symmetry

MDPI

Article

On the Performance of Physics-Based Neural Networks for Symmetric and Asymmetric Domains: A Comparative Study and Hyperparameter Analysis

Rafał Brociek [1], Mariusz Pleszczyński [2,*] and Dawood Asghar Mughal [3]

[1] Department of Artificial Intelligence Modelling, Faculty of Applied Mathematics, Silesian University of Technology, 44-100 Gliwice, Poland; rafal.brociek@polsl.pl
[2] Department of Mathematical Methods in Technology and Computer Science, Faculty of Applied Mathematics, Silesian University of Technology, 44-100 Gliwice, Poland
[3] Faculty of Applied Mathematics, Silesian University of Technology, 44-100 Gliwice, Poland; dm318702@student.polsl.pl
* Correspondence: mariusz.pleszczynski@polsl.pl

Abstract

This work investigates the use of physics-informed neural networks (PINNs) for solving representative classes of differential and integro-differential equations, including the Burgers, Poisson, and Volterra equations. The examples presented are chosen to address both symmetric and asymmetric domains. PINNs integrate prior physical knowledge with the approximation capabilities of neural networks, allowing the modeling of physical phenomena without explicit domain discretization. In addition to evaluating accuracy against analytical solutions (where available) and established numerical methods, the study systematically examines the impact of key hyperparameters—such as the number of hidden layers, neurons per layer, and training points—on solution quality and stability. The impact of a symmetric domain on solution speed is also analyzed. The experimental results highlight the strengths and limitations of PINNs and provide practical guidelines for their effective application as an alternative or complement to traditional computational approaches.

Keywords: numerical methods; physical-informed neural network; applied mathematics; partial differential equation; ordinary differential equation; boundary value problems

check for updates

Academic Editor: Zhibin Du

Received: 5 September 2025
Revised: 26 September 2025
Accepted: 1 October 2025
Published: 10 October 2025

Citation: Brociek, R.; Pleszczyński, M.; Mughal, D.A. On the Performance of Physics-Based Neural Networks for Symmetric and Asymmetric Domains: A Comparative Study and Hyperparameter Analysis. *Symmetry* **2025**, *17*, 1698. https://doi.org/10.3390/sym17101698

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The increase in computational power, the growing importance of computer simulations (e.g., digital twins), and the development of soft computing and artificial intelligence systems are driving the creation of various new computational methods and approaches. A fundamental challenge in computational methods is solving initial-boundary value problems (e.g., differential equations, integro-differential equations, and their systems). One of the important and relatively recent approaches to solving such problems involves neural networks, specifically physics-informed neural networks (PINNs). This approach incorporates knowledge of the governing equations, initial and boundary conditions, and possibly additional information directly into the neural network. The network then uses this information to train the model. Once the model is trained, the values of the unknown functions can be obtained directly at specified points in the domain. Some of the earliest works describing PINNs can be seen in the articles [1–3], which focus on the structure of such networks and their application to solving forward and inverse problems.

An interesting work by Cuomo et al. [4] presents the structure of PINN-type networks along with various variants. The authors provide a comprehensive review of such networks. In [5], the application of PINNs in solving a variational calculus problem is discussed. The neural network-based approach proved effective for the problem considered. Additionally, the study compares PINNs with the Differential Transform Method (DTM). The article [6] focuses on the presentation of physics-guided neural networks (PgNNs), physics-informed neural networks (PiNNs), and physics-encoded neural networks (PeNNs) in the context of fluid and solid mechanics. The conducted experiments demonstrate that the proper use of PINNs can be an effective tool in numerical simulations. In [7], PINNs were used to solve a traffic state estimation problem. The authors showed that applying PINNs to the LWR physical traffic flow model is effective, as evidenced by the experimental results. More information on potential improvements and different variants of PINN networks—such as self-adaptive loss balancing, auxiliary PINNs, adaptive collocation point movement, and adaptive loss weighting—can be found in references [8–11]. The work in [12] proposes an anti-derivatives approximator, offering a new architectural perspective to enhance the approximation of derivatives within PINNs. Self-Adaptive PINNs represent a recent paradigm for training physics-informed neural networks, where the weights of different loss components are treated as trainable parameters. This approach allows the network to dynamically balance competing objectives during optimization, improving stability and reducing the need for manual hyperparameter tuning. Several recent works have demonstrated the effectiveness of SA-PINNs in enhancing convergence and accuracy [13]. More information regarding PINNs and their applications can be found in [14–19].

In this study, we focus on three representative equations: the Poisson equation, Burgers' equation, and the Volterra integro-differential equation. While simplified, each of these test cases reflects important classes of real-world phenomena. The Poisson equation underlies models of electrostatics, incompressible fluid flow, and steady-state heat transfer. The Burgers' equation, often regarded as a prototype for the Navier–Stokes equations, is used to study nonlinear wave propagation, turbulence, and shock formation. Volterra-type integro-differential equations naturally arise in viscoelastic materials, population dynamics, and systems with memory effects. By studying these canonical problems, we can systematically evaluate the strengths and limitations of PINNs in controlled settings, while maintaining direct relevance to practical applications. Section 2 provides an overview of PINN architecture. It discusses the loss function, possible network architectures, and how such networks operate. Section 3 is dedicated to numerical experiments and demonstrates the effectiveness of PINNs on three selected examples. The focus is placed on the impact of hyperparameters on method performance, and comparisons are also made with classical numerical methods and exact solutions. The tests indicate that PINNs can be an effective tool for solving various initial-boundary value problems. Finally, Section 4 presents the conclusions.

2. Overview of Physics-Informed Neural Networks

Physics-informed neural networks (PINNs) represent a significant advancement in the field of neural networks, offering a completely new approach by integrating knowledge of physical laws into the training of deep learning models. This is achieved by embedding the governing physical laws, often expressed as differential equations, directly into the loss functions of the neural networks. This integration guides the learning process, encouraging the network to find solutions that not only fit any available data but also adhere to the fundamental physical principles governing the system. PINNs are also sometimes referred to as Theory-Trained Neural Networks (TTNs), emphasizing the incorporation of theoretical knowledge into the learning process. PINNs can approximate solutions to

forward and inverse PDE problems without the need for a discretized mesh, which is a common requirement in traditional numerical methods like Finite Element or Finite Difference methods. By incorporating physical constraints, PINNs can generalize well even with limited or imperfect data [20]. This is the reason PINNs are considered as a powerful tool for problems where data is sparse but physics is well understood, such as fluid dynamics, material science, and inverse modeling. This capability is particularly valuable in situations where obtaining complete or high-quality data is challenging. PINNs have found extensive applications across diverse scientific and engineering disciplines such as computational fluid dynamics, heat transfer, structural mechanics, and geophysics etc.

2.1. Structure of the Physics-Informed Neural Networks

The core of PINNs architecture is typically a standard feed-forward neural network. Architectures like recurrent neural networks or convolutional neural networks can also be used depending on the problem. The core structure of a PINN can be broken down into two primary components.

2.1.1. Neural Network Architecture

PINNs utilizes a neural network, most commonly a feedforward neural network or a multilayer perceptron, as a universal function approximator. This network takes spatio-temporal coordinates (x, y, z, t) or other relevant input parameters as input. The output is then the predicted solution to the differential equations describing the physical system. Mathematically, the neural network can be represented as

$$\hat{u}(x, t; \theta) = NN(x, t; \theta)$$

where $\hat{u}$ is the predicted solution, (x, t) are the input coordinates and θ denotes the parameters of the neural network. The universal approximation theorem ensures that, with sufficient depth and width, the network can approximate any continuous function to arbitrary accuracy [21].

2.1.2. Physics-Informed Loss Function

The feature that sets apart PINNs from other neural networks is the incorporation of physical laws into the loss function, which guides the training process. The total loss $\mathcal{L}$ consists of two main components $\mathcal{L}_{data}$ and $\mathcal{L}_{physics}$. The component $\mathcal{L}_{data}$ enforces agreement with observed or boundary condition data while the second component $\mathcal{L}_{physics}$ ensures the solution satisfies the governing differential equations.

The term $\mathcal{L}_{data}$ ensures that the neural network's predictions match available experimental or simulation data. For example, if we have data points $\{x_i, t_i, u_i\}_{i=1}^{N}$, the data loss can be defined by the equation

$$\mathcal{L}_{data} = \frac{1}{N} \sum_{i=1}^{N} |\hat{u}(x_i, t_i; \theta) - u_i|^2.$$

While the term $\mathcal{L}_{physics}$ encodes the governing physical laws, typically represented as PDEs. Consider a general PDE of the form

$$\mathcal{F}(u; x, t) = 0,$$

where $\mathcal{F}$ is a differential operator. The physics loss, $\mathcal{L}_{\text{physics}}$, is defined as follows:

$$\mathcal{L}_{\text{physics}} = \frac{1}{M} \sum_{j=1}^{M} \left| \mathcal{F}(\hat{u}(x_j, t_j; \theta); x_j, t_j) \right|^2,$$

where $\{(x_j, t_j)\}_{j=1}^{M}$ are collocation points sampled from the domain. The derivatives of $\hat{u}$ required for $\mathcal{F}$ are computed using an interesting feature of PINNs mapped to automatic differentiation [22].

The mathematical formulation of the physics loss often involves the Mean Squared Error (MSE) of the PDE residual, calculated at a set of collocation points within the problem domain. This MSE provides a quantitative measure of how well the neural network's predicted solution satisfies the governing physical laws.

2.2. Automatic Differentiation

Automatic differentiation is an important technique in PINNs that enables the network to learn solutions to differential equations. The applications of automatic differentiation will be presented in Section 3. We know that PINNs are designed to solve differential equations by leveraging deep neural networks. It takes input variables t and spatial coordinates (x, y, z) and outputs an approximation of the solution $u_{NN}(t, x, y, z; \theta)$. To check how well this approximate solution satisfies a given PDE, we need to compute its partial derivatives with respect to the input variables. Automatic differentiation makes these derivatives analytically accurate at any given collocation point within the domain. These derivatives are then plugged into the PDE to form the residual term. For example, for a PDE of the form $N[u(t, x)] = f(t, x)$, the residual would be $r(t, x) = N[u_{NN}(t, x; \theta)] - f(t, x)$.

When a neural network is trained by minimizing the composite loss function, automatic differentiation is used in this step to compute the gradients of the total loss function with respect to the network parameters θ. This is the standard backpropagation algorithm, which is a special case of reverse-mode automatic differentiation. These gradients guide the update of the network's weights and biases to reduce the loss. Some highly noticeable features of automatic differentiation are as follows: high accuracy, computational efficiency, ease of implementation, and handling complex geometries and high dimensions.

2.3. How PINNs Work

PINNs function by utilizing a neural network, often a deep learning model, to approximate the solution of a given differential equation. The core of the PINN approach is the incorporation of the differential equation itself into the network's training process as an additional term within the loss function [23]. This is made possible through the use of automatic differentiation, a powerful technique that allows for the efficient and accurate computation of the derivatives of the neural network's output with respect to its input variables such as space and time. These computed derivatives enable the evaluation of the residual of the differential equation. During training, the neural network's parameters such as weights and biases are adjusted by an optimization algorithm to minimize a loss function. This loss function typically comprises the error in satisfying the differential equation, the physics loss and optionally the terms that quantify the error between the network's predictions and any available labeled data, as well as the error in satisfying the specified boundary and initial conditions [10]. By minimizing this combined loss, the PINN is guided towards learning a solution that is not only consistent with any provided data but also adheres to the known physical laws encoded in the differential equation.

A distinctive feature of PINNs is the integration of the governing equations into the training loop. Through automatic differentiation, the derivatives of $u(x)$ with respect to x are efficiently computed. These derivatives are then used to formulate the residual of the governing equation $\mathcal{F}(\mathcal{D}[u], \mathcal{I}[u], u, x) = 0$ where $\mathcal{D}[u]$ and $\mathcal{I}[u]$ denote differential and integral operators applied to $u(x)$, respectively. The loss function is constructed to penalize deviations from the equation's residual, initial and boundary conditions.

$$\mathcal{L}(\theta) = \lambda_f \mathcal{L}_{\text{PDE}} + \lambda_b \mathcal{L}_{\text{BC}} + \lambda_d \mathcal{L}_{\text{Data}}$$

where each term represents the MSE at collocation, boundary, or data points. The weights λ_f, λ_b, and λ_d control the contribution of each loss component.

The flexibility of PINNs allows them to solve both forward and inverse problems. In forward problems, the objective is to compute $u(x)$ given known coefficients and source terms. In inverse problems, unknown parameters in the PDE are treated as trainable variables and inferred simultaneously with the solution.

In Figure 1, we illustrate the architecture and working mechanism of PINNs. The process begins with a neural network $NN(x, \theta)$, where x represents the input variables and θ denotes the trainable parameters of the network. The network—composed of multiple hidden layers with activation functions denoted by σ—outputs an approximation $u(x)$ of the solution to a given differential equation. This predicted output u is then used to compute the residuals of the governing equations within the domain. These include differential operators $\mathcal{D}[u]$ and possibly integral operators $\mathcal{I}[u]$, which are evaluated through automatic differentiation. The residuals are substituted into a composite physics expression $\mathcal{F}(\mathcal{D}[u], \mathcal{I}[u], u, x)$, representing the original ODE, PDE, or IDE.

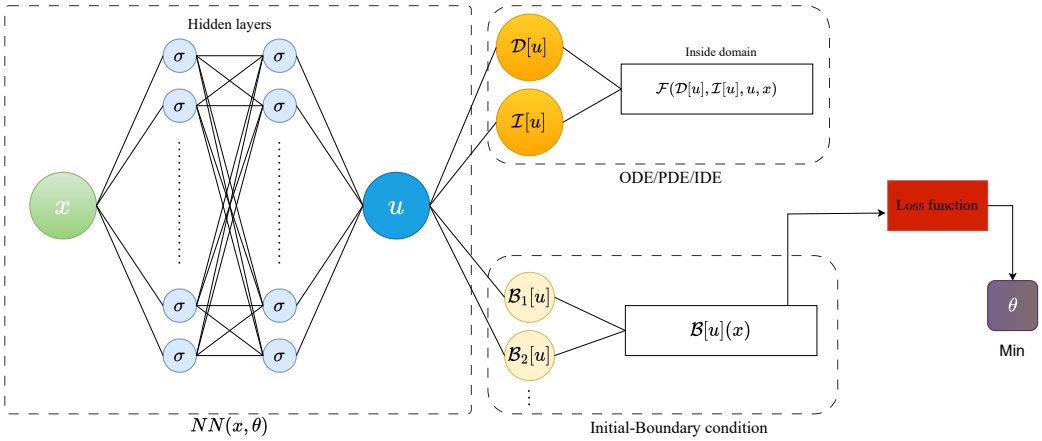


Figure 1. Schematic diagram of a physics-informed neural network.

Simultaneously, the network output is also tested against initial and boundary conditions via constraint functions $\mathcal{B}_1[u]$, $\mathcal{B}_2[u]$, etc., forming the condition residual $\mathcal{B}[u](x)$. Both the domain-based residual and the boundary-based residual are combined into a total loss function. This loss function, typically composed of multiple weighted terms, quantifies how well the neural network output adheres to the physical laws and constraints. It is then minimized using gradient-based optimization to update the neural network parameters θ, guiding the network toward a solution that satisfies the physical system.

2.4. Theoretical Properties and Analysis

2.4.1. Some Popular Activation Functions

A crucial component of neural networks is the activation function, which introduces nonlinearity into the model. This nonlinearity is essential because without it, a neural network, regardless of its depth, would essentially function as a single linear layer, severely limiting its ability to learn complex patterns and relationships present in real-world data. Various types of activation functions are commonly used in neural networks, each with its own characteristics and suitability for different tasks. The most popular of some of these are Rectified Linear Unit (ReLU) and its variants like Leaky ReLU, Parametric ReLU (PReLU), and Exponential Linear Unit (ELU). Some other notable activation functions are Sigmoid, tanh, Swish, and Softmax.

2.4.2. Error and Convergence Analysis

To understand the theory behind the working principles of PINNs, we need to study the approximation capabilities. In this section, we will discuss key theoretical aspects of PINNs, including convergence and error analysis.

Convergence refers to the process where the neural network progressively approaches a solution that satisfies both the physics constraints and any given initial and boundary conditions. For linear PDEs, Shin et al. [24] proved that as the number of training points grows, any sequence of PINNs minimizer converges to the true solution. In particular, for second-order linear elliptic and parabolic PDEs, the PINNs minimizer u_{θ_n}, trained with n collocation points, converges strongly to the unique PDE solution in C^0 norm. If initial/boundary conditions are enforced at all collocation points, convergence is even maintained in the H^1 norm.

On the optimization side, overfitting can occur if PINNs fit the data points too precisely without capturing global physics. Doumeche et al. [25] found that to prevent the overfitting and make the neural network's learning more reliable and consistent, we need to use the regularization technique. With a standard L^2 (ridge) penalty on weights, the trained PINN risk converges to the minimum possible risk in the network class as data increase. We will describe these results below.

Let $\mathcal{R}^{(\text{ridge})}_{n,n_e,n_r}(u_\theta)$ denote the class of ridge functions where n represent the input dimension of the ridge function, n_e denote the number of extracted features, n_r refer to the number of ridge components, and u_θ is the parametric function. Then

$$\mathcal{R}^{(\text{ridge})}_{n,n_e,n_r}(u_\theta) = \mathcal{R}_{n,n_e,n_r}(u_\theta) + \lambda_{(\text{ridge})}\|\theta\|_2^2, \tag{1}$$

where $\lambda_{(\text{ridge})} > 0$ is the ridge hyperparameter. We denote by $(\hat{\theta}_{(\text{p,n_e,n_r,D})})_{p\in\mathbb{N}}$ a minimizing sequence of this risk, i.e.,

$$\lim_{p\to\infty} \mathcal{R}^{(\text{ridge})}_{n,n_e,n_r}\left(u^{(\text{ridge})}_{\hat{\theta}_{(\text{p,n_e,n_r,D})}}\right) = \inf_{\theta\in\Theta} \mathcal{R}^{(\text{ridge})}_{n,n_e,n_r}(u_\theta).$$

Theorem 1 (after Shin et al. [24]). *Consider the ridge PINN problem (1), over the class $\mathcal{NN}_H(D) = \{u_\theta, \theta \in \Theta_{H,D}\}$, where $H \geq 2$. Assume that the condition function h is Lipschitz and that $\mathcal{F}_1, \ldots, \mathcal{F}_M$ are polynomial operators. Assume, in addition, that the ridge parameter is of the form*

$$\lambda_{(\text{ridge})} = \min(n_e, n_r)^{-\kappa}, \quad \textit{where} \quad \kappa = \frac{1}{12 + 4H(1 + (2 + H)\max_k \deg(\mathcal{F}_k))}.$$

Then, almost surely,

$$\lim_{n_e,n_r\to\infty} \lim_{p\to\infty} \mathcal{R}_n(u_{\hat{\theta}^{(\text{ridge})}(p,n_e,n_r,D)}) = \inf_{u\in\mathcal{NN}_H(D)} \mathcal{R}_n(u).$$

Theorem 2 (after Shin et al. [24]). *[The ridge PINN is asymptotically unbiased] Under the same assumptions as in Theorem 1, one has, almost surely,*

$$\lim_{D\to\infty} \lim_{n_e,n_r\to\infty} \lim_{p\to\infty} \mathcal{R}_n(u_{\hat{\theta}^{(\text{ridge})}(p,n_e,n_r,D)}) = \inf_{u\in C^\infty(\Omega,\mathbb{R}^{d_2})} \mathcal{R}_n(u).$$

The fundamental results in [24] shows that if a PINN is trained at increasingly many collocation points, its error converges strongly. In the limit, the PINN solution matches the exact solution in uniform norm. In the work of Yoo et al. [26], stable error bound of 1D linear elliptic boundary-value problems has been proved. The L^2 norm of $u - u_\theta$ is bounded by the PINN loss, independent of the differential equation coefficients.

Mishra et al. [27] carried out a detailed analysis for linear and semi-linear parabolic PDEs in high dimensions. They show that there exist PINNs that achieve arbitrary accuracy ε in approximating the solution, with network size scaling only polynomial in d and $\frac{1}{\varepsilon}$. The construction of their work is summarized in the following theorem.

Theorem 3 (after Mishra et al. [27]). *Let $\alpha, \beta, \varpi, \zeta, T > 0$ and let $p > 2$. For every $d \in \mathbb{N}$, let $D_d = [0,1]^d$, $\varphi_d \in C^5(\mathbb{R}^d)$ with bounded first partial derivatives, let $(D_d \times [0,T], \mathcal{F}, \mu)$ be a probability space, and let $u_d \in C^{2,1}(D_d \times [0,T])$ be a function that satisfies*

$$(\partial_t u_d)(x,t) = \mathcal{L}[u_d](x,t), \quad u_d(x,0) = \varphi_d(x) \quad \text{for all } (x,t) \in D_d \times [0,T].$$

Moreover, assume that for every $\xi, \delta, c > 0$, there exist tanh *neural networks $\widehat{\varphi}_{\xi,d} : \mathbb{R}^d \to \mathbb{R}$ and $(\widehat{\mathcal{F}\varphi})_{\delta,d} : \mathbb{R}^d \to \mathbb{R}$ with, respectively, $\mathcal{O}(d^\alpha \xi^{-\beta})$ and $\mathcal{O}(d^\alpha \delta^{-\beta})$ neurons and weights that grow as $\mathcal{O}(d^\varpi \xi^{-\zeta})$ and $\mathcal{O}(d^\varpi \delta^{-\zeta})$ such that*

$$\left\| \varphi_d - \widehat{\varphi}_{\xi,d} \right\|_{C^2(D_d)} \leq \xi \quad \text{and} \quad \left\| \mathcal{F}\varphi - (\widehat{\mathcal{F}\varphi})_{\delta,d} \right\|_{C^2([-c,c]^d)} \leq \delta.$$

Then there exist constants $C, \lambda > 0$ such that for every $\varepsilon > 0$ and $d \in \mathbb{N}$, there exist a constant $\rho_d > 0$ and a tanh *neural network $\Psi_{\varepsilon,d}$ with at most $C(d\rho_d)^\lambda \varepsilon^{-\max\{5p+3, 2+p+\beta\}}$ neurons and weights that grow at most as $C(d\rho_d)^\lambda \varepsilon^{-\max\{\zeta, 8p+6\}}$ for $\varepsilon \to 0$ such that*

$$\left\| \partial_t \Psi_{\varepsilon,d} - \mathcal{L}[\Psi_{\varepsilon,d}] \right\|_{L^2(D_d \times [0,T])} + \left\| \Psi_{\varepsilon,d} - u_d \right\|_{H^1(D_d \times [0,T])} + \left\| \Psi_{\varepsilon,d} - u_d \right\|_{L^2(\partial(D_d \times [0,T]))} \leq \varepsilon.$$

Moreover, ρ_d is defined as

$$\rho_d := \max_{x \in D_d} \sup_{\substack{s,t \in [0,T], \\ s<t}} \frac{\| X_s^x - X_t^x \|_{\mathcal{L}^q(F, \|\cdot\|_{\mathbb{R}^d})}}{\|s-t\|^{\frac{1}{p}}} < \infty,$$

where X^x is the solution of the stochastic differential equation

$$dX_t^x = \mu(X_t^x)dt + \sigma(X_t^x)dB_t, \quad X_0^x = x, \quad x \in D, t \in [0,T],$$

and $q > 2$ is independent of d.

They further prove quantitative generalization bounds. If the PINN training loss is below ε, then the L^2-solution error is also $O(\varepsilon)$. They present this result precisely in the following theorem.

Theorem 4 (after Mishra et al. [27]). *Let u be a (classical) solution to a linear Kolmogorov equation*

$$\begin{cases} u_t(t,x) = \frac{1}{2}\mathrm{Tr}\big(\sigma(x)\sigma(x)^T H_x u(t,x)\big) \\ \qquad\qquad + \mu(x)^T \cdot \nabla_x u(t,x), & \text{for } (t,x) \in [0,T] \times D, \\ u(0,x) = \varphi(x), & \text{for } x \in D, \\ u(t,x) = \psi(x,t), & \text{for } (t,x) \in [0,T] \times \partial D, \end{cases}$$

where $\sigma : \mathbb{R}^d \to \mathbb{R}^{d \times d}$ and $\mu : \mathbb{R}^d \to \mathbb{R}^d$ are affine functions, ∇_x denotes the gradient and H_x the Hessian. with $\mu \in C^1(D; \mathbb{R}^d)$ and $\sigma \in C^2(D; \mathbb{R}^{d \times d})$, u_θ a PINN and let the residuals be defined by

$$\begin{cases} \mathcal{R}_i[v](x,t) = \partial_t v(x,t) - \mathcal{L}[v](x,t), & (x,t) \in D \times [0,T], \\ \mathcal{R}_s[v](y,t) = v(y,t) - \psi(y,t), & (y,t) \in \partial D \times [0,T], \\ \mathcal{R}_t[v](x) = v(0,x) - \varphi(x), & \forall x \in D. \end{cases}$$

Then

$$\begin{aligned} \|u - u_\theta\|^2_{L^2(D\times[0,T])} \leq C_1 \Big[& \|R_t[u_\theta]\|^2_{L^2(D\times[0,T])} + \|R_t[u_\theta]\|^2_{L^2(D)} \\ & + C_2 \|R_s[u_\theta]\|_{L^2(\partial D\times[0,T])} + C_3 \|R_s[u_\theta]\|^2_{L^2(\partial D\times[0,T])} \Big], \end{aligned}$$

where

$$\begin{aligned} C_0 &= \sum_{i,j=1}^{d} \|\partial_{ij}(\sigma\sigma^T)_{ij}\|_{L^\infty(D\times[0,T])}, \\ C_1 &= Te^{(C_0 + \|\operatorname{div}\mu\|_\infty + 1)T}, \\ C_2 &= \sum_{i=1}^{d} \|(\sigma\sigma^T J_x[u - u_\theta]^T)_i\|_{L^2(\partial D\times[0,T])}, \\ C_3 &= \|\mu\|_\infty + \sum_{i,j,k=1}^{d} \|\partial_i(\sigma_{ik}\sigma_{jk})\|_{L^\infty(\partial D\times[0,T])}. \end{aligned}$$

2.5. Training and Optimization in Neural Networks

Training a neural network involves adjusting its parameters to minimize the loss function. This minimization occurs by iteratively adjusting the internal parameters like weights (w_i) and biases (b). This process typically involves four steps to constitute one training iteration. We will describe these four steps briefly.

(i) Forward Pass: Input data is fed into the network and propagates through its layers. Each neuron in a layer receives inputs from the previous layer, applies a weighted sum and an activation function, and passes the output to the next layer. This process continues until an output is generated by the final layer.

(ii) Loss Computation: The network's output is compared to the true target values using a loss function. This function quantifies the error or discrepancy between the predicted output and the expected output. Common loss functions include measures prediction errors like Mean Squared Error and Cross-Entropy. A higher loss value indicates a greater error.

(iii) Backward Pass: The error calculated by the loss function is propagated backward through the network. It computes the gradient of the loss function with respect to each weight and bias in the network by using chain rule.

(iv) Parameter Update: Using the gradients computed during backpropagation, an optimizer algorithm adjusts the weights and biases of the network. The goal is to update the parameters in a way that reduces the loss function. The size of the steps taken during this adjustment is controlled by the learning rate.

The training process involves performing many such iterations, often grouped into epochs, where one epoch represents a full pass through the entire training dataset. The cycle repeats for each batch of training data. The efficiency and overall success of neural network training process are profoundly influenced by the selection of optimization algorithms and the careful tuning of hyperparameters. Modern optimizers like Adam [28] rapidly adjust learning rates during training that often leads to faster convergence and improved performance compared to traditional methods. There are also other optimization algorithms used in neural networks like RMSprop and Adagrad. Each optimizer has its own strengths and

weaknesses, and the most suitable choice can depend on the specific network architecture, the input data, and the nature of the problem being solved. Performing experiments with different optimizers is common practice to find the one that yields the best results for a given task.

3. Application of PINNs to Selected Equations

This section is dedicated to the practical applications of physics-informed neural networks in the domain of ordinary differential equations, partial differential equations, and integro-differential equations. The manuscript considers the following equations: the Poisson equation, the Burgers' equation, and the Volterra integro-differential equation. For clarity and to better justify the choice of test problems, we briefly outline the key differences between the Poisson Equation (2) and the Burgers' Equation (3). The Poisson equation in the considered form is a linear, second-order elliptic equation depending only on the spatial variable, thus describing a stationary boundary-value problem. In contrast, the Burgers' equation is an evolutionary equation, which for $\nu > 0$ has a parabolic character, involves the time derivative, and requires both an initial condition and boundary conditions. Unlike the linear diffusion operator in the Poisson equation, Burgers' equation contains the nonlinear advection term $u\partial_x u$, leading to much more complex phenomena such as the formation of steep gradients or shock waves in the limit $\nu \to 0$. These differences also result in distinct solution behaviors: for the Poisson equation, solutions are smooth (for smooth input data) and free of discontinuities, whereas the Burgers' equation can generate boundary layers and high-gradient structures, requiring careful consideration when choosing numerical methods. In practice, this means that solutions of the Poisson equation can be relatively easily approximated using PINNs, while for the Burgers' equation, the additional challenge lies in accurately capturing the temporal dynamics and the nonlinear advection term. This sometimes necessitates denser sampling in regions with steep gradients or appropriate weighting of the loss function components during the training of the network. To explicitly quantify the accuracy of our implementations, we directly compared the PINN solutions with the exact analytical solutions for each test case. This provides implementation-specific error bounds, expressed as mean and maximum errors, which are reported in the corresponding tables and figures.

We calculated the solutions and compared them using different hyperparameters in the constructed network. We also considered some traditional numerical methods for benchmarking. For the sake of completeness, our focus was on nonlinear problems.

3.1. Poisson Equation

As our first problem to be tested for physics-informed neural networks (PINNs), we are considering a nonlinear yet simple one-dimensional Poisson equation. The Poisson equation arises in various fields such as electrostatics, fluid dynamics, and heat transfer [29,30]. Numerical and exact solutions of Poisson equation have been found in studies using different approaches, such as R.W. Klopfenstein et al., who studied the Poisson equation for semiconductors doped with an ion-implanted profile using the mesh method of solution [31]. Using deep neural networks, S. Bhardwaj et al. studied a solution of the Poisson equation [32].

When one dimensional (1D), the Poisson equation is simpler but still captures the essential features of the problem. We are considering the equation of the form

$$\frac{d^2y}{dx^2} + \pi^2 \sin(\pi x) = 0 \tag{2}$$

with boundary conditions $y(-1) = 0$, $y(1) = 0$ and $-1 \leq x \leq 1$. The exact solution for this problem is given as $y(x) = \sin(\pi x)$. When this problem is solved using PINNs, quite satisfactory results were obtained. We have tested different hyperparameters by changing the number of training points on the domain, number of neurons, and number of hidden layers.

The performance of the PINNs in solving the one-dimensional Poisson equation is summarized in Table 1. To compute the errors, a grid independent of the training points was used. The table reveals that the mean absolute error varies significantly with different hyperparameter combinations. Notably, the smallest error 0.0001092 was achieved with higher neuron counts and intermediate layer depths, suggesting that a moderate network size can yield highly accurate results.

Table 1. Performance of different hyperparameter combinations for Equation (2).

Training Points	Neurons	Layers	Mean Error	Max Error
32	10	5	0.0001789	0.0003578
64	10	5	0.0042591	0.0018514
64	15	7	0.0012387	0.0027478
128	15	5	0.0001534	0.0004693
128	20	7	0.0001092	0.0003296

Figure 2 compares the PINNs solution with the exact solution, showcasing the network's ability to closely match the theoretical result. The close alignment between the predicted and exact solutions confirms the robustness of PINNs for this class of problems. A plot of the absolute error as a function of x reveals that the error is expected to be minimal across the domain.

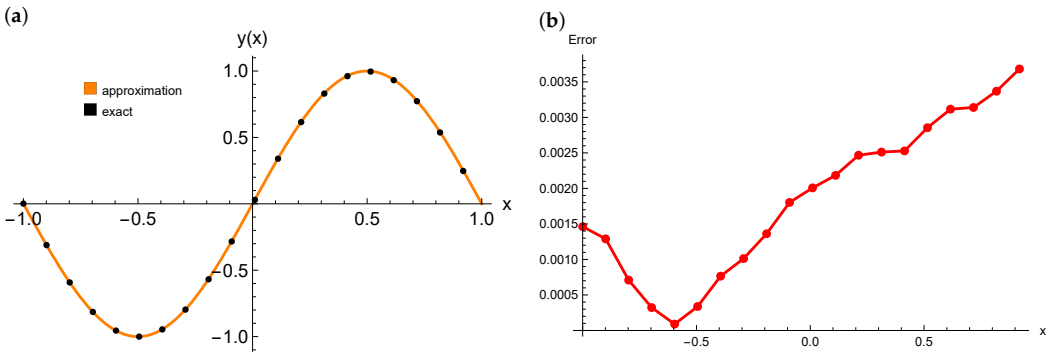


Figure 2. Plots of the approximated and exact solution (**a**) and the absolute approximation error (**b**).

The train loss and test loss during the process of solving Equation (2) is shown in Figure 3. It can be seen that the training loss and test loss exhibit a strong correlation, with both decreasing sharply in early steps before stabilizing. The minimal gap between training and test loss highlights the model's balanced capacity.

The influence of the number of hidden layers and neurons on the mean and maximum errors was also investigated. These errors were computed on a uniform test grid of dimension 100, independent of the training data. In all experiments, 200 collocation points and 10,000 iterations were used. The last column in Table 2 (Params) refers to the total number of network parameters (weights and biases). The tests showed that the training time increases almost linearly with the number of parameters (correlation coefficient ≈ 0.95). The lowest mean and maximum errors were obtained for moderately small architectures, e.g., 2×20 (20 neurons, two hidden layers, 4.81×10^2 parameters). These networks achieved errors on the order of 10^{-4} with relatively short computation time

(≈13 s). For very large architectures (e.g., 20×100 with 192,201 parameters), the results were significantly worse, with errors reaching the range of 10^{-2}. This indicates optimization difficulties and instability of PINNs in this regime, likely due to the limited number of iterations (all tests used 10,000 iterations). The best trade-offs between accuracy and training time are achieved by shallower and moderately wide networks, e.g., 2×20 or 6×5. Based on the data in Table 2, Figures 4 and 5 illustrate the mean error as a function of the number of network parameters, as well as the computation time as a function of the number of parameters.

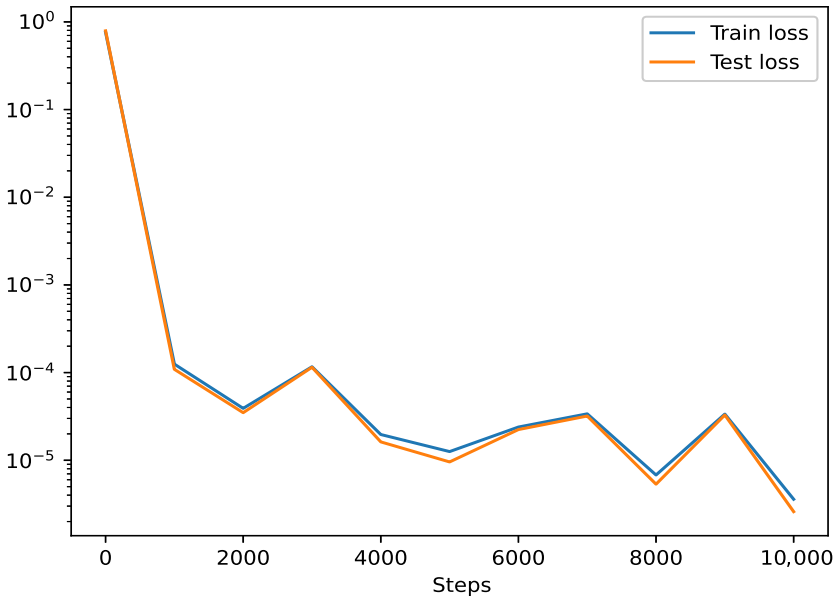


Figure 3. Train and test loss during solution of (2).

Table 2. Performance of different hyperparameter combinations for Poisson equation.

Neurons	Hidden Layers	Mean Error	Max Error	Training Time [s]	Params
5	1	0.0000962	0.0004039	8.35	16
10	1	0.0002057	0.0004992	9.92	31
5	2	0.0000971	0.0002125	10.65	46
10	2	0.0002535	0.0007126	11.88	141
20	2	0.0000545	0.0001862	13.12	481
50	4	0.0012889	0.0023040	20.81	7801
100	4	0.0033755	0.0063511	27.10	30,601
5	6	0.0000933	0.0002061	17.19	166
10	6	0.0013967	0.0028064	17.80	581
50	6	0.0042395	0.0137589	24.37	12,901
20	10	0.0132887	0.0235891	27.25	3841
30	10	0.0041084	0.0084747	30.95	8461
100	10	0.0120098	0.0292455	59.57	91,201
30	20	0.0023960	0.0056850	54.97	17,761
50	20	0.0149564	0.0352005	76.35	48,601
100	20	0.0183402	0.0355639	147.95	192,201

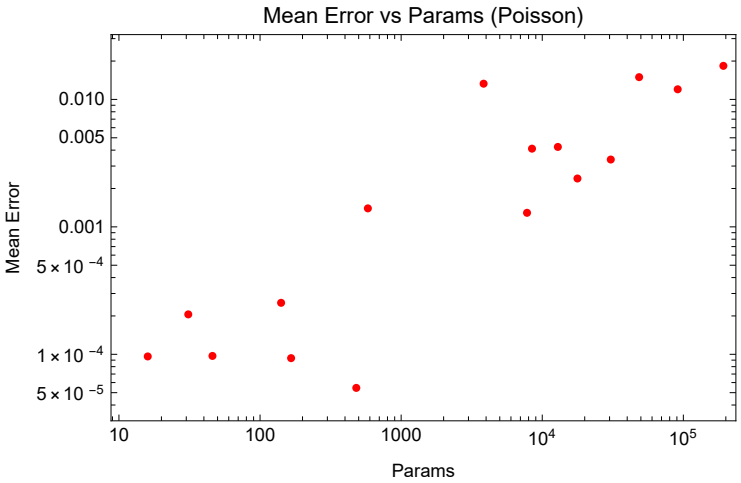


Figure 4. Mean error as a function of the number of network parameters (Poisson equation).

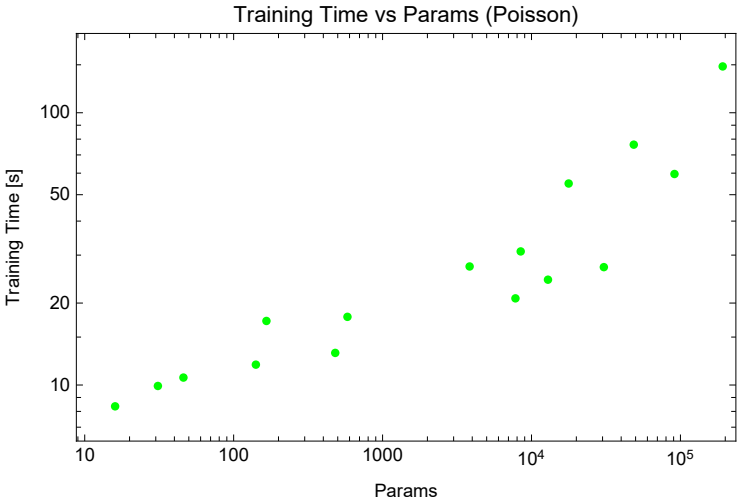
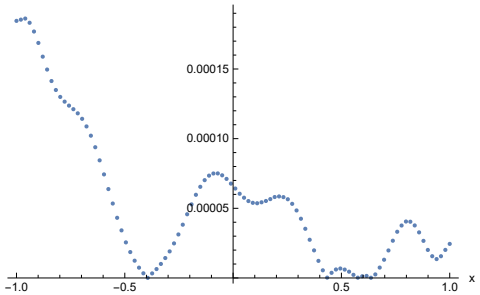
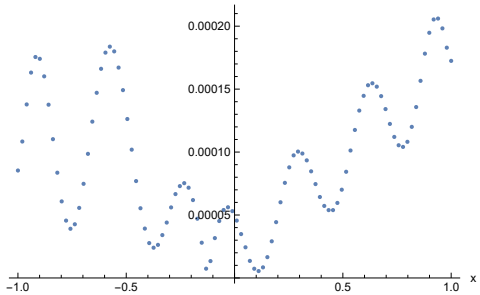


Figure 5. Training time as a function of the number of network parameters (Poisson equation).

In Figure 6, the error distribution at the grid points is shown for four selected network architectures, while Figure 7 shows prediction from PINN for two hidden layers and 20 neurons.

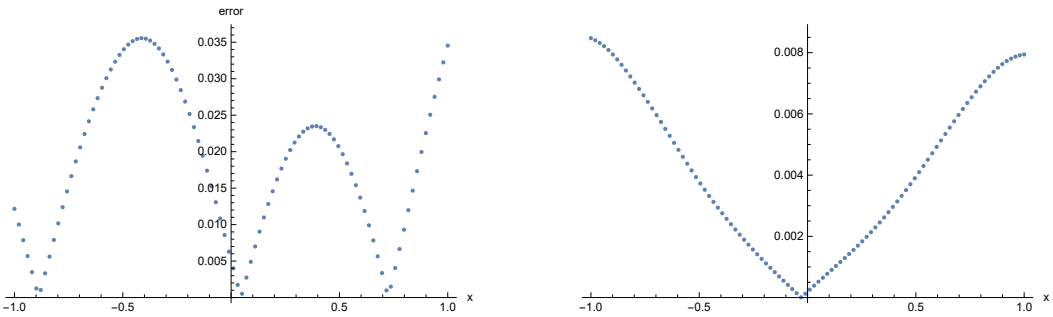


(**a**) 2 hidden layer with 20 neurons



(**b**) 6 hidden layers with 5 neurons

Figure 6. *Cont.*



(**c**) 20 hidden layers with 100 neurons (**d**) 10 hidden layers with 30 neurons

Figure 6. Error distribution obtained from the PINN for selected hyperparameters concerning the number of hidden layers and neurons (Poisson equation).

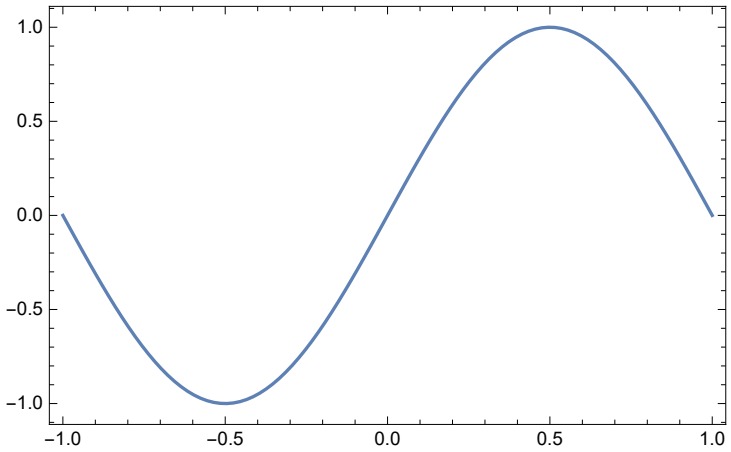


Figure 7. Prediction from PINN for 2 hidden layers and 20 neurons (Poisson equation).

The results, summarized in Table 3, confirm the expected trend that the mean error decreases as the number of training points increases, thus providing numerical evidence for the theoretical convergence. We note, however, that simply increasing the number of training points is not sufficient by itself. To achieve stable convergence, it is also necessary to appropriately adjust the training procedure, in particular the number of iterations and the learning rate schedule of the Adam optimizer. Our experiments implement such a multi-stage training scheme, ensuring that the optimizer has the capacity to effectively use the additional information provided by more collocation points. In the cases of 5000 and 10,000 training points (compared to the other cases), the number of training iterations was significantly increased.

Table 3. Mean and maximum error for different numbers of training points in the Poisson equation experiment.

Training Points	Mean Error	Max Error
100	0.0000069	0.0000126
500	0.0000030	0.0000067
1000	0.0000022	0.0000053
5000	0.0000013	0.0000037
10,000	0.0000011	0.0000022

Additional experiments on the Poisson equation were conducted using several commonly employed activation functions, including ReLU, Sigmoid, sin, Swish, tanh, and ELU. The results are summarized in Table 4. The comparison clearly shows that tanh yields

the lowest errors among the tested functions, with Swish and Sigmoid also performing competitively well. In contrast, ReLU and ELU result in significantly larger prediction errors, confirming that smoother nonlinearities are more suitable for this PDE problem. These results justify our choice of tanh as the primary activation function in the paper, while also highlighting the potential of Swish and Sigmoid as alternatives in related applications.

Table 4. Mean and maximum error for different activation functions in the Poisson equation experiment.

Activation	Mean Error	Max Error
ReLU	0.6331564	0.9996860
Sigmoid	0.0000459	0.0000838
sin	0.0000770	0.0001597
Swish	0.0000285	0.0000698
tanh	0.0000158	0.0000494
ELU	0.0045412	0.0107337

3.2. Burgers' Equation

Burgers' equation is a fundamental partial differential equation (PDE) that combines nonlinear advection and diffusion, serving as a simplified model for fluid dynamics, shock waves, and traffic flow [33]. Burgers' equation is a one-dimensional analog of the Navier–Stokes equations, making it a valuable tool for studying. It is given by the following:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \qquad a \leq x \leq b, \quad t \geq 0 \tag{3}$$

with boundary conditions and initial condition $u(a,t) = f(t), u(b,t) = g(t)$ and $u(x,0) = \phi(x)$. In this case:

- u is the solution as a function of space and time, $u(x,t)$;
- ν is the viscosity coefficient which controls the smoothness of the solution;
- f, g and ϕ are known functions.

Burgers' equation has its basis in the mathematical literature as shown by S.-S. Xie et al., who solved it using reproducing kernel function [34], and B. Inan et al., who solved it using implicit and fully implicit exponential finite difference methods [35]; even more extensive literature on Burgers' equation can be seen in the work of M.P. Bonkile [36].

In this work we consider Equation (3) for $\nu = 1$ with the initial condition

$$u(x,0) = \sin(\pi x), \quad 0 < x < 1$$

and homogeneous boundary conditions

$$u(0,t) = u(1,t) = 0, \quad t > 0.$$

This equation is solved by S. Kutluay et al. in [37]. They have used the explicit and exact-explicit finite difference methods to solve the equation with two different initial conditions. We explain the explicit and exact-explicit finite difference methods here and then compare the results through these methods with the physics-informed neural networks. First, we briefly describe these methods, and then we proceed with the comparison of the results.

3.2.1. Explicit Finite Difference Method

In this approach, the Burgers' equation is discretized using a standard explicit scheme applied to the linear heat equation obtained via the Hopf–Cole transformation. The spatial domain $[0,1]$ is divided into N intervals with step size h, and the time domain is discretized

with step size k. The finite difference approximation for the linear heat equation is given by the following:

$$\theta_{i,j+1} = (1-2r)\theta_{i,j} + r(\theta_{i-1,j} + \theta_{i+1,j}), \quad i = 1, \ldots, N-1,$$

where $r = \frac{kv}{h^2}$ and $\theta_{i,j}$ approximates the solution at grid point (x_i, t_j). The boundary conditions are handled separately for $i = 0$ and $i = N$. The stability condition for this method is $k \leq \frac{h^2}{2v}$. Once the solution $\theta_{i,j}$ is computed, the Hopf–Cole transformation is applied to obtain the numerical solution for the Burgers' equation:

$$u(x_i, t_j) = -\frac{v}{h}\left(\frac{\theta_{i+1,j} - \theta_{i-1,j}}{\theta_{i,j}}\right).$$

This method is straightforward but requires careful attention to the stability constraint, especially for small values of viscosity ν.

3.2.2. Exact-Explicit Finite Difference Method

This approach derives an exact solution to the finite difference scheme itself, rather than discretizing the continuous equation. The method assumes a product solution of the form $\theta_{i,j} = f_i g_j$, separating spatial (f_i) and temporal (g_j) components. The spatial part yields the following:

$$f_i = B\cos\left(\frac{is\pi}{N}\right), \quad s = 0, 1, 2, \ldots$$

while the temporal part satisfies the following:

$$g_j = A\left(1 - 4r\sin^2\left(\frac{s\pi}{2N}\right)\right)^j.$$

The complete solution combines these through superposition:

$$\theta_{i,j} = \sum_{s=0}^{\infty} D_s\left(1 - 4r\sin^2\left(\frac{s\pi}{2N}\right)\right)^j \cos\left(\frac{is\pi}{N}\right).$$

The coefficients D_s are determined from the initial condition using Fourier cosine series. Finally, the Hopf–Cole transformation converts this to the Burgers' equation solution:

$$u(x_i, t_j) = 2\pi v \frac{\sum_{s=1}^{\infty} D_s\left(1 - 4r\sin^2\left(\frac{s\pi}{2N}\right)^j s\sin(s\pi x_i)\right)}{D_0 + \sum_{s=1}^{\infty} D_s\left(1 - 4r\sin^2\left(\frac{s\pi}{2N}\right)\right)^j \cos(s\pi x_i)}.$$

This method provides an exact solution to the discrete equations, converging to the Fourier solution as $h \to 0$.

3.2.3. Accuracy Comparison Between Classical Numerical Methods and PINNs

We have solved Equation (3) using PINNs and then compare the results with the results obtained from explicit, exact-explicit methods, and with exact solution [37]. The solutions are obtained using 200 training points and 10 layers of neurons, each with five neurons, while the activation function is tanh. Table 5 shows a comparison of numerical solutions obtained using the explicit method, exact-explicit method, and PINNs with the exact solution at time $t = 0.1$ and $\nu = 1$. We can see that all methods closely follow the exact solution, but there are small differences. The PINNs method gives values that are very close to the exact solution, especially at points where x is around 0.5.

Table 5. Comparison of numerical solutions using explicit method, exact-explicit method, and PINNs with exact solution at $t = 0.1$ and $v = 1$.

x	Explicit [37]	Exact-Explicit [37]	PINNs	Exact Solution
0.1	0.10863	0.11048	0.11166	0.10954
0.2	0.20805	0.21159	0.21210	0.20979
0.3	0.28946	0.29435	0.29440	0.29190
0.4	0.34501	0.35080	0.35059	0.34792
0.5	0.36845	0.37458	0.37446	0.37158
0.6	0.35601	0.36189	0.36214	0.35905
0.7	0.30728	0.31231	0.31310	0.30991
0.8	0.22588	0.22955	0.23133	0.22782
0.9	0.11966	0.12160	0.12559	0.12069

A comparison of exact solution and approximate solutions is given in Figure 8. Figure 8a plots these solutions visually. The PINNs, Explicit, and Exact-Explicit solutions all follow the exact solution curve very well. However, the small differences are more noticeable in Figure 8b, which shows the absolute error between each method and the exact solution. The Explicit method has the largest error near $x = 0.5$, while PINNs and Exact-Explicit methods have smaller and more consistent errors.

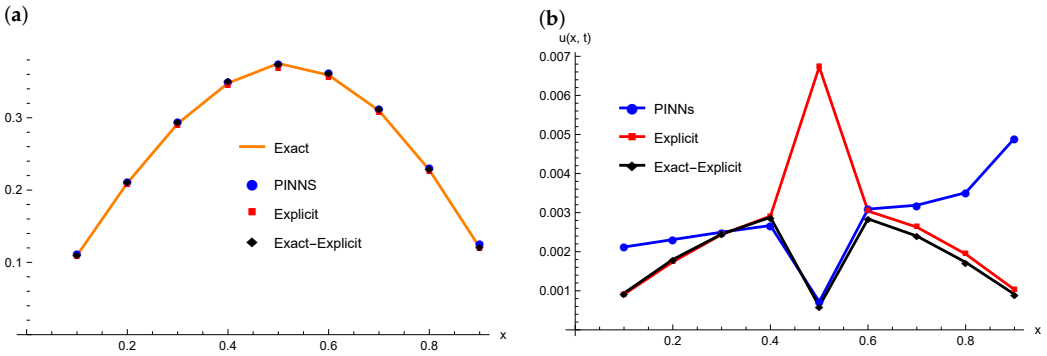


Figure 8. Plots of the approximated and exact solution (**a**) and the absolute approximation error (**b**).

Figure 9 shows the training and testing loss of the PINNs model. Both the training and testing losses decrease quickly in the first few steps and continue to go down gradually. This suggests that the model is learning well and generalizing properly without overfitting.

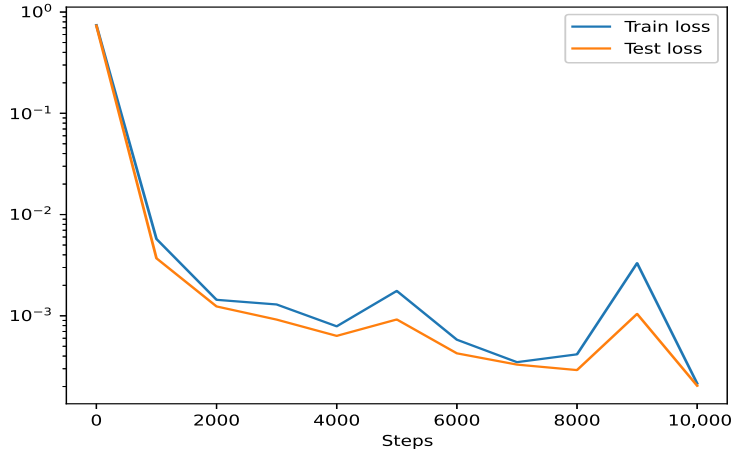


Figure 9. Train and test loss.

To further validate the performance of the physics-informed neural networks approach in solving Burgers' equation, we investigated the evolution of the solution at a fixed spatial

point $x = 0.5$. Table 6 presents a comparison between the PINNs solution and traditional numerical methods (explicit and exact-explicit schemes) against the exact solution for various time values. It is evident from the table that the PINNs approach provides results with high accuracy, closely matching the exact solution.

Table 6. Comparison of the numerical solutions with exact solution at different times for $v = 0.1$.

x	t	Numerical Solution			Exact Solution
		Explicit [37]	Exact-Explicit [37]	PINNs	
0.5	0.4	0.56911	0.56964	0.57055	0.56963
	0.6	0.44676	0.44721	0.44644	0.44721
	0.8	0.35888	0.35924	0.35721	0.35924
	1.0	0.29162	0.29192	0.29031	0.29192
	3.0	0.04017	0.04021	0.04003	0.04021

As time progresses from $t = 0.4$ to $t = 3.0$, the solution $u(x, t)$ exhibits a smooth decay, which is characteristic of the dissipative nature of Burgers' equation with viscosity $\nu = 0.1$. This behavior is clearly illustrated in Figure 10, where the PINNs prediction aligns tightly with both explicit schemes and the exact solution.

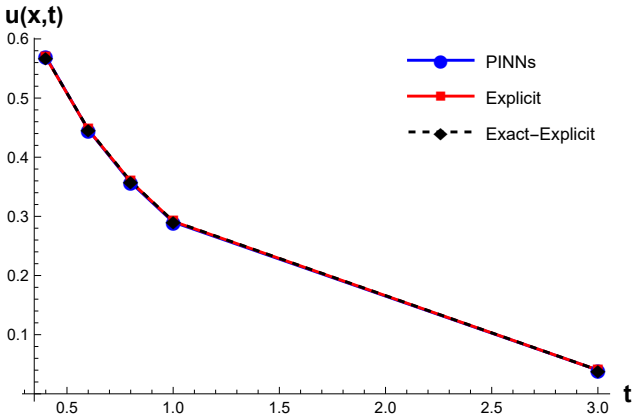


Figure 10. Comparison of solutions for different values of t and $x = 0.5$.

As part of the tests, a comparison was also conducted with another numerical method described in [38]. This method depends on the mesh density (parameter N). Table 7 presents the values obtained with the referenced numerical method, the results from PINN, as well as the exact solutions. Figure 11 shows the error plots for the referenced method (for $N = 20, 40, 80, 100$) and for the PINN results at $t = 0.5$. The errors of the PINN solution are comparable to those reported in [38]. However, it can be observed that for $x > 0.5$, the errors obtained with PINN are smaller.

Table 7. Comparison of the numerical solution (BDF-1) with the exact solution and PINN method at different space points for $t = 0.5$.

x	$N = 20$	$N = 40$	$N = 80$	$N = 100$	Exact Solution	PINN
0.1	2.281×10^{-3}	2.271×10^{-3}	2.268×10^{-3}	2.268×10^{-3}	2.213×10^{-3}	2.315×10^{-3}
0.2	4.339×10^{-3}	4.319×10^{-3}	4.315×10^{-3}	4.314×10^{-3}	4.210×10^{-3}	4.382×10^{-3}
0.3	5.973×10^{-3}	5.947×10^{-3}	5.940×10^{-3}	5.939×10^{-3}	5.796×10^{-3}	5.998×10^{-3}
0.4	7.024×10^{-3}	6.993×10^{-3}	6.985×10^{-3}	6.984×10^{-3}	6.816×10^{-3}	7.018×10^{-3}
0.5	7.388×10^{-3}	7.356×10^{-3}	7.347×10^{-3}	7.347×10^{-3}	7.169×10^{-3}	7.348×10^{-3}

Table 7. *Cont.*

x	$N = 20$	$N = 40$	$N = 80$	$N = 100$	Exact Solution	PINN
0.6	7.029×10^{-3}	6.998×10^{-3}	6.990×10^{-3}	6.989×10^{-3}	6.821×10^{-3}	6.965×10^{-3}
0.7	5.982×10^{-3}	5.955×10^{-3}	5.948×10^{-3}	5.948×10^{-3}	5.804×10^{-3}	5.910×10^{-3}
0.8	4.347×10^{-3}	4.328×10^{-3}	4.323×10^{-3}	4.322×10^{-3}	4.218×10^{-3}	4.285×10^{-3}
0.9	2.286×10^{-3}	2.276×10^{-3}	2.273×10^{-3}	2.273×10^{-3}	2.218×10^{-3}	2.246×10^{-3}

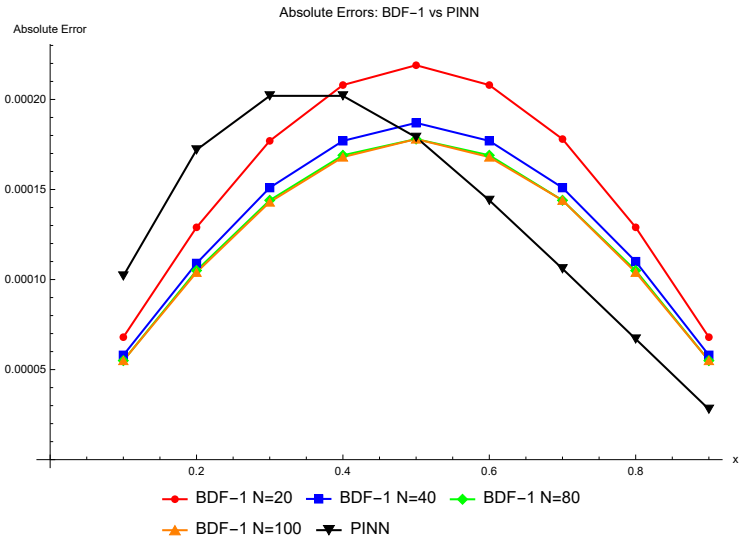


Figure 11. Comparison of absolute errors of the BDF-1 method for different values of N with the PINN method as a function of the spatial coordinate x at $t = 0.5$.

Another test involved examining how the weights in the loss function on the PDE and the initial–boundary conditions affected the obtained results. For this purpose, a set of several weights was assumed, and after training the model, the mean error computed on a 100×100 grid was evaluated (see Table 8). In this test, the network architecture and hyperparameters were as follows: hidden layers: 5; neurons per layer: 10; Adam optimizer; collocation points inside domain: 200; boundary/initial condition points: 128; number of iterations: 10,000. It is also assumed that $x, t \in [0, 1]$.

Table 8. The effect of loss function weights on the mean and maximum error of the results.

Weights $(\lambda_{PDE}, \lambda_{IC}, \lambda_{BC_{left}}, \lambda_{BC_{right}})$	Mean Error	Max Error
(5.0, 1.0, 1.0, 1.0)	0.0126755	0.0766495
(1.0, 5.0, 1.0, 1.0)	0.00730921	0.0546295
(1.0, 1.0, 5.0, 5.0)	0.00579474	0.0411806
(10.0, 1.0, 1.0, 1.0)	0.0111069	0.103218
(1.0, 10.0, 10.0, 10.0)	0.00258062	0.0196944
(1.0, 5.0, 5.0, 5.0)	0.00197403	0.0177932

The weights assigned to individual components of the loss function have a moderate impact on the obtained errors. In particular, it can be clearly observed that when the weights for the initial–boundary conditions are several times larger than those for the PDE inside the domain, the resulting solution errors are significantly reduced.

The effect of the distribution of collocation points on the mean and max errors in the domain, computed on a 100×100 grid, was also examined (see Table 9). The impact of the number of collocation points inside domain on this error was also investigated (see Table 10). In these tests, equal weights were used, while the settings of the remaining parameters and the network architecture are the same as in the previous test.

Table 9. Effect of collocation point distribution type on the mean and maximum error.

Collocation Point Distribution Type	Mean Error	Max Error
uniform (equispaced grid)	0.00511845	0.0278437
pseudo (pseudorandom)	0.00418709	0.029266
LHS (Latin hypercube sampling)	0.00696592	0.0361623
Halton (Halton sequence)	0.00759538	0.0478001
Hammersley (Hammersley sequence)	0.00372326	0.0329703
Sobol (Sobol sequence)	0.00485239	0.0341695

Table 10. Effect of the number of collocation points on the mean and maximum error.

Number of Collocation Points	Mean Error	Max Error	Time of Training Model [s]
50	0.00541526	0.0308579	25.12
100	0.00516718	0.0317111	28.76
500	0.00630318	0.0403584	31.06
1000	0.00418513	0.0254188	34.44
10,000	0.00140849	0.0123031	120.84

The differences in the obtained errors when changing the distribution of collocation points are minor. The smallest errors were obtained using the Hammersley method. When increasing the number of collocation points, the obtained errors were slightly smaller. For example, taking 10,000 points results in the max error ≈ 0.012, while for 100 points it was ≈ 0.031. However, even with 50 points, the results are satisfactory. Increasing the number of collocation points only slightly reduces the errors while extending the computation time. Additionally, the training times of the model are provided in Table 10. Increasing the number of collocation points leads to longer computation times; however, the growth is not linearly proportional. For example, for 1000 collocation points, the computation time was 34 s, whereas for 10,000 points, the model required approximately 120 s to train.

The final test conducted on this example was to examine the impact of the network architecture (the number of hidden layers and neurons per layer) on the obtained errors. The results of this test are shown in Table 11. The last column in Table 11, named Params, refers to the total number of network parameters (the number of weights and biases). An insufficient number of hidden layers or neurons in the network is not enough for the model to provide satisfactory predictions. The mean and maximum errors for a single hidden layer are unsatisfactory, as are the errors for two hidden layers with five neurons. Moving from one to two layers yields the largest quality jump (e.g., with 10 neurons Mean Error drops 0.097 to 0.0038 at similar training time). The best results were obtained with six layers and 30 neurons. Increasing the number of hidden layers and neurons has a significant impact on the computation time. The training time grows almost linearly with the number of parameters (correlation ≈ 0.93). Large and deep models may require significantly longer training time without a guaranteed improvement in error. The results indicate that some deeper yet still moderately wide models (e.g., 6×30) are more efficient per parameter and per unit of time than very wide ones (e.g., 10×100, 20×100), which suffer from optimization difficulties. Based on the data in the Table, Figures 12 and 13 show the mean error as a function of the number of network parameters, as well as the computation time as a function of the number of parameters. The solution obtained from PINN for best case is presented in Figure 14. Figure 15 shows the error distribution obtained from the PINN model for selected network architecture settings. The smallest errors were obtained with six hidden layers and 30 neurons (Figure 15c), while the largest errors occurred with one hidden layer and 10 neurons (Figure 15a).

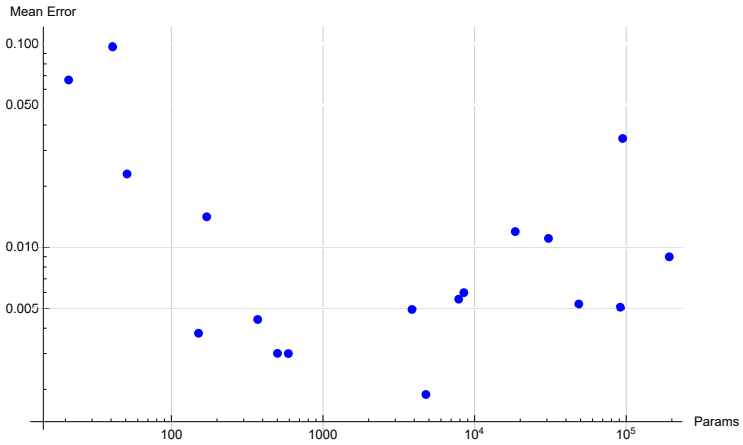


Figure 12. Mean error as a function of the number of network parameters (Burgers' equation).

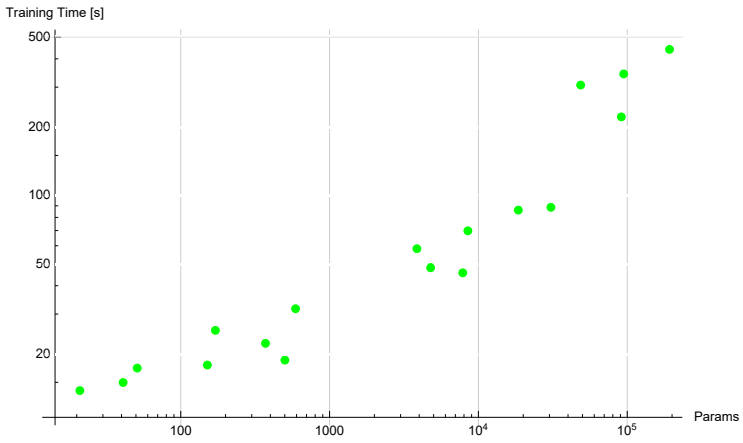


Figure 13. Training time as a function of the number of network parameters (Burgers' equation).

Table 11. Performance of different hyperparameter combinations concerning number of hidden layers and neurons.

Neurons	Hidden Layers	Mean Error	Max Error	Training Time [s]	Params
5	1	0.0666834	0.436997	13.79	21
10	1	0.0969972	0.445789	14.97	41
5	2	0.0229704	0.147473	17.33	51
10	2	0.0037768	0.045616	17.88	151
20	2	0.0030077	0.024509	18.79	501
10	4	0.0044155	0.035810	22.28	371
50	4	0.0055573	0.029546	45.59	7851
100	4	0.0110681	0.020511	88.62	30,701
5	6	0.0141401	0.084471	25.42	171
10	6	0.0029999	0.018322	31.66	591
30	6	0.0018869	0.017416	48.00	4771
60	6	0.0119612	0.058198	86.14	18,541
20	10	0.0049465	0.024834	58.23	3861
30	10	0.0059858	0.033673	69.82	8491
100	10	0.0050738	0.059602	221.90	91,301
50	20	0.0052591	0.034503	306.45	48,651
70	20	0.0343433	0.0484721	343.27	94,711
100	20	0.0089822	0.0848633	440.17	192,301

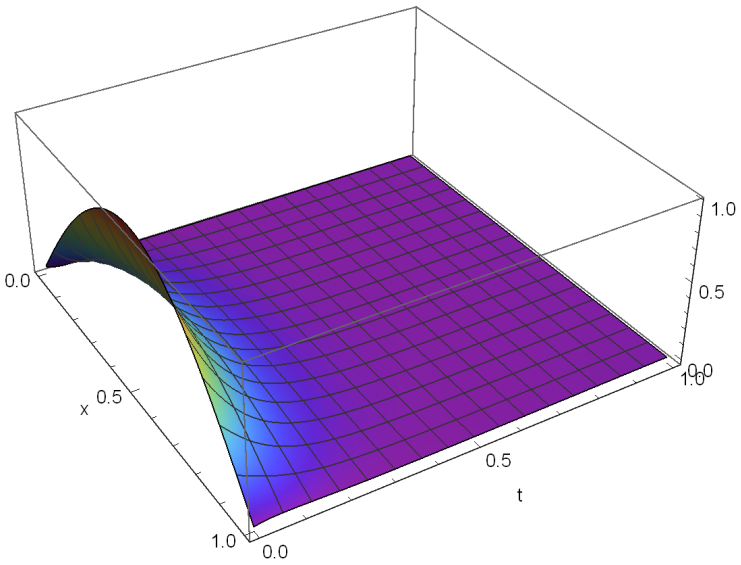
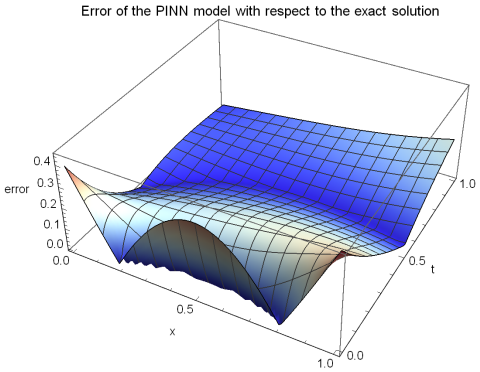
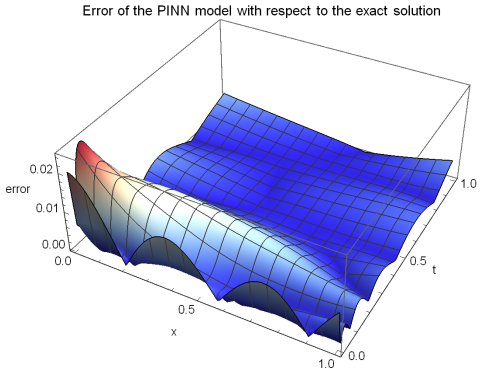


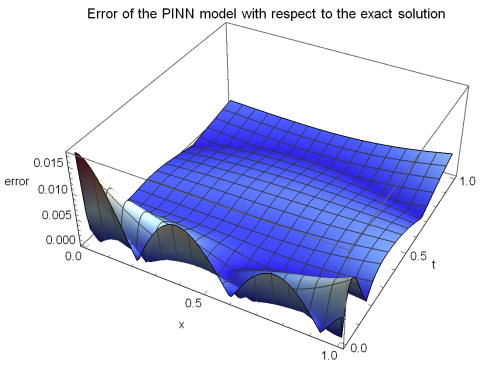
Figure 14. Prediction from PINN for 6 hidden layers and 30 neurons (Burgers' equation).



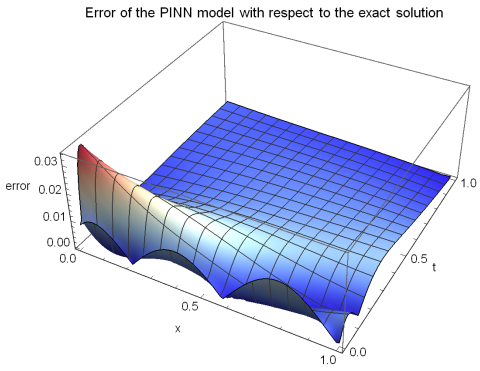
(**a**) 1 hidden layer with 10 neurons



(**b**) 2 hidden layers with 20 neurons



(**c**) 6 hidden layers with 30 neurons



(**d**) 10 hidden layers with 30 neurons

Figure 15. Error distribution obtained from the PINN for selected hyperparameters concerning the number of hidden layers and neurons (Burgers' equation).

The results demonstrate that physics-informed neural networks (PINNs) can effectively solve the Burgers' equation with high accuracy as compared to traditional methods like the explicit and exact-explicit schemes. PINNs produced smaller approximation errors across most of the domain.

3.3. Volterra Integro-Differential Equation

Volterra integro-differential equations (VIDEs) are a class of functional equations that combine differential and integral operations on an unknown function. A standard form of a first-order VIDE is as follows:

$$\frac{d}{dt}y(t) = f(t, y(t)) + \int_0^t K(t, s, y(s))\, ds,$$

where $K(t, s, y(s))$ is a known kernel function, and f defines the local dynamics. VIDEs naturally arise in systems where the future state depends not only on the current state but also on the historical evolution of the system [39].

Volterra integro-differential equations appear in various scientific and engineering contexts such as population dynamics, heat transfer in materials with memory, neuroscience, finance, etc. A detailed analysis and overview on these models can be seen in the works [40,41]. Thus, these equations are crucial for modeling systems where historical data fundamentally influences future dynamics.

Traditional numerical techniques, such as quadrature methods, finite difference schemes, or collocation methods, often face difficulties due to high computational cost from evaluating the integral term at each time step, stability issues over long time intervals, and accuracy demands for capturing both differential and integral contributions. These challenges intensify for higher-order VIDEs due to the presence of multiple derivatives and their interactions with history-dependent terms. PINNs offer a modern approach by embedding the physical laws into the loss function of a neural network.

In this section we are going to solve a fourth order Volterra integro-differential equation using PINNs and then compare the results with a traditional numerical method where the same equation is solved using variational iteration method with collocation. First we will describe this method and see its convergence analysis and then compare the results through figures and tables. The equation is solved by Otaide et al. in their work on "Numerical treatment of linear Volterra integro differential equations using variational iteration algorithm with collocation" [42]. The equation is described as

$$\begin{cases} y^{(iv)}(t) = 1 + t - \dfrac{t^2}{2} - \dfrac{t^3}{6} + \displaystyle\int_0^t (t - s) y(s)\, ds, \\ y(0) = 2, \quad y'(0) = 2, \quad y''(0) = 1, \quad y'''(0) = 1. \end{cases} \tag{4}$$

The exact solution for this problem is as follows:

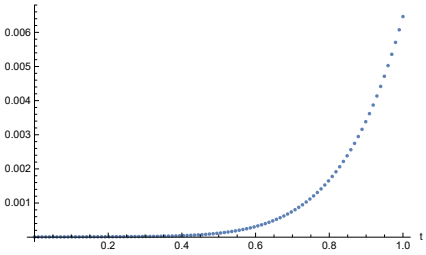
$$y(t) = \exp(t) + t + 1.$$

In the work of Otaide et al. [42], the equation is solved using fourth-kind Chebyshev polynomials combined with the variational iteration method and a collocation technique, resulting in a hybrid approach that integrates variational iteration with collocation. The method is briefly described in Section 3.3.1.

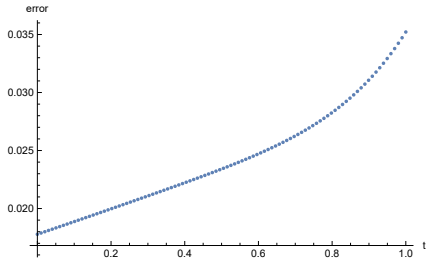
Before comparing the results obtained with PINNs to those from the method described in Section 3.3.1, computations were performed on various network architectures (see Table 12). Similar to the previous examples, the lowest mean errors were achieved by moderate architectures: 2×20 (Mean $= 9.49 \times 10^{-4}$), 2×10 (Mean $= 9.61 \times 10^{-4}$), and 6×50 (Mean $= 8.91 \times 10^{-4}$). Analogous to the previously analyzed examples, Figure 16 presents the error distribution at the grid points for four sample cases from Table 12. The solution obtained with PINN using six hidden layers and 50 neurons is shown in Figure 17.

Table 12. Performance of different hyperparameter combinations for VIDE.

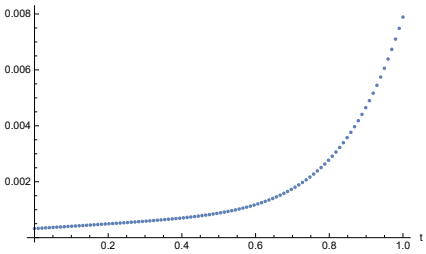
Neurons	Hidden Layers	Mean Error	Max Error	Training Time [s]	Params
5	1	0.00111956	0.00684174	13.32	16
10	1	0.00102483	0.00675019	14.14	31
5	2	0.00174907	0.00788888	20.45	46
10	2	0.000961112	0.00644263	26.92	141
20	2	0.000949299	0.00646266	38.24	481
50	4	0.000996002	0.00644883	335.16	7801
100	4	0.0242298	0.0352155	706.06	30,601
5	6	0.00109214	0.00678214	129.42	166
10	6	0.00102154	0.00660905	314.78	581
50	6	0.000891232	0.00629672	663.80	12,901
20	10	0.0424468	0.0460251	1036.78	3841
30	10	0.00881867	0.016485	1168.29	8461
100	10	0.00157222	0.00746592	1870.17	91,201



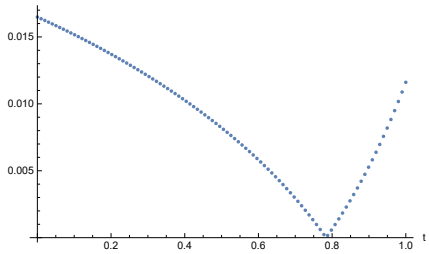
(**a**) 2 hidden layer with 20 neurons



(**b**) 4 hidden layers with 100 neurons



(**c**) 2 hidden layers with 5 neurons



(**d**) 10 hidden layers with 30 neurons

Figure 16. Error distribution obtained from the PINN for selected hyperparameters concerning the number of hidden layers and neurons (Volterra equation).

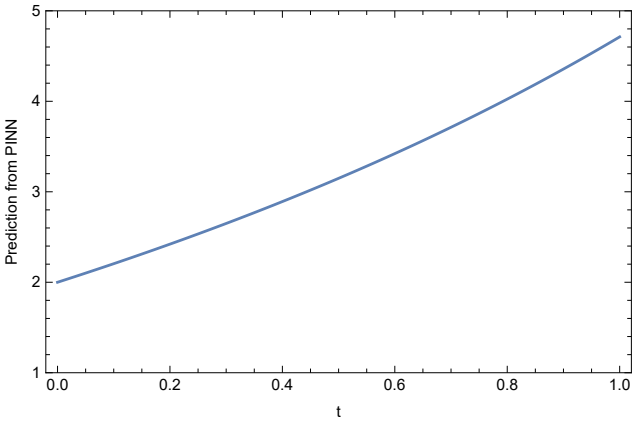


Figure 17. Prediction from PINN for 6 hidden layers and 50 neurons (VIDE).

3.3.1. The Standard Variational Iteration Method Combined with Shifted Chebyshev Polynomials of the Fourth Kind

Consider a Volterra integro-differential equation of the form

$$y^{(n)}(t) = f(t) + \int_0^t K(t,s)y(s)\,ds, \tag{5}$$

where $y^{(n)}(t)$ is the unknown function, $f(t)$ is a known forcing term, and $K(t,s)$ is the kernel. The correction functional for Variational Iteration Method (VIM) is

$$y_{m+1}(t) = y_m(t) + \int_0^t \lambda(s)\left(y_m^{(n)}(s) - f(s) - \int_0^s K(s,\tau)y_m(\tau)\,d\tau\right)ds,$$

with $\lambda(s)$ as the Lagrange multiplier determined optimally using variational principles. The subscript m indicates the mth iteration, and $y_m(t)$ is considered as a restricted variation ($\delta y_m = 0$) during the optimization.

To discretize the problem, we apply the standard collocation technique by choosing collocation points uniformly distributed over $[a,b]$

$$t_i = a + \frac{(b-a)i}{N}, \quad i = 1,2,\ldots,N,$$

where N denotes the number of collocation points. This discretization converts the continuous problem into a system of algebraic equations.

For better approximation, we expand the solution using Chebyshev polynomials of the fourth kind $W_n(t)$, which are orthogonal with respect to the weight function

$$w(t) = \sqrt{\frac{1-t}{1+t}}, \quad t \in [-1,1].$$

These polynomials satisfy the recurrence relation

$$W_{n+1}(t) = 2tW_n(t) - W_{n-1}(t), \quad W_0(t) = 1, \quad W_1(t) = 2t+1.$$

Chebyshev polynomials exhibit excellent approximation properties, such as rapid convergence and minimization of the maximum error.

To adapt Chebyshev polynomials to the interval $[0,1]$, we employ the shifted Chebyshev polynomials $W_n^*(t)$ defined by the following:

$$W_n^*(t) = W_n(2t-1),$$

with the following recurrence relation:

$$W_{n+1}^*(t) = 2(2t-1)W_n^*(t) - W_{n-1}^*(t), \quad W_0^*(t) = 1, \quad W_1^*(t) = 4t-1.$$

These shifted polynomials retain the favorable properties of their unshifted counterparts while matching the domain of the problem. The hybrid method approximates the solution as

$$y_{m,N}(t) = \sum_{k=0}^{N-1} c_{k,N}W_k^*(t),$$

and iteratively refines it using

$$y_{m+1,N}(t) = \sum_{k=0}^{N-1} c_{k,N}W_k^*(t) + \int_0^t \lambda(s)\left(\frac{d^n y_{m,N}}{ds^n} - f(s) - \int_0^s K(s,\tau)y_{m,N}(\tau)\,d\tau\right)ds,$$

where $c_{k,N}$ are coefficients to be determined.

3.3.2. Convergence of the Method

When solving a differential equation numerically using the iteration methods, it is important to prove that the method actually converges, i.e., the sequence of approximations $y_m(t)$ produced by the method will become closer and closer to the true solution as $m \to \infty$. This analysis—in this case—relies on the Banach's Fixed-Point Theorem.

Theorem 5. *Let X be a Banach space with norm $\|\cdot\|$, and let $\mathcal{F} : X \to X$ be the linear operator defined by the following:*

$$\mathcal{F}[y] = y_{m,N}(t) + \int_0^t \lambda(s)\left(\frac{d^n y_m}{ds^n} - f(s) - \int_0^s K(s,\tau)y_m(\tau)\,d\tau\right)ds,$$

where $y_{m,N}(t) = \sum_{k=0}^{N-1} c_{k,N} W_k^*(t)$ is the approximate solution using shifted Chebyshev polynomials W_k^*. If $\mathcal{F}$ is a contraction, i.e., there exists $\zeta \in [0,1)$ such that

$$\|\mathcal{F}[y] - \mathcal{F}[\tilde{y}]\| \leq \zeta \|y - \tilde{y}\| \quad \forall y, \tilde{y} \in X,$$

then

1. *$\mathcal{F}$ has a unique fixed point $y^* \in X$.*
2. *The sequence $\{y_{m,N}\}$ generated by $y_{m+1,N} = \mathcal{F}[y_{m,N}]$ converges to y^* for any initial guess $y_{0,N}$.*

Proof. By the Banach Fixed-Point Theorem, $\mathcal{F}$ has a unique fixed point y^* since X is complete and $\mathcal{F}$ is a contraction. For $p > q \geq 1$, the triangle inequality and contraction property yield

$$\|y_p - y_q\| \leq \sum_{k=q}^{p-1} \|y_{k+1} - y_k\| \leq \left(\zeta^q + \cdots + \zeta^{p-1}\right)\|y_1 - y_0\|.$$

The series converges as $\zeta < 1$, so $\{y_{m,N}\}$ is Cauchy and thus convergent. At the fixed point y^*, the iteration reduces to

$$y^*(t) = y^*(t) + \int_0^t \lambda(s)\left(\frac{d^n y^*}{ds^n} - f(s) - \int_0^s K(s,\tau)y^*(\tau)\,d\tau\right)ds,$$

implying $\frac{d^n y^*}{dt^n} = f(t) + \int_0^t K(t,\tau)y^*(\tau)\,d\tau$. Hence, y^* solves the VIDE. □

Proposition 1. *According to Theorem 5, for the linear mapping $\mathcal{F}$ defined as the following:*

$$\mathcal{F}[y] = y_{m,N}(t) + \int_0^t \lambda(t)(Ly_m(t) + N\widetilde{y_m(t)} - g(t))dt$$

or equivalently

$$\mathcal{F}[y] = \sum_{m=0}^{N} c_{m,N} W_{m,N}^*(t) + \int_0^t \lambda(t)\left(\frac{d^n y}{ds^n} - f(s) - \int_0^t K(s,t)y(t)dt\right)ds,$$

this is a necessary condition for the variational iteration approach to converge. The sequence $y_{n+1} = \mathcal{F}[y_n]$ converges to a fixed point of $\mathcal{F}$ which is also a solution of (5).

3.3.3. Comparison of Results

Now we show the comparison between results of Equation (5) solved through the standard variational iteration method in [42] and through PINNs. Solving Volterra integro-differential equations by PINNs, the integral term is approximated via numerical quadrature. Gaussian quadrature with a prescribed degree to approximate the integral was used. In Figure 18a the plot shows excellent agreement between the approximate solutions and the exact analytical solution (in case of PINNs) over the interval $t \in [0, 1]$. Figure 18b illustrates the absolute error between the approximate and exact solutions, showing that the error remains minimal throughout the domain, with a slight increase near the upper boundary.

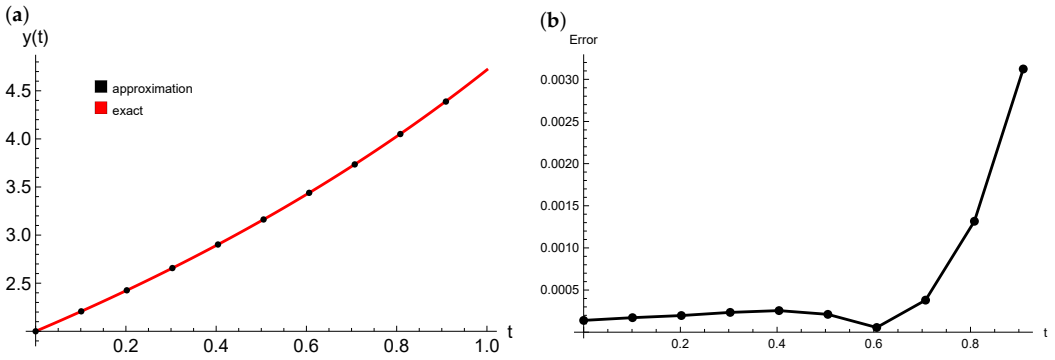


Figure 18. Plots of the approximated and exact solution (**a**) and the absolute approximation error (**b**).

The results are obtained using 50 training points and three layers of neurons, each with eight neurons, while the activation function is tanh.

In Table 13, we provide numerical values of the solution at selected time points. It compares the results from the variational iteration method [42], PINNs, and the exact solution. It can be seen that at $t = 0.0$ both solutions perfectly match the exact solution. The solutions [42] are generally better than PINNs for $t \leq 0.5$ while for larger t values PINNs unexpectedly outperform solutions [42].

Table 13. Comparison of approximate and exact solutions.

t	Solutions [42]	PINNs	Exact Solution
0.0	2.00000	2.00014	2.00000
0.1	2.20517	2.20747	2.20730
0.2	2.42140	2.42609	2.42589
0.3	2.64986	2.65722	2.65699
0.4	2.89182	2.90216	2.90190
0.5	3.14867	3.16233	3.16212
0.6	3.42200	3.43931	3.43925
0.7	3.71440	3.73473	3.73511
0.8	4.03267	4.05036	4.05168
0.9	4.39798	4.38803	4.39115
1.0	4.87221	4.71233	4.71828

Comparing in terms of error metrics we can see that in Table 14 the mean and median errors for PINNs are an order of magnitude smaller than the reference solutions [42]. For example, PINNs' mean error 0.000609 is $\sim$27 times smaller than the reference error, i.e, 0.016428. The standard deviation for PINNs is much lower, which testifies that it has more consistent performance as compared to the reference solutions [42].

Table 14. Error metrics comparison between PINNs and solutions [42].

Metric	PINNs Error	Solutions [42] Error
Mean	0.000609	0.016428
Median	0.000223	0.011769
Standard Deviation	0.000953	0.016485
Mean Absolute Error	0.000609	0.016428
Root Mean Square Error	0.001090	0.022682

In Figure 19, we give a comparative analysis of the absolute errors between the PINNs and the reference solutions [42]. In Figure 19a, the line plot shows that the PINNs approach consistently maintains a significantly lower error across the entire time interval compared to solutions [42]. The error for the reference method [42] grows steadily with time and reaches above 0.05 near $t = 0.9$, whereas the error for the PINNs solution remains very close to zero.

In Figure 19b, the boxplot provides a statistical summary of the error distributions. The median error and interquartile range for the PINNs is lower than those of reference solutions [42]. Moreover, the PINNs error exhibits minimal spread and lower maximum error.

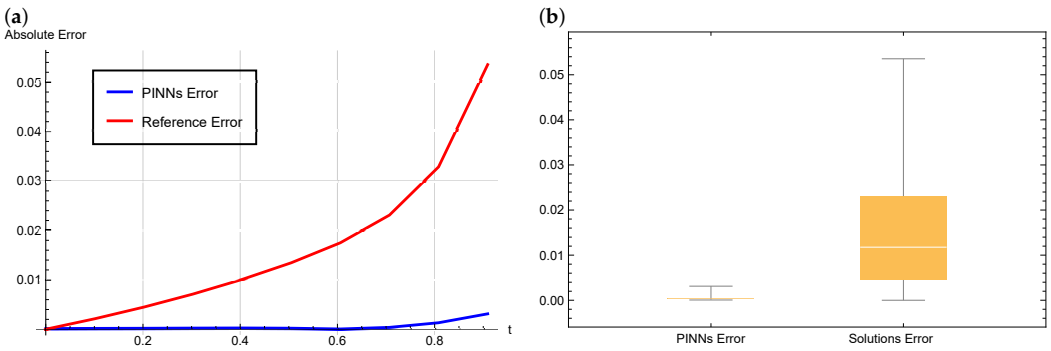


Figure 19. Illustration of comparative analysis of the absolute errors, (**a**) absolute error for the reference solution and that obtained with PINN; (**b**) boxplot of the statistical summary of the error distributions.

These comparisons clearly demonstrate that the PINNs-based approach outperforms the traditional method in both average accuracy and robustness, making it a more reliable choice for solving these kind of Volterra integro-differential equations.

4. Conclusions

This work investigated the application of physics-informed neural networks (PINNs) to solve the Poisson equation, Burgers' equation, and a fourth-order Volterra integro-differential equation. Comparative analysis with traditional numerical methods such as finite difference schemes and variational iteration techniques highlighted the flexibility and accuracy of PINNs, especially in scenarios where classical approaches encounter limitations. A systematic study of key hyperparameters—including the distribution of collocation points, network architecture, and activation functions—revealed their significant influence on both training efficiency and solution accuracy. Theoretical results further demonstrated that, for second-order linear elliptic and parabolic PDEs, PINN solutions converge strongly to the exact solution in the C^0 norm, and in the H^1 norm when boundary or initial conditions are enforced at all collocation points.

While the findings support the potential of PINNs as a viable alternative to conventional numerical solvers, challenges such as sensitivity to hyperparameter choices and high computational cost persist. All experiments were conducted using the DeepXDE

library [43], which proved to be a reliable platform for model implementation and evaluation. Future work may focus on improving training strategies, automating hyperparameter tuning, and extending the framework to more complex or high-dimensional PDEs and inverse problems.

Author Contributions: Conceptualization, R.B., M.P. and D.A.M.; methodology, R.B., M.P. and D.A.M.; software, R.B., M.P. and D.A.M.; validation, R.B., M.P. and D.A.M.; investigation, R.B., M.P. and D.A.M.; writing—original draft preparation, R.B., M.P. and D.A.M.; writing—review and editing, R.B., M.P. and D.A.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author(s).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Raissi, M.; Perdikaris, P.; Karniadakis, G.E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.* **2019**, *378*, 686–707. [CrossRef]
2. Raissi, M.; Perdikaris, P.; Karniadakis, G.E. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. *arXiv* **2017**, arXiv:1711.10561. [CrossRef]
3. Raissi, M.; Perdikaris, P.; Karniadakis, G.E. Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations. *arXiv* **2017**, arXiv:1711.10566. [CrossRef]
4. Cuomo, S.; Di Cola, V.S.; Giampaolo, F.; Rozza, G.; Raissi, M.; Piccialli, F. Scientific machine learning through physics–informed neural networks: Where we are and what's next. *J. Sci. Comput.* **2022**, *92*, 88.
5. Brociek, R.; Pleszczyński, M. Differential Transform Method and Neural Network for Solving Variational Calculus Problems. *Mathematics* **2024**, *12*, 2182. [CrossRef]
6. Faroughi, S.A.; Pawar, N.M.; Fernandes, C.; Raissi, M.; Das, S.; Kalantari, N.K.; Kourosh Mahjour, S. Physics-Guided, Physics-Informed, and Physics-Encoded Neural Networks and Operators in Scientific Computing: Fluid and Solid Mechanics. *J. Comput. Inf. Sci. Eng.* **2024**, *24*, 040802 . [CrossRef]
7. Usama, M.; Ma, R.; Hart, J.; Wojcik, M. Physics-Informed Neural Networks (PINNs)-Based Traffic State Estimation: An Application to Traffic Network. *Algorithms* **2022**, *15*, 447. [CrossRef]
8. Wang, S.; Wang, H.; Perdikaris, P. On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks. *Comput. Methods Appl. Mech. Eng.* **2021**, *384*, 113938. [CrossRef]
9. Yuan, L.; Ni, Y.Q.; Deng, X.Y.; Hao, S. A-PINN: Auxiliary physics informed neural networks for forward and inverse problems of nonlinear integro-differential equations. *J. Comput. Phys.* **2022**, *462*, 111260. [CrossRef]
10. Xiang, Z. ; Peng, W.; Liu, X.; Yao, W. Self-adaptive loss balanced Physics-informed neural networks. *Neurocomputing* **2022**, *496*, 11–34. [CrossRef]
11. Dwivedi, V.; Parashar, N.; Srinivasan, B. Distributed learning machines for solving forward and inverse problems in partial differential equations. *Neurocomputing* **2021**, *420*, 299–316. [CrossRef]
12. Lee, J. Anti-derivatives approximator for enhancing physics-informed neural networks. *Comput. Methods Appl. Mech. Eng.* **2024**, *426*, 117000. [CrossRef]
13. McClenny, L.D.; Braga-Neto, U.M. Self-adaptive physics-informed neural networks. *J. Comput. Phys.* **2023**, *474*, 111722. [CrossRef]
14. Brociek, R.; Pleszczyński, M. Differential Transform Method (DTM) and Physics-Informed Neural Networks (PINNs) in Solving Integral–Algebraic Equation Systems. *Symmetry* **2024**, *16*, 1619. [CrossRef]
15. Ren, Z.; Zhou, S.; Liu, D.; Liu, Q. Physics-Informed Neural Networks: A Review of Methodological Evolution, Theoretical Foundations, and Interdisciplinary Frontiers Toward Next-Generation Scientific Computing. *Appl. Sci.* **2025**, *15*, 92. [CrossRef]
16. Lawal, Z.K.; Yassin, H.; Lai, D.T.C.; Che Idris, A. Physics-Informed Neural Network (PINN) Evolution and Beyond: A Systematic Literature Review and Bibliometric Analysis. *Big Data Cogn. Comput.* **2022**, *6*, 140. [CrossRef]
17. Coutinho, E.J.R.; Dall'Aqua, M.; McClenny, L.; Zhong, M.; Braga-Neto, U.; Gildin, E. Physics-informed neural networks with adaptive localized artificial viscosity. *J. Comput. Phys.* **2023**, *489*, 112265. [CrossRef]
18. Diao, Y.; Yang, J.; Zhang, Y.; Zhang, D.; Du, Y. Solving multi-material problems in solid mechanics using physics-informed neural networks based on domain decomposition technology. *Comput. Methods Appl. Mech. Eng.* **2023**, *413*, 116120. [CrossRef]
19. Lazovskaya, T.; Malykhina, G.; Tarkhov, D. Physics-Based Neural Network Methods for Solving Parameterized Singular Perturbation Problem *Computation* **2021**, *9*, 97. [CrossRef]

20. Jagtap, A.D.; Mao, Z.; Adams, N.; Karniadakis, G.E. Physics-informed neural networks for inverse problems in supersonic flows. *J. Comput. Phys.* **2022**, *466*, 111402. [CrossRef]
21. Hornik, K.; Stinchcombe, M.; White, H. Multilayer feedforward networks are universal approximators. *Neural Netw.* **1989**, *2*, 359–366. [CrossRef]
22. Baydin, A.G.; Pearlmutter, B.A.; Radul, A.A.; Siskind, J.M. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.* **2018**, *18*, 1–43.
23. Uddin, Z.; Ganga, S.; Asthana, R.; Ibrahim, W. Wavelets based physics informed neural networks to solve non-linear differential equations. *Sci. Rep.* **2023**, *13*, 2882. [CrossRef] [PubMed]
24. Shin, Y.; Darbon, J.; Karniadakis, G.E. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs. *arXiv* **2020**, arXiv:2004.01806. [CrossRef]
25. Doumèche, N.; Biau, G.; Boyer, C. Convergence and error analysis of PINNs. *arXiv* **2023**, arXiv:2305.01240. [CrossRef]
26. Yoo, J.; Lee, H. Robust error estimates of PINN in one-dimensional boundary value problems for linear elliptic equations. *arXiv* **2024**, arXiv:2407.14051. [CrossRef]
27. De Ryck, T.; Mishra, S. Error analysis for physics-informed neural networks (PINNs) approximating Kolmogorov PDEs. *Adv. Comput. Math.* **2022**, *48*, 79. [CrossRef]
28. Kingma, D.P. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980.
29. Holm, D.D. Applications of Poisson geometry to physical problems. *Geom. Topol. Monogr.* **2011**, *17*, 221–384.
30. Nolasco, C.; Jácome, N.; Hurtado-Lugo, N. Applications of the Poisson and diffusion equations to materials science. *J. Phys. Conf. Ser.* **2020**, *1587*, 012014.
31. Klopfenstein, R.; Wu, C. Computer solution of one-dimensional Poisson's equation. *IEEE Trans. Electron Devices* **1975**, *22*, 329–333. [CrossRef]
32. Bhardwaj, S.; Gohel, H.; Namuduri, S. A Multiple-Input Deep Neural Network Architecture for Solution of One-Dimensional Poisson Equation. *IEEE Antennas Wirel. Propag. Lett.* **2019**, *18*, 2244–2248. [CrossRef]
33. Kraichnan, R.H. Lagrangian-history statistical theory for Burgers' equation. *Phys. Fluids* **1968**, *11*, 265–277. [CrossRef]
34. Xie, S.S.; Heo, S.; Kim, S.; Woo, G.; Yi, S. Numerical solution of one-dimensional Burgers' equation using reproducing kernel function. *J. Comput. Appl. Math.* **2008**, *214*, 417–434. [CrossRef]
35. Inan, B.; Bahadir, A.R. Numerical solution of the one-dimensional Burgers' equation: Implicit and fully implicit exponential finite difference methods. *Pramana* **2013**, *81*, 547–556. [CrossRef]
36. Bonkile, M.P.; Awasthi, A.; Lakshmi, C.; Mukundan, V.; Aswin, V. A systematic literature review of Burgers' equation with recent advances. *Pramana* **2018**, *90*, 69. [CrossRef]
37. Kutluay, S.; Bahadir, A.; Özdeş, A. Numerical solution of one-dimensional Burgers equation: Explicit and exact-explicit finite difference methods. *J. Comput. Appl. Math.* **1999**, *103*, 251–261. [CrossRef]
38. Mukundan, V.; Awasthi, A. Efficient numerical techniques for Burgers' equation. *Appl. Math. Comput.* **2015**, *262*, 282–297. [CrossRef]
39. Brunner, H. *Collocation Methods for Volterra Integral and Related Functional Differential Equations*; Cambridge University Press: Cambridge, UK , 2004; Volume 15.
40. Volterra, V. *Leçons sur la Théorie Mathématique de la Lutte pour la Vie*; Gauthier Villars: Paris, France, 1931.
41. Joseph, D.D.; Preziosi, L. Heat waves. *Rev. Mod. Phys.* **1989**, *61*, 41. [CrossRef]
42. Otaide, I.J.; Oluwayemi, M.O. Numerical treatment of linear volterra integro differential equations using variational iteration algorithm with collocation. *Partial. Differ. Equ. Appl. Math.* **2024**, *10*, 100693. [CrossRef]
43. Lu, L.; Meng, X.; Mao, Z.; Karniadakis, G.E. DeepXDE: A deep learning library for solving differential equations. *SIAM Rev.* **2021**, *63*, 208–228. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.