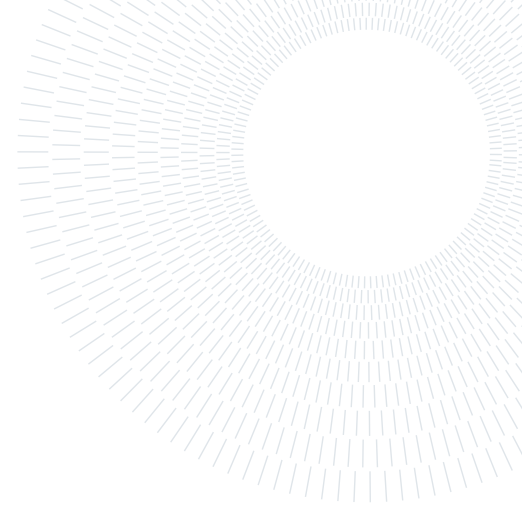




**POLITECNICO**  
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE  
E DELL'INFORMAZIONE



# A fully-comprehensive library for Physics-Informed Deep Learning under uncertainty

## Project Report

## Advanced Programming for Scientific Computing

---

### Authors:

Giulia Mescolini  
Luca Sosta

### Professor:

Prof. Luca Formaggia

### Tutors:

Prof. Andrea Manzoni  
Prof. Stefano Pagani

### Academic Year:

2021-2022

**Abstract:** in this project, we developed a new library implementing several variants of the Bayesian Physics-Informed Neural Networks (B-PINNs) method, a framework to tackle physical problems with Neural Networks that are able to take into account information coming from partial differential equations and quantify the uncertainty around their prediction.

To implement this strategy, we designed from scratch a modular and flexible pipeline; in our design, indeed, on one hand we wanted to mirror the peculiarities of the network tasks involved in the B-PINN framework into the hierarchy of the classes, while on the other hand we aimed at proposing a structure on which a variety of functionalities for different tasks could be inserted.

In the proposed release, indeed, we provided different variants of the various aspects of the overall process, ranging from data management to training algorithms, among which we implemented Adam, Hamiltonian Monte Carlo, Variational Inference and Stein Variational Gradient Descent.

The library design is also predisposed to further extensions, because the blocks representing the different components that detach from the main skeleton are thought to be defined by few case-specific behaviors and not to have heavy constraints on the functionalities, as validated when implementing algorithms with distinct routines and structures involved or different problems.

Finally, we proposed a showcase of results to highlight the main features of method and library: algorithm comparison, introduction of PDE residuals into Neural Networks, portability to high dimension and quality of results that can be obtained with fine-tuning of model parameters.

**Key-words:** B-PINNs, Uncertainty Quantification, Scientific Learning

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Project Overview	4
1.2	Report Structure	5
<b>2</b>	<b>Methods</b>	<b>6</b>
2.1	An overview on Neural Networks	6
2.1.1	Core Structure	6
2.1.2	Backpropagation	8
2.1.3	Optimizers	9
2.1.4	Training Neural Networks	11
2.2	Physics Informed Neural Networks	12
2.2.1	Structure of a PINN	12
2.2.2	Optimization Problems	13
2.2.3	Dataset for PINNs	14
2.3	Bayesian Neural Networks	15
2.3.1	Likelihood Distribution	16
2.3.2	Prior Distribution	16
2.3.3	Posterior Distribution	17
2.4	Bayesian Physics Informed Neural Networks	17
2.4.1	Physical Likelihood	18
2.4.2	Prior for Inverse Problems	18
<b>3</b>	<b>Algorithms</b>	<b>19</b>
3.1	Hamiltonian Monte Carlo	19
3.1.1	Theoretical Foundations	20
3.1.2	Algorithm	21
3.1.3	Parameters' choice	22
3.2	Variational Inference	22
3.2.1	Theoretical Foundations	22
3.2.2	Algorithm	23
3.2.3	Parameters' choice	24
3.3	Stein Variational Gradient Descent	24
3.3.1	Theoretical Foundations	24
3.3.2	Algorithm	25
3.3.3	Parameters' choice	25
<b>4</b>	<b>Code Overview</b>	<b>26</b>
4.1	Working Environment	26
4.1.1	Interpreter	26
4.1.2	Virtual Environment	26
4.2	Libraries	26
4.2.1	Built-in Packages	26
4.2.2	External Packages	27
4.2.3	TensorFlow	27
4.3	Object Oriented Features	28
4.3.1	Inheritance	28
4.3.2	Abstract Base Class	29
4.3.3	Dataclass	30
4.3.4	Iterators	30
4.3.5	Property	31
4.4	Repository Structure	31
4.4.1	Configuration - <code>config</code> Folder	31
4.4.2	Datasets - <code>data</code> Folder	33
4.4.3	Outputs - <code>outs</code> Folder	33

<b>5</b>	<b>Source Code</b>	<b>35</b>
5.1	Main Files . . . . .	36
5.1.1	Main Pipeline . . . . .	36
5.2	Parameters Handling . . . . .	36
5.2.1	Configuration files . . . . .	36
5.2.2	Command line arguments . . . . .	37
5.2.3	Param object . . . . .	37
5.3	Data Generation . . . . .	37
5.3.1	Domain creation . . . . .	37
5.3.2	Multi-domain creation . . . . .	38
5.3.3	Boundary Points . . . . .	38
5.3.4	Data Configuration . . . . .	39
5.4	Pre-Processing . . . . .	39
5.4.1	Dataset creation . . . . .	40
5.4.2	Data normalization . . . . .	40
5.4.3	Data Batching . . . . .	40
5.5	Equations . . . . .	41
5.5.1	Operators . . . . .	41
5.5.2	Abstract Equation . . . . .	42
5.5.3	Problems Implemented . . . . .	42
5.6	Model . . . . .	43
5.6.1	Parameters object - Theta . . . . .	43
5.6.2	Parameters and forward pass - CoreNN . . . . .	45
5.6.3	Physics enforcement - PhysNN . . . . .	46
5.6.4	Training functionalities - LossNN . . . . .	46
5.6.5	Prediction phase - PredNN . . . . .	47
5.6.6	Final wrapper - BayesNN . . . . .	48
5.7	Optimizers . . . . .	49
5.7.1	Training interface . . . . .	49
5.7.2	Algorithm abstract class . . . . .	49
5.7.3	Adam optimizer . . . . .	50
5.7.4	Hamiltonian Monte Carlo . . . . .	50
5.7.5	Variational Inference . . . . .	51
5.7.6	Stein Variational Gradient Descent . . . . .	51
5.8	Postprocessing . . . . .	52
5.8.1	Storage . . . . .	52
5.8.2	Plotter . . . . .	53
<b>6</b>	<b>Results</b>	<b>54</b>
6.1	Regression Problem . . . . .	54
6.1.1	Adam . . . . .	54
6.1.2	HMC . . . . .	55
6.1.3	SVGD . . . . .	56
6.1.4	VI . . . . .	56
6.2	Damped Harmonic Oscillator Problem . . . . .	57
6.2.1	NN vs PINN . . . . .	57
6.3	Multi-dimensional domain . . . . .	57
6.3.1	Adam 2D cosine . . . . .	57
6.4	HMC Showcase . . . . .	58
6.4.1	Regression Problem - sine . . . . .	58
6.4.2	Oscillator Problem . . . . .	59
6.4.3	Poisson/Laplace Problem - cosine . . . . .	60
6.4.4	Poisson/Laplace Problem - sine . . . . .	61
<b>7</b>	<b>Conclusions</b>	<b>62</b>
<b>A</b>	<b>Installation Guide</b>	<b>63</b>

# 1. Introduction

## 1.1. Project Overview

The integration of Machine Learning-based techniques in Scientific Computing is nowadays becoming more and more popular. A challenging scenario is the one of uncertainty quantification, where the efficiency of the Neural Networks can be help in overcoming the computational costs of demanding approaches based on PDE solvers.

Bayesian Physics-Informed Neural Networks (B-PINNs) represent a way to address such problems, thanks both to the inclusion of the residuals of differential equations during the learning process and to training algorithms which reconstruct a distribution of the Network's output rather than a single prediction.

This method, recently proposed in few papers such as [11] and [7], is an evolution of Physics-Informed Neural Networks (PINNs), a Deep Learning framework for solving problems involving partial differential equations ([9], [10]); B-PINNs take inspiration from the theory of Bayesian Statistics to empower the PINNs method and enable the introduction of uncertainty quantification.

In this project, we implemented a library for Bayesian Physics-Informed Machine Learning designed to be modular, flexible and open to plugins. In the state-of-the-art of this niche application of PINNs, it is indeed more common to find implementations of a specific Bayesian training method applied to one differential problem; therefore, the goal we set was the development of a library with a wide offer of features.

Given the project aim, the design of some components of the library played a crucial role. The backbone of the method was thought as a rigid modular structure, built in such a way that the blocks for the desired application could be assembled into it without affecting the skeleton. Each block can be defined with its own characteristic features, that are a small number and do not introduce heavy constraints on the functionalities, so that it is easy to generate new blocks for different methods or problems in which we may be interested.

The library's characteristics are then highlighted by a selection of applications, aimed at showing the main pillars: the set-up of four training algorithms very different in structure and intuition one from each other, the contribution of the introduction of the physical information into the model and the library portability to higher dimensions.

For the method performance on the specific applications, the parameters' tuning plays a crucial role. Though in the project we focused more on the horizontal expansion of library feature, we stressed on performance quality for the case of one specific Bayesian training algorithm with sustainable computational time and interesting challenges in parameters' choices, developing a wider variety of test cases with fine-tuned method options.

## 1.2. Report Structure

The theoretical foundations of the proposed implementation are presented in [section 2](#), with the explanation of the building blocks of a B-PINN. Starting from the basics of neural networks and from deterministic training algorithms described in [subsection 2.1](#), we then illustrate the theory behind Physics-Informed Machine Learning in [subsection 2.2](#). Then, we present the learning mechanisms of Bayesian Neural Networks in [subsection 2.3](#) and illustrate how to combine them with PINNs to obtain B-PINNs in [subsection 2.4](#).

In [section 3](#) we deepen into the non-deterministic training algorithms implemented. We illustrate the theoretical background behind them and their procedures, problematizing and discussing the choice of parameters but without deepening into the technical implementation details (postponed to [section 5](#)). Each subsection is dedicated to one algorithm: [subsection 3.1](#) to Hamiltonian Monte Carlo, [subsection 3.2](#) to Variational Inference and [subsection 3.3](#) to Stein Variational Gradient Descent.

In [section 4](#) we first present and motivate the coding choices in terms of environment ([subsection 4.1](#)) and libraries ([subsection 4.2](#)). Then, we devote [subsection 4.3](#) to Object-Oriented Programming in Python, focusing on the features implemented in the project. Finally, [subsection 4.4](#) presents the structure of the repository and the content of the configuration, data and outputs files.

In [section 5](#), we illustrate the source code: we describe the three executable scripts devote specific sections to the various modules, whose tasks are related to the implementation of the B-PINNs method or to post-processing utilities such as UQ and performance evaluation.

In [section 6](#), we present the application of the library implemented to a series of test cases, with different datasets and training configurations. The showcase of applications is organized in such a way to highlight separately the main features of the library: in [subsection 6.1](#), we focus on a comparison of the training algorithms on the same task. Then, in [subsection 6.2](#), we show the power of the inclusion of the physical information on the Damped Harmonic Oscillator problem. In [subsection 6.3](#) we exhibit an example in 2D dimension and finally in [subsection 6.4](#) we present the results on several applications obtained with a massive fine-tuning of a model trained with HMC.

Finally, in [section 7](#), we draw conclusions on this project and suggest further developments.

## 2. Methods

### 2.1. An overview on Neural Networks

#### 2.1.1. Core Structure

With the term *Artificial Neural Network* (NN), we refer to a computing system inspired by the biological neural networks constituting animal brains. These tools were first introduced in the second half of the XX century, and nowadays they are witnessing an amazing popularity.

They fall into the framework of *Deep Learning* (DL), which, in turn, is one of the specialization of *Machine Learning* (ML), and they enable to process the information in a surprisingly complete way. The main strength of NNs is that, while classical ML predictions (e.g. linear regression) work well only when the features (i.e. the inputs or a hand-made combination of them) are good, they can directly learn the features from the data and therefore fall in the *Feature Learning*<sup>1</sup> context.

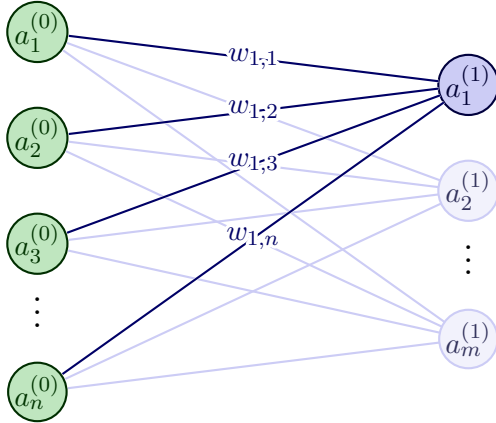
#### The neuron

In order to understand how these tools work, we first introduce their basis components: **neurons**.

They consists in processing units, which:

1. take as input a finite number of signals  $a_j$  from input data or other previous neurons outputs;
2. compute their weighted sum with weights  $w_{i,j}$ , where  $j$  refers to the signal and  $i$  identifies the neuron;
3. subtract a threshold by adding a bias term  $b_i$ , which is a special weight relative to the neuron itself;
4. produce one single output  $z_i$  and apply a non-linear activation function  $\phi$  obtaining  $a_i$ .

It is fundamental that the activation function  $\phi$  is *non-linear*; else, the whole Neural Network would be only a highly factorized linear function of the input data.



$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma \left[ \begin{pmatrix} w_{1,1}^{(0)} & w_{1,2}^{(0)} & \dots & w_{1,n}^{(0)} \\ w_{2,1}^{(0)} & w_{2,2}^{(0)} & \dots & w_{2,n}^{(0)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(0)} & w_{m,2}^{(0)} & \dots & w_{m,n}^{(0)} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix} \right]$$

$$\mathbf{a}_1 = \sigma \left( \mathbf{W}_0^T \mathbf{a}_0 + \mathbf{b}_0 \right), \quad \mathbf{W}_0 \in \mathbb{R}^{n \times m}$$

Figure 1: The neuron, basic component of a Neural Network.

#### Fully Connected Neural Networks

Neurons are stacked in layers, and in *Fully Connected Neural Networks* (FCNN), as the one involved in our project, each neuron is connected to all the neurons of the next and the previous layer.

Now, let us define the structure of a Neural Network (NN), which enables the understanding of the already described process of features learning from data. As Figure 2 shows, the three main sections are:

- **Input Layer**, receiving  $N_i$  inputs. Input data can be, for example, domains, images, time-series ...;
- **Hidden Layers**:  $L - 1$  layers, each one with  $N_h$  nodes. Their task is to find suitable features;
- **Output Layer**, depending on the desired task (e.g. regression, classification ...), returns  $N_o$  output.

Using this notation, the output of the neuron  $i$  in layer  $l$ , which depends only on the activations  $\{a_j^{(l-1)}\}_{j=1}^m$  of the previous layers, is:

$$a_i^{(l)} = \phi \left( \sum_{j=1}^{N_{l-1}} w_{j,i} a_j^{(l-1)} + b_i^{(l)} \right) \quad (1)$$

<sup>1</sup>In Machine Learning, Feature Learning or Representation Learning is a set of techniques that allows a system to automatically discover the representations needed for feature detection or classification from raw data. This replaces manual feature engineering and allows a machine to both learn the features and use them to perform a specific task.

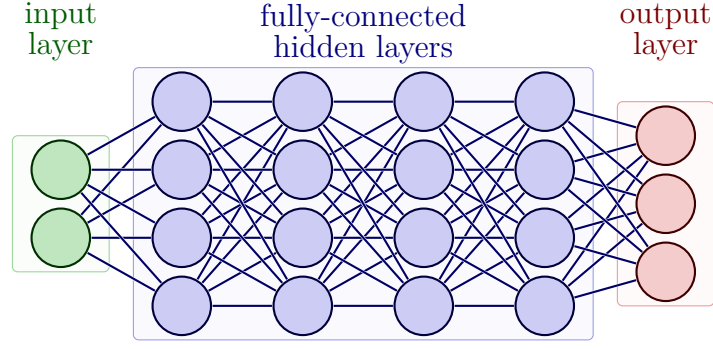


Figure 2: General structure of a Neural Network

## Activation Functions

In this section we present some common choices for the activation function  $\phi$ :

- **Sigmoid**: this function is smooth everywhere, but it presents a weak point:  $|\sigma'(x)| \ll 1$  for  $|x| \gg 1$ , therefore the gradient is *vanishing*, and the learning can be slow for deep NNs.
- **Hyperbolic Tangent**: balanced version of the Sigmoid, which is differentiable and has a similar  $S$ -shape. It has the advantage to push the input values to 1 and  $-1$  instead of 1 and 0.
- **Rectified Linear Unit (ReLU)**: in this case, the gradient is not vanishing for  $x > 0$ , since the derivative is 1, but the function is not differentiable at  $x = 0$  and for  $x < 0$  the derivative is 0.
- **Leaky ReLU**: this slightly modified version of ReLU solves the issue of the 0-gradient for  $x < 0$ .
- **Swish**: another variation on the ReLU, which, according to experiments, tends to work better than ReLU on deeper models across a number of challenging datasets.

Activation	Analytical Expression	Derivative of Activation
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1 - \sigma(x))$
Hyperbolic Tangent	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$	$\tanh'(x) = 1 - \tanh^2(x)$
ReLU	$\text{ReLU}(x) = \max\{0, x\}$	$\text{ReLU}'(x) = \mathbb{1}_{\mathbb{R}_+}(x)$
Leaky ReLU	$\text{LR}(x) = \max\{\alpha x, x\}$	$\text{RL}'(x) = \alpha + (1 - \alpha)\mathbb{1}_{\mathbb{R}_+}(x)$
Swish	$S(x) = x\sigma(x)$	$S'(x) = S(x) + \sigma(x)(1 - S(x))$

Table 1: Common Activation Functions for Neural Networks and their derivatives

## Structure of our Neural Network

Reproducing the architecture of the Neural Network proposed in [11], in the project we built a *feed-forward fully connected* Neural Network with 2 hidden layers having the *same number of nodes* (usually 16 or 50), as sketched in Figure 3. The activation function chosen for hidden layers can be specified by the user; in the test cases proposed, we selected the *swish* function.

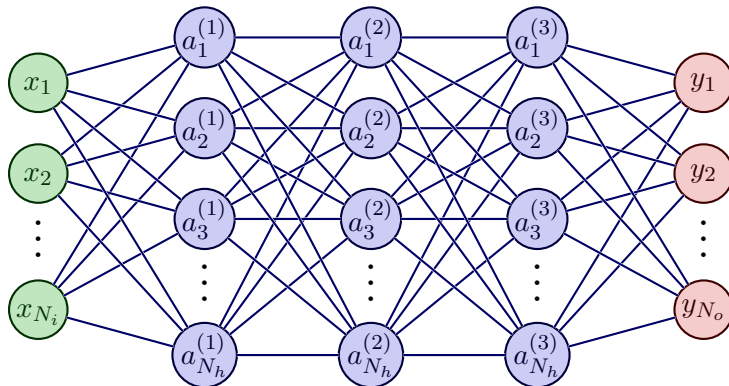


Figure 3: Architecture of a neural network with  $N_i$  input, 3 hidden layers and  $N_o$  output.

### 2.1.2. Backpropagation

In Machine Learning, the algorithm through which the NN learns the right weights is generally the so-called *backpropagation*. First of all, let us define the *loss* function  $\mathcal{L}(f)$ , which provides a quantitative measure of the performance of the NN by expressing the distance between the output estimated by the model  $\mathbf{y}^* = f(\mathbf{x})$  and its true value  $\mathbf{y}$  for all the data. The optimal solution is obtained by minimizing the Loss function, with respect to all weights and biases:

$$w_{i,j}^{*,(l)}, b_i^{*,(l)} = \underset{w_{i,j}^{(l)}, b_i^{(l)}}{\operatorname{argmin}} \mathcal{L}(f) \quad (2)$$

To perform this task, we have first to choose the *optimizer*, i.e. the optimization algorithm.

There exist several possibilities, with different memory consumption and time needed to reach convergence.

Whatever algorithm we choose, we need the computation of  $\nabla \mathcal{L}$ , in particular:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \left( \frac{\partial \mathcal{L}_n}{\partial \mathbf{W}_l} \right)_{i,j} \quad \frac{\partial \mathcal{L}_n}{\partial b_i^{(l)}} = \left( \frac{\partial \mathcal{L}_n}{\partial \mathbf{b}_l} \right)_i \quad \forall (i, j, l) \quad (3)$$

This means that the number of derivatives that must be computed is of the order of  $\mathcal{O}(K^2 L)$ , supposing that we have  $K$  nodes in each of the  $L$  layers for the sake of simplicity; therefore, finding an efficient manner to compute the derivatives jointly becomes fundamental.

### General Notations

To illustrate the backpropagation algorithm, let us introduce the following notations:

- **Weight matrices:**  $\mathbf{W}_l$ , where  $\mathbf{W}_1 \in \mathbb{R}^{N_i \times N_h}$ ,  $\mathbf{W}_l \in \mathbb{R}^{N_h \times N_h} \ \forall 2 \leq l \leq L-1$ ,  $\mathbf{W}_L \in \mathbb{R}^{N_h \times N_o}$ .
- **Bias vectors:**  $\mathbf{b}_l$ , where  $\mathbf{b}_l \in \mathbb{R}^{N_h} \ \forall 1 \leq l \leq L$ ,  $\mathbf{b}_L \in \mathbb{R}^{N_o}$ .
- **Trainable Parameters:**  $\boldsymbol{\theta}$  usually used to indicate weights and biases of all layers together.

With this notation,

$$\begin{aligned} \mathbf{a}_1 &= f^{(1)}(\mathbf{a}_0) = \Phi(\mathbf{z}_0) = \Phi(\mathbf{W}_1^T \mathbf{a}_0 + \mathbf{b}_1) \\ \mathbf{a}_2 &= f^{(2)}(\mathbf{a}_1) = \Phi(\mathbf{z}_1) = \Phi(\mathbf{W}_2^T \mathbf{a}_1 + \mathbf{b}_2) \\ &\vdots \\ \mathbf{a}_L &= f^{(L)}(\mathbf{a}_{L-1}) = \Phi(\mathbf{z}_{L-1}) = \Phi(\mathbf{W}_L^T \mathbf{a}_{L-1} + \mathbf{b}_L) \end{aligned}$$

Therefore, writing all in a more compact form, the output of the Neural Network is:

$$\mathbf{y}^* = f(\mathbf{x}) \text{ with } f = f^{(L)} \circ f^{(L-1)} \circ \dots \circ f^{(2)} \circ f^{(1)} \text{ and } \mathbf{x} = \mathbf{a}_0, \mathbf{y}^* = \mathbf{a}_L \quad (4)$$

### Loss Functions

All the test cases presented in this work fall into the regression framework, hence our dataset generally consist in a sequence of  $N$  couples of inputs  $\{\mathbf{x}\}_{n=1}^N$  and targets  $\{\mathbf{y}\}_{n=1}^N$  vectors.

The most common used loss function for regression problems is the *Mean Squared Error (MSE)*, defined as:

$$\text{MSE} = \frac{1}{N} \sum_{n=1}^N (\mathbf{y}_n - \mathbf{y}_n^*)^2$$

Using the MSE as *Loss Function* and exploiting the model we have built (written in Equation 4) to predict over a given dataset, we obtain the following equation:

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{n=1}^N \mathcal{L}_n \text{ where } \mathcal{L}_n = (\mathbf{y}_n - f(\mathbf{x}_n))^2 = (\mathbf{y}_n - f^{(L)} \circ f^{(L-1)} \circ \dots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n))^2 \quad (5)$$

It is important to notice that the loss can be seen as a function of all trainable parameters  $\boldsymbol{\theta}$  and inputs  $\mathbf{x}$  or as function of neural network outputs  $\mathbf{y}^*$ ; the last interpretation will be very useful when dealing with backpropagation of the errors, because we can divide the contribution from the model itself and the loss. In both case targets  $\mathbf{y}$  are involved because we are in a supervised learning<sup>2</sup> context but they are fixed values.

<sup>2</sup>Supervised Learning (SL) is a machine learning paradigm for problems where the available data consists of labelled examples, meaning that each data point contains features (covariates) and an associated label. The goal of supervised learning algorithms is learning a function that maps feature vectors (inputs) to labels (output), based on example input-output pairs.



## Chain Derivatives

As said above, the quantities we are interested on are the partial derivatives in Equation 3; we will use the chain rule for the derivative computation for compound fractions and define as auxiliary quantities the outputs  $\mathbf{z}_l$  and activation  $\mathbf{a}_l$  of each layer  $l$ :

$$\mathbf{z}_l := \mathbf{W}_l^T \mathbf{a}_{l-1} + \mathbf{b}_l = \mathbf{W}_l^T \Phi(\mathbf{z}_{l-1}) + \mathbf{b}_l \quad (6)$$

Writing component-wise the derivatives, we get that:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{N_l} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} a_i^{(l-1)} = \delta_j^{(l)} a_i^{(l-1)} \quad (7)$$

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \sum_{k=1}^{N_l} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)} \quad (8)$$

$\delta_j^{(l)}$  represents the error due to the neuron  $j$  of layer  $l$  and can be easily computed in the following way:

$$\begin{aligned} \delta_j^{(l)} &= \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \sum_{k=1}^{N_{l+1}} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_{k=1}^{N_{l+1}} \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \\ \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} &= \frac{\partial}{\partial z_j^{(l)}} \sum_{i=1}^{N_l} w_{i,k}^{(l+1)} \Phi(z_i^{(l)}) = \sum_{i=1}^{N_l} w_{i,k}^{(l+1)} \frac{\partial \Phi(z_i^{(l)})}{\partial z_j^{(l)}} = \sum_{i=1}^{N_l} w_{i,k}^{(l+1)} \Phi'(z_i^{(l)}) \delta_{i,j} = \Phi'(z_j^{(l)}) w_{j,k}^{(l+1)} \\ \delta_j^{(l)} &= \sum_{k=1}^{N_{l+1}} \delta_k^{(l+1)} \Phi'(z_j^{(l)}) w_{j,k}^{(l+1)} \quad \forall 1 \leq j \leq N_l, \quad \forall 1 \leq l \leq L-1 \end{aligned} \quad (9)$$

Therefore, we could split the computation made by NN into two distinct phases:

- **Forward Pass:** the network is crossed from the input layer to the output layer with Equation 6 and generates the desired output of the model  $\mathbf{y}_n^*$  for each sample  $\mathbf{x}_n$ .

$$\mathbf{z}_l = \mathbf{W}_l^T \mathbf{a}_{l-1} + \mathbf{b}_l \quad l = 1, \dots, L$$

$$\mathbf{a}_l = \Phi(\mathbf{a}_{l-1}) \quad l = 1, \dots, L$$

$$\mathbf{a}_0 = \mathbf{x}_n \in \mathbb{R}^{N_i}, \mathbf{y}_n^* = \mathbf{a}_L \in \mathbb{R}^{N_o}$$

- **Backward Pass:** now the NN is crossed in the opposite direction, computing recursively errors related to single neurons  $\{\delta_l\}_{l=1}^L$  of all the layers with Equation 9, then it is transferred to single weights and biases with Equation 7 and Equation 8. In vector form<sup>3</sup>, we get:

$$\begin{aligned} \delta_L &= -2(\mathbf{y}_n - \mathbf{y}_n^*) \odot \Phi'(\mathbf{z}_L) \\ \delta_l &= (\mathbf{W}_{l+1} \delta_{l+1}) \odot \Phi'(\mathbf{z}_l) \quad l = (L-1), \dots, 1 \\ \frac{\partial \mathcal{L}_n}{\partial \mathbf{W}_l} &= \delta_l \odot \mathbf{a}_l \quad \frac{\partial \mathcal{L}_n}{\partial \mathbf{b}_l} = \delta_l \quad l = 1, \dots, L \end{aligned}$$

It is important to notice that the choice of the loss function is involved directly only on the computation of the last error term  $\delta_L$  and that derivative of the activation function  $\Phi$  is computed a lot of times, therefore we want it to be easily differentiable as it happens for the common choices shown in Table 1.

### 2.1.3. Optimizers

For a model, learning means solving the optimization problem of finding the values of the parameters that minimize the loss function as stated in Equation 2. Since Neural Networks often contain thousands of parameters, the exact solution is not available in closed form, hence we proceed with algorithms that reduce step after step the loss function in  $N$  iterations, also called *epochs*.

The choice of the optimizer, affects both the accuracy and the learning speed of the model. We now present some common available choices for classical Neural Networks, which will be useful to understand the more complex optimizers presented later in this project for Bayesian Neural Networks.

<sup>3</sup>Those Equations can be translated into vector form using the elementwise product  $\odot$  i.e. the Hadamard Product

## Gradient Descent

Every optimization algorithm, to which we refer as *optimizer*, updates all trainable parameters of the NN with the simple update rule  $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \delta\boldsymbol{\theta}^{(t)}$ , so the difference among optimizers consist in the term  $\delta\boldsymbol{\theta}$ .

The *Gradient Descent* (GD) algorithm originates from the simplest idea: the gradient of a function points to the direction of largest increase of the function, hence, to minimize the function, we take a step in the opposite direction. So,  $\delta\boldsymbol{\theta} = -\gamma\mathcal{L}(\boldsymbol{\theta})$ , where  $\gamma$  is called *Learning Rate* (LR) and is an hyper-parameter<sup>4</sup> of the NN.

---

**Algorithm 1:** Gradient Descent (GD)

---

**Initialization:** learning rate  $\gamma > 0$ , number of iterations  $N_{epochs}$ , initial state  $\boldsymbol{\theta}^{(0)}$   
**for**  $t = 1, \dots, N_{epochs}$  **do**  
    Compute the gradient of the full loss function  $\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)})$   
    Update the trainable parameters:  $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \gamma\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)})$

---

## Stochastic Gradient Descent

In ML, many loss functions are formulated as a sum or mean over a function computed on single data samples, so a common practice is to use the batched version of the optimizer to speed up the computing.

To obtain it, we evaluate for each iteration the loss function not on the whole dataset, but only on a subset of it, called *batch* (and its cardinality is called *batch size*).

In the case of *Stochastic Gradient Descent* (SGD) we take the limit case and we choose a batch size of one and compute its loss contribute, denoted as  $\mathcal{L}_n$ ; for example, for MSE, it is equal to  $(\mathbf{y}_n - \mathbf{y}_n^*)^2$ .

---

**Algorithm 2:** Stochastic Gradient Descent (SGD)

---

**Initialization:** learning rate  $\gamma > 0$ , number of iterations  $N_{epochs}$ , initial state  $\boldsymbol{\theta}^{(0)}$   
**for**  $t = 1, \dots, N_{epochs}$  **do**  
    Sample uniformly  $n \in \{1, 2, \dots, N_{samples}\}$   
    Compute the gradient of the loss on the  $n^{th}$  sample  $\nabla\mathcal{L}_n(\boldsymbol{\theta}^{(t-1)})$   
    Update the trainable parameters:  $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \gamma\nabla\mathcal{L}_n(\boldsymbol{\theta}^{(t-1)})$

---

In this way, we avoided the computation of the full gradient every time, but we lost the guarantee that the update is pointed towards the minimum at each iteration. However, we expect that on average the ensemble direction is the right one.

## Adam Optimizer

Among the first order optimizers, the most popular choice is *Adam*, an empowered version of Stochastic Gradient Descent. This is an *adaptive learning rate method*, meaning that *Adam* is able to learn an optimal LR, which represents the width of displacements for each step of gradient descent. Its main strength is combining the pros of other two common optimizers derived from SGD:

- *AdaGrad*, that maintains a per-parameter LR improving performance on problems with sparse gradients;
- *RMSProp*, whose LR are adapted basing on the average of recent magnitudes of the gradients for the weights, so the algorithm performs well on online and non-stationary problems.

*Adam* can be defined as a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments and it is considered the best benchmark in the choice of optimizers.

---

**Algorithm 3:** Adaptive Moment Estimation (Adam)

---

**Initialization:**  $\alpha, \beta_1, \beta_2, \epsilon$ , number of iterations  $N_{epochs}$ , initial state  $\boldsymbol{\theta}^{(0)}$ , momenta  $\mathbf{m}^{(0)}, \mathbf{v}^{(0)}$   
**for**  $t = 1, \dots, N_{epochs}$  **do**  
     $\mathbf{m}^{(t)} = \beta_1\mathbf{m}^{(t-1)} + (1 - \beta_1)\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)})$   
     $\mathbf{v}^{(t)} = \beta_2\mathbf{v}^{(t-1)} + (1 - \beta_2)(\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)}))^2$   
     $\hat{\mathbf{m}} = (1 - \beta_1)^{-1}\mathbf{m}^{(t)}, \hat{\mathbf{v}} = (1 - \beta_2)^{-1}\mathbf{v}^{(t)}$   
     $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \alpha \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}} + \epsilon}}$

---

We reported in [algorithm 3](#) *Adam*'s general scheme, using the following notation:

- $\mathbf{m}, \mathbf{v}$  are the first and second moment of the gradient, initialized at  $\mathbf{0}$ ;
- $\hat{\mathbf{m}}, \hat{\mathbf{v}}$  are unbiased estimators of  $\mathbf{m}, \mathbf{v}$ ;
- $\alpha, \beta_1, \beta_2$ , and  $\epsilon$  are fixed hyper parameters.

---

<sup>4</sup>A more detailed explanation of parameters and hyper-parameters will be given in [subsubsection 2.1.4](#)

### 2.1.4. Training Neural Networks

In this section we present the traditional pipeline that is followed to improve the performances for the most of Deep Learning models used in literature. The success of the prediction task depends on model parameters; indeed, artificial Neural Networks are called *parametric models* because they capture all the information about their predictions within a finite set of parameters and they need to be trained to select the best parameters to represent the map we want to be learned by the model.

First of all, we need to clarify some distinctions among parameters, because a group of parameters is updated during learning, while others have to be fixed to a value. We therefore make a distinction into:

- **Trainable parameters:** they appear directly in the loss evaluation and typically gradients are taken with respect of this set of parameters. The mechanism of their choice is usually referred to as *training*. In most of the cases, they are the ensemble of weights and biases of the NN, but there can be exceptions. For example when dealing with inverse problems (presented in the [dedicated section](#)), there can be other trainable parameters called  $\lambda$ , or in the transfer learning context not all NN parameters are trainable.
- **Hyper parameters:** they are all the other parameters necessary to build the model; their choice is called *tuning* and it is usually performed by hand with literature-based knowledge or through grid search-like algorithms. Some relevant examples are all the optimizers' parameters (e.g. learning rate for GD), the number of epochs, the architecture of the NN itself . . .

## Data Processing

A high quality of data can play a key role in the model performance. This is the reason why it may be useful to work with transformed versions of data, more friendly for the NN range of optimal domain, obtained through a preliminary *pre-processing*, or ask the network for an output that is easier to reconstruct, which then need to be restored into the desired one with some *post-processing*.

Moreover, it is an important practice in ML to partition the dataset into *training*, *validation* and *test* subset, because for the evaluation of the model performance it is not fair to use the data already seen and used to improve the model. This means that we cannot use the whole dataset for training, but we should reserve a part of them for assessing model goodness in an unbiased way; this could sound be penalizing in ML as it common to wish to train with as many data as possible, but in our application, as we will state in [subsection 2.2](#), we rely also on the physical information from partial differential equations rather than on big number of data, hence we did not encounter problems of shortage of data.

## Learning Workflow

After the pre-processing of the dataset, a precise method should be follow in order to code a ML model for the desired task which is capable of both accurate and generalizable predictions:

1. **Training phase:** updating the NN's trainable parameters in order to minimize the loss on the training dataset, during subsequent  $N_{epochs}$  iterations, increasing *accuracy*. This is the only phase in which the optimizer comes into play, and it requires the heaviest computations (mainly due to the gradients).
2. **Validating phase:** evaluating model's *generality* to ensure that the model did not overfit on the training set, that is, learn also noise or peculiarities in the data provided. This enables us to tune the hyperparameters in order to improve the model, also with the aid of performance metrics aside from the loss.
3. **Testing phase:** using unseen data is able to provide the true unbiased *evaluation* of model performance.

## Approximation Power

A possible rising question could be: *what are the functions that Neural Networks are able to approximate?* The comforting answer, when the activation  $\phi$  is sigmoid-like, is given by Barron's Approximation Result.

**Theorem 2.1.** *Given a function  $f : \mathbb{R}^D \rightarrow \mathbb{R}$ , let  $\hat{f}$  be its Fourier Transform.*

$$\text{If } \exists C > 0 \text{ such that } \int_{\mathbb{R}^D} |\omega| |\hat{f}(\omega)| d\omega \leq C. \text{ Then } \forall n \geq 1, \exists f_n, \{c_j\}_{j=1}^n \subset \mathbb{R}, \text{ with:}$$
$$f_n(x) = \sum_{j=1}^n c_j \phi(x^T w_j + b_j) + c_0 \text{ so that } \int_{|x| \leq r} |f(x) - f_n(x)|^2 dx \leq \frac{(2Cr)^2}{n}$$

This is equivalent to state that every NN with a single hidden layer and a sigmoid-like activation function  $\phi$  can approximate with arbitrarily small error (in the sense of the  $L^2$  norm any continuous function on a compact set, provided that a sufficient number of hidden neurons are employed. There exist a similiar result also for the pointwise approximation error, obtained by Shekhtman; in this case, the activation function is the ReLU.

## 2.2. Physics Informed Neural Networks

In this section we present *Physics Informed Neural Networks* (PINN), which represent a DL framework for solving problems involving partial differential equations. They have been introduced in the last 5 years, and in their development Karniadakis, Perdikaris and Raissi have been pioneers (see [9], [10]). Nowadays, these tools are being empowered even outside academia, as can be seen from solutions such as NVIDIA Modulus (see [here](#)). PINNs consist in Neural Networks that are trained to solve *supervised learning* tasks while respecting laws of physics governed by ordinal (ODE) or partial (PDE) differential equations.

In standard ML, Neural Networks are common in “big data” frameworks, when plenty of observations are available to train our model, but in some applications data may be extremely costly and difficult to evaluate, such as in the medical field. Therefore, the exploitation of the physical knowledge that we have on the phenomena, represented by differential models, plays a key role, since it enables us to work even in a “small data” regime.

### 2.2.1. Structure of a PINN

We now illustrate the general structure of Physics-Informed Neural Networks. Given a generic partial differential equation (linear/nonlinear, steady/unsteady ...), we introduce the following notation:

- $\Omega$  is a bounded domain in  $\mathbb{R}^n$  and  $[0, T]$  is the time domain;
- $\mathbf{x}$  and  $t$  are, respectively, space and time variable;  $\mathbf{x} \in \Omega$ ,  $t \in [0, T]$ , used as inputs;
- $\mathbf{u}(\mathbf{x}, t)$  and  $\mathbf{f}(\mathbf{x}, t)$  are the solution and parametric field, which can depend in general on space and time;
- $\mathcal{N}$  is the fully known differential operator applied to  $\mathbf{u}$ ;
- $\lambda$  represents the physical parameters of the problem which could be unknown.

The analytical formulation of the problem to solve is therefore:

$$\begin{cases} \mathcal{N}(\mathbf{u}(\mathbf{x}, t); \lambda) = \mathbf{f}(\mathbf{x}, t) & \text{in } \Omega \times [0, T] \\ + \text{ boundary conditions} & \text{on } \partial\Omega \times [0, T] \\ + \text{ initial conditions} & \text{on } \Omega \times \{0\} \end{cases} \quad (10)$$

In Figure 4 we show how PINNs work: a classical NN is used to make a prediction on the solution  $\mathbf{u}$  with its forward pass, then, during loss evaluation (MSE, in the example) a new regularization term is added: the *residual loss* denoted as  $MSE_R$ . The evaluation of residual is not made by NN layers but is performed by computing partial derivatives with automatic differentiation<sup>5</sup> on model outputs with respect to the inputs and then combine in the differential operator.

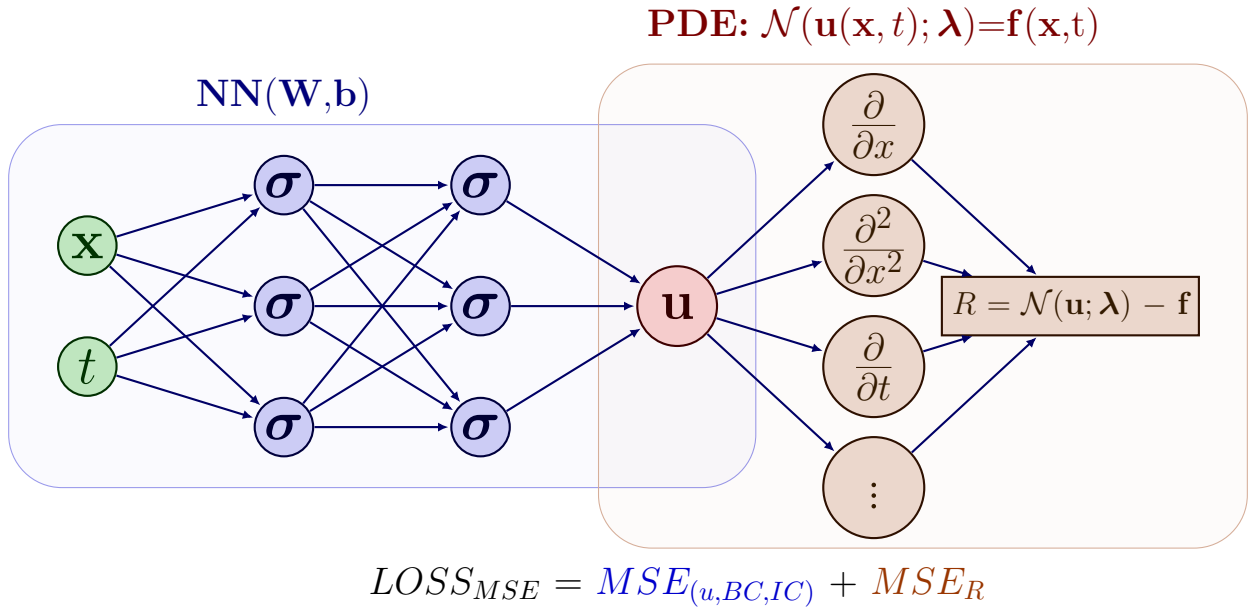


Figure 4: Structure of a Physics-Informed Neural Network.

<sup>5</sup>Automatic Differentiation (AD) is a set of techniques to evaluate the derivative of a function specified by a computer program. It exploits the fact that every computer program, regardless of its complications, executes a sequence of arithmetic operations and elementary functions.

### 2.2.2. Optimization Problems

Introducing a ML model on this kind of analytical problem allow us to consider the latter equivalent to solving an optimization problem. In our project we deal with a steady PDE-constrained problem, such as:

$$\begin{cases} \min_u \mathcal{L}(u) = \|u - t\|_{L^2} \\ s.t. \mathcal{N}(u; \lambda) = f \text{ in } \Omega \end{cases}$$

where we adopt the following notation:

- $\mathcal{L}$  the quantity to optimize;
- $u$  the variable of minimization;
- $t$  the target function we want to learn;
- $f$  the parametric field in the PDE.

### Discrete Problem

The first step is to discretize the  $t$  and  $f$  functions, because we will work with sparse measurements of them. Those will be available only in some points, denoted with  $\mathbf{x}$ , so we approximate the continuum problem with vectors  $\mathbf{t} = \mathbf{t}(\mathbf{x})$  and  $\mathbf{f} = \mathbf{f}(\mathbf{x})$  with variable of minimization  $\mathbf{u} = \mathbf{u}(\mathbf{x})$ . The discretized problem reads as:

$$\begin{cases} \min_{\mathbf{u}} \mathcal{L}(\mathbf{u}) = MSE(\mathbf{u}, \mathbf{t}) \\ s.t. \mathcal{N}(\mathbf{u}; \lambda) = \mathbf{f} \quad \forall \mathbf{x} \end{cases} \quad (11)$$

Note that the  $L^2$ -norm is now replaced by its discrete counter-part the MSE which play the role of empirical mean, computed on the set of observation  $\mathbf{x}$ .

Using a NN as model, we can rewrite the first equation in Equation 11 in term of parameter  $\theta = (\mathbf{W}, \mathbf{b})$  using it as the variable of minimization, resulting in:

$$\min_{\theta} \mathcal{L}(\mathbf{u}), \text{ where } \mathbf{u}(\mathbf{x}; \theta) = \mathcal{NN}(\mathbf{x}; \theta)$$

Finally, the optimization part of our problem depends only on the trainable parameters of the NN. The remaining part of Equation 11 is the differential constraint; to solve that, the PINN section of our model comes into play.

### Differential Residuals

If the couple given by a function  $\mathbf{u}$  and a parametric field  $\mathbf{f}$  is the solution of the PDE, then we can define the residual of the operator as:

$$R := \mathcal{N}(\mathbf{u}; \lambda) - \mathbf{f} \quad (12)$$

Therefore, we can impose  $R = 0$  through a loss function for values of  $\mathbf{u}$  and  $\mathbf{f}$  that approximate the exact ones with our prediction and help the model learning functions reasonably close to the correct ones by including  $MSE(\mathbf{R}, \mathbf{0})$  in  $\mathcal{L}$  during the training phase.

Boundary and initial conditions can be easily implemented just by considering domain points  $\mathbf{x}$  belonging to the parabolic boundary. More details on the explicit formulation of the losses, regularization and on the dataset needed, will be provided in subsection 2.2.3 and case-specific sections.

### Direct and Inverse Problems

When dealing with PINNs, we can consider both *direct* or *inverse* problems and is important to remark the difference, because we need to treat the regularization term with the introduction of PINN.

The difference between the two consists basically in the knowledge that we have on some parameters  $\lambda$ , representing physical coefficients (such as a diffusion coefficient): in the direct setting they are known, while in the inverse one they need to be estimated from other measurements available.

In *direct problems* the value of physical parameters  $\lambda$  is known, so it can be overlooked as it was part of the differential operator  $\mathcal{N}$ , leading us to the following formulation:

$$\begin{cases} \min_{\theta} \mathcal{L}(\mathbf{u}) = MSE(\mathbf{u}, \mathbf{t}) + R(\mathbf{u}) \\ \text{with } \mathbf{u}(\mathbf{x}; \theta) = \mathcal{NN}(\mathbf{x}; \theta) \end{cases} \quad (13)$$

In the *inverse setting*, the model physical parameters  $\boldsymbol{\lambda}$  are unknown and are reconstructed by the model. In this case, the optimization problem solved by the PINN is a generalization of the above one: in fact, it is only needed to include the set of physical parameters of interest  $\boldsymbol{\lambda}$  among the residuals  $R$ , leading to:

$$\begin{cases} \min_{\boldsymbol{\theta}, \boldsymbol{\lambda}} \mathcal{L}(\mathbf{u}; \boldsymbol{\lambda}) = MSE(\mathbf{u}, \mathbf{t}) + R(\mathbf{u}; \boldsymbol{\lambda}) \\ \text{with } \mathbf{u}(\mathbf{x}; \boldsymbol{\theta}) = \mathcal{NN}(\mathbf{x}; \boldsymbol{\theta}) \end{cases} \quad (14)$$

### 2.2.3. Dataset for PINNs

Training and validating a model for a PINN follows a similar schedule as the one stated in [subsubsection 2.1.4](#), but with the addition of some peculiarity, in fact computing the forward pass is a much more easy task than the whole evaluation of loss needed with the insertion of the residuals.

For validation and testing we can rely in the same way as before on *validation points* and *test points* without any issue but we can distinguish three important subdatasets within the *training dataset*:

**Fitting Points**, i.e. points inside the domain  $\Omega$  where we impose a value for the solution (obtained by the known analytical expression or, for example, from numerical simulations) and act in the classical training set fashion. Those are a few because we work in a *small data regime* and can be limited only to certain subdomain of  $\Omega$  where is easier to take real measurements of  $u_{ex}$ . The loss to add is in this case:

$$\mathcal{L}_{fit} = \frac{1}{N_{fit}} \sum_{n=1}^{N_{fit}} (u(x_n, t_n) - u_{ex}(x_n, t_n))^2$$

**Collocation Points**, being the points inside the domain  $\Omega$  where we impose the PDE constraints, by requiring to minimize the residual of the equation. Referring to [Equation 12](#), our residual is:

$$\mathcal{R}(\mathbf{x}, t) = \mathcal{L}(\mathbf{u}(\mathbf{x}, t)) - \mathbf{f}(\mathbf{x}, t)$$

and using MSE loss function as in [Figure 4](#) we aim at minimizing the loss:

$$\mathcal{L}_{col} = \frac{1}{N_{col}} \sum_{n=1}^{N_{col}} \mathcal{R}(\mathbf{x}_n, t_n)^2$$

with  $\{(\mathbf{x}_n, t_n)\}_{n=1}^{N_{col}}$  denoting the elements of the set of collocation points. Those points are totally available because there is no need of target function but they affect in the bigger way the computational cost for the evaluation of gradients with respect to inputs.

**Boundary Points**, on which we impose the exact value when we have a Dirichlet Boundary condition, or the minimization of the residual of the equation describing the Neumann condition; depending on that they are more similar to the first or second set presented respectively. For example, with a Dirichlet Boundary Condition of the form  $u = f$  on  $\Gamma_D$  we add the loss:

$$\mathcal{L}_{bnd} = \frac{1}{N_{Bnd}} \sum_{n=1}^{N_{Bnd}} (u(x_n, t_n) - f(x_n, t_n))^2$$

Those points come from physical equations, involving only  $f$  and not  $u_{ex}$ , so we can use a lot of them without worries. The same can be done for the initial condition

### Combination of the Losses

The final loss function that we want the net to minimize includes the effect of all the losses introduced above, with a weighted combination of them. For example, with  $M$  losses  $\mathcal{L}_i$ , we have

$$\mathcal{L} = \sum_{i=1}^M q_i \mathcal{L}_i$$

with  $q_i$  indicating the weight of the  $i$ -th loss.

Note that  $\mathbf{q}$  is an hyperparameter of the net, and its choice will be crucial for the performance.

## 2.3. Bayesian Neural Networks

*Bayesian Neural Networks* (BNNs) represent a way to introduce *uncertainty quantification* (UQ) in the framework of modern deep-learning methods, which constitute incredibly powerful tools to tackle challenging problems, but they often operate as black boxes on the significance of their predictions.

### Uncertainty Quantification

Up to now, we have presented methods to reconstruct functions with Neural Networks which are not able to provide information on the *confidence* of the prediction. With NNs, we are indeed unable to quantify the uncertainty of the prediction, because we obtain from the model just one sample of the solution, namely the network output correspondent to the given input, using as network parameters the ones chosen after the process of loss minimization with the already presented methods.

The main idea is to replace  $\theta := \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^L$  with a suitable joint probability distribution and change the way we perform the forward pass. For the validation and test phase  $N$  new instances of  $\theta$  are sampled and they generate  $N$  different independent predictions, which are actually the output of a classical NN whose parameters are each member the set at a time. Finally, having  $N$  samples of the output, we take the relevant statistics and use the mean for prediction and the variance for the UQ.

The real burden occurs during the model training, because it is mandatory to have *ad hoc* algorithms for the BNN; this discussion is postponed to [section 3](#), that is entirely devoted to describe the task.

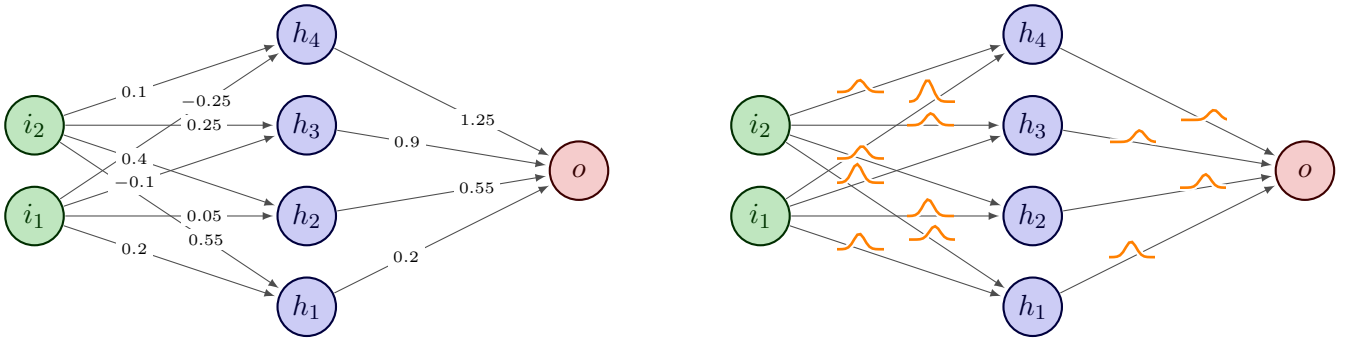


Figure 5: Difference between classical Neural Network (NN) and Bayesian Neural Network (BNN)

### Bayesian Framework

The BNN framework is based on the idea of combining neural network theory with Bayesian inference. The fundamental concept on which the method relies comes indeed from Bayesian statistic, and it expresses the fact that prior beliefs influence posterior beliefs. This idea can be formalized into the following key theorem:

**Theorem 2.2 (Bayes Theorem).** *Given two events  $A, B$ , such that  $\mathbb{P}(B) \neq 0$ , the conditional probability of  $A$  given  $B$ , denoted as  $\mathbb{P}(A|B)$ , can be computed as*

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}.$$

While working with BNNs, we will apply the same theorem in its continuous version to probability distributions (denoted with  $p$ ), and the distribution of interest will be the one of the network parameters  $\theta := \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^L$ , because from it we can infer the distribution of the output.

**Theorem 2.3 (Bayes Rule).** *For two continuous random variables  $X$  and  $Y$ , let us denote by  $f_X$  and  $f_Y$  their probability density. Then, the density of the conditioned random variable  $X|Y = y$  can be computed as follows:*

$$f_{X|Y=y}(x|y) = \frac{f_{Y|X=x}(y|x)f_Y(y)}{f_X(x)}, \quad \text{where } f_{Y|X}(y|x)f_X(x) = f_{X,Y}(x,y) = f_{X|Y}(x|y)f_Y(y)$$

The Bayes theorem is used to update prior knowledge on the network parameters' distribution exploiting the information coming from the observed data/the physics. At the same time, it turns out that we can establish a straightforward relation between probability distributions and loss functions: the good quality of the prediction can be indeed either represented by

1. a high value of the probability density;
2. a low value of the loss function.

Hence, probability densities can be inserted into the NNs presented in [subsection 2.1](#), by considering their opposite inside the loss function and recovering in this way the classical optimization problem framework.



### 2.3.1. Likelihood Distribution

We consider a scenario in which the dataset  $D$  consists of measurements of the external forces (denoted by  $D_f$ ) and of the PDE solutions (denoted by  $D_u$  when internal and by  $D_b$  when on the boundary).

As in the case of the direct problem it is common not to rely on measurements of the solution inside the domain but only on the boundary, it is convenient to distinguish these points in the definition of the dataset  $D$ , which is then splitted into:

$$D = D_u \cup D_b \cup D_f.$$

It is reasonable to consider that the available measurements are affected by some *noise*, due to inevitable defaults of the measurement tools.

Moreover, in principle we shall not include any co-implication between the measurements of the different quantities, hence we assume that  $\bar{\mathbf{u}}, \bar{\mathbf{f}}$  are independent gaussians, centered at the real hidden value  $\mathbf{u}, \mathbf{f}$ :

$$\bar{\mathbf{u}}^{(i)} = \mathbf{u}(\mathbf{x}^{(i)}) + \varepsilon_u^{(i)} \text{ with } \varepsilon_u^{(i)} \sim \mathcal{N}(0, \sigma_u^2) \quad i = 1, \dots, N_u$$

$$\bar{\mathbf{u}}^{(i)} = \mathbf{u}(\mathbf{x}^{(i)}) + \varepsilon_b^{(i)} \text{ with } \varepsilon_b^{(i)} \sim \mathcal{N}(0, \sigma_b^2) \quad i = 1, \dots, N_b$$

$$\bar{\mathbf{f}}^{(i)} = \mathbf{f}(\mathbf{x}^{(i)}) + \varepsilon_f^{(i)} \text{ with } \varepsilon_f^{(i)} \sim \mathcal{N}(0, \sigma_f^2) \quad i = 1, \dots, N_f$$

and that the variances  $\sigma_u^2, \sigma_b^2, \sigma_f^2$  of the noise terms are known.

In particular, following the choices made in [11], we set them to equal values  $\sigma_u, \sigma_b, \sigma_f$ , resulting in working with noises that are *independent and identically distributed* for each group. Since the measurements on  $\mathbf{u}$  and  $\mathbf{f}$  are assumed independent, the *likelihood* of data  $p(D|\boldsymbol{\theta})$  is computed as

$$p(D|\boldsymbol{\theta}) = p(D_u|\boldsymbol{\theta})p(D_b|\boldsymbol{\theta})p(D_f|\boldsymbol{\theta}). \quad (15)$$

By the former assumption on gaussianity, each density factor reads as:

$$p(D_u|\boldsymbol{\theta}) = \prod_{i=1}^{N_u} \frac{1}{\sqrt{2\pi\sigma_u^2}} \exp\left(-\frac{(\bar{\mathbf{u}}^{(i)} - \mathbf{u}^{(i)})^2}{2\sigma_u^2}\right), \quad (16)$$

$$p(D_b|\boldsymbol{\theta}) = \prod_{i=1}^{N_b} \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(-\frac{(\bar{\mathbf{u}}^{(i)} - \mathbf{u}^{(i)})^2}{2\sigma_b^2}\right), \quad (17)$$

$$p(D_f|\boldsymbol{\theta}) = \prod_{i=1}^{N_f} \frac{1}{\sqrt{2\pi\sigma_f^2}} \exp\left(-\frac{(\bar{\mathbf{f}}^{(i)} - \mathbf{f}^{(i)})^2}{2\sigma_f^2}\right). \quad (18)$$

### 2.3.2. Prior Distribution

The initial knowledge on the distribution of network parameters  $\boldsymbol{\theta}$  is summarized by  $p(\boldsymbol{\theta})$ , which is called *prior*. A classic choice for NN parameters is considering  $\boldsymbol{\theta}$  to be *a priori* distributed as a multivariate normal.

As stated in [11], a common choice in the Bayesian learning framework is to assume *independence* among each weight and bias of the neural network and *zero mean*. For what concerns the variance, the reference paper assumes it to be 1 for each network parameter, and in the proposed implementation the variance is set to

$$\sigma_\theta^2 = \frac{50}{N_l}$$

where  $N_l$  the number of neurons per layer.  $N_l = 50$  is the choice made in the reference paper, hence we set the same variance if the number of neurons per layer is the same, and ensure a bigger flexibility (represented by a bigger variance) in the case of a network with less parameters. Therefore,

$$p(\boldsymbol{\theta}) = \prod_{i=1}^{N_\theta} \frac{1}{\sqrt{2\pi\sigma_\theta^2}} \exp\left(-\frac{(\theta^{(i)} - 0)^2}{2\sigma_\theta^2}\right), \quad (19)$$

where  $N_\theta$  is the total number of network parameters (considering both weights and biases).



### 2.3.3. Posterior Distribution

The Bayes theorem enables to reconstruct, starting from *prior* and *likelihood*, the quantity  $p(\boldsymbol{\theta}|D)$ , that is called *posterior* distribution of  $\boldsymbol{\theta}$  and represents the update of the prior knowledge on the parameters' distribution after having obtained the information from data. The theorem's formulation reads then as:

$$p(\boldsymbol{\theta}|D) = \frac{p(D|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(D)}.$$

$p(D)$  represents the distribution of the dataset, but its calculation is typically unfeasible. Hence, we exploit the Bayes Theorem only to get the relation of proportionality:

$$p(\boldsymbol{\theta}|D) \propto p(D|\boldsymbol{\theta})p(\boldsymbol{\theta}) \quad (20)$$

The reconstruction of the posterior as presented above is then exploited to sample a set  $\{\boldsymbol{\theta}_i\}_{i=1}^M$  where

$$\boldsymbol{\theta}_i \sim p(\boldsymbol{\theta}|D) \quad i = 1, \dots, M$$

However, computing the posterior directly is impossible in general with Neural Networks, therefore methods involving Markov Chain Monte Carlo or Variational Inference (as the ones presented in 3) can be used to sample directly from the posterior distribution.

Having reconstructed the posterior distribution of parameters, they are then used to reconstruct the posterior predictive distribution of the neural network - namely, the distribution of the network output  $f_{\boldsymbol{\theta}}(x)$ . Indeed, by producing the neural networks output corresponding to the sampled parameters, the estimation of the mean and of the variance of the output  $f$  given a fixed input  $\mathbf{x}^*$  is performed as follows:

$$\mathbb{E}[f|\mathbf{x}^*, D] \approx \frac{1}{M} \sum_{i=1}^M f_{\boldsymbol{\theta}_i}(\mathbf{x}^*)$$

$$\text{Var}[f|\mathbf{x}^*, D] \approx \frac{1}{M} \sum_{i=1}^M (f_{\boldsymbol{\theta}_i}(\mathbf{x}^*) - \mathbb{E}[f|\mathbf{x}^*, D])^2$$

The two quantities obtain represent meaningful metrics for UQ, embedded into a probabilistic framework, and can be used in the production of credible confidence intervals for the neural network output.

## 2.4. Bayesian Physics Informed Neural Networks

The above discussion does not take into account any information coming from the physics, but relies simply on data. The Bayesian-Physics Informed Neural Network (B-PINNs) approach consists in adding to a BNN model the knowledge that the problem solution is required to satisfy given partial differential equations and boundary conditions. This method is implemented in the main papers of reference, that are [11], [7].

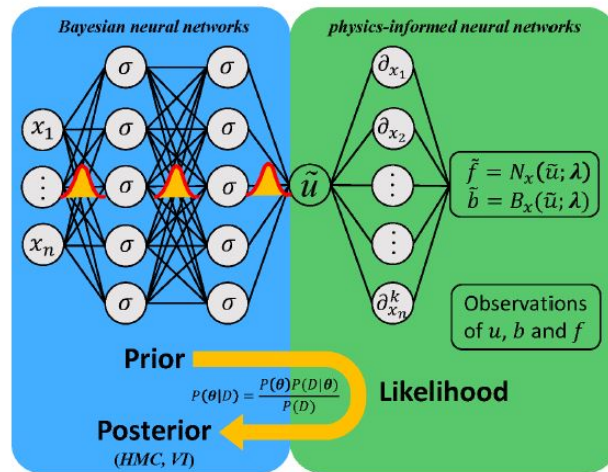


Figure 6: Structure of a B-PINN, extracted from the reference paper [11].

### 2.4.1. Physical Likelihood

They represent an integration of the PINNs into a Bayesian framework, as the constraint of the PDE appears as additional term of likelihood. In the formulation of the Bayes Theorem the information coming from the partial differential law is present, given the modification of the likelihood term:

$$p(\boldsymbol{\theta}|D, R) \propto p(D, R|\boldsymbol{\theta})p(\boldsymbol{\theta}). \quad (21)$$

The likelihood of data  $p(D, R|\boldsymbol{\theta})$  is computed as

$$p(D, R|\boldsymbol{\theta}) = p(D|\boldsymbol{\theta})p(R|\boldsymbol{\theta}) \quad (22)$$

In Equation 22,  $p(D|\boldsymbol{\theta})$  is the same quantity present in Equation 15 for the not physics-informed case, and  $p(R|\boldsymbol{\theta})$  plays the same role in an artificial way, but representing the information coming from the physical law rather than from measurements.

Considering the set of *collocation points*  $\{x_i\}_{i=1}^{N_{col}}$  already defined in subsection 2.2, the goal is to require that at these points the solution approximated by the neural network solves the partial differential equation. Therefore, one can proceed similarly as done for the *fitting points* while imposing the solution to be close to the measurement, and it is common in the B-PINNs framework to consider this part of the likelihood to be gaussian as well:

$$p(R|\boldsymbol{\theta}) = \prod_{i=1}^{N_{col}} \frac{1}{\sqrt{2\pi\sigma_r^2}} \exp\left(-\frac{(\mathcal{R}(\tilde{\mathbf{u}}^{(i)}) - 0)^2}{2\sigma_r^2}\right). \quad (23)$$

In Equation 23,  $\mathcal{R}$  denotes the residual as in subsection 2.2, while  $\sigma_r$  plays the role of  $\sigma_u$  or  $\sigma_f$  in the fitting case and represents an artificial uncertainty related to the physical knowledge. The better the model is in approximating the phenomenon, the smaller will  $\sigma_r$  be; note also that, by introducing differences among the values of uncertainty for data and for physical knowledge, one can give a different weight to the two components of the likelihood, and therefore ensure that the most reliable information (i.e., the one to whom less uncertainty is associated) weighs more and will be more determinant in the learning process.

### 2.4.2. Prior for Inverse Problems

The Bayesian framework can be applied to inverse problems as well, by considering that the model parameters  $\boldsymbol{\lambda}$  (following the same notation as in subsection 2.2) follow a certain probability distribution.

We can assume to have a prior knowledge on  $\boldsymbol{\lambda}$ , and it can reasonably assumed to be independent from  $\boldsymbol{\theta}$ , because there is no relation between model and network parameters.

Hence, the prior term containing information on both of them  $p(\boldsymbol{\theta}, \boldsymbol{\lambda})$  can be computed as

$$p(\boldsymbol{\theta}, \boldsymbol{\lambda}) = p(\boldsymbol{\theta})p(\boldsymbol{\lambda}).$$

With the above expression for the prior, the same discussion made in the case of the direct problem still holds, provided that we consider that the likelihood of data depends on fixed model parameters as well, and  $p(D|\boldsymbol{\theta})$  is replaced by  $p(D|\boldsymbol{\theta}, \boldsymbol{\lambda})$ . Consequently, the formulation of the Bayes theorem reads as

$$p(\boldsymbol{\theta}, \boldsymbol{\lambda}|D, R) = \frac{p(D, R|\boldsymbol{\theta}, \boldsymbol{\lambda})p(\boldsymbol{\theta})p(\boldsymbol{\lambda})}{p(D)p(R)}.$$

The latter can be rewritten with the all aforementioned simplifications as:

$$p(\boldsymbol{\theta}, \boldsymbol{\lambda}|D, R) \propto p(D|\boldsymbol{\theta}, \boldsymbol{\lambda})p(R|\boldsymbol{\theta}, \boldsymbol{\lambda})p(\boldsymbol{\theta})p(\boldsymbol{\lambda}) \quad (24)$$

and we obtain a proportionality relation ready-to-use. Finally, naming  $L = \log(p)$  for all the functions, we get:

$$Loss = L(D; \boldsymbol{\theta}, \boldsymbol{\lambda}) + L(R; \boldsymbol{\theta}, \boldsymbol{\lambda}) + L(\boldsymbol{\theta}) + L(\boldsymbol{\lambda}) = Loss_{data} + Loss_{pde} + Prior_{\boldsymbol{\theta}} + Prior_{\boldsymbol{\lambda}} \quad (25)$$

### 3. Algorithms

As stated in [subsection 2.4](#), the core concept of Bayesian Machine Learning is to sample from the the posterior distribution of weights  $p(\boldsymbol{\theta}|D)$  (or  $p(\boldsymbol{\theta}, \boldsymbol{\lambda}|D)$ , in the case of the inverse problem).

However, the exact reconstruction of the posterior is unfeasible, as witnessed by the necessity to approximate the Bayes theorem only with the proportionality relation in [Equation 20](#).

In this chapter, we propose the three algorithms implemented that represent different solutions to sample  $\boldsymbol{\theta}$  values from the a distribution approximating the real one reasonably well:

- **Hamiltonian Monte Carlo (HMC)**, a common Markov Chain Monte Carlo method which exploits a discrete simulation of Hamiltonian Dynamics and a correction with an acceptance-rejection step;
- **Variational Inference (VI)**, that aims at reconstructing a surrogate for the target distribution by looking for its best approximation within a parametric family that is simpler than the unknown target;
- **Stein Variational Gradient Descent (SVGD)**, that is based on a similar philosophy of VI and is the counterpart of gradient descent which iteratively transports a set of “particles” to match the target.

We will devote to each of them a specific section covering theoretical foundations and algorithm mechanisms.

#### Log-posterior

In these algorithms we operate with the logarithm of the posterior, hence we devote this section to the computation of this quantity that will be required later on.

With the rule of the logarithm of the product, the relation in [Equation 21](#) becomes:

$$L(\boldsymbol{\theta}) := \log(p(\boldsymbol{\theta}|D, R)) \propto \log(p(D, R|\boldsymbol{\theta})) + \log(p(\boldsymbol{\theta})) \quad (26)$$

By substituting [22](#) and [15](#) in [Equation 26](#), we get

$$L(\boldsymbol{\theta}) \propto \log(p(D_u|\boldsymbol{\theta})) + \log(p(D_b|\boldsymbol{\theta})) + \log(p(D_f|\boldsymbol{\theta})) + \log(p(R|\boldsymbol{\theta})) + \log(p(\boldsymbol{\theta})) \quad (27)$$

Thanks to the exponential nature of the terms, the above relation can be reduced to a sum of mean squared errors. Indeed, from [16–18](#) and [23](#) we get

$$\log(p(D_u|\boldsymbol{\theta})) \propto -\frac{1}{2\sigma_u^2} \sum_{i=1}^{N_u} (\tilde{\mathbf{u}}^{(i)} - \bar{\mathbf{u}}^{(i)})^2 + \frac{N_u}{2} \log\left(\frac{1}{\sigma_u^2}\right), \quad (28)$$

$$\log(p(D_b|\boldsymbol{\theta})) \propto -\frac{1}{2\sigma_b^2} \sum_{i=1}^{N_b} (\tilde{\mathbf{u}}^{(i)} - \bar{\mathbf{u}}^{(i)})^2 + \frac{N_b}{2} \log\left(\frac{1}{\sigma_b^2}\right), \quad (29)$$

$$\log(p(D_f|\boldsymbol{\theta})) \propto -\frac{1}{2\sigma_f^2} \sum_{i=1}^{N_f} (\tilde{\mathbf{f}}^{(i)} - \bar{\mathbf{f}}^{(i)})^2 + \frac{N_f}{2} \log\left(\frac{1}{\sigma_f^2}\right), \quad (30)$$

$$\log(p(R|\boldsymbol{\theta})) \propto -\frac{1}{2\sigma_r^2} \sum_{i=1}^{N_{col}} (\mathcal{R}(\tilde{\mathbf{u}}^{(i)}) - 0)^2 + \frac{N_{col}}{2} \log\left(\frac{1}{\sigma_r^2}\right). \quad (31)$$

In [\[3\]](#), [Equation 27](#) is seen as a linear combination of the terms with coefficients  $\{\alpha_i\}_{i=1}^5$ :

$$L(\boldsymbol{\theta}) \propto \alpha_1 \log(p(D_u|\boldsymbol{\theta})) + \alpha_2 \log(p(D_b|\boldsymbol{\theta})) + \alpha_3 \log(p(D_f|\boldsymbol{\theta})) + \alpha_4 \log(p(R|\boldsymbol{\theta})) + \alpha_5 \log(p(\boldsymbol{\theta}))$$

However, in this work we excluded this option for two main reasons:

1. Due to the removal of the denominator in the application of the Bayes’s Theorem, [Equation 20](#) cannot represent a probability distribution anymore, but only an approximation. However, the introduction of coefficients in [Equation 27](#) would still imply a further dissociation from the Bayes’s Theorem.
2. The goal of give a different weight to each term can still be reached by remaining coherent with the probabilistic interpretation, thanks to the user’s choice of the uncertainties.

#### 3.1. Hamiltonian Monte Carlo

The first method we propose is Hamiltonian Monte Carlo, to whom we will refer as HMC; by integrating Monte Carlo techniques and Hamiltonian Dynamics, it produces a sequence of random samples which converge to being distributed according to our target probability distribution.

### 3.1.1. Theoretical Foundations

This algorithm is a classic in the family of Markov Chain Monte Carlo (MCMC) methods, which enable sampling from a probability distribution by constructing a Markov chain that has the desired distribution as its equilibrium distribution. In HMC, the purpose consists in obtaining a sequence of random samples which converge to being distributed according to a target probability distribution for which direct sampling is difficult; to do so, we first simulate an Hamiltonian dynamics using numerical integration, and then correct with an acceptance-rejection step, to reduce the discretization error.

## Hamiltonian Dynamics

Let us consider the logarithm of the posterior in 27, and define a *potential energy* as its opposite:

$$U(\boldsymbol{\theta}) := -L(\boldsymbol{\theta})$$

Note that this relationship in terms of sign between potential energy and (log) posterior is meaningful: the natural goal in this setting is indeed maximizing the probability and, by following the usual physical convention, minimizing the potential energy.

Then, we use it to define the Hamiltonian function, that will describe the dynamics of a continuum system; note that a term which mimics kinetic energy is added as well:

$$H(\boldsymbol{\theta}, \mathbf{r}) := U(\boldsymbol{\theta}) + \frac{1}{2} \mathbf{r}^T \mathbf{M}^{-1} \mathbf{r}, \quad (32)$$

in the above formula,  $\mathbf{r}$  is an auxiliary momentum variable and  $\mathbf{M}$  plays the role of a mass matrix; it is common to set  $\mathbf{M} = \mathbf{I}$ , or  $\mathbf{M} = \alpha \mathbf{I}$ , for some  $\alpha > 0$ .

The Hamiltonian system associated to Equation 32 reads as

$$\begin{cases} \frac{d\boldsymbol{\theta}}{dt} = \frac{dH}{d\mathbf{r}} = \mathbf{M}^{-1} \mathbf{r} \\ \frac{d\mathbf{r}}{dt} = -\frac{dH}{d\boldsymbol{\theta}} = -\nabla U(\boldsymbol{\theta}). \end{cases} \quad (33)$$

Now, we establish a link between Hamiltonian dynamics and probabilities: basically, solving the system in 33 corresponds to generate samples of  $(\boldsymbol{\theta}, \mathbf{r})$  from the joint distribution

$$\pi(\boldsymbol{\theta}, \mathbf{r}) \sim \exp(-H(\boldsymbol{\theta}, \mathbf{r})) \quad (34)$$

and the sampling from the above is used to mimic the sampling from the posterior distribution  $p(\boldsymbol{\theta}|D, R)$ , which - up to a constant - coincides with  $\exp(-U(\boldsymbol{\theta}))$ .

As we are not interested in the samples of  $\mathbf{r}$ , we then discard them and consider only the samples of  $\boldsymbol{\theta}$ , which follow the marginal distribution  $p(\boldsymbol{\theta}|D, R)$ .

## Leap Frog

For what concerns the solution of Equation 33, we proceed numerically with the *leap-frog* method, consisting in introducing  $L$  additional steps between subsequent updates of  $\boldsymbol{\theta}$  of the form:

$$\begin{aligned} & \text{for } k = 1, \dots, L \\ & \begin{cases} \mathbf{r}^{(k+\frac{1}{2})} = \mathbf{r}^{(k)} - \frac{\Delta t}{2} \nabla(U(\boldsymbol{\theta}^{(k)})) \\ \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} + \Delta t \mathbf{M}^{-1} \mathbf{r}^{(k+\frac{1}{2})} \\ \mathbf{r}^{(k+1)} = \mathbf{r}^{(k+\frac{1}{2})} - \frac{\Delta t}{2} \nabla(U(\boldsymbol{\theta}^{(k+1)})). \end{cases} \end{aligned} \quad (35)$$

In Equation 35,  $\Delta t$  denotes the step size, and it should be related to  $L$ , which represents the number of steps to perform for each update, as we will illustrate in subsection 3.1.3.

## Acceptance-Rejection

In order to reduce the error introduced with the discretization, the new values are not always accepted, and this choice is controlled in the following acceptance-rejection step.

Considering two subsequent values of  $\boldsymbol{\theta}$ , that will be denoted by  $\boldsymbol{\theta}^{(0)}$  and  $\boldsymbol{\theta}^{(1)}$  for simplicity, we compute  $\alpha$ :

$$\alpha := \min \left( 1, \frac{\exp(-H(\boldsymbol{\theta}^{(1)}, \mathbf{r}^{(1)}))}{\exp(-H(\boldsymbol{\theta}^{(0)}, \mathbf{r}^{(0)}))} \right) = \min(1, \exp(-(H(\boldsymbol{\theta}^{(1)}, \mathbf{r}^{(1)}) - H(\boldsymbol{\theta}^{(0)}, \mathbf{r}^{(0)}))))).$$

This value is the probability of acceptance of  $\boldsymbol{\theta}^{(1)}$ : we accept the sampled value  $\boldsymbol{\theta}^{(1)}$  if  $\alpha \geq p$  with  $p \sim \mathcal{U}(0, 1)$ . Defining  $h := H(\boldsymbol{\theta}_1, \mathbf{r}_1) - H(\boldsymbol{\theta}_0, \mathbf{r}_0)$ , we can rewrite the expression of the acceptance rate as:

$$\alpha = \min(1, \exp(-h)) \implies h < 0 \implies \alpha = 1$$

When  $h < 0$ , it means that the Hamiltonian has decreased, hence we would like to accept always the new parameters. By contrast, when  $h > 0$  the Hamiltonian get worse and we would like to accept with a probability decreasing with  $h$ . The overall process outputs a sequence of network parameters  $\{\boldsymbol{\theta}^{(i)}\}_{i=1}^N$ ; not all of them are taken into account, but:

- we discard some initial samples, introducing a *burn-in*, to gain in the quality of the final prediction;
- we introduce a *stride*, and consider in the final list of  $\boldsymbol{\theta}$ s one parameter set each  $S$ . In this way, we reduced the correlation that could arise between subsequent samples, and improving the assumption that the final sampling of the parameters is independent and identically distributed.

### 3.1.2. Algorithm

We now report the complete algorithm that has been implemented:

---

#### Algorithm 4: Hamiltonian Monte Carlo

---

**Initialization:**

$\boldsymbol{\theta}^{(0)}$  initial value of  $\boldsymbol{\theta}$   
 $N$  total number of iterations  
 $B$  burn-in (number of initial samples to exclude)  
 $L$  number of leap-frog steps  
 $S$  stride  
 $\Delta t$  step size  
 $\alpha$  for defining the mass matrix  $\mathbf{M} = \alpha \mathbf{I}$

**for**  $k = 1, \dots, N$  **do**

**Leap-frog step:**

sample  $\mathbf{r}^{(k-1)} \sim \mathcal{N}(0, \mathbf{M})$   
 set  $(\boldsymbol{\theta}_0, \mathbf{r}_0) = (\boldsymbol{\theta}^{(k-1)}, \mathbf{r}^{(k-1)})$   
**for**  $i = 0, \dots, (L-1)$  **do**  
      $\mathbf{r}_i = \mathbf{r}_i - \frac{\Delta t}{2} \nabla U(\boldsymbol{\theta}_i)$   
      $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \Delta t \mathbf{M}^{-1} \mathbf{r}_i$   
      $\mathbf{r}_{i+1} = \mathbf{r}_i - \frac{\Delta t}{2} \nabla U(\boldsymbol{\theta}_{i+1})$

**Acceptance-Rejection step:**

sample  $p \sim \mathcal{U}(0, 1)$   
 $\alpha = \min(1, \exp(-(H(\boldsymbol{\theta}_L, \mathbf{r}_L) - H(\boldsymbol{\theta}_0, \mathbf{r}_0))))$   
**if**  $p \geq \alpha$  **then**  
      $\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}_L$   
**else**  
      $\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)}$

From the set  $\{\boldsymbol{\theta}^{(i)}\}_{i=1}^N$ , extract a subset  $\{\boldsymbol{\theta}_i^*\}_{i=1}^{(N-B)/S}$  using

Burn-in: discard the first  $B$  samples

Stride: take one sample each  $S$ , starting from the final one and going backwards

---

Then, the set of weights is used to compute samples of the solution  $\mathbf{u}$  and of the parametric field  $\mathbf{f}$  by producing neural network outputs setting each element of  $\{\boldsymbol{\theta}_i^*\}_i$  as network parameters, and obtaining the sets  $\{\tilde{\mathbf{u}}(\boldsymbol{\theta}_i^*)\}_i$  and  $\{\tilde{\mathbf{f}}(\boldsymbol{\theta}_i^*)\}_i$ . Thanks to statistics of these samples, we are able to quantify uncertainty and to produce a confidence interval for the network output, after having computed mean and standard deviations of the samples.

### 3.1.3. Parameters' choice

The choice of the number of epochs  $N$  or of the burn-in  $B$  has been performed with fine-tuning; we set  $N$  to a value able to reconstruct with a fairly low error the functions in a suitable computational time (no simulation with HMC training exceeded 20 min) and to keep the acceptance rate above 80%: such value is a trade-off between computational efficiency and guarantee of being close to the optimal solution.

For what concerns  $B$ , we always kept at least the 50% of the samples. Note that in our implementation the choice of this value has consequences on two different aspects: on one hand, indeed, we use the burn-in to discard the initial samples which usually are not meaningful and affected by the model initialization. To perform the same task without discarding samples we implemented also another possibility, that is letting HMC start after some Adam iterations: in this way the network parameters at the beginning of the bayesian algorithm are closer to the optimal ones and not completely random.

On the other hand, we let  $B$  determine the entrance of the information coming from physics into the log posterior; indeed, we noticed that performing differentiation on the network output when it was far from its true value was harmful for the learning process. The contribution of the physical loss is indeed in general heavier than the one of the fitting loss, given the computational burden of automatic differentiation. Therefore, the strategy that turned out to be optimal is to first get closer to the output only with the fitting points, then let the PDE come into play and improve the reconstruction – or reconstruct the part of the output which was lacking of fitting points.

At the current status, there is no strategy in literature for fixing the optimal values of  $L$  and  $\Delta t$ . A straightforward idea is to establish an inverse proportionality relation between them, by fixing their product to be equal to a constant  $K$ :

$$K = L\Delta t.$$

In this way indeed, we can fix the total “distance” travelled, but covering it in more smaller steps is clearly computationally more demanding. Hence, once  $K$  is chosen, the general rule to be followed is to try to reduce  $L$  the most possible in favour of bigger  $\Delta t$  (that does not affect computational time), but retaining a fairly high acceptance rate. The risk of big  $\Delta t$  is indeed a drop in performance, as we remarked that the update of the network parameters needed to be gradual to prevent turning to a regime in which the new proposal is always rejected (due to the explosion of the gradients in Equation 35).

## 3.2. Variational Inference

We will illustrate in this section another two methods which fall into a different framework: the one of Variational Bayesian Methods, and we first present the Variational Inference (VI) method.

In the purpose of approximating a posterior probability, variational inference-like methods are an alternative to Monte Carlo sampling methods for taking a fully Bayesian approach to statistical inference over complex distributions that are difficult to evaluate directly or sample.

In particular, Monte Carlo techniques can be slow and have difficulties in reaching the convergence. In variational methods we changed perspective, because we do not provide an indirect approximation of the posterior through a set samples, but we obtain a locally-optimal, exact analytical solution for an approximation of the posterior from which samples can later be generated.

### 3.2.1. Theoretical Foundations

The family of Variational Inference methods is based on the strategy to turn the problem (in this case, the sampling of  $\theta$ s from a distribution) into a deterministic optimization that approximates the target distribution  $P(\theta|D)$  with a simpler distribution  $Q(\theta|\zeta)$ .

### Distance of Probabilities

To perform this step, we need to establish how accurate is the approximation of a probability measure, hence a tool which captures similarities between probability distributions is needed. We use the Kullback–Leibler (KL) divergence between the target distribution and the one we are reconstructing: the KL divergence is a *statistical distance*, and lets us capture how much one probability distribution is similar to another.

**Definition 3.1** (Kullback–Leibler divergence). *Given a measure space  $(X, \mu)$  and two probability distributions  $P$  and  $Q$  with densities, respectively,  $p(x)d\mu$  and  $q(x)d\mu$ , the KL divergence between  $P$  and  $Q$  is defined as*

$$\mathcal{D}_{KL}(P||Q) = \int_X p(x) \log \left( \frac{p(x)}{q(x)} \right) d\mu$$

Then, algorithms of this family try to minimize the KL divergence between the target and the approximated distribution. In our context of B-PINNs we are interested in the value of  $\zeta$  that minimizes  $\mathcal{D}_{KL}(Q(\boldsymbol{\theta}|\zeta)||P(\boldsymbol{\theta}|D))$

$$\begin{aligned}\zeta^* &= \operatorname{argmin}[\mathcal{D}_{KL}(Q(\boldsymbol{\theta}|\zeta)||P(\boldsymbol{\theta}|D))] = \operatorname{argmin} \int Q(\boldsymbol{\theta}|\zeta) \log \frac{Q(\boldsymbol{\theta}|\zeta)P(D)}{P(\boldsymbol{\theta})P(D|\boldsymbol{\theta})} d\boldsymbol{\theta} = \\ &= \operatorname{argmin} \int Q(\boldsymbol{\theta}|\zeta) \left[ \log \frac{Q(\boldsymbol{\theta}|\zeta)}{P(\boldsymbol{\theta})} - \log P(D|\boldsymbol{\theta}) + \log P(D) \right] d\boldsymbol{\theta} = \\ &= \operatorname{argmin}[\mathcal{D}_{KL}(Q(\boldsymbol{\theta}|\zeta)||P(\boldsymbol{\theta})) - \mathbb{E}_{Q(\boldsymbol{\theta})}[\log P(D|\boldsymbol{\theta})] + \text{constant}]\end{aligned}$$

## Training Phase

The minimization problem can then be summarized as finding  $\zeta^* = \operatorname{argmin} \mathcal{F}(D, \zeta)$ , where:

$$\mathcal{F}(D, \zeta) = \mathcal{D}_{KL}(Q(\boldsymbol{\theta}|\zeta)||P(\boldsymbol{\theta})) - \mathbb{E}_{Q(\boldsymbol{\theta})}[\log P(D|\boldsymbol{\theta})] \quad (36)$$

In 36, the first term is prior-dependent and is called *complexity cost*, while the second is the classical *likelihood cost* depending on data. Our computational task will be to approximate  $\mathcal{F}(D, \zeta)$  with its Monte-Carlo expectation and use it as training loss function optimized with Adam.

As auxiliary distribution  $Q$ , we choose a family that is described by a set of parameters that we will denote by  $\zeta \in \mathbb{R}^{2d_\theta}$  following the common practice implemented also in [11], and taking a factorizable Gaussian distribution

$$Q(\boldsymbol{\theta}, \zeta) = \prod_{i=1}^{d_\theta} q(\theta_i, \zeta_i^\mu, \zeta_i^\rho) \quad (37)$$

which also allows us to compute analytically some of the quantities needed in the algorithm.

In Equation 37, the vector of parameters  $\zeta$  has as components the pairs  $(\zeta_i^\mu, \zeta_i^\rho)$  for  $i = 1, \dots, d_\theta$ . Each pair characterizes the distribution of a network parameter, that is taken to be an uni-variate Gaussian independent one from each other and having mean  $\zeta_i^\mu$  and standard deviation  $\log(1 + \exp(\zeta_i^\rho))$ . Note that we choose to work with the quantity  $\zeta^\rho$  instead of the standard deviation in order to get rid of the constraint of having a positive quantity, because gradient descent-like algorithms work optimally with parameters living in the whole set  $\mathbb{R}$ .

## Prediction Phase

The above optimization process enables us to recover the best approximating distribution in the parametric family, that we will denote as  $Q(\zeta^*)$ , from which we can sample as it was the real posterior distribution.

Differently with respect to algorithms like HMC, indeed, this time the process returns a distribution instead of a set of samples from a distribution, and an evident consequence of this is that we are able to generate directly an arbitrary number of samples of parameters  $\{\boldsymbol{\theta}_i^*\}_i$  from the output distribution.

Again by setting each element of the set as network parameter, we can then obtain sets of samples for solution  $\{\hat{\mathbf{u}}(\boldsymbol{\theta}_i^*)\}_i$  and parametric field  $\{\hat{\mathbf{f}}(\boldsymbol{\theta}_i^*)\}_i$ , to reconstruct their distribution.

### 3.2.2. Algorithm

We now report the complete algorithm that has been implemented:

---

#### Algorithm 5: Variational Inference

---

##### Initialization:

$N$  total number of epochs  
 $N_z$  number of samples of  $\boldsymbol{\theta}$  to optimize  $\zeta$   
 $M$  number of desired samples  
 $\zeta_i$  initial value of the auxiliary parameters

**for**  $k = 1, \dots, N$  **do**

sample  $\{\mathbf{z}^{(j)}\}_{j=1}^{N_z}$  independently from  $\mathcal{N}(\mathbf{0}, \mathbf{I}_{d_\theta})$   
 set  $\boldsymbol{\theta}^{(j)} = \zeta^\mu + \log(1 + \exp(\zeta^\rho)) \odot \mathbf{z}^{(j)} \quad j = 1, \dots, N_z$   
 $L(\zeta) = \frac{1}{N_z} \sum_{j=1}^{N_z} [\log(Q(\boldsymbol{\theta}^{(j)}; \zeta)) - \log P(\boldsymbol{\theta}^{(j)}) - \log P(D|\boldsymbol{\theta}^{(j)})]$   
 update  $\zeta$  with gradient  $\nabla_\zeta L(\zeta)$  using the Adam optimizer

sample  $\{\mathbf{z}^{(j)}\}_{j=1}^M$  independently from  $\mathcal{N}(\mathbf{0}, \mathbf{I}_{d_\theta})$

set  $\boldsymbol{\theta}^{(j)} = \zeta^\mu + \log(1 + \exp(\zeta^\rho)) \odot \mathbf{z}^{(j)} \quad j = 1, \dots, M.$

---



### 3.2.3. Parameters' choice

The VI method shares some characteristics with algorithms outside the Bayesian framework, as [GD](#) or [Adam](#) in the part of training of the distribution parameters; then, its final step guarantees to enter the probabilistic framework, because the output is sampled for the learned distribution.

This is reflected in the choice of the method parameters as well: the number of epochs  $N$  should be chosen having in mind the same principle as for Adam and GD, because there is an improvement of the learned parameters with the number of epochs and for the prediction only the final values of the learned parameters are used. In this stage, also the learning rate  $\alpha$  should be chosen in a similar way to deterministic algorithms.

On the other hand, the number of samples  $M$  does not affect training and therefore computational time; with respect to methods such as SVGD, we are free to increase it to produce more significant confidence intervals, without a computational overload.

## 3.3. Stein Variational Gradient Descent

Another method that approximates the posterior minimizing the KL divergence with a different strategy is Stein Variational Gradient Descent (SVGD). This method can be treated as a natural counterpart of gradient descent for full Bayesian inference.

### 3.3.1. Theoretical Foundations

The theoretical foundations of the method are shared with VI, as the underlying major idea is still learning the posterior by minimizing the KL divergence.

#### Analytical Derivation

The analytical derivation of the algorithm is in this case longer than for the previous cases, hence we will simply highlight the key passages; further details can be found in the reference [\[8\]](#).

1. the auxiliary family  $Q$  this time consists of distributions obtained by smooth transforms  $T$  of random variable  $x$  drawn from a tractable reference distribution  $q_0$ :  $z = T(x)$  with  $x \sim q_0$ ;
2. we look for an iterative algorithm to look for the best  $T$ , which is dealt with through its perturbative formulation  $T(x) = x + \varepsilon\phi(x)$ , with a scheme of the form  $T_{n+1}(x) = T_n(x) + \varepsilon\phi_n(x)$ ;
3. we insert this formulation inside the KL divergence, and since we want to minimize this quantity, we search for the steepest descent direction  $\phi$ , that coincides with maximizing  $-\nabla_\varepsilon KL(Q_T||P)|_{\varepsilon=0}$ ;
4. for a vectorial function  $\phi(x)$ , we can define the Stein operator  $A_p\phi(x) = \phi(x)\nabla_x \log p(x)^T + \nabla_x \phi(x)$ . Expanding the computations, we obtain:  $-\nabla_\varepsilon KL(Q_T||P)|_{\varepsilon=0} = \mathbb{E}_{x \sim Q_T}[\text{tr}(A_p(\phi(x)))]$ ;
5. the optimization problem now reads as maximizing the so-called kernelized stein discrepancy  $\mathbb{E}_{x \sim Q_T}[\text{tr}(A_p(\phi(x)))]$  in the space *RKHS* (Reproducing Kernel Hilbert Space with kernel  $k$ );
6. the solution to this problem is known, and it is  $\phi_{Q_T, P}^*(x) = \mathbb{E}_{x \sim Q_T}[k(x, \cdot)\nabla_x \log(p(x)) + \nabla_x k(x, \cdot)] = \mathbb{E}_{x \sim Q_T}[A_p k(x)]$ ;
7. finally, we want to estimate the expected value with Monte Carlo, and we can do this with  $N$  particles sampled from  $q_0$ , iteratively updated with the formula indicated in step 2.

#### Numerical Simulation

The intuition behind the application of this method is basically to work with a finite number of neural networks (to whom we will refer as *particles*) and transport, iteration after iteration, all the particles' parameters towards the target distribution, following the direction given by the minimization of the KL divergence.

Those particles represent samples for the computation of Monte Carlo expectation provided by [Equation 38](#) of the derivation. At first iteration they are randomly drawn from the distribution  $q_0$  and each time they are used to compute the Stein operator. Lastly, given the increments, we update directly the samples as they were drawn from  $q_1$  and so on.

$$\mathbb{E}_{\theta \sim Q}[k(\theta, \cdot)\nabla_\theta \log p(\theta) + \nabla_\theta k(\theta, \cdot)], \quad (38)$$

During the analytical derivation we never mention the meaning of  $k$ ; it is a strictly positive definite kernel and one suitable example is the RBF kernel that we exploit in our implementation:

$$k(x, x') = \exp\left(-\frac{1}{h}\|x - x'\|^2\right) \quad (39)$$

with  $h > 0$  positive bandwidth.



Note that to the two addends in Equation 38 we can associate a specific meaning:

1.  $k(\boldsymbol{\theta}, \cdot) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta})$  drives the particles towards the high probability areas of  $p$  (or of  $\log p$ , that is  $L(\boldsymbol{\theta})$  of Equation 26) by following a smoothed gradient direction, which is the weighted sum of the gradients of all the points weighted by the kernel function;
2.  $\nabla_{\boldsymbol{\theta}} k(\boldsymbol{\theta}, \cdot)$  acts as a *repulsive force* that prevents all the points to collapse together into the same point, that is the maximum-a-posteriori of  $L(\boldsymbol{\theta})$ . Indeed, considering the RBF kernel, we have that:

$$\nabla_x k(x, x') = \frac{2}{h}(x - x')k(x, x') \quad (40)$$

hence  $x$  is driven away from neighbours that are such that  $k(x, x')$  is big. If we let the bandwidth  $h \rightarrow 0$ , the repulsive term vanishes: this means that the parameters' update reduces to a set of independent chains of typical gradient ascent for maximizing  $\log p$  and all the particles would collapse into the local modes.

The update of the parameters can then be done as in gradient descent introducing a term that plays the role of the *learning rate* as well. Moreover, we could also implement an approach similar to *stochastic gradient descent* (algorithm 2), that can be useful when dealing with large datasets.

### 3.3.2. Algorithm

We now report the complete algorithm that has been implemented:

---

#### Algorithm 6: Stein Variational Gradient Descent

---

**Initialization:**

$N$  number of neural networks  
 $\{\boldsymbol{\theta}_i\}_{i=1}^N$  initial value of parameters for each network  
 $E$  total number of epochs  
 $\varepsilon$  step size

**for**  $k = 1, \dots, E$  **do**

$\phi(\boldsymbol{\theta}_i) = \frac{1}{N} \sum_{j=1}^N k(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) \nabla_{\boldsymbol{\theta}_j} L(\boldsymbol{\theta}_j) + \nabla_{\boldsymbol{\theta}_j} k(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) \quad \forall i = 1, \dots, N$   
 $\boldsymbol{\theta}_i = \boldsymbol{\theta}_i + \varepsilon \phi(\boldsymbol{\theta}_i) \quad \forall i = 1, \dots, N$

collect the parameters of all networks after  $E$  epochs  $\{\boldsymbol{\theta}^{(i)}\}_{i=1}^N$ .

---

### 3.3.3. Parameters' choice

In terms of parameters selection, the decision effort required by SVGD is not that high, especially compared to HMC. In fact, the parameters to be set are only  $N$ ,  $E$  and  $\varepsilon$ , but the real obstacle for tuning consists in the fact that SVGD is the most computational demanding algorithm, issue that could be partially solved with a parallelization of the computations made by the particles.

The number of epochs  $E$  affects the quality of the prediction, while the number of particles  $N$  can determine the quality of the UQ; in [8] it is suggested that the convergence to the target distribution is reached for  $E, N \rightarrow \infty$ , hence (clearly) the bigger the  $E$  and the  $N$ , the better the approximation of the distribution. On the other hand, large  $E$  and  $N$  require higher computational time, and moreover enough RAM memory to store the parameters of all the particles.

The learning rate  $\varepsilon$  played a fundamental role during parameters' tuning, as learning rates parameters for other common algorithms, because its choice enabled to prevent the network's trainable parameters to explode to **nan**, due to a growth out of control.

## 4. Code Overview

In this chapter, we first present and motivate the coding choices in terms of environment and libraries. Then, we devote a section to Object-Oriented Programming in Python, focusing on the features implemented in the project. The final section presents the structure of the repository and the content of the configuration, data and outputs files.

### 4.1. Working Environment

The code implemented in the project - starting from scratch - is contained in the GitLab repository PACS-BPINNS, that can be found [here](#), accompanied by a [README](#).

In [Appendix A](#), we present the procedure to follow, for the different operating systems, to set-up our working environment and download all the required packages, that are listed in the file `requirements.txt`<sup>6</sup> and presented more in detail in [subsection 4.2](#).

#### 4.1.1. Interpreter

The code is written in Python, and in order to run the scripts contained in the executable repository, the user needs `python` version 3.10.\* (download from [here](#)).

The constraint on the version is due to the presence of the `match-case` statement, introduced in Python 3.10, that is particularly suitable to mimic the behaviour of `switch` in Java or C++ and guarantees more readability than chains of conditional statements when we need to order a different instruction for each case.

#### 4.1.2. Virtual Environment

Our choice of work setting was to code inside a Python Virtual Environment, because this programming language relies a lot on external dependencies and version updates are frequent. Indeed, with the choice of creating and managing separate environments for different projects we enable a safe management of versions: each environment can use different versions of package dependencies and/or Python, and this ensures that projects are not able to cause dependency conflicts for one another.

Moreover, this choice can also simplify the reproduction of the results from other programmers, because they can install the required packages without creating conflicts with the ones already present on their machines.

This tools are provided by the module `virtualenv` (version 20.14.\*, downloadable from [here](#)).

Another possibility could have been to set up a Conda environment, but being interested in an environment for this single project, we preferred to rely on this Python's native feature (available after Python 3.3), without requiring any third-part distribution.

## 4.2. Libraries

In the code, several popular Python libraries are exploited; we now list them all, deepening more into the ones that are relevant for the implementation.

### 4.2.1. Built-in Packages

In this project we also relied on few common Python modules for simpler and side tasks.

All these modules are among the *Python Runtime Services* or belong to the *Python Standard Library*; therefore, they do not appear in `requirements.txt` and they are directly distributed with Python.

`os`, module providing a portable way of using operating system dependent functionality. We mainly exploit the submodule `os.path`, that is used to manipulate file paths.

`shutil`, module offering a number of high-level operations on files and collections of files.

`json`, module used to work with JSON data, that consist in text, written with JavaScript object notation.

`argparse`, module that enables to write user-friendly command-line interfaces.

`warnings`, module for printing user-defined warnings.

`time`, `datetime`, modules supplying time-related functions and classes for manipulating dates and times.

---

<sup>6</sup>This file presents more libraries than the ones installed and described in [subsubsection 4.2.2](#), because it already contains all the related dependencies of the presented ones.

For debugging purposes we relied on `pdb`, that is a module defining an interactive source code debugger for Python programs. By inserting in the desired point of the code `import pdb; pdb.set_trace()`, the user can break into the debugger, and then continue running until the next breakpoint by typing `continue`.

### 4.2.2. External Packages

The massive presence of tailor-made libraries provided both by important open source projects and big tech companies such as Google and Meta is one of the strenghts of Python's language; in this section, we present the main external libraries for Data Science and ML tasks exploited in the code.

#### Numpy

`numpy` is one of the most popular Python packages for scientific computing; it provides a multidimensional array object (the `numpy.array`), various derived objects such as masked arrays and matrices, and an assortment of routines for fast operations (especially of linear algebra) on arrays.

We underline some features of the `numpy.array` type compared to Python native `lists`:

- differently than Python's `list` type, it has a strict requirement on the homogeneity of the objects stored and supports element-wise operations;
- with respect to a `list`, a `numpy.array` is much faster and its elements are stored contiguously in memory;
- a `numpy.array` supports item assignment; this feature will not be shared by another structure used in the project: Tensorflow's `Tensor` (subsubsection 4.2.3).

#### Matplotlib

`matplotlib.pyplot` is a user-friendly interface to `matplotlib`, which is the comprehensive library for visualization in Python. It consists of a collection of functions that make `matplotlib` work similarly to `MATLAB`. Each `pyplot` function makes some action on a figure: for example, it creates a figure, plots some lines in a plotting area, decorates the plot with labels, titles and legends...

#### Scipy

`scipy.stats` is the module devoted to Statistics within SciPy (another common library for scientific computing); it offers a large number of probability distributions, summary and frequency statistics, correlation functions and statistical tests. We exploit in particular the submodule `qmc`, providing Quasi-Monte Carlo generators and associated helper functions.

#### Tqdm

`tqdm` is a module for displaying progress bars during iterative procedures. The computational overhead added by this feature is low (about 60ns per iteration), according to the tests (see the [documentation](#)), and lower than other similar tools such as *ProgressBar*. Moreover, `tqdm` uses smart algorithms to predict the remaining time and to skip unnecessary iteration displays, which allows for a negligible overhead in most cases.

### 4.2.3. TensorFlow

`tensorflow (tf)` is the most relevant requirement for the NN implementation and it is a free and open-source software library for machine learning developed by the Google Brain Team in 2015.

Multidimensional arrays of elements are handled in TensorFlow within `tf.Tensor` objects.

This data structure is characterized by the following properties:

- a single data type (`float32`, `int32`, or `string`, for example);
- a shape, representing the length of each `Tensor` axes.

We also remark two fundamental differences with respect to the `numpy.array` type:

- `Tensors` have accelerator support, like GPU and TPU;
- `Tensors` do not support item assignment.

In the code, we work with the TensorFlow version 2.9.1, and, more specifically, we download the `tensorflow-cpu` library to avoid unnecessary warnings due to the absence of GPU on the machines that we used to run the code, but if a CUDA-compatible GPU is available, by choosing `tensorflow-gpu` or `tensorflow` (that will then detect the presence of GPU), one is able to run the code on GPU (see [here](#) for more details). However, for this project the computational time required was always feasible for laptops without GPU, given the limited size of the neural network employed.

## Automatic Differentiation

Automatic differentiation was already introduced in [subsection 2.2](#). TensorFlow can automatically compute the gradients for the parameters in a model, which are then used in algorithms such as [backpropagation](#). To do so, the framework must keep track of the order of operations done to the input **Tensors** in a model, and then compute the gradients with respect to the appropriate parameters.

To keep track of the operations, we perform automatic differentiation tasks inside a `tf.GradientTape`; this is a *context manager* that enables to record the operations inside it and to calculate gradients with respect to given variables. In this context, it is required that the variables are *watched* by the `tape`, and this happens by default only with trainable variables. Therefore, for an arbitrary tensor as the network input, we need to call the `tape.watch()` function.

## Other major features

The TensorFlow framework contains other useful features for tasks involved in the training of NNs that are built for the **Tensor** data structure and automatic differentiation purposes, such as:

- Eager execution: a mode in which operations are evaluated immediately as opposed to being added to a computational graph which is executed later. With this type of execution, the code can be examined step-by-step through a debugger, since data is augmented at each line of code rather than later in a computational graph.
- API access to the most common losses, metrics, optimizers and math operations for **Tensor** objects, implemented to be compatible with automatic differentiation and gradient tapes.
- Keras submodule: `tensorflow.keras` is a high-level API of TensorFlow providing many tools for neural network implementation that are designed to be user-friendly. Keras indeed offers consistent and simple APIs which minimize the number of user actions required for common use cases. We rely on it for an easy access to built-in activation and loss functions, parameter initializers, layers and models.

## 4.3. Object Oriented Features

Python belongs to the family of Object Oriented Programming (OOP) Languages, representing a paradigm based on the concept of “objects” which can contain data and code, rather than on “procedures” which contain a series of computational steps to be carried out.

Python provides a variety of typical OOP features (such as encapsulation, inheritance, polymorphism...), but clearly, as other OOP languages, it can be used even for programming in a procedural style.

In this section, we present some modules and decorators<sup>7</sup> exploited to give intuitive properties to objects and data structures that appear in the B-PINNs framework. Later in [section 5](#) we will specify the classes in which we have introduced them and the reasons behind the choice, while here we propose a general overview on the main functionalities exploited.

### 4.3.1. Inheritance

In the code, you can find several examples of the OOP concept of inheritance, having sub-classes which are built on parent classes: we have examples of both *single inheritance*, where subclasses inherit the features of one superclass, and *multiple inheritance*, where one class has more than one superclass and inherits from all.

When dealing with inheritance, we often override methods and we need to choose which method to refer to between the child’s and the parent’s one. This holds for all methods, also for special ones like constructors.

Indeed, given one parent class, when building a subclass upon it we may want to define a custom constructor for the child class, that will be called when instantiating the class; otherwise, the constructor of the parent class will be called. In other cases, like ours, we may want to define a constructor for the child and use also the constructor of the parent class. In this case, we rely on the Python built-in function `super()`.

## MRO and `super()` function

While dealing with inheritance, `super()` allows us to call methods of the superclass in our subclass, therefore it becomes fundamental when accessing inherited methods that have been overridden. The primary use case of this is to extend the functionality of the inherited method. Technically speaking, `super()` returns a proxy object that delegates method calls to the parent class.

---

<sup>7</sup>In Python, a decorator is a design pattern that allows to modify the functionality of a function by wrapping it in another function.

Having to deal with calls to parents' methods, when the inheritance tree is a complex structure it becomes determinant to understand which is the family tree, namely what is the *Method Resolution Order* (MRO). `super()` provides the next method according to the MRO, on which you can find information in each class's private class attribute `__mro__`. The MRO consists in ordering parent classes, following the order indicated in the child class declaration and completing each branch; when common ancestors are found, they are always delayed to the end of the branch. The process stops at the "object" class.

In our context, we use `super()` to retrieve the constructor of the parent class, to which we pass the arguments required to instantiate an object of the parent class. Basically, for the constructors, when we call `super().__init__()` it provides the next `__init__()` method according to the used MRO algorithm in the context of the complete inheritance hierarchy.

`super()` can also take two parameters, which determine from which point in the tree to start looking for methods: the first is the subclass, and the second parameter is an object that is an instance of that subclass.

```
1 class First():
2     def __init__(self):
3         print("Building First...")
4         self.x = "Attribute First"
5
6 class Second(First):
7     def __init__(self):
8         super(Second, self).__init__()
9         print("Building Second...")
10        self.y = "Attribute Second"
11
12 class Third(First):
13     def __init__(self):
14         super(Third, self).__init__()
15         print("Building Third...")
16         self.z = "Attribute Third"
17
18 class Fourth(Second,Third):
19     def __init__(self):
20         super(Fourth, self).__init__()
21         print("Building Fourth...")
```

Figure 7: Single and Multiple Inheritance in Python

## Diamond inheritance

In our implementation for the classes in [subsection 5.6](#), we rely on a family tree that is similar to the so-called *diamond shape*: this consists of a class having two parents that share their parent, as the ones considered in [Figure 7](#), indeed, the class `Fourth` inherits both from `Second` and `Third` which inherit both from `First`.

With the code shown above, an instance of `Fourth` has the attributes `y` and `z` from the direct parents and `x` from the "grandparent" class. An instance of `Second` has only the attributes `y` and `x`, while an instance of `Third` has only the attributes `z` and `x`.

The chain of calls to `super()` are used to activate all the ancestors' `__init__()`, and in our case are also used to pass the arguments to the ancestors' constructors, coming from the structure defined in [subsection 5.2.3](#).

### 4.3.2. Abstract Base Class

Abstract classes are used to represent general concepts, which can be used as base classes for concrete classes. In our case, the abstract concept will be for example the one of a training algorithm, where the training pipeline is common to all the methods, but the recipe to sample new parameters changes according to the specific

algorithm. So, we want to store all the common features in an abstract class that can't work without being used in a specific children (more details in [subsection 5.7](#)).

With this inheritance feature, we separate, in a certain sense, the interface from the implementation, because abstract base classes only define generic methods and properties, while a part of their implementation is then handled by the concrete subclasses, of which we can instantiate objects that can handle tasks.

In Python, the module `abc` provides the infrastructure for defining them. From this module, we import specifically `ABC`, the built-in class that is passed in the class declaration as parent class to the classes that we want to make abstract. This forbids the instantiation of the abstract class.

From `abc`, we import as well the decorator `@abstractmethod`, that can be applied to methods of an abstract class if we want to force their overriding in children classes. This prevents us to create instances of classes with abstract parents if not all their abstract methods (also called virtual methods) have been overridden.

This features help to avoid bugs and make the class hierarchies easier to maintain by providing strict recipes to follow when coding methods in the subclasses.

### 4.3.3. Dataclass

It may happen that we are interested in storing data and exploiting at the same time some features of OOP, but we would like to have an object more specific and light than a traditional class (e.g. [subsubsection 5.3.4](#)). From Python 3.7, for this task we can rely on Data Classes that are characterized by the `@dataclass` decorator, provided by the `dataclasses` module and they consist in Python classes with some limitations, thought mainly for containing data (although there is no strict restriction about that).

They allow the user to define classes with less code, because for example, methods such as `__init__()` and `__repr__()` are automatically added without requiring their implementation and all methods are class methods so we do not require the typically redundant `self` syntax of classes in this simpler context.

They also provide more functionalities out of the box; for example, with the option `@dataclass(frozen=True)` we can define a class whose instances are immutable, and with methods such as `asdict()`, `astuple()` an object of a dataclass can be converted to a dictionary or a tuple, other natural data structures for the storage of data.

### 4.3.4. Iterators

In Python, there is the possibility to create classes as iterators, which are objects that you can traverse through all its values; in our project, they are useful for defining training batches ([subsubsection 5.4.3](#)).

An example of implementation of iterators is shown in [Figure 8](#):

- `__init__()` initializes the data attribute that is expected to be an iterable;
- `__iter__()` returns the iterator object. In our example, this is the Data object on which it is called itself using `iter()` on the instance. We initialize the `current_index` with zero, to start with the first datum;
- `__next__()` returns the next value after one iteration. We increment the `current_index` attribute to keep track of the current index of the element in data. This special method is called when `next()` is invoked on the instance.

```
1 class Data:
2     def __init__(self, data):
3         self.data = data
4     def __iter__(self):
5         self.current_index = 0
6         return self
7     def __next__(self):
8         if self.current_index < len(self.data):
9             x = self.data[self.current_index]
10            self.current_index += 1
11            return x
12            raise StopIteration
```

Figure 8: `__iter__()` and `__next__()` special methods

### 4.3.5. Property

Properties can be considered the “Pythonic” way of working with attributes, which are massively present in the data management (subsubsection 5.3.4) because as showed in subsection 2.2 datasets for PINNs are articulated.

Properties’ main strengths are the following:

- you can access instance attributes exactly as if they were public attributes while using intermediaries (getters and setters) to validate new values and to avoid accessing or modifying the data directly;
- by using `@property`, you can reuse the name of a property to avoid creating new names for the getters, setters, and deleters, hence the syntax is concise and readable.

In Figure 9 we show with a basic example the syntax to define the custom functions with this decorator.

```
1 class Person:
2     def __init__(self, firstname, lastname):
3         self.first = firstname
4         self.last = lastname
5
6     @property
7     def fullname(self):
8         return self.first + ' ' + self.last
9
10    @fullname.setter
11    def fullname(self, name):
12        firstname, lastname = name.split()
13        self.first = firstname
14        self.last = lastname
15
```

Figure 9: Example with `@property` decorator

## 4.4. Repository Structure

The detailed description of the repository starts in this section, with a deepening into the purpose and content of the folders `config`, `data` and `outs`.

The presentation of the folder `src`, containing the actual implementation, is postponed to section 5.

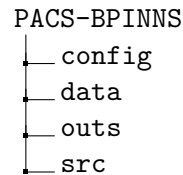


Figure 10: PACS-BPINNS repository

### 4.4.1. Configuration - config Folder

This folder contains the `.json` files for the configuration of the options of test cases, which are the first instructions read by the main executable script and contain information about algorithm parameters, network architecture, dataset choices or generation and various utilities. The files are divided into two main subfolders:

- `test_models`, whose files are work-in-progress test cases;
- `best_models`, containing options of test cases that gave rise to interesting results.

Apart from the distinction in the folder of membership, all configuration files have the same structure: they store `json` dictionaries of dictionaries where the first level of keys represents different categories of settings to be chosen, while the second level of keys consists of the names of the variables to be set.

The first level-keys describing clusters of options are:

1. `general`, storing:

- `problem`, string with the name of the problem to solve (eg. `Regression` for function interpolation, or the name of the PDE for Physics-Informed tasks);
- `case_name`, string with a name identifying the specific dataset for the experiment;
- `method`, string containing the training algorithm to choose;
- `init`, string that is empty to indicate that there is no pre-training, or contains the name of the desired pre-training method.



2. **architecture**, containing information on network architecture:
  - **activation**, string with the desired activation function; we chose the *swish* function;
  - **n\_layers**, number of hidden layers  $L$ ;
  - **n\_neurons**, number of neurons per hidden layer  $K_l$ .
3. **losses**, containing a boolean for each part of the loss to include in the total loss used for learning;
  - **data\_u**, representing the component of solution fitting (Equation 16);
  - **data\_f**, representing the component of parametric field fitting (Equation 18);
  - **data\_b**, representing the component of boundary loss (Equation 17);
  - **pde**, representing the component of the PDE residual (Equation 23);
  - **prior**, representing the prior distribution of network parameters (Equation 19).
4. **metrics**, containing the same keys of **losses** to establish which component of the loss function we want to compute (regardless of the fact that we are using it for learning) and plot its history.
5. **num\_points**, containing the number of data points to use:
  - **sol**, for solution fitting points;
  - **par**, for parametric field fitting points;
  - **bnd**, for boundary points;
  - **pde**, for collocation points.
6. **uncertainty**, containing the values of the std linked to noise and uncertainty mentioned in subsection 2.4:
  - **sol**,  $\sigma_u$  for the solution;
  - **par**,  $\sigma_f$  for the parametric field;
  - **bnd**,  $\sigma_b$  for the boundary data;
  - **pde**,  $\sigma_r$  for the PDE residual.
7. **utils**, containing:
  - **random\_seed**, the random seed set;
  - **debug\_flag**, boolean to print details useful for debugging tasks;
  - **save\_flag**, boolean to save the on-going test in a specific folder in **outs** (see subsection 4.4.3);
  - **gen\_flag**, boolean to generate the dataset requested; it is necessary to set it to true only when the dataset is not already present in the **data** folder.
8. **{method\_name}<sup>8</sup>**, dictionary that stores the parameters of the training algorithm.
9. **{method\_name}\_0<sup>8</sup>**, dictionary that stores the parameters of the method used to pre-train (optional).

The keys of the last two dictionaries are the method names; the second-level keys of the latter two dictionaries depend on the method chosen; for the algorithms implemented in the project, we have:

1. **ADAM** (algorithm 3)
  - **epochs**, number of epochs  $N$ ;
  - **burn\_in**, delay for the physical loss in the learning process;
  - **lr**, learning rate  $\eta$ .
  - **beta\_1**, exponential decay rate for the first moment estimates  $\beta_1$ ;
  - **beta\_2**, exponential decay rate for the second moment estimates  $\beta_2$ ;
  - **eps**, small number to prevent any division by zero  $\epsilon$ ;
2. **HMC** (algorithm 4)
  - **epochs**, total number of samples  $N$ ;
  - **burn\_in**, burn-in  $B$  and delay for the physical loss in the learning process;
  - **skip**, number of samples to skip among subsequent samples  $S$ ;
  - **HMC\_L**, number of leap-frog steps  $L$ ;
  - **HMC\_dt**, temporal step  $\Delta t$ ;
  - **HMC\_eta**, mass matrix parameter  $\alpha$ .
3. **VI** (algorithm 5)
  - **epochs**, number of epochs  $N$ ;
  - **burn\_in**, delay for the physical loss in the learning process;
  - **samples**, number of desired samples of  $\theta$ ;
  - **alpha**, learning rate while learning  $\zeta$ .
4. **SVGD** (algorithm 6)
  - **epochs**, number of epochs  $E$ ;
  - **burn\_in**, delay for the physical loss in the learning process;
  - **N**, total number of particles  $N$ ;
  - **h**, bandwidth in kernel definition  $h$ ;
  - **eps**, step size  $\varepsilon$ .

---

<sup>8</sup> The name of the dictionary is one among the algorithms implemented: **ADAM**, **HMC**, **VI**, **SVGD**.



#### 4.4.2. Datasets - data Folder

This folder contains the data generated for the proposed test cases, stored in `.npy` files (NumPy array files). Each time the `gen_flag` cited in [subsection 4.4.1](#) is set to `True`, a folder for the current test case is generated or overwritten.

If instead the data do not need to be generated, but are given from external sources or have already been created, the user needs to set the `gen_flag` to `False`, and, in the first case, upload the files following the same name convention of the created data.

As shown in [Figure 11](#), data are separated by type of problem in folders named after the task (for the **Regression** case), or the differential equation in the physics-informed problem (**Laplace1D**, **Oscillator...**).

Then, a second level of folders can be found where we store the specific dataset for that problem, to distinguish, for example, the Laplace 1D problem with different functions.

Inside each subfolder, we finally find the data, split into different files:

- **Boundary data:** `dom_bnd.npy`, `sol_bnd.npy` for the inputs and solution values of boundaries;
- **Collocation data:** `dom_pde.npy` for the inputs of collocation points;
- **Fitting data:** `dom_sol.npy`, `sol_train.npy` and `dom_par.npy`, `par_train.npy` for inputs and values for solution or parametric field;
- **Test data:** `dom_test.npy`, `sol_test.npy`, `par_test.npy` for inputs and values of test points.

#### 4.4.3. Outputs - outs Folder

This folder contains the results and the summarizing statistics of each test case and the folder hierarchy (see [Figure 12](#)) is similar to the `data` folder: the first level denotes the type of problem, the second enables to fully characterize the test case by specifying the dataset.

There is another level of indentation: inside each subfolder, each run of the code with the option `save_flag` equal to `True`, generates a folder, whose name contains date and time of the execution.

The `trash` folder instead contains the output of the latest test case run with the `save_flag` equal to `False` and is cleaned each time.

Inside each test case folder, there are the following subfolders:

1. `log`, storing the `.txt` files:

- `parameters.txt`, reproducing the `.json` configuration file;
- `errors.txt`, reproducing the output on the screen after each run, with relative errors on solution (and eventually parametric field) and uncertainty quantification,
- `keys.txt`, with the components of the loss used in the learning process;

and the `.npy` files:

- `loglikelihood.npy`, history of values of the log-posterior (up to a constant);
- `posterior.npy`, history of values of the posterior (up to a constant);

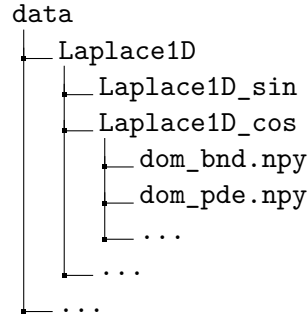


Figure 11: Hierarchy of the folder `data`.

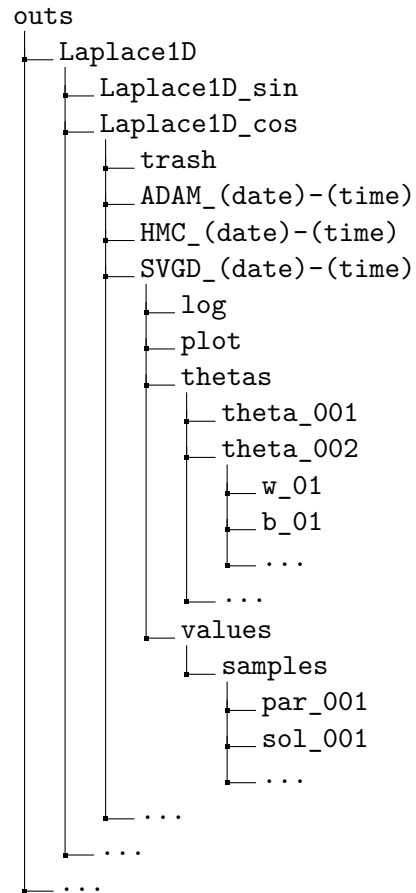


Figure 12: Hierarchy of the folder `outs`.

2. `plot` storing the graphical output. The files that can be found are:
  - `u_confidence.png` mean of the predicted distribution for the solution, with confidence interval given by mean  $\pm$  standard deviation;
  - `u_nn_samples.png` plot of several samples from the predicted distribution for the solution;
  - `f_confidence.png` mean of the predicted distribution for the parametric field, with confidence interval given by mean  $\pm$  standard deviation;
  - `f_nn_samples.png` plot of several samples from the predicted distribution for the parametric field;
  - `Mean Squared Error.png`, plot of the MSE;
  - `Loss (Log-Likelihood).png`, plot of the log-posterior (up to a constant);
3. `thetas`, storing all the sampled network parameters. For each sample of parameters, a subfolder named `theta_(number)` is created, containing as many `.numpy` files as the double of the number of layers: each file indeed stores the values of the weights or of the biases in each layer;
4. `values`, containing:
  - `sol_NN.npy` mean of the predicted distribution for the solution;
  - `sol_std.npy` standard deviation of the predicted distribution for the solution;
  - `par_NN.npy` mean of the predicted distribution for the parametric field;
  - `par_std.npy` standard deviation of the predicted distribution for the parametric field;
  - the subfolder `samples`, containing in `.numpy` files the values of solution and parametric field corresponding to each selected sample of network parameters.

## 5. Source Code

In this section, we deepen into the source code, that is separated from configuration, data and outputs and stored in the folder `src`.

The outline of this folder is shown in Figure 13: it includes three executable scripts and specific modules either for data generation and preprocessing, implementation of the B-PINNs method, performance and UQ evaluation and for post-processing utilities such as visualization and saving.

In the code development, we established and followed three general guidelines:

```
src
├── algorithms
├── equations
├── networks
├── postprocessing
├── setup
├── utility
├── main_data.py
├── main_loader.py
└── main.py
```

Figure 13: Hierarchy of `src` folder.

### I - Mirror the theoretical framework in the class hierarchy

The theory of B-PINNs suggests that the model should have a modular structure, because a B-PINN can be obtained mounting blocks one on each other:

- starting from a fully connected *Neural Network*;
- adding a non-deterministic training algorithm to make it *Bayesian*;
- including PDEs in the loss to make it *Physics-Informed*.

In this way, one can select which blocks to assemble in order to implement the whole hierarchy of models, such as a BNN or a PINN, and compare them with a BPINN.

The three above components are highlighted in the code by the membership to different building blocks: the general idea is to build the final model starting from the class `CoreNN`, containing only the essential Neural Network features (initialization, forward step); on that, we mount the information coming from physics with the child class `PhysNN`, from which in turn the `BayesNN` is created.

Moreover, side children classes are devoted to specific functionalities, such as the operations on loss functions (`LossNN`) and all that concerns the computation of metrics for uncertainty quantification (`PredNN`).

### II - Promote code reuse and extension addition.

As the main goal of the project is to lay the foundations for a library for Bayesian Machine Learning, we paid attention at choosing an enough flexible architecture so that the code could be reused for similar tasks.

A straightforward example of this is the effort to enable a handy application of the training algorithm (either bayesian or deterministic) to the network: to avoid code duplication, indeed, we managed to keep the main structure of training process shared in an abstract class and then limit the differences to specific methods into the various algorithm classes, without making repetition in how the algorithm acts on the network or on how the network stores and upgrades the parameters.

We also developed specific features which are of common utility for different training algorithms, such the overloading of the algebraic operations on the ensemble of network parameters (with the class `Theta`), which turned to be fundamental for a readable and compact code in the SVGD implementation.

For what concerns the addition of extension, this structure is enough flexible to enable the implementation of new training algorithms: this has been assessed while developing the four training algorithms proposed. Indeed, as explained in section 3, the algorithms differ significantly in terms of steps to be performed, selection of the parameters sample and structures to involve, but the proposed framework enabled to take each algorithm's peculiarities into account avoiding code duplication.

### III - Abstract from the single application.

In this project, we proposed some test cases used to validate the method and aimed at showing the main characteristics of Bayesian Physics-Informed Machine Learning, but the code was first of all designed with the purpose of limiting the influence of the single case of application into the core structure.

What changes according to the application is the dimension of the domain and of the output space (hence, the number of components of solution and parametric field). The code executed, until the stage of error computation and UQ, is exactly the same for all input and output dimensions. What changes then is only the visualization stage, for which we necessarily need to invoke different visual strategies according to the dimensions.

Another aspect that varies with the test case is, for example, the underlying physic law, represented by the PDE. Depending on its complexity, ad hoc methods must be used for pre-processing of raw physical data and computation of residuals and those could lead to change in model structure. Our code manage all of this possibilities with few constraint on virtual methods, allowing full independence between physical and computational data.

All these aspect are important because usually raw data don't allow DL models to work in their euristic "confort zone" where they work in an optimal way.

## 5.1. Main Files

As shown in [Figure 13](#) inside the `src` folder, the user can find executable `.py` scripts.

- **main.py** This is the main executable script, as it performs the task of approaching a problem with a B-PINN. Since all the specific tasks are performed in separate scripts, the job of the `main` consists of calling them all in the correct order as explained below.
- **main\_data.py**; with this script, the user can generate a new dataset without training a B-PINN. The new dataset is stored into a new subfolder in the `data` folder.
- **main\_loader.py**; with this script, the user can load the output of a previously saved test case. The user can read from the terminal the available test cases, and the prompts asks for selecting first the problem and then the dataset. Once the choice is performed, all the plots - namely, the content of the folder `plot` in `outs`, are opened.

### 5.1.1. Main Pipeline

The `main.py` script present our working pipeline and can be divided into sections achieving various tasks:

1. **Import and Setup:** macros for printing and path management are imported, then all modules in the `src` folder are loaded and the configuration file is chosen.
2. **Creation of Parameters:** here, information in the configuration file (the `.json`) is integrated or updated with command-line arguments, and are all stored in a `Param` object.
3. **Creation of Dataset:** if the generation of a new dataset is asked, calls the data generator for the specific test case; otherwise, loads the suitable dataset. Then perform the preprocess needed for the training.
4. **Model building:** initialization of `Equation` and `BayesNN` objects with parameters of `Param` object.
5. **Model training:** optimization algorithm is initialized and perform training (and pre-training if needed) on the previously created `BayesNN` with data coming from the already pre-processed dataset.
6. **Model evaluation:** computation of statistics for UQ with input data of the test set and then comparison of prediction through computation with test dataset real values for model performance evaluation.
7. **Saving:** storage of the results and of the relevant test case details, generating a folder as described in [subsection 4.4.3](#) or temporary storing all into the `trash` folder.
8. **Plotting:** the stored results are then read by the `Plotter` class and displayed on the screen.

## 5.2. Parameters Handling

The management of the information on test-case options is contained within the `setup` module, which, among its functionalities, provides classes for finalizing the complete list of settings for the case study.

The finalization of this set of instructions requires the integration of information coming from different sources, because we separated the details relative to the test case (such as method and simulation options) from the ones more specifically concerning the dataset (such as the domain and the functions involved).

Moreover, options can be specified by the user from command line, hence it is necessary to include a way to update them.

### 5.2.1. Configuration files

At the beginning of the main script, we load the information coming from different sources, as said above, among which the major part is represented by are the configuration files described in [subsection 4.4.1](#). They are written in `.json` format and imported in a Python dictionary.

Each of them represents a specific test case and uses data coming from a specified `data` folder and stores the parameters needed for collect those data.

Although most of the parameters are provided by the configuration files and must be modified there, we also provide a faster way to make small minor changes to the current test case through command line arguments.

### 5.2.2. Command line arguments

In the code, we included the possibility to specify from command line each test case and method parameter, to update the information that already have default values stored in the `.json` configuration files.

To guarantee this feature, we implemented the class `Parser`: it inherits from the Python's built-in class `ArgumentParser`, that enables to parse command line strings into Python's objects.

This class does not have any method, as it plays the role of a container of information to which we do not require any active action; for this reason, it has only a constructor in which we add command line arguments with the built-in method `add_argument()`.

### 5.2.3. Param object

The class `Param` is designed to contain the complete set of user-defined options for the test case; its constructor asks indeed both for the content of the configuration files and for the command line arguments handled with the aforementioned class `Parser`.

Apart from reading the test case parameters from the two sources, this class is in charge of reformat them into more practical versions, for example by converting strings of booleans to booleans or by separating pre-training from training options.

In order to have a comprehensive collection of the method details within objects of this class, we included in this class also the information on the dataset (mesh, subdomains, functions...); while information on the test case are directly stored into class attributes, the ones on the dataset are stored in the property `data_config`, whose custom setter enables to store the content of a dataset object as the ones presented in `datasets` in suitable class attributes.

This object only purpose is to be passed to all high level features and classes of our code to provide them of all mandatory information and parameters needed for all their processes.

## 5.3. Data Generation

The class `DataGenerator`, within the `setup` module, is devoted to the generation of the content of the folder `data`: the `.npy` files storing the spatial coordinates and the exact values of solution and parametric field.

The creation of all the files is performed by the private method `create_domains()`, that comprehends the generation of collocation, fitting, boundary and test data.

### 5.3.1. Domain creation

The basic tool for the generation of a domain is the private method `create_domain()`, which, given extrema, can build a specific mesh type on a domain. The available types are:

- **Uniform domain**, where points are collocated on the corners of a regular grid; this type of mesh has been used for test points, in order to ensure validation on all regions and to produce clean plots. Instead, for what concerns fitting and collocation, this mesh type is not suitable.  
Consider for example what happens by asking a limited amount of points when working with periodic functions: if we put points on a grid we could have bad luck and obtain inputs corresponding to the same function output, which suggests to the network that the function is constant.  
With test data, instead, we are typically free from this risk - not only because they are not used for learning, but also because they are in a generally pretty high number.
- **Random domain**, where points are randomly sampled from the interior of the domain. However, as highlighted by Figure 14, this choice does not guarantee a good coverage of the domain, and for example in the regression task the presence of holes in the domain can have consequences on the quality of the prediction. This limitation can be overcome with the implementation of the following pseudo-random set of points;
- **Sobol domain**, that consists of a purely deterministic sampling of points generated with a common technique in the family of Quasi Monte-Carlo (QMC) methods, which enables to improve the convergence of Monte-Carlo estimators without complicating their structure. The characteristic of the point set that enables this gain is its low *discrepancy*, that, without deepening into its mathematical definition, is a measure of the equidistribution of points.

In order to define Sobol points, we need first to introduce general categories of low-discrepancy points that are  $t - m - d$  nets and  $t - d$  sequences, that we define in a  $d$ -dimensional space.

**Definition 5.1** (*t - m - d nets*). Let  $0 \leq t \leq m \in \mathbb{N}, b \geq 2$ . A *t - m - d net* in base *b* is a point set  $\mathcal{P}_n$  with  $n = b^m$  points such that each elementary rectangle of volume  $b^{t-m}$

$$R_a = \prod_{j=1}^d \left[ \frac{a_j - 1}{b^{p_j}}, \frac{a_j}{b^{p_j}} \right) \quad a_j = 1, \dots, b^{p_j}$$

with  $p_1 + \dots + p_d = m - t$  contains exactly  $b^t$  points.

**Definition 5.2** (*t - d sequences*). A *(t - d) sequence* in base *b* is a sequence of points  $\mathcal{S} = \{X_0, X_1, \dots\}$  such that for any  $m > t$ , every block of  $b^m$  points  $\{X^{(lb^n)}, \dots, X^{((l+1)b^n-1)}\}, l \in \mathbb{N}$  is a *t - m - d net* in base *b*.

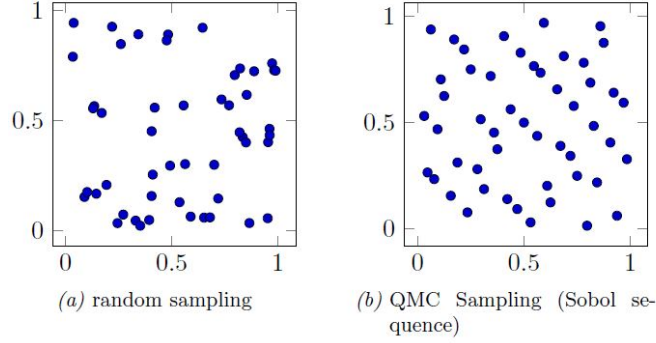


Figure 14: Comparison between Random and Sobol sets of points.

The set of Sobol points is a particular type of *(t - d)* sequences in base  $b = 2$ ; note that Sobol sequences are optimized for samples of dimension  $n$  that is a power of 2, and lose their balance properties if one does not respect this choice.

In practice, to generate the sequence of points in the code we relied on the `qmc` module of `scipy.stats`, which provides a generator for Sobol sequences in the unit hypercube. Among the options provided for the draw, we chose `random_base2()`, that safely draws  $n = 2^m$  points guaranteeing the balance properties of the sequence. Then, we added a posteriori the right bounds of the sequence which were not included in the sample, and rescaled the sample from the unit hypercube to our general domain.

### 5.3.2. Multi-domain creation

For fitting data, representing measurements, we included the possibility to localize points into some specific parts of the domain, in order to mimic a real measurement situation in which the localization data is bounded to the feasibility of localization sensors. In a real domain, indeed, there can be some unreachable regions where sensors can not be installed: for example, they can be situated only near the border.

Test cases with localized data can also highlight the contribution that can arise from the knowledge of the underlying PDE in regions where data are not available.

Note that in our implementation for collocation points, we assumed the PDE to hold in the whole domain and therefore do not include the creation of multidomains; however, the latter can still be implemented with a minor change, if we want to represent a physical law which only holds on a part of the domain.

Data in subdomains are generated with the private method `create_multidomain()`, which reads from the dataset configuration file the extrema of each subdomain, equally distributes the number of points among the subdomains and creates a list of subdomains by calling iteratively the aforementioned method `create_domain()`.

However, an additional detail should be taken into account: we wish to locate the points in the final data structure in such an order that, when taking less points than the available resolution, the points in the subset are equally distributed between the subdomains, and for example avoid to cover only the first region.

To this aim, we implemented the auxiliary private method `merge_2points()`, which merges two sequences of points into one alternating the points.

### 5.3.3. Boundary Points

The generation of the data on the boundary required a different strategy, as it consists of a domain having one dimension less with respect to the physical one, which lives, let us say, in  $\mathbb{R}^d$ .

The implementation relied on the idea of taking the projection of the  $d$ -dimensional domain on each edge of the boundaries: clearly, this strategy is limited to rectangular domains.

### 5.3.4. Data Configuration

This module is devoted to the configuration of the dataset: the information contained in its classes are in a certain sense complementary with respect to the ones in the `config` files, because here the focus is not on training methods but on all the information needed to generate a dataset for a test case.

Here you can find all the specification for the generation of a stand-alone dataset, such as the computational domain, the mesh and resolution options, the coefficients appearing in the PDE and the expression of the functions involved in the problem.

The information is organized hierarchically in classes, with a gradual level of specification as it can be seen for the Laplace case in Figure 15. All the classes proposed in this folder are only designed to store data, hence we accompanied them by the decorator `@dataclass` (presented in subsection 4.3.3).

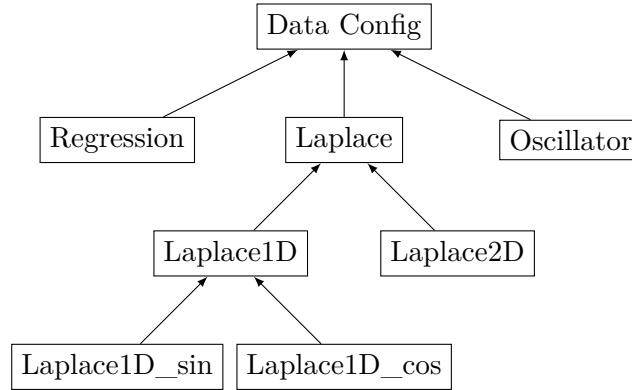


Figure 15: Inheritance tree of the dataset configuration classes.

## Templates

The templates here presented are in charge of fixing a structure whose details can be specified in the different test cases. We created an abstract base class `Data_Config` that acts as a blueprint for the various dataset configuration classes, which all share attributes like the domain and resolution definition.

Then, we implemented children templates for the various problems analyzed (`Laplace`, `Oscillator`, ...) with a specified problem name and a dictionary for the problem parameters.

From these, we obtained the final templates that represent both the problem and the dimension under analysis, after assigning the correct dimensions to spatial input, solution and parametric field.

## Config

Having built templates for each specification of problem and dimension, we then generated children classes for the specific test case, that can correspond to a specific choice of functions involved or values of PDE coefficients.

Each class of the submodule has a property, `values`, returning a dictionary with `lambda` expressions of the solution and the parametric field.

Among the attributes, we highlight `domains`, that contains the specification of the subdomains in which data of the solution and the parametric field data are concentrated. Instances of these classes represent the dataset configuration that is then set into the `Param`'s `data_config`.

## 5.4. Pre-Processing

Apart from `data_generation`, the module `setup` is in charge of other two dataset-related tasks, that enable to assemble with order the articulated dataset required to train PINNs.

On one hand, we need indeed to assembly the actual dataset used to train the network, which needs to be either transformed and organized in the categories described in subsection 2.2. These tasks are performed in `data_creation`. On the other hand, to improve further data for training, we can also organize them into training batches; this feature is implemented in `data_batcher`.



### 5.4.1. Dataset creation

The class `Dataset` is devoted to the management of the data generated with `data_generation` in the folder `data`, which represent the stock material to draw from when assembling the dataset used by the network.

To prepare them for usage in the model, for example, data can be pre-processed with an arbitrary transformation that can improve the model's performance, or they can be modified by adding some noise if we want to train the network on blurry measurements of the quantities.

Therefore, we introduce a discrepancy between the *real* data and the *synthetic* data, that are the transformed quantities exploited for model training. All these transformations are performed by methods of the class `Dataset`, which in the end represents the dataset to which training and test procedures have access.

### Data selection

Still in the configuration files or from command line, the user can specify how many data to use for training (and specifically for collocation, fitting and boundary): if the quantity asked  $N$  is smaller than the maximum domain resolution available, we need to make a selection of data points from the files in the `data` folder.

Note that in the selector properties (`data_pde`, `data_bnd`, `data_sol`, `data_par`), we just take a slice of the first  $N$  elements of the complete data structure, because during the generation we already ensured a storage mechanism that enables to obtain representative samples of the whole domain even by slicing the array in this straightforward way.

### Addition of noise

The values of solution and parametric field stored in the `data` folder correspond to their exact values; with this choice, in experiments where the noise level changes, we do not need to re-generate the whole dataset.

When the user asks to work with noisy data, it specifies the noise level (standard deviation of the gaussian noise) from command line or from the configuration files: the private method `add_noise()` has then the task of generating the dataset with the requested level of noise, read from the inputs.

### 5.4.2. Data normalization

The public methods `normalize_dataset()` and `denormalize_dataset()` are in charge of the normalizing transformation, that turned out to be determinant for model performance.

Indeed, during the validation phase, we noticed that from unnormalized data inaccurate results arose, because even in the regression task the network showed difficulties while trying to reconstruct simultaneously two functions living in a different range of values.

Therefore, we temporarily brought them in the same order of magnitude, by training the model on a normalized version of data; the strategy adopted consisted in normalizing the solution and parametric field values by subtracting the mean  $\mu$  and dividing by the standard deviation  $\sigma$ :

$$\mathbf{u}_{norm} = \frac{\mathbf{u} - \mu_u}{\sigma_u} \quad \mathbf{f}_{norm} = \frac{\mathbf{f} - \mu_f}{\sigma_f}$$

The normalizing statistics  $\mu$  and  $\sigma$  have been computed after having read the numerical values from the folder `data`, and have been then stored in a dictionary class attribute `norm_coeff`. This is then exploited to de-normalize the prediction, bringing it back to its true range.

Normalization has consequences on the entrance of the PDE in the model as well: first of all, we need to re-write the PDE in a normalized form, because the network sees only the normalized data and therefore needs to learn a suitably scaled physical law, which is described by a transformed set of parameters  $\lambda_{norm}$ .

### 5.4.3. Data Batching

While loading a dataset in Machine Learning models, it is common to split them in *batches*. This strategy consists in loading not the whole dataset at once into the memory, but only a sample set of it at a time.

The resulting advantage is in terms of speed: when working with a big dataset, indeed, evaluating the loss with the whole dataset can be computationally demanding, and it could be better to evaluate the loss more times, but on less data.

Therefore, we implemented the possibility to train with batches in the code: the class `BatcherDataset` performs this task, because it plays the role of a dataset with the same properties of the `Dataset` mentioned in `data_creation` that represent training data divided and organized in the required number  $N_{batch}$  of batches.



In the class constructor, starting from a dataset of length  $N_{tot}$  we build batchers of size  $\lfloor \frac{N_{tot}}{N_{batch}} \rfloor$  for each type of training points; each category is represented by a **Batcher** object, a class storing values of data and indexes from the original dataset. The latter are stored into instances of the class **BatcherIndex**, which implements a random way to split the  $N_{tot}$  indexes into batches.

We iterated over all batches by calling the method `next()`: since this method, as stated in [subsection 4.3](#), calls the object's `__next__()` method, in all the three batch-related classes you can find the overriding of this private method. Note indeed that, even if at the beginning of each training step we call `next()` only on the **BatcherDataset** object, the method relies on the `next()` methods of the class **Batcher**, which, in turn, calls internally `next()` on **BatcherIndex**.

## 5.5. Equations

The module **equations** is devoted to the computation of the residuals of PDEs that are required in the context of Physics-Informed Machine Learning; these quantities, as shown in [subsection 2.2](#), contribute to the value of the log-likelihood and, since we are dealing with differential equations, require tools for differentiation of the functions reconstructed by the network.

In the **equation** module, we implemented two main features:

- a module which enables to evaluate differential operators on scalars, vectors and tensors, which are then used within the methods that compute the PDE residuals;
- specific classes for the analyzed equations, whose instances are designed to become class attributes of the network, and, specifically, will be put under the network part devoted to physical information (**PhysNN**).

### 5.5.1. Operators

The class **Operators** contains static methods for the differential operators and auxiliary static methods for Tensor manipulation, mainly aimed at reshaping Tensors in a portable way.

We implemented the operators trying to rely on the basic ones already coded; we started with the basic implementation of **gradient\_scalar**, which computes the gradient of a scalar function. From this operation, we were able to implement **gradient\_vector** which calls the scalar version on each function component with list comprehension.

For what concerns the divergence of a vector, we computed it as the trace of the tensor representing its gradient, while the divergence of a tensor calls the vector version within list comprehension.

Finally, the laplacian of a scalar was obtained as divergence of the gradient vector, while for the laplacian of a vector we relied again on the scalar version with list comprehension.

Operator	Version	Analytical Expression	Input Shape	Output Shape
Gradient	scalar vector	$\nabla u = [\partial_1 u, \dots, \partial_{n_{in}} u]^T$ $\nabla \mathbf{u} = \begin{bmatrix} [\partial_1 u_1, \dots, \partial_{n_{in}} u_1]^T \\ \vdots \\ [\partial_1 u_{n_{out}}, \dots, \partial_{n_{in}} u_{n_{out}}]^T \end{bmatrix}$	$n_s \times 1$ $n_s \times 1 \times n_{out}$	$n_s \times n_{in}$ $n_s \times n_{in} \times n_{out}$
Divergence	vector tensor	$\text{div} \mathbf{u} = \partial_1 u_1 + \dots + \partial_{n_{in}} u_{n_{in}}$ $\text{div} \underline{\underline{\mathbf{u}}} = \begin{bmatrix} \partial_1 u_{1,1} + \dots + \partial_{n_{in}} u_{1,n_{in}} \\ \vdots \\ \partial_1 u_{n_{out},1} + \dots + \partial_{n_{in}} u_{n_{out},n_{in}} \end{bmatrix}$	$n_s \times n_{in}$ $n_s \times n_{in} \times n_{out}$	$n_s \times 1$ $n_s \times 1 \times n_{out}$
Laplacian	scalar vector	$\Delta u = \partial_1^2 u + \dots + \partial_{n_{in}}^2 u$ $\Delta \mathbf{u} = \begin{bmatrix} \partial_1^2 u_1 + \dots + \partial_{n_{in}}^2 u_1 \\ \vdots \\ \partial_1^2 u_{n_{out}} + \dots + \partial_{n_{in}}^2 u_{n_{out}} \end{bmatrix}$	$n_s \times 1$ $n_s \times 1 \times n_{out}$	$n_s \times 1$ $n_s \times 1 \times n_{out}$

Table 2: Overview of the differential operators provided by the class **Operators**.

In [Table 2](#) you can find a summary of the available operators, with a specification of the input and output shape. The notation adopted is the following:

- $n_s$  is the number of samples in which the function is evaluated;
- $n_{in}$  is the dimension of the input space;
- $n_{out}$  is the dimension of the output space;
- $u$  represents a scalar function;
- $\mathbf{u}$  represents a vector-valued function;
- $\underline{\underline{u}}$  represents a tensor-valued function.

### 5.5.2. Abstract Equation

The class **Equation** is an abstract base class and represents the common blueprint from which children classes representing the various PDE constraints are derived (Figure 16).

This class contains attributes storing information about the physical dimensions of the problem and its coefficients and one abstract method (`comp_residual()`) devoted to the computation of the residual of the equation, overridden in each children class.

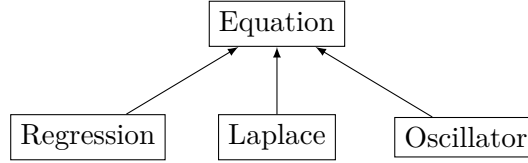


Figure 16: Inheritance tree of the **equation** classes.

### 5.5.3. Problems Implemented

We now deepen into the problems implemented: the first is the regression application case, not involving any differential model, while the following ones represent physical phenomena on which we have both data and knowledge of the underlying PDE.

#### Regression

Objects from the **Regression** children class are instantiated during test cases which involve the simple regression task, which is the reconstruction of a function interpolating some given points.

In this case, there is no PDE whose residual needs to be computed, hence only the attributes about dimension are retained; the `comp_residual()` methods just raises an exception to highlight an error in case it is (wrongly) invoked during the regression task.

#### Laplace

The children class **Laplace** implements the computation of the residual of the elliptic Poisson/Laplace PDE with diffusion coefficient  $\mu \in \mathbb{R}$ , which reads as:

$$-\mu \Delta \mathbf{u} = \mathbf{f}$$

However, the equation that the function reconstructed by the network has to satisfy is slightly different from the one describing the physical problem, because it should be adapted after the normalization of the dataset as anticipated in subsection 5.4.1.

The normalization information is stored in the class property `norm_coeff`, which has the normalization coefficients, that are enforced into the equation object during pre-process with the information from the dataset.

Then, having the normalization coefficients available, the `comp_residual()` method computes the residual of the equation for the normalized functions, that reads as:

$$-\mu \frac{\sigma_u}{\sigma_f} \Delta \mathbf{u}_{norm} = \mathbf{f}_{norm} + \frac{\mu_f}{\sigma_f}$$

#### Oscillator

The children class **Oscillator** implements the computation of the residual for the damped harmonic oscillator. A harmonic oscillator is a system with a mass  $m$  that, when displaced from its equilibrium position, experiences a restoring force  $F$  proportional to the mass displacement  $x$ ,  $F = -kx$ , where  $k > 0$  is a constant that can represent for example the elastic constant of a spring.

By recalling Newton's Second Law, for which  $F = ma = m \frac{d^2x}{dt^2}$ , we obtain:

$$m \frac{d^2x}{dt^2} + kx = 0 \quad (41)$$

The solution of the differential equation 41 describes a periodic motion, repeating itself in a sinusoidal fashion with constant amplitude  $A$ :

$$x(t) = A \cos(\omega t + \phi)$$

where  $\omega = \sqrt{\frac{k}{m}}$  and  $\phi$ , called phase, determines the starting point on the sine wave.

The period and frequency are determined by the mass  $m$  and the force constant  $k$ , while the amplitude  $A$  and phase  $\phi$  are determined by the starting position and velocity.

The system may undergo frictional forces as well, which are proportional to the velocity  $\frac{dx}{dt}$ : in this case, we obtain the damped harmonic oscillator, which is the one of the application implemented in the code. Denoting the friction constant by  $c$ , the equation of the force balance reads as:

$$F = -kx - c \frac{dx}{dt} \quad (42)$$

and again Newton's second law enables us to rewrite 42 as

$$\frac{d^2x}{dt^2} + 2\delta \frac{dx}{dt} + \omega^2 x = 0 \quad (43)$$

where  $\delta = \frac{c}{m}$ . In this case, the amplitude of the oscillations decreases with time, and the evolution of the position  $x$  reads as:

$$x(t) = Ae^{-\delta t} \sin(\psi t)$$

where  $\psi$  is linked to  $\omega$  and  $\delta$  by the relation  $\phi = \sqrt{\omega^2 - \delta^2}$ .

This problem differs from previous one because it is a time dependent ODE, so it might seem like a simplification of the previous PDE case, but still it is a very useful way to show a powerful application of PINN.

In fact, an obvious configuration for this problem is to split time domain in two parts: the first one with data measurements and the following without any clue, to then try to reconstruct this side through the knowledge of the physics behind the problem as shown in [subsection 6.2](#).

## 5.6. Model

In this section, we present the content of the module **networks**, devoted to all that concerns the neural network functionalities.

The strategy adopted consists in building children class over the parent class **CoreNN**, that contains only the basic neural network functionalities: initialization, storage of weights and biases, forward step with given parameters. Upon that, we build the children class **PhysNN**, whose main role is to contain the information coming from physics and normalization constants (which can be in turn problem-specific).

From **PhysNN**, two children classes derive: **LossNN**, ideated for loss and loss gradient computation, and **PredNN**, whose methods are the functionalities used for the *prediction* and *testing*, as it contains uncertainty quantification and error computation functions.

The class of which we instantiate an object in the **main** script is **BayesNN**, combining all the features of the ancestor classes and managing the history of loss during training. The above described inheritance chain is sketched in [Figure 17](#), and works with the tools described in [subsection 4.3](#).

Apart from network classes, this module contains also a class, **Theta** designed to handle neural network parameters (weights and biases) and containing the overloading of operations for Theta objects and few recurrent methods used in the algorithms.

### 5.6.1. Parameters object - Theta

This class is used to produce objects that represent the NN parameters  $\theta := \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^L$ . The choice of devoting a specific class to them comes basically from the opportunity to improve code readability by overloading the recurrent algebraic operations on them.

In the training algorithms, indeed, it is often required to perform algebraic operations on NN parameters, but the data structure that represents them by default in TensorFlow does not enable a simple usage of them. This structure consists is a list of **tensors** of different shapes, whose length is twice the number of layers; each

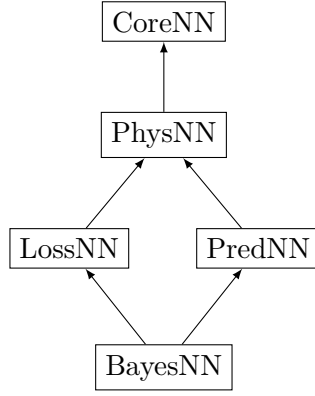


Figure 17: Inheritance tree of the `network` classes.

element of the list is, alternatively, a `tensor` with the weights or a `tensor` with the biases for each layer. Notice that for `tensors` the most common algebraic operations have been overloaded, so, for example, you can do the element-wise sum of `tensors` just by using the `+` operator. But for a list of `tensors`, the `+` operator would not perform the desired task, as `+` has been overloaded for lists to stand for list concatenation.

Among the advantages of overloading the principal operators, we particularly mention:

- an improvement of *code readability* by allowing the use of familiar operators;
- a guarantee that objects of a class behave consistently with built-in types and other user-defined types;
- a *simplification in writing code*, especially when many operations are required for complex data types;
- an enhancement of *code reuse* by implementing one operator defined in chain.

## Class constructor and properties

The constructor asks for the NN parameters, meant to be represented as a list of tensors by following TensorFlow's convention. They are then stored in the attribute `values`.

The class has two properties: `weights`, containing the implementation of the custom getter for weights, which extracts elements from the list `values` starting from the first with a step of two.

The property `biases` represents the same, but for biases, and it extracts the rest of list elements.

## Operation Overloading

In Python, there are specific method names for operator overloading, and these method names are unique for each operator. The overloading consist therefore in overriding the private methods having the names shown in Table 3. Sum, subtraction, multiplication and division have been implemented either when both terms are `Theta` objects, either when one of the two is a scalar `int` or `float`.

We can divide the function that we redefine in one of the following categories, according to the operation:

- classical binary operation  $(\theta_1, \theta_2)$ : return a `Theta` object whose `values` are the element-wise operation applied to the pair of `Theta` objects;
- polymorphic binary operation  $(\theta, n)$ : works as a `Theta` object with all entries equal to  $n$  is passed as second argument;
- reverse polymorphic binary operation  $(n, \theta)$ : works as before but changing position of arguments. This is obtained because when overloading if the first argument of a binary operation is unable to deal with it an error is raised, but can be caught by special methods in the second argument;
- unary operations  $(\theta)$ : the implementation heavily depends on the task. For example, the opposite is implemented through a multiplication by the scalar  $-1$ , the power applies the TensorFlow power function to each list element, ...

In addition, we implemented the following public methods:

- `ssum()`, to compute the squared sum of all the entries in `self.values`;
- `size()`, to count all the entries in `self.values`;
- `copy()`, that returns a `Theta` object whose `values` are an independent copy of the current object's `values`;
- `normal()`, that creates a `Theta` object with random normal initialization of `values`.

In Table 3 we report all the operations that we have overloaded.

Operation	Method	Sign	Input	Output
Opposite	<code>--neg--</code>		$\theta$	$-\theta$
Sum	<code>--add--</code> <code>--radd--</code>	+	$\theta_1, \theta_2$	$\theta_1 + \theta_2$
Subtraction	<code>--sub--</code> <code>--rsub--</code>	-	$\theta_1, \theta_2$	$\theta_1 - \theta_2$
Multiplication <sup>9</sup>	<code>--mul--</code> <code>--rmul--</code>	*	$\theta_1, \theta_2$	$\theta_1 * \theta_2$
Division <sup>9</sup>	<code>--truediv--</code> <code>--rtruediv--</code>	/	$\theta_1, \theta_2$	$\theta_1 / \theta_2$
Power <sup>9</sup>	<code>--pow--</code>	**	$\theta, 2$ $\theta, 0.5$ $\theta, -1$	$\theta^2$ $\sqrt{\theta}$ $1/\theta$
Class Methods	<code>--len--</code> <code>--str--</code>	len print	$\theta$	len( $\theta$ ) print( $\theta$ )

Table 3: Overloaded operations for **Theta** objects.

### 5.6.2. Parameters and forward pass - CoreNN

This class is designed to:

- store attributes concerning the neural network architecture;
- store and set network parameters;
- store the `model`, instance of the class `tf.keras.Sequential` that groups a linear stack of layers into a model;
- initialize the `model` with a given seed;
- perform the forward step `output = model(inputs)`.

#### Constructor and properties

The class constructor requires `par`, object of the `Param` class, needed to retrieve the information about input/output dimensions and network architecture. It also initializes the `model` by calling the private method `build_NN`, and computes the dimension of the vector of network parameters.

For network parameters, we introduced the property `nn_params` in order to have the possibility to define a custom getter and setter for them, needed to fill the gap between the representations of network parameters in the `model` and in the algorithms.

The getter loops over the list `model.layers`, and on each layer calls the Tensorflow built-in function `get_weights()` to store in two separate lists weights and biases. Then, the two lists are merged into one, `thetas`, where weights and biases are stored alternatively. The same work is done backwards by the property setter.

Last, the `thetas` list is converted in a `Thetas` object, that is the data structure with which the algorithms work.

#### Main methods

The private method `build_NN()`, given a `seed`, initializes a fully connected Neural Network with a random normal initialization of weights, coherently with the prior information on weights from the Bayesian theory (more clarification in [subsection 2.3](#)) and zero initialization for biases, as it is common practice with NN.

The building blocks for the construction of the net are instances of classes inheriting from `tf.keras.layers`, the general Tensorflow blueprint for layers; in particular, we exploit the `Input` layer and the `Dense` layer for the hidden layers and the output layer.

This method has a public interface `initialize_NN()`, which simply calls it by passing the `seed` as input.

The public method `forward()` performs the forward pass: it takes a Numpy array input, converts it to a `Tensor`, and returns to the user the result of the `model` application on it.

CoreNN
+ <code>n_inputs</code> + <code>n_out_sol</code> + <code>n_out_par</code> + <code>n_layers</code> + <code>n_neurons</code> + <code>activation</code> + <code>stddev</code> + <code>model</code> + <code>dim_theta</code> + <code>nn_params</code>
- <code>build_NN()</code> - <code>compute_dim_theta()</code> + <code>--init--()</code> + <code>initialize_NN()</code> + <code>forward()</code>

<sup>9</sup>The operation is intended to be element-wise.

### 5.6.3. Physics enforcement - PhysNN

This class inherits from `CoreNN` and is designed to contain all the parts needed to tackle problem with physical information such as:

- store an instance of the selected equation class (not `Equation`, that is an ABC, but children). This object represents the entrance of physics into the problem, as it contains the method to compute the residual of the partial differential equation;
- store the flag for the inverse problem and  $\lambda$  parameters of PDE for future development as sketched in [section 7](#);
- store coefficients for normalization of the problem in a property;
- split the network output into solution and parametric field.

PhysNN
+ <code>pinn</code>
+ <code>inv_flag</code>
+ <code>u_coeff, f_coeff</code>
+ <code>norm_coeff</code>
+ <code>__init__()</code>
+ <code>tf_convert()</code>
+ <code>forward()</code>

### Constructor and properties

The constructor of this class requires two arguments: `par`, as in `CoreNN`, and `equation`, that is an object of the class corresponding to the test case. We use `super()` in this context to retrieve the constructor of the parent class `CoreNN`, to which we pass the required `par` argument.

In the constructor we also build an object for the test-case equation and store it in the attribute `pinn`, as well as storing the flag for the inverse problem and initialize the normalization coefficients to `None`.

The `norm_coeff` property represents the normalization coefficients of solution and parametric field.

### Main methods

This class contains the overriding of the parent's `forward` method. We make use again of the `super` function to retrieve the parent's `forward`; then, its output is split into solution and parametric field by separating the first and last  $n_{out\_sol}$  components, being  $n_{out\_sol}$  the dimension of the solution. This is also very useful for Regression, because it creates an empty parametric field and allow us to solve it like all the other cases.

### 5.6.4. Training functionalities - LossNN

This class inherits from `PhysNN` and is designed to:

- compute separately all the components of the loss function;
- combine the components according to the test-specific options;
- compute also other metrics not appearing in loss function
- compute the gradient of the loss with respect to the network trainable parameters.

### Constructor

The constructor relies upon the constructor of the parent class `PhysNN` with the same mechanism described for `PhysNN`.

The attribute `vars` is the dictionary of uncertainties, needed to compute the losses.

Then, it stores into the attributes `metrics` and `keys` a list of the components of the log-posterior that need to be included in loss computation respectively for history monitoring and learning.

Indeed, only the components declared in `keys` have an influence on the parameters' updates, while the ones in `metrics` do not enter in the learning process and are just plotted.

LossNN
+ <code>metric</code>
+ <code>keys</code>
+ <code>vars</code>
- <code>normal_loglikelihood()</code>
- <code>mse()</code>
- <code>mse_theta()</code>
- <code>loss_data()</code>
- <code>loss_data_u()</code>
- <code>loss_data_f()</code>
- <code>loss_data_b()</code>
- <code>loss_residual()</code>
- <code>loss_prior()</code>
- <code>compute_loss()</code>
+ <code>__init__()</code>
+ <code>metric_total()</code>
+ <code>loss_total()</code>
+ <code>grad_loss()</code>

### Loss components evaluation

Before the description of various methods is important to highlight that all the computation needed for the loss function and the gradients evaluation must be build keeping in mind the they have to be written in a way that they are automatic differentiable for the TensorFlow Gradient Tape (presented in [subsubsection 4.2.3](#)).

We implemented a private method, `mse()` to compute a MSE-like score suitable for the tensors with which we operate. As this method is called on the difference between the output and the target, it performs the

computation of the MSE between the input **Tensor** and the zero **Tensor** as ground truth (in fact, the desired target for the difference is zero).

This method is accompanied by the decorator `@staticmethod`, that makes the method bounded to the whole class rather than its instances. Since the computation of the MSE of a tensor is a general task detached from the class, in fact, it is reasonable to enable the user to call the method without the `self` argument.

The output of the `mse()` function is then exploited by `normal_loglikelihood()`, private static method which is the blueprint for the computation of the terms 28–31, having all the same structure.

Then, we implemented `loss_data()`, auxiliary private method for the fitting and boundary components of the loss. It consists in the computation of the MSE between target and output and the log-posterior component and it will be used as common blueprint for the cases in which we need to invoke `mse()` and `normal_loglikelihood()`. Having this method at hand, we can call it on the different set of data with the private methods `loss_data_u()`, `loss_data_f()`, `loss_data_b()` respectively on the solution, parametric field and boundary data.

On the other hand, `loss_residual()` contains the computation of the physical loss, which consists in retrieving from the `pinn` the residual of the partial differential equation. We perform the forward step inside a `tf.GradientTape`, letting the collocation dataset to be watched by the tape.

The prior component is managed by `loss_prior`, a private method which essentially performs the same task as `loss_data` (giving as output MSE and log-posterior component) for the prior distribution. Notice that in this case the std of the gaussian is not read from `par`, but is an attribute inherited from `CoreNN`.

## Main methods

We organize the storage of the components of the loss with dictionaries: this task is assigned to the private method `compute_loss()`, that returns a dictionary whose key-value pairs are the names of the components presented above and the corresponding MSE and loss values.

Method `loss_total` and `metric_total` both rely on evaluation from `compute_loss()` and sum all the components, respectively written in `keys` and `metrics` attributes. Furthermore the loss contains only the sum of log-likelihoods, instead the metrics store also the posterior value and each individual component.

Finally, the public method `grad_loss()` has the task of the computation of the gradient of the loss function with respect to the network trainable parameters. The operation is performed inside a `tf.GradientTape`, after registering the trainable parameters inside the tape.

### 5.6.5. Prediction phase - PredNN

This class inherits from `PhysNN` and is designed to:

- compute a sample with given parameters and inputs;
- store the samples of NN parameters;
- perform and display error computation;
- compute and display the statistics for UQ.

## Constructor

The constructor relies upon the constructor of the parent class `PhysNN` with the same mechanism described for `LossNN`. Then, it initializes an empty list `thetas` to store the samples of NN parameters.

## Main methods

The class has private methods to compute samples of the output: `compute_sample()` is a private method which, given one sample of NN parameters, performs one forward pass by relying on `PhysNN`'s `forward()`.

Then, it returns the de-normalized outputs for solution and parametric field, by performing the operation

$$\mathbf{u} = \mathbf{u}_{norm} \cdot \sigma_u + \mu_u$$

The `predict()` private method, then, computes a list of `n_thetas` output samples given a set of NN parameters' samples. First it checks whether the requested number of samples is available, then it iteratively calls

PredNN
+ thetas
- compute_sample()
- predict()
- statistics()
- compute_UQ()
- metric()
- compute_errors()
- disp_UQ()
- disp_err()
+ __init__()
+ mean_and_std()
+ draw_samples()
+ test_errors()
+ show_errors()



the `compute_sample` auxiliary method. It returns two separate lists, one for the solution and one for the parametric field, whose elements are `tf.Tensors` of shape `(n_sample, n_out_sol)` and `(n_sample, n_out_par)`, respectively.

This method is then exploited within the public method `draw_samples()`, that draws samples of the solution and of the parametric field given inputs, with a call to `predict`, then it converts them to the `numpy` type. The output is a dictionary, whose keys are solution and parametric field.

The `predict` method is also invoked by `mean_and_std()`, a public method that computes mean and standard deviation of the output samples. This function performs a customized forward step with `predict`, then it computes mean and standard deviation with the auxiliary method `statistics`. The metrics - which are arrays of length `n_samples`, representing mean/std at each coordinate - are then stored inside the dictionary `function_confidence`.

The private methods devoted to the error and UQ computations are `compute_errors` and `compute_UQ()`; for what concerns the errors, computed both on the solution and the parametric field, we make a comparison between the values of the functions at the validation points and the mean of the distribution predicted by the NN. The errors, computed in the discrete  $L^2$  norm (MSE), are then normalized, to see their percentage.

In the case of UQ, we return a dictionary containing scalar quantifications of the uncertainty of the reconstructed functions. The metrics stored are mean and max standard deviation on solution and parametric field.

The above private methods are then called by `test_errors()`, a public method that is devoted to the creation of a dictionary storing errors and UQ. For what concerns the communication of the errors to the user, the private auxiliary methods `disp_UQ()` and `disp_err()` display respectively UQ and errors on the terminal. Then, the public method `show_errors()` prints errors and UQ to the terminal, both for solution and parametric field.

#### 5.6.6. Final wrapper - BayesNN

This class inherits both from `LossNN` and `PredNN`.

It is the one from which we instantiate an object in the `main.py` script, and is designed to:

- contain all history of training in two dictionaries;
- append new loss value to the history;
- be passed to an children class of `Algorithm` as will be explained in subsection 5.7 to be trained.

BayesNN
+ seed
+ history
+ constructors
- initialize_losses()
+ __init__()
+ loss_step()

### Constructor

The constructor of this class requires two arguments: `par`, an instance from the class `Param` described in subsection 5.2.3 and `equation`, an object among the ones described in subsection 5.5.

This class has two parent classes (`LossNN` and `PredNN`), and therefore falls into the framework of *multiple inheritance* as in the scenario described in Figure 7. With the `super()` method, we access the parents' constructors, to whom we pass the arguments `par` and `equation`.

Those arguments will be then passed to the classes higher in the hierarchy chain (`PhysNN` and then `CoreNN`); these classes will then exploit in their own constructors only the selected information already described and build their own attributes following the encapsulation principle.

The constructor arguments are then stored in the attribute `constructors`, in order to ensure an easy generation of copies of a `BayesNN`-derived object (as will be particles for the SVGD algorithm in subsection 5.7.6).

### Main methods

The main aim of the class is providing a wrapper of the other structures, to enable in the executable script the instantiation of an object from a comprehensive class.

Indeed, it is passed as argument to the algorithms' classes, where we will store in an attribute the `BayesNN` instance that will be trained with the data provided by the algorithm.

Nevertheless, this class contains also two methods: `initialize_losses()`, private method that initializes empty dictionaries both for MSE and log-likelihood, and `loss_step()`, public method that appends a new loss to loss history.

## 5.7. Optimizers

In this section, we present the relevant implementations for training, on which we make use of the OOP feature of abstract classes (see [subsection 4.3](#)).

As for equations and datasets, indeed, we want to define a blueprint that will be shared by an ensemble of sibling classes: apart from their specific dynamics, indeed, there are tasks that all training algorithms, either deterministic or probabilistic, have to perform.

Therefore, it is reasonable to create an abstract base class (**Algorithm**, in our case) where shared tasks are fixed; all its virtual methods are then overridden, to make effective the different behaviors across algorithms.

Moreover, we implemented the class **Trainer**, conceived as a training manager and designed to schedule trainings which can include a pre-training phase as well and handle training utilities.

### 5.7.1. Training interface

Objects belonging to this class have the task of selecting different algorithms and mounting one training algorithm after another. Its constructor asks for the **bayes\_nn** object, the parameters (needed for the training options) and the dataset to pass to the various algorithms.

#### Main methods

The private method **switch\_algorithm()** returns an instance of the class corresponding to the selected method, and raises an exception if the requested algorithm has not been implemented.

Then, the private method **algorithm()** uses the above method for algorithm selection and builds then the **algorithm** object. Finally, it assigns the training dataset through its property **data\_train**.

The public interface of the class consists in the methods **pre\_train()** and **train()**, which call **algorithm()** for the requested (pre-)training and perform model training. In the case of pre-training, at the end of the algorithm the NN parameters are set to the ones obtained at the end of the preliminary phase.

Trainer
+ debug_flag + model + params + dataset
- init() - switch_algorithm() - algorithm() + pre_train() + train()

### 5.7.2. Algorithm abstract class

This is the class devoted to training algorithms, which, in our design, acts on the **BayesNN** instance that is passed as argument to the algorithm object.

The **Algorithm** class is an abstract base class, which acts as a blueprint for all the specific training algorithms. Indeed, they all share some tasks that are common, such as the update of the history of NN parameters and the call to the parameters' update in  $N_{epochs}$  subsequent iterations, that constitute the training main pipeline.

The methods referring to the common training workflow are not overridden in general - apart from minor changes such as the printing of some algorithm-specific logs during training.

On the other hand, the method of sampling and selection of the new NN parameters are abstract methods, because they need to be overridden in the various algorithms.

#### Constructor

The constructor of the algorithm object requires the **BayesNN** object, the method-specific parameters and the debug flag.

Moreover, we save the initial time to compute the training length, and initialize a variable to keep track of the current epoch.

The training dataset is managed by the property **data\_train**; in its setter, we do not only set the data, but also the normalization coefficients for solution and parametric field.

Algorithm
+ t0 + model + params + epochs + debug_flag + curr_ep + data_train()
- init() - compute_time() - train_step() - train_loop() + train() + train_log() + update_history() + sample_theta() + select_thetas()

## Main methods

The common training pipeline consists in performing a training loop with successive repetitions of a training step: this is implemented into the public method `train()`, which normalizes the dataset, organizes the examples in batches, manages and performs the epochs in a loop. At its end, it denormalizes the dataset and updates the NN parameters with the sampled ones.

The two abstract methods that need to be overridden in children classes are `sample_theta()`, for sampling a single new theta, and `select_thetas()`, for retrieving the list of sampled thetas).

The method `train()` relies on two private methods for the training loop: `train_loop()`, which builds the epochs loop and introduces a `tqdm` progress bar to display on terminal training progress, and `train_step()`, which, through a `match case` statement, calls the selected `sample_theta()` method and performs one step of update of NN parameters. It also updates the learning history by appending the freshly sampled parameters.

The method in charge of the evolution of NN parameters through training is `update_history()`; this public method updates the NN parameters with the new ones, retrieves the computations of MSE and log-likelihood values and updates the loss history.

Among the utilities, we mention `compute_time()`, which computes training time and prints it in a formatted way to terminal, and `train_log()`, that reports the end of training and calls the former method.

### 5.7.3. Adam optimizer

This class implements the Adam training algorithm (algorithm 3), deriving it from the abstract base class `Algorithm`. Although Adam optimizer can be found within the `tf.keras` API, we implemented our own version and we able to easily mount it on the same infrastructure designed for Bayesian training algorithms.

The constructor reads Adam's parameters from the `param_method` dictionary, and stores them into class attributes. Then, it initializes the momentum vectors with the private auxiliary function `initialize_momentum`, which initializes with zeros variables for the momentum vectors `m` and `v` with the desired length.

Then, we overrode the method `sample_theta()` implementing in it the Adam update rule presented in algorithm 3 for sampling one parameter vector given its previous value. Note that in this case the purpose of the method is performing one round of parameters' update, rather than sampling from a distribution as will be the case for Bayesian algorithms.

The other overridden method is `select_thetas()`: since the Adam method is deterministic, it improved the parameters' reconstruction iteration after iteration. Therefore, the final prediction will be the finest parameter vector, that is the last element of the parameters' history. This is the only sample of  $\theta$  that is output by Adam, which constitutes its only prediction; clearly, with just one prediction, it is not possible to quantify uncertainty, as expected with a non-Bayesian algorithm.

### 5.7.4. Hamiltonian Monte Carlo

This class implements the HMC training algorithm (algorithm 4), based on the abstract base class `Algorithm`. The constructor reads HMC's parameters from the `param_method` dictionary, and stores them into class attributes after having checked the constraint that the `burn-in` is smaller than the number of epochs.

This class contains separate private methods for the different stages of algorithm 4: the evaluation of the Hamiltonian function (`hamiltonian()`), the leap-frog step (`leapfrog_step()`), the computation of the acceptance probability (`compute_alpha()`) and the acceptance-rejection step (`accept_reject()`).

We remark that `compute_alpha()` either computes the acceptance probability  $\alpha$  and samples  $p$ ; we then compare the logarithms of both quantities, that ensures to work with smaller quantities.

The `accept_reject()` method contains utilities for debugging as well, because it provides as well a computation of the overall acceptance rate, and if the debug flag is activated, it enables the monitoring of the evolution of the hamiltonian values from the terminal, which helped a lot in fine-tuning HMC.

Again, we overrode the abstract methods of the parent class: `sample_thetas()`, that samples one parameter vector given its previous value according to HMC's dynamics by calling `leapfrog_step()` and `accept_reject()`.

In this case, `select_thetas()` returns the final set of samples of NN parameters taking into account burn-in and stride. The meaning of this set of  $\theta$ s is a collection of samples from the posterior distribution; in principle, the full set of samples could be used for prediction and UQ.

However, to improve the efficacy, we discard the initial samples in which there is actually a gradual improvement and we discard close samples to reduce internal correlation, that would weaken UQ.

### 5.7.5. Variational Inference

This class implements the VI training algorithm (algorithm 5), deriving it from the abstract base class `Algorithm`. Its constructor first initializes two attributes for the vectors  $\zeta_\mu$  and  $\zeta_\rho$  describing the parametric family of distributions, by calling the private method `initialize_VI_params()`.

To sample NN parameters, we first have to learn  $\zeta_\mu$  and  $\zeta_\rho$ , as presented in subsection 3.2, and we do that through GD. This task requires the computation of the gradient of the loss  $f(\theta, \zeta) := \log(Q(\theta; \zeta)) - \log(P(\theta)P(D|\theta))$  with respect to  $\zeta$ ; to make this more feasible from computational and implementation point of view, we developed the following analytical expressions for the gradient with respect of a  $\zeta_i^\mu$  and of a  $\zeta_i^\rho$ :

$$\begin{aligned}\Delta_i^\mu &= \frac{\partial f(\theta, \zeta)}{\partial \theta} + \frac{\partial f(\theta, \zeta)}{\partial \zeta_i^\mu} \quad i = 1, \dots, d_\theta \\ \Delta_i^\rho &= \frac{\partial f(\theta, \zeta)}{\partial \theta} \frac{\varepsilon}{1 + e^{-\zeta_i^\rho}} + \frac{\partial f(\theta, \zeta)}{\partial \zeta_i^\rho} \quad i = 1, \dots, d_\theta\end{aligned}$$

Inserting the definition of  $f$  into the above computations, we obtain that:

$$\begin{aligned}\Delta_i^\mu &= \frac{\partial \log(Q(\theta; \zeta))}{\partial \theta} - \frac{\partial \log(P(\theta)P(D|\theta))}{\partial \theta} + \frac{\partial \log(Q(\theta; \zeta))}{\partial \zeta_i^\mu} \quad i = 1, \dots, d_\theta \\ \Delta_i^\rho &= \frac{\varepsilon}{1 + e^{-\zeta_i^\rho}} \left[ \frac{\partial \log(Q(\theta; \zeta))}{\partial \theta} - \frac{\partial \log(P(\theta)P(D|\theta))}{\partial \theta} \right] + \frac{\partial \log(Q(\theta; \zeta))}{\partial \zeta_i^\rho} \quad i = 1, \dots, d_\theta\end{aligned}$$

At this stage, we are able to retrieve the  $\partial_\theta \log(P(\theta)P(D|\theta))$  already met term (that will be retrieved from the `grad_loss()` function of `LossNN`), and the rest of the terms can be easily evaluated as we can explicit them thanks to the known expression of the multivariate gaussian distribution  $Q(\theta; \zeta)$ . In particular, we have:

$$\begin{aligned}\frac{\partial \log(Q(\theta; \zeta))}{\partial \theta} &= -\frac{\partial \log(Q(\theta; \zeta))}{\partial \zeta_i^\mu} = \frac{(\zeta_i^\mu - \theta)}{\zeta_i^{\sigma^2}} \\ \frac{\partial \log(Q(\theta; \zeta))}{\partial \zeta_i^\rho} &= \frac{(\zeta_i^\mu - \theta)^2}{\zeta_i^\sigma} \left( \frac{1}{\zeta_i^{\sigma^2}} - 1 \right)\end{aligned}$$

considering that  $\zeta_i^\sigma = \log(1 + e^{\zeta_i^\rho})$ .

Thanks to the analytical expression of derivatives we notice that  $\Delta_i^\mu$  has simple form when we are using gaussian distributions. Indeed, the first and third term in its definition are exactly opposite, so they annihilate each other and  $\Delta_i^\mu$  is only defined with by classical log-likelihood.

In the implementation, `compute_grad_rho()` returns a vector whose components are  $\Delta_i^\rho$  by performing the above computations, then the private method `update_VI_params()` performs the GD updates on  $\zeta_\mu$  and  $\zeta_\rho$ .

The proposed overriding of `sample_theta()` enables to mount the VI training on the already built infrastructure: its fundamental task is the call to `update_VI_params()`, which is therefore execute as many times as the number of epochs, being inserted into the loop.

The rest of its implementation is only aimed at updating NN parameters to be consistent with the blueprint, but, differently from methods such as HMC, the production of the final samples will be postponed to `select_thetas()`. In its overriding, we indeed produce the set of samples of NN parameters by transforming a multivariate standard gaussian into a gaussian with mean  $\zeta_\mu$  and standard deviation  $\log(1 + e^{\zeta_\rho})$ .

### 5.7.6. Stein Variational Gradient Descent

This class implements the SVGD training algorithm (algorithm 6), deriving it from the abstract base class `Algorithm`. Apart from the algorithm mechanism, this method required the implementation of a class for the particles, because, in addition to the functionalities that objects of the class `BayesNN` have, we need for the SVGD mechanism a practical way to update the particles' parameters by broadcasting the quantity  $\phi$ .

#### Particles

We implemented a specific class for the multiple NN that are involved in the SVGD training mechanism; the class `Particle` inherits from `BayesNN`, and the only additional attribute that we store is the learning rate, peculiar for networks employed in this method.

The class is quite small, as it has only two public methods: `set_norm_coeff()`, that is needed to share with the particles the normalization coefficients (else the residual loss could not be computed by the particles) and `update_theta()`, that adds to the current particle's parameters a given quantity, multiplied by learning rate.

## Algorithm class

The algorithm constructor reads SVGD's parameters from the `param_method` dictionary, and stores them into class attributes. Then, it builds the set of particles used during training by calling the private method `build_particles()`, which is in charge of the initialization of the particles, that are then stored in a list which is attribute of the SVGD class. Particles are represented by objects of the aforementioned class `Particle`, and they are initialized each with a different seed.

In this algorithm, we implemented a syntax inspired by parallel programming for the communication of the computations that have to be made by each particle, and, as further development, could be performed in parallel. The private methods `gather_thetas()` and `gather_grads()` are indeed in charge of information collection from the particles: the algorithm collects indeed respectively particles' parameters and gradients of loss functions with respect to particles' parameters and stores them. On the other hand, the method `scatter()` sends an information from the algorithm back to the particles, by broadcasting the correction  $\phi$  to be made to each particle's parameters.

The parameters' updates require the RBF kernel and its gradient: we therefore implemented the private method `kernel()`, which returns a matrix  $K$  with the RBF kernel (Equation 39) evaluated on each pair of particles' parameters:

$$K_{ij} = k(\theta_i, \theta_j)$$

The other output is the matrix  $GK$ , computed from  $K$  with a matrix multiplication as in Equation 40:

$$GK_{ij} = \nabla_{\theta_i} k(\theta_i, \theta_j)$$

Again, we needed to override `sample_theta()`, which retrieves parameters and gradients from the particles and then performs the computations in algorithm 6 with matrix-vector products. In order to perform efficiently this multiplication, we relied on `np.matmul` and therefore we converted the vectors and matrices to the `np.array` data type. The list of thetas sampled produced by `select_thetas()` consists instead of the collection of the last SVGD epoch of all the particles' parameters.

## 5.8. Postprocessing

In this section, we present the relevant implementations for postprocessing, focussing on data storage and visualization. In the `main` script, the strategy adopted consists in first storing the overall outcome of the test case into a specific folder. Then, we produce plots within the same folder with methods from the `plotter` object by reading the quantities already saved during storage.

### 5.8.1. Storage

The `Storage` class is the class devoted to managing the details and outcome of a simulation. Objects from it can be instantiated with two purposes: either to save the results in the folder created for the test case, either to load them and exploit them for plotting.

The class has several properties representing information on the model outcome:

- `data`, for the coordinates of the validation points and corresponding solution and parametric field values;
- `history`, for the evolution of the MSE and log-likelihood during training;
- `thetas`, for the sampled network parameters;
- `nn_samples`, for the solution and parametric field values reconstructed with the samples  $\theta$ s;
- `confidence`, for the mean and standard deviation of the network output.

They are all set from the `main` script after the simulation, with a custom setter that saves the numerical values of the above quantities in the files described in `outs`. On the other hand, their getter has a dual structure, because it reads the values from the same files.

The strength of the usage of properties, in this case, consists in the possibility of definition of a custom way of outcome storage by keeping an intuitive and clean syntax in the main executable script. In addition, if we need to make any change to the storage details (for example, in terms of file format), we could just redefine the property without changing the external interface.

Moreover, the class has two public methods: `save_parameter` and `save_errors`, that store in `.txt` files respectively the information on the simulation details and errors and UQ of the reconstructed functions. In both of them, we make use of private auxiliary methods for a readable formatting of the file content.

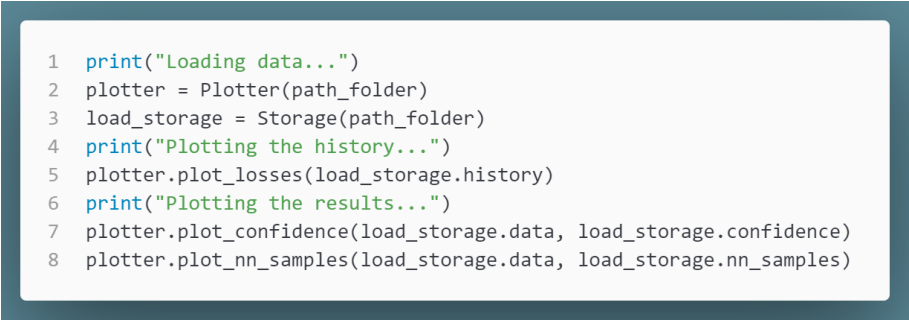
### 5.8.2. Plotter

The class **Plotter** is designed to save and show to the user a visual comparison between the network output and the ground truth.

The class constructor asks for the path to the folder where the test outcome has been stored, that will be the destination of the plots as well. Here, from the parameters' summary produced by **Storage** it can detect if we need to produce plots only for one function, such as in the oscillator case, or for solution and parametric field.

All the methods in the class require as mandatory argument the data to be plotted: indeed, in our design the plotter class works independently from the main already executed and jointly with a **Storage** object which acts as loader of the files in the test case folder.

In [Figure 18](#), you can see how the **Plotter** object is used in the main script, but the same in-pair mechanism of the plotter and the loading storage is exploited also in the `main_loader.py`, an example of separate script which does not train any model but only re-generates plots referred to previously stored test cases.



```
1 print("Loading data...")
2 plotter = Plotter(path_folder)
3 load_storage = Storage(path_folder)
4 print("Plotting the history...")
5 plotter.plot_losses(load_storage.history)
6 print("Plotting the results...")
7 plotter.plot_confidence(load_storage.data, load_storage.confidence)
8 plotter.plot_nn_samples(load_storage.data, load_storage.nn_samples)
```

Figure 18: Call to plotter and loading storage in `main.py`

The public methods implemented in the class enable to plot:

- the MSE and loss history during training;
- the ensemble of network outputs corresponding to the samples of network parameters;
- the mean and standard deviation of the reconstructed function distribution.

For the case of the 2D plot, we provided only a visual comparison between the average network prediction and the ground truth with a heatmap (see the devoted test case in [subsection 6.3](#)), while the UQ information about the variance is only printed to terminal and stored in the log file.



## 6. Results

This chapter is devoted to the presentation of the results obtained in different applications, ranging from the regression task to differential problems such as the Laplace and Damped Harmonic Oscillator models.

On these test cases, we made experiments aimed at showing the main characteristics of the models that our library includes, especially comparing training algorithms among each other (either deterministic and bayesians) and physics-informed versus non-physics-informed learning. We also made trials of solving the same problem with different datasets, in order to estimate possible issues linked to the presence of more complex functions.

The organization of the sections reflects different aspects captured by our implementation: indeed, we focus separately on a comparison among training algorithms on the same task (subsection 6.1), on the implications of physics-informed learning (subsection 6.2) and on code portability to higher dimension (subsection 6.3).

The last section, subsection 6.4, presents some results obtained after an intense tuning of the parameters.

In this phase of result production, indeed, the tuning of parameters had a big input: as stated in subsection 1.1, the focus of the project was the library feature expansion, but we showed as well the possibility to proceed with the optimization of the results in one specific application.

An intensive tuning effort was made with the B-PINN trained to solve the Laplace problem with the HMC training (presented in subsection 6.4): this represented a challenging task, because HMC is the method having the most careful parameters' choice as discussed in subsection 3.1.3, but on the other hand it enabled to perform tuning with test cases having a feasible computational time, also thanks to the presence of debug utilities such as the continuous monitoring of the acceptance rate during training.

This needs to be taken into account when reading the following showroom of results, in order to avoid to make hasty comparisons among the methods just basing on the quality of the prediction in terms of errors and UQ.

At the beginning of each test case, we reported the options set to obtain the results shown (that could be retrieved even from the logs in the corresponding `outs` subfolder).

### 6.1. Regression Problem

This section is devoted to the presentation on the results obtained for the function interpolation task. This case of application falls outside the physics-informed framework, because the model does not learn a PDE during training, but it represented a suitable starting point, both because in this way we got results to make comparisons on the contribution of the physics and because, from the tuning point of view, it represented a faster way to identify a good set of algorithm parameters which enabled to obtain good-quality results.

Moreover, as stated in subsection 4.4.1, the regression can be seen as preparatory to the physics-informed case also because we do not let the PDE information enter immediately into the loss used for training, but start including it after having reached a suitable approximation of the measurement data with precisely some regression-only epochs.

In this section, we show for all the methods implemented the reconstruction of the function  $u(x) = \cos(8x)$  in the domain  $\Omega = [0, 1]$ , having  $N_{fit} = 16$  equidistributed measurements available which are affected by a noise with level  $\sigma_u = 0.1$ .

#### 6.1.1. Adam

We first present the results obtained with the Adam method; this training choice is deterministic, hence it cannot output any UQ estimate on the prediction, but it enabled accurate results in an extremely short time, as witnessed by Figure 19.

Thanks to its light computational overload, the Adam algorithm turned out to be a handy initializer for letting the bayesians algorithms start from not completely random initial values.

The number of epoch was set to  $N = 5000$  and the training lasted  $\sim 2$  minutes; the Adam-specific parameters were fixed (actually, in all the Adam test cases of the project) to the good default values suggested in [5], that are:

Parameter	Value
$\beta_1$	0.9
$\beta_2$	0.999
$\epsilon$	$10^{-8}$
$\eta$	$10^{-3}$



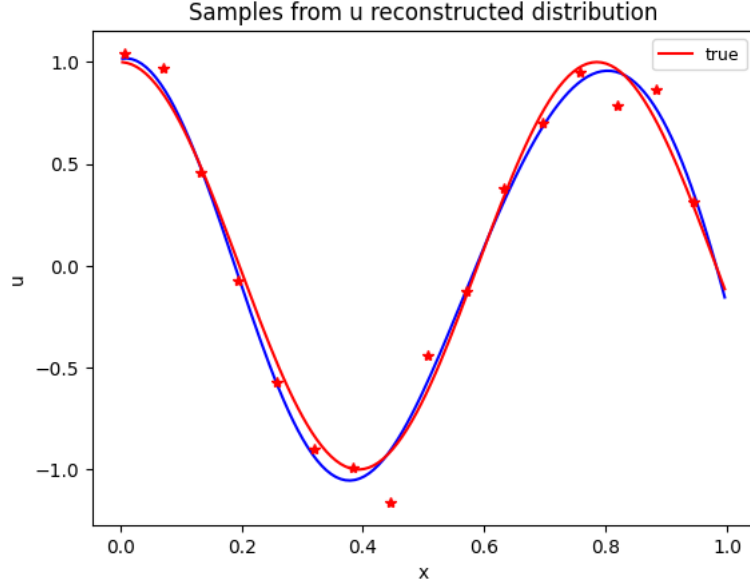


Figure 19: Function regression with Adam on  $u(x) = \cos(8x)$

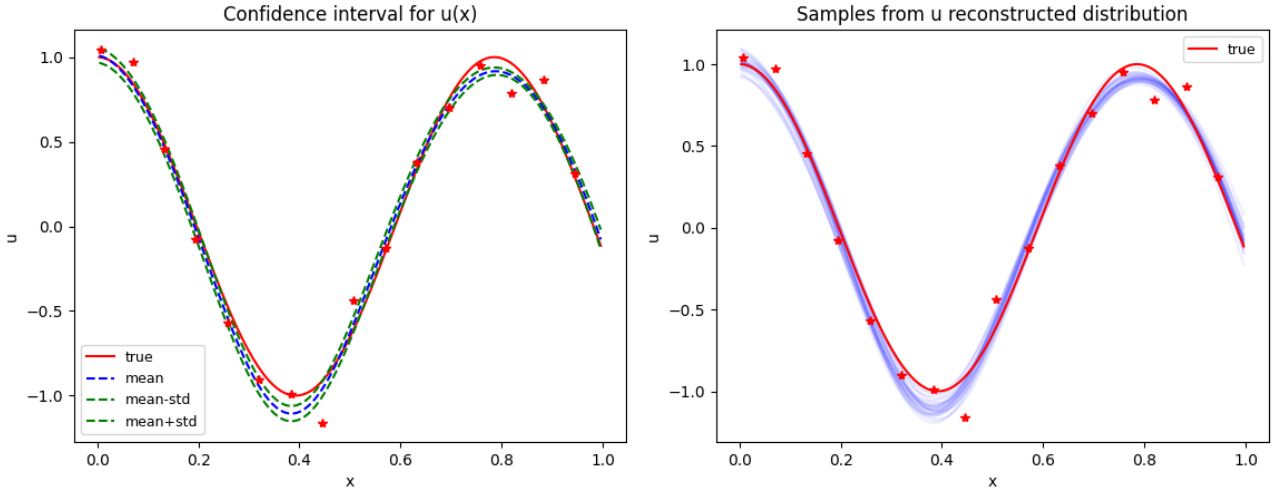
### 6.1.2. HMC

We then reconstructed the function with the first Bayesian model, with the HMC training method, imposing these options.

This parameter set was obtained through fine-tuning, which was not a trivial task given the amount of options and the co-implications behind them. This challenge became even more difficult when dealing with physics-informed tasks, as we will present in [subsection 6.4](#).

On the other hand, HMC has a computational demand of medium level, if compared with the other algorithms implemented, and for this reason tuning to reach function reconstruction below the noise level was possible.

Parameter	Value
$N$	200
$B$	50
$S$	5
$L$	20
$dt$	$10^{-3}$
$\eta$	1



(a) Prediction with confidence interval.

(b) Ensemble of all the samples.

Figure 20: Function regression with HMC on  $u(x) = \cos(8x)$

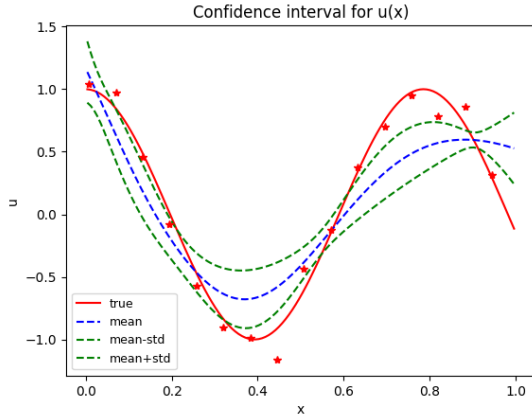
### 6.1.3. SVGD

For the SVGD case, the method-specific options were those one. They enabled to obtain reasonable results, but the high computational demand of the algorithm represented the main obstacle for parameter tuning.

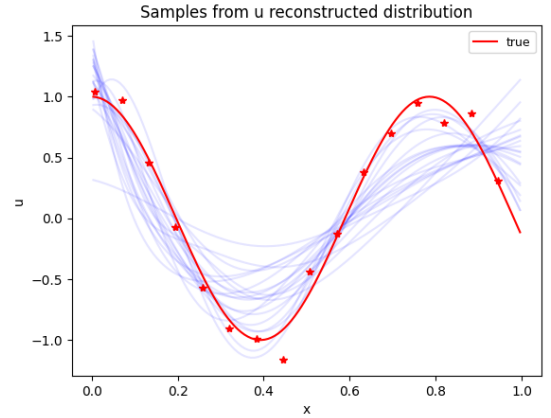
What affects significantly the computational overload requested by this algorithm is the number of particles  $N$ , and the limitation on their number represents a weakened factor for the quality of UQ. On the other hand, the heuristic recipe to improve the prediction is this time simpler than with HMC, because it is necessary to increase number of epochs  $E$  and of particles  $N$ .

To reduce the computational overload, a possibility that could be exploited in further extension of the project is the parallelization of the algorithm, that in the current implementation has already a syntax inspired to parallel programming.

Parameter	Value
$E$	2000
$B$	100
$N$	20
$h$	1000
$\varepsilon$	$10^{-5}$



(a) Prediction with confidence interval.



(b) Ensemble of all the samples.

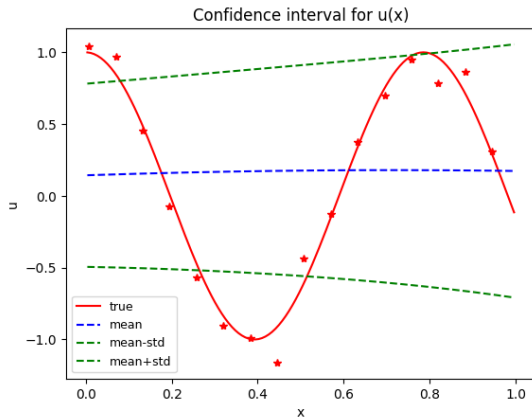
Figure 21: Function regression with SVGD on  $u(x) = \cos(8x)$

### 6.1.4. VI

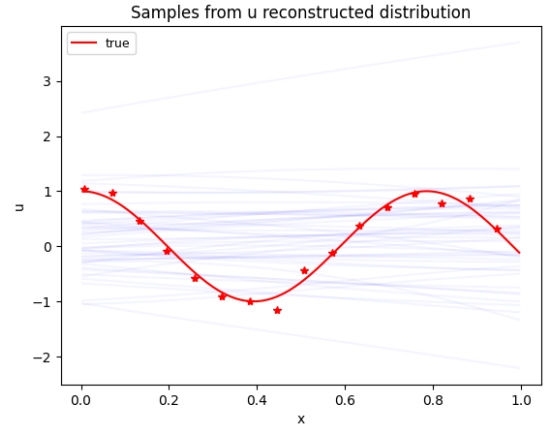
This algorithm was the less time-demanding for training among the Bayesian choices, and, as remarked in [subsection 3.2](#), it enables to get cheaply as many function samples as desired, improving therefore the UQ.

As a downside, we obtained a poor performance in this case as it happened to VI cases in the reference [11]. This is due to the underlying absence of correlation in the distribution of the reconstructed network parameters, due to the choice of the multivariate gaussian with diagonal covariance matrix as proposed in [2].

Parameter	Value
$N$	2000
$M$	50
$\alpha$	$10^{-5}$



(a) Prediction with confidence interval.



(b) Ensemble of all the samples.

Figure 22: Function regression with VI on  $u(x) = \cos(8x)$

## 6.2. Damped Harmonic Oscillator Problem

In this section, we propose two models to tackle the differential problem of the Damped Harmonic Oscillator, presented in subsection 5.5. The aim of this application is mainly to highlight the model improvement that arises from the introduction of the physical information, which was completely absent in subsection 6.1.

This contribution becomes indeed evident when available data are not equidistributed, but restricted to some domain regions, as it can happen when trying to make a prediction for the evolution of a quantity over time, having at hand only data referred to the past time window (as done in [7] for COVID-19 previsions).

In this section, we present a comparison between a NN and a PINN that rely on the Adam deterministic training. However, since especially in the forecasting setting it makes sense to produce confidence intervals for the prediction, in subsection 6.4 we will present a tuned B-PINN for this problem.

The Damped Harmonic Oscillator under analysis has  $\delta = 2$  and  $\omega = 8$ ; moreover, we suppose to have  $N_{fit} = 8$  noisy measurements ( $\sigma_u = 0.01$ ), situated in the time subinterval  $[0, 0.7]$ , which mimic the initial evolution of the quantity under analysis.

### 6.2.1. NN vs PINN

In Figure 23a, we show the reconstruction of the solution provided by a NN that relies only on measurements: in the time subinterval  $(0.7, 2]$ , the profile is totally missed, and retrieved only at the final time thanks to the imposition of the boundary conditions.

We therefore trained a PINN to reconstruct the profile of the oscillations thanks to the presence of the physical law within the loss function. The introduction of this information needed to be carefully added because of the magnitude of the pde residual component of the loss.

Therefore, we balanced it with the choice of a higher uncertainty on the PDE, setting  $\sigma_r = 10$ .

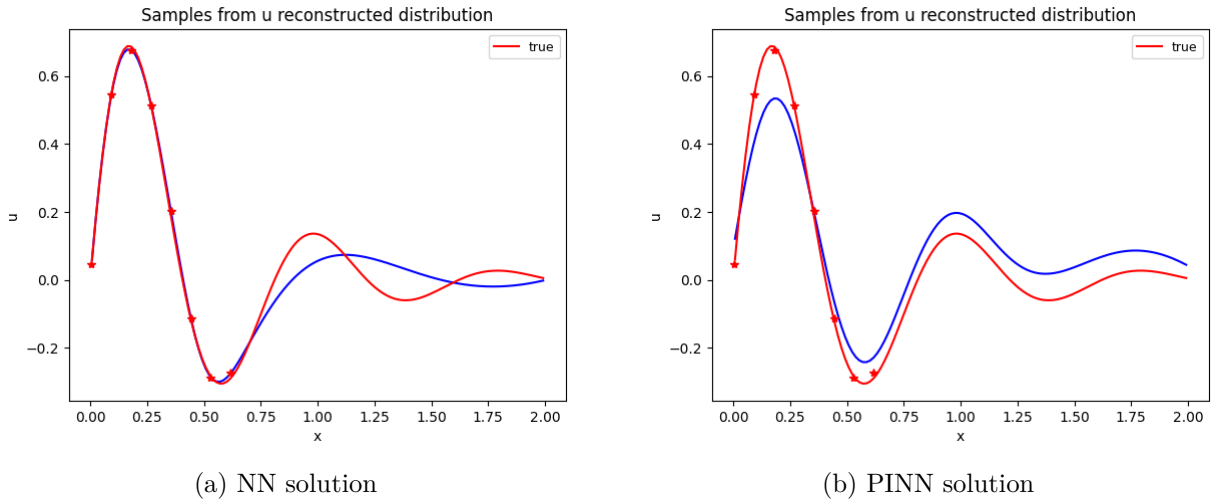


Figure 23: Damped Harmonic Oscillator Problem

## 6.3. Multi-dimensional domain

The proposed implementation is able to consider even multi-dimensional cases, because all the operations (even the differential ones) are coded in such a way that they can handle non-scalar quantities.

While the code executed is exactly the same for any arbitrary dimension up to the error and UQ, the only difference consists in the visualization option, for which we chose heatmaps for the 2D case.

### 6.3.1. Adam 2D cosine

Figure 24 and Figure 25 show a comparison between exact and mean predicted PINN output when solving the 2D Laplace problem on the unit square  $[0, 1] \times [0, 1]$ .

The functions under analysis are:

$$u(x, y) = \cos(8x) + \cos(8y) \quad f(x, y) = 64 \cos(8x) + 64 \cos(8y)$$

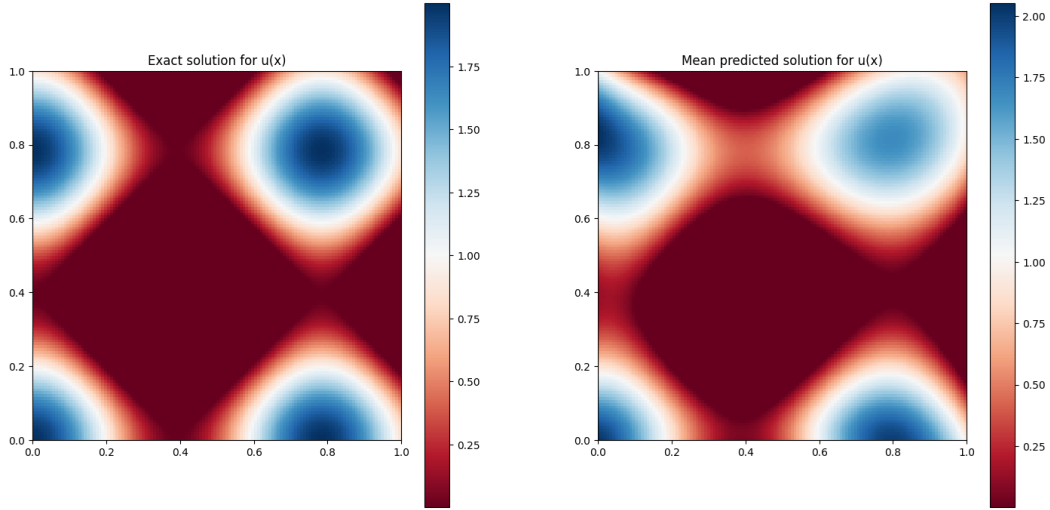


Figure 24: Reconstruction of  $u(x, y)$  with an Adam-trained PINN

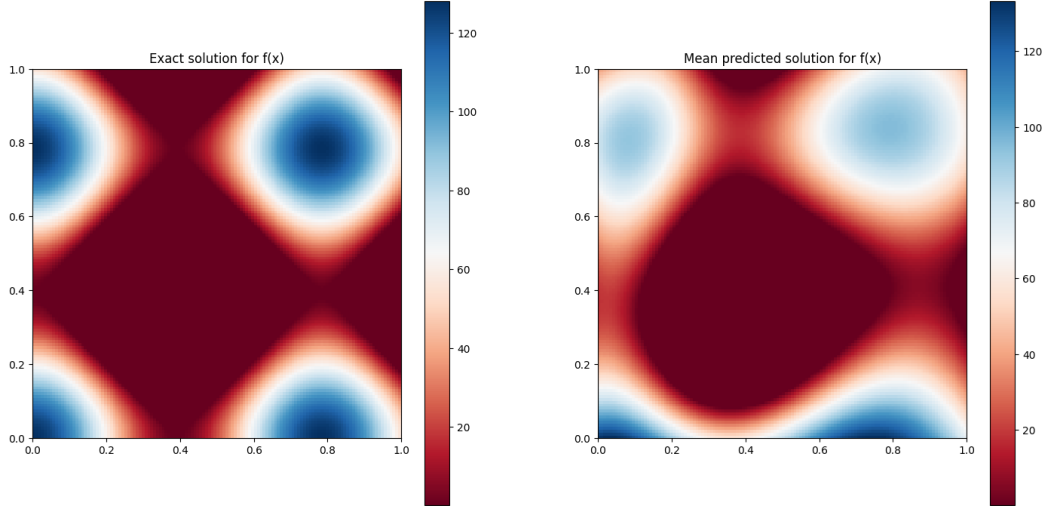


Figure 25: Reconstruction of  $f(x, y)$  with an Adam-trained PINN

## 6.4. HMC Showcase

In this section, we deepen into some results obtained with the HMC training algorithm, either on function interpolation and on differential problems involving the Poisson/Laplace and Damped Harmonic Oscillator equations.

We concentrated the tuning effort on this algorithm because of the interesting parameters co-implications in their choice, and also because the medium computational time made it possible to run several trainings on our CPUs.

### 6.4.1. Regression Problem - sine

In this experiment, we implemented a BNN for the reconstruction of the function  $u(x) = \sin^3(6x)$  in the domain  $[-1, 1]$ , relying on  $N_{fit} = 16$  noisy measurements ( $\sigma = 0.1$ ).

This test case shows a function with a more tangled graph with respect to the cosine example proposed in subsection 6.1, whose details we will try to teach to the network in a physics-informed framework as the one proposed at the end of this section.

We generated our data localizing them in two subdomains with the feature described in subsection 5.3, assuming to have measurements available only in the set  $[-0.75, -0.25] \cup [0.25, 0.75]$ .

When analyzing the results in Figure 26, apart from remarking the evidence of the missing information in the region  $[-0.25, 0.25]$  that will be overcome with the introduction of the PDE residual in the loss, we focus on what happens to UQ in the region where data are not available.

Indeed, we can see that the confidence interval is reasonably broader, because in the areas where the constraint of data is not available, the network is more flexible and leaves the door open to bigger uncertainty in the prediction.

Parameter	Value
$N$	500
$B$	100
$S$	10
$L$	20
$dt$	$10^{-3}$
$\eta$	1

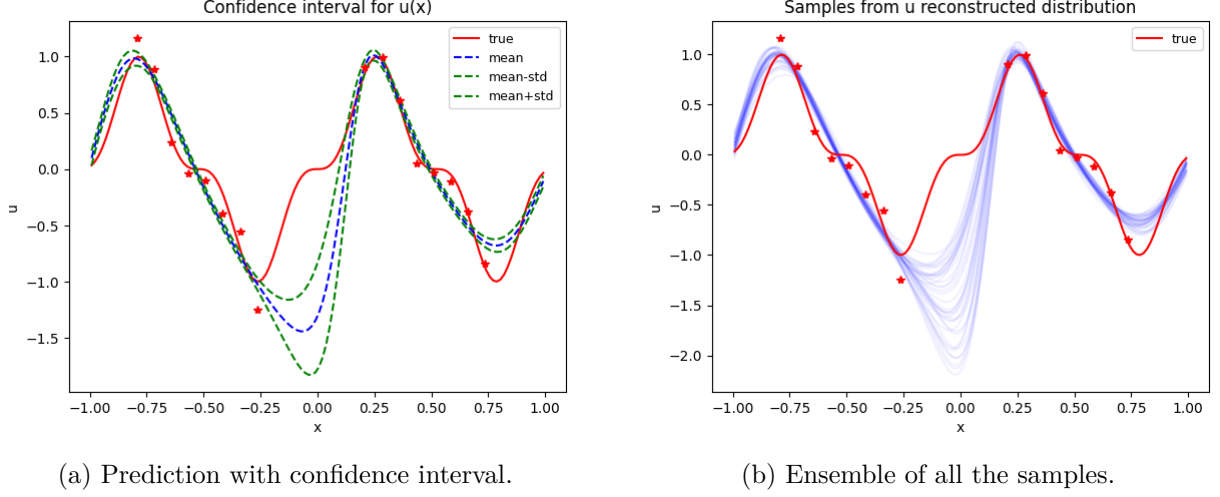


Figure 26: Function regression with HMC on  $u(x) = \sin^3(6x)$

#### 6.4.2. Oscillator Problem

This experiment shows the HMC counterpart of the same oscillator problem presented in subsection 6.2, by adding a third element of comparison: after NN and PINN, we here propose a BPINN.

While on the physics-informed side the PINN and the BPINN are exactly in parity, what the BPINN adds is a confidence interval for the prediction.

To reproduce the test case in Figure 27, we ran  $N = 3000$  Adam epochs, then let HMC start the options aside. For the uncertainty on the PDE residual, we employed the same noise level of the PINN  $\sigma_r = 10$ .

Parameter	Value
$N$	100
$B$	0
$S$	10
$L$	200
$dt$	$5 \times 10^{-5}$
$\eta$	0.5

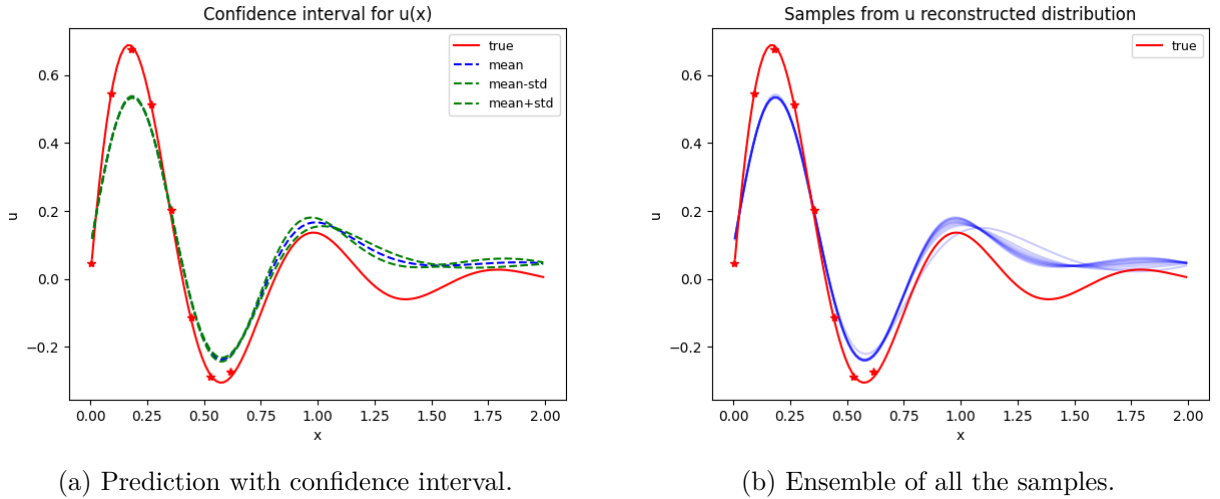


Figure 27: Damped Oscillator Problem with HMC

### 6.4.3. Poisson/Laplace Problem - cosine

In these last two sections, we propose the solution of the Laplace 1D problem with two different datasets. The first one we present involves two cosine functions in the domain  $[0, 1]$ :

$$u(x) = \cos(8x) \quad f(x) = 64 \cos(8x)$$

For what concerns the availability of measurements, we assume to have  $N_{fit} = 16$  noisy data ( $\sigma_f = 0.1$ ) on the forcing term  $f$  and only the boundary conditions for the solution  $u$ . Indeed, we want to reconstruct the solution using the PDE without relying on any measurements - apart from boundary conditions.

In this case, the magnitude of the physical loss was comparable to the one of the fitting loss. Therefore, we set its uncertainty to the same value of the noise level, resulting in  $\sigma_r = 0.1$ .

In Figure 28 and Figure 29 we show the reconstruction of the functions involved in the problem: coherently with the intuition, confidence intervals are wider for the solution, for which the constraint of measurements lacks, and samples of the solution create an envelope around the ground truth, while samples of the forcing term are closer to each other.

Parameter	Value
$N$	1000
$B$	70
$S$	10
$L$	20
$dt$	$10^{-3}$
$\eta$	0.5

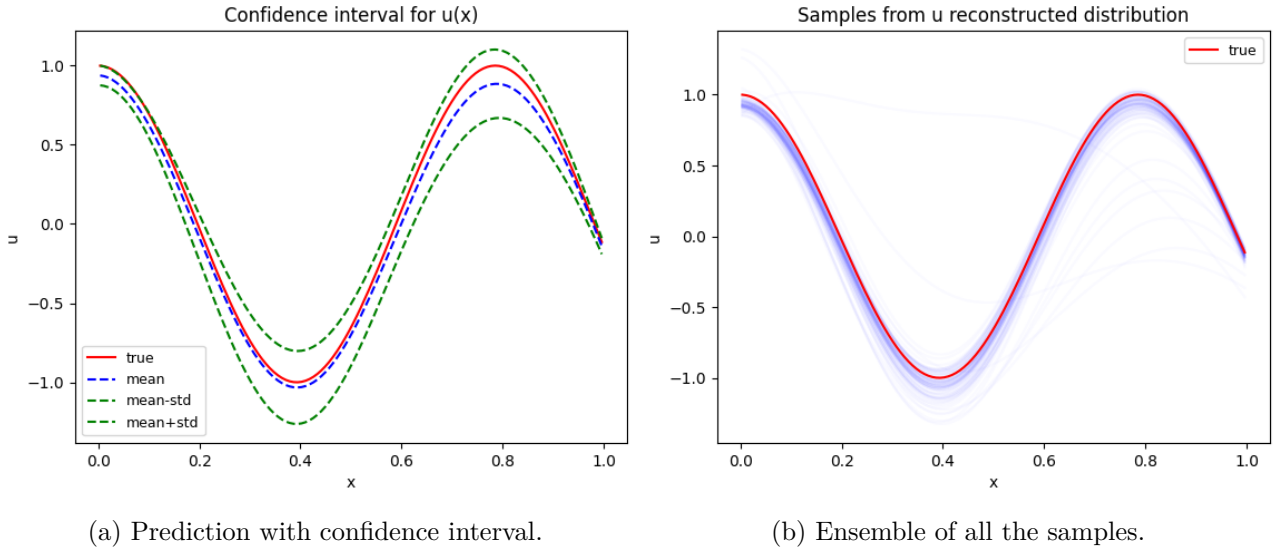


Figure 28: Poisson/Laplace Cosine Problem with HMC - solution

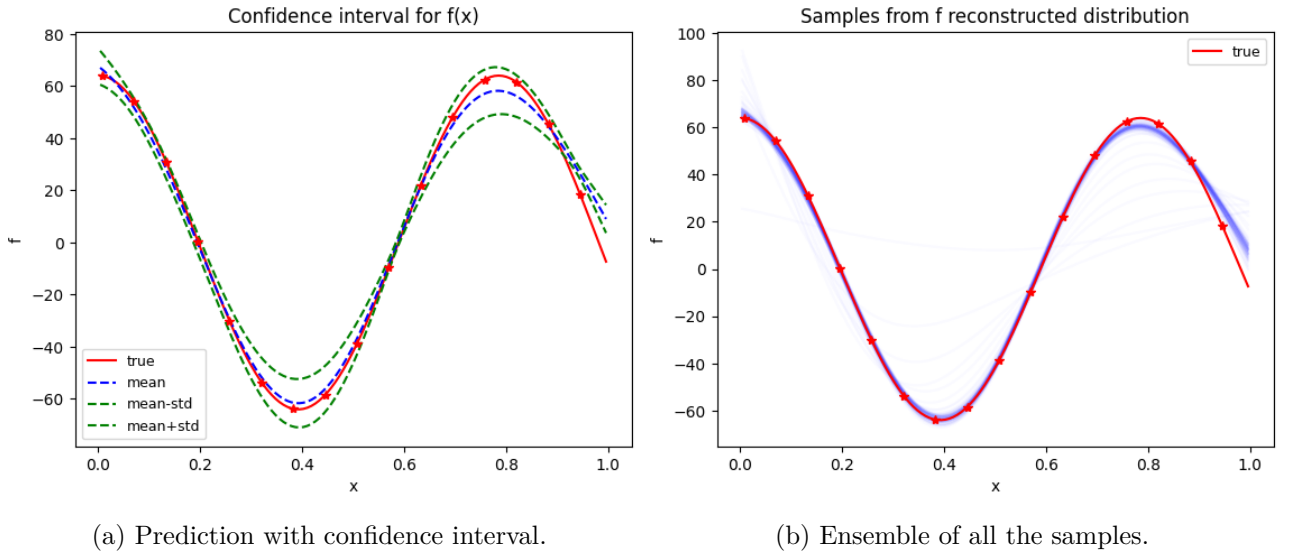


Figure 29: Poisson/Laplace Cosine Problem with HMC - parametric field

#### 6.4.4. Poisson/Laplace Problem - sine

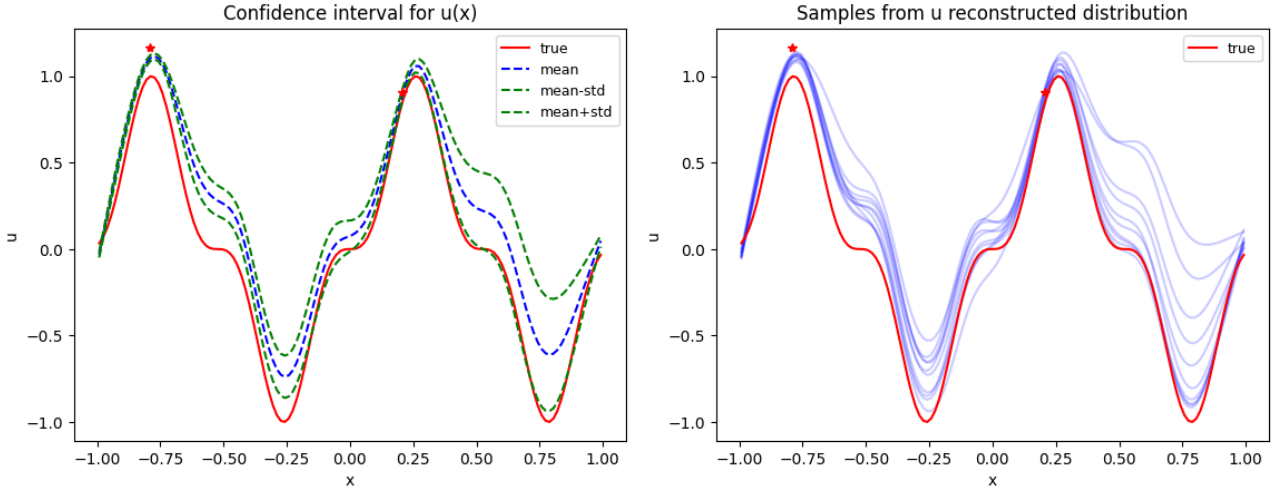
Finally, we proposed the same problem on a different dataset: we considered the Poisson Problem with the function  $u(x) = \sin^3(6x)$  as solution in the domain  $[-1, 1]$ . The forcing term corresponding to this solution is  $f(x) = 108 \sin(6x)(-2 \cos(6x)^2 + \sin(6x)^2)$ , and we assumed to have  $N_{fit} = 32$  noisy measurements on that, with  $\sigma_f = 0.1$ .

The higher number of fitting point required to have satisfying results is linked to the extremely oscillatory behavior of the function  $f$ , which was difficult to learn for the network.

Another consequence of the difficulty derived by the tangled function profiles was the necessity to include a deterministic Adam pre-training before the start of the bayesian algorithm, to make HMC start after a fairly good reconstruction of the oscillations of the function  $f$ .

We ran  $N = 11000$  epochs of Adam with the optimal parameters ([5]) and then, HMC with the training showed here. After performing a test case without measurements on the solution  $u$ , we noticed that the BPINN was able to reconstruct the function profile up to an additive constant. Therefore, we propose in Figure 30 a solution with  $N_{fit} = 2$  noisy data also on the solution  $u$ , attempting to fix the level of the constant missed by the network.

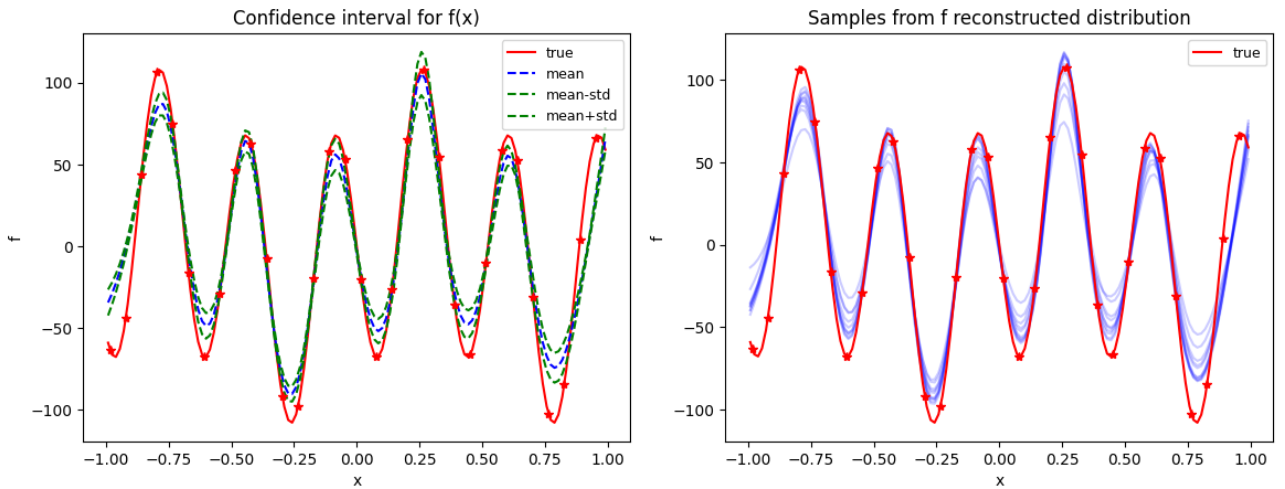
Parameter	Value
$N$	100
$B$	0
$S$	10
$L$	200
$dt$	$10^{-4}$
$\eta$	0.5



(a) Prediction with confidence interval.

(b) Ensemble of all the samples.

Figure 30: Poisson/Laplace Sine Problem with HMC - solution



(a) Prediction with confidence interval.

(b) Ensemble of all the samples.

Figure 31: Poisson/Laplace Sine Problem with HMC - solution



## 7. Conclusions

In this project, we implemented from scratch a library proposing a variety of functionalities for Bayesian Physics-Informed Machine Learning applied to Scientific Computing, an approach that enabled to tackle differential problems with an empowered version of Neural Networks that can take physical information into account and quantify uncertainty of the prediction.

The extension of the library is mainly in the horizontal direction, because it can provide a variety of functionalities for many tasks related to the B-PINNs framework, ranging from model training to dataset management. Its flexibility should moreover make it a suitable starting base for the refinement of the functionalities, with the ideal ultimate goal of producing a fully-comprehensive library for Bayesian Physics-Informed Deep Learning.

In the implementation, we relied on Python’s available tools for OOP, which enabled to design a skeleton for the library that should be enough rigid to guarantee a coherent pipeline and enough flexible to reduce the amount of functionalities constraints.

On the other hand, for what concerns the library validation, with the proposed showcase of results we stressed on the main implementation pillars (algorithms, physical information and code portability), to then concentrate the effort for obtaining quality results in a variety of applications for the specific HMC method.

Further developments of the library are highly encouraged; the library is open to plugins by design, because it has been thought for the extension either of new training algorithms and of new differential problems that can be dealt with, by implementing the corresponding classes inheriting from `Algorithm` or from `Equation`.

For more advanced problems, third part datasets require to be in the same format as the ones generated in the project to represent a suitable source of data for the model. Therefore, it could be very useful to add a simple interface for integrating and exploring real word data before pre-processing.

Consequently, another feature that can be added is the possibility to deal with the visualization of higher dimensional cases, by implementing suitable visualization tools for 2D uncertainty quantification and for the overall visualization of the 3D case.

For what concerns extensions of functionalities to the library, a proposed challenge can be the inclusion of tools to deal with parameter estimation problems, for which we already set-up the premises and the space to build upon with `PhysNN`. The crucial part in this development is the addition of the gradients on physical parameters  $\lambda$  to be learnt into the class `LossNN` and probably a wrapper for  $\lambda$  and  $\theta$ .

On the user-experience side, we could propose the implementation of a structured pipeline, based on grid search or more advanced techniques with the aim of partially automatize the tuning task as well, still being careful to the computational times required when asking for many queries of the parameters.

## A. Installation Guide

The code is contained in the GitLab repository [PACS\\_bpinnns](#).

To run the scripts contained in the executable repository, the user needs:

- `python` version 3.10.\* (download from [here](#))
- `virtualenv` version 20.14.\* (download from [here](#))

### Installation for Windows:

1. Go into the directory of your project with `cd project_folder_path`
2. Create an empty virtual environment with `py -m venv .\my_env_name`
3. Enter into the virtual environment with `my_env_name\scripts\activate`
4. Check that the environment is empty with `pip freeze`; normally, it should print nothing
5. Install the required packages from the `.txt` file `requirements.txt` with  
`pip install -r requirements.txt`
6. Run again `pip freeze` and check that the environment is no longer empty
7. Add the environment folder to your `.gitignore` (in order to avoid pushing the packages on git!)
8. To exit from the virtual environment, use `deactivate`

### Installation for Linux and Mac:

1. Go into the directory of your project with `cd project_folder_path`
2. Create an empty virtual environment with `virtualenv .\my_env_name`
3. Enter into the virtual environment with `source my_env_name/bin/activate`
4. Check that the environment is empty with `pip freeze`; normally, it should print nothing
5. Install the required packages from the `.txt` file `requirements.txt` with  
`pip install -r requirements.txt`
6. Run again `pip freeze` and check that the environment is no longer empty
7. Add the environment folder to your `.gitignore` (in order to avoid pushing the packages on git!)
8. To exit from the virtual environment, use `deactivate`

## References

- [1] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational Inference: a review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, apr 2017.
- [2] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks, 2015.
- [3] Daniele Ceccarelli. Bayesian Physics-Informed Neural Networks for inverse uncertainty quantification problems in cardiac electrophysiology. Master’s thesis, Politecnico di Milano, 2021.
- [4] Laurent Valentin Jospin, Hamid Laga, Farid Boussaid, Wray Buntine, and Mohammed Bannamoun. Hands-on Bayesian Neural Networks—a tutorial for deep learning users. *IEEE Computational Intelligence Magazine*, 17(2):29–48, 2022.
- [5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR 2015*, 2014.
- [6] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86, 1951.
- [7] Kevin Linka, Amelie Schäfer, Xuhui Meng, Zongren Zou, George Em Karniadakis, and Ellen Kuhl. Bayesian Physics Informed Neural Networks for real-world nonlinear dynamical systems. *Computer Methods in Applied Mechanics and Engineering*, 402:115346, 2022. A Special Issue in Honor of the Lifetime Achievements of J. Tinsley Oden.
- [8] Qiang Liu and Dilin Wang. Stein variational gradient descent: A general purpose bayesian inference algorithm. *NIPS 2016*, 2016.
- [9] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part I): data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.
- [10] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part II): data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.
- [11] Liu Yang, Xuhui Meng, and George Em Karniadakis. B-PINNs: Bayesian Physics-Informed Neural Networks for forward and inverse PDE problems with noisy data. *Journal of Computational Physics*, 425:109913, jan 2021.

The Neural Network sketches (figures [1,2,3,4,5](#)) have been designed using **TikZ** and taking inspiration from a [post](#) in the website **TikZ.net** by Izaak Neutelings, protected by the [Attribution-ShareAlike 4.0 International \(CC BY-SA 4.0\)](#) license.