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:

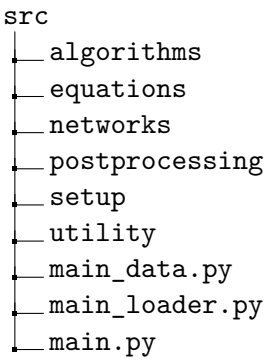


Figure 13: Hierarchy of `src` folder.

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 subsubsection 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 subsubsection 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 ouput, 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 overcomed 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.

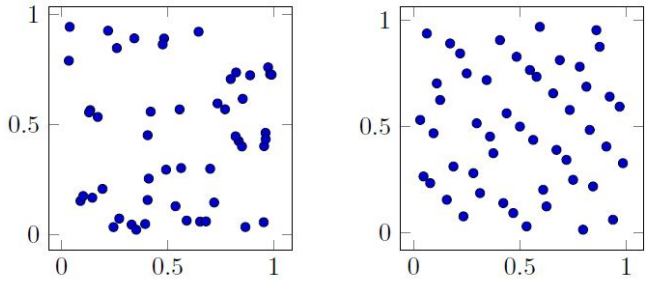
In order to define Sobol points, we need first to introduce general categories of low-discrepancy points that are $t - m - d$ nets and $t - d$ sequences, that we define in a $d-$dimensional space.

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 $p_1 + \cdots + p_d = m - t$ 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 $\{X^{(lb^n)}, \dots, X^{((l+1)b^m-1)}\}, l \in \mathbb{N}$ is a $t-m-d$ net in base b.*



(a) random sampling

(b) QMC Sampling (Sobol sequence)

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 feasiblity 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 aformentioned 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 subsubsection 4.3.3).

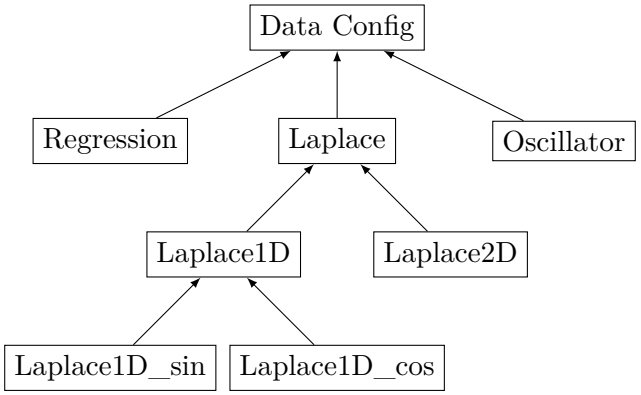


Figure 15: Inheritance tree of the dataset configuration classes.

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 $\boldsymbol{\mu}$ and dividing by the standard deviation σ:

$$\mathbf{u}_{norm} = \frac{\mathbf{u} - \boldsymbol{\mu}_u}{\sigma_u} \qquad \mathbf{f}_{norm} = \frac{\mathbf{f} - \boldsymbol{\mu}_f}{\sigma_f}$$

The normalizing statistics $\boldsymbol{\mu}$ and σ 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 $\boldsymbol{\lambda}_{norm}$.

5.4.3. Data Batching

While loading a dataset in Machine Learning models, it is common to split them it *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, contibute 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	$\nabla u = [\partial_1 u, \dots, \partial_{n_{in}} u]^T$	$n_s \times 1$	$n_s \times n_{in}$
	vector	$\nabla \mathbf{u} = \begin{bmatrix} [\partial_1 u_1, \dots, \partial_{n_{in}} u_1]^T \\ \vdots \\ [\partial_1 u_{n_{out}}, \dots, \partial_{n_{in}} u_{n_{out}}]^T \end{bmatrix}$	$n_s \times 1 \times n_{out}$	$n_s \times n_{in} \times n_{out}$
Divergence	vector	$\mathrm{div}\mathbf{u} = \partial_1 u_1 + \dots + \partial_{n_{in}} u_{n_{in}}$	$n_s \times n_{in}$	$n_s \times 1$
	tensor	$\mathrm{div}\underline{\underline{\mathbf{u}}} = \begin{bmatrix} \partial_1 u_{1,1} + \dots + \partial_{n_{in}} u_{1,n_{in}} \\ \vdots \\ \partial_1 u_{n_{out},1} + \dots + \partial_{n_{in}} u_{n_{out},n_{in}} \end{bmatrix}$	$n_s \times n_{in} \times n_{out}$	$n_s \times 1 \times n_{out}$
Laplacian	scalar	$\Delta u = \partial_1^2 u + \dots + \partial_{n_{in}}^2 u$	$n_s \times 1$	$n_s \times 1$
	vector	$\Delta \mathbf{u} = \begin{bmatrix} \partial_1^2 u_1 + \dots + \partial_{n_{in}}^2 u_1 \\ \vdots \\ \partial_1^2 u_{n_{out}} \dots + \partial_{n_{in}}^2 u_{n_{out}} \end{bmatrix}$	$n_s \times 1 \times n_{out}$	$n_s \times 1 \times n_{out}$

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;
- $\underline{\underline{\mathbf{u}}}$ 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.

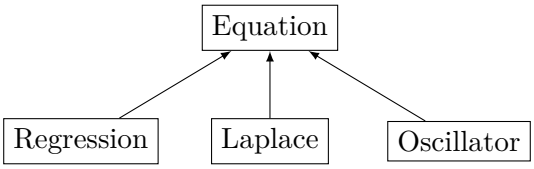


Figure 16: Inheritance tree of the `equation` classes.

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 subsubsection 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:

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

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:

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

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 ϕ, 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 ϕ 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:

$$F = -kx - c\frac{dx}{dt} \tag{42}$$

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

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

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 ψ is linked to ω and δ 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 $\boldsymbol{\theta} \coloneqq \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$. 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

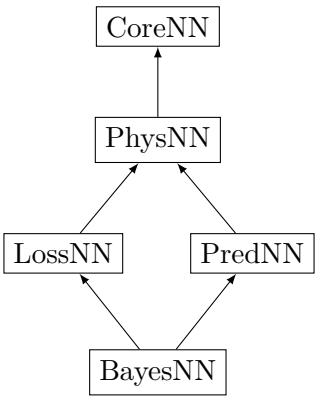


Figure 17: Inheritance tree of the `network` classes.

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 he function that we redefine in one of the following categories, according to the operation:
- classical binary operation $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$: return a `Theta` object whose `values` are the element-wise operation applied to the pair of `Theta` objects;
- polymorphic binary operation $(\boldsymbol{\theta}, n)$: works as a `Theta` object with all entries equal to n is passed as second argument;
- reverse polymorphic binary operation $(n, \boldsymbol{\theta})$: 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 cought by special methods in the second argument;
- unary operations $(\boldsymbol{\theta})$: 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	`__neg__`		$\boldsymbol{\theta}$	$-\boldsymbol{\theta}$
Sum	`__add__` `__radd__`	$+$	$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1 + \boldsymbol{\theta}_2$
Subtraction	`__sub__` `__rsub__`	$-$	$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2$
Multiplication[9]	`__mul__` `__rmul__`	$*$	$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1 * \boldsymbol{\theta}_2$
Division[9]	`__truediv__` `__rtruediv__`	$/$	$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1 / \boldsymbol{\theta}_2$
Power[9]	`__pow__`	$**$	$\boldsymbol{\theta}$, 2 $\boldsymbol{\theta}$, 0.5 $\boldsymbol{\theta}$, -1	$\boldsymbol{\theta}^2$ $\sqrt{\boldsymbol{\theta}}$ $1/\boldsymbol{\theta}$
Class Methods	`__len__` `__str__`	len print	$\boldsymbol{\theta}$	len($\boldsymbol{\theta}$) print($\boldsymbol{\theta}$)

Table 3: Overloaded operations for `Theta` objects.

5.6.2. Parameters and forward pass - CoreNN

CoreNN
+ n_inputs + n_out_sol + n_out_par + n_layers + n_neurons + activation + stddev + model + dim_theta + nn_params
- build_NN() - compute_dim_theta() + __init__() + initialize_NN() + forward()

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.

[9] The operation is intended to be element-wise.

5.6.3. Physics enforcement - PhysNN

PhysNN
+ pinn + inv_flag + u_coeff, f_coeff + norm_coeff
+ __init__() + tf_convert() + forward()

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 $\boldsymbol{\lambda}$ 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.

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 normalzation 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

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

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.

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.

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()

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

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 uncertainity 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 subsubsection 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 subsubsection 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 ensable 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.

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

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.

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.

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()

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.

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 optimzer

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 $\mathbf{m}$ and $\mathbf{v}$ 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 $\boldsymbol{\theta}$ 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 (`hamiltionian()`), 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 α 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 $\boldsymbol{\theta}$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 $\boldsymbol{\zeta}_\mu$ and $\boldsymbol{\zeta}_\rho$ describing the parametric family of distributions, by calling the private method `initialize_VI_params()`.

To sample NN parameters, we first have to learn $\boldsymbol{\zeta}_\mu$ and $\boldsymbol{\zeta}_\rho$, as presented in subsection 3.2, and we do that through GD. This task requires the computation of the gradient of the loss $f(\boldsymbol{\theta}, \boldsymbol{\zeta}) \coloneqq \log(Q(\boldsymbol{\theta}; \boldsymbol{\zeta})) - \log(P(\boldsymbol{\theta})P(D|\boldsymbol{\theta}))$ with respect to $\boldsymbol{\zeta}$; 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 ζ_i^μ and of a ζ_i^ρ:

$$\Delta_i^\mu = \frac{\partial f(\boldsymbol{\theta}, \boldsymbol{\zeta})}{\partial \boldsymbol{\theta}} + \frac{\partial f(\boldsymbol{\theta}, \boldsymbol{\zeta})}{\partial \zeta_i^\mu} \qquad i = 1, \dots, d_\theta$$

$$\Delta_i^\rho = \frac{\partial f(\boldsymbol{\theta}, \boldsymbol{\zeta})}{\partial \boldsymbol{\theta}} \frac{\varepsilon}{1 + e^{-\zeta_i^\rho}} + \frac{\partial f(\boldsymbol{\theta}, \boldsymbol{\zeta})}{\partial \zeta_i^\rho} \qquad i = 1, \dots, d_\theta$$

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

$$\Delta_i^\mu = \frac{\partial \log(Q(\boldsymbol{\theta}; \boldsymbol{\zeta}))}{\partial \boldsymbol{\theta}} - \frac{\partial \log(P(\boldsymbol{\theta})P(D|\boldsymbol{\theta}))}{\partial \boldsymbol{\theta}} + \frac{\partial \log(Q(\boldsymbol{\theta}; \boldsymbol{\zeta}))}{\partial \zeta_i^\mu} \qquad i = 1, \dots, d_\theta$$

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

At this stage, we are able to retrieve the $\partial_{\boldsymbol{\theta}} \log(P(\boldsymbol{\theta})P(D|\boldsymbol{\theta}))$ 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 $Q(\boldsymbol{\theta}; \boldsymbol{\zeta})$. In particular, we have:

$$\frac{\partial \log(Q(\boldsymbol{\theta}; \boldsymbol{\zeta}))}{\partial \boldsymbol{\theta}} = -\frac{\partial \log(Q(\boldsymbol{\theta}; \boldsymbol{\zeta}))}{\partial \zeta_i^\mu} = \frac{(\zeta_i^\mu - \boldsymbol{\theta})}{{\zeta_i^\sigma}^2}$$

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

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

Thanks to the analytical expression of derivatives we notice that Δ_i^μ 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 Δ_i^μ is only defined with by classical log-likelihood.

In the implementation, `compute_grad_rho()` returns a vector whose components are Δ_i^ρ by performing the above computations, then the private method `update_VI_params()` performs the GD updates on $\boldsymbol{\zeta}_\mu$ and $\boldsymbol{\zeta}_\rho$.

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 $\boldsymbol{\zeta}_\mu$ and standard deviation $\log(1 + e^{\boldsymbol{\zeta}_\rho})$.

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 ϕ.

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 ϕ 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:

$$K_{ij} = k(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j)$$

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

$$GK_{ij} = \nabla_{\boldsymbol{\theta}_i} k(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j)$$

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 $\boldsymbol{\theta}$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 detects 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.