








Physics-Informed Neural networks for Advanced modeling

Dario Coscia ^{1*}, Anna Ivagnes ^{1*}, Nicola Demo ^{1*}, and Gianluigi Rozza ^{1*}

¹ SISSA, International School of Advanced Studies, Via Bonomea 265, Trieste, Italy * These authors contributed equally.

DOI: [10.21105/joss.05352](https://doi.org/10.21105/joss.05352)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Daniel S. Katz](#) 

Reviewers:

- [@yorkiva](#)
- [@y-yao](#)
- [@akshaysubr](#)

Submitted: 21 March 2023

Published: 19 July 2023

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

Introduction

Artificial Intelligence (AI) strategies are massively emerging in several fields of academia and industrial research Wang & Liao (2004) due to the growing disposal of data, as well as the great improvement in computational resources. In the area of applied mathematics and simulations, AI strategies are being used to solve problems where classical methods fail (Cuomo et al., 2022). However, the amount of data required to analyze complex systems is often insufficient to make AI predictions reliable and robust. Physics-informed neural networks (PINNs) have been formulated (Raissi et al., 2019) to overcome the issues of missing data, by incorporating the physical knowledge into the neural network training. Thus, PINNs aim to approximate any differential equation by solving a minimization problem in an unsupervised learning setting, learning the unknown field in order to preserve the imposed constraints (boundaries and physical residuals). Formally, we consider the general form of a differential equation, which typically presents the most challenging issues from a numerical point of view:

$$\begin{aligned}\mathcal{F}(\mathbf{u}(\mathbf{z}); \alpha) &= \mathbf{f}(\mathbf{z}) \quad \mathbf{z} \in \Omega, \\ \mathcal{B}(\mathbf{u}(\mathbf{z})) &= \mathbf{g}(\mathbf{z}) \quad \mathbf{z} \in \partial\Omega,\end{aligned}\tag{1}$$

where $\Omega \subset \mathbb{R}^d$ is the domain and $\partial\Omega$ the boundaries of the latter. In particular, \mathbf{z} indicates the spatio-temporal coordinates vector, \mathbf{u} the unknown field, α the physical parameters, \mathbf{f} the forcing term, and \mathcal{F} the differential operator. In addition, \mathcal{B} identifies the operator indicating arbitrary initial or boundary conditions and \mathbf{g} the boundary function. The PINN's objective is to find a solution to the problem, which is done by approximating the true solution \mathbf{u} with a neural network $\hat{\mathbf{u}}_\theta : \Omega \rightarrow \mathbb{R}$, with θ network's parameters. Such a model is trained to find the optimal parameters θ^* whose minimizing the physical loss function depending on the physical conditions $\mathcal{L}_\mathcal{F}$, boundary conditions $\mathcal{L}_\mathcal{B}$ and, if available, real data $\mathcal{L}_{\text{data}}$:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \mathcal{L} = \underset{\theta}{\operatorname{argmin}} (\mathcal{L}_\mathcal{F} + \mathcal{L}_\mathcal{B} + \mathcal{L}_{\text{data}}).\tag{2}$$

The PINNs framework is completely general and applicable to different types of ordinary differential equations (ODEs), or partial differential equations (PDEs). Nevertheless, the loss function strictly depends on the problem chosen to be solved, since different operators or boundary conditions lead to different losses, increasing the difficulty to write a general and portable code for different problems.



Figure 1: PINA logo.

PINA, *Physics-Informed Neural networks for Advanced modeling*, is a Python library built using PyTorch that provides a user-friendly API to formalize a large variety of physical problems and solve it using PINNs easily.

Statement of need

PINA is an open-source Python library that provides an intuitive interface for the approximated resolution of Ordinary Differential Equations and Partial Differential Equations using a deep learning paradigm, in particular via PINNs. The gain of popularity for PINNs in recent years, and the evolution of open-source frameworks, such as TensorFlow, Keras, and PyTorch, led to the development of several libraries, whose focus is the exploitation of PINNs to approximately solve ODEs and PDEs. We here mention some PyTorch-based libraries, NeuroDiffEq (Chen et al., 2020), IDRLNet (Peng et al., 2021), NVIDIA Modulus (NVIDIA Modulus, 2023), and some TensorFlow-based libraries, such as DeepXDE (Lu et al., 2021), TensorDiffEq (McClenny et al., 2021), SciANN (Haghighat & Juanes, 2021) (which is both TensorFlow and Keras-based), PyDens (Koryagin et al., 2019), Elvet (Araz et al., 2021), NVIDIA SimNet (Hennigh et al., 2021). Among all these frameworks, PINA wants to emerge for its easiness of usage, allowing the users to quickly formulate the problem at hand and solve it, resulting in an intuitive framework designed by researchers for researchers.

Built over PyTorch — in order to inherit the autograd module and all the other features already implemented — PINA provides indeed documented API to explain usage and capabilities of the different classes. We have built several abstract interfaces not only for better structure of the source code but especially to give the final user an easy entry point to implement their own extensions, like new loss functions, new training procedures, and so on. This aspect, together with the capability to use all the PyTorch models, makes it possible to incorporate almost any existing architecture into the PINA framework. We have decided to build it on top of PyTorch in order to exploit the autograd module, as well as all the other features implemented in this framework. The final outcome is then a library with incremental complexity, capable of being used by the new users to perform the first investigation using PINNs, but also as a core framework to actively develop new features to improve the discussed methodology.

The high-level structure of the package is depicted in our API; the approximated solution of a differential equation can be implemented using PINA in a few lines of code thanks to the intuitive and user-friendly interface. Besides the user-friendly interface, PINA also offers several examples and tutorials, aiming to guide new users toward an easy exploration of the software features. The online documentation is released at <https://mathlab.github.io/PINA/>, while the robustness of the package is continuously monitored by unit tests.

PINA workflow is characterized by 3 main steps: the problem formulation, the model definition, i.e. the structure of the neural network used, and the training, eventually followed by the data visualization.

Problem definition in PINA

The first step is the formalization of the problem. The problem definition in the PINA framework is inherited from one or more problem classes (at the moment the available classes are `SpatialProblem`, `TimeDependentProblem`, `ParametricProblem`), depending on the nature of the problem treated. The user has to include in the problem formulation the following components:

- the information about the domain, i.e. the spatial and temporal variables, the parameters of the problem (if any), with the corresponding range of variation;
- the output variables, i.e. the unknowns of the problem;
- the conditions that the neural network has to satisfy, i.e. the differential equations, the boundary and initial conditions.

We highlight that in PINA we abandoned the classical division between physical loss, boundary loss, and data loss: all these terms are encapsulated within the `Condition` class, in order to keep the framework as general as possible. The users can indeed define all the constraints the unknown field needs to satisfy, avoiding any forced structure in the formulation and allowing them to mix heterogeneous constraints — e.g. data values, differential boundary conditions. Moreover PINA already implements functions to easily compute the differential operations (gradient, divergence, laplacian) over the output(s) of interest, aiming to make the problem definition an easy task for the users.

Model definition in PINA

The second fundamental step is the definition of the model of the neural network employed to find the approximated solution to the differential problem in question. In PINA, the user has the possibility to use either a custom torch network model, or to exploit one of the built-in models such as `FeedForward`, `MultiFeedForward` and `DeepONet`, defining their characteristics during instantiation — i.e. number of layers, number of neurons, activation functions. The list of the built-in models will be extended in the next release of the library.

Training in PINA

In the last step, the actual training of the model in order to solve the problem at hand is computed. In this phase, the residuals of the conditions (expressed in the problem) are minimized in order to provide the target approximation. The sampling points where the physical residuals are evaluated can be passed by the user, or automatically sampled from the original domain using one of the available sampling techniques. The training is then computed for a certain amount of epochs, or until reaching the user-defined loss threshold. Once the model is ready to be inferred, the user can save it onto a binary file for future reusing, by inheriting the PyTorch functionality. The user can also evaluate the (trained) model for any new input, or just use it together with the `Plotter` in order to render the predicted output variables.

Acknowledgements

We thank our colleagues and research partners who contributed in the former and current developments of PINA library. This work was partially funded by European Union Funding for Research and Innovation — Horizon 2020 Program — in the framework of European Research Council Executive Agency: H2020 ERC CoG 2015 AROMA-CFD project 681447 “Advanced Reduced Order Methods with Applications in Computational Fluid Dynamics” P.I. Professor Gianluigi Rozza.

References

- Araz, J. Y., Criado, J. C., & Spannowsky, M. (2021). Elvet—a neural network-based differential equation and variational problem solver. *arXiv Preprint arXiv:2103.14575*.
- Chen, F., Sondak, D., Protopapas, P., Mattheakis, M., Liu, S., Agarwal, D., & Di Giovanni, M. (2020). NeurodiffEq: A python package for solving differential equations with neural networks. *Journal of Open Source Software*, 5(46), 1931. <https://doi.org/10.21105/joss.01931>
- Cuomo, S., Cola, V. S. di, Giampaolo, F., Rozza, G., Raissi, M., & Piccialli, F. (2022). *Scientific machine learning through physics-informed neural networks: Where we are and what's next*. arXiv. <https://doi.org/10.48550/ARXIV.2201.05624>
- Deng, L., Yu, D., & others. (2014). Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4), 197–387. <https://doi.org/10.1561/9781601988157>
- Haghighat, E., & Juanes, R. (2021). Sciann: A keras/tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks. *Computer Methods in Applied Mechanics and Engineering*, 373, 113552. <https://doi.org/10.1016/j.cma.2020.113552>
- Hennigh, O., Narasimhan, S., Nabian, M. A., Subramaniam, A., Tangsali, K., Fang, Z., Rietmann, M., Byeon, W., & Choudhry, S. (2021). NVIDIA SimNet™: An AI-accelerated multi-physics simulation framework. *International Conference on Computational Science*, 447–461. https://doi.org/10.1007/978-3-030-77977-1_36
- Koryagin, A., Khudorozkov, R., & Tsimfer, S. (2019). PyDEns: A python framework for solving differential equations with neural networks. *arXiv Preprint arXiv:1909.11544*.
- Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2021). DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 63(1), 208–228. <https://doi.org/10.1137/19m1274067>
- McClenny, L. D., Haile, M. A., & Braga-Neto, U. M. (2021). TensorDiffEq: Scalable multi-GPU forward and inverse solvers for physics informed neural networks. *arXiv Preprint arXiv:2103.16034*.
- NVIDIA Modulus. (2023). <https://github.com/NVIDIA/modulus>.
- Peng, W., Zhang, J., Zhou, W., Zhao, X., Yao, W., & Chen, X. (2021). IDRLnet: A physics-informed neural network library. *arXiv Preprint arXiv:2107.04320*.
- Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378, 686–707. <https://doi.org/10.1016/j.jcp.2018.10.045>
- Wang, D. H., & Liao, W. H. (2004). Modeling and control of magnetorheological fluid dampers using neural networks. *Smart Materials and Structures*, 14(1), 111. <https://doi.org/10.1088/0964-1726/14/1/011>