DeepXDE: A Deep Learning Library for Solving Differential Equations

Lu Lu[1], **Xuhui Meng**[1], **Zhiping Mao**[1], **George Em Karniadakis**[1]
[1]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. PINNs solve inverse problems similarly to forward problems. We also present a Python library for PINNs, DeepXDE. 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 we also demonstrate the capability of PINNs and the user-friendliness of DeepXDE for two different examples.

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. 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, and could break the curse of dimensionality. Among these approaches, one could use the PDE in strong form directly; in this form, automatic differentiation could be used directly to avoid truncation errors. This approach is called physics-informed neural networks (PINNs). An attractive feature of PINNs is that it can be used to solve inverse problems similarly to forward problems.

In this paper, we present PINN algorithms implemented in a Python library DeepXDE (https://github.com/lululxvi/deepxde). DeepXDE can be used to solve multi-physics problems, and supports complex-geometry domains based on the technique of constructive solid geometry (CSG), hence avoiding tedious and time-consuming computational geometry tasks. Last but not least, DeepXDE is designed to make the user code stay compact and manageable, resembling closely the mathematical formulation.

Copyright © 2020, for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CCBY 4.0).

1 Physics-informed neural networks

We consider the PDE parameterized by $\boldsymbol{\lambda}$ for the solution $u(\mathbf{x})$ with $\mathbf{x} = (x_1, \ldots, x_d)$ defined on a domain $\Omega \subset \mathbb{R}^d$:

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

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 $\mathbf{x}$.

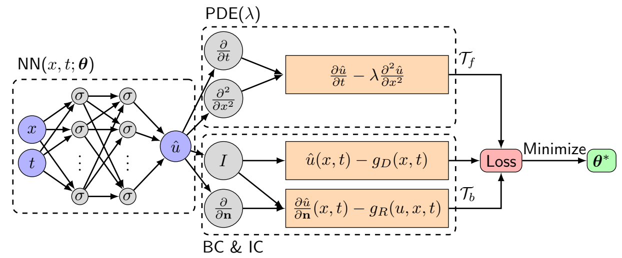

Figure 1: Schematic of a PINN for solving the diffusion equation $\frac{\partial u}{\partial t} = \lambda \frac{\partial^2 u}{\partial x^2}$ with mixed BCs $u(x,t) = g_D(x,t)$ on $\Gamma_D \subset \partial\Omega$ and $\frac{\partial u}{\partial \mathbf{n}}(x,t) = g_R(u,x,t)$ on $\Gamma_R \subset \partial\Omega$.

The algorithm of PINN is shown 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 $u(\mathbf{x})$. 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:

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

CEUR-WS.org/Vol-2587/article_14.pdf

where $\mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} \left\| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots; \boldsymbol{\lambda}\right) \right\|_2^2$, $\mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x} \in \mathcal{T}_b} \|\mathcal{B}(\hat{u}, \mathbf{x})\|_2^2$, 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".

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 by adding an extra term to Eq. (2): $\mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i) = \frac{1}{|\mathcal{T}_i|} \sum_{\mathbf{x} \in \mathcal{T}_i} \|\mathcal{I}(\hat{u}, \mathbf{x})\|_2^2$. We then optimize $\boldsymbol{\theta}$ and $\boldsymbol{\lambda}$ together: $\boldsymbol{\theta}^*, \boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\theta}, \boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T})$.

2 DeepXDE usage

In this section, we introduce the usage of DeepXDE. 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 1.

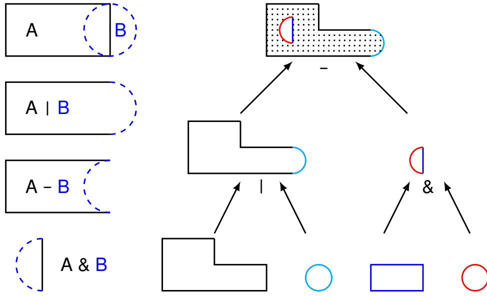

Figure 2: CSG examples. (**left**) A and B represent the rectangle and circle, respectively. $A|B$, $A - B$, and $A\&B$ are constructed from A and B. (**right**) A complex geometry (top) is constructed from a polygon, a rectangle and two circles (bottom).

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 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.

3 Demonstration example

We use PINNs and DeepXDE to solve inverse problems. 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. 3.

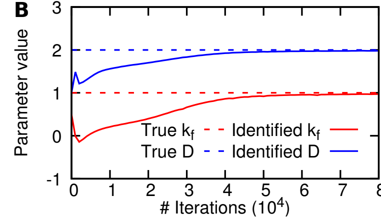

Figure 3: The identified values of diffusion-reaction system converge to the true values during the training process.

Procedure 1 Usage of DeepXDE for solving differential equations.

1. Specify the computational domain using the **`geometry`** module.
2. Specify the differential equations using the grammar of **`TensorFlow`**.
3. Specify the boundary and initial conditions.
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.
5. Construct a neural network using the **`maps`** module.
6. Define a **`Model`** by combining the PDE problem in Step 4 and the neural net in Step 5.
7. Call **`Model.compile`** to set the optimization hyperparameters, such as optimizer and learning rate. The weights in Eq. (2) can be set here by **`loss_weights`**.
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`**.
9. Call **`Model.predict`** to predict the PDE solution at different locations.

References

Lu, L.; Meng, X.; Mao, Z.; and Karniadakis, G. E. 2019. Deepxde: A deep learning library for solving differential equations. *arXiv preprint arXiv:1907.04502*.