Neural Networks-based reduced-order modeling of PDEs

Numerical Analysis for Partial Differential Equations
Alfio Maria Quarteroni

Academic Year 2020/2021

Mathematical Engineering, Politecnico di Milano

Tutors: Stefano Pagani, Francesco Regazzoni

Andrea Boselli, Carlo Ghiglione, Leonardo Perelli

03-01-2022

Contents

1 Abstract

The objective of this project is to implement and test the ability of deep Neural Networks to approximate parametric maps underlying the solution of PDEs. In particular, given a PDE depending on some parameters, we implement a system being able to accurately approximate the solution of the equation.
Provided that high fidelity numerical solutions of a given problem are available, and that selection of parameters and hyperparameters and training of the architecture are properly performed (off-line phase), the Neural Network is able to provide the approximations of the solutions of the problem in an extremely fast way (on-line phase), compared to numerical solvers based on Finite Element Methods. This finds application in all the areas that require to quickly compute solutions of problems that depend on unknown values of parameters.
With this work, we explore the possibility of implementing such architectures, studying their characteristics and limitations. To do so, we first focus on a case study problem, a linear Advection-Diffusion equation that features as parameters the diffusion and transport coefficient.
Next, we focus on the Monodomain equation, contributing to model cardiac electrical activity, under different configurations of the PDE domain. We cope with this variability parametrising the domain as a function of some scalar variables, and considering such variables as parameters of the problem.
For the practical implementation, we work on Python, in particular using Nisaba, a machine learning library built on top of Tensorflow.
This report is organized as follows.

1. Introduction to the *Physics Informed Neural Network* approach, with a focus on rationale and methodologies;

2. Definition and implementation of a Neural Network as emulator of PDE solutions, applied to an Advection Diffusion problem;

3. Quality check of the emulator accuracy through the extraction of the input parameters given a fixed solution (Identification);

4. Implementation of an alternative emulator through Proper Orthogonal Decomposition and comparison with previous one;

5. Application of PINN framework to Monodomain equation, featuring new parameters that control geometrical properties of the PDE domain.

2 Introduction

2.1 Methods

In this work we consider PDEs depending on some parameters, and we test different approaches to represent the manifold of the PDE solutions, depending on those

parameters. For the purpose we employ Neural Networks, very flexible and powerful function approximators: flexible because input size, output size and internal structure can be configured arbitrarily; powerful because their output depends on model parameters called weights, that can be tuned through minimization of a proper error function called loss.
The novelty of our approach is representing the manifold of the PDE solutions through a parametric map u_{NN}, whose input consists of both a point in the PDE domain and an instance of the PDE parameters, and whose output is the solution of the PDE on the given point for the given instance. In the upcoming chapters, we represent such manifold first directly through a Neural Network (chapter 3), next as a linear combination of proper basis functions (chapter 5). Moreover, we apply some of these techniques to the case of parameters defining the geometry of the PDE domain, while not explicitly appearing in the PDE (chapter 6)
The quality of the solution for a given task is represented by a loss. In this work, we consider the *Physics Informed Neural Networks* (PINN) approach, in which both the information from given high fidelity solutions and the knowledge of the physics of the problem are exploited. As such, the loss function is the weighted sum of three components, that contribute to keep into account both the aspects.
In general, let $\mathcal{D}_{int}$ and $\mathcal{D}_{BC}$ be a grid of internal and boundary points of the PDE domain respectively; let $MSE_{\mathcal{D}}(f)$ be the mean square of a function f on grid $\mathcal{D}$:

$$MSE_{\mathcal{D}}(f) = \frac{1}{|\mathcal{D}|}\sum_{\underline{x}\in\mathcal{D}}[f(\underline{x})]^2$$

Let $\mathbb{U} = \{u_{HF}^{(i)}\}_{i=1}^{N}$ be a set of functions used to train the Neural Network and let $\{u_{NN}^{(i)}\}_{i=1}^{N}$ be the corresponding functions learned through the Neural Network, where $u_{NN}^{(i)}$ is the approximation of $u_{HF}^{(i)}$, each generated for different choices of the values of the parameters. $\mathbb{U}$ is task dependent, but in most of the cases it features solutions of the PDE for a given set of parameters: in such case, let $RES_i(u_{NN}^{(i)})$ be the residual of the learned $u_{NN}^{(i)}$ with respect to the PDE solved by $u_{HF}^{(i)}$. Then we define:

$$\textit{Fit error}: \mathcal{L}_{fit} = \frac{1}{|\mathbb{U}|}\sum_{u_{HF}^{(i)}\in\mathbb{U}} MSE_{\mathcal{D}_{int}}(u_{HF}^{(i)} - u_{NN}^{(i)})$$

$$\textit{PDE residual}: \mathcal{L}_{PDE} = \frac{1}{|\mathbb{U}|}\sum_{u_{HF}^{(i)}\in\mathbb{U}} MSE_{\mathcal{D}_{int}}(RES_i(u_{NN}^{(i)}))$$

$$\textit{BC residual}: \mathcal{L}_{BC} = \frac{1}{|\mathbb{U}|}\sum_{u_{HF}^{(i)}\in\mathbb{U}} MSE_{\mathcal{D}_{BC}}(u_{HF}^{(i)} - u_{NN}^{(i)})$$

As a result, the loss expression is:

$$\mathcal{L} = w_{fit}\cdot\mathcal{L}_{fit} + w_{PDE}\cdot\mathcal{L}_{PDE} + w_{BC}\cdot\mathcal{L}_{BC}$$

with w_{fit}, w_{PDE}, w_{BC} the weights of the three components of the loss. The actual expressions of the summands of $\mathcal{L}$ are given in each chapter, together with the choices

for the weights.
The loss can be a function either of the weights of the Neural Network (chapters 3, 5) or of the PDE parameters (chapter 4). In both cases, minimisation is performed through gradient descent, exploiting Python library Nisaba. The trainings always start with Adam algorithm and only later switch to L-BFGS which requires heavier computations.

2.2 Case study problem: Comet Equation

The mathematical problem we want to address first is the following partial differential Advection-Diffusion equation, the so called *Comet Equation*:

$$\begin{cases} -\mu\Delta u + 10(\cos\theta, \sin\theta) \cdot \nabla u = 10e^{-100|\underline{x}-\underline{x}_0|} & \underline{x} \in \Omega = [0,1]^2 \\ u = 0 & \underline{x} \in \partial\Omega \end{cases}$$

The name is due to the particular shape of the solution of the problem, whose radial direction is controlled by the θ parameter whereas how much the comet shape is accentuated is controlled by the diffusion parameter μ.
The interest of the problem is in the fact that, due to its simplicity, the two coefficients control clearly separate and well identifiable characteristics of the solution so that it can be studied how our implementations is able to tackle different kind of transformations.

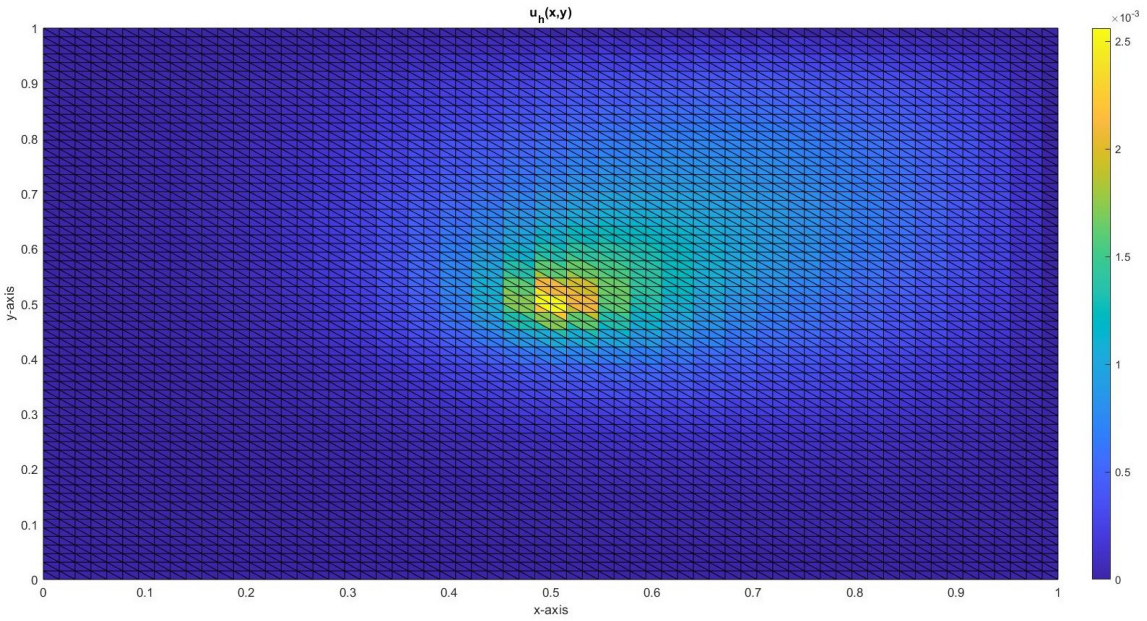

Figure 1: Numerical solution relative to $\mu = 1$ and $\theta = \pi/4$ obtained with a P1 FEM solver with grid refinement of 6 ($h = 0.0110$)

2.3 Dataset Preparation

Working with Neural Networks necessarily requires a dataset to train the model. In this case, the data units are the numerical solutions for the couples of parameters μ and θ in certain ranges computed with a FEM P1 solver always using a grid refinement of 5 ($h = 0.0221$). Depending on the specific applications, we use slight variations on the range and the numerosity of the grid of parameters μ and θ that are shown in this table:

Name	Range of μ	N° of μ	N° of μ train	Range of θ	N° of θ	N° of θ train
A	$[1, 5]$	10	5	$[0, 2\pi]$	20	10
B	$[0.01, 5]$	11	6	$[0, 2\pi]$	20	10
C	$[0.01, 5]$	17	10	$[0, 2\pi]$	20	10

It can be observed that with respect to coefficient μ:

- Dataset A covers a smaller and more precise range of values only for solutions in a balanced regime;
- Dataset B covers a larger and sparser range of values, containing solutions in the advection dominated regime too;
- Dataset C covers the same range of B but it is richer.

In the table, the training and test set proportion splits are reported too in such a way that test is done for solutions relative to couples of parameters unseen in the training but still in the range of chosen values.

3 Emulators

3.1 Objective

In this chapter we represent the manifold of the PDE solutions as a Neural Network that, given the two parameters μ and θ in a selected range and a geometrical point in the problem domain, is able to return an accurate solution of the PDE for the provided coefficients on that point. Formally, we seach for a map u_{NN} such that:

$$\begin{aligned} u_{NN}: \quad & [0,1]^2 \times \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \\ & (x, y, \mu, \theta) \mapsto u_{NN}(x, y, \mu, \theta) \end{aligned}$$

The purpose is twofold: first, assess if a manifold of PDE solutions can actually be represented through a Neural Network; next, investigate the effect that introducing knowledge of the physics of the problem has on training.

3.2 Implementation

Once the training dataset is built, we compute its overall maximal and minimal values of the solutions (M and m respectively) and we use them to normalize the numerical solutions (both of training and test dataset) in the $[0, 1]$ interval:

$$u^* = \frac{u - m}{M - m}$$

To be coherent, we have to normalize the PDE too:

$$\begin{cases} -(M-m)\mu\Delta u^* + 10(M-m)(\cos\theta, \sin\theta) \cdot \nabla u^* = 10e^{-100|\underline{x}-\underline{x}_0|} & \underline{x} \in \Omega = [0,1]^2 \\ u^* = \frac{f-m}{M-m} & \underline{x} \in \partial\Omega \end{cases}$$

The implemented Neural Network has a very simple architecture: 3 dense fully connected layers of 20 neurons each with $tanh(\cdot)$ activation function plus the final linear neuron for the output, having in total less than 1000 parameters.

```
_________________________________________________________________
Layer   (type)                Output Shape              Param #
=================================================================
dense_0 (Dense)               (None, 20)                80
_________________________________________________________________
dense_1 (Dense)               (None, 20)                420
_________________________________________________________________
dense_2 (Dense)               (None, 20)                420
_________________________________________________________________
dense_3 (Dense)               (None, 1)                 21
=================================================================
Total params: 941
Trainable params: 941
Non-trainable params: 0
_________________________________________________________________
```

3.3 Loss definition

Let $\mathbb{U}$ be a given dataset of numerical solutions, $\mathcal{D}_{int}$ and $\mathcal{D}_{BC}$ the set of internal and boundary points respectively on which $u_i \in \mathbb{U}$ are defined. Then, referring to the notation from paragraph 2.1, we set:

$$\mathcal{L}_{fit} = \frac{1}{|\mathbb{U}||\mathcal{D}_{int}|} \sum_{u_i \in \mathbb{U}} \sum_{(x,y) \in \mathcal{D}_{int}} [u_i^*(x,y) - u_{NN}(x,y,\mu_i,\theta_i)]^2$$

$$\mathcal{L}_{PDE} = \frac{1}{|\mathbb{U}||\mathcal{D}_{int}|} \sum_{u_i \in \mathbb{U}} \sum_{(x,y) \in \mathcal{D}_{int}} [Res^*(u_{NN},x,y,\mu_i,\theta_i)]^2$$

$$\mathcal{L}_{BC} = \frac{1}{|\mathbb{U}||\mathcal{D}_{BC}|} \sum_{u_i \in \mathbb{U}} \sum_{(x,y) \in \mathcal{D}_{BC}} [u_i^*(x,y) - u_{NN}(x,y,\mu_i,\theta_i)]^2$$

where u_i^* is the normalized solution corresponding to u_i, (μ_i, θ_i) are the coefficients of the corresponding PDE, and $Res^*(u_{NN}, x, y, \mu, \theta)$ is the residual of $u_{NN}(\cdot, \mu, \theta)$ function on (x, y) with respect to the normalized PDE with coefficients (μ, θ). Loss weights w_{fit}, w_{PDE}, w_{BC} are decided case by case in each training.
Finally, gradient descent is performed with respect to the weights of the implemented Neural Network u_{NN}.

3.4 Results for Dataset A

We train the network separately on dataset A, B and C, doing different experiments. For dataset A, three different trainings are done always performing first 250 epochs with *Adam* optimizer with *learning rate* of 10^{-3} and then 10^4 epochs with *BFGS* optimizer. For all three, w_{fit} and w_{BC} are set to 1 and w_{PDE} is set to 1, 10 and 0 respectively. In all the cases the losses for the test set reach an accuracy between 10^{-2} and 10^{-3} apart for the PDE loss in the case where its weight is null.
These trainings suggest first that the PDE loss term has little impact on the fitting accuracy of the model, and makes the training significantly slower (in the order of one third on average); nonetheless, it is crucial to actually reduce the residual of the PDE, hence setting $w_{PDE} = 0$ is not suggested. Moreover, we notice that for higher values of w_{PDE} the training tends to be more unstable and to reach lower accuracy.
On account of these conclusions, we decide to adopt for the next developments the emulator trained with all the weights being equal to 1.

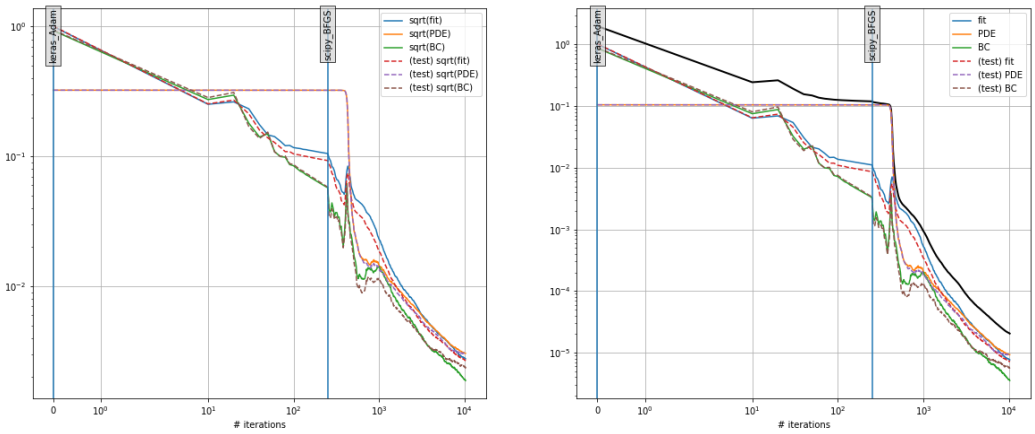

Figure 2: Training for Dataset A with $w_{PDE} = 1$

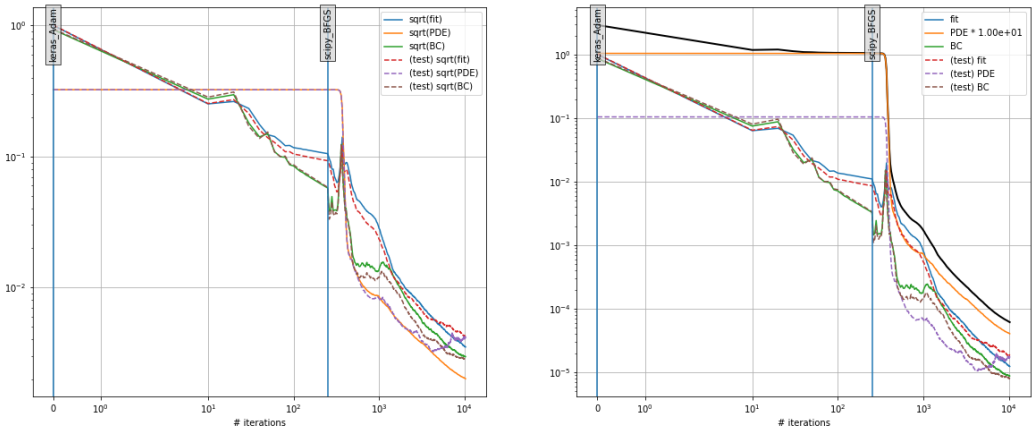

Figure 3: Training for Dataset A with $w_{PDE} = 10$

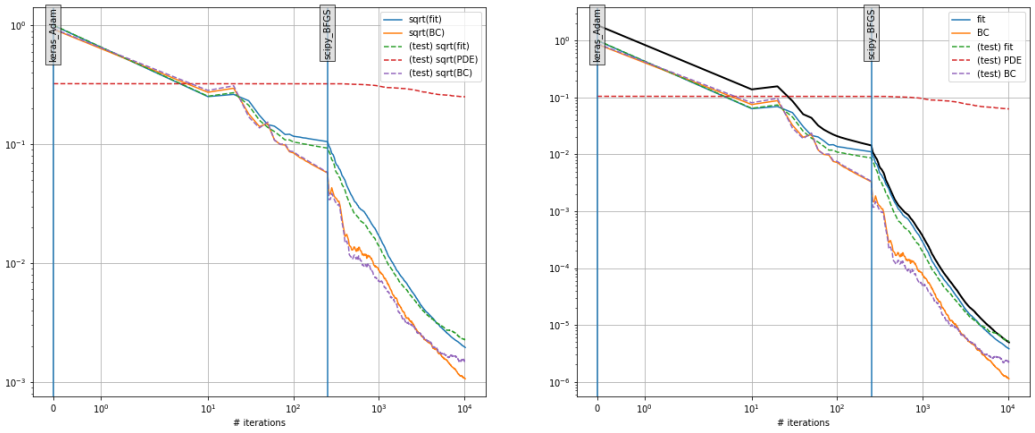

Figure 4: Training for Dataset A with $w_{PDE} = 0$

3.5 Results for Dataset B and C

For Dataset B and C, a training is done with 10^3 epochs with *Adam* optimizer with learning rate of 10^{-3} and then $9 \cdot 10^3$ epochs with *BFGS* optimizer, setting $w_{fit} = 10$ and the others equal to 1. Moreover, the number of neurons per layer is increased to 24.

We notice that with Dataset B the network seems unable to minimize the PDE loss, whereas the training is more stable using the richer Dataset C: probably, Dataset B is not rich enough to capture the different scales of the numerical solutions the network has to learn. This suggests that too little datasets make learning unstable and unable to reach a high accuracy.

Performing one last training on dataset C with neurons per layer increased to 25 and BFGS epochs increased to 10^4, we highly increase overfitting. We deduce that increasing the network size may not provide benefits in terms of accuracy, despite the heavier and longer training.

In conclusion, we observe that the solutions reach test loss values not lower than 10^{-2}, significantly worse than Dataset A. We think this is due on the one side to the accuracy limits of the available numerical solutions and, on the other side, on the irregular shape of the solutions for low values of the diffusion coefficient μ.

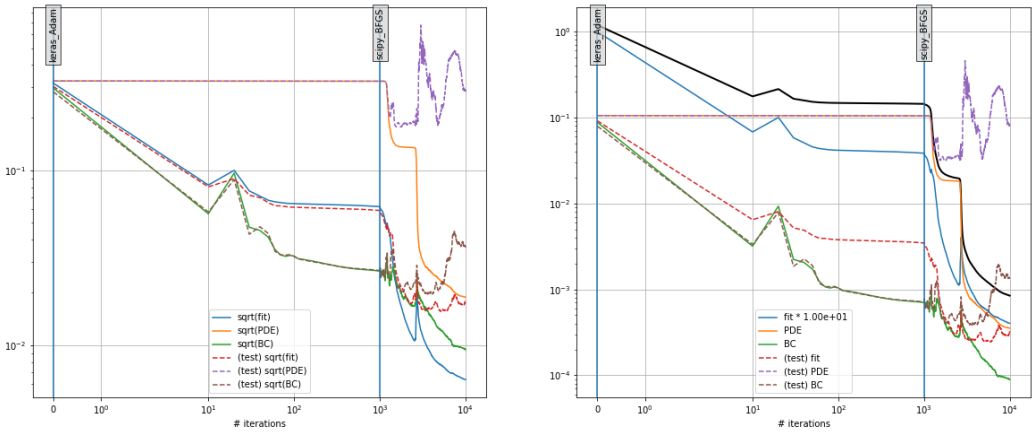

Figure 5: Training for Dataset B with 24 neurons per layer

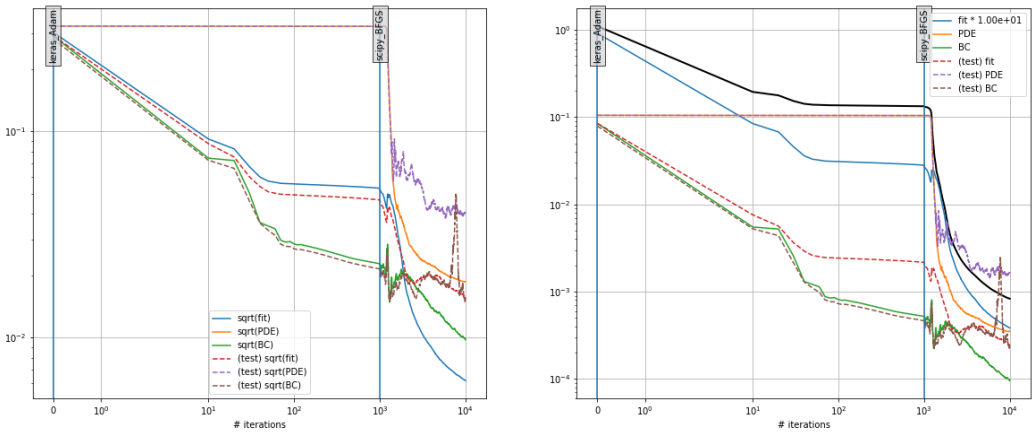

Figure 6: Training for Dataset C with 24 neurons per layer

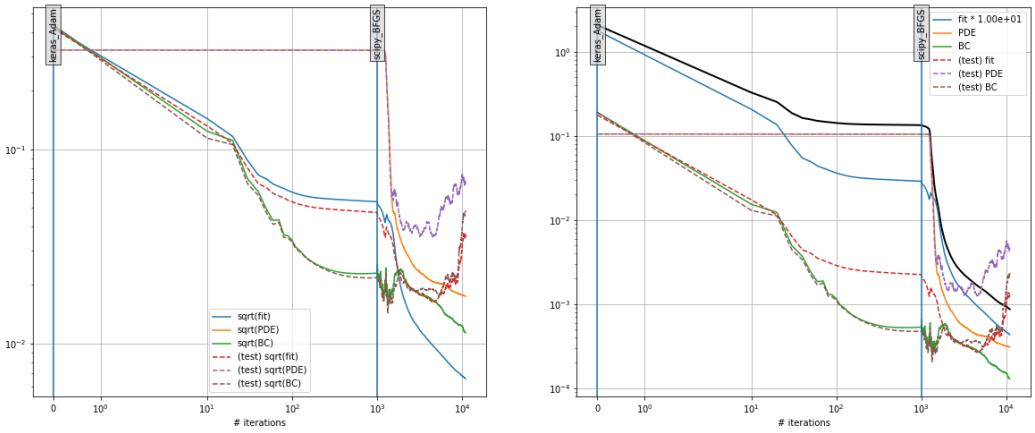

Figure 7: Training for Dataset C with 25 neurons per layer

3.6 Conclusions

The above results show that Neural Networks can approximate the manifold of the solutions of a PDE, dependent on some parameters, even at different scales. In particular, we are able to achieve successful trainings if we supply a rich enough dataset, featuring sufficiently accurate solutions.
PDE loss can be useful to ensure the drop of the PDE residual for the Neural Network function, hence it can be regarded as a further check on the quality of our solution.
Finally, considering that the networks are trained on our home laptops, the emulator proves to be reliable in terms of accuracy. With more computational power, we could deeply explore the selection of the loss weights, in particular for the PDE loss, and of the other hyperparameters.

4 Identification

4.1 Objective

We want to test the goodness of the trained emulator performing *identification* that is: for a given solution u of the PDE, find the corresponding couple of coefficients $(\tilde{\mu}, \tilde{\theta})$ of the PDE associated to u.
Formally, setting:

$$\mathcal{U} := \{u : [0,1]^2 \to \mathbb{R} : u \text{ solves the PDE for some } \mu \in \mathbb{R}^+, \theta \in \mathbb{R}\}.$$

we look for the map $\mathcal{M}$ such that:

$$\begin{aligned} \mathcal{M}: \quad & \mathcal{U} \to \mathbb{R}^+ \times \mathbb{R} \\ & u \mapsto (\tilde{\mu}, \tilde{\theta}) \text{ for which } u \text{ solves the PDE} \end{aligned}$$

4.2 Implementation

To perform identification, we need the parametric map:

$$\begin{aligned} u_{NN}: \quad & [0,1]^2 \times \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \\ & (x, y, \mu, \theta) \mapsto u_{NN}(x, y, \mu, \theta) \end{aligned}$$

that maps a point (x, y) of the physical domain and a couple (μ, θ) of parameters into the solution of the PDE on the given point, for the given parameters. We choose as u_{NN} the Neural Network model we have implemented in Section 3.4. Recall that u_{NN} is trained on a dataset of rescaled functions, hence it must be compared with functions that have gone through the same rescaling. Consequently, for the rest of this chapter, $\mathcal{U}$ denotes the set of solutions of the normalized PDE introduced in section 3.2: M and m are maximum and minimum value of dataset A used to train u_{NN}.
Next, we choose as loss function the distance induced by the discrete L^2 norm between the given function $u \in \mathcal{U}$ and the NN model u_{NN}, for a fixed couple (μ, θ):

$$\mathcal{L}(\mu, \theta) = \mathcal{L}_{fit}(\mu, \theta) = \frac{1}{|\mathcal{D}_{int}|} \sum_{(x,y) \in \mathcal{D}_{int}} \big[u(x, y) - u_{NN}(x, y, \mu, \theta)\big]^2$$

where $\mathcal{D}_{int}$ is the grid of the internal points (x, y) of the physical domain on which $u(x, y)$ is known.
Finally, minimization is performed with respect to the PDE parameters (μ, θ):

$$(\tilde{\mu}, \tilde{\theta}) = \underset{(\mu,\theta) \in \mathbb{R}^+ \times \mathbb{R}}{\operatorname{argmin}} \mathcal{L}(\mu, \theta)$$

4.3 Test 1

We first try the method on a function coming from the training Dataset A.

	μ	θ
Exact Value	1.00	0.00
Initial Guess	5.00	3.141592
Estimated Value	0.999286	6.297931

We notice that, instead of obtaining $\theta \approx 0$, we obtain $\theta \approx 2\pi$, which makes sense as well: as θ appears in the equation only through $cos(\theta)$ and $sin(\theta)$, the two outcomes are equivalent. Plotting $\mathcal{L}(\mu, \theta)$, we notice both the local minima:

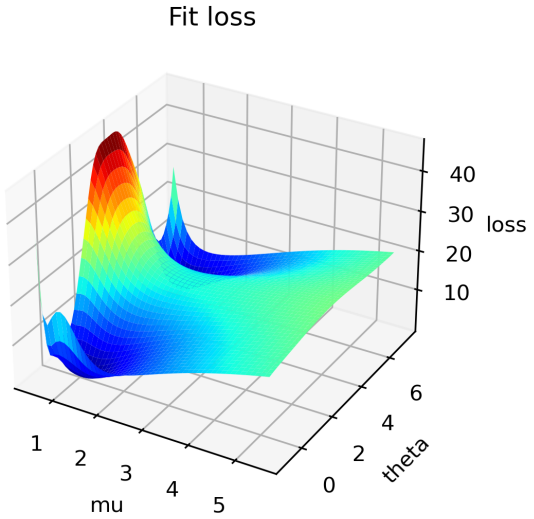

Figure 8: Plot of $\mathcal{L}(\mu, \theta)$ for Test 1

4.4 Test 2

Next, we try with a function $u \in \mathcal{U}$ not included in the training data, and we consider three different initializations:

Attempt 1	μ	θ
Exact Value	3.00	4.45
Initial Guess	1.00	3.00
Estimated Value	3.137339	4.481961

Attempt 2	μ	θ
Exact Value	3.00	4.45
Initial Guess	3.00	4.45
Estimated Value	3.138033	4.481983

Attempt 3	μ	θ
Exact Value	3.00	4.45
Initial Guess	3.00	1.00
Estimated Value	4.092807	-1.043056

The first two initializations show a decrease in the identification quality. Indeed, the given function u is not used to train u_{NN}. The third attempt instead completely fails in predicting the correct μ and θ, because of a further local minimum in $\mathcal{L}(\mu, \theta)$. We could solve this issue by means of multiple random initializations, as each optimization episode is not much time consuming.

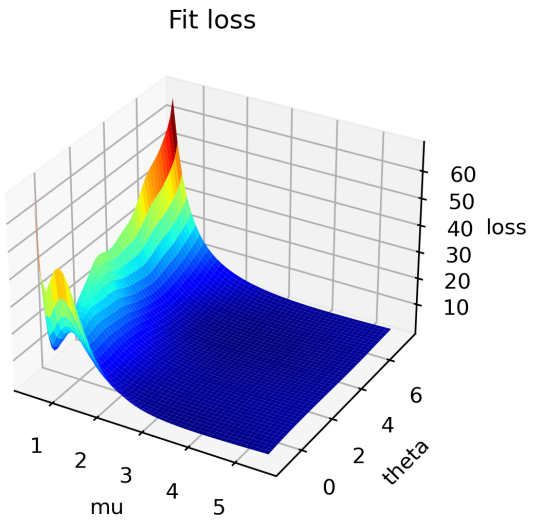

Figure 9: Plot of $\mathcal{L}(\mu, \theta)$ for Test 2

4.5 Observations

With the described method, we carry out a sound identification. In particular, the success in exploiting the NN emulator u_{NN} confirms its reliability, and consequently the quality of the performed training.
Of course, with higher computational resources, we could train more complex models on larger datasets, and choose broader ranges for the parameters, thus achieving better performances.
Finally, identification has applications in a variety of fields, notably Monte Carlo Markov Chain from statistics and Sensitivity Analysis from mathematical modeling, but these are beyond our purposes.

5 Proper Orthogonal Decomposition

5.1 Objective

The *Proper Orthogonal Decomposition* (POD) technique is an alternative method to represent the manifold of the solutions of a PDE depending on some parameters as a linear combination of some basis functions. We want to implement this method for our problem, finding proper basis functions and building a map that associates to any couple of parameters (μ, θ) the set of coefficients to accurately approximate any solution of the *Comet Equation*.
Formally, let $\mathcal{D}_{int}$ and $\mathcal{D}_{BC}$ sets of internal and boundary points of $[0,1]^2$ respectively, and $\mathcal{D}$ their union:

$$\mathcal{D} := \mathcal{D}_{int} \cup \mathcal{D}_{BC}$$

Let $\mathbb{U}$ be a set of numerical solutions of the PDE for given couples (μ, θ), evaluated on $\mathcal{D}$. In this chapter, we approximate each $u \in \mathbb{U}$ through a linear combination of K proper basis functions $\{\phi_k\}_{k=1}^K$, with coefficients $\{u_k\}_{k=1}^K$ depending on (μ, θ):

$$u(x,y) \approx \sum_{k=1}^{K} u_k(\mu,\theta)\phi_k(x,y) + u_0(x,y) + b(\mu,\theta)$$

where u_0 is the pointwise mean of all $u \in \mathbb{U}$ on $\mathcal{D}$, while the *bias* $b(\mu, \theta)$ copes with potential numerical issues.
After finding the basis functions $\{\phi_k\}_{k=1}^K$, we approximate the map $\mathcal{C}$ from the parameters (μ, θ) of a given PDE into bias b and coefficients $\{u_k\}_{k=1}^K$ costituting its reduced order representation:

$$\begin{aligned} \mathcal{C}: \quad & \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}^{K+1} \\ & (\mu,\theta) \mapsto \{u_k(\mu,\theta)\}_{k=1}^K, b(\mu,\theta) \end{aligned}$$

5.2 Basis extraction

In order to extract $\{\phi_k\}_{k=1}^K$ and u_0, we perform POD, which is the application of *Principal Component Analysis* (PCA) in the field of PDEs, on the numerical solutions from Dataset A.
PCA is a statistical method that performs dimensionality reduction of a multivariate dataset, looking for the main reasons of data variability in the features space. In POD, the single numerical solution $u \in \mathbb{U} = A$ corresponds to a single statistical datum, while its value $u(x, y)$ on a fixed $(x, y) \in \mathcal{D}$ corresponds to a statistical feature.
First, we gather all the p numerical solutions $u \in \mathbb{U}$ in matrix S, whose ith row stores the values of the numerical solution u_i on all the domain points $\underline{x} \in \mathcal{D}$:

$$S = \begin{array}{c} \\ (\mu_1,\theta_1) \\ (\mu_2,\theta_2) \\ \vdots \\ (\mu_p,\theta_p) \end{array} \begin{array}{c} \begin{array}{cccc} \underline{x}_1 & \underline{x}_2 & \cdots & \underline{x}_n \end{array} \\ \begin{bmatrix} u_1(\underline{x}_1) & u_1(\underline{x}_2) & \cdots & u_1(\underline{x}_n) \\ u_2(\underline{x}_1) & u_2(\underline{x}_2) & \cdots & u_2(\underline{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ u_p(\underline{x}_1) & u_p(\underline{x}_2) & \cdots & u_p(\underline{x}_n) \end{bmatrix} \end{array} = \begin{array}{c} \\ (\mu_1,\theta_1) \\ (\mu_2,\theta_2) \\ \vdots \\ (\mu_p,\theta_p) \end{array} \begin{bmatrix} \underline{u}_1^T \\ \underline{u}_2^T \\ \vdots \\ \underline{u}_p^T \end{bmatrix}$$

Next, we compute $\forall \mathbf{x} \in \mathcal{D}$ the above mentioned $u_0(\underline{x})$:

$$u_0(\underline{x}) = \frac{1}{p} \sum_{i=1}^{p} u_i(\underline{x})$$

and we gather all the functions u–u_0 as done for matrix S, obtaining matrix U:

$$\underline{u}_0^T = \begin{array}{c} \begin{array}{cccc} \underline{x}_1 & \underline{x}_2 & \cdots & \underline{x}_n \end{array} \\ \begin{bmatrix} u_0(\underline{x}_1) & u_0(\underline{x}_2) & \cdots & u_0(\underline{x}_n) \end{bmatrix} \end{array} \qquad U = \begin{array}{c} \\ (\mu_1,\theta_1) \\ (\mu_2,\theta_2) \\ \vdots \\ (\mu_p,\theta_p) \end{array} \begin{bmatrix} \underline{u}_1^T - \underline{u}_0^T \\ \underline{u}_2^T - \underline{u}_0^T \\ \vdots \\ \underline{u}_p^T - \underline{u}_0^T \end{bmatrix}$$

Now we compute the covariance matrix C:

$$C = \frac{1}{p-1} U^T U$$

corresponding to the covariance matrix from statistics. As such, we compute eigenvalues and eigenvectors $\{(\lambda^{(k)}, \underline{\phi}^{(k)})\}_{k=1}^{n}$ of matrix C, and select the K eigenvectors corresponding to the K highest eigenvalues (after carefully removing complex eigenvalues, which are due to numerical issues). Each eigenfunction $\underline{\phi}^{(k)}$ stores the values of the basis function ϕ_k on the domain points $\underline{x} \in \mathcal{D}$:

$$\forall \underline{x}_j \in \mathcal{D} \ \phi_k(\underline{x}_j) = \phi_j^{(k)}$$

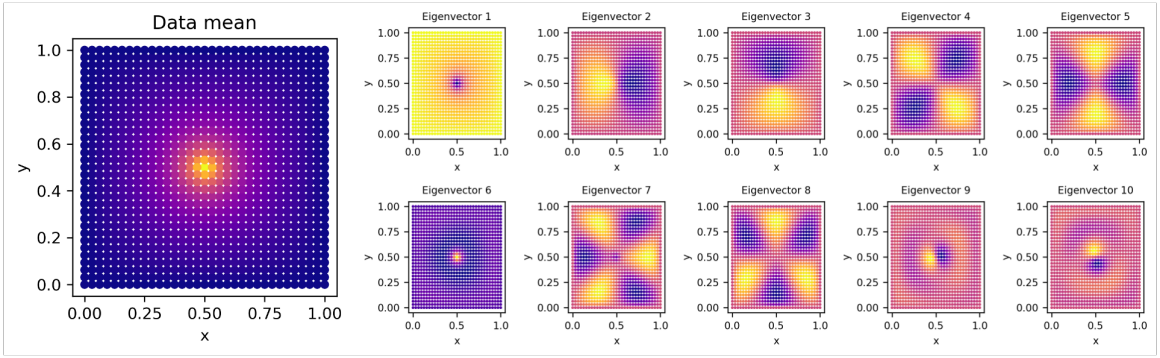

Figure 10: Plot on $[0,1]^2$ of u_0 (left) and of $\{\phi_k\}_{k=1}^{10}$ (right)

5.3 Basis generalization

Next, we introduce a first Neural Network $\underline{r}^{(1)} : [0,1]^2 \to \mathbb{R}^{K+1}$, meant to learn $\{\phi_k\}_{k=1}^K$ and u_0, through minimization of a properly defined $\mathcal{L}_1$ loss. Referring to the notation from paragraph 2.1:

$$\mathcal{L}_{fit} = \frac{1}{(K+1)|\mathcal{D}_{int}|} \sum_{\underline{x} \in \mathcal{D}_{int}} \left\| \underline{r}^{(1)}(\underline{x}) - (u_0(\underline{x}), \phi_1(\underline{x}), ...\phi_K(\underline{x})) \right\|_2^2$$

$$\mathcal{L}_{BC} = \frac{1}{(K+1)|\mathcal{D}_{BC}|} \sum_{\underline{x} \in \mathcal{D}_{BC}} \left\| \underline{r}^{(1)}(\underline{x}) - (u_0(\underline{x}), \phi_1(\underline{x}), ...\phi_K(\underline{x})) \right\|_2^2$$

$$\mathcal{L}_1 = \mathcal{L}_{fit} + \mathcal{L}_{BC}$$

where $||\underline{x} - \underline{y}||_2$ is the euclidean norm in $\mathbb{R}^{K+1}$. $\underline{r}^{(1)}$ features 2 hidden layers with 26 neurons each and $tanh(\cdot)$ as activation. Moreover, as the eigenvectors are normalized in L^2 discrete norm, we normalize also u_0, to balance the relevance of all the functions to be learned. This normalization is kept into account in the upcoming step.
The NN $\underline{r}^{(1)}$ is necessary both to reconstruct a solution of the PDE (as done in the next paragraph) and to compute its derivatives with respect to the domain coordinates (x, y). In this case, a training set split ratio of 90% is used.

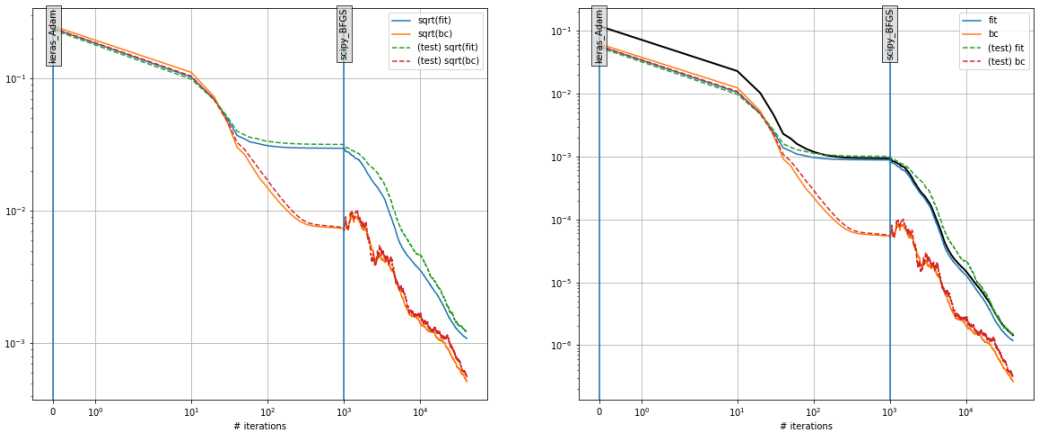

Figure 11: Training logs of $\underline{r}^{(1)}$ for $K = 10$

5.4 Map implementation

We now introduce a second Neural Network $\underline{r}^{(2)} : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}^{K+1}$, which is meant to map the parameters (μ, θ) into $\{r_k^{(2)}(\mu,\theta)\}_{k=1}^K \approx \{u_k(\mu,\theta)\}_{k=1}^K$ and $r_{K+1}^{(2)}(\mu,\theta) \approx b(\mu,\theta)$. With both the $\underline{r}^{(1)}$ and $\underline{r}^{(2)}$, we are able to express the manifold u_{POD} of solutions of the PDE depending on (μ, θ):

$$u_{POD}(x,y,\mu,\theta) = \sum_{k=1}^{K} r_k^{(2)}(\mu,\theta) r_k^{(1)}(x,y) + r_{K+1}^{(1)}(x,y) + r_{K+1}^{(2)}(\mu,\theta)$$

We train $\underline{r}^{(2)}$ minimizing $\mathcal{L}_2$, a distance between each numerical solution $u \in \mathbb{U}$ and the corresponding reconstruction from u_{POD}:

$$\mathcal{L}_{fit} = \frac{1}{|\mathbb{U}||\mathcal{D}_{int}|} \sum_{u_i \in \mathbb{U}} \sum_{(x,y) \in \mathcal{D}_{int}} [u_i(x,y) - u_{POD}(x,y,\mu_i,\theta_i)]^2$$

$$\mathcal{L}_{PDE} = \frac{1}{|\mathbb{U}||\mathcal{D}_{int}|} \sum_{u_i \in \mathbb{U}} \sum_{(x,y) \in \mathcal{D}_{int}} [Res(u_{POD},x,y,\mu_i,\theta_i)]^2$$

$$\mathcal{L}_{BC} = \frac{1}{|\mathbb{U}||\mathcal{D}_{BC}|} \sum_{u_i \in \mathbb{U}} \sum_{(x,y) \in \mathcal{D}_{BC}} [u_i(x,y) - u_{POD}(x,y,\mu_i,\theta_i)]^2$$

$$\mathcal{L}_2 = \mathcal{L}_{fit} + w_{PDE} \cdot \mathcal{L}_{PDE} + \mathcal{L}_{BC}$$

where (μ_i, θ_i) are the parameters of the PDE corresponding to each solution $u_i \in \mathbb{U}$, and $Res(u_{POD}, x, y, \mu, \theta)$ is the residual of $u_{POD}(\cdot, \mu, \theta)$ function on (x, y) with respect to the PDE with coefficients (μ, θ).
$\underline{r}^{(2)}$ features 2 hidden layers with 24 neurons each and $tanh(\cdot)$ as activation function.

5.5 Choice of K

The optimal number of eigenvectors to be used is chosen inspecting the proportional cumulative sum of the eigenvalues (representing the percentage of total variability explained in the statistical PCA framework), selecting a K above a certain treshold. It is set $w_{PDE} = 0$ since, with $w_{PDE} = 1$, the training results to be extremely longer (more than 10 times) and provides lower results, since the global loss $\mathcal{L}_2$ does not go below 10^{-3}.

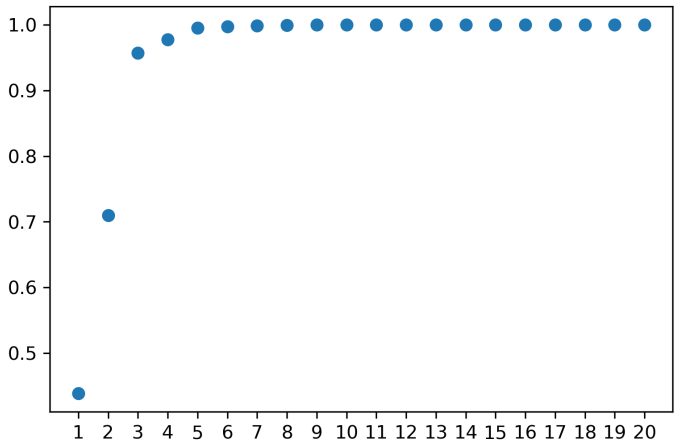

Figure 12: Proportional cumulative sum of the eigenvalues of PCA.

The cases $K = 3, 5, 10$ are analyzed in specific, seeing that just the first three eigenvectors are able to explain almost 96% of the total sum. Training of $\underline{r}^{(2)}$ is done for

these three levels of approximation, using a training dataset proportion of 80% always performing 500 epochs with *Adam* with $lr = 10^{-3}$ and 5000 with *BFGS* opimizers.

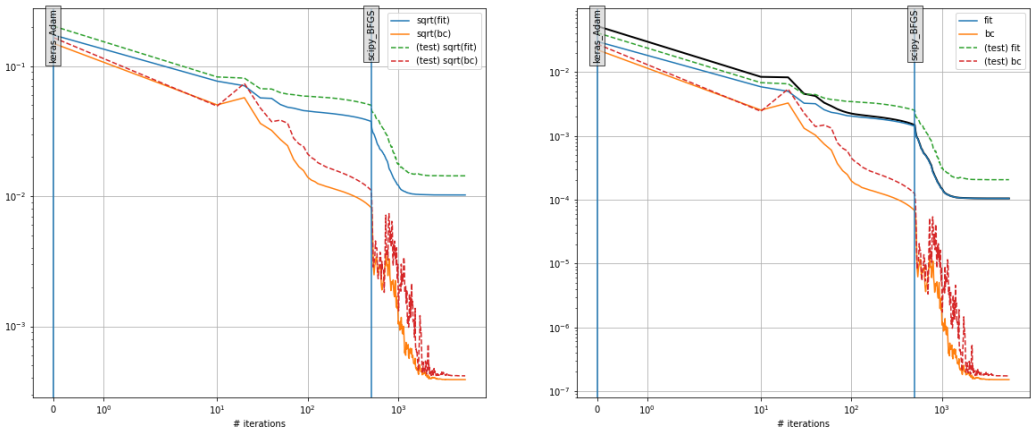

Figure 13: Training logs of $\underline{r}^{(2)}$ with $K = 3$.

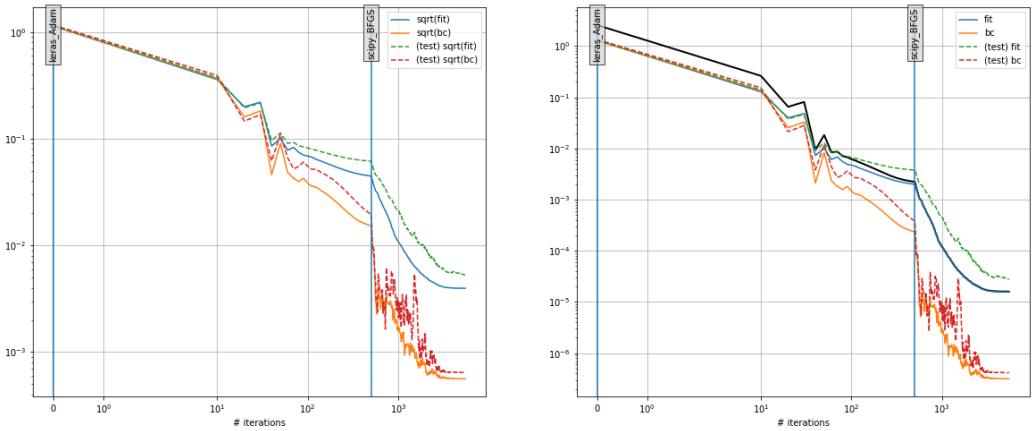

Figure 14: Training logs of $\underline{r}^{(2)}$ with $K = 5$.

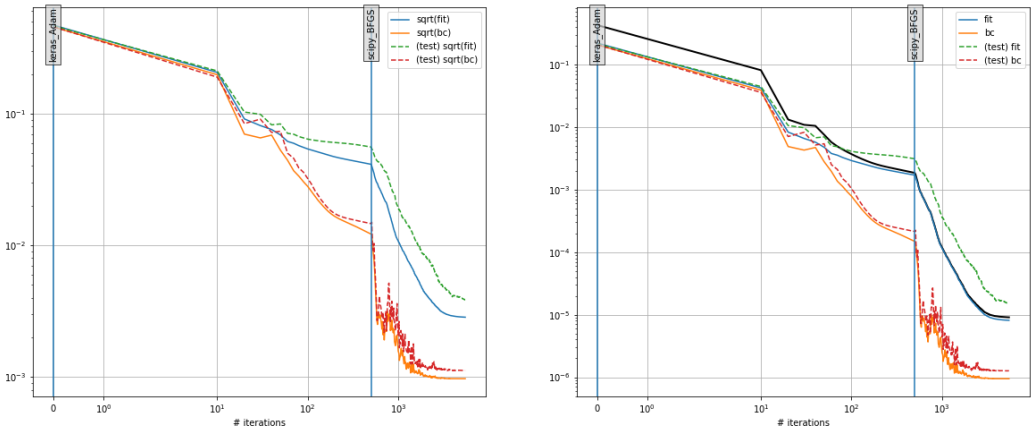

Figure 15: Training logs of $\underline{r}^{(2)}$ with $K = 10$.

The second and the third configurations are comparable in terms of accuracy and efficiency since, for $K = 5$ the global loss reaches $\mathcal{L}_2 = 1.600 \cdot 10^{-5}$ and the training time is $t = 293s$ whereas for $K = 10$ they are $\mathcal{L}_2 = 9.071 \cdot 10^{-6}$ and $t = 362$. The case $K = 3$ provides a significantly higher loss ($\mathcal{L}_2 = 1.046 \cdot 10^{-4}$) even if it is the faster ($t = 270s$). So, we choose $K = 5$ as good compromise between precison and training time.

K	prop. cumsum	$\mathcal{L}_2$	train time (s)
3	0.957	$1.046 \cdot 10^{-4}$	270
5	0.995	$1.600 \cdot 10^{-5}$	293
10	0.999	$9.071 \cdot 10^{-6}$	362

5.6 Results and comments

We can check graphically that the reconstruction is performed correctly.

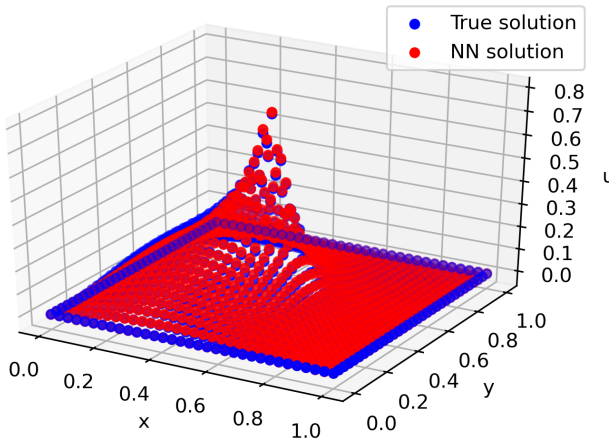

Figure 16: Comparison between a given numerical solution (blue) and its reconstruction through Neural Networks (red) for K = 5

We can observe graphically the effect of the parameters (μ, θ) on the coefficients $u_k(\mu, \theta)$. For instance, u_1 depends on μ only, while dependence of u_2 on θ decreases with μ.

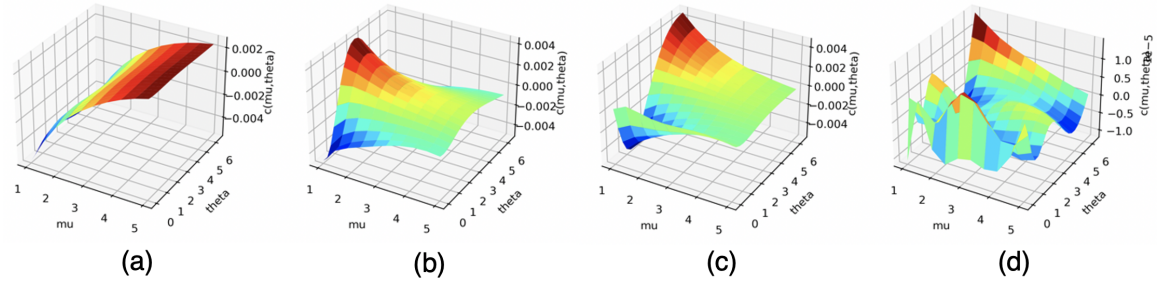

Figure 17: Values of 1^{st} coefficient (a), 2^{nd} coefficient (b), 3^{rd} coefficient (c), and bias (d) as functions of (μ, θ)

5.7 Comparison between POD and emulator

Now the representation performances of the POD and of the emulator of Chapter 3 are compared. Thanks to the additional hypotesis of linear structure of the manifold, it is reasonable to expect that POD needs a smaller network to obtain the same reconstruction accuracy of a normal emulator.
To address this problem, two trainings using identical networks (two layers of 24 neurons each) and identical hyperparameters (500 epochs with *Adam* with $lr = 10^{-3}$ and 5000 with *BFGS* opimizers, $w_{PDE} = 0$, training split of 80%) are done, the first with the POD with $K = 5$ and the second with the emulator.

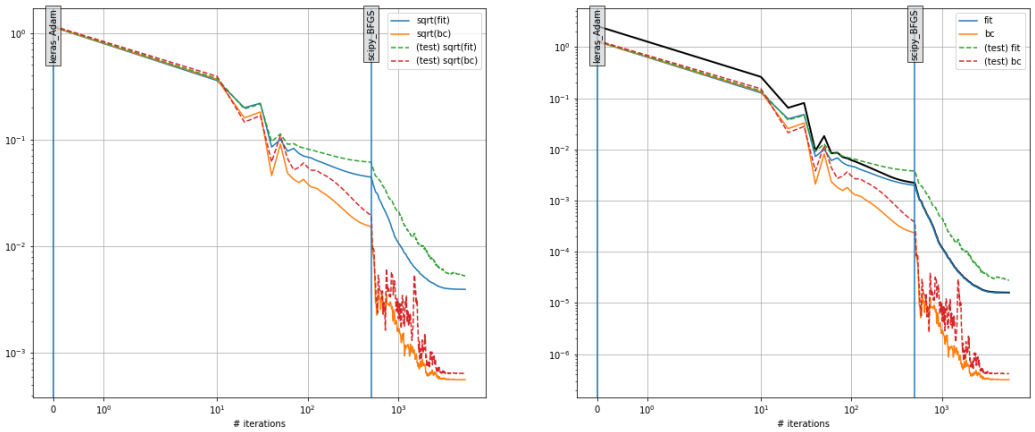

Figure 18: Training logs for the POD with $K = 5$.

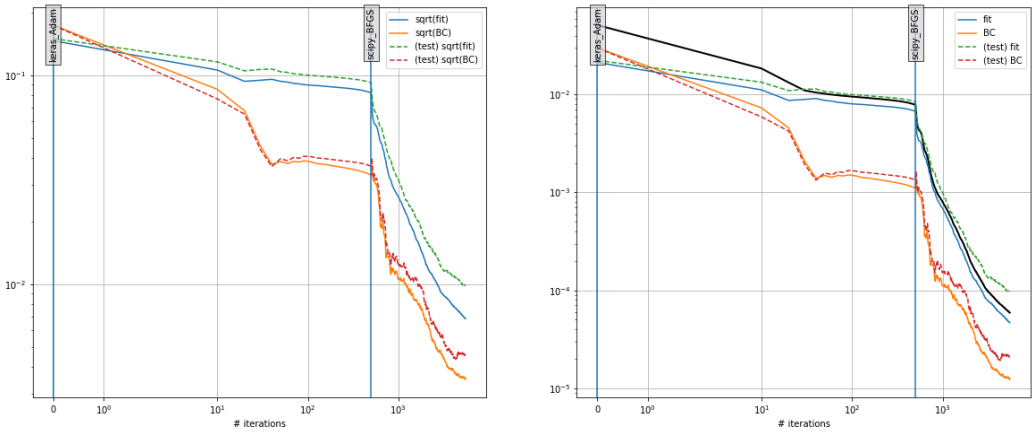

Figure 19: Training logs for the emulator.

The intuition is correct, since the POD reaches a significantly lower global loss $\mathcal{L}_2$ in almost three-quarters of the time.

	$\mathcal{L}_2$	train time (s)
POD	$1.600 \cdot 10^{-5}$	293
Emulator	$5.886 \cdot 10^{-5}$	381

6 Variable Geometry Setting

6.1 Objective

Up to now, our implementations of PINNs deal with a specific type of problems: the domain is fixed, while parameters of the equations change.
Thanks to the inherent flexibility of PINNs, we can very easily extend to a much wider class of problems. If we can provide a parametrization of the geometrical domain where we want to solve the equations, we can create a emulator which outputs solutions for different geometries. Combining this approach with the one used in previous experiments, the PINN can be trained to represent the manifold of PDE solutions defined not only from problems with different parameters, but also on various geometries.

6.2 Problem and Mesh Specifics

Our main focus is on the Monodomain Equation, occurring in modeling of cardiac electrical activity:

$$\begin{cases} \chi_m C_m \frac{\partial u}{\partial t} - \nabla \cdot (\Sigma \nabla u) + \chi_m I_{ion} = I^{ext} & \underline{x} \in \Omega, \ \ t \in (0, T] \\ (\Sigma \nabla u) \cdot \underline{\nu} = 0 & \underline{x} \in \partial\Omega, t \in (0, T] \\ u(\underline{x}, 0) = u_0(\underline{x}) & \underline{x} \in \Omega \end{cases}$$

The solution $u(\underline{x}, t)$ represents the transmembrane potential, generated by a current I^{ext} per unit volume. Concerning the other quantities, χ_m is the surface area-to-volume ratio, C_m is the membrane capacitance, Σ is the effective conductivity, and $I_{ion}(\underline{x}, t)$ are the ionic currents. Finally, $\underline{\nu}$ is the unit outward normal vector to $\partial\Omega$, and u_0 is an initial condition for u.
Keeping in mind that the problem we are interested in is solved in domains which model parts of the human heart, only meshes where middle points are raised up to 2.5H are considered. As can be seen in the following figure, some regions have a relatively high curvature, so we construct our meshes in order to capture these more extreme deformations.

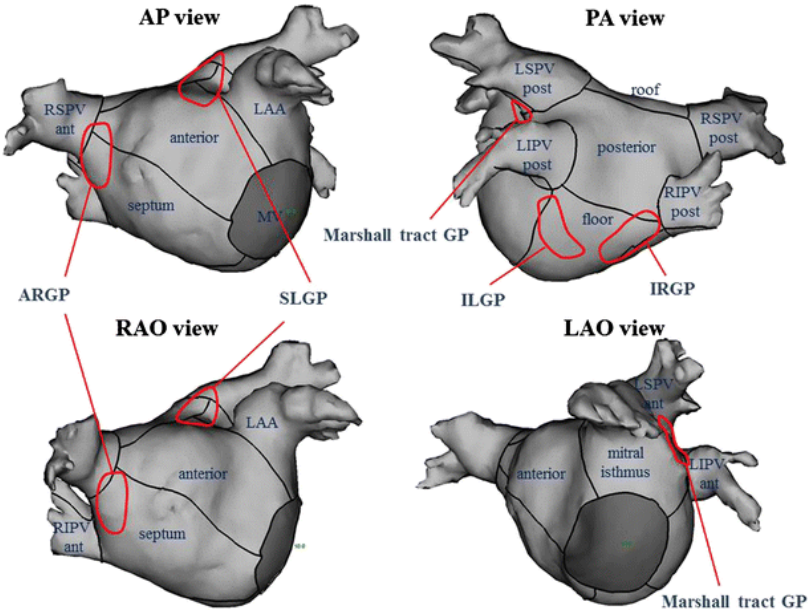

Figure 20: High Curvature Regions

6.3 Activation Time

In this application we focus on finding the Activation Time AT(x) for every point inside the mesh, rather than on finding the solution of the PDE.
We define the Activation Time for a point as:

$$AT(x) := \arg\max_t \frac{\partial u}{\partial t} \tag{1}$$

where u(x,t) is the solution of the PDE.
A key metric associated to each mesh is the Maximum Activation Time, defined as:

$$Max_{AT} := \max_x AT(x) \tag{2}$$

6.4 Geometrical Parametrization

In particular, in our experiments we focus on solving a problem for different configurations of the mesh.
The starting mesh is a flat parallelepiped with square basis of edge L and height H. The dimensions are 2L = 2.5, 2H = 0.3.
The deformation consists in raising or lowering the middle points of each side by a certain height, which can be different on different sides. For every side we define the *Geometrical Deformation Parameter* (GDP) as the height of the middle point normalized to the thickness H of the mesh. This gives rise to a big variety of meshes with different curvatures. Some examples are shown in the following figures.

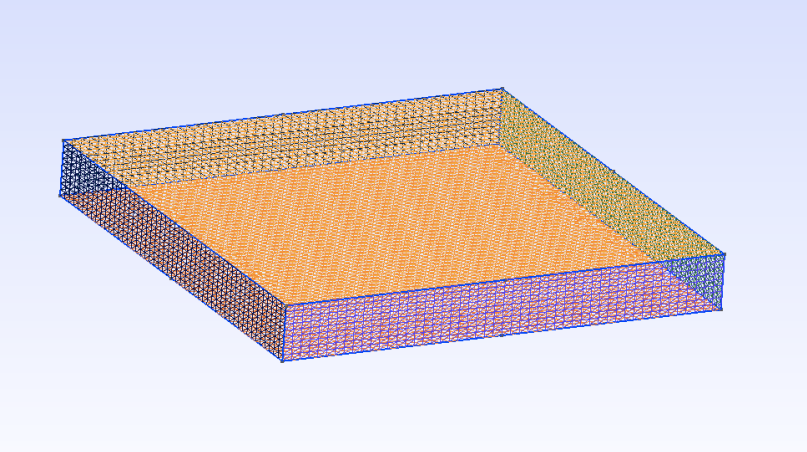

Figure 21: Flat Mesh

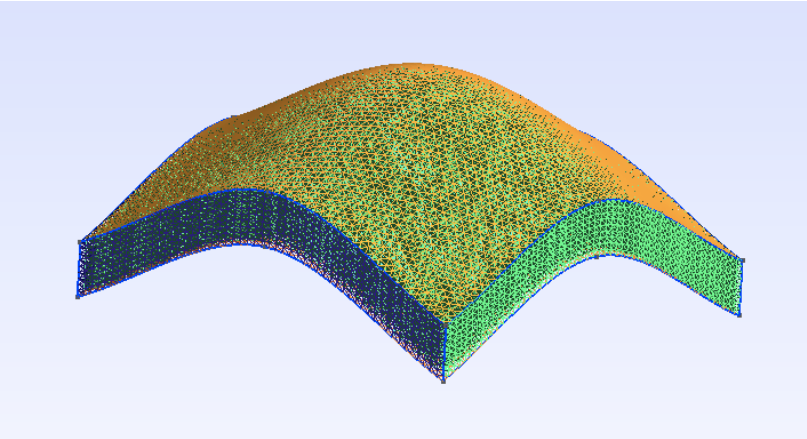

Figure 22: Curved Mesh

Relatively to the Activation Time definition, we approximate the following map:

$$\begin{aligned} AT_{NN}: \quad & [-L,L]\times[-L,L]\times[-H,H]\times\mathcal{P}\times\mathbb{R}^4 \to \mathbb{R} \\ & (x,y,z,\theta,GDP) \mapsto AT_{NN}(x,y,z,\theta,GDP) \end{aligned}$$

where $\mathcal{P}$ is the space of the parameters of the Monodomain Equation.
The x,y,z coordinates are the coordinates of a node in the flat mesh configuration, before it is deformed. We provide as input to the NN the GDP and the coordinates (x,y,z) in the reference configuration, and we want to approximate the AT in the corresponding point of the transformed geometry.
Another option, which is not investigated in this work, is to directly use the coordinates (x,y,z) in the transformed configuration.

6.5 Dataset Creation

Each mesh is comprised of 12500 nodes. To keep a reasonable dataset size, we randomly sample 1000 nodes from the mesh. Hence, the dataset is built as follows:

x	y	z	GDP_1	GDP_2	GDP_3	GDP_4	AT(x,y,z)
0	0	0	0.5	0.5	0.5	0.5	3

The ATs are normalized to the [0,1] range.
Training set is made up by 1000 nodes sampled from the meshes with GDP parameters which vary inside a certain range. The test sets are constructed as follows:

1. Test1 is made up by the same nodes of the training, with different GDP values. This way we measure the performance of the PINN at learning mesh deformation.

2. Test2 instead is made up by new nodes and all the GDP values. This way we measure the performance of the PINN at learning how the solution behaves both in different parts of the mesh and with different GDP values.

6.6 Loss Definition

Of course, since we focus only on the AT, the loss only consists of the Fit loss, while the PDE loss is dropped. The training is done using the same Neural Network architecture as in section 3.2.

6.7 Domain with one degree of freedom

We start off choosing meshes with one degree of freedom only, that is with all middle points raised to the same height. GDPs in training set have 40 values chosen uniformly from a sample of 50 values in the range [0, 2.5]. The remaining 10 are used for testing.

The results from our trainings show that the PINN is able to reach a loss of 10^{-2}, then starts to overfit.

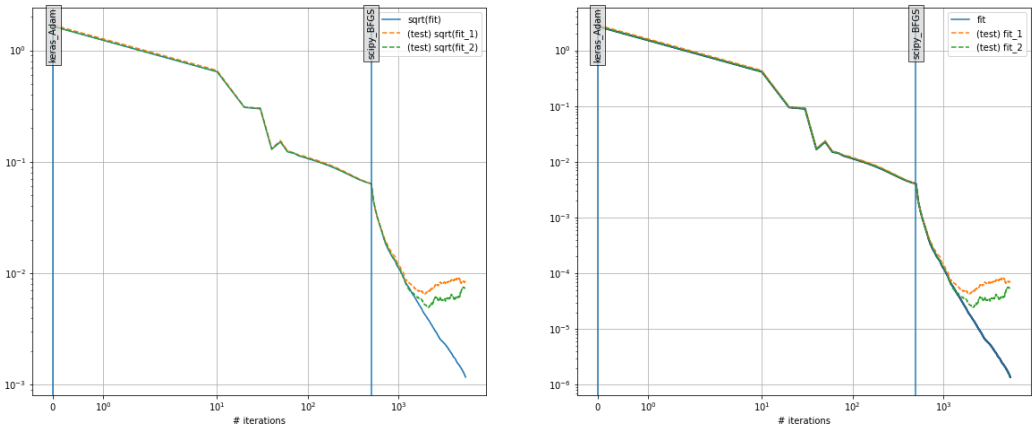

Figure 23: Training log

Given the $AT(x)$, it is possible to compute the Max_{AT} both from the Matlab solution and from the PINN. The PINN seems to do a good job at approximating the curve, as can be seen in the following figure:

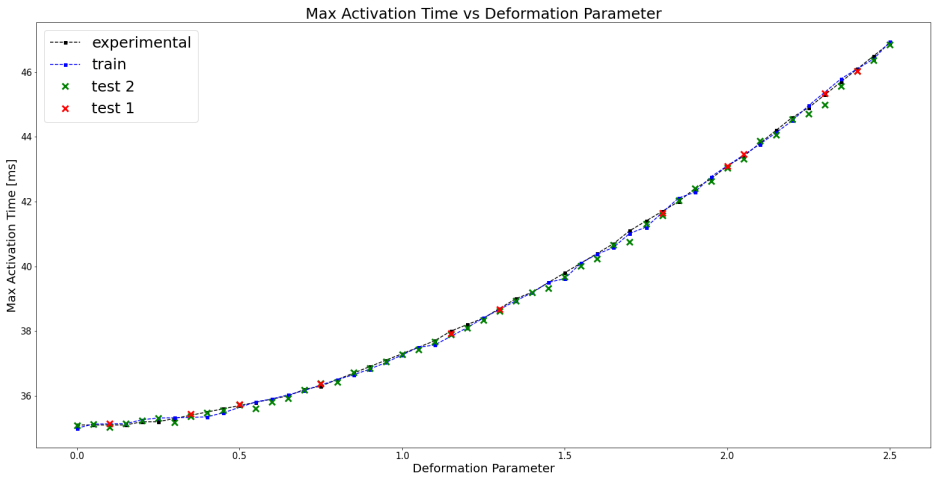

Figure 24: Max_{AT} with different GDPs

6.8 Domains with two degrees of freedom

To test the PINN performances, we now enlarge the possible space of geometrical domains.
In particular, we also consider meshes with saddle points. This is obtained fixing two facing sides to the same height, which we allow to be different in the two couples of sides. The GDP couple values are also sampled from the range [-2.5, 2.5], giving rise to more complex geometries such as the following:

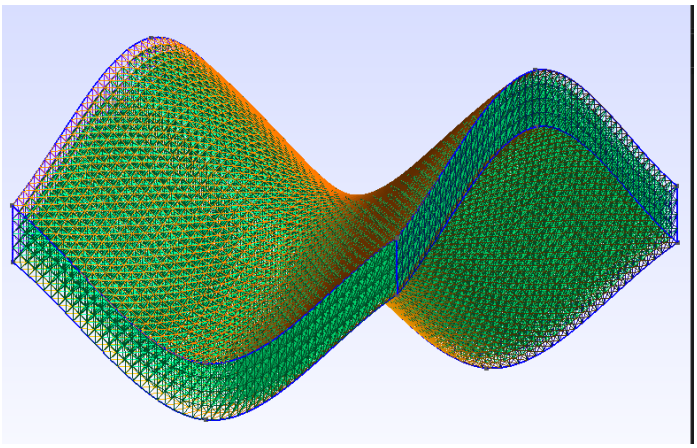

Figure 25: Complex Geometry

In the previous section, 50 values are sampled from the range [0, 2.5]. Now we sample

150 values from the range [-2.5, 2.5] $\times$ [-2.5, 2.5].
The values are randomly sampled to overcome the massive increase in the sample space, instead of taking a cartesian product resulting in 10000 different data points. For this reason, given the same PINN architecture, we expect lower performances. The results from our trainings show that the lowest value of the loss function reached before starting to overfit is 4×10^{-2}. As expected, this is higher than the best performance obtained before, equal to 10^{-2}.

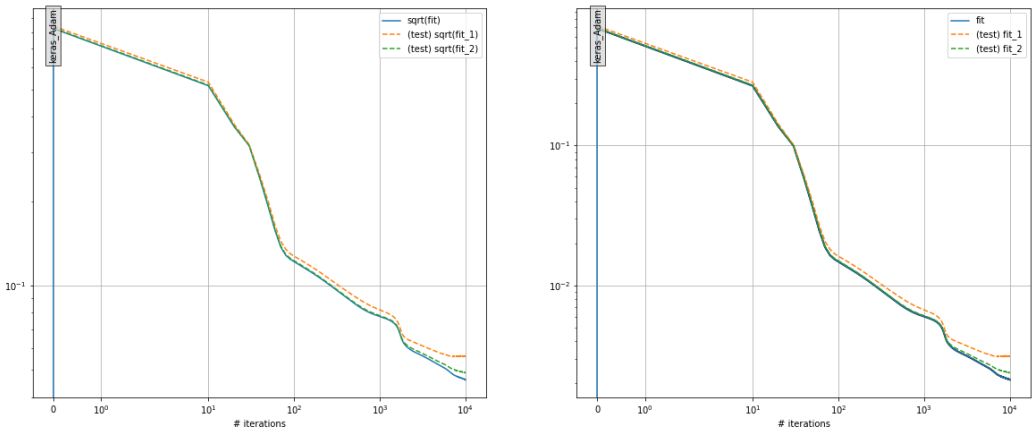

Figure 26: Training log

Many other PINN architectures are tested, for example using 50 neurons per layer instead of 20, or doubling the number of nodes included in the dataset. None of these tweaks seems to greatly affect performances, as the loss has the same order of mangitude. This suggests that the current limitation is the dimension of the dataset, specifically the number of sampled couples of the GDP which we were not able to increase, due to computational issues.

6.9 Geometrical and Physical Parameters

Finally, we now include both geometrical parameters and physical parameters. We vary both the mesh deformation, as in previous experiments, and the conductivity of the material. The conductivity parameter is sampled in the range [0.5, 1.5], in order to increase it or decrease it up to 50% with respect to the original value.
The conductivity parameter directly influences the AT, which becomes smaller if the conductivity is raised. The mesh deformations are symmetric in all 4 sides, as in the first section. The training can be found in the following figure.

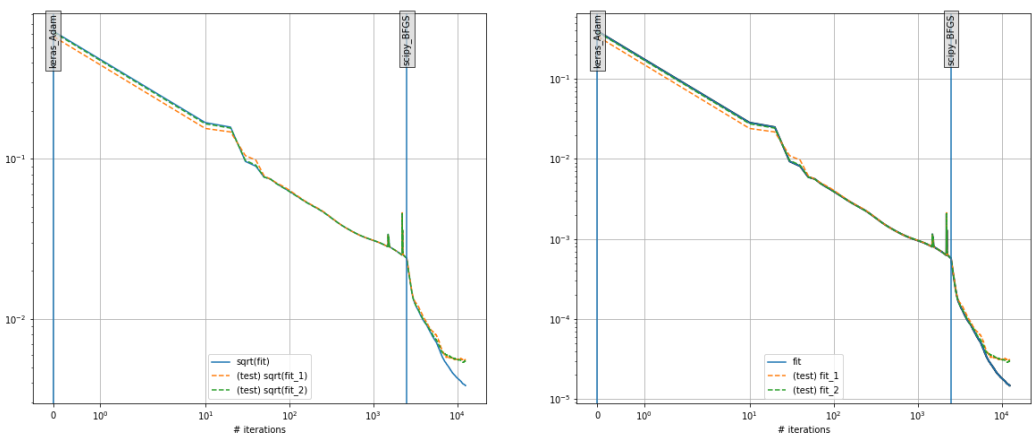

Figure 27: Training log

Once again, the PINN is able to reach values lower than 10^{-2}, stopping at 7×10^{-3} before starting to overfit.

7 Conclusions

In summary, we conclude that the implemented methods based on *Physics Informed Neural Networks* achieve accurate results with respect to both the Advection-Diffusion problem and the Monodomain equation.
With the simple Advection-Diffusion case study, we focus on understanding how the introduction of the loss based on the PDE residual affects the training and the overall accuracy of the method. Our training results suggest that introducing such loss largely decreases the residual of the PDE (Section 3.4) and provides a further check of the quality of the solution (Section 3.5), but it doesn't affect greatly the fitting performances. Moreover, using this loss results in a much higher computational cost as, in our experiment, it triplicates training time. Finally, an unbalanced weighting of the loss results to be detrimental (Section 3.4).
As observed in Section 3.5 and 6.7, a key aspect for a successful training is the choice of a rich enough training dataset: many experiments show that enlarging its size proves to be more effective than increasing the complexity of the model (Section 3.5). In our tests PINNs prove to be a very flexible tool, able to learn manifolds depending not only on different physical parameters but also on different geometrical configurations (Section 6.8). The drawback is that there is no standard procedure to control a priori the error through the proper design of the architecture. The error, instead, can actually be estimated only empirically after the training and this makes the tuning of the hyperparameters more difficult and time consuming.To limit the computational burden, the considered case studies feature a small number of parameters. Hence, as further development, performances of PINNs could be explored in context of high dimensionality of the space of parameters, inspecting the most effective way to construct the dataset and design the Neural Network architecture.

References

[1] Francesco Regazzoni, Luca Dede, and Alfio Quarteroni. Machine learning for fast and reliable solution of time-dependent differential equations. *Journal of Computational physics*, 397:108852, 2019.

[2] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.

[3] Julien Weiss. A tutorial on the proper orthogonal decomposition. *AIAA Aviation 2019 Forum*, 2019.

[4] Piero Colli Franzone, Luca Franco Pavarino, and Simone Scacchi. *Mathematical cardiac electrophysiology*, volume 13. Springer, 2014.

[5] TensorFlow: Large-scale machine learning on heterogeneous systems. `http://tensorflow.org/`. Software available from tensorflow.org.

[6] Francesco Regazzoni. Nisaba. `https://sci-learning.gitlab.io/nisaba/`.