 5.14: Test 3: Comparison of the PINNs-FOM solution with the FEM solution. (5.14a) the FEM solution $\boldsymbol{u}_h$, (5.14b) the PINNs-FOM solution $\hat{\boldsymbol{u}}$, (5.14c) the absolute error $|\boldsymbol{u}_h - \hat{\boldsymbol{u}}|$.

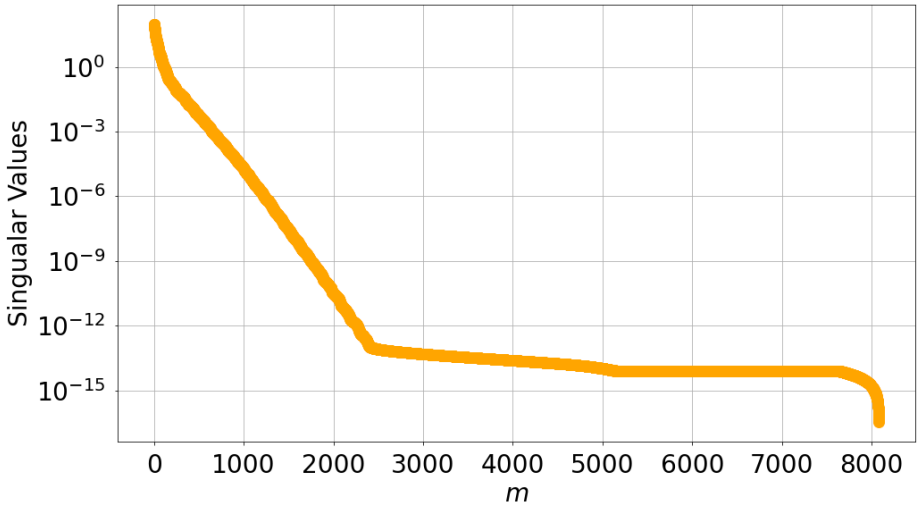


Figure 5.15: Test 3: Singular Values decay.

The configurations of the PDNN networks employed were the same as the ones in Test 1, but as the dimension r of the latent representation increased, the width and depth of the network were increased too to enhance the approximation capabilities of the network.

The average L_2 relative error between the two different approaches and the FEM solutions, as a function of the dimension r of the corresponding reduced trial manifold is presented in Table (5.6) and graphically in Figure (5.16).

Approach	r						
	2	20	40	80	120	160	200
PDNNs-SVD	9.32e-01	3.59e-01	1.53e-01	5.96e-02	5.62e-02	5.93e-02	5.84e-02
PDNNs-AE	8.45e-02	2.22e-02	2.37e-02	1.60e-02	1.61e-02	1.70e-02	1.86e-02

Table 5.6: Test 3: L_2 relative error e_{l_2} as a function of the reduced trial manifold dimension r for the 2 different ROM approaches.

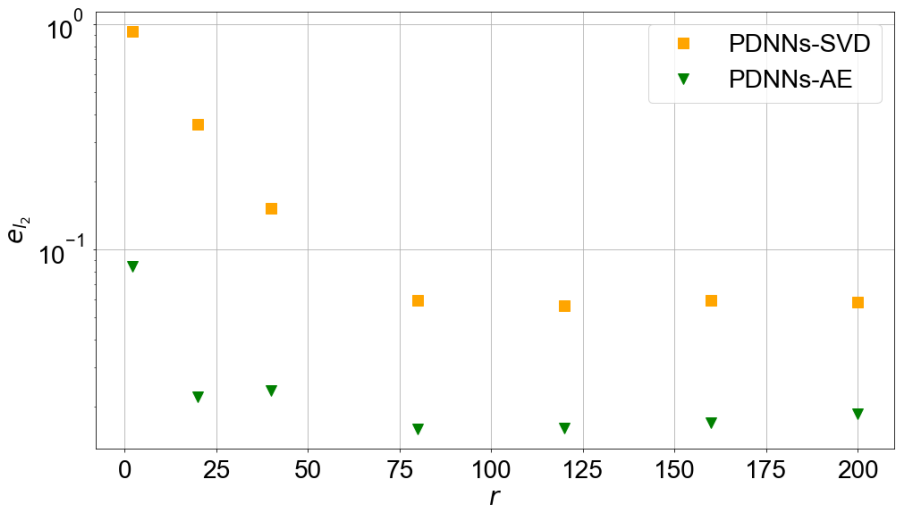
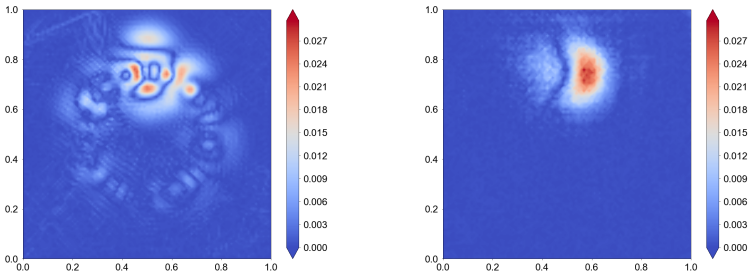


Figure 5.16: Test 3: L_2 relative error e_{l_2} vs reduced trial manifold dimension r: comparison between the ROM approaches.

Not surprisingly, Figure (5.16) shows the superiority of the PDNNs-Autoencoder approach over the PDNNs-SVD one in presence of data misalignment in the snapshot matrix $\boldsymbol{A}$ for the whole range of the reduced trial manifold dimension r considered.

In Figure (5.17) the lowest absolute errors for each approach considered are plotted for comparison:



(a) (b)

Figure 5.17: Test 3: Comparison of the ROM approaches absolute error with respect to the FEM solution for $\sigma^2 = 5e-03$. (5.17a) PDNNs-SVD for $r = 120$, (5.17b) for $r = 80$

5.4. Test 4: Burgers Equation

Finally, let us consider the parameterized mono-dimensional unsteady and non-linear Burgers equation defined as:

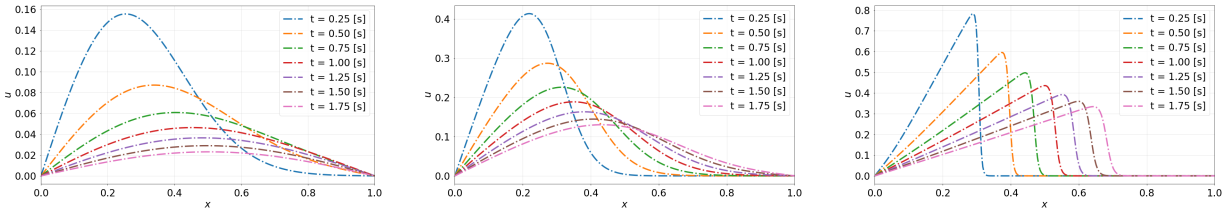
$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} - \mu\dfrac{\partial^2 u}{\partial x^2} = f & \text{in } \Omega \times I = (0,1) \times (0, 2.0), \\ u = g_D & \text{on } \Gamma_D = \delta\Omega \times I, \\ u|_{t=0} = u_0 & \text{in } \Omega, \end{cases} \tag{5.10}$$

where:

$$\begin{aligned} & g_D = 0, \\ & u_0 = \begin{cases} 0.5(1 - cos(8\pi x)), & \text{for } x < 0.25; \\ 0, & \text{otherwise}, \end{cases} \\ & f = 0. \end{aligned} \tag{5.11}$$

System (5.10) has been discretized in space by means of quadratic finite elements, with $N_h = 255$ grid points. As concern the time discretization, the generic θ-method formula has been coded and finally the method chosen was the Crank-Nicholson one, i.e., $\theta = 0.5$. The amount of time steps has been fixed to 2000. The single parameter μ space is given by $\boldsymbol{M} = [1e-03, 1e-01]$.

In Figure (5.18) the evolution of the FEM solution is plotted for different μ samples.



(a) (b) (c)

Figure 5.18: Test 4: evolution of the FEM solution at different instances of the parameter μ. (5.18a) $\boldsymbol{u}_h$ for $\mu = 9.23e{-}02$, (5.18b) $\boldsymbol{u}_h$ for $\mu = 2.24e{-}02$, (5.18c) $\boldsymbol{u}_h$ for $\mu = 1.65e{-}03$,

In agreement with what expected, it can be noticed that the wave front becomes sharper as the viscosity μ decreases, while for high μ values the diffusion term plays a major role.

Despite being a mono-dimensional test problem, we were interested into discovering how the two new strategies proposed (PINNs-FOM and PDNNs-Autoencoder) behaved in an unsteady non-linear parameterized context in comparison with the more traditional SVD based approach (PDNNs-SVD).

As in the previous case, 10 pairs of time-parameter samples have been generated to build a high-fidelity test set ($\boldsymbol{u}_h(\boldsymbol{t}_{test}, \boldsymbol{\mu}_{test})$), used to compare the various approaches.

5.4.1. PINNs-FOM

The computational domain has been set up as $D = (0, 1, 0) \times (1, 3, 2)$. The second dimension has been assigned to the auxiliary parameter η to avoid dealing with small numbers, given the relationship:

$$\mu = 10^{-\eta}. \tag{5.12}$$

The hyperparameter-tuning process led to the definition of the following optimal configuration:

- dimension N_f of the residual collocation points data set: 5000;
- dimension N_b of the boundary collocation points data set: 200 (assigned to the initial condition).
- hidden layers: 5 with 50 neurons each;
- activation functions: swish;
- optimizer: Adam for 1e04 epochs with a learning rate $lr = 1e-03$ and L-BFGS for 3.5e04 epochs.

As in the previous tests, the network has been equipped with a first expanding layer (see subsection (3.4.3) for reference). While the imposition of the boundary conditions in the hard way has been straightforward to code, the initial conditions has been treated as a soft constraint, given their piece-wise definition.

A comparison between the high-fidelity FEM solution and the PINNs-FOM one can be appreciated in Figure (5.19). The parameter value chosen to conduct the comparison is $\mu = 1e-03$, the one that gives the sharpest wave front, and the time chosen was t = 0.2 [s], when the wave peak is still high in magnitude.

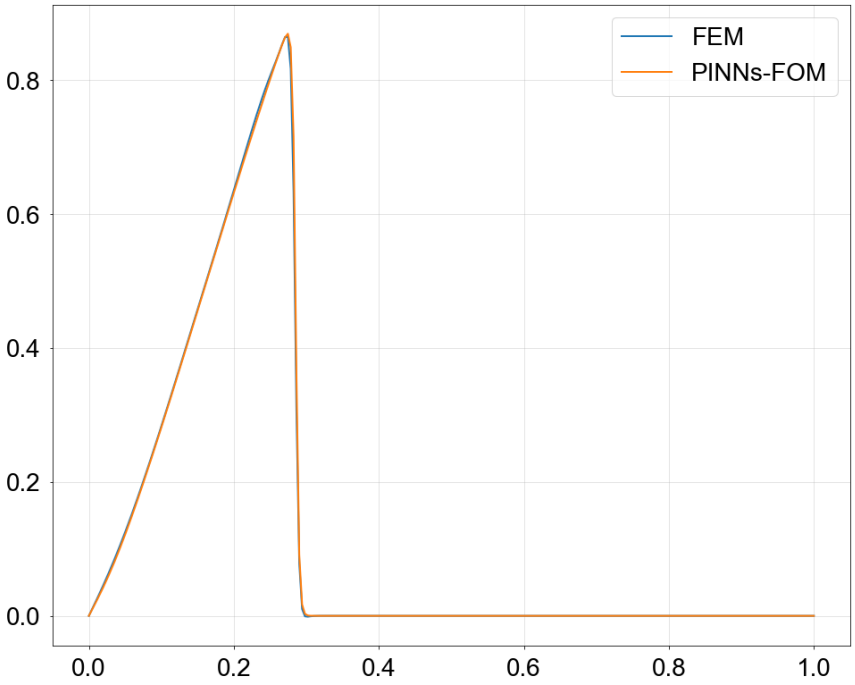


Figure 5.19: Test 4: Comparison between the PINNs-FOM and the FEM solution for $\mu = 1e-03$ and $t = 0.2$.

As it can be seen by the figure above, the accuracy of the proposed method is very high. The two solutions overlaps almost perfectly and there's no evidence of unphysical artificial artifacts. On the test set the method achieve an L_2 relative error $e_{l_2} = 1.57e-2$

5.4.2. PDNNs-SVD & PDNNs-Autoencoder

Due to the non-linear time-varying nature of the problem at hand, by performing the SVD on the snapshots matrix $\boldsymbol{A}$ it has been found that 27 modes are required to capture the 99.99% of the energy of this simple system. The slow Singular Value decay is showed in Figure (5.20).

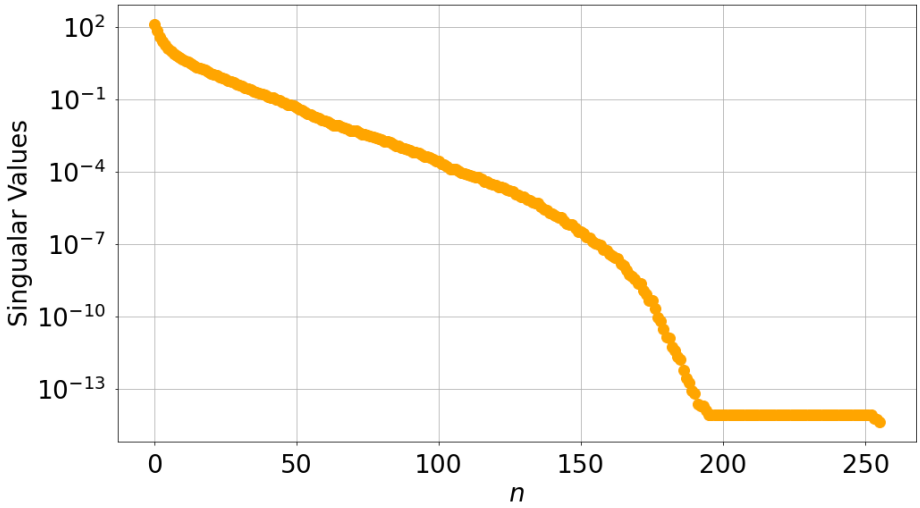


Figure 5.20: Test 4: Singular Values decay.

In this case the autoencoder architecture has been a little modified in order to cope with the reduced size of the input images, compared to the previous cases. In particular, the first convolution (and the last transpose convolution) of the autoencoder has been removed, the number of convolutional filters per layer has been halved and the kernel size has been reduced to 3×3 for the two deepest couples of convolutional and transpose-convolutional layers.

The average L_2 relative error between the two different approaches and the FEM solutions, as a function of the dimension r of the corresponding reduced trial manifold is presented in Table (5.7) and graphically in Figure (5.21).

Approach	r						
	2	4	6	10	15	30	60
PDNNs-SVD	2.81e-01	1.37e-01	7.77e-02	4.13e-02	1.99e-02	3.79e-03	3.97e-03
PDNNs-AE	1.64e-02	1.02e-02	8.84e-03	6.77e-03	6.87e-03	5.46e-03	5.74e-03

Table 5.7: Test 4: L_2 relative error e_{l_2} as a function of the reduced trial manifold dimension r for the 2 different ROM approaches.

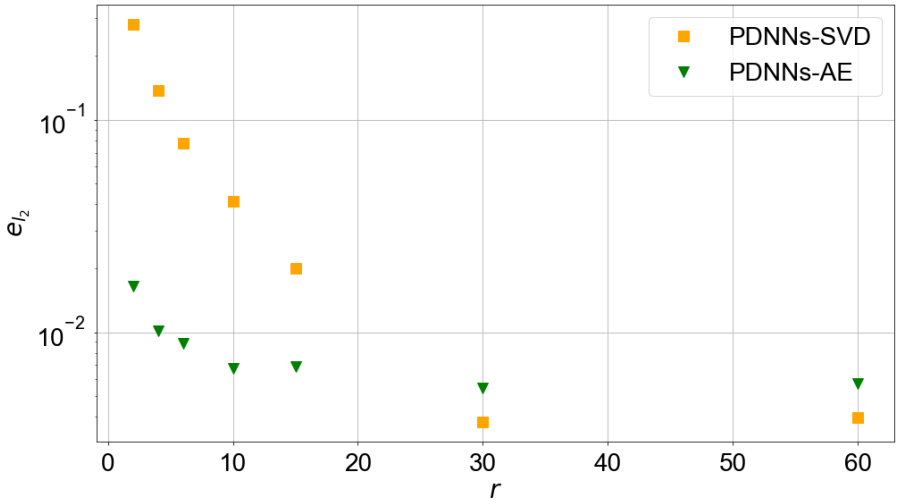


Figure 5.21: Test 4: L_2 relative error e_{l_2} vs reduced trial manifold dimension r: comparison between the ROM approaches.

The strong dimensionality reduction offered by the PDNNs-Autoencoder approach has been confirmed also in this case: for $r < 10$ the performances are one order of magnitude better than the ones characterizing the PDNNs-SVD approach. As r increases, the increase in accuracy of the autoencoder model rapidly saturates, until the output of the linear ROM becomes the most accurate.

Let us now consider the same time-parameter instance of the PINNs-FOM example, i.e, $\mu = 1e-03$ and $t = 0.2$[s]. The plot of the solutions obtained via the two approaches is compared with the high fidelity one in Figure (5.22).

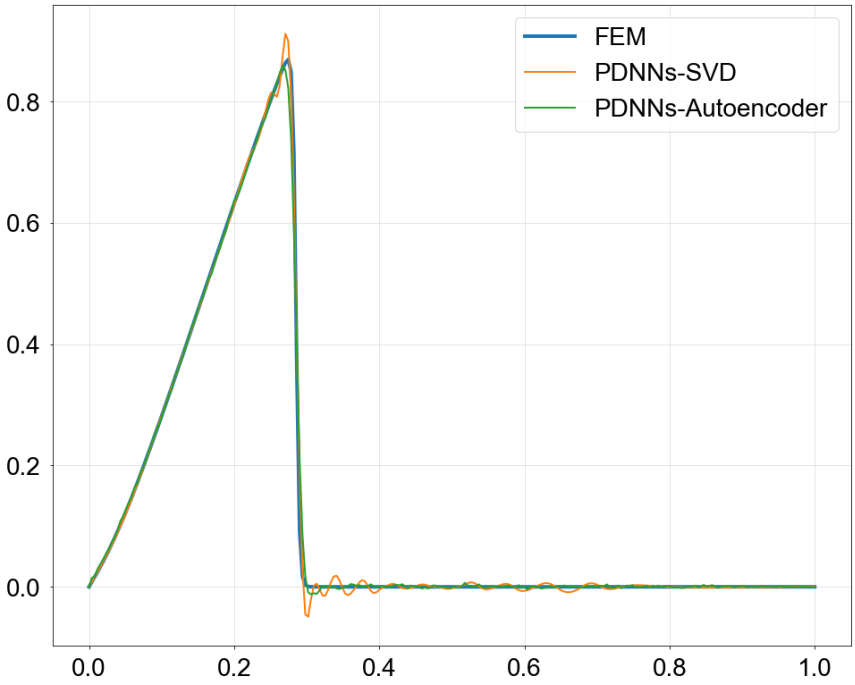


Figure 5.22: Test 4: Comparison of the ROM approaches with the FEM solution for $\mu = 1e-03$ and $t = 0.2$.

From the plot above is clearly visible, from a qualitative point of view, that the behaviour of the PDNNs-Autoencoder approach is much more realistic than the one of the PDNNs-SVD in presence of a sharp wave. In fact, the latter presents an unphysical overshoot in the region of the wave crest, as well as notable oscillations in the downstream area.

On the countrary, the solution offered by the PDNNs-Autoencoder strategy is very accurate, and the error is mainly made up of low-magnitude high-frequency noise, plus a small damping of the wave crest and a single low-magnitude undershoot in the downstream area.

5.5. Results summary

An overall overview on the performances of the various approaches for each test is offered in Table (5.8):

Test \ Method	PINNs-FOM	PDNNs-Ae	PDNNs-SVD	POD-G-NN
1	**1.32e-03**	1.16e-02	8.60e-03	5.55e-03
2	3.08e-02	1.99e-03	3.03e-05	**2.14e-05**
3	**1.20e-02**	1.60e-02	5.62e-02	-
4	1.57e-02	5.46e-03	**3.79e-03**	-

Table 5.8: Overall comparison between the investigated methods. The ROMs techniques values has been chosen case by case as the ones which led to achieve the best result. No constraint on the reduced trial manifold maximum dimension has been considered.

6 | Conclusions and future developments

In the present work two new promising strategies to solve parameterized PDEs while allowing at the same time a fast online prediction have been investigated and compared with more traditional SVD-based methods.

Concerning the PINNs-FOM method, the results it achieved are very satisfactory. In fact, it turned out to be the most accurate method on 2 out of 4 tests (Test 1 (5.1) and Test 2 (5.3)), outperforming the other strategies, as well as being the easiest to implement. It suffered more on the tests which involved the development of a thin boundary layer (5.2) and of a moving wave front (5.4).

It must be noted that in Test 2 the particular nature of the boundary conditions may have been the main reason for a less accurate result, since a PINN is a continuous composite function and thus not much suitable to model discontinuities.

To improve the PINNs-FOM accuracy, various advanced techniques have been proposed in literature which are worthy to be tested in further researches:

- the residual-based adaptive refinement [42] along with its variants. They are a class of methods thought to increase the PINN accuracy by clustering at each epoch additional collocation points in the areas of the computational domain in which the PDE residual is higher;
- a first adaptive loss weights balancing algorithm [40], which aims to mitigate a series of gradient pathologies afflicting the network;
- a second adaptive loss weights balancing algorithm [41], which aims to improve the network training phase by respecting the causality of the problem, e.g., giving more importance in the initial epochs to collocation points which are close to the time origin.

Beside all the mentioned, another interesting way to improve the method is generating

and adding some high-fidelity snapshots as additional boundary conditions to match. The loss function will then present a term which takes into account the distance between the network output and the snapshots values. While it is likely that this strategy will lead to better performances, its drawback consists of an additional offline cost given by the need of solving the FOM system via a traditional numerical solver at least for few instances of the parameters.

Regarding instead the PDNNs-Autoencoder approach, it has been proved that its dimensionality reduction capabilities are higher than a traditional SVD-based method, thanks to its non-linear framework. Moreover, as depicted in Figure(5.22), in presence of sharp wave fronts, the approximation delivered is much more realistic than the PDNNs-SVD oscillatory one.

However, this strategy is not drawbacks-free too. First of all, it must be noticed that the hyperparameter tuning process of the autoencoder architecture and its training procedure is very time consuming. This may represent a bottleneck, especially if the programmer deals with such a problem for the first time. Gaining experience will certainly mitigate the issue.

Moreover, in all the test cases it can be noticed that as the latent space representation dimension increases, the enhancements in the solution accuracy rapidly saturates, as opposed to the SVD-based approaches. This behaviour can be explained by the fact that the shape of the time-parameters to latent representation map $\boldsymbol{g}$ is in general harder to learn than the relationship between the time-parameters space and the corresponding reduced coefficients.

Therefore, the training process of the network in the PDNNs-Autoencoder case should be improved. A possible solution is given by the L-BFGS-B optimizer, which in the PDEs context demonstrated to be always the most effective. Due to python libraries conflicts, in this specific approach its usage has not been possible. In the future it is expected to find some workarounds to this limitation.

Finally, regarding the time-dependent problems, an improvement in the autoencoder reconstruction capability could be given by equipping its dense portion with recurrent or LSTM units, as done in [31]. In this way, the autoencoder will be aware of the temporal dependence which exists between two consecutive snapshots and could exploit it as an additional information to increase its accuracy.

In conclusion it can be said that this two approaches, despite being totally different from a conceptual point of view, represent both a great alternative to the traditional methods

for fast online parameterized PDEs solution. They both have their drawbacks but, as mentioned above, the ideas to achieve better performances are not lacking, and thus their potential is high for possible future industrial applications.

Bibliography

[1] J. Abbasi and P. Ø. Andersen. Physical activation functions (pafs): An approach for more efficient induction of physics into physics-informed neural networks (pinns). *arXiv preprint arXiv:2205.14630*, 2022.

[2] P. Astrid, S. Weiland, K. Willcox, and T. Backx. Missing point estimation in models described by proper orthogonal decomposition. *IEEE Transactions on Automatic Control*, 53(10):2237–2251, 2008.

[3] G. Berkooz, P. Holmes, and J. L. Lumley. The proper orthogonal decomposition in the analysis of turbulent flows. *Annual review of fluid mechanics*, 25(1):539–575, 1993.

[4] S. L. Brunton and J. N. Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2022.

[5] S. Chaturantabut and D. C. Sorensen. Nonlinear model reduction via discrete empirical interpolation. *SIAM Journal on Scientific Computing*, 32(5):2737–2764, 2010.

[6] W. Chen, Q. Wang, J. S. Hesthaven, and C. Zhang. Physics-informed machine learning for reduced-order modeling of nonlinear problems. *Journal of Computational Physics*, 446:110666, 2021. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2021.110666. URL `https://www.sciencedirect.com/science/article/pii/S0021999121005611`.

[7] X. Chen, R. Chen, Q. Wan, R. Xu, and J. Liu. An improved data-free surrogate model for solving partial differential equations using deep neural networks. *Scientific reports*, 11(1):19507, September 2021. ISSN 2045-2322. doi: 10.1038/s41598-021-99037-x. URL `https://europepmc.org/articles/PMC8484684`.

[8] J. Crank and P. Nicolson. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Mathematical Proceedings of the Cambridge Philosophical Society*, 43(1):50–67, 1947. doi: 10.1017/S0305004100023197.

[9] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[10] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[11] S. Fresca, L. Dede, and A. Manzoni. A comprehensive deep learning-based approach to reduced order modeling of nonlinear time-dependent parametrized pdes. *Journal of Scientific Computing*, 87(2):1–36, 2021.

[12] P. Gavrikov. visualkeras. `https://github.com/paulgavrikov/visualkeras`, 2020.

[13] Github contributors. Bringing parallelism to the web with river trail, 2015. URL `http://intellabs.github.io/RiverTrail/tutorial/`. [Online; accessed 28-October-2022].

[14] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[15] H. Gunes, S. Sirisup, and G. E. Karniadakis. Gappy data: To krig or not to krig? *Journal of Computational Physics*, 212(1):358–382, 2006.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

[17] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[18] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *CoRR*, abs/1709.01507, 2017. URL `http://arxiv.org/abs/1709.01507`.

[19] A. D. Jagtap, K. Kawaguchi, and G. Em Karniadakis. Locally adaptive activation functions with slope recovery for deep and physics-informed neural networks. *Proceedings of the Royal Society A*, 476(2239):20200334, 2020.

[20] C. Johnson and U. Nävert. An analysis of some finite element methods for advection-diffusion problems. In O. Axelsson, L. Frank, and A. Van Der Sluis, editors, *Analytical and Numerical Approaches to Asymptotic Problems in Analysis*, volume 47 of *North-Holland Mathematics Studies*, pages 99–116. North-Holland, 1981. doi: https://doi.org/10.1016/S0304-0208(08)71104-6. URL `https://www.sciencedirect.com/science/article/pii/S0304020808711046`.

[21] G. Kerschen, J.-c. Golinval, A. F. Vakakis, and L. A. Bergman. The method of

proper orthogonal decomposition for dynamical characterization and order reduction of mechanical systems: an overview. *Nonlinear dynamics*, 41(1):147–169, 2005.

[22] Y. Kim, Y. Choi, D. Widemann, and T. Zohdi. A fast and accurate physics-informed neural network reduced order model with shallow masked autoencoder. *Journal of Computational Physics*, 451:110841, 2022. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2021.110841. URL `https://www.sciencedirect.com/science/article/pii/S0021999121007361`.

[23] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL `https://doi.org/10.1145/3065386`.

[25] I. Lagaris, A. Likas, and D. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998. doi: 10.1109/72.712178. URL `https://doi.org/10.1109%2F72.712178`.

[26] H. P. Langtangen and A. Logg. *Solving PDEs in Python*. Springer, 2017. ISBN 978-3-319-52461-0. doi: 10.1007/978-3-319-52462-7.

[27] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[28] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

[29] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[30] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis. Deepxde: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021.

[31] R. Maulik, B. Lusch, and P. Balaprakash. Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders. *Physics of Fluids*, 33(3):037106, 2021.

[32] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[33] A. Quarteroni, A. Manzoni, and F. Negri. *Reduced basis methods for partial differential equations: an introduction*, volume 92. Springer, 2015.

[34] M. Raissi, P. Perdikaris, and G. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2018.10.045. URL `https://www.sciencedirect.com/science/article/pii/S0021999118307125`.

[35] S. S. Rao. *The finite element method in engineering*. Butterworth-heinemann, 2017.

[36] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[37] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[38] T. Tieleman, G. Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[39] H. K. Versteeg and W. Malalasekera. *An introduction to computational fluid dynamics: the finite volume method*. Pearson education, 2007.

[40] S. Wang, Y. Teng, and P. Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021.

[41] S. Wang, S. Sankaran, and P. Perdikaris. Respecting causality is all you need for training physics-informed neural networks, 2022. URL `https://arxiv.org/abs/2203.07404`.

[42] C. Wu, M. Zhu, Q. Tan, Y. Kartha, and L. Lu. A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 403:115671, 2023. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2022.115671. URL `https://www.sciencedirect.com/science/article/pii/S0045782522006260`.

[43] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

A | Appendix A

Table A.1: PINN accuracy vs amount of collocation points

Layer	Number of filters	Kernel size	Stride	Activation	Output size
InputLayer					$96 \times 96 \times 1$
Conv2D	16	7×7	1×1	elu	$96 \times 96 \times 16$
BatchNormalization					$96 \times 96 \times 16$
MaxPooling2D					$48 \times 48 \times 16$
Conv2D	32	5×5	1×1	elu	$48 \times 48 \times 32$
BatchNormalization					$48 \times 48 \times 32$
MaxPooling2D					$24 \times 24 \times 32$
Conv2D	64	5×5	1×1	elu	$24 \times 24 \times 64$
BatchNormalization					$24 \times 24 \times 64$
MaxPooling2D					$12 \times 12 \times 64$
Conv2D	128	5×5	1×1	elu	$12 \times 12 \times 128$
BatchNormalization					$12 \times 12 \times 128$
MaxPooling2D					$6 \times 6 \times 128$
Conv2D	256	5×5	1×1	elu	$6 \times 6 \times 256$
BatchNormalization					$6 \times 6 \times 256$
MaxPooling2D					$3 \times 3 \times 256$
Flatten					2304
Dense				elu	256
BatchNormalization					256
Dense				elu	128
BatchNormalization					128
Dense				elu	64
BatchNormalization					64
Dense				elu	32

BatchNormalization					32
Dense				elu	16
BatchNormalization					16
Dense				elu	32
BatchNormalization					32
Dense				elu	64
BatchNormalization					64
Dense				elu	128
BatchNormalization					128
Dense				elu	256
BatchNormalization					256
Dense				linear	2304
Reshape					$3 \times 3 \times 256$
Conv2DTranspose	256	5×5	2×2	elu	$6 \times 6 \times 256$
BatchNormalization					$6 \times 6 \times 256$
Conv2DTranspose	128	5×5	2×2	elu	$12 \times 12 \times 128$
BatchNormalization					$12 \times 12 \times 128$
Conv2DTranspose	64	5×5	2×2	elu	$24 \times 24 \times 64$
BatchNormalization					$24 \times 24 \times 64$
Conv2DTranspose	32	5×5	2×2	elu	$48 \times 48 \times 32$
BatchNormalization					$48 \times 48 \times 32$
Conv2DTranspose	16	7×7	2×2	elu	$96 \times 96 \times 16$
BatchNormalization					$96 \times 96 \times 16$
Conv2DTranspose	1	5×5	1×1	linear	$96 \times 96 \times 1$

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my family: my parents, both for their huge moral and financial support, and my relatives, which they have always rooted for me (special mention for my grandmother Rosa).

I would like to thank all the people that I met during these 5 years in Milan, especially the ones I made strong ties with: thank you for having made me feeling alive.

And finally thanks to my girlfriend, who supported me in the last month mostly spent on the pc.