DeepXDE: A Deep Learning Library for Solving Forward and Inverse Differential Equations

Lu Lu

Division of Applied Mathematics, Brown University

SIAM MDS20

Deep learning for partial differential equations (PDEs)

PDE-dependent approaches:

- image-like domain
  - e.g., (Long et al., *ICML*, 2018), (Zhu et al., *J Comput Phys*, 2019)
- parabolic PDEs, e.g., through Feynman-Kac formula
  - e.g., (Beck et al., *J Nonlinear Sci*, 2017), (Han et al., *PNAS*, 2018)
- variational form
  - e.g., (E & Yu, *Commun Math Stat*, 2018)

General approaches:

- Galerkin type projection
  - e.g., (Meade & Fernandez, *Math Comput Model*, 1994), (Kharazmi et al., *arXiv*, 2019)
- strong form (Physics-informed neural networks)
  - e.g., (Dissanayake & Phan-Thien, *Commun Numer Meth En*, 1994), (Lagaris et al., *IEEE Trans Neural Netw*, 1998), (Raissi et al., *J Comput Phys*, 2019)

BROWN

Inverse problems

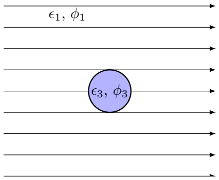
Challenge: *small* data + *incomplete* physics laws

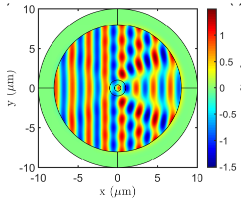
Invisible cloaking

joint work with Prof. Luca Dal Negro (Boston U)

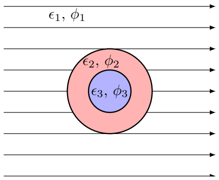
Permittivity ϵ

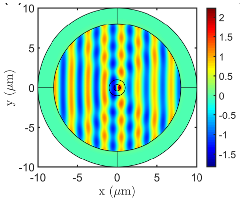
Electric field ϕ without coating





Electric field ϕ with coating

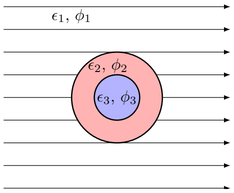




Chen, Lu, et al., *Opt Express*, 2020

Invisible cloaking

Goal: Given ϵ_1 and ϵ_3, find $\epsilon_2(x, y)$ s.t. $\phi_1 \approx \phi_{1,target}$



Helmholtz equation ($k = \frac{2\pi}{\lambda}$)

$$\nabla^2\phi_i + \epsilon_i k^2 \phi_i = 0, \qquad i = 1, 2, 3$$

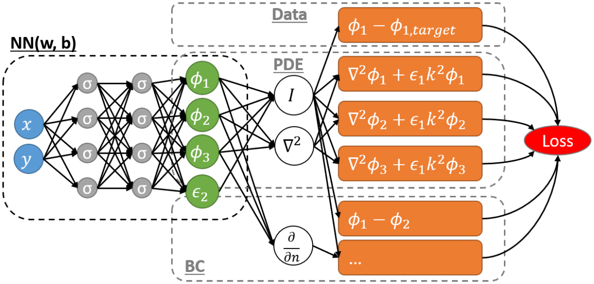
Boundary conditions:

- Outer circle: $\phi_1 = \phi_2,\ \epsilon_1 \frac{\partial \phi_1}{\partial \mathbf{n}} = \epsilon_2 \frac{\partial \phi_2}{\partial \mathbf{n}}$
- Inner circle: $\phi_2 = \phi_3,\ \epsilon_2 \frac{\partial \phi_2}{\partial \mathbf{n}} = \epsilon_3 \frac{\partial \phi_3}{\partial \mathbf{n}}$

Chen, Lu, et al., *Opt Express*, 2020

Physics-informed neural networks (PINNs)

Idea: Embed a PDE into the loss via automatic differentiation (AD)

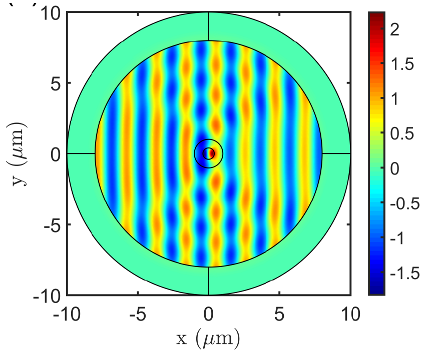


NN(w, b)
Data
PDE
BC
x
y
σ
ϕ_1
ϕ_2
ϕ_3
ϵ_2
I
∇^2
$\frac{\partial}{\partial n}$
$\phi_1 - \phi_{1,target}$
$\nabla^2\phi_1 + \epsilon_1 k^2 \phi_1$
$\nabla^2\phi_2 + \epsilon_1 k^2 \phi_2$
$\nabla^2\phi_3 + \epsilon_1 k^2 \phi_3$
$\phi_1 - \phi_2$
...
Loss

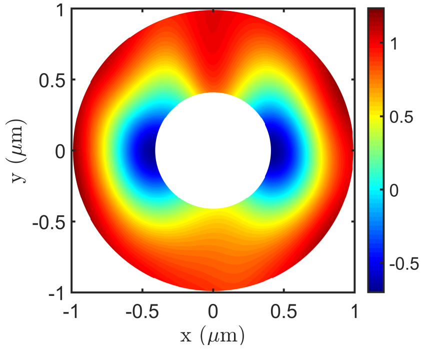
- mesh-free & particle-free
- inverse problems: seamlessly integrate data and physics
- black-box or noisy IC/BC/forcing terms (Pang*, Lu*, et al., *SIAM J Sci Comput*, 2019)
- a unified framework: PDE, integro-differential equations (Lu et al., *SIAM Rev*, accepted), fractional PDE (Pang*, Lu*, et al., *SIAM J Sci Comput*, 2019), stochastic PDE (Zhang, Lu, et al., *J Comput Phys*, 2019)

Invisible cloaking

Electric field ϕ_i



Permittivity ϵ_2



Chen, Lu, et al., *Opt Express*, 2020

Approximation: Loss $\to$ 0?

Theorem (Universal approximation theorem; Cybenko, 1989)

Let σ be any continuous sigmoidal function. Then finite sums of the form $G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j \cdot x + \theta_j)$ are dense in $C(I_d)$.

Theorem (Pinkus, 1999)

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 σ is not a polynomial. Then the space of single hidden layer neural nets

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

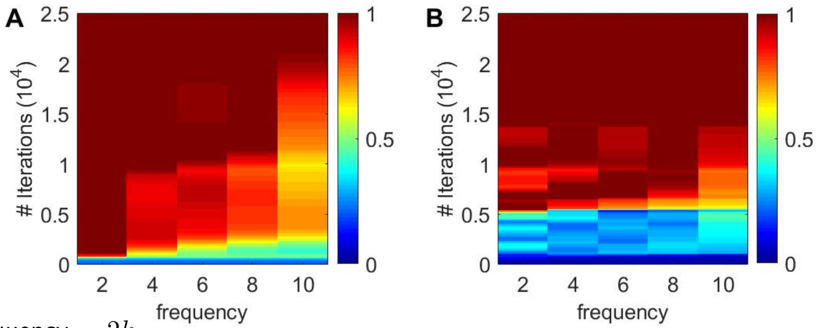
is dense in

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

Lu et al., *SIAM Rev*, accepted

Optimization

- A: approximate $f(x) = \sum_{k=1}^{5} \sin(2kx)/(2k)$
  - learn from low to high frequencies (Rahaman et al., *ICML*, 2019; Xu et al., arXiv, 2019)
- B: solve the Poisson equation $-f_{xx} = \sum_{k=1}^{5} 2k \sin(2kx)$
  - all frequencies are learned almost simultaneously
  - faster learning



Frequency $= 2k$

Lu et al., *SIAM Rev*, accepted

Generalization: Residual-based adaptive refinement (RAR)

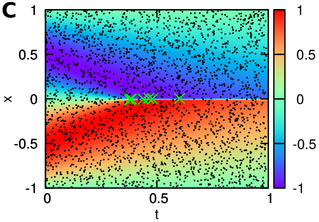
Challenge: Uniform residual points are not efficient for PDEs with steep solutions.

e.g., Burgers equation ($x \in [-1,1],\ t \in [0,1]$):

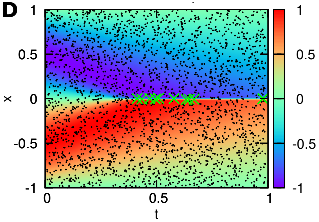
$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad u(x,0) = -\sin(\pi x), \quad u(-1,t) = u(1,t) = 0.$$

- **Idea**: adaptively add more points in locations with large PDE residual $\left\|f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right)\right\|$

C



D

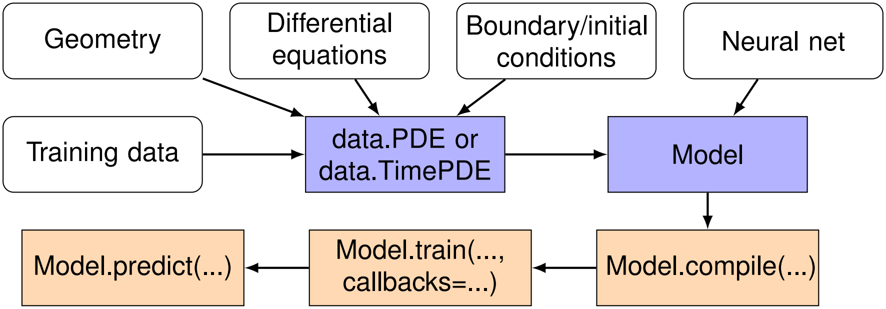


10,000 (Raissi et al., *J Comput Phys*, 2019) ↓ 2,540

Lu et al., *SIAM Rev*, accepted

Usage of DeepXDE

Solving differential equations in DeepXDE is no more than **specifying the problem using the build-in modules**.



I. Time-independent problems: Poisson equation

2D Poisson equation over an L-shaped domain $\Omega = [-1,1]^2 \setminus [0,1]^2$:

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

1. geometry

```python
geom = dde.geometry.Polygon(
    [[0, 0], [1, 0], [1, -1], [-1, -1], [-1, 1], [0, 1]])
```

2. PDE via automatic differentiation in TensorFlow (`tf.gradients`)

```python
def pde(x, y):
    dy_x = tf.gradients(y, x)[0]
    dy_x, dy_y = dy_x[:, 0:1], dy_x[:, 1:]
    dy_xx = tf.gradients(dy_x, x)[0][:, 0:1]
    dy_yy = tf.gradients(dy_y, x)[0][:, 1:]
    return -dy_xx - dy_yy - 1
```

3. BC: Dirichlet, Neumann, Robin, periodic, and a general BC

```python
def boundary(x, on_boundary):
    return on_boundary  # Default: entire geometry boundary
def func(x):
    return np.zeros([len(x), 1])  # Value

bc = dde.DirichletBC(geom, func, boundary)
```

I. Time-independent problems: Poisson equation

4. "data": geometry + PDE + BC + "training" points

```
data = dde.data.PDE(
    geom, pde, bc, num_domain=1000, num_boundary=100, ...)
```

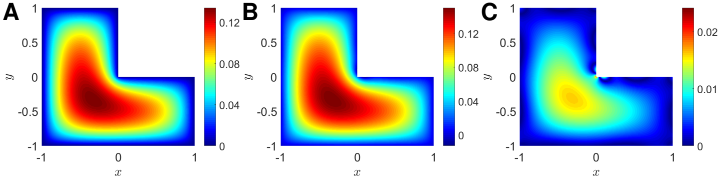
5. network, e.g., feed-forward network

```
net = dde.maps.FNN([2]+[50]*4+[1], "tanh", "Glorot uniform",
    ...)
```

6. model: data + network, and train

```
model = dde.Model(data, net)
model.compile("adam", lr=0.001, ...)
model.train(epochs=50000, ...)
```

(A) spectral element, (B) PINN, (C) error



II. Time-dependent problems: Diffusion equation

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

with Dirichlet BC. (Exact solution $u(x,t) = e^{-t}\sin(\pi x)$)

- geometry

```
geom = dde.geometry.Interval(-1, 1)
timedomain = dde.geometry.TimeDomain(0, 1)
geomtime = dde.geometry.GeometryXTime(geom, timedomain)
```

- IC, similar to Dirichlet BC

```
def func(x):
    return np.sin(np.pi * x[:, 0:1]) * np.exp(-x[:, 1:])

ic = dde.IC(geomtime, func, lambda _, on_initial: on_initial)
```

- “data”: geometry + PDE + BC/IC + “training” points

```
data = dde.data.TimePDE(
    geomtime, ..., [bc, ic],
    num_domain=40, num_boundary=20, num_initial=10, ...)
```

III. ODE/PDE system: Lorenz system

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

- ODE system

```python
def Lorenz_system(x, y):
    y1, y2, y3 = y[:, 0:1], y[:, 1:2], y[:, 2:]
    dy1_x = tf.gradients(y1, x)[0]
    dy2_x = tf.gradients(y2, x)[0]
    dy3_x = tf.gradients(y3, x)[0]
    return [
        dy1_x - C1 * (y2 - y1),
        dy2_x - y1 * (C2 - y3) + y2,
        dy3_x - y1 * y2 + C3 * y3
    ]
```

- ICs

```python
ic1 = dde.IC(geom, ..., ..., component=0)
ic2 = dde.IC(geom, ..., ..., component=1)
ic3 = dde.IC(geom, ..., ..., component=2)
```

IV. Integro-differential equations: Volterra IDE

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

- kernel

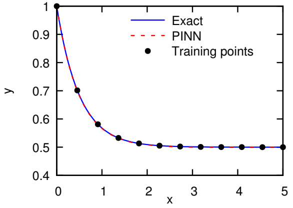
```python
def kernel(x, s):
    return np.exp(s - x)
```

- IDE

```python
def ide(x, y, int_mat):
    rhs = tf.matmul(int_mat, y)  # \int_0^x exp(t-x) y(t) dt
    lhs1 = tf.gradients(y, x)[0]  # dy/dx
    return (lhs1 + y)[: tf.size(rhs)] - rhs
```

- "data": geometry + IDE + BC + "training" points

```python
data = dde.data.IDE(..., ide, ..., quad_deg=20, kernel=kernel,
    ...)
```



V. Inverse problems

A diffusion-reaction system on $x \in [0,1], t \in [0,10]$:

$$\frac{\partial C_A}{\partial t} = D\frac{\partial^2 C_A}{\partial x^2} - k_f C_A C_B^2, \quad \frac{\partial C_B}{\partial t} = D\frac{\partial^2 C_B}{\partial x^2} - 2k_f C_A C_B^2$$

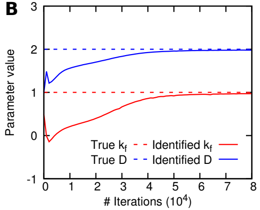
- Define D and k_f as trainable variables

```
kf = tf.Variable(0.05)
D = tf.Variable(1.0)
```

- Define C_A measurements as Dirichlet BC (the same for C_B)

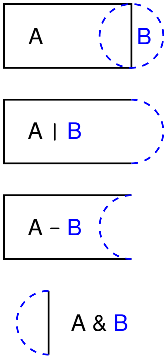
```
observe_x = ...  # All the locations of measurements
observe_Ca = ...  # The corresponding measurements of Ca
ptset = dde.bc.PointSet(observe_x)
observe = dde.DirichletBC(geomtime, ptset.values_to_func(
    observe_Ca), lambda x, _: ptset.inside(x), component=0)
```

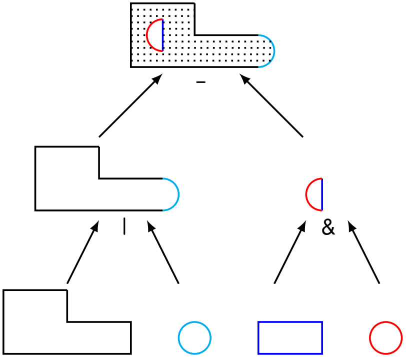
B



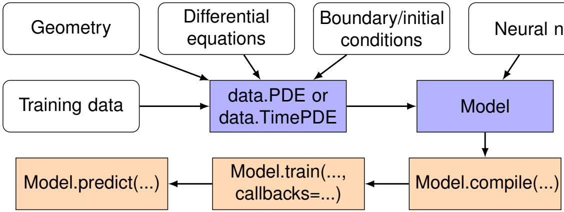
Complex geometry: Constructive solid geometry

- Primitive geometries
  - **`interval`**, **`triangle`**, **`rectangle`**, **`polygon`**, **`disk`**, **`cuboid`**, **`sphere`**
- boolean operations:
  - Union $A|B$
  - difference $A - B$
  - intersection $A\&B$





DeepXDE



- Short and comprehensive code. NO redundant code!
- Only TensorFlow knowledge: `tf.gradients`
- Almost all useful techniques, e.g., uncertainty quantification via dropout, early stopping, saving & loading network
- Well-structured and highly configurable; `callback` mechanism

Other functions

- Multi-fidelity learning (Lu et al., *PNAS*, 2020)
- Learning nonlinear operators (Lu et al., arXiv:1910.03193, 2019)

DeepXDE

Installation

- `pip install deepxde`
- `conda install -c conda-forge deepxde`

`https://github.com/lululxvi/deepxde`
`https://deepxde.readthedocs.io`

- Reporting bugs
- Suggesting enhancements
- Pull requests
- Questions