DeepXDE: A deep learning library for solving differential equations

Lu Lu
Division of Applied Mathematics
Brown University
Providence, RI 02906
`lu_lu_1@brown.edu`

Xuhui Meng
Division of Applied Mathematics
Brown University
Providence, RI 02906
`xuhui_meng@brown.edu`

Zhiping Mao
Division of Applied Mathematics
Brown University
Providence, RI 02906
`zhiping_mao@brown.edu`

George Em Karniadakis
Division of Applied Mathematics
Brown University
Providence, RI 02906
`george_karniadakis@brown.edu`

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, PINNs solve inverse problems as easily as forward problems. We also present a Python library for PINNs: DeepXDE, which 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 two examples.

1 Introduction

Despite the remarkable success of deep learning in diverse applications, deep learning has not yet been widely used in the field of scientific computing. However, more recently, solving partial differential equations (PDEs) via deep learning has emerged as a potentially new sub-field under the name of Scientific Machine Learning [1]. To solve 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 the traditional mesh-based methods, such as the finite difference method and the finite element method, deep learning could be a mesh-free approach by taking advantage of the automatic differentiation [17], and could break the curse of dimensionality [16, 6]. Among these approaches, some can only be applied to particular types of problems, such as image-like input domain [9, 12, 24] or parabolic PDEs [2, 7]. Some researchers adopt the variational form of PDEs and minimize the corresponding energy functional [5, 8]. However, not all PDEs can be derived from a known functional, and thus Galerkin type projections have also been considered [13]. Alternatively, one could use the PDE in strong form directly [4, 20, 10, 11, 3, 19, 17]; 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 [17] coining the term physics-informed neural networks (PINNs). An attractive feature of PINNs is that it can be

Second Workshop on Machine Learning and the Physical Sciences (NeurIPS 2019), Vancouver, Canada.

used to solve inverse problems with minimum change of the code for forward problems [17, 18]. In addition, PINNs have been further extended to solve integro-differential equations (IDEs), fractional differential equations (FDEs) [15], and stochastic differential equations (SDEs) [23, 21, 14, 22].

In this paper, we present the PINN algorithm and a Python library DeepXDE (`https://github.com/lululxvi/deepxde`), which can be used to solve multi-physics problems and supports complex-geometry domains based on the technique of constructive solid geometry, hence avoiding tedious and time-consuming computational geometry tasks. By using DeepXDE, time-dependent PDEs can be solved as easily as steady states by only defining the initial conditions. DeepXDE is designed to make user codes stay compact and manageable, resembling closely the mathematical formulation.

The paper is organized as follows. In Section 2, we present the algorithm of PINNs. In Section 3, we introduce the usage of our library, DeepXDE, and its customizability. In Section 4, we demonstrate the capability of PINNs and DeepXDE. Finally, we conclude the paper in Section 5.

2 Physics-informed neural networks

In this section, we first provide a overview of PINNs for solving forward and inverse PDEs.

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 $\Omega \subset \mathbb{R}^d$:

$$f\left(\mathbf{x}; \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right) = 0, \quad \mathbf{x} \in \Omega, \tag{1}$$

with suitable boundary conditions (BCs) $\mathcal{B}(u, \mathbf{x}) = 0$ on $\partial\Omega$. For time-dependent problems, we consider time t as a special component of $\mathbf{x}$, and Ω contains the temporal domain.

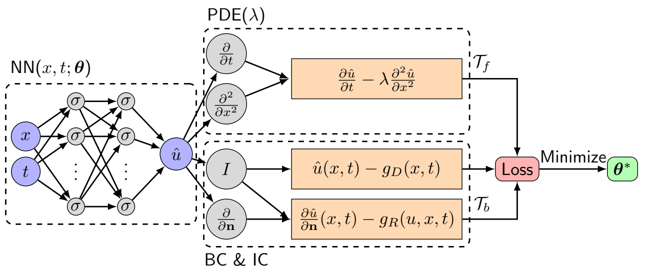
PDE(λ)
NN($x, t; \boldsymbol{\theta}$)
x
t
σ
$\hat{u}$
$\frac{\partial}{\partial t}$
$\frac{\partial^2}{\partial x^2}$
I
$\frac{\partial}{\partial \mathbf{n}}$
$\frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2}$
$\hat{u}(x,t) - g_D(x,t)$
$\frac{\partial \hat{u}}{\partial \mathbf{n}}(x,t) - g_R(u,x,t)$
$\mathcal{T}_f$
$\mathcal{T}_b$
Loss
Minimize
$\boldsymbol{\theta}^*$
BC & IC

Figure 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 BCs $u(x,t) = g_D(x,t)$ 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 Dirichlet BC on the spatio-temporal domain. $\mathcal{T}_f$ and $\mathcal{T}_b$ denote the two sets of residual points for the equation and BC/IC.

Procedure 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 by summing the weighted L^2 norm of both the PDE and BC residuals.

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

The algorithm of PINN [11, 17] is shown in Procedure 1, and visually in the schematic of Fig. 1 solving a diffusion equation. 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 network $\hat{u}$. One advantage of choosing neural networks as the surrogate of u is that we can take the derivatives of $\hat{u}$ with respect to $\mathbf{x}$ by the automatic differentiation.

In the next step, we need to restrict $\hat{u}$ to satisfy the PDE and BCs. We only restrict $\hat{u}$ on some scattered points, i.e., the training data $\mathcal{T} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{|\mathcal{T}|}\}$ of size $|\mathcal{T}|$. $\mathcal{T}$ is comprised of 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 $\mathcal{T}_f$ and $\mathcal{T}_b$ as the sets of "residual points". To measure the discrepancy between $\hat{u}$ and the constraints, we consider the loss defined as the weighted summation of the residuals for the equation and BCs:

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b), \tag{2}$$

where $\mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} \left\| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right) \right\|_2^2$, $\mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x} \in \mathcal{T}_b} \|\mathcal{B}(\hat{u}, \mathbf{x})\|_2^2$, and w_f and w_b are the weights. In the last step, the procedure of searching for a good $\boldsymbol{\theta}$ by minimizing the loss $\mathcal{L}(\boldsymbol{\theta}; \mathcal{T})$ using gradient-based optimizers is called "training".

PINNs for solving inverse problems In inverse problems, there are some unknown parameters $\boldsymbol{\lambda}$ in Eq. (1), but we have extra information on points $\mathcal{T}_i \subset \Omega$: $\mathcal{I}(u, \mathbf{x}) = 0$, for $\mathbf{x} \in \mathcal{T}_i$. PINNs solve inverse problems as easily as forward problems, and we only add an extra term to Eq. (2):

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_b) + w_i \mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i), \tag{3}$$

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: $\boldsymbol{\theta}^*, \boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\theta}, \boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T})$.

3 DeepXDE usage and customization

In this section, we introduce the usage of DeepXDE and how to customize DeepXDE.

Usage DeepXDE makes the code stay compact and nice, resembling closely the mathematical formulation. Solving differential equations in DeepXDE is no more than specifying the problem using the build-in modules, including computational domain (geometry and time), differential equations, ICs, BCs, constraints, training data, network architecture, and training hyperparameters. The workflow is shown in Procedure 2 and Fig. 2(left).

Procedure 2 Usage of DeepXDE for solving differential equations.

```
Step 1 Specify the computational domain using the geometry module.
Step 2 Specify the differential equations using the grammar of TensorFlow.
Step 3 Specify the boundary and initial conditions.
Step 4 Combine the geometry, PDE, and IC/BCs together into data.PDE or data.TimePDE for
       time-independent or 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 learn-
       ing rate. The weights in Eq. (2) can be set here by loss_weights.
Step 8 Call Model.train to train the network from random initialization or a pre-trained 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 Fig. 2 for examples.

DeepXDE supports four standard BCs, including `Dirichlet`, `Neumann`, `Robin`, and `Periodic`, and a more general BC can be defined using `OperatorBC`. The initial condition can be defined using `IC`. There are two networks available in DeepXDE: feed-forward neural network (`maps.FNN`) and residual

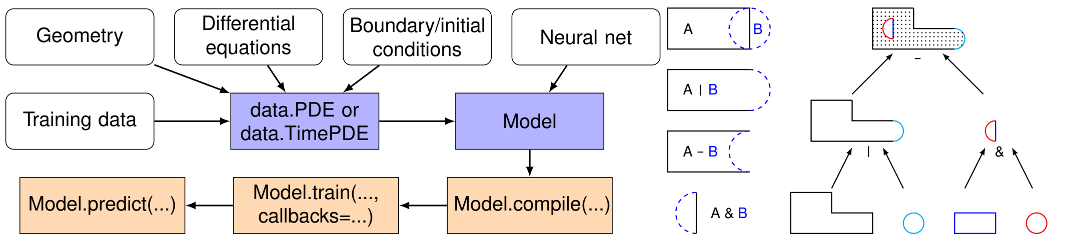

Figure 2: Usage of DeepXDE. (**left**) Flowchart of DeepXDE corresponding to Procedure 2. The white boxes define the PDE problem and the training hyperparameters, which are combined into the blue boxes. The orange boxes are the three steps to solve the PDE. (**middle**) CSG examples. A and B represent the rectangle and circle, respectively. $A|B$, $A - B$, and $A\&B$ are constructed from A and B. (**right**) A complex geometry (top) is constructed from a polygon, a rectangle and two circles (bottom). This capability is included in the module `geometry` of DeepXDE.

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.

It is usually a good strategy to monitor the training process of the neural network, and then make modifications in real time. In DeepXDE, this can be implemented by using a callback function, e.g., `ModelCheckpoint`, which saves the model after certain epochs or when a better model is found.

Customizability All the components of DeepXDE are loosely coupled, and thus DeepXDE is well-structured and highly configurable. Please refer to the documentation (`https://deepxde.readthedocs.io`) for the customization of new `geometry`, `neural network` and `callback`.

4 Demonstration examples

We use PINNs and DeepXDE to solve two inverse problems. We use the $\tanh$ as the activation function, Adam optimizer, and learning rate 0.001. For the Lorenz system we use a network of depth 3 and width 40, and for the diffusion-reaction system we use a network of depth 3 and width 20. The weights w_f, w_b and w_i in Eq. (3) are set as 1. The codes are published in GitHub.

Inverse problem for the Lorenz system Consider the Lorenz system $\frac{dx}{dt} = \rho(y - x)$, $\frac{dy}{dt} = x(\sigma - z) - y$, $\frac{dz}{dt} = xy - \beta z$, with the IC $(x(0), y(0), z(0)) = (-8, 7, 27)$, where ρ, σ and β are the parameters to be identified from the observations. The observations are produced by solving the above system to $t = 3$ with the true parameters $(\rho, \sigma, \beta) = (10, 15, 8/3)$. We choose 400 random points and 25 equispaced points as the residual points $\mathcal{T}_f$ and $\mathcal{T}_i$, respectively. The evolution trajectories are presented in Fig. 3A, with the final identified values of $(\rho, \sigma, \beta) = (10.002, 14.999, 2.668)$.

Inverse problem for the diffusion-reaction system 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 $\frac{\partial C_A}{\partial t} = D\frac{\partial^2 C_A}{\partial x^2} - k_f C_A C_B^2$, $\frac{\partial C_B}{\partial t} = D\frac{\partial^2 C_B}{\partial x^2} - 2k_f C_A C_B^2$ for $x \in [0, 1], t \in [0, 10]$ with IC $C_A(x, 0) = C_B(x, 0) = e^{-20x}$ and BCs $C_A(0, t) = C_B(0, t) = 1$, $C_A(1, t) = C_B(1, t) = 0$. We estimate the diffusion coefficient $D = 2 \times 10^{-3}$ and the reaction rate $k_f = 0.1$ 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) are displayed in Fig. 3B, which agree well with their true values.

5 Concluding Remarks

In this paper, we present the physics-informed neural networks (PINNs) for solving forward and inverse partial differential equations (PDEs). We 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. Our numerical examples verify the effectiveness of PINNs and the capability of DeepXDE.

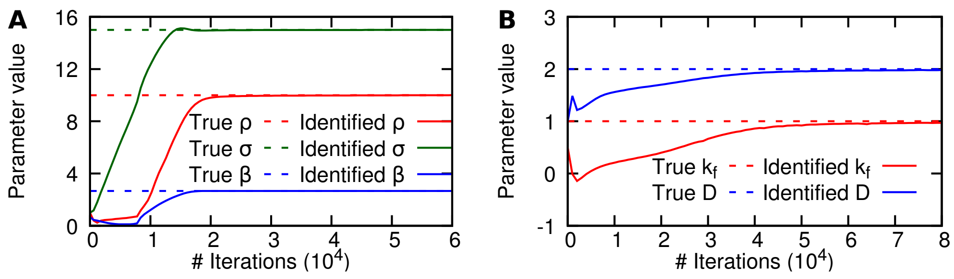

Figure 3: The identified values of (**A**) the Lorenz system and (**B**) diffusion-reaction system converge to the true values during the training process. The parameter values are scaled for plotting.

References

[1] 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. Technical report, US DOE Office of Science, Washington, DC (United States), 2019.

[2] 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. *Journal of Nonlinear Science*, pages 1–57, 2017.

[3] J. Berg and K. Nyström. A unified deep artificial neural network approach to partial differential equations in complex geometries. *Neurocomputing*, 317:28–41, 2018.

[4] M. Dissanayake and N. Phan-Thien. Neural-network-based approximations for solving partial differential equations. *Communications in Numerical Methods in Engineering*, 10(3):195–201, 1994.

[5] W. E and B. Yu. The deep Ritz method: A deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6(1):1–12, 2018.

[6] P. Grohs, F. Hornung, A. Jentzen, and P. Von Wurstemberger. A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of black-scholes partial differential equations. *arXiv preprint arXiv:1809.02362*, 2018.

[7] J. Han, A. Jentzen, and W. E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.

[8] J. He, L. Li, J. Xu, and C. Zheng. ReLU deep neural networks and linear finite elements. *arXiv preprint arXiv:1807.03973*, 2018.

[9] Y. Khoo, J. Lu, and L. Ying. Solving parametric PDE problems with artificial neural networks. *arXiv preprint arXiv:1707.03351*, 2017.

[10] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[11] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou. Neural-network methods for boundary value problems with irregular boundaries. *IEEE Transactions on Neural Networks*, 11(5):1041–1049, 2000.

[12] Z. Long, Y. Lu, X. Ma, and B. Dong. PDE-net: Learning PDEs from data. In *International Conference on Machine Learning*, pages 3214–3222, 2018.

[13] A. J. Meade Jr and A. A. Fernandez. The numerical solution of linear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling*, 19(12):1–25, 1994.

[14] M. A. Nabian and H. Meidani. A deep neural network surrogate for high-dimensional random partial differential equations. *arXiv preprint arXiv:1806.02957*, 2018.

[15] G. Pang, L. Lu, and G. E. Karniadakis. fPINNs: Fractional physics-informed neural networks. *SIAM Journal on Scientific Computing*, page to appear, 2019.

[16] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review. *International Journal of Automation and Computing*, 14(5):503–519, 2017.

[17] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[18] M. Raissi, A. Yazdani, and G. E. Karniadakis. Hidden fluid mechanics: A Navier-Stokes informed deep learning framework for assimilating flow visualization data. *arXiv preprint arXiv:1808.04327*, 2018.

[19] J. Sirignano and K. Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

[20] B. P. van Milligen, V. Tribaldos, and J. Jiménez. Neural network differential equation and plasma equilibrium solver. *Physical Review Letters*, 75(20):3594, 1995.

[21] L. Yang, D. Zhang, and G. E. Karniadakis. Physics-informed generative adversarial networks for stochastic differential equations. *arXiv preprint arXiv:1811.02033*, 2018.

[22] D. Zhang, L. Guo, and G. E. Karniadakis. Learning in modal space: Solving time-dependent stochastic PDEs using physics-informed neural networks. *arXiv preprint arXiv:1905.01205*, 2019.

[23] D. Zhang, L. Lu, L. Guo, and G. E. Karniadakis. Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems. *arXiv preprint arXiv:1809.08327*, 2018.

[24] Y. Zhu, N. Zabaras, P.-S. Koutsourelakis, and P. Perdikaris. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *arXiv preprint arXiv:1901.06314*, 2019.