Quest Journals
Journal of Research in Applied Mathematics
Volume 11 ~ Issue 5 (May 2025) pp: 100-115
ISSN (Online): 2394-0743 ISSN (Print): 2394-0735
www.questjournals.org

Review Paper

Using Physics-Informed Transformers to Solve Nonlinear Partial Differential Equations in Applications of Physics and Modern Engineering

Dr. Adel Ahmed Hassan Kubba[1], Ragab Mostafa Mohamed Abdelbar[2]
University Nile Vally, Department of Graduate Studies, PhD student in mathematics at University Nile Vally.

Abstract :
Solving nonlinear partial differential equations (PDEs) is fundamental to many applications in physics and modern engineering, ranging from fluid dynamics and electromagnetism to structural mechanics and quantum mechanics. Traditional numerical methods, such as finite element and finite difference methods, often face computational limitations, particularly for high-dimensional and complex systems. Recent advancements in deep learning have introduced physics-informed neural networks (PINNs) as an alternative approach, but they still struggle with capturing long-range dependencies efficiently.
This paper explores the integration of transformer architectures into physics-informed learning frameworks, leveraging their self-attention mechanism to enhance the accuracy and efficiency of solving nonlinear PDEs. Physics-Informed Transformers (PITs) embed physical laws into the model architecture, allowing them to learn solution operators effectively while maintaining interpretability and generalization across different boundary conditions. By incorporating inductive biases from physics and optimizing training through self-supervised learning, PITs demonstrate superior performance compared to traditional PINNs in solving complex PDEs with improved computational efficiency and accuracy.
We present numerical experiments on benchmark problems, including the Navier-Stokes equations, nonlinear wave equations, and reaction-diffusion systems, illustrating the potential of PITs in accelerating scientific computing. The results highlight the transformative role of transformer-based models in engineering and physics applications, paving the way for more robust and scalable solvers for nonlinear PDEs.

Keywords :
Physics-Informed Transformers (PITs), Nonlinear Partial Differential Equations (PDEs), Scientific Machine Learning, Deep Learning, Physics-Informed Neural Networks (PINNs), Navier-Stokes Equations, Reaction-Diffusion Systems, Transformer Networks .

*Received 11 May., 2025; Revised 20 May., 2025; Accepted 22 May., 2025 © The author(s) 2025. Published with open access at **www.questjournas.org***

(1) Introduction

Nonlinear partial differential equations (PDEs) play a crucial role in modeling various physical and engineering phenomena, including fluid dynamics, heat transfer, electromagnetism, structural mechanics, and quantum mechanics. However, solving these equations analytically is often infeasible due to their inherent complexity, necessitating numerical methods such as finite difference, finite element, and spectral methods. While these conventional approaches have been widely used, they are computationally expensive, especially for high-dimensional problems or real-time applications.
In recent years, deep learning techniques have emerged as promising alternatives for solving PDEs, with Physics-Informed Neural Networks (PINNs) gaining significant attention. PINNs incorporate physical constraints directly into neural network architectures, enabling data-efficient learning and improved generalization. However, conventional PINNs often struggle with long-range dependencies, high-frequency solution components, and training instability, limiting their effectiveness in solving complex nonlinear PDEs.

To address these challenges, we propose the use of Physics-Informed Transformers (PITs), leveraging the transformer architecture's self-attention mechanism to enhance the learning of PDE solutions. Transformers have revolutionized natural language processing by capturing global dependencies efficiently, and their application in scientific computing is a promising frontier. By integrating physics-informed constraints with transformer networks, PITs offer improved accuracy, faster convergence, and better generalization to diverse boundary conditions.
This study explores the potential of PITs in solving nonlinear PDEs arising in various fields of physics and modern engineering. We demonstrate their effectiveness through benchmark problems, such as the Navier-Stokes equations for fluid dynamics, reaction-diffusion systems for chemical processes, and wave equations for electromagnetics. Our findings indicate that PITs can significantly enhance computational efficiency while preserving physical consistency, paving the way for advanced AI-driven solvers for complex engineering and physics applications.

(2) Background

In this section we give a short review of a physics-informed neural network (PINN) and deep operator network (DeepONet) which are the basis of the algorithms and models implemented in PinnDE.

(2.1) Physics-Informed Neural Networks

Physics-Informed Neural Networks (PINNs) were first introduced in [5] and were later popularized when re-introduced in [10]. The general idea consists of taking a deep neural network as a surrogate approximation for the solution of a system of differential equations. This has the advantage that derivatives of the neural network approximation to the solution of the system of differential equations can be computed with automatic differentiation [2] rather than relying on numerical differentiation as is typically the case for standard numerical methods such as finite difference, finite volume or finite element methods. As a consequence, physics-informed neural networks are truly meshless methods as the derivative computations can be done in single points, without relying on the introduction of a computational mesh as is the case for most standard numerical methods. The neural network surrogate solution is being obtained by solving an optimization problem that involves fitting the weights and biases of the network. This is done such that at a collection of finitely many (typically randomly chosen) collocation points a loss function combining the differential equation along with any supplied initial and/or boundary conditions is minimized. We make this general procedure more precise in the following.

We consider a general initial-boundary value problem for a system of L partial differential equations over a spatio-temporal domain $[t_0, t_f\,] \times \Omega$, where $\Omega \subset R^d, t_0 \in R$ denotes the initial integration time, and $t_f \in R$ denotes the final integration time, given by

$$\Delta^l\big(t, x, u_{(n)}\big) = 0, l = 1, \ldots, L\,,\ t \in [t_0, t_f\,], x \in \Omega, \tag{1a}$$
$$I^{l_i}\big(\,x, u_{(n_i)}|_{t=t_0}\big) = 0, l_i = 1, \ldots, L_i\,, x \in \Omega, \tag{1b}$$
$$B^{l_b}\big(t, x, u_{(n_b)}\big) = 0, l_b = 1, \ldots, L_b\,,\ t \in [t_0, t_f\,], x \in \partial\Omega, \tag{1c}$$

where t denotes the time variable and $x = (x_1, \ldots, x_d)$ denotes the tuple of spatial independent variables. The dependent variables are denoted by $u = (u^1, \ldots, u^q)$, and u(n) denotes the tuple of all derivatives of the dependent variable with respect to both t and x up to order n . The initial conditions are represented through the initial value operator $I = (I^1, \ldots, I^{L_i}\,)$, and similarly the boundary conditions are included using the boundary value operator $B = (B^1, \ldots, B^{L_b}\,)$.
We denote a deep neural network as N^θ with parameters θ, which includes all the weights and biases of all layers of the neural network. The goal of a physics-informed neural network is to learn to interpolate the global solution of the system of differential equations over $[t_0, t_f\,] \times \Omega$ as the parameterization $u^\theta = N^\theta(t, x)$, where $u^\theta(\mathrm{t}, \mathrm{x}) \approx \mathrm{u}(\mathrm{t}, \mathrm{x})$. This is done by minimizing the loss function

$$L(\theta) = L_\Delta(\theta) + \gamma_i L_i(\theta) + \gamma_b L_b(\theta)\ (2)$$

where $\gamma_i\,, \gamma_b \in \mathrm{R}^+$ are weighting parameters for the individual loss contributions, and we composite the differential equation, initial value, and boundary value loss, respectively, which are given by

$$L_\Delta(\theta) = \frac{1}{N_\Delta}\sum_{i=1}^{N_\Delta}\sum_{l=1}^{L} |\Delta^l(\,t_\Delta^i, x_\Delta^i, u_{(n)}^\theta(t_\Delta^i, x_\Delta^i)|^2\,, \tag{3a}$$

$$L_\mathrm{i}(\theta) = \frac{1}{N_\mathrm{i}}\sum_{i=1}^{N_\mathrm{i}}\sum_{l_\mathrm{i}=1}^{L_\mathrm{i}} |\mathrm{I}^{l_\mathrm{i}}(\,t_\mathrm{i}^i, x_\mathrm{i}^i, u_{(n_\mathrm{i})}^\theta(t_\mathrm{i}^i, x_\mathrm{i}^i)|^2\,, \tag{3b}$$

$$L_b(\theta) = \frac{1}{N_b}\sum_{i=1}^{N_b}\sum_{l_b=1}^{L_b} |B^{l_b}(\,t_b^i, x_b^i, u_{(n_b)}^\theta(t_b^i, x_b^i)|^2\,, \tag{3c}$$

where $L_\Delta(\theta)$ corresponds to the differential equation loss based on Eqn. (1a) , $L_i(\theta)$ is the initial value loss stemming from Eqn. (1b), and $L_b(\theta)$ denotes the boundary value loss derived from Eqn. (1c) . The neural network evaluates the losses over a set of collocation points, where we have $\{(t_\Delta^i, x_\Delta^i)\}_{i=1}^{N_\Delta}$ for the system, $\{(t_\mathrm{i}^i, x_\mathrm{i}^i)\}_{i=1}^{N_\mathrm{i}}$ for the initial values, and $\{(t_b^i, x_b^i)\}_{i=1}^{N_b}$ for the boundary values, with N_Δ, N_i and N_b denoting the number of differential equation, initial condition and boundary condition collocation points, respectively.

This method is considered soft constrained, as the network is forced to learn the initial and boundary conditions and composite their loss, which is described in [10]. A downside of this approach is that while the initial and boundary values are known exactly, they will in general not be satisfied exactly by the learned neural network owing to being enforced only through the loss function. An alternative strategy is to hard-constrain the network with the initial and/or boundary conditions, following what was first proposed in [5], which structures the neural network itself in a way so that the output of the network automatically satisfies the initial and/or boundary conditions. As such, only the differential equation loss then has to be minimized and the initial and boundary loss components in (2) are not required. The general idea of hard-constraining the initial and boundary conditions is hence to design a suitable ansatz for the neural network surrogate solution that makes sure that the initial and boundary conditions are exactly satisfied for all values of the trainable neural network. The precise form of the class of suitable hard constraints depends on the particular form of the initial and boundary value operators. We show some specific implementations in Section 3.1.1. A more formalized discussion can also be found in [12]. For many problems, it has been shown that hard constraining the initial and boundary conditions improves the training performance of neural network based differential equations solvers, and leads to lower overall errors in the obtained solutions of these solvers, cf. [12]. One notable exception are cases where the solution of a differential equation is not differentiable everywhere, in which case soft constraints typically outperform hard constraints, see e.g. [11].

Above we have introduced physics-informed neural networks for systems of partial differential equations for general initial-boundary value problems. The same method can also be applied for ordinary differential equations. For ordinary differential equations, both initial value problems and boundary value problems can be considered.

We first consider a system of L ordinary differential equations over the temporal domain $[t_0, t_f\ ]$,

$$\Delta^l(t,\ u_{(n)}) = 0\ , \qquad l = 1, \dots, L\ , \qquad t \in [t_0, t_f\ ] \tag{4}$$

where $u_{(n)}$ are the total derivatives with respect to t , and Δ^l are $(m+1)-th$ order differential functions, with initial conditions,

$$\mathrm{I}^{l_i}(u_{(ni)}|_{t=t_0}) = 0\ , \qquad l_i = 1, \dots, L_i \tag{5}$$

where the associated loss function for the physics-informed neural network is represented as the composite loss of Eqn. (3a) and Eqn. (3b), with Δ^l and I^{l_i} corresponding to Eqn. (4) and Eqn. (5).

One can also consider a boundary value problem with the boundary conditions

$$B^{l_b}(t,\ u_{(n_b)}) = 0\ , \qquad l_b = 1, \dots, L_b\ , \qquad t \in [t_0, t_f\ ] \tag{6}$$

in which case the physics-informed loss function is represented as the composite loss of Eqn. (3a) and Eqn. (3c), with Δ^l and B^{l_b} corresponding to Eqn. (4) and Eqn. (6).

Hard constraining these networks follows a similar procedure as described above, with the initial/boundary values being exactly enforced for all values of the neural network surrogate solution.

We refer to Section 3.1.1 for further details.

(2.2) Deep Operator Networks

Deep operator networks (DeepONets) were first introduced in [7] based on theoretical results of [14].This idea replaces learning of a particular solution of the system of differential equations with learning the solution operator acting on a specific initial‑boundary condition for the system of differential equations (1). That is, let $G(u_0)(t,x)$ be a solution operator for (1) acting on the particular initial condition u0 yielding the solution of the the initial‑boundary value at the spatio-temporal point (t,x), i.e. $u(t,x) = G(u_0)(t,x)$. That is, in comparison to standard physics-informed neural networks, deep operator networks require two inputs, the spatio-temporal point where the solution should be evaluated (this is the same input as for physics-informed neural networks), and the particular initial (or boundary) condition to which the solution operator should be applied. Since the initial condition is a function, it has to first be sampled on a finite dimensional subspace to be included into the neural network. The most prominent way to accomplish this is to sample a finite collection of Ns sensor points $\{x_i\}_{i=1}^{N_s}$, $x_i \in \Omega$ and evaluate the initial condition at those sensor points, yielding the tuple $(u_0(x_1),\ldots,u_0(x_{N_s}))$ that is used as an input for the deep operator network. A similar strategy is followed for pure boundary value problems. A deep operator network then uses two separate networks to combine the sampled initial conditions and independent variables. The branch network processes the initial condition sampled values, and the trunk network processes the independent variables. In the vanilla deep operator network the output vectors of the two sub-networks are then combined via a dot product. The output of deep operator network can be expressed as

$$\mathcal{G}^{\theta}\big(u_0(x)\big)(t,x) = \mathcal{B}^{\theta}\left(u_0(x_1),\ldots,u_0\big(x_{N_s}\big)\right)\cdot \mathcal{T}^{\theta}(t,x)$$

With $\mathcal{B}^{\theta} = \left(\mathcal{B}_1^{\theta},\ldots.,\mathcal{B}_p^{\theta}\right)$ representing the output of the branch net and $\mathcal{T}^{\theta} = \left(\mathcal{T}_1^{\theta},\ldots.,\mathcal{T}_p^{\theta}\right)$ representing the output of the trunk net, and p being a hyper-parameter. The loss function associated with these networks is the same as (2) when soft constrained. Analogously to physics-informed neural networks, also deep operator networks can be hard-constrained with either standard PINN hard constraining methods, or further modified methods designed for deep operator networks, cf. [12]. A trained deep operator network thus is a solution interpolant for the solution operator, i.e. $\mathcal{G}^{\theta} \approx G$.

These networks come with multiple main benefits. Firstly, these networks can easily replace one initial condition with a new condition and do not require re-training. Secondly, as presented in [13], since these networks learn the solution operator corresponding to an initial condition, they can readily be used for time-stepping and thus extend to long time intervals and maintain the ability to further predict the solution, unlike a standard PINN, which typically fails if $t_f \gg 0$.

(3) Methods

In this work, we aim to learn the operator $G_\theta : A \to U$, where A is our input function space, U is our solution function space, and θ are the learnable model parameters. We use a combination of novel equation tokenization and numerical method-like updates to learn model operators G_θ. Our novel equation tokenization and embedding method is described first, followed by a detailed explanation of the numerical update scheme.

(3.1) Equation tokenization

In order to utilize the text view of our data, the equations must be tokenized as input to our transformer. Following Lampe and Charton , each equation is parsed and split into its constituent symbols. The tokens are given in table 1. The delimiter marks—decimal points for numerical values, commas for separating numerical values, and ampersand for separating equations, sampled values, and boundary conditions—are also added. The 1D equations are tokenized so the governing equation, forcing term, initial condition, sampled values, and output simulation time are all separated because each component controls distinct properties of the system. The 2D equations are tokenized so that the governing equations remain intact because some of the governing equations, such as the continuity equation, are self-contained. All of the tokens are then compiled into a single list, where each token in the tokenized equation is the index at which it occurs in this list. For example, we have the following tokenization:

$$\begin{aligned}\frac{\partial}{\partial t}u(x,t) &= Derivative(u(x,t),t)\\ &= [Derivative, (, u, (, x, ,, t,), ,, t,)]\\ &= [6, 0, 3, 15, 0, 16, 33, 14, 1, 33, 1]\end{aligned}$$

After each equation has been tokenized, the target time value is appended in tokenized form to the equation, and the total equation is padded with a placeholder token so that each text embedding is the same

Table 1. Collection of all tokens used in tokenizing governing equations, sampled values, and system parameters.

Category	Available tokens
Equation	$(,), \partial, \Sigma, j, Aj, lj, \omega j, \phi j, sin, t, u, x, y, +, -, *, /$
Boundary condition	*Neumann, Dirichlet, None*
Numerical	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\hat{}, E, e$
Delimiter	$, (comma), . (decimal\ point)$
Separator	&
2D equations	$\nabla, =, \Delta, \cdot\ (dot\ product)$

length. Sampled values are truncated at 15 digits of precision. Data handling code is adapted from PDE Bench .

(3.2) PITT

The PITT utilizes tokenized equation information to construct an update operator F_P, similar to numerical integration techniques: $x_{t+1} = x_t + F_P(x_t)$. We see in figure 1, PITT takes in the numerical values and grid spacing, similar to operator learning architectures such as FNO, as well as the tokenized equation and the explicit time differential between simulation steps. The tokenized equation is passed through a Multi Head Attention block seen in figure 1(a). In our case we use self attention (SA) . The tokens are shifted and scaled to be between -1 and 1 upon input, which significantly boosts performance. This latent equation representation is then used to construct the keys and queries for a subsequent Multi Head Attention block that is used in conjunction with output from the underlying neural operator to construct the update values for the final input frame. The time difference between steps is encoded, allowing use of arbitrary timesteps. Intuitively, we can view the model as using a neural operator to passthrough the previous state, as well as calculate the update, like in numerical methods. The tokenized information is then used to construct an analytically driven update operator that acts as a correction to the neural operator state update. This intuitive understanding of PITT is explored with our 1D benchmarks.

Two different embedding methods are used for the tokenized equations. In the first method, the token attention block first embeds the tokens, T, as key, query, and values with learnable weight matrices :

$T_1 = W_{T_1}T, T_2 = W_{T2}T, T_3 = W_{T3}T$. While this approach introduces unconventional correlations between tokens, only numerical values are modified in many of our experiments, and so the correlation between numerical values is useful. The second method uses standard fixed positional encoding [15] and lookup table embedding.

The standard approach does not introduce unconventional correlations between numerical values. Dropout and Multi-head SA are then used to compute a hidden representation:

$T_h = Dropout(SA(T_1, T_2, T_3))$. We use a single layer of self-attention for the tokens. The update attention blocks seen in figure 1(b) then uses the token attention block output as queries and keys, the neural operator output as values, and embeds them using trainable matrices as $V_0 = W_x X_0, T_{h1} = W_{Th1}T_h, T_{h2} = W_{Th2}T_h$

The output is passed through a fully connected projection layer to match the target output dimension. This update scheme mimics numerical methods and is given in algorithm 1.

Algorithm 1. PITT numerical update scheme.

```
Require : $V_0, T_{h1}, T_{h2}, time\ t, L\ layers$
        $For\ l = 1,2,3, \dots, L\ do$
        $X_l \leftarrow Dropout\ (LA(T_{h1}, T_{h2}, V_{l-1}))$
        $t_l \leftarrow MLP\ (\frac{l-t}{L})$
        $V_l \leftarrow V_{l-1} + MLP([X_l, t_l])$
        End for
```

A standard, fully connected multi-layer perceptron is used to calculate the update after concatenating the attention output with an embedding of the fractional timestep. This block uses softmax-free LA, computed as $\boldsymbol{z} = Q(\tilde{K}^T\tilde{V}/n. \tilde{K}$ and $\tilde{V}$ indicate instance normalization. Note that the target time t is incremented fractionally to more closely model numerical method updates. Using multiple update layers and a fractional timestep is useful for long target times, such as steady-state or fixed-future type experiments.

Using a single update layer works will with small timesteps, such as predicting the next simulation step.

(4) Data generation

In order to properly assess performance, multiple data sets that represent distinct challenges are used. In the 1D case, we have the Heat equation, which is a linear parabolic equation, the KdV equation which is a nonlinear hyperbolic equation, and Burgers' equation. In 2D, we have the Navier Stokes equations and the steady state Poisson equation. Many parameters and forcing functions are sampled in order to generate large data sets.

(4.1) Heat, Burgers', KdV equations

Following the setup from Brandstetter *et al*, we generate the 1D data. In this case, a large number of sampled parameters allow us to generate many different initial conditions and forcing terms for each equation. In our case, $J = 5$ and $L = 16$

$\partial_t u + \partial_x(\alpha u^2 - \beta\partial_x u + \gamma\partial_{xx}u)](t,x) = \delta(t,x)$ (1)

where the forcing term is given by : $\delta(t,x) = \sum_{j=1}^{J} A_j \sin\left(w_j t + \frac{2\pi l_j x}{L} + \emptyset_j\right)$, and the initial condition is the forcing term at time $t = 0$: $u(0,x) = \delta(0,x)$. The parameters in the forcing term are sampled as follows : $A_j \sim u(-0.5, 0.5)$, $\omega_j \sim U(-0.4, 0.4)$, $l_j \sim \{1,2,3\}$, $\phi_j \sim U(0, 2\pi)$. The parameters, (α, β, γ) of equation (1) can be set to define different, famous equations. When $\gamma = 0, \beta = 0$ we have the Heat equation, when only $\gamma = 0$ we have Burgers' equation, and when $\beta = 0$ we have the KdV equation. Each equation has at least one parameter that we modify in order to generate large data sets.

For the Heat and Burgers' equations, we used diffusion values of $\beta \in \{0.01, 0.05, 0.1, 0.2, 0.5, 1\}$. For the Heat equation, we generated 10000 simulations from each β value for 60000 total samples. For Burgers' equation, we used advection values of $\alpha \in \{0.01, 0.05, 0.1, 0.2, 0.5, 1\}$, and generated 2500 simulations for each combination of values, for 90000 total simulations. For the KdV equation, we used an advection value of $\alpha = 0.01$, with $\gamma \in \{2, 4, 6, 8, 10, 12\}$, and generated 2500 simulations for each parameter combination, for 15000 total simulations. The 1D equations text tokenization is padded to a length of 500. Tokenized equations are long here due to the many sampled values.

(4.2) Navier–Stokes equation

In 2D, we use the incompressible, viscous Navier–Stokes equations in vorticity form, given in equation (2), Data generation code was adapted from Li *et al* ,

$$\frac{\partial}{\partial t} w(x,t) + u(x,t) \cdot \nabla w(x,t) = \nu\Delta w(x,t) + f(x)$$
$$\nabla \cdot u(x,t) = 0, w(x,0) = w_0(x)$$
$$f(x) = A9\sin\big(2\pi(x_1 + x_2)\big) + \cos\left(2\pi(x_1 + x_2)\right)$$

where $u(x,t)$ is the velocity field, $w(x,t) = \nabla \times u(x,t)$ is the vorticity, $w_0(x)$ is the initial vorticity, $f(x)$ is the forcing term, and ν is the viscosity parameter.

We use viscosities $\nu\epsilon\{10^{-9}, 2\cdot10^{-9}, 3\cdot10^{-9}, \ldots, 10^{-8}, 2\cdot10^{-8}, 3\cdot10^{-8}, \ldots, 10^{-8}\}$ and forcing term amplitudes $A\epsilon\{0.001, 0.002, 0.003, \ldots, 0.01\}$, for 370 total parameter combinations. 120 frames are saved over $30\ s$ of simulation time. The initial vorticity is sampled according to a gaussian random field. For each combination of ν and A, 1 random initialization was used for the next-step and rollout experiments and 5 random initializations were used for the fixed-future experiments. The tokenized equations are padded to a length of 100. Simulations are run on a 1×1 unit cell with periodic boundary conditions. The space is discretized with a 256×256 grid for numerical stability that is evenly downsampled to 64×64 during training and testing.

(4.3) Steady-state Poisson equation

The last benchmark we perform is on the steady-state Poisson equation given in equation (3),

$$\nabla^2 u\ (x, y) = g\ (x, y) \tag{3}$$

where $u(x, y)$ is the electric potential, $-\nabla u(x, y)$ is the electric field, and $g(x, y)$ contains boundary condition and charge information. The simulation cell is discretized with 100 points in the horizontal direction and 60 points in the vertical direction. Capacitor plates are added with various widths, x and y positions, and charges. An example of input and target electric field magnitude is given in figure 2.

This represents a substantially different task compared to previous benchmarks. Due to the large difference between initial and final states, models must learn to extract significantly more information from provided input. This benchmark also easily allows for testing how well models are able to learn Neumann, and generalize to different combinations of boundary conditions. In two dimensions, we have four different boundaries on the simulation cell. Each boundary takes either Dirichlet or Neumann boundary conditions, allowing for 16 different combinations. In this case, since steady state is at infinite time, we pass the same time of 1 for each sample into the explicitly time-dependent models. The tokenized equation and system parameters are padded to a length of 100. Code is adapted from Zaman for this case.

(5) PinnDE

Physics-informed neural networks for differential equations (PinnDE) is an open-source library in Python providing the ability to solve differential equations with both physics-informed neural networks (PINNs) and deep operator networks (DeepONets). PinnDE is built in mainly using TensorFlow [9] with a small amount of JAX [6], serving as the backend, two popular software packages for deep learning. Functionality to provide multiple backends and other deep learning frameworks such as PyTorch for use with PinnDE is in development and will be added in a future version.

The goal of PinnDE is to provide a user-friendly interface for solving systems of differential equations which can easily be shared with collaborators even if they do not have experience working with machine learning in Python. Many alternative packages for solving differential equations with PINNs or DeepONets, such as DeepXDE [8] and PINA [4], are powerful packages in this field. They provide a high level of customization and control in how a specific problem is being solved. This naturally leads to a higher amount of software needing to be written and understood by a user, which when sharing with collaborators can lead to difficulties if not everyone understands the package to the same degree. PinnDE is a tool in which we aim to bridge this gap between functionality and ease-of-use, stripping away complexity from the code resulting in an easy to read interface non-specialists can understand. This package has user set conditions defining the type of differential equations, initial conditions, boundary conditions, and model parameters, which is then fully built with a specific training routine already implemented based off of theses flags. This allows users to have a small amount of interfacing while still being able to have access to a wide range of applications without needing to do any low-level manual implementations.

In these next sections we outline the overall structure of PinnDE, providing the current extent of this packages' capabilities. We then show how the process of solving a system of differential equations can be done using PinnDE.

(5.1) Overview

Figure 1 shows the flowchart of how a user goes about solving a system of differential equations using PinnDE. Note how a user only interfaces directly with Boundaries, Initials and Solvers. This gives a user a small amount of requirements to specify, all of which are related to the problem specification itself as well as to the basic neural network architecture and optimization procedure, but as will be demonstrated below still does not take away from the users' ability to interact with all further parts of the solution process.

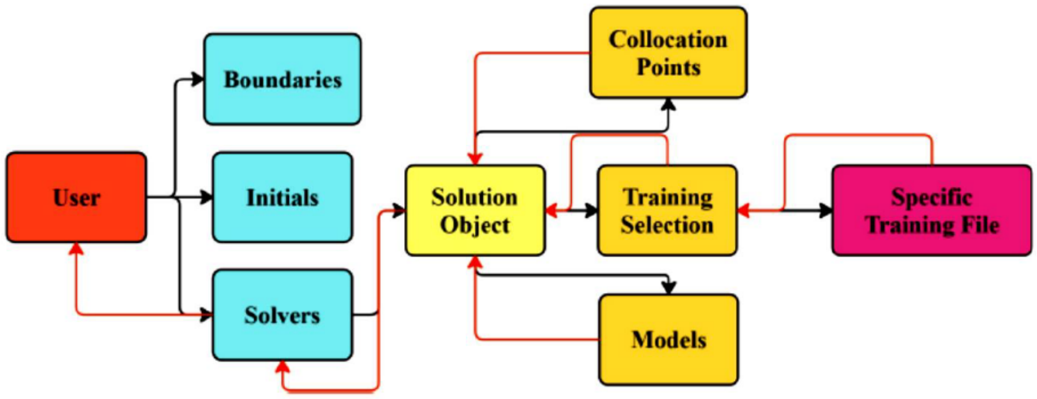

Figure 1: Flowchart of the package structure of PinnDE. Black lines represent forward function calls, red lines represent returned values.

A user first initializes the boundaries of their problem using a single function from the Boundaries module. If the problem is time-dependent and involves initial conditions, a user initializes the initial conditions using a single function from the Initials module. Then a user calls a single function in one of the Solvers modules corresponding to the system of differential equations they wish to solve. This function call creates a solution object corresponding to the Solution Object modules. This class in its constructor sets up the corresponding Collocation Points and Models to generate collocation points and a neural network with specifications from user in Solvers call. A user then calls train_model on the returned object class invokes a corresponding Training Selection function which directs the problem and generated data to a Specific Training File, where the model is trained and all training data is returned to the user through the Solution Object from the original Solvers call.

In these next sections we give a brief overview of what boundaries, solvers, and which models are currently available in PinnDE, specifying limitations in implementation if possible. We only outline the general ability of PinnDE .

(5.1.1) Initial and Boundaries

The boundaries module provides boundary generation functions for partial differential equations. Ordinary differential equations can be solved with both initial and boundary values, however the values are passed directly into the solvers function, resulting in no need of a function call from any separate module. PinnDE is presently designed to solve equations on rectangular domains. The boundaries currently available to be used are Periodic, Dirichlet, and Neumann for both equations of variables $(t, x_1) = (t, x)$ and of $(x_1, x_2) = (x, y)$. That is, PinnDE currently supports $(1+1)$ −dimensional evolution equations or two-dimensional boundary value problems. Higher-dimensional problems will be added in a future version.

PinnDE also offers both soft and hard constraining for these boundaries. The availability of what PinnDE provides is further described in Section 3.1.3 where the models of PinnDE are described. Here we give some examples of specific implementations of how hard constraining of a boundary (temporal or spatial) is done in PinnDE.

For constraining a periodic boundary condition on a PDE in variables (t, x) over a spatio-temporal domain $[t_0, t_f\,] \times \Omega, \Omega = [x_l, x_r], x_l, x_r \in R$, we only have to hard-constrain the initial conditions as periodic boundaries can be implemented using a coordinate transform layer in the neural network itself, by using the transformation $f(x) = (cos\ 2\pi x/(x_r - x_l), sin\ 2\pi x/(x_r - x_l))$ applied to the spatial input x to the neural network, This ensures that the solution will be periodic on the interval $[x_l, x_r]$.

As an example for hard constraining the initial condition itself, we assume that the equation is of second order in t. Then, we can include the initial condition as a hard constraint in the neural network surrogate solution $u^\theta(t,x)$ using the ansatz

$$u^\theta(t,x) = u_0(x) + \frac{\partial u}{\partial x}\bigg|_{t=t_0}(x)(t - t_0) + N^\theta(t,x)\left(\frac{t - t_0}{t_f - t_0}\right)^2,$$

Where u_0 and $\frac{\partial u}{\partial t}\big|_{t=t_0}$ represent the initial conditions at $t = t_0$, and $N^\theta(t,x)$ is the neural network output. Multiple variations of possible hard constraining formulas for this problem can be created which enforces the desired initial conditions. We found this particular equation to be effective for many problems we have considered, and it is therefore what is implemented in PinnDE as a default method. Alternative hard constraints will be implemented in a future version of PinnDE.

We also present the hard constraint for a Dirichlet boundary-value problem for a PDE in variables (x, y) over the rectangular domain $\Omega = [x_l, x_r] \times [y_l, y_u]$, with$x_l, x_r, y_l, y_u \in$ R. The approximate solution $u^\theta(x,y)$ is given by the equation

$$u^\theta(t,x) = A(x,y) + x^*(1 - x^*)y^*(1 - y^*)N^\theta(x,y)$$

Where we have

$$A(x,y) = (1 - x^*)f_{x1}(y) + x^* f_{xr}(y) + (1 - y^*)\left[g_{y1}(x) - [(1 - x^*)g_{y1}(x_1) + x^* g_{y1}(x_r)]\right]$$
$$+y^*\left[g_{yr}(x) - [(1 - x^*)g_{yu}(x_1) + x^* g_{yu}(x_r)]\right]$$

where f_{x1} , f_{xr} , g_{y1} , g_{yu} represent some boundary functions $f(x_l, y), f(x_r, y), g(x, y_l), g(x, y_u)$ respectively , and

$$x^* = \frac{x - x_1}{x_r - x_1}, \qquad y^* = \frac{y - y_1}{y_u - y_1}$$

This formula enforces the specific boundary conditions for all values of the neural network. This is based on what is described in [5] and is what is implemented in PinnDE.

(5.1.2) Solvers

The solvers are split into ode_Solvers and pde_Solvers, where each provide different functions for solving specific types of problems. The functions currently available are:

Table 2: Functions for solving PDEs in PinnDE

PINNs	DeepONets
solvePDE_tx()	solvePDE_DeepONet_tx()
solvePDE_xy()	solvePDE_DeepONet_xy()

Table 3 : Functions for solving ODEs in PinnDE

PINNs	DeepONets
solveODE_IVP()	solveODE_DeepONet_IVP()
solveODE_BVP()	solveODE_DeepONet_BVP()
solveODE_System_IVP()	solveODE_DeepONetSystem_IVP()

(5.1.3) Models

PinnDE currently provides the ability to use PINNs and DeepONets for solving systems of differential equations. The number of layers and nodes can both be selected within each solver function, as well as either soft or hard constraining of the network. We employ multi-layer perceptron for all neural network solution surrogates, and implement DeepONets using the architecture proposed in [7]. Both physics-informed neural networks and physics-informed deep operator networks are trained by minimizing the composite loss function (2) to approximate the solution (or solution operator) of a system of differential equations. Derivative approximations are computed using automatic differentiation [2]. Longer time integrations using DeepONets via time-stepping is realized using the procedure of [13], which iteratively applies the learned solution operator to its own output to advance the solution for arbitrary time intervals, similar as done in standard numerical methods.

Hard constraining is done using and expanded upon from the ideas from [5] and [12], which outline basic equations to force a network output to conform to initial or boundary constraints, explained further in Sections 2.1 and 3.1.1. However currently not all boundary-equation type combinations have a hard constrained model designed for, but this is can be implemented via user ' s requests if the need for them arises. We outline in Tables 4 and 5 what does and what does not currently have the ability to be used in terms of hard constraints.

Table 4 : Hard constraints for initial and boundary value problems available for solving ordinary differential equations in PinnDE.

ODE Problems/Models	PinnDEepONet
IVP	Soft, Hard Soft, Hard
BVP	Soft, Hard Soft, Hard
System of IVPs	Soft Soft

Table 5 : Hard constraints for initial and boundary value problems available for solving partial differential equations in PinnDE.

PDE Problems/Models	PinnDEepONet
(t, x) - Periodic Boundaries	Soft, Hard Soft, Hard
(t, x) - Dirichlet Boundaries	Soft Soft
(t, x) - Neumann Boundaries	Soft Soft
(x, y) - Periodic Boundaries	Soft, Hard Soft, Hard
(x, y) - Dirichlet Boundaries	Soft, Hard Soft, Hard
(x, y) - Neumann Boundaries	Soft Soft

(5.2) Usage

In this section we give the simple outline for how to use PinnDE. We describe what data must be declared when solving a system of differential equations. We give an example of the standard process solving a single partial differential equation. The process outline is described in Table 6.

Table 6 : Workflow to solve PDE

Step 1	Define the domain for the independent variables
Step 2	Define the number of collocation points along each boundary to use
Step 3	Define boundary functions as Python lambda functions, and use pde_Boundaries_2var to set up Boundaries
Step 4	Define all initial conditions as Python lambda functions, number of initial value points, and use pde_Initials to set up Initials
Step 5	Define the equation as a string, number of collocation points for the pdes, and number of training epochs
Step 6	Optionally declare number of layers of network, nodes of network , and constraint of network to interface with model
Step 7	Call corresponding solvePDE function from Solvers and input data to solve equation
Step 8	Call train_model(epochs) on returned object from solvePDE to train the network
Step 9	Use returned class to plot data with given functions or take data from class to use independently

Example code that follows this structure is given in Appendix A, where the code for the following examples in Section 4 is presented. Many more tutorials are given along with the API in the linked documentation above.

(6) Examples

In this section we present a few short examples of how PinnDE can be used to solve a system of ordinary differential equations, and multiple partial differential equations using different methods implemented in PinnDE. We provide a concise description of each equation and a comparison of each trained model against the analytical solution using the mean squared error as comparison metric. All code for these examples is provided in Appendix A.

(6.1) System of ordinary differential equations

We first solve a system of ordinary differential equations with a deep operator network which uses soft constrained initial conditions. We solve the system of second order ordinary differential equations

$$u''(t) + u(t) = 0\,, v'(t) + u(t) = 0\,, \qquad (7a)$$

with initial conditions

$$u(0) = 0.5\,,\ u'(0) = 1\,,\ v(0) = 2 \qquad (7b)$$

We solve this initial value problem over the interval $t \in [0, 1]$ using a deep operator network. Figure 2 presents the numerical solution and the exact solution as well as the time series of the loss for training the network. The analytical solution for this problem is $u(t) = -0.5\, sint + cost + 1$ and $v(t) = sint + 0.5\, cost$.

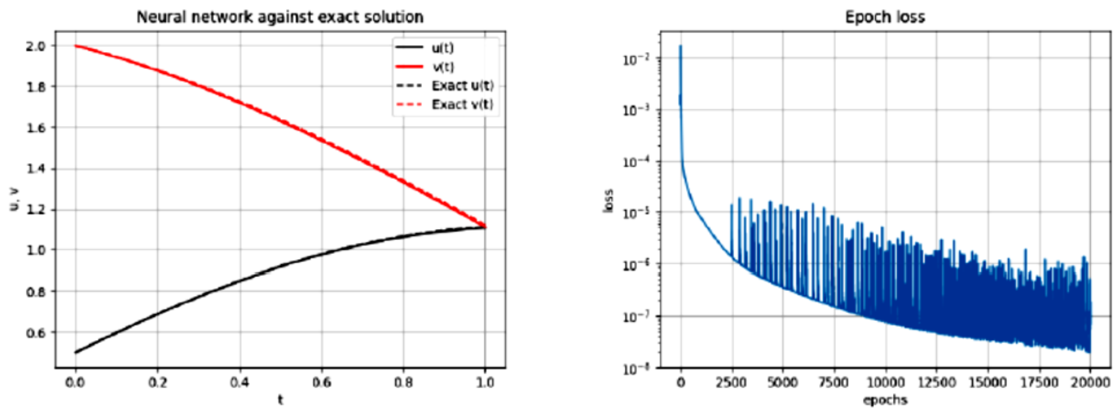

Figure 2: Solution predicted using a DeepONet against the exact solution for system (7) (left) and the time series of the physics-informed loss over the training period of the DeepONet (right).

The specific architecture used was a DeepONet with 4 hidden layers with 40 nodes per layer, using the hyperbolic tangent as activation function, and Adam as the chosen optimizer. We use 150 collocation points across the domain for the network to learn the solution, sampled with Latin hypercube sampling. We sample 5000 different initial conditions for the DeepONet in the range of $v(0), u(0), u'(0) \in [-3, 3]$. We train this network for 20000 epochs which is where the epoch loss starts to level out. For the following examples, we will also train until a minimum loss has been reached.

We further demonstrate the time-stepping ability using a DeepONet in PinnDE. We time-step the trained DeepONet for 10 steps, increasing the domain to $[0, 10]$, and give the error of the neural network solution against the exact solution. The learned neural network solution operator demonstrates the ability to time-step with a consistently low error.

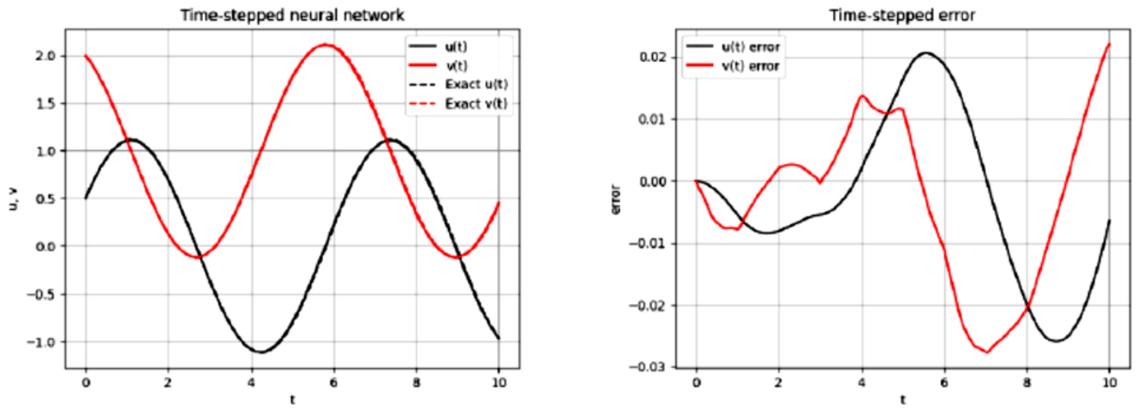

Figure 3: Time-stepped solution predicted by the trained DeepONet against the exact solution for system (7) (left), and the error between the network and analytical solution over the time-stepped solution interval (right).

The code for solving this equation can be found in Appendix A.1.

(6.2) Linear advection equation

We next use PinnDE to solve the linear advection equation with a standard PINN which uses a soft constrained initial condition. We use periodic boundary conditions, which in PinnDE are implemented at the neural network level to imposes periodic boundaries on the independent spatial variable x. This removes the boundary component of the composite loss function described in Eqn. (2) when using periodic boundaries. The linear advection with advection speed $c = 1$ is described as

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0 \qquad (8a)$$

and we use the initial condition $u_0(x) = cos\pi x \qquad (8b)$

We solve this equation over the interval $t \in [0, 1]$, on the spatial domain $x \in [-1, 1]$. We compare the neural network solution to the analytical solution $u(t, x) = cos\pi(x - t)$ in Figure 4, showing the success of the trained model, as it gives small point-wise and overall mean-squared errors, respectively.

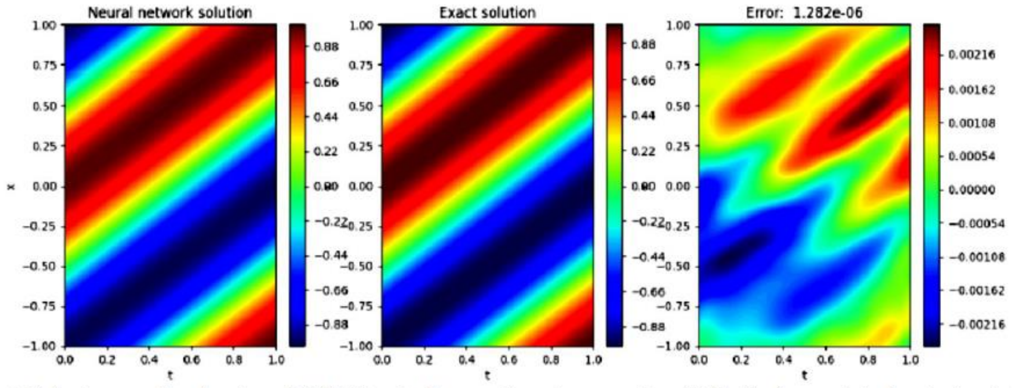

Figure 4: Solution predicted using a PINN for the linear advection equation (8) (left), the analytical solution (middle), and the mean squared error between them (right).

For this example we used a PINN with 4 hidden layers with 60 nodes per layer, the hyperbolic tangent activation function, and trained the neural network using the Adam optimizer. A total of 100 initial value collocation points were used for the network to learn the initial condition, and 10000 collocation points were used across the spatio-temporal domain for the network to learn the solution, both sampled using Latin hypercube sampling. The minimum of the loss was reached after 5000 epochs of training.
The code for solving this equation can be found in Appendix A.2.

(6.3) Poisson equation

We next use PinnDE to solve a specific instance of the Poisson equation with a PINN which uses hard constrained boundary conditions. This forces boundary values to be satisfied by the neural network exactly, leading to zero errors at the boundaries. We solve this equation using Dirichlet boundary values. The particular form of this equation we solve is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2\pi^2 cos\ \pi x\ sin\ \pi y \qquad (9)$$

with the Dirichlet boundary conditions being obtained from the exact solution for this problem given by :
$u(x, y) = cos\ \pi x\ sin\ \pi y$. We solve this equation over the square domain $(x, y) \in [-1, 1] \times [-1, 1]$.
We compare the neural network solution to the analytical solution in Figure 5.

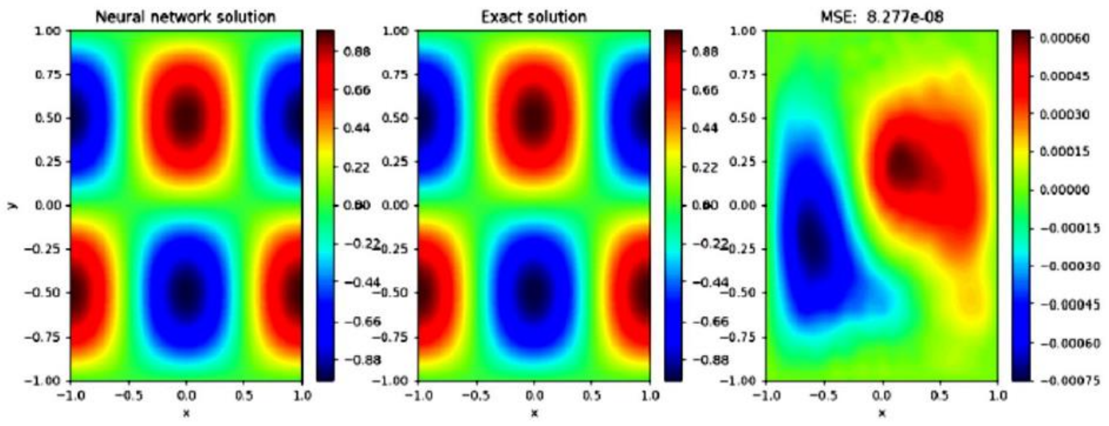

Figure 5: Solution predicted using a PINN for the Poisson equation (9) (left), the analytical solution (middle), and the mean squared error between them (right).

This solution was obtained by training a PINN with 5 hidden layers with 40 nodes per layer, using the hyperbolic tangent as activation function and Adam as the chosen optimizer. A total of 10000 collocation points were used across the domain for the network to learn the solution, sampled using Latin hypercube sampling. As the network was hard constrained, no points along the boundaries needed to be used to learn the boundary values, as these are enforced automatically. The loss reached a minimum after 5000 epochs by which point we stopped training.
The code for solving this equation can be found in Appendix A.3.

(6.4) Heat Equation

We lastly use PinnDE to solve the linear heat equation with a deep operator network which uses soft constrained initial and boundary conditions, meaning we use a composite loss function consisting of the PDE, initial condition, and boundary condition losses. We solve this equation with Dirichlet boundary values. Specifically, we consider the equation

$$\frac{\partial u}{\partial t} = v\frac{\partial^2 u}{\partial x^2} \quad , v = 0.1 \qquad (10a)$$

with initial condition

$$u_0(x) = sin\ \pi x\ , \qquad (10b)$$

and with Dirichlet boundary conditions

$$u(t,0) = u(t,1) = 0 \qquad (10c)$$

We solve this equation over the interval $t \in [0,1]$, on the spatial domain $x \in [0,1]$. We compare the neural network solution to the analytical solution $u(t,x) = exp(v\pi^2 t)sin\ \pi x$ in Figure 6.

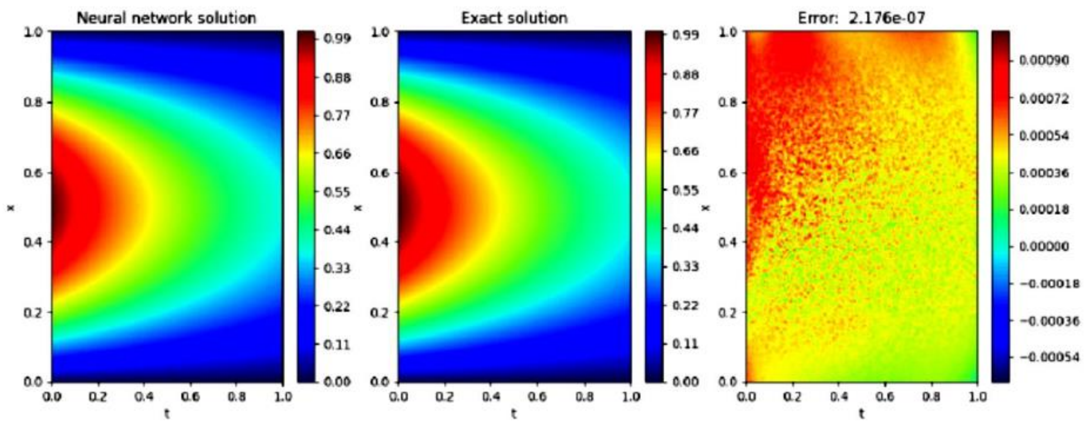

Figure 6: Solution predicted by DeepONet for the heat equation (10) (left), the analytical solution (middle), and the mean squared error between them (right).

For this example we trained a physics-informed DeepONet with 4 hidden layers with 60 nodes per layer, using the hyperbolic tangent activation function and Adam as our optimizer. 100 initial value points were used to be sampled along $t = 0$ for the network to learn the initial condition. The soft-constrained boundary was sampled using 100 boundary value points along $x = 0$ and $x = 1$ for the neural network to learn the boundary conditions, and 10000 collocation points were used across the domain for the network to learn the solution itself, both sampled with Latin hypercube sampling. We sample 30000 initial conditions for the DeepONet within a range of $u_0(x) \in [-2,2]$. We train this network for 3000 epochs which is where we have already reached a minimum loss.
The code for solving this equation can be found in Appendix A.4.

(7) Conclusion

In this paper we presented PinnDE, an open-source software package in Python for PINN and DeepONet implementations for solving ordinary and partial differential equations. We reviewed the methodologies behind PINN and DeepONet architectures, then gave a summary of PinnDE and its functionality, then presented several worked examples to show the effectiveness of this package. In PinnDE, we highlight the overall structure of the package and display the methods available for a user to solve differential equations using physics-informed neural networks. In contrast to other packages, PinnDE stresses simplicity and requires minimal coding from the side of the user, thus alleviating the need of writing higher amounts of software to support the higher level of customizability other packages offer. As such, PinnDE provides a simple interface where users with little experience can still write and understand the implementation of software written in this package. This gives researchers the ability to use this package with collaborators outside of the field who may be interested in this area of research but lacking the skills to use a more advanced package. This also provides a tool for educators to possibly use as the simple implementations gives a low barrier of entry to new users.

And introduces a novel transformer-based architecture, PITT, that learns analytically-driven numerical update operators from governing equations. A novel equation embedding method is developed and compared against standard positional encoding and embedding. PITT is able to learn physically relevant information from tokenized equations and outperforms baseline neural operators on a wide variety of challenging 1D and 2D benchmarks. We have also found our baseline models and their PITT variants with both embedding strategies have lower time-to-solution than the numerical methods used for data generation. Future work includes benchmarking on 3D systems, more effective tokenization and efficient embedding, as our novel method uses a naive approach that introduces unconventional correlation between tokens, but standard positional encoding and embedding does not use useful correlation between tokens. Additionally, the current experiments have redundancy in equations as only system parameters such as viscosity vary. Testing on multiple systems simultaneously would serve as a test for PITT's generalization capability. In addition, other works [13] have used recurrent rollout prediction as well as training rollout trajectories, which we have currently have not evaluated. These strategies can be employed to help stabilize rollout predictions.

PinnDE is continuously being developed to integrate new advancements in research surrounding PINNs, while aiming to maintain a straightforward user experience. Some possible additions which will be implemented in the future are adaptive collocation point methods, other variants on DeepONet and PINN architectures, meta-learned optimization, the availability of different backends, and general geometries which differential equations can be solved over.

References

[1]. Alex Bihlo , Jason Matthews PinnDE: Physics-Informed Neural Networks for Solving Differential Equations

[2]. Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. Journal of machine learning research, 18(153):1–43, 2018.

[3]. Cooper Lorsung , Zijie Li and Amir Barati Farimani 2024 Physics informed token transformer for solving partial differential equations

[4]. Dario Coscia, Anna Ivagnes, Nicola Demo, and Gianluigi Rozza. Physics-informed neural networks for advanced modeling. Journal of Open Source Software, 8(87):5352, 2023.

[5]. Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. IEEE transactions on neural networks, 9(5):987– 1000, 1998.

[6]. James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[7]. Lu Lu, Pengzhan Jin, and George Em Karniadakis. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. arXiv preprint arXiv:1910.03193, 2019.13

[8]. 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, 2021.

[9]. Martin 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 Mane, 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 Viegas, Oriol Vinyals, Pete Warden, MartinWattenberg, MartinWicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Largescale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 12

[10]. 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.
[11]. Rudiger Brecht, Elsa Cardoso-Bihlo, and Alex Bihlo. Physics-informed neural networks for tsunami inundation modeling. arXiv:2406.16236, 2024.
[12]. Rudiger Brecht, Dmytro R Popovych, Alex Bihlo, and Roman O Popovych. Improving physicsinformed deeponets with hard constraints. arXiv preprint arXiv:2309.07899, 2023.
[13]. Sifan Wang and Paris Perdikaris. Long-time integration of parametric evolution equations with physics-informed deeponets. Journal of Computational Physics, 475:111855, 2023.
[14]. 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.
[15]. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A N, Kaiser L and Polosukhin I 2017 Attention is all you need Advances in Neural Informat