POLITECNICO
MILANO 1863

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

Advanced Programming for Scientific Computing Project

Students: Giulia Mescolini, Luca Sosta

Tutors: Andrea Manzoni, Stefano Pagani

27 Feb 2023

Introduction

Project Overview

The efficiency of Deep Learning-based techniques can be exploited in Scientific Computing and interesting challenges arise when we want to consider physical information and uncertainty quantification (UQ).

We chose to implement from scratch the Bayesian Physics-Informed Neural Networks (B-PINNs) method within the context of a new library, designed to be flexible and predisposed to new extensions.

Nowadays, there are a lot of tools and libraries for Deep Learning, but in our specific domain they are mostly single application-oriented and created without the aim of being re-used by others.

Library Aim

In this project, we focused on the development of foundations for a B-PINNs library, prioritizing its implementation and potential rather than early results.

First of all, we wanted to make the simplest workflow possible to maintain flexibility for addition of other development of features.

This is also highlighted by a modular structure where each step of the main pipeline is independent and structured for multiple methods and settings, such as different physical problems or choices of optimizers.

These properties can be granted using the Object Oriented paradigm.

Object-Oriented Programming in Python

Nowadays, Deep Learning-based techniques are in the spotlight and the majority of the implementations rely on the Python language due to its fast development and rich offer of high level library such as TensorFlow.

Sometimes Python is considered overrated or weaker than compiled languages, but still it offers a wide variety of functionalities and *ad hoc* methods when dealing with Object Oriented Programming such as:

- `@property`: advanced version for attributes' getters and setters
- `@dataclass`: lightweight version of class for data handling
- `ABC`: *Abstract Base Class* module for virtuality in inheritance

Method and Library Foundations

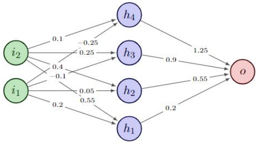
Bayesian Neural Networks

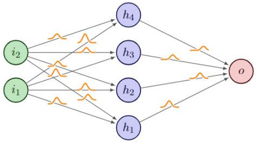
BNNs reconstruct the posterior distribution of the output $p(u|D)$, given the data.

Through the relation $u(x) = \mathcal{N}\mathcal{N}_\theta(x)$, we can obtain it by considering the posterior distribution of the NN parameters, $p(\theta|D)$.

The Bayes's Theorem enables us to retrieve it from prior and likelihood, thanks to:

$$p(\theta|D) \propto p(\theta)p(D|\theta)$$





Optimizers

We do not have a computationally feasible expression for $p(\theta|D)$, therefore we need algorithms for two purposes:

- Improving the approximation of the posterior
- Obtaining samples from the reconstructed distribution

The optimizers are in charge of these tasks, and they perform them in iterative procedures involving an epochs' loop, as in classical NN theory.

Physics-Informed Neural Networks

The introduction of the physical information comes with PINNs, which train the network to produce outputs satisfying PDEs by adding to the loss a term representing the equation residual.

Training datasets for PINNs are articulated in two main categories:

- Fitting points, where we impose the values of measurements
- Collocation points, where we impose the fulfillment of the equation

By integrating these aspects into the BNN framework already described, we finally obtain B-PINNs.

Core Featuures

In this context, we chose to build a modular pipeline composed of various blocks that are available in different versions to handle various application and methods.

Within the library, it is necessary to organize the communication among the three fundamental blocks:

- Data and batch organization
- Physics-Informed Neural Networks
- Uncertainty Quantification

To this aim, the organization of the attributes and the properties of the classes played a crucial role.

Main Pipeline

The main workflow of our library can be schematized in 7 points:

1. Handling of configuration parameters
2. Data generation
3. Pre-processing of dataset
4. Model and optimizer initialization
5. Training phase
6. Prediction and performance evaluation
7. Storage and plot of results

Side Implementations

The parameters' handling section manages to deal with three sources of information: `.json` configuration files and `@dataclass` for data generation. Those are merged and can be overridden by command-line arguments and stored into a single `Param` object.

The `Storage` class saves plots, logs and values into a folder. So, they are ready to be regenerated by the `Plotter`, which works only asking for the path folder.

The `Equation` class manages all physical information of the test case and computes the residual for PDE with a module for Automatical Differentiation. It also converts the physical domain into the computational one.

Core Implementations

Data Generation

In this project we considered a complete test-case workflow which starts right from the generation of synthetic data.

When dealing with PINNs, we need different types of datasets which require different sampling methods:

- Uniform: generated on a grid, useful to compute errors and plot
- Random: random uniform sampling of points inside the domain
- Sobol: pseudo-random sampling of points with low-discrepancy

We also implemented the generation of data in subdomain or in multiple subdomains, to mimic the lack of measurements in some areas.

Dataset Creation

The pre-processing phase starts by asking for subsets of generated data to avoid useless regeneration for similar test cases. In addiction, it transforms the dataset with the methods provided by `Equation`; in our test cases, we applied a standardization of data range.

Dealing with PINNs, we need to split the dataset into multiple sections without modifying the original data and this is achieved through the `@property` decorator, which provides all privatization features.

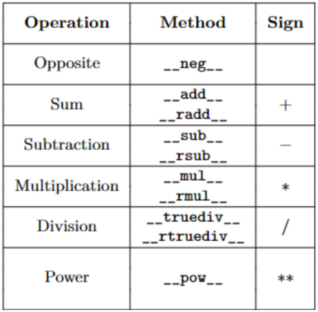
Finally, for improving efficiency during training all the components of dataset are batched, allowing to use a small sample of data in each epoch. This is performed with the class's special method `__next__()`.

Theta

TensorFlow stores NN parameters in a list of Tensors, but it is unpractical and redundant for element-wise or norm-based operations used in the algorithms.

We implemented the class `Theta`, where we overrode the algebraic operations to avoid the repetition of code blocks.

This class improves readability and allows us to be less error-prone when dealing with differentiability preservation.



Operation	**Method**	**Sign**
Opposite	`__neg__`	
Sum	`__add__` `__radd__`	+
Subtraction	`__sub__` `__rsub__`	−
Multiplication	`__mul__` `__rmul__`	*
Division	`__truediv__` `__rtruediv__`	/
Power	`__pow__`	**

CoreNN and PhysNN

We implemented B-PINNs as a sovrastructure built with inheritance on a TensorFlow fully connected model. As typical in Physics-Informed contexts, the network is quite small.

The class `CoreNN` is the first building block, representing the basic NN: it stores and manages the model and its parameters, and it performs a forward step.

`PhysNN` inherits from it and contains the `Equation`, the PDE parameters and the information needed for the transformation from the real to the computational domain.

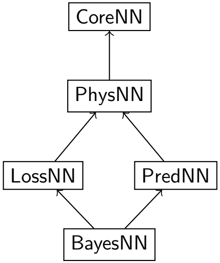
BayesNN

BayesNN is the third superstructure and it is the class instantiated in the executable script.

It is in charge of two disjoint tasks:

- the loss computation
- the prediction, providing also UQ

The above functionalities have been separated in two distinct classes, from which BayesNN inherits in the context of multiple inheritance.



LossNN

The class `LossNN` computes losses and metrics, by separating their components.

For its computations, it relies on implemented static methods and specially for the physical loss it interacts with the parent class `PhysNN` to retrieve the equation residual.

It also computes the gradient of the loss with respect to NN parameters, quantity needed by all training algorithms.

`PredNN`

The functionalities of this class are invoked after the action of the optimizer, which fills the list of NN parameters samples.

Given the set of parameters, it reconstructs the set of samples of the ultimate output, obtained not only though a forward pass, but also with a transformation back to the physical domain.

It is also responsible for error and uncertainty quantification, automatically evaluated across the different directions.

`Algorithm`

Optimizers inherit from an abstract base class, `Algorithm`, which fixes the blueprint for the training pipeline.

It receives and stores the dataset in a property, and acts on the instance of `BayesNN` with its methods.

Two abstract methods enable to differentiate algortihms one from each other: `sample_theta()`, generating at each iteration a NN parameters' sample, and `select_thetas()`, which finalizes the set of samples at the end of the training.

Trainer and Adam

The class `Trainer` is an interface between the main script and the algorithms and plays the role of a training manager.
In this sense, it also includes the possibility to include a deterministic pre-training.

The Adam optimizer is one of the most popular, fast and effective. We mounted it on the same structure of Bayesian algorithms, simply by overriding:

- `sample_theta()` as the classical Adam training step
- `select_thetas()` as a selector only of the final sample of the list

Hamiltonian Monte Carlo

The HMC method combines MCMC and Hamiltonian Dynamics.

- `sample_theta()` is in charge of the leap-frog and acceptance-rejection step
- `select_thetas()` selects the final set of samples by introducing burn-in and stride

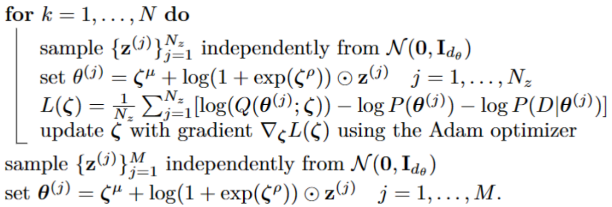
This method requires reasonable computational times, but it involves many parameters and therefore it is challenging to tune.

```
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
        $\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)}$
```

Variational Inference

This algorithms approximates the posterior with a multivariate gaussian whose parameters μ and ρ are learned in a deterministic training. Then, as many samples as desired can be drawn from the learned distribution.

- `sample_theta()` performs updates of μ and ρ in the epoch loop
- `select_thetas()` as a selector only of the final sample of the list



```
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 $\theta^{(j)} = \zeta^\mu + \log(1 + \exp(\zeta^\rho)) \odot \mathbf{z}^{(j)} \quad j = 1, \dots, N_z$
    $L(\zeta) = \frac{1}{N_z} \sum_{j=1}^{N_z} [\log(Q(\theta^{(j)}; \zeta)) - \log P(\theta^{(j)}) - \log P(D|\theta^{(j)})]$
    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 $\theta^{(j)} = \zeta^\mu + \log(1 + \exp(\zeta^\rho)) \odot \mathbf{z}^{(j)} \quad j = 1, \dots, M.$
```

Stein Variational Gradient Descent

This method has a complex underlying theory, involving the minimization of the KL divergence starting from an initial approximating distribution.

Its implementation proposes challenges, such as the task of managing and exchaning loss information among N networks in parallel.

- `sample_theta()` trains the ensemble of the networks
- `select_thetas()` returns all the final parameters

```
**for** $k = 1, \dots, E$ **do**
    $\phi(\boldsymbol{\theta}_i) = \frac{1}{N} \sum_{j=1}^{N} k(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) \nabla_{\boldsymbol{\theta}_j} L(\boldsymbol{\theta}_j) + \nabla_{\boldsymbol{\theta}_j} k(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) \qquad \forall i = 1, \dots, N$
    $\boldsymbol{\theta}_i = \boldsymbol{\theta}_i + \varepsilon \phi(\boldsymbol{\theta}_i) \qquad \forall i = 1, \dots, N$
collect the parameters of all networks after $E$ epochs $\{\boldsymbol{\theta}^{(i)}\}_{i=1}^{N}$.
```

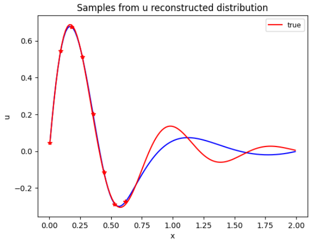
Results Showcase

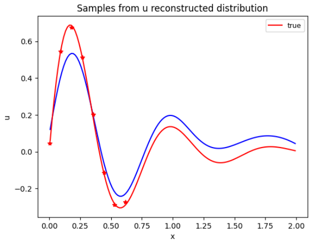
Damped Harmonic Oscillator

This test case involves an ODE with availability of measures only for initial times.

$$\frac{d^2x}{dt^2} + 2\delta\frac{dx}{dt} + \omega^2 x = 0 \quad \delta = 2, \omega = 8$$

The NN (above) performs a good interpolation only in the first region.
Thanks to the physical law, instead, the PINN (below) can reconstruct the whole evolution.

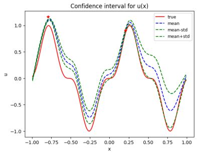


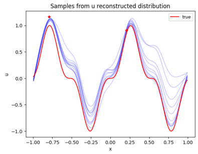


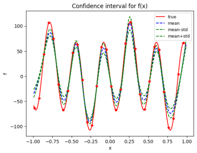
Poisson problem with HMC

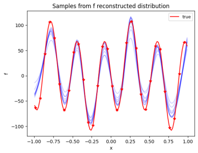
We consider the Poisson problem $-\Delta u = f$ with solution $u(x) = \sin^3(6x)$.

The knowledge of f is limited to sparse and noisy measurements. Moreover, with this method we can also exploit information on u, either on the boundary or inside the domain.









Conclusions

Conclusions

The main purpose of this project was to prioritize new implementations and scalability of the code rather than obtaining in depth results.

In this presentation, we showed only some of the major features and few results needed to highlight them.

The proposed library release managed to achieve the purposes initially defined on flexibilty and modularity.

Further Developments

Suggested developments are :

- Implementation of other `Equations` and `Algorithms`
- Interface for third-part data and relative Plotter utilities
- Parameter Estimation Problem set-up
- Automatization of tuning procedure

Main References

1. *B-PINNs: Bayesian Physics-Informed Neural Networks for Forward and Inverse PDE Problems with Noisy Data*, Liu Yang, Xuhui Meng, George Em Karniadakis, Mar 2020.
2. *Bayesian Physics Informed Neural Networks for real-world nonlinear dynamical systems*, Kevin Linka, Amelie Schäfer, Xuhui Meng, Zongren Zou, George Em Karniadakis, and Ellen Kuhl, May 2022.
3. *Bayesian Physics-Informed Neural Networks for Inverse Uncertainty Quantification problems in Cardiac Electrophysiology*, Master Thesis at Politecnico di Milano by Daniele Ceccarelli.

Ending

Thank you for the attention!