PINA: A PYTORCH FRAMEWORK FOR DEEP DIFFERENTIAL EQUATION LEARNING FOR RESEARCH AND PRODUCTION ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

The last years have manifested the artificial intelligence revolution in several fields. Within the scientific computing community, there has been a remarkable effort to exploit the advancements in machine learning to address the limitations of conventional methods for solving differential equations. Notably, physics-informed neural network (PINN) and neural operator (NO) approaches have emerged as central players due to their promising and innovative approaches to computing differential equations' solutions. In this contribution, we are going to present a versatile software designed for tackling differential equation learning using PINN and NO methodologies. The package is called `PINA`, an open-source Python library built upon the robust foundations of `PyTorch` and `Lightning`. It empowers end-users to formulate their problem and craft their models to effortlessly compute the solution. The modular structure of `PINA` permits it to adapt PINN and NO schemas for user specifics, thus offering the freedom to select the most suitable learning techniques for their particular problem domain. Furthermore, by leveraging the capabilities of the `Lightning` package, `PINA` adapts to various hardware setups, including GPUs and TPUs. This adaptability positions `PINA` as an ideal candidate for the transition of these methodologies into production and industrial pipelines, where computational efficiency and scalability are of paramount importance.

1 INTRODUCTION

In recent years, the world has seen an unprecedented revolution in artificial intelligence (AI) and machine learning (ML), that has permeated numerous sectors, transforming solutions and processes. Within the scientific computing community, this revolution has manifested itself as a powerful tool for overcoming the limitations inherent in traditional methods for solving complex differential equations.

Among the promising developments in this arena, two standout approaches have emerged as central players for differential equation solutions: Physics-Informed Neural Networks (PINN) and Neural Operator (NO) methodologies. These models exploit the knowledge of the equations, using them directly during the training in order to approximate the unknown output of interest, such that the final distribution satisfies the physical constraints (differential terms, boundary, and initial conditions). This integration of the equations into the training process simplifies the formulation of the problem, making it easier to formalize new problems effectively in practice. For this reason, we present in this contribution an innovative software tool designed to harness the potential of PINN and NO methodologies for differential equation learning. This package is called `PINA`, and it is an open-source Python library meticulously crafted to empower users in the realm of differential equation solving. Built upon the robust foundations of PyTorch and Lightning, `PINA` represents a versatile and user-friendly resource that wants to simplify the way we approach complex mathematical problems.

PINA presents a modular architectural design, endowing users with the capacity to define and construct models tailored to their specific problem domains, thereby facilitating the computation of solutions. Within the framework, each component is extensible, allowing for the exploration of an extensive array of techniques and strategies without theoretical limitations. This design supports

PINA as a collaborative, open-source framework for the PINN and NO methodologies, where collective development efforts converge to streamline implementation processes and perpetually integrate state-of-the-art techniques. PINA aspires to be the central hub for the PINN and NO research community, harmonizing development efforts to simplify implementation while perpetually maintaining a repository of cutting-edge methodologies. This collaborative paradigm seeks to empower users with a versatile toolkit that not only streamlines the implementation journey but also ensures their continuous alignment with the latest advancements in these techniques.

Furthermore, `PINA`'s integration with the Lightning package ensures that it can seamlessly adapt to various hardware setups, including GPUs and TPUs. This adaptability is crucial for transitioning these cutting-edge methodologies into real-world production and industrial pipelines. In domains where computational efficiency and scalability are paramount, PINA emerges as an ideal candidate to drive such methods to their industrial maturity.

In the pages that follow, we will delve deeper into the features, capabilities, and practical applications of `PINA`, illustrating how this innovative software tool can empower you to unlock the full potential of AI in solving complex differential equations. Welcome to the future of scientific computing with `PINA`—a tool that brings the power of AI to your fingertips.

2 Background

ODEs and PDEs are used to describe different physical phenomena in a mathematical form. Local updates expressed by partial or total derivatives are used to represent the evolution of a function characterizing a system. Formally, the general form of a differential equation, which we aim to solve, can be written as:

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

where $\Omega \in \mathbb{R}^{d_z}$ is a suitable domain with $\partial\Omega$ its boundaries. The solution field is $\mathbf{u} \in \mathbb{U}$, the variables $\mathbf{z}$ indicate all the spatio-temporal coordinates, $\alpha \in \mathbb{A}$ the physical parameters, $\mathbf{f}$ the forcing term, and $\mathcal{F}$ a differential operator describing the dynamics. Finally, $\mathcal{B}$ is the operator which indicates arbitrary initial or boundary conditions, with $\mathbf{g}$ the function on the boundaries.

Solving ODEs and PDEs of the form in Equation equation 1 is one of the main computational challenges in mathematics and engineering. Numerical solvers, such as finite element methods (FEM), finite difference methods (FDM), or finite volume method (FVM), rely on discretizing the domain Ω (Morton & Mayers, 2005; Quarteroni & Quarteroni, 2009). For many complex domains, the discretization is not straightforward, and a specific study is needed to ensure the final accuracy of the solver. Moreover, these solvers are often computationally expensive, resulting in high energy consumption, and slow computational time.

Over the past decades, multiple deep learning methods have risen for solving the problem formalized in equation 1, aiming to overcome the classical numerical solver issues. Eventually, a dichotomy of methodologies can be made: Neural Operator (NO) methods, which assume knowledge of the system in the form of data; and Physics Informed Neural Networks (PINNs), which use the underlying equation to learn the solution.

2.1 Neural Operator Methods

Neural Operator (NO) methods (Li et al., 2020; Lu et al., 2021a; Bhattacharya et al., 2021; Kovachki et al., 2021; Brandstetter et al., 2022) build a mapping from infinite-dimensional function spaces by using a supervised learning strategy. Given a specific ODE or PDE as in the form of equation 1, a neural operator $\mathrm{G} : \mathbb{U}' \to \mathbb{U}$ is trained by a finite set of N observations $\{(\mathbf{u}'_i, \mathbf{u}_i)\}_{i=1}^N$, such that:

$$\mathrm{G}(\mathbf{u}') = \mathbf{u}. \tag{2}$$

For example, a NO could map the field at the initial temporal condition of a PDE, to the evolution at a specific time step; or the parameter of a differential equation to its solution for the specific parameter. NO possesses important characteristics: they are discretization invariant, i.e. the model is not defined on a fixed grid; the input and output is a function; the universal approximation theorem for operator holds Chen & Chen (1995).

2.2 PHYSICS INFORMED NEURAL NETWORKS

In many situations training data are not available, and NO can not be trained using a supervised loss. As an alternative, PINNs (Raissi et al., 2019) have been proposed. PINNs are trained by approximating the true solution of equation 1 with a neural network $\mathbf{u}_\theta \approx \mathbf{u}$ with parameters θ. In PINNs the network is trained directly with the ODE or PDE itself, ensuring that equation 1 is satisfied by the network:

$$\mathcal{L}(\theta) = \mathcal{L}_{\mathcal{F}} + \mathcal{L}_{\mathcal{B}}. \tag{3}$$

The first term is the *physics-informed* loss inside the domain Ω, while the second one is a supervised loss for boundary or initial conditions. Different types of losses can be implemented, for example using the MSE loss equation 3 becomes:

$$\mathcal{L}(\theta) = \frac{1}{N_f}\sum_{i=1}^{N_f} \|\mathcal{F}(\mathbf{u}_\theta(\mathbf{z}_i); \alpha_i) - \mathbf{f}(\mathbf{z}_i)\|_2^2 + \frac{1}{N_b}\sum_{i=1}^{N_b} \|\mathcal{B}(\mathbf{u}_\theta(\mathbf{z}_i)) - \mathbf{g}(\mathbf{z}_i)\|_2^2, \tag{4}$$

where N_f is the number of collocation points sampled inside Ω, and N_b the number of collocation points sampled in $\partial\Omega$.

Since PINN's inception, many follow-up improvements have been made to improve training stability and convergence. Examples of further research include studying different losses (Kharazmi et al., 2019; Wang et al., 2022; McClenny & Braga-Neto, 2020), sample strategies for collocation points (Wu et al., 2023; Nabian et al., 2021; Daw et al., 2023) to speed up convergence, or specific network architecture (Wang et al., 2021a;b) and input augmentation (Demo et al., 2023; Lu et al., 2021c) to ease the neural network training.

2.3 DEEP DIFFERENTIAL EQUATION LEARNING SOFTWARE

The recent advancements in the field of deep learning for differential equations learning, as well as the evolution of open-source frameworks, such as `TensorFlow` (Abadi et al., 2015), and `PyTorch` (Paszke et al., 2019) led to the development of several libraries for solving ODEs and PDEs via PINNs and NO. PINN `TensorFlow`-based libraries include `DeepXDE` (Lu et al., 2021b) (which also supports `PyTorch`), `TensorDiffEq` (McClenny et al., 2021) and `PyDEns` (Koryagin et al., 2019); while `PyTorch`-based libraries include `NeuroDiffEq` (Chen et al., 2020), `IDRLNet` (Peng et al., 2021). For NO `NeuralOperator` (Li et al., 2020; Kovachki et al., 2021) is the main library.

There are multiple challenges with the packages mentioned above that limit their usage for research and production environments. First, most of the packages lack abstract interfaces which limit the possibility of adding extensions, like new loss functions or training procedures. Additionally, the packages presented are sectorized to only PINNs or NOs, without the possibility of combining the two methodologies, which is a new research direction in the field (Li et al., 2021; Wang et al., 2021b). Another common problem of the libraries is the absence of common deep learning advancements for training such as multiple device training, modern model compression techniques, gradient accumulation, and so on. Finally, the possibility of inserting common deep-learning loggers into the training for monitoring is missing.

3 PINA

`P`hysics `I`nformed `N`eural network for `A`dvanced modeling (`PINA`) is an open-source Python library providing an intuitive interface for solving differential equations using PINNs, NOs or both together. The software is built-in `PyTorch`, with `PyTorchLightning` (Falcon & The PyTorch Lightning team, 2019) as backhand. Employing `PyTorchLightning` as backhand is done to offer professional AI researchers and machine learning engineers the possibility of using advancement training strategies provided by the library. In addition, it provides the possibility to add arbitrary self-contained routines (callbacks) to the training for easy extensions without the need to touch the underlying code.

`PINA` is an open-interface software that gives the final user an easy entry point to implement their extensions. The application programming interface (API) of `PINA` is schematized in Figure 1. The

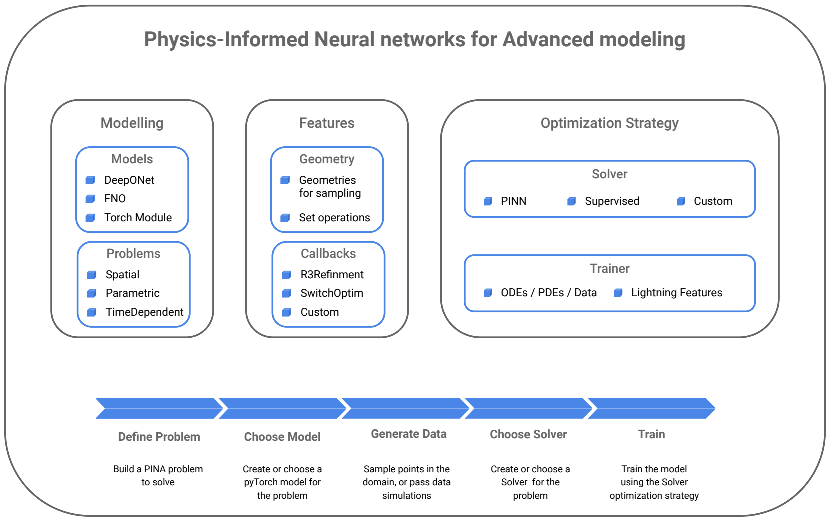


Figure 1: PINA package application programming interface. Starting from the problem definition, a specific model is passed to the solver, which defines, together with the trainer, the optimization strategy of the model.

pipeline to solve differential equations with PINA follows five steps: problem definition, model selection, data generation, solver selection, and training. Furthermore, to enhance the training stage multiple callbacks have already been implemented, such as switching optimizers during training, advanced sampling strategies refinements, and so on.

To show the full capabilities of PINA the next sections will follow the prototypical pipeline for solving a problem, highlighting the various features provided by the software.

3.1 Problem Definition

```python
from pina.problem import SpatialProblem
from pina.operators import grad
from pina.geometry import CartesianDomain
from pina.equation import Equation, FixedValue
from pina import Condition

class SimpleODE(SpatialProblem):

    output_variables = ['u']
    spatial_domain = CartesianDomain({'x': [0, 1]})

    # defining the ode equation
    def ode_equation(input_, output_):
        # computing the derivative
        u_x = grad(output_, input_, components=['u'], d=['x'])
        # extracting u input variable
        u = output_.extract(['u'])
        # calculate residual and return it
        return u_x - u

    # Conditions to hold
    conditions = {
        'x0': Condition(
                        location=CartesianDomain({'x': 0.}),
                        equation=Equation(initial_condition)
                       ),
        'D':  Condition(
                        location=CartesianDomain({'x': [0, 1]}),
                        equation=Equation(ode_equation)
                       ),
    }
```

Figure 2: PINA problem definition.

The first step is the formalization of the problem. In PINA the problem is formulated by constructing a class inheriting from one or more problem classes (at the moment the available classes are **AbstractProblem**, **SpatialProblem**, **TimeDependentProblem**, **ParametricProblem**), depending on the nature of the problem treated. For example, a simple ODE that depends only on a spatial variable is defined via a class inherits only from **SpatialProblem**. Differently, for a parametric time-independent PDE, the problem class inherits from both **SpatialProblem** and **ParametricProblem**. In case the user wants to define its own problem, the **AbstractProblem** interface must be used as the base class. In the problem formulation class, as shown in the snippet of Figure 2, the user must include the information about the domain, the output variables, and the conditions that the neural network has to satisfy. The domain must be a valid domain, e.g. spatial, temporal, or parametric domain, with the corresponding range of variation. Multiple types of geometries are available currently in `PINA` for defining the domain (see Section 3.3). The output variables are represented by a list of symbols constituting the unknowns of the problem. Indeed, standard `PyTorch` tensors carry a label (**LabelTensor**) in `PINA`, allowing the user maximal flexibility for the manipulation of the tensor. Finally, for training PINNs and NOs it is essential to give correct constraints as a form of loss function. The **Condition** class encapsulates all the possible ways the loss could be defined, i.e., physical loss, boundary loss, or data loss. The users must use the **Condition** class to define all the constraints the unknown field needs to satisfy. Moreover, `PINA` already implements functions (e.g. **laplacian** or **grad**) and common equations (e.g. Dirichlet boundary conditions, systems of equations) to ease the problem formulation for the users.

3.2 Model and Solver Selection

Once the model is defined, the user must choose the neural network model to optimize, and the optimization strategy. In `PINA` the model is represented as a standard **torch.nn.Module**. The

Table 1: `PyTorch` models and layers available in `PINA`.

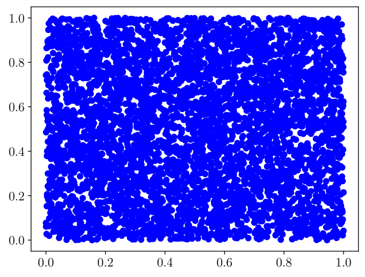
	Method	**Source**
Models	Feed Forward Neural Network (MLP)	-
	Modified MLP	Wang et al. (2021a)
	DeepONet	Lu et al. (2021a)
	MiONet	Jin et al. (2022)
	Fourier Neural Operator (FNO)	Li et al. (2020)
Layers	Residual Layer	Li et al. (2020)
	Fourier Layer	He et al. (2016)
	Continuous Convolution	Coscia et al. (2023b)
	Spectral Convolution	-

package is equipped with many models and layers (see Table 1) already implemented in `PyTorch` and customizable. The user can then decide to use built-in models (e.g. for benchmarking) or build new models and layers for research purposes.

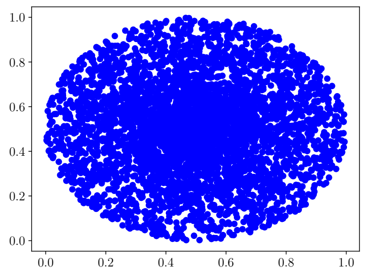
For optimizing the model a specific solver must be used. A solver is a Python object which defines the optimization strategy for the model. In `PINA` the solver is constructed by inheriting from **SolverInterface**, an abstract class wrapping **lightning.pytorch.LightningModule**. Available solvers include a supervised learning solver (**SupervisedSolver**), particularly crafted for NO problems, a physics-informed solver (**PINN**) (Raissi et al., 2019), and an adversarial solver (**GAROM**) (Coscia et al., 2023a). We plan to continuously add solvers as the state–of–the–art evolves. Notice that all solvers are customizable by the user. For example, the **PINN** solver allows changing the loss (e.g. using a variational loss (Kharazmi et al., 2019)), or extending the solver with regularization strategies (Yu et al., 2022), or modifying the optimizer (Davi & Braga-Neto, 2022). All of these, apparently different solvers, can be changed by a keyword argument in the **PINN** class.

3.3 Data Generation

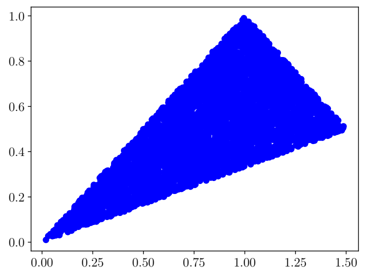
NO learning procedure uses a finite set of observations and it is trained in a fully supervised manner. These observations, obtained by numerical solver solutions, in `PINA` can be passed as **LabelTensor** in the **Condition** class defined in Section 3.1. Differently, some training strategies, e.g. PINNs, use collocation points sampled inside the domain where the residual of the differential equation (see equation 3) must be evaluated. For these types of solvers, `PINA` provides a simple sampling strategy for multiple different geometries. In `PINA` each domain is a **Location** object, which defines the geometry of the domain. There are already multiple sampling methods in `PINA` (random uniform, grid sampling, latin hypercube sampling, Chebyshev sampling) for the different available multidimensional geometries (**CartesianDomain**, **EllispoidDomain**, and so on) as depicted in Figure 3. In addition to multidimensional geometries, the software also provides set operations (difference,



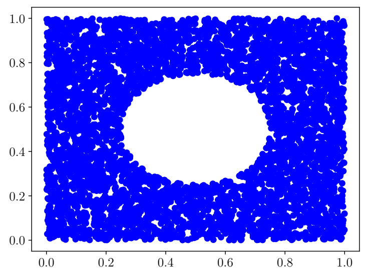
(a) Cartesian domain samples



(b) Circle domain samples



(c) Simplex domain samples



(d) Difference of cartesian and circular domain

Figure 3: Examples of possible domain generated using PINA.

```
from pina.solvers import PINN
from pina.trainer import Trainer
from pina.callbacks import MetricTracker, R3Refinement
from pina.model import FeedForward

# define the problem
problem = SimpleODE()
# choose model + solver
model = FeedForward(input_dimensions=len(problem.input_variables),
                    output_dimensions=len(problem.output_variables),
                    n_layers=3,
                    inner_size=120,
                    func=torch.nn.Tanh)
solver = PINN(problem=problem, model=model)
# sampling strategy
problem.discretise_domain(n=1000)
# training the model
trainer = Trainer(solver=solver, device = 'gpu', callbacks=[MetricTracker(), R3Refinement(sample_every=10)])
trainer.train()
# saving model
trainer.save_checkpoint(filepath='saveingfilefolder/')
```

Figure 4: PINA snippet for the solution of a generic problem: problem and model definition, creation of sampling points, training procedure and saving to file.

union, intersection, and so on) allowing the user to build a custom domain. Finally, in **Condition** class the user can also employ available scatter points, and pass them as **LabelTensor**s.

3.4 PINA Training

The last stage on the PINA pipeline is to train the model. This is done using the **Trainer** class, which wraps the **lightning.pytorch.Trainer** class. In the **Trainer** class the user must pass a **SolverInterface** object in addition to all the available arguments of **lightning.pytorch.Trainer**. This strategy allows the user maximal training flexibility by exploiting fully **PytorchLightning** capabilities, e.g. low precision training, gradient accumulation, multiple GPU training, and different hardware training. Figure 4 reports the complete pipeline for training a PINA model. The problem is defined by instantiating the problem class. A model is then constructed, in this case, an MLP of 3 layers of size 120, with hyperbolic tangent activation. The solver is instantiated using the problem and the model, in this case, the PINN solver is used. Finally the trainer is created using the solver and extra **PytorchLightning** arguments, e.g. **device** and **callbacks**. In this case, the trainer will track the metric using the built-in **MetricTracker**, so no logging will be used, which is pretty convenient for assessing quickly model results. In addition, adaptive $R3$ refinement (Daw et al., 2023) is each 10 iterations to update the collocation points.

4 EXPERIMENTS

In this section, we show possible benchmark results obtainable with `PINA`. We want to highlight that the purpose of this section is not to provide accurate measurements of model performance, but rather to show how easily is to benchmark on `PINA`. As model cases, we use four different models (see Appendix B for model specifics):

- **MLP**: A standard Feed Forward Neural Network implemented in `PINA`
- **m-MLP**: A modified Feed Forward Neural Network implemented in `PINA`, with skip connection (Wang et al., 2021a)
- **hard-MLP**: A standard Feed Forward Neural Network implemented in `PyTorch` with hard constraint (Lu et al., 2021c) on the spatial dimensions for boundary conditions
- **DeepONet**: A DeepONet Network (Lu et al., 2021a) implemented in `PINA`.

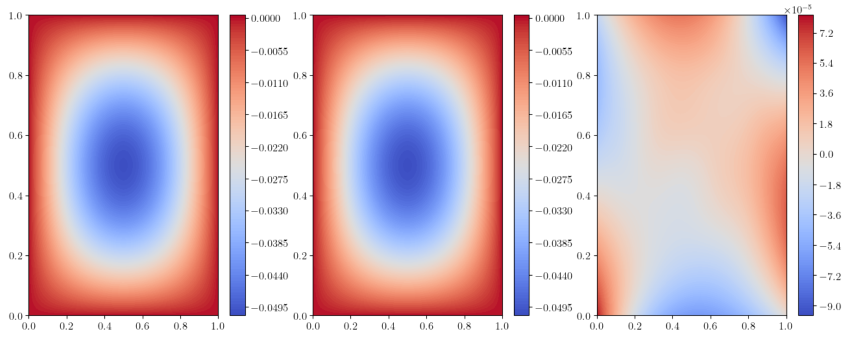


Figure 5: Example of visualization API for the Poisson problem in `PINA`. *Left*: `PINA` solution, *center*: real solution, *right*: absolute value difference of real and predicted solution.

We test on four different problems (see Appendix A for mathematical formulation) with different PINN's learning methodologies:

- **Burger**: The one dimensional time dependent Burgers's equation, with classical PINN learning
- **Poisson**: The two dimensional Poisson's equation, with extra feature PINN learning (Demo et al., 2023)
- **Wave**: The two dimensional time dependent Wave's equation, with $R3$ adaptive refinement PINN learning (Daw et al., 2023)
- **Parametric Poisson**: The two dimensional parametric dependent Poisson's equation, with classical PINN learning (Demo et al., 2023).

In table 2 the final loss for all the simulations done employing `PINA` are reported. It is worth mentioning that all simulations have been done by changing just a few lines of code (the problem class, and model definition), which shows the great versatility of the software. Finally, in Figure 5 we show how solutions can be visualized in `PINA` via the software plotting API with the Poisson problem example.

Table 2: Benchmark results for multiple problems and training model in PINA.

Model	**Burger**	**Poisson**	**Wave**	**Parametric Poisson**
MLP	6.20×10^{-4}	1.87×10^{-7}	1.02×10^{-3}	8.13×10^{-5}
m-MLP	4.60×10^{-4}	2.30×10^{-7}	1.71×10^{-4}	6.91×10^{-6}
hard-MLP	9.55×10^{-4}	1.67×10^{-6}	4.64×10^{-4}	2.95×10^{-4}
DeepONet	2.49×10^{-2}	5.71×10^{-7}	2.02×10^{-2}	5.66×10^{-3}

5 Conclusions

We present in this contribution a software framework for PINN and NO learning. `PINA` is an emerging package that wants to centralize the research activities related to these methodologies, making its application to the production environment or improvements with respect to the status quo easier. We introduced the most important features, highlighting the modular structure, the `PyTorch` and `PyTorchLighting` inheritance, the extensibility for defining problems and domains, the capability to use several built-in models or crafting from scratch a new one. We showed how `PINA` can be used to solve different problems, using benchmarking cases. Such frameworks open the door to faster development in the field, reducing the required effort to actively apply such techniques even to complex problems, and fostering new implementations and innovations.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Kaushik Bhattacharya, Bamdad Hosseini, Nikola B Kovachki, and Andrew M Stuart. Model reduction and neural networks for parametric pdes. *The SMAI journal of computational mathematics*, 7:121–157, 2021.

Johannes Brandstetter, Daniel Worrall, and Max Welling. Message passing neural pde solvers. *arXiv preprint arXiv:2202.03376*, 2022.

Feiyu Chen, David Sondak, Pavlos Protopapas, Marios Mattheakis, Shuheng Liu, Devansh Agarwal, and Marco Di Giovanni. Neurodiffeq: A Python package for solving differential equations with neural networks. *Journal of Open Source Software*, 5(46):1931, 2020. doi: 10.21105/joss.01931.

Tianping Chen and Hong Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE transactions on neural networks*, 6(4):911–917, 1995.

Dario Coscia, Nicola Demo, and Gianluigi Rozza. Generative adversarial reduced order modelling. *arXiv preprint arXiv:2305.15881*, 2023a.

Dario Coscia, Laura Meneghetti, Nicola Demo, Giovanni Stabile, and Gianluigi Rozza. A continuous convolutional trainable filter for modelling unstructured data. *Computational Mechanics*, pp. 1–13, 2023b.

Caio Davi and Ulisses Braga-Neto. Pso-pinn: Physics-informed neural networks trained with particle swarm optimization. *arXiv preprint arXiv:2202.01943*, 2022.

Arka Daw, Jie Bu, Sifan Wang, Paris Perdikaris, and Anuj Karpatne. Mitigating propagation failures in physics-informed neural networks using retain-resample-release (r3) sampling. 2023.

Nicola Demo, Maria Strazzullo, and Gianluigi Rozza. An extended physics informed neural network for preliminary analysis of parametric optimal control problems. *Computers & Mathematics with Applications*, 143:383–396, 2023.

William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019. URL `https://github.com/Lightning-AI/lightning`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Pengzhan Jin, Shuai Meng, and Lu Lu. Mionet: Learning multiple-input operators via tensor product. *SIAM Journal on Scientific Computing*, 44(6):A3490–A3514, 2022.

Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. Variational physics-informed neural networks for solving partial differential equations. *arXiv preprint arXiv:1912.00873*, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alexander Koryagin, Roman Khudorozkov, and Sergey Tsimfer. Pydens: A Python framework for solving differential equations with neural networks. *arXiv preprint arXiv:1909.11544*, 2019. doi: 10.48550/arXiv.1909.11544.

Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces. *arXiv preprint arXiv:2108.08481*, 2021.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Zongyi Li, Hongkai Zheng, Nikola Kovachki, David Jin, Haoxuan Chen, Burigede Liu, Kamyar Azizzadenesheli, and Anima Anandkumar. Physics-informed neural operator for learning partial differential equations. *arXiv preprint arXiv:2111.03794*, 2021.

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

Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. Deepxde: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021b. doi: 10.1137/19m1274067.

Lu Lu, Raphael Pestourie, Wenjie Yao, Zhicheng Wang, Francesc Verdugo, and Steven G Johnson. Physics-informed neural networks with hard constraints for inverse design. *SIAM Journal on Scientific Computing*, 43(6):B1105–B1132, 2021c.

Levi McClenny and Ulisses Braga-Neto. Self-adaptive physics-informed neural networks using a soft attention mechanism. *arXiv preprint arXiv:2009.04544*, 2020.

Levi D McClenny, Mulugeta A Haile, and Ulisses M Braga-Neto. Tensordiffeq: Scalable multi-gpu forward and inverse solvers for physics informed neural networks. *arXiv preprint arXiv:2103.16034*, 2021. doi: 10.48550/arXiv.2103.16034.

Keith W Morton and David Francis Mayers. *Numerical solution of partial differential equations: an introduction*. Cambridge university press, 2005.

Mohammad Amin Nabian, Rini Jasmine Gladstone, and Hadi Meidani. Efficient training of physics-informed neural networks via importance sampling. *Computer-Aided Civil and Infrastructure Engineering*, 36(8):962–977, 2021.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Wei Peng, Jun Zhang, Weien Zhou, Xiaoyu Zhao, Wen Yao, and Xiaoqian Chen. Idrlnet: A physics-informed neural network library. *arXiv preprint arXiv:2107.04320*, 2021. doi: 10.48550/arXiv.2107.04320.

Alfio Quarteroni and Silvia Quarteroni. *Numerical models for differential problems*, volume 2. Springer, 2009.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. 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, 2019.

Chuwei Wang, Shanda Li, Di He, and Liwei Wang. Is l^2 physics informed loss always suitable for training physics informed neural network? *Advances in Neural Information Processing Systems*, 35:8278–8290, 2022.

Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021a.

Sifan Wang, Hanwen Wang, and Paris Perdikaris. Learning the solution operator of parametric partial differential equations with physics-informed deeponets. *Science advances*, 7(40):eabi8605, 2021b.

Chenxi Wu, Min Zhu, Qinyang Tan, Yadhu Kartha, and Lu Lu. A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 403:115671, 2023.

Jeremy Yu, Lu Lu, Xuhui Meng, and George Em Karniadakis. Gradient-enhanced physics-informed neural networks for forward and inverse pde problems. *Computer Methods in Applied Mechanics and Engineering*, 393:114823, 2022.

A APPENDIX A

In this section, we provide the mathematical formulations of the problem presented in the experiment section 4.

A.1 BURGER'S EQUATION

Burger's equation is a convection-diffusion equation widely used in many fields of mathematics. The problem is crafted as the benchmark presented in (Raissi et al., 2019). Let $\boldsymbol{x} = (x, t)$ be the spatio-temporal variables, and u be the unknown field. The Burger equation is:

$$\begin{cases} \frac{\partial}{\partial t}u(\boldsymbol{x}) + u(\boldsymbol{x})\frac{\partial}{\partial x}u(\boldsymbol{x}) - \frac{0.01}{\pi}\frac{\partial^2}{\partial x^2}u(\boldsymbol{x}) = 0 & x \in [-1,1]\ , t \in [0,1] \\ u(1,t) = u(-1,t) = 0 & t \in [0,1] \\ u(x,0) = -\sin(\pi x) & x \in [-1,1]. \end{cases} \tag{5}$$

For solving the equation we sample 10000 points uniformly random in the domain $[-1, 1] \times [0, 1]$.

A.2 POISSON'S EQUATION

Poisson's equation is an elliptic partial differential equation widely used in physics. The problem is crafted as the benchmark presented in (Demo et al., 2023). Let $\boldsymbol{x} = (x, y)$ be the spatial variables, u be the unknown field, and $\Omega = [-1, 1]^2$ the domain. The Poisson equation is:

$$\begin{cases} \nabla^2 u(\boldsymbol{x}) = \sin(\pi x)\sin(\pi y) & \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}) = 0 & \boldsymbol{x} \in \partial\Omega, \end{cases} \tag{6}$$

where $\partial\Omega$ indicates the boundary of the domain, and the Laplacian operator ∇^2 acts on the spatial variables. For solving the equation we sample 10000 points uniformly random in the domain Ω. During problem learning we employ extra features, by augmenting the input with the forcing term, i.e. the model input is given by $(x, y, \sin(\pi x)\sin(\pi y))$.

A.3 Wave's equation

The Wave's Equation is a linear differential equation vastly used in fluid dynamics. Let $\boldsymbol{x} = (x, y, t)$ be the spatio-temporal variables, u be the unknown field, $\Omega = [0,1]^2$ the domain, and $\mathbb{T} = [0,1]^2$ the parameter domain. The Wave equation is:

$$\begin{cases} \nabla^2 u(\boldsymbol{x}) = \frac{\partial^2}{\partial t^2} u(\boldsymbol{x}) & \boldsymbol{x} \in \Omega \times \mathbb{T} \\ u(\boldsymbol{x}) = 0 & \boldsymbol{x} \in \partial\Omega \times \mathbb{T}, \\ u(\boldsymbol{x}) = \sin(\pi x)\sin(\pi y) & \boldsymbol{x} \in \Omega \times \partial\mathbb{T}, \end{cases} \tag{7}$$

where $\partial\Omega$ indicates the boundary of the domain, and the Laplacian operator ∇^2 acts on the spatial variables. For solving the equation we sample 10000 points uniformly random in the domain Ω. We use $R3$ adaptive refinement for moving the collocation points during training every 100 epochs.

A.4 Parametric Poisson's equation

Parametric Poisson's equation is an example of a Poisson equation where the forcing term depends on external parameters. The problem is crafted as the benchmark presented in (Demo et al., 2023), where the objective is to learn a function for different parameters. The problem can be considered as a NO problem since we map different initial functions (for different parameters) to the field functions. In the experiment section, we use PINN learning to solve the problem. Let $\boldsymbol{x} = (x, y)$ be the spatial variables, u be the unknown field, $\Omega = [0,1]^2$ the domain, and $\Xi = [-1,1]^2$ the parameter domain. The Poisson equation is:

$$\begin{cases} \nabla^2 u(\boldsymbol{x}) = e^{-2[(x-\xi_1)^2 + (y-\xi_2)^2]} & \boldsymbol{x} \in \Omega \times \Xi \\ u(\boldsymbol{x}) = 0 & \boldsymbol{x} \in \partial\Omega \times \Xi, \end{cases} \tag{8}$$

where $\partial\Omega$ indicates the boundary of the domain, and the Laplacian operator ∇^2 acts on the spatial variables. For solving the equation we sample 10000 points uniformly random in the domain Ω.

B Appendix B

In this section, we provide the network specifics for the experiments performed in Section 4. All the models were trained using the Adam optimizer (Kingma & Ba, 2014), with learning rate of 0.001 for 10000 epochs minimizing the mean square error loss. The training was done on Intel CPU.

B.1 Burger's equation

The networks' composition:

- **MLP**: Three linear layers of size $[20, 10, 5]$ with hyperbolic tangent activation on all layers except the last
- **m-MLP**: Three linear layers of size $[20, 20, 20]$ with hyperbolic tangent activation on all layers except the last. The transformer networks were two linear layers mapping the input to the inner size of 20
- **hard-MLP**: Same as **MLP**. Hard constraints on boundary conditions are imposed by multiplying the network output with the term $(1 + x)(1 - x)$
- **DeepONet**: The branch and trunk net are the same architecture of two linear layers of size $[20, 20]$ with hyperbolic tangent activation on all layers except the last. The reduction is done by aggregating with a linear layer with input dimension 20 and output dimension 1. The trunk net takes t as input. The branch net takes x as input.

The input dimension of the problem is 2 (one spatial + one temporal variables), and the output dimension is 1.

B.2 Poisson's equation

The networks' composition:

- **MLP**: Same architecture as Burgers's problem.
- **m-MLP**: Same architecture as Burgers's problem.
- **hard-MLP**: Same architecture as Burgers's problem.
- **DeepONet**: Same architecture as Burgers's problem, but the trunk net takes x, t as input. The branch net takes $\sin(\pi x)\sin(\pi y)$ as input.

The input dimension of the problem is 3 (two spatial + augmentation variables), and the output dimension is 1.

B.3 Wave's equation

The networks' composition:

- **MLP**: Same architecture as Burgers's problem.
- **m-MLP**: Same architecture as Burgers's problem.
- **hard-MLP**: Same architecture as Burgers's problem.
- **DeepONet**: Same architecture as Burgers's problem, but the trunk net takes t as input. The branch net takes x, y as input.

The input dimension of the problem is 3 (two spatial + two parametric variables), and the output dimension is 1.

B.4 Parametric Poisson's equation

The networks' composition:

- **MLP**: Same architecture as Burgers's problem.
- **m-MLP**: Same architecture as Burgers's problem.
- **hard-MLP**: Same architecture as Burgers's problem.
- **DeepONet**: Same architecture as Burgers's problem, but the trunk net takes x, t as input. The branch net takes ξ_1, ξ_2 as input.

The input dimension of the problem is 4 (two spatial + two parametric variables), and the output dimension is 1.