Contributions
1) Overview of PINNs → embeds a PDE into loss of ANN
2) New RAR method to improve PINNs' training
3) DeepXDE to solve different type of PDEs
Residual based adaptive refinement

SIAM REVIEW
Vol. 63, No. 1, pp. 208–228

DeepXDE: A Deep Learning Library for Solving Differential Equations*

Lu Lu†
Xuhui Meng‡
Zhiping Mao§
George Em Karniadakis¶

Abstract. Deep learning has achieved remarkable success in diverse applications; however, its use in solving partial differential equations (PDEs) has emerged only recently. Here, we present an overview of physics-informed neural networks (PINNs), which embed a PDE into the loss of the neural network using automatic differentiation. The PINN algorithm is simple, and it can be applied to different types of PDEs, including integro-differential equations, fractional PDEs, and stochastic PDEs. Moreover, from an implementation point of view, PINNs solve inverse problems as easily as forward problems. We propose a new residual-based adaptive refinement (RAR) method to improve the training efficiency of PINNs. For pedagogical reasons, we compare the PINN algorithm to a standard finite element method. We also present a Python library for PINNs, DeepXDE, which is designed to serve both as an educational tool to be used in the classroom as well as a research tool for solving problems in computational science and engineering. Specifically, DeepXDE can solve forward problems given initial and boundary conditions, as well as inverse problems given some extra measurements. DeepXDE supports complex-geometry domains based on the technique of constructive solid geometry and enables the user code to be compact, resembling closely the mathematical formulation. We introduce the usage of DeepXDE and its customizability, and we also demonstrate the capability of PINNs and the user-friendliness of DeepXDE for five different examples. More broadly, DeepXDE contributes to the more rapid development of the emerging scientific machine learning field.

Key words. education software, DeepXDE, differential equations, deep learning, physics-informed neural networks, scientific machine learning

AMS subject classifications. 65-01, 65-04, 65L99, 65M99, 65N99

DOI. 10.1137/19M1274067

*Received by the editors July 10, 2019; accepted for publication (in revised form) May 8, 2020; published electronically February 4, 2021.
https://doi.org/10.1137/19M1274067
Funding: This work was supported by DOE PhILMs project de-sc0019453, by AFOSR grant FA9550-17-1-0013, and by DARPA-AIRA grant HR00111990025.
†Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (lu_lu@mit.edu)
‡Division of Applied Mathematics, Brown University, Providence, RI 02912 USA (xuhui_meng@brown.edu).
§School of Mathematical Sciences, Xiamen University, Xiamen, Fujian 361005, China (zpmao@xmu.edu.cn).
¶Division of Applied Mathematics, Brown University, Providence, RI 02912 USA, and Pacific Northwest National Laboratory, Richland, WA 99354 USA (George_Karniadakis@brown.edu).

1. Introduction. In the last 15 years, deep learning in the form of deep neural networks has been used very effectively in diverse applications [32], such as computer vision and natural language processing. Despite the remarkable success in these and related areas, deep learning has not yet been widely used in the field of scientific computing. However, more recently, solving partial differential equations (PDEs), e.g., in the standard differential form or in the integral form, via deep learning has emerged as a potentially new subfield under the name of scientific machine learning (SciML) [4]. In particular, we can replace traditional numerical discretization methods with a neural network that approximates the solution to a PDE.

To obtain the approximate solution of a PDE via deep learning, a key step is to constrain the neural network to minimize the PDE residual, and several approaches have been proposed to accomplish this. Compared to traditional mesh-based methods, such as the finite difference method (FDM) and the finite element method (FEM), deep learning could be a mesh-free approach by taking advantage of automatic differentiation [47], and could break the curse of dimensionality [45, 18]. Among these approaches, some can only be applied to particular types of problems, such as image-like input domain [28, 33, 58] or parabolic PDEs [6, 19]. Some researchers adopt the variational form of PDEs and minimize the corresponding energy functional [16, 20]. However, not all PDEs can be derived from a known functional, and thus Galerkin-type projections have also been considered [39]. Alternatively, one could use the PDE in strong form directly [15, 52, 30, 31, 7, 50, 47]; in this form, automatic differentiation could be used directly to avoid truncation errors and the numerical quadrature errors of variational forms. This strong form approach was introduced in [47], coining the term physics-informed neural networks (PINNs). An attractive feature of PINNs is that it can be used to solve inverse problems with minimum change of the code for forward problems [47, 48, 51, 21, 13]. In addition, PINNs have been further extended to solve integro-differential equations (IDEs), fractional differential equations (FDEs) [42], and stochastic differential equations (SDEs) [57, 55, 41, 56].

In this paper, we present various PINN algorithms implemented in the Python library DeepXDE,[1] which is designed to serve both as an educational tool to be used in the classroom as well as a research tool for solving problems in computational science and engineering (CSE). DeepXDE can be used to solve multiphysics problems, and supports complex-geometry domains based on the technique of constructive solid geometry (CSG), hence avoiding tedious and time-consuming computational geometry tasks. By using DeepXDE, from an implementation point of view, time-dependent PDEs can be solved as easily as steady states by only defining the initial conditions. In addition to the main workflow of DeepXDE, users can readily monitor and modify the solution process via `callback` functions, e.g., monitoring the Fourier spectrum of the neural network solution, which can reveal the learning mode of the neural network shown later in Figure 2. Last but not least, DeepXDE is designed to make the user code stay compact and manageable, resembling closely the mathematical formulation.

The paper is organized as follows. In section 2, after briefly introducing deep neural networks and automatic differentiation, we present the algorithm, approximation theory, and error analysis of PINNs, and we make a comparison between PINNs and FEM. We then discuss how to use PINNs to solve IDEs and inverse problems. In addition, we propose the RAR method to improve the training efficiency of PINNs. In section 3, we introduce the usage of our library, DeepXDE, and its customizability.

[1] Source code is published under the Apache License, Version 2.0, on GitHub: https://github.com/lululxvi/deepxde.

In section 4, we demonstrate the capability of PINNs and friendly use of DeepXDE in five different examples. Finally, we conclude the paper in section 5.

2. Algorithm and Theory of Physics-Informed Neural Networks. In this section, we first provide a brief overview of deep neural networks and automatic differentiation and present the algorithm and theory of PINNs for solving PDEs. We then make a comparison between PINNs and FEM and discuss how to use PINNs to solve IDEs and inverse problems. Next we propose RAR, an efficient way to select the residual points adaptively during the training process.

2.1. Deep Neural Networks. Mathematically, a deep neural network is a particular choice of a compositional function. The simplest neural network is the feed-forward neural network (FNN), also called multilayer perceptron (MLP), which applies linear and nonlinear transformations to the inputs recursively. Although many different types of neural networks have been developed in the past decades, such as the convolutional neural network and the recurrent neural network, we consider in this paper FNN, which is sufficient for most PDE problems, and the residual neural network (ResNet), which is easier to train for deep networks. However, it is straightforward to employ other types of neural networks.

Let $\mathcal{N}^L(\mathbf{x}) : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ be an L-layer neural network, or an $(L-1)$-hidden layer neural network, with N_ℓ neurons in the ℓth layer ($N_0 = d_{\text{in}}$, $N_L = d_{\text{out}}$). Let us denote the weight matrix and bias vector in the ℓth layer by $\boldsymbol{W}^\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ and $\mathbf{b}^\ell \in \mathbb{R}^{N_\ell}$, respectively. Given a nonlinear activation function σ, which is applied elementwisely, the FNN is recursively defined as follows:

$$\begin{aligned}
&\text{input layer:} && \mathcal{N}^0(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^{d_{\text{in}}},\\
&\text{hidden layers:} && \mathcal{N}^\ell(\mathbf{x}) = \sigma(\boldsymbol{W}^\ell \mathcal{N}^{\ell-1}(\mathbf{x}) + \boldsymbol{b}^\ell) \in \mathbb{R}^{N_\ell} \quad \text{for} \quad 1 \le \ell \le L-1,\\
&\text{output layer:} && \mathcal{N}^L(\mathbf{x}) = \boldsymbol{W}^L \mathcal{N}^{L-1}(\mathbf{x}) + \boldsymbol{b}^L \in \mathbb{R}^{d_{\text{out}}};
\end{aligned}$$

see also a visualization of a neural network in Figure 1. Commonly used activation functions include the logistic sigmoid $1/(1+e^{-x})$, the hyperbolic tangent (tanh), and the rectified linear unit (ReLU, $\max\{x, 0\}$).

2.2. Automatic Differentiation. In PINNs, it is required to compute the derivatives of the network outputs with respect to the network inputs. There are four possible methods for computing the derivatives [5, 38]: (1) hand-coded analytical derivative; (2) finite difference or other numerical approximations; (3) symbolic differentiation (used in software programs such as *Mathematica*, Maxima, and Maple); and (4) automatic differentiation (AD; also called algorithmic differentiation). In deep learning, the derivatives are evaluated using backpropagation [49], a specialized technique of AD.

Considering the fact that the neural network represents a compositional function, then AD applies the chain rule repeatedly to compute the derivatives. There are two steps in AD: one forward pass to compute the values of all variables, and one backward pass to compute the derivatives. To demonstrate AD, we consider an FNN of only one hidden layer with two inputs, x_1 and x_2, and one output, y:

$$\begin{aligned}
v &= -2x_1 + 3x_2 + 0.5,\\
h &= \tanh v,\\
y &= 2h - 1.
\end{aligned}$$

The forward pass and backward pass of AD for computing the partial derivatives $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ at $(x_1, x_2) = (2, 1)$ are shown in Table 1.

We can see that AD requires only one forward pass and one backward pass to compute all the partial derivatives, no matter what the input dimension is. In contrast, using finite differences computing each partial derivative $\frac{\partial y}{\partial x_i}$ requires two function valuations $y(x_1, \ldots, x_i, \ldots, x_{d_{\text{in}}})$ and $y(x_1, \ldots, x_i + \Delta x_i, \ldots, x_{d_{\text{in}}})$ for some small number Δx_i, and thus in total $d_{\text{in}} + 1$ forward passes are required to evaluate all the partial derivatives. Hence, AD is much more efficient than finite difference when the input dimension is high (see [5, 38] for more details of the comparison between AD and the other three methods). To compute nth-order derivatives, AD can be applied recursively n times. However, this nested approach may lead to inefficiency and numerical instability, and hence other methods, e.g., Taylor-mode AD, have been developed for this purpose [8, 9]. Finally we note that with AD we differentiate the neural network and therefore we can deal with noisy data [42].

Table 1 *Example of AD to compute the partial derivatives* $\frac{\partial y}{\partial x_1}$ *and* $\frac{\partial y}{\partial x_2}$ *at* $(x_1, x_2) = (2, 1)$.

Forward pass	Backward pass
$x_1 = 2$ $x_2 = 1$	$\frac{\partial y}{\partial y} = 1$
$v = -2x_1 + 3x_2 + 0.5 = -0.5$ $h = \tanh v \approx -0.462$	$\frac{\partial y}{\partial h} = \frac{\partial (2h-1)}{\partial h} = 2$ $\frac{\partial y}{\partial v} = \frac{\partial y}{\partial h}\frac{\partial h}{\partial v} = \frac{\partial y}{\partial h}\text{sech}^2(v) \approx 1.573$
$y = 2h - 1 = -1.924$	$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial v}\frac{\partial v}{\partial x_1} = \frac{\partial y}{\partial v} \times (-2) = -3.146$ $\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial v}\frac{\partial v}{\partial x_2} = \frac{\partial y}{\partial v} \times 3 = 4.719$

2.3. Physics-Informed Neural Networks (PINNs) for Solving PDEs. We consider the following PDE parameterized by $\boldsymbol{\lambda}$ for the solution $u(\mathbf{x})$ with $\mathbf{x} = (x_1, \ldots, x_d)$ defined on a domain $\Omega \subset \mathbb{R}^d$:

$$f\left(\mathbf{x}; \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right) = 0, \quad \mathbf{x} \in \Omega, \tag{2.1}$$

with suitable boundary conditions

$$\mathcal{B}(u, \mathbf{x}) = 0 \quad \text{on} \quad \partial\Omega,$$

where $\mathcal{B}(u, \mathbf{x})$ could be Dirichlet, Neumann, Robin, or periodic boundary conditions. For time-dependent problems, we consider time t as a special component of $\mathbf{x}$, and Ω contains the temporal domain. The initial condition can be simply treated as a special type of Dirichlet boundary condition on the spatio-temporal domain.

The algorithm of PINN [31, 47] is shown in Procedure 2.1, and visually in the schematic of Figure 1 solving a diffusion equation $\frac{\partial u}{\partial t} = \lambda \frac{\partial^2 u}{\partial x^2}$ with mixed boundary conditions $u(x, t) = g_D(x, t)$ on $\Gamma_D \subset \partial\Omega$ and $\frac{\partial u}{\partial \mathbf{n}}(x, t) = g_R(u, x, t)$ on $\Gamma_R \subset \partial\Omega$. We explain each step as follows. In a PINN, we first construct a neural network $\hat{u}(\mathbf{x}; \boldsymbol{\theta})$ as a surrogate of the solution $u(\mathbf{x})$, which takes the input $\mathbf{x}$ and outputs a vector with the same dimension as u. Here, $\boldsymbol{\theta} = \{\boldsymbol{W}^\ell, \boldsymbol{b}^\ell\}_{1 \le \ell \le L}$ is the set of all weight matrices and bias vectors in the neural network $\hat{u}$. One advantage of PINNs by choosing neural networks as the surrogate of u is that we can take the derivatives of $\hat{u}$ with respect to its input $\mathbf{x}$ by applying the chain rule for differentiating compositions of

Procedure 2.1 The PINN algorithm for solving differential equations.

Step 1 Construct a neural network $\hat{u}(\mathbf{x};\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$.

Step 2 Specify the two training sets $\mathcal{T}_f$ and $\mathcal{T}_b$ for the equation and boundary/initial conditions.

Step 3 Specify a loss function by summing the weighted L^2 norm of both the PDE equation and boundary condition residuals.

Step 4 Train the neural network to find the best parameters $\boldsymbol{\theta}^*$ by minimizing the loss function $\mathcal{L}(\boldsymbol{\theta};\mathcal{T})$.

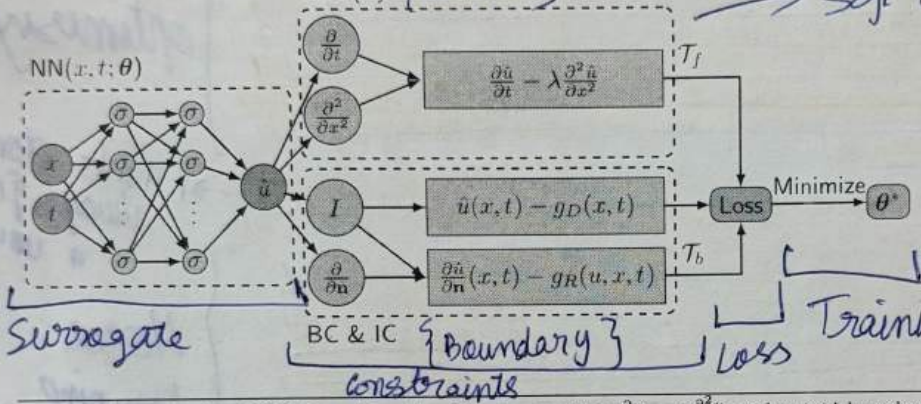


Fig. 1 *Schematic of a PINN for solving the diffusion equation* $\frac{\partial u}{\partial t} = \lambda \frac{\partial^2 u}{\partial x^2}$ *with mixed boundary conditions (BC)* $u(x,t) = g_D(x,t)$ *on* $\Gamma_D \subset \partial\Omega$ *and* $\frac{\partial u}{\partial \mathbf{n}}(x,t) = g_R(u,x,t)$ *on* $\Gamma_R \subset \partial\Omega$. *The initial condition (IC) is treated as a special type of boundary condition.* $\mathcal{T}_f$ *and* $\mathcal{T}_b$ *denote the two sets of residual points for the equation and BC/IC.*

functions using AD, which is conveniently integrated in machine learning packages such as TensorFlow [1] and PyTorch [43].

In the next step, we need to restrict the neural network $\hat{u}$ to satisfy the physics imposed by the PDE and boundary conditions. In practice, we restrict $\hat{u}$ on some scattered points (e.g., randomly distributed points, or clustered points in the domain [37]), i.e., the training data $\mathcal{T} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{|\mathcal{T}|}\}$ of size $|\mathcal{T}|$. In addition, $\mathcal{T}$ comprises two sets, $\mathcal{T}_f \subset \Omega$ and $\mathcal{T}_b \subset \partial\Omega$, which are the points in the domain and on the boundary, respectively. We refer to $\mathcal{T}_f$ and $\mathcal{T}_b$ as the sets of "residual points."

To measure the discrepancy between the neural network $\hat{u}$ and the constraints, we consider the loss function defined as the weighted summation of the L^2 norm of residuals for the equation and boundary conditions:

$$\mathcal{L}(\boldsymbol{\theta};\mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta};\mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta};\mathcal{T}_b), \tag{2.2}$$

where

$$\begin{cases} \mathcal{L}_f(\boldsymbol{\theta};\mathcal{T}_f) = \dfrac{1}{|\mathcal{T}_f|} \displaystyle\sum_{\mathbf{x}\in\mathcal{T}_f} \left\| f\left(\mathbf{x}; \dfrac{\partial \hat{u}}{\partial x_1}, \ldots, \dfrac{\partial \hat{u}}{\partial x_d}; \dfrac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \dfrac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right)\right\|_2^2, \\ \mathcal{L}_b(\boldsymbol{\theta};\mathcal{T}_b) = \dfrac{1}{|\mathcal{T}_b|} \displaystyle\sum_{\mathbf{x}\in\mathcal{T}_b} \|\mathcal{B}(\hat{u},\mathbf{x})\|_2^2, \end{cases}$$

and w_f and w_b are the weights. The loss involves derivatives, such as the partial derivative $\partial \hat{u}/\partial x_1$ or the normal derivative at the boundary $\partial \hat{u}/\partial \mathbf{n} = \nabla \hat{u} \cdot \mathbf{n}$, which are handled via AD.

In the last step, the procedure of searching for a good $\boldsymbol{\theta}$ by minimizing the loss $\mathcal{L}(\boldsymbol{\theta}; \mathcal{T})$ is called "training." Considering the fact that the loss is highly nonlinear and nonconvex with respect to $\boldsymbol{\theta}$ [10], we usually minimize the loss function by gradient-based optimizers such as gradient descent, Adam [29], and L-BFGS [12]. We remark that, based on our experience, for smooth PDE solutions L-BFGS can find a good solution with fewer iterations than Adam, because L-BFGS uses second-order derivatives of the loss function, while Adam relies only on first-order derivatives. However, for stiff solutions L-BFGS is more likely to be stuck at a bad local minimum. The required number of iterations highly depends on the problem (e.g., the smoothness of the solution), and to check whether the network converges or not, we can monitor the loss function or the PDE residual using `callback` functions. We also note that acceleration of training can be achieved by using an adaptive activation function that may remove bad local minima; see [24, 23].

Unlike traditional numerical methods, for PINNs there is no guarantee of unique solutions, because PINN solutions are obtained by solving nonconvex optimization problems, which in general do not have a unique solution. In practice, to achieve a good level of accuracy, we need to tune all the hyperparameters, e.g., network size, learning rate, and the number of residual points. The required network size depends highly on the smoothness of the PDE solution. For example, a small network (e.g., a few layers and twenty neurons per layer) is sufficient for solving the 1D Poisson equation, but a deeper and wider network is required for the 1D Burgers equation to achieve a similar level of accuracy. We also note that PINNs may converge to different solutions from different network initial values [42, 51, 21], and thus a common strategy is that we train PINNs from random initialization a few times (e.g., 10 independent runs) and choose the network with the smallest training loss as the final solution.

In the algorithm of PINN introduced above, we enforce soft constraints of boundary/initial conditions through the loss $\mathcal{L}_b$. This approach can be used for complex domains and any type of boundary conditions. On the other hand, it is possible to enforce hard constraints for simple cases [30]. For example, when the boundary condition is $u(0) = u(1) = 0$ with $\Omega = [0, 1]$, we can simply choose the surrogate model as $u(x) = x(x-1)\mathcal{N}(x)$ to satisfy the boundary condition automatically, where $\mathcal{N}(x)$ is a neural network.

We note that we have great flexibility in choosing the residual points $\mathcal{T}$, and here we provide three possible strategies:

1. We can specify the residual points at the beginning of training, which could be grid points on a lattice or random points, and never change them during the training process.
2. In each optimization iteration, we could select randomly different residual points.
3. We could improve the location of the residual points adaptively during training, e.g., the method proposed in subsection 2.8.

When the number of residual points required is very large, e.g., in multiscale problems, it is computationally expensive to calculate the loss and gradient in every iteration. Instead of using all residual points, we can split the residual points into small batches, and in each iteration we only use one batch to calculate the loss and update model parameters: this is the so-called "mini-batch" gradient descent. The aforementioned

strategy (2), i.e., resampling in each step, is a special case of mini-batch gradient descent by choosing $\mathcal{T} = \Omega$ with $|\mathcal{T}| = \infty$.

Recent studies show that for function approximation, neural networks learn target functions from low to high frequencies [46, 54], but here we show that the learning mode of PINNs is different due to the existence of high-order derivatives. For example, when we approximate the function $f(x) = \sum_{k=1}^{5} \sin(2kx)/(2k)$ in $[-\pi, \pi]$ by a neural network, the function is learned from low to high frequency (Figure 2A). However, when we employ a PINN to solve the Poisson equation $-f_{xx} = \sum_{k=1}^{5} 2k \sin(2kx)$ with zero boundary conditions in the same domain, all frequencies are learned almost simultaneously (Figure 2B). Interestingly, by comparing Figure 2A and Figure 2B we can see that at least in this case solving the PDE using a PINN is faster than approximating a function using a neural network. We can monitor this training process using the `callback` functions in our library DeepXDE as discussed later.

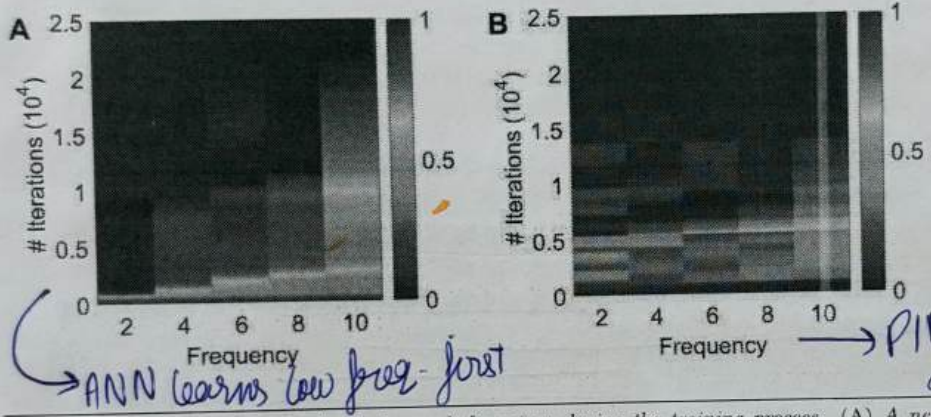


Fig. 2 *Convergence of the amplitude for each frequency during the training process. (A) A neural network is trained to approximate the function $f(x) = \sum_{k=1}^{5} \sin(2kx)/(2k)$. The color represents amplitude values with the maximum amplitude for each frequency normalized to 1. (B) A PINN is used to solve the Poisson equation $-f_{xx} = \sum_{k=1}^{5} 2k \sin(2kx)$ with zero boundary conditions. We use a neural network of 4 hidden layers and 20 neurons per layer. The learning rate is chosen as 10^{-4}, and 500 random points are sampled for training.*

2.4. Approximation Theory and Error Analysis for PINNs. One fundamental question related to PINNs is whether there exists a neural network satisfying both the PDE equation and the boundary conditions, i.e., whether there exists a neural network that can simultaneously and uniformly approximate a function and its partial derivatives. To address this question, we first introduce some notation. Let $\mathbb{Z}_+^d$ be the set of d-dimensional nonnegative integers. For $\mathbf{m} = (m_1, \ldots, m_d) \in \mathbb{Z}_+^d$, we set $|\mathbf{m}| := m_1 + \cdots + m_d$, and

$$D^{\mathbf{m}} := \frac{\partial^{|\mathbf{m}|}}{\partial x_1^{m_1} \ldots \partial x_d^{m_d}}.$$

We say $f \in C^{\mathbf{m}}(\mathbb{R}^d)$ if $D^{\mathbf{k}} f \in C(\mathbb{R}^d)$ for all $\mathbf{k} \le \mathbf{m}$, $\mathbf{k} \in \mathbb{Z}_+^d$, where $C(\mathbb{R}^d) = \{f : \mathbb{R}^d \to \mathbb{R} | f \text{ is continuous}\}$ is the space of continuous functions. Then we recall the following theorem of derivative approximation using single hidden layer neural networks due to Pinkus [44].

THEOREM 2.1. *Let $\mathbf{m}^i \in \mathbb{Z}_+^d$, $i = 1, \ldots, s$, and set $m = \max_{i=1,\ldots,s} |\mathbf{m}^i|$. Assume $\sigma \in C^m(\mathbb{R})$ and that σ is not a polynomial. Then the space of single hidden layer*

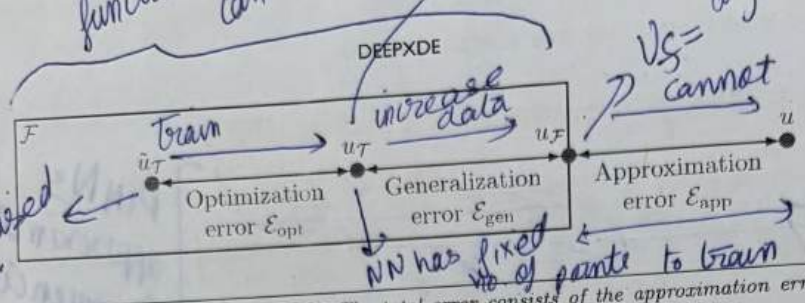


Fig. 3 *Illustration of errors of a PINN. The total error consists of the approximation error, the optimization error, and the generalization error. Here, u is the PDE solution, $u_{\mathcal{F}}$ is the best function close to u in the function space $\mathcal{F}$, $u_{\mathcal{T}}$ is the neural network whose loss is at a global minimum, and $\tilde{u}_{\mathcal{T}}$ is the function obtained by training a neural network.*

neural nets

$$\mathcal{M}(\sigma) := \text{span}\{\sigma(\mathbf{w}\cdot\mathbf{x}+b) : \mathbf{w}\in\mathbb{R}^d, b\in\mathbb{R}\}$$

is dense in

$$C^{\mathbf{m}^1,\dots,\mathbf{m}^s}(\mathbb{R}^d) := \cap_{i=1}^s C^{\mathbf{m}^i}(\mathbb{R}^d),$$

i.e., for any $f\in C^{\mathbf{m}^1,\dots,\mathbf{m}^s}(\mathbb{R}^d)$, any compact $K\subset\mathbb{R}^d$, and any $\varepsilon>0$, there exists a $g\in\mathcal{M}(\sigma)$ satisfying

$$\max_{\mathbf{x}\in K}|D^{\mathbf{k}}f(\mathbf{x}) - D^{\mathbf{k}}g(\mathbf{x})| < \varepsilon$$

for all $\mathbf{k}\in\mathbb{Z}_+^d$ for which $\mathbf{k}\le\mathbf{m}^i$ for some i.

Theorem 2.1 shows that feed-forward neural nets (FNNs) with enough neurons can simultaneously and uniformly approximate any function and its partial derivatives. However, neural networks in practice have limited size. Let $\mathcal{F}$ denote the family of all the functions that can be represented by our chosen neural network architecture. The solution u is unlikely to belong to the family $\mathcal{F}$, and we define $u_{\mathcal{F}} = \arg\min_{f\in\mathcal{F}}\|f-u\|$ as the best function in $\mathcal{F}$ close to u (Figure 3). Because we only train the neural network on the training set $\mathcal{T}$, we define $u_{\mathcal{T}} = \arg\min_{f\in\mathcal{F}}\mathcal{L}(f;\mathcal{T})$ as the neural network whose loss is at global minimum. For simplicity, we assume that u, $u_{\mathcal{F}}$, and $u_{\mathcal{T}}$ are well defined and unique. Finding $u_{\mathcal{T}}$ by minimizing the loss is often computationally intractable [10], and our optimizer returns an approximate solution $\tilde{u}_{\mathcal{T}}$.

We can then decompose the total error $\mathcal{E}$ as [11]

$$\mathcal{E} := \|\tilde{u}_{\mathcal{T}} - u\| \le \underbrace{\|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\|}_{\mathcal{E}_{\text{opt}}} + \underbrace{\|u_{\mathcal{T}} - u_{\mathcal{F}}\|}_{\mathcal{E}_{\text{gen}}} + \underbrace{\|u_{\mathcal{F}} - u\|}_{\mathcal{E}_{\text{app}}}.$$

The approximation error $\mathcal{E}_{\text{app}}$ measures how closely $u_{\mathcal{F}}$ can approximate u. The generalization error $\mathcal{E}_{\text{gen}}$ is determined by the number/locations of residual points in $\mathcal{T}$ and the capacity of the family $\mathcal{F}$. Neural networks of larger size have smaller approximation errors but could lead to higher generalization errors, which is called bias-variance tradeoff. Overfitting occurs when the generalization error dominates. In addition, the optimization error $\mathcal{E}_{\text{opt}}$ stems from the loss function complexity and the optimization setup, such as learning rate and number of iterations. However, currently there is no error estimation for PINNs yet, and even quantifying the three errors for supervised learning is still an open research problem [36, 35, 25].

2.5. Comparison between PINNs and FEM. To further explain the ideas of PINNs and to help those with knowledge of FEM to understand PINNs more easily, we make a comparison between PINNs and FEM point by point (Table 2):

Table 2 *Comparison between PINN and FEM.*

	PINN	FEM
Basis function	Neural network (nonlinear)	Piecewise polynomial (linear)
Parameters	Weights and biases	Point values
Training points	Scattered points (mesh-free)	Mesh points
PDE embedding	Loss function	Algebraic system
Parameter solver	Gradient-based optimizer	Linear solver
Errors	$\mathcal{E}_{\text{app}}$, $\mathcal{E}_{\text{gen}}$, and $\mathcal{E}_{\text{opt}}$ (subsection 2.4)	Approximation/quadrature errors
Error bounds	Not available yet	Partially available [14, 26]

- In FEM we approximate the solution u by a piecewise polynomial with point values to be determined, while in PINNs we construct a neural network as the surrogate model parameterized by weights and biases.
- FEM typically requires a mesh generation, while PINN is totally mesh-free, and we can use either a grid or random points.
- FEM converts a PDE to an algebraic system, using the stiffness matrix and mass matrix, while PINN embeds the PDE and boundary conditions into the loss function.
- In the last step, the algebraic system in FEM is solved exactly by a linear solver, but the network in PINN is learned by a gradient-based optimizer.

At a more fundamental level, PINNs provide a nonlinear approximation to the function and its derivatives, whereas FEM represents a linear approximation.

2.6. PINNs for Solving Integro-differential Equations. When solving integro-differential equations (IDEs), we still employ the automatic differentiation technique to analytically derive the integer-order derivatives, while we approximate integral operators numerically using classical methods (Figure 4) [42], such as Gaussian quadrature. Therefore, we introduce a fourth error component, the discretization error $\mathcal{E}_{\text{dis}}$, due to the approximation of the integral by Gaussian quadrature.

For example, when solving

$$\frac{dy}{dx} + y(x) = \int_0^x e^{t-x} y(t)dt,$$

we first use Gaussian quadrature of degree n to approximate the integral

$$\int_0^x e^{t-x} y(t)dt \approx \sum_{i=1}^{n} w_i e^{t_i(x)-x} y(t_i(x)),$$

and then we use a PINN to solve the following PDE instead of the original equation:

$$\frac{dy}{dx} + y(x) \approx \sum_{i=1}^{n} w_i e^{t_i(x)-x} y(t_i(x)).$$

PINNs can also be easily extended to solve FDEs [42] and SDEs [57, 55, 41, 56], but we do not discuss such cases here due to page limit constraints.

2.7. PINNs for Solving Inverse Problems. In inverse problems, there are some unknown parameters $\boldsymbol{\lambda}$ in (2.1), but we have some extra information on some points $\mathcal{T}_i \subset \Omega$ besides the differential equation and boundary conditions:

$$\mathcal{I}(u, \mathbf{x}) = 0 \quad \text{for} \quad \mathbf{x} \in \mathcal{T}_i.$$

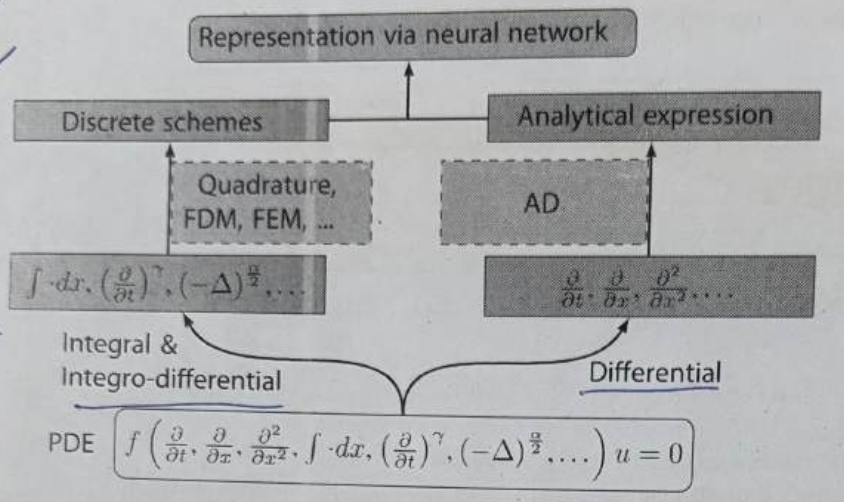


Fig. 4 *Schematic illustrating the modification of the PINN algorithm for solving IDEs. We employ the AD technique to analytically derive the integer-order derivatives, and we approximate integral operators numerically using standard methods. (The figure is reproduced from [42].)*

From an implementation point of view, PINNs solve inverse problems as easily as forward problems [47, 42]. The only difference between solving forward and inverse problems is that we add an extra loss term to (2.2):

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_b) + w_i \mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i),$$

where

$$\mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i) = \frac{1}{|\mathcal{T}_i|} \sum_{\mathbf{x} \in \mathcal{T}_i} \|\mathcal{I}(\hat{u}, \mathbf{x})\|_2^2.$$

We then optimize $\boldsymbol{\theta}$ and $\boldsymbol{\lambda}$ together, and our solution is $\boldsymbol{\theta}^*, \boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\theta}, \boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T})$.

2.8. Residual-Based Adaptive Refinement (RAR). As we discussed in subsection 2.3, the residual points $\mathcal{T}$ are usually randomly distributed in the domain. This works well for most cases, but it may not be efficient for certain PDEs that exhibit solutions with steep gradients. Take the Burgers equation as an example; intuitively we should put more points near the sharp front to capture the discontinuity well. However, it is challenging, in general, to design a good distribution of residual points for problems whose solution is unknown. To overcome this challenge, we propose a residual-based adaptive refinement (RAR) method to improve the distribution of residual points during the training process (Procedure 2.2), conceptually similar to FEM refinement methods [2]. The idea of RAR is that we will add more residual points in the locations where the PDE residual $\left\|f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}, \ldots; \boldsymbol{\lambda}\right)\right\|$ is large, and we repeatedly add points until the mean residual

$$\mathcal{E}_r = \frac{1}{V} \int_\Omega \left\| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}, \ldots; \boldsymbol{\lambda}\right) \right\| d\mathbf{x} \tag{2.3}$$

is smaller than a threshold $\mathcal{E}_0$, where V is the volume of Ω.

Procedure 2.2 RAR for improving the distribution of residual points for training.

Step 1 Select the initial residual points $\mathcal{T}$, and train the neural network for a limited number of iterations.

Step 2 Estimate the mean PDE residual $\mathcal{E}_r$ in (2.3) by Monte Carlo integration, i.e., by the average of values at a set of randomly sampled dense locations $\mathcal{S} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{|\mathcal{S}|}\}$:

$$\mathcal{E}_r \approx \frac{1}{|\mathcal{S}|} \sum_{\mathbf{x} \in \mathcal{S}} \left\| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \dots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \dots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \dots; \boldsymbol{\lambda}\right) \right\|.$$

Step 3 Stop if $\mathcal{E}_r < \mathcal{E}_0$. Otherwise, add m new points with the largest residuals in $\mathcal{S}$ to $\mathcal{T}$, retrain the network, and go to Step 2.

3. DeepXDE Usage and Customization.

In this section, we introduce the usage of DeepXDE and how to customize DeepXDE to meet new problem requirements.

3.1. Usage.

Compared to traditional numerical methods, the code written with DeepXDE is much shorter and more comprehensive, resembling closely the mathematical formulation. Solving differential equations in DeepXDE is no more than specifying the problem using the built-in modules, including computational domain (geometry and time), PDE equations, boundary/initial conditions, constraints, training data, neural network architecture, and training hyperparameters. The workflow is shown in Procedure 3.1 and Figure 5.

Procedure 3.1 Usage of DeepXDE for solving differential equations.

Step 1 Specify the computational domain using the **`geometry`** module.

Step 2 Specify the PDE using the grammar of **`TensorFlow`**.

Step 3 Specify the boundary and initial conditions.

Step 4 Combine the geometry, PDE and boundary/initial conditions together into **`data.PDE`** or **`data.TimePDE`** for time-independent problems or for time-dependent problems, respectively. To specify training data, we can either set the specific point locations, or only set the number of points and then DeepXDE will sample the required number of points on a grid or randomly.

Step 5 Construct a neural network using the **`maps`** module.

Step 6 Define a **`Model`** by combining the PDE problem in Step 4 and the neural net in Step 5.

Step 7 Call **`Model.compile`** to set the optimization hyperparameters, such as optimizer and learning rate. The weights in (2.2) can be set here by **`loss_weights`**.

Step 8 Call **`Model.train`** to train the network from random initialization or a pretrained model using the argument **`model_restore_path`**. It is extremely flexible to monitor and modify the training behavior using **`callbacks`**.

Step 9 Call **`Model.predict`** to predict the PDE solution at different locations.

In DeepXDE, the built-in primitive geometries include **`interval`**, **`triangle`**, **`rectangle`**, **`polygon`**, **`disk`**, **`cuboid`**, and **`sphere`**. Other geometries can be constructed from these primitive geometries using three boolean operations: **`union (|)`**, **`difference (-)`**, and **`intersection (&)`**. This technique is called constructive solid geometry (CSG); see Figure 6 for examples. CSG supports both 2D and 3D geometries.

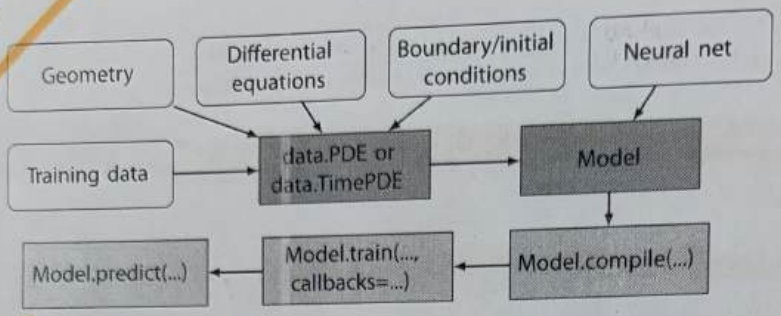


Fig. 5 *Flowchart of DeepXDE corresponding to Procedure 3.1. The white boxes define the PDE problem and the training hyperparameters. The blue boxes combine the PDE problem and training hyperparameters in the white boxes. The orange boxes are the three steps (from right to left) to solve the PDE.*

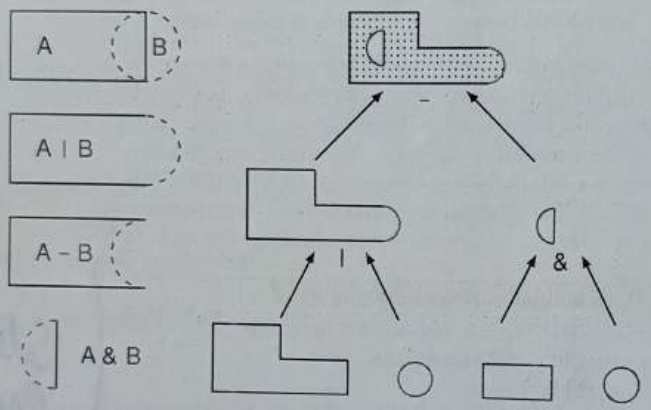


Fig. 6 *Examples of constructive solid geometry (CSG) in 2D. (left) A and B represent the rectangle and circle, respectively. The union $A|B$, difference $A - B$, and intersection $A \& B$ are constructed from A and B. (right) A complex geometry (top) is constructed from a polygon, a rectangle, and two circles (bottom) through the union, difference, and intersection operations. This capability is included in the module* `geometry` *of DeepXDE.*

DeepXDE supports four standard boundary conditions, including Dirichlet (`DirichletBC`), Neumann (`NeumannBC`), Robin (`RobinBC`), and periodic (`PeriodicBC`), and a more general boundary condition can be defined using `OperatorBC`. The initial condition can be defined using `IC`. There are two types of neural networks available in DeepXDE: feed-forward neural network (`maps.FNN`) and residual neural network (`maps.ResNet`). It is also convenient to choose different training hyperparameters, such as loss types, metrics, optimizers, learning rate schedules, initializations, and regularizations.

In addition to solving differential equations, DeepXDE can also be used to approximate functions from multifidelity data [40] and learn nonlinear operators [34].

3.2. Customizability. All the components of DeepXDE are loosely coupled, and thus DeepXDE is well structured and highly configurable. In this subsection, we discuss how to customize DeepXDE to address new problem requirements, e.g., new geometry or network architecture.

3.2.1. Geometry. As we introduced above, DeepXDE has already supported 7 basic geometries and the CSG technique. However, it is still possible that the user needs a new geometry, which cannot be constructed in DeepXDE. In this situation, a new geometry can be defined as shown in Procedure 3.2. Currently DeepXDE does not accurately support descriptions of complex curvilinear boundaries; however, a future extension could be incorporation of the nonuniform rational basis spline (NURBS) [22] for such representations.

Procedure 3.2 Customization of the new geometry module **MyGeometry**. The class methods should only be implemented as needed.

```
class MyGeometry(Geometry):
    def inside(self, x):
        """Check if x is inside the geometry."""
    def on_boundary(self, x):
        """Check if x is on the geometry boundary."""
    def boundary_normal(self, x):
        """Compute the unit normal at x for Neumann or Robin boundary conditions."""
    def periodic_point(self, x, component):
        """Compute the periodic image of x for periodic boundary condition."""
    def uniform_points(self, n, boundary=True):
        """Compute the equispaced point locations in the geometry."""
    def random_points(self, n, random="pseudo"):
        """Compute the random point locations in the geometry."""
    def uniform_boundary_points(self, n):
        """Compute the equispaced point locations on the boundary."""
    def random_boundary_points(self, n, random="pseudo"):
        """Compute the random point locations on the boundary."""
```

Extending the base geometry class

3.2.2. Neural Networks. DeepXDE currently supports two neural networks: feed-forward neural network (**maps.FNN**) and residual neural network (**maps.ResNet**). A new network can be added, as shown in Procedure 3.3.

Procedure 3.3 Customization of the neural network **MyNet**.

```
class MyNet(Map):
    @property
    def inputs(self):
        """Return the net inputs."""
    @property
    def outputs(self):
        """Return the net outputs."""
    @property
    def targets(self):
        """Return the targets of the net outputs."""
    def build(self):
        """Construct the network."""
```

Extending the base NN class

3.2.3. Callbacks. It is usually a good strategy to monitor the training process of the neural network, and then make modifications in real time, e.g., change the learning rate. In DeepXDE, this can be implemented by adding a callback function, and here we only list a few commonly used ones already implemented in DeepXDE:

- **ModelCheckpoint**, which saves the model after certain epochs or when a better model is found.
- **OperatorPredictor**, which calculates the values of the operator applied to the outputs.

Standard TensorFlow/Keras/PyTorch/JAX/MXNet/Flux/ML type callbacks

- **FirstDerivative**, which calculates the first derivative of the outputs with respect to the inputs. This is a special case of **OperatorPredictor** with the operator being the first derivative.
- **MovieDumper**, which dumps the movie of the function during the training progress, and/or the movie of the spectrum of its Fourier transform.

It is very convenient to add other **callback** functions, which will be called at different stages of the training process; see Procedure 3.4.

Procedure 3.4 Customization of the callback **MyCallback**. Here, we only show how to add functions to be called at the beginning/end of every epoch. Similarly, we can call functions at the other training stages, such as at the beginning of training.

```
class MyCallback(Callback):
    def on_epoch_begin(self):
        """Called at the beginning of every epoch."""
    def on_epoch_end(self):
        """Called at the end of every epoch."""
```

4. Demonstration Examples. In this section, we use PINNs and DeepXDE to solve different problems. In all examples, we use the tanh as the activation function, and the other hyperparameters are listed in Table 3. The weights w_f, w_b, and w_i in the loss function are set as 1. The codes of all examples are published in GitHub.

Table 3 *Hyperparameters used for the following 5 examples. "Adam, L-BFGS" represents that we first use Adam for a certain number of iterations, and then switch to L-BFGS. The optimizer L-BFGS does not require learning rate, and the neural network (NN) is trained until convergence, so the number of iterations is also ignored for L-BFGS.*

Example	NN depth	NN width	Optimizer	Learning rate	# Iterations
1	4	50	Adam, L-BFGS	0.001	50000
2	3	20	Adam, L-BFGS	0.001	10000
3	3	40	Adam	0.001	60000
4	3	20	Adam	0.001	80000
5	4	20	L-BFGS	-	-

4.1. Poisson Equation over an L-shaped Domain. Consider the following 2D Poisson equation over an L-shaped domain $\Omega = [-1, 1]^2 \setminus [0, 1]^2$:

$$-\Delta u(x, y) = 1, \quad (x, y) \in \Omega,$$
$$u(x, y) = 0, \quad (x, y) \in \partial\Omega.$$

We choose 1200 and 120 random points drawn from a uniform distribution as $\mathcal{T}_f$ and $\mathcal{T}_b$, respectively. The PINN solution is given in Figure 7B. For comparison, we also present the numerical solution obtained using the spectral element method (SEM) [27] (Figure 7A). The result of the absolute error is shown in Figure 7C.

4.2. RAR for the Burgers Equation. We first consider the 1D Burgers equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad x \in [-1, 1], \ t \in [0, 1],$$
$$u(x, 0) = -\sin(\pi x), \quad u(-1, t) = u(1, t) = 0.$$

Let $\nu = 0.01/\pi$. Initially, we randomly select 2500 points (spatio-temporal domain) as the residual points, and then 40 more residual points are added adaptively via

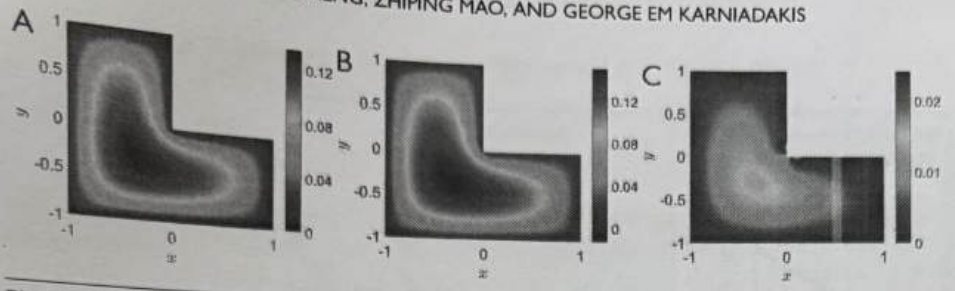


Fig. 7 *Example in subsection 4.1. Comparison of the PINN solution with the solution obtained by using the spectral element method (SEM). (A) The SEM solution u_{SEM}, (B) the PINN solution u_{NN}, (C) the absolute error $|u_{SEM} - u_{NN}|$.*

RAR developed in subsection 2.8 with $m = 1$, $|\mathcal{S}| = 100000$, and $\mathcal{E}_0 = 0.005$. The training procedure after adding new points is shown in Table 3. We compare the PINN solution with RAR and the PINN solution based on 2540 randomly selected training data (panels A and B of Figure 8), and demonstrate that PINN with RAR can capture the discontinuity much better. For a comparison, the finite difference solutions using the central difference scheme for space discretization and the forward Euler scheme for time discretization for the Burgers equation in the conservative form are also shown in Figure 8A. Here, we also present two examples for the residual points added via the RAR method. As shown in panels C and D of Figure 8, the added points (green crosses) are quite close to the sharp interface, which indicates the effectiveness of RAR.

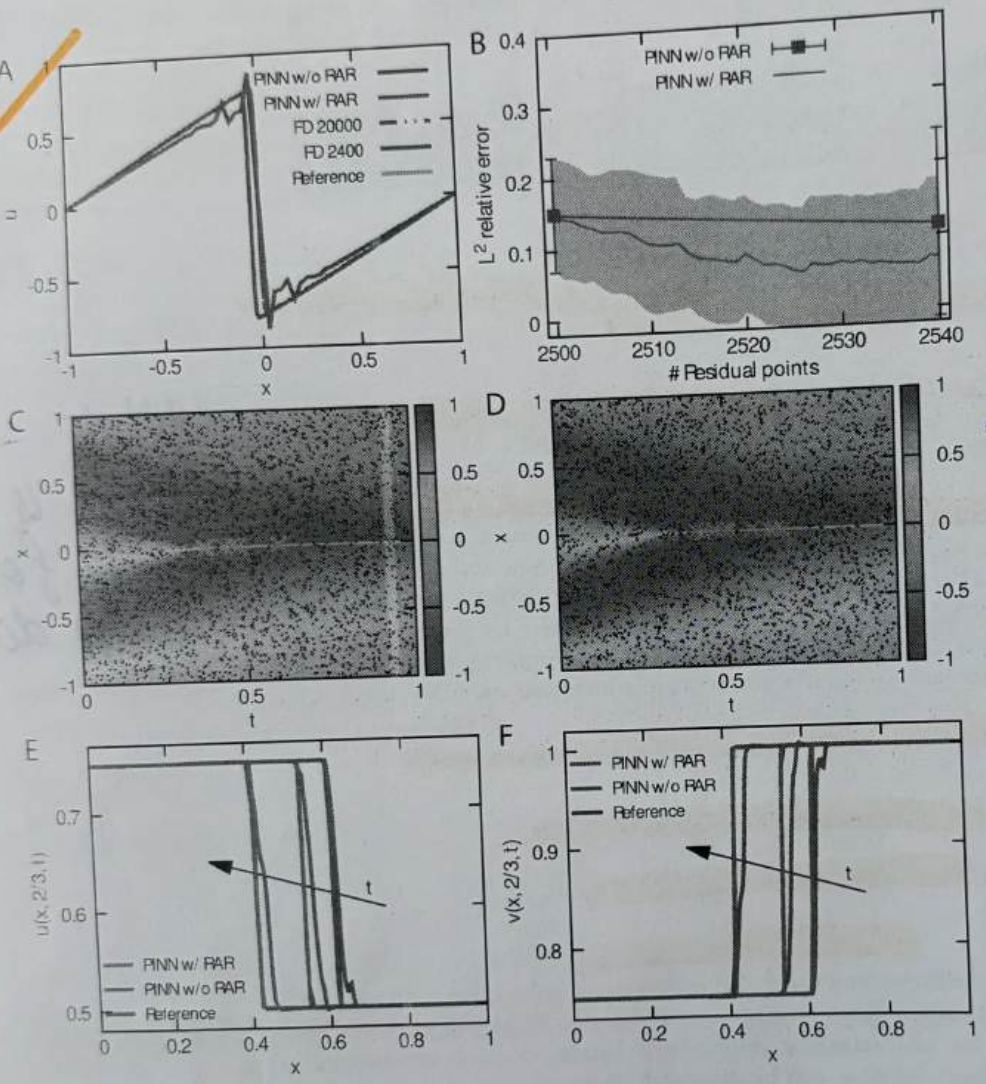
We further solve the following 2D Burgers equation using the RAR:

$$\partial_t u + u\partial_x u + v\partial_y u = \frac{1}{\mathrm{Re}}(\partial_x^2 u + \partial_y^2 u),$$
$$\partial_t v + u\partial_x v + v\partial_y v = \frac{1}{\mathrm{Re}}(\partial_x^2 v + \partial_y^2 v),$$
$$x, y \in [0,1], \text{ and } t \in [0,1],$$

where u and v are the velocities along the x and y directions, respectively. In addition, Re is a nondimensional number (Reynolds number) defined as $\mathrm{Re} = UL/\nu$, in which U and L are, respectively, the characteristic velocity and length, and ν is the kinematic viscosity of fluid. The exact solution can be obtained [3] as

$$u(x,y,t) = \frac{3}{4} - \frac{1}{4\,[1 + \exp((-4x + 4y - t)\mathrm{Re}/32)]},$$
$$v(x,y,t) = \frac{3}{4} + \frac{1}{4\,[1 + \exp((-4x + 4y - t)\mathrm{Re}/32)]},$$

using the Dirichlet boundary conditions on all boundaries. In the present study, Re is set to be 5000, which is quite challenging due to the fact that the high Reynolds number leads to a steep gradient in the solution. Initially, 200 residual points are randomly sampled in the spatio-temporal domain, and 5000 and 1000 random points are used for each initial and boundary condition, respectively. We only add 10 extra residual points using RAR, and the total number of the residual points is 210 after convergence. For comparison, we also test the case without RAR using 210 randomly sampled residual points. The results are displayed in panels E and F of Figure 8, demonstrating the effectiveness of RAR.



look in PDF

Fig. 8 *Example in subsection 4.2. Comparisons of the PINN solutions of 1D (A, B, C, and D) and 2D (E and F) Burgers equations with and without RAR. (A) The cyan, green, and red lines represent the reference solution of u from [47], the PINN solution without RAR, and the PINN solution with RAR at $t = 0.9$, respectively. For the finite difference (FD) method, $200 \times 100 = 20000$ spatial-temporal grid points are used to achieve a good solution (blue line). If only $60 \times 40 = 2400$ points are used, the FD solution has large oscillations around the discontinuity (brown line). (B) L^2 relative error versus the number of residual points. The red solid line and shaded region correspond to the mean and one-standard-deviation band for the L^2 relative error of PINN with RAR, respectively. The blue dashed line is the mean and one-standard-deviation for the error of PINN using 2540 random residual points. The mean and standard deviation are obtained from 10 runs with random initial residual points. (C) and (D) Two representative examples for the residual points added via RAR. Black dots: initial residual points; green cross: added residual points; 6 and 11 residual points are added in C and D, respectively. (E) and (F) Comparison of the velocity profiles at $y = \frac{2}{3}$ from the PINNs with and without RAR for the 2D Burgers equation. The profiles at three different times ($t = 0.2$, 0.5, and 1) are presented with t increasing along the direction of the arrow.*

4.3. Inverse Problem for the Lorenz System. Consider the parameter identification problem of the Lorenz system

$$\frac{dx}{dt} = \rho(y - x), \quad \frac{dy}{dt} = x(\sigma - z) - y, \quad \frac{dz}{dt} = xy - \beta z,$$

with initial condition $(x(0), y(0), z(0)) = (-8, 7, 27)$, where ρ, σ, and β are the three parameters to be identified from the observations at certain times. The observations are produced by solving the above system to $t = 3$ using Runge–Kutta $(4, 5)$ with the underlying true parameters $(\rho, \sigma, \beta) = (10, 15, 8/3)$. We choose 400 uniformly distributed random points and 25 equispaced points as the residual points $\mathcal{T}_f$ and $\mathcal{T}_i$, respectively. The evolution trajectories of ρ, σ, and β are presented in Figure 9A, with the final identified values of $(\rho, \sigma, \beta) = (10.002, 14.999, 2.668)$.

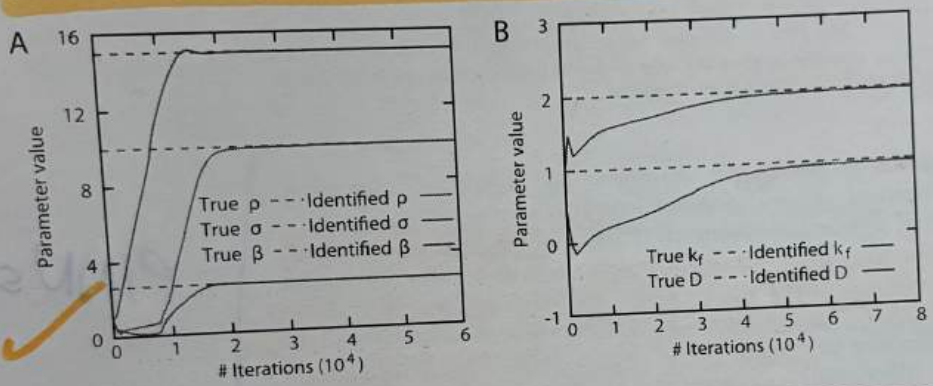


Fig. 9 *Examples in subsections 4.3 and 4.4. The identified values of* (A) *the Lorenz system and* (B) *diffusion-reaction system converge to the true values during the training process. The parameter values are scaled for plotting.*

4.4. Inverse Problem for Diffusion-Reaction Systems. A diffusion-reaction system in porous media for the solute concentrations C_A, C_B, and C_C $(A + 2B \to C)$ is described by

$$\frac{\partial C_A}{\partial t} = D\frac{\partial^2 C_A}{\partial x^2} - k_f C_A C_B^2, \quad \frac{\partial C_B}{\partial t} = D\frac{\partial^2 C_B}{\partial x^2} - 2k_f C_A C_B^2, \quad x \in [0, 1], t \in [0, 10],$$
$$C_A(x, 0) = C_B(x, 0) = e^{-20x}, \quad C_A(0, t) = C_B(0, t) = 1, \quad C_A(1, t) = C_B(1, t) = 0,$$

where $D = 2 \times 10^{-3}$ is the effective diffusion coefficient, and $k_f = 0.1$ is the effective reaction rate. Because D and k_f depend on the pore structure and are difficult to measure directly, we estimate D and k_f based on 40000 observations of the concentrations C_A and C_B in the spatio-temporal domain. The identified D (1.98×10^{-3}) and k_f (0.0971), displayed in Figure 9B, agree well with their true values.

4.5. Volterra IDE. Here, we consider the first-order IDE of Volterra type in the domain $[0, 5]$:

$$\frac{dy}{dx} + y(x) = \int_0^x e^{t-x} y(t)dt, \quad y(0) = 1,$$

with the exact solution $y(x) = e^{-x}\cosh x$. We solve this IDE using the method in subsection 2.6 and approximate the integral using Gaussian-Legendre quadrature of degree 20. The L^2 relative error is 2×10^{-3}, and the solution is shown in Figure 10.

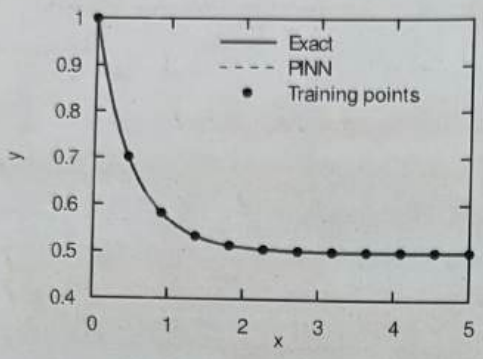


Fig. 10 *Example in subsection 4.5. The PINN algorithm for solving Volterra IDE. The solid blue line is the exact solution, and the dashed red line is the numerical solution from PINN. Twelve equispaced residual points (black dots) are used.*

5. Concluding Remarks. In this paper, we present the algorithm, approximation theory, and error analysis of physics-informed neural networks (PINNs) for solving different types of partial differential equations (PDEs). Compared to traditional numerical methods, PINNs employ automatic differentiation to handle differential operators, and thus they are mesh-free. Unlike numerical differentiation, automatic differentiation does not differentiate the data, and hence it can tolerate noisy data for training. We also discuss how to extend PINNs to solve other types of differential equations, such as integro-differential equations, and also how to solve inverse problems. In addition, we propose a residual-based adaptive refinement (RAR) method to improve the distribution of residual points during the training process, and thus increase the training efficiency.

To benefit both the educational and the computational science communities, we have developed the Python library DeepXDE, an implementation of PINNs. By introducing the usage of DeepXDE, we show that DeepXDE enables user codes to be compact and follow closely the mathematical formulation. We also demonstrate how to customize DeepXDE to meet new problem requirements. Our numerical examples for forward and inverse problems verify the effectiveness of PINNs and the capability of DeepXDE. Scientific machine learning is emerging as a new and potentially powerful alternative to classical scientific computing, so we hope that libraries such as DeepXDE will accelerate this development and make it accessible not only to students but also to other researchers who may find the need to adopt PINN-like methods in their research, which can be very effective, especially for inverse problems.

Despite the aforementioned advantages, PINNs still have some limitations. For forward problems, PINNs are currently slower than finite elements, but this can be alleviated via offline training [58, 53]. For long time integration, one can also use time-parallel methods to simultaneously compute on multiple GPUs for shorter time domains. Another limitation is the search for effective neural network architectures, which is currently done empirically by users; however, emerging meta-learning techniques can be used to automate this search; see [59, 17]. Moreover, while here we enforce the strong form of PDEs, which is easily implemented using automatic differentiation, alternative weak/variational forms may also be effective, although they require the use of quadrature grids. Many other extensions for multiphysics and

PINNs

Deep XDE

Limitations

1. Slower than FEMs
2. Requires humans to search for NNs & hyperparams
3. Weak/Variational form of PDEs is tough to implement

→ Also proposes solutions

multiscale problems are possible across different scientific disciplines by creatively designing the loss function and introducing suitable solution spaces. For instance, in the five examples we present here, we only assume data on scattered points; however, in geophysics or biomedicine we may have mixed data in the form of images and point measurements. In this case, we can design a composite neural network consisting of one convolutional neural network and one PINN sharing the same set of parameters, and minimize the total loss which could be a weighted summation of multiple losses from each neural network.

Combining CNNs & PINNs

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., *TensorFlow: A system for large-scale machine learning*, in 12th USENIX Symposium on Operating Systems Design and Implementation, 2016, pp. 265–283. (Cited on p. 212)

[2] M. Ainsworth and J. T. Oden, *A Posteriori Error Estimation in Finite Element Analysis*, John Wiley & Sons, 2011. (Cited on p. 217)

[3] A. Baeza and P. Mulet, *Adaptive mesh refinement techniques for high-order shock capturing schemes for multi-dimensional hydrodynamic simulations*, Internat. J. Numer. Methods Fluids, 52 (2006), pp. 455–471. (Cited on p. 222)

[4] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild et al., *Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence*, Tech. report, U.S. DOE Office of Science, Washington, DC, 2019. (Cited on p. 209)

[5] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: A survey*, J. Mach. Learn. Res., 18 (2017), pp. 5595–5637. (Cited on pp. 210, 211)

[6] C. Beck, W. E, and A. Jentzen, *Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations*, J. Nonlinear Sci., 29 (2019), pp. 1563–1619. (Cited on p. 209)

[7] J. Berg and K. Nyström, *A unified deep artificial neural network approach to partial differential equations in complex geometries*, Neurocomput., 317 (2018), pp. 28–41. (Cited on p. 209)

[8] M. Betancourt, *A Geometric Theory of Higher-Order Automatic Differentiation*, preprint, https://arxiv.org/abs/1812.11592, 2018. (Cited on p. 211)

[9] J. Bettencourt, M. J. Johnson, and D. Duvenaud, *Taylor-mode automatic differentiation for higher-order derivatives in JAX*, in NeurIPS 2019 Workshop on Program Transformations, 2019, https://openreview.net/forum?id=SkxEF3FNPH. (Cited on p. 211)

[10] A. Blum and R. L. Rivest, *Training a 3-node neural network is NP-complete*, in Advances in Neural Information Processing Systems, Morgan Kaufmann, 1989, pp. 494–501. (Cited on pp. 213, 215)

[11] L. Bottou and O. Bousquet, *The tradeoffs of large scale learning*, in Advances in Neural Information Processing Systems, Morgan Kaufmann, 2008, pp. 161–168. (Cited on p. 215)

[12] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, *A limited memory algorithm for bound constrained optimization*, SIAM J. Sci. Comput., 16 (1995), pp. 1190–1208, https://doi.org/10.1137/0916069. (Cited on p. 213)

[13] Y. Chen, L. Lu, G. E. Karniadakis, and L. D. Negro, *Physics-Informed Neural Networks for Inverse Problems in Nano-optics and Metamaterials*, preprint, https://arxiv.org/abs/1912.01085, 2019. (Cited on p. 209)

[14] P. G. Ciarlet, *The Finite Element Method for Elliptic Problems*, Classics Appl. Math. 40, SIAM, 2002, https://doi.org/10.1137/1.9780898719208. (Cited on p. 216)

[15] M. Dissanayake and N. Phan-Thien, *Neural-network-based approximations for solving partial differential equations*, Comm. Numer. Methods Engrg., 10 (1994), pp. 195–201. (Cited on p. 209)

[16] W. E and B. Yu, *The deep Ritz method: A deep learning-based numerical algorithm for solving variational problems*, Commun. Math. Stat., 6 (2018), pp. 1–12. (Cited on p. 209)

[17] C. Finn, P. Abbeel, and S. Levine, *Model-agnostic meta-learning for fast adaptation of deep networks*, in Proceedings of the 34th International Conference on Machine Learning, PMLR, 2017, pp. 1126–1135. (Cited on p. 225)