Implementation and Python Library Tutorial for PINNs to Handle Dynamical Systems

2021-11-28

Hyemin Gu

Outline

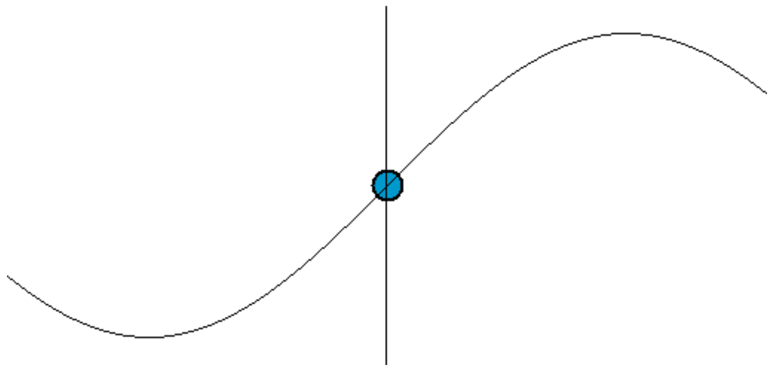
1. PINNs for Dynamical systems and Implementation
    - Damped Harmonic Oscillator
    - NN vs. PINN
    - Implementations using pytorch (A general ML library)
2. Python library specialized for PINNs: DeepXDE
    - Installation and Dependencies
    - Workflow
    - How to specify DE w/ Inverse Lorenz model
    - Example problem: Forward/Inverse Lorenz model

1. PINNs for Dynamical systems and Implementation

- Damped harmonic oscillator:

$$m\frac{d^2u}{dx^2} + \mu\frac{du}{dx} + ku = 0$$

- Given finite train data $\{x_i, u(x_i)\}_{i=1,\cdots,N}$
- Learning problem!



Neural Network vs. PINN

Neural Network

- $\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_i^N \left(u_{NN}(x_i;\boldsymbol{\theta}) - u(x_i)\right)^2$
- **Learn from data**

Good interpolation if trained well

Extrapolation?

PINN

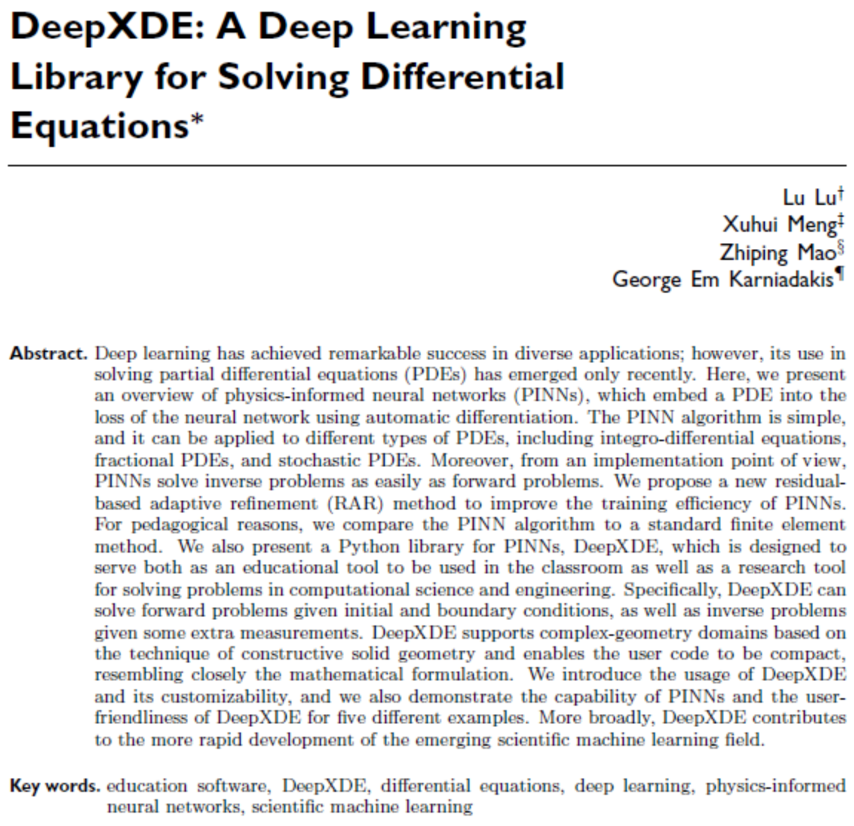
- $\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_i^N \left(u_{NN}(x_i;\boldsymbol{\theta}) - u(x_i)\right)^2 + \frac{1}{M}\sum_j^M \left(R u_{NN}(x_j;\boldsymbol{\theta})\right)^2$

  where $R = m\frac{d^2u}{dx^2} + \mu\frac{du}{dx} + ku$.
- **Learn from data + differential equation (Physics rule)**

https://benmoseley.blog/blog/ : Illustration with Python codes using Pytorch

2. Python library for PINNs: DeepXDE

- A specialized library for PINNs
- Key features:
  - Various applicable problems:
    - Forward problem
    - Inverse problem
    - Integro-differential equations
  - Complex domain
  - Residual-based Adaptive Refinement



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

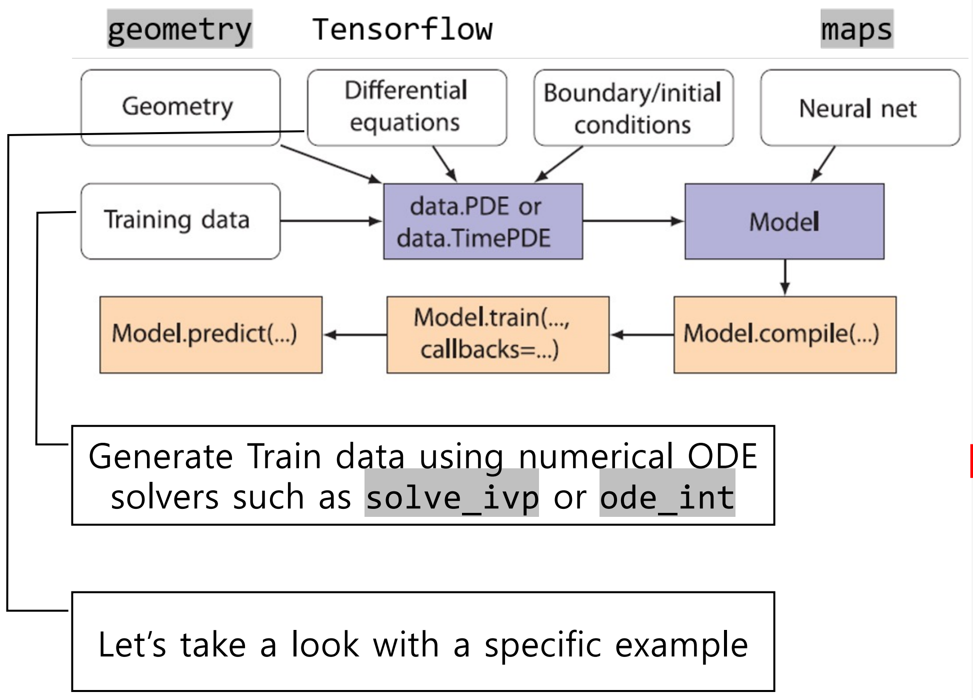
Installation and Dependencies

- Require one of these python libraries:
  - TensorFlow 1.x: TensorFlow>=2.2.0 **(Default)**
  - TensorFlow 2.x: TensorFlow>=2.2.0 and TensorFlow Probability
  - PyTorch: PyTorch
- Install using pip

```
$ pip install deepxde
```

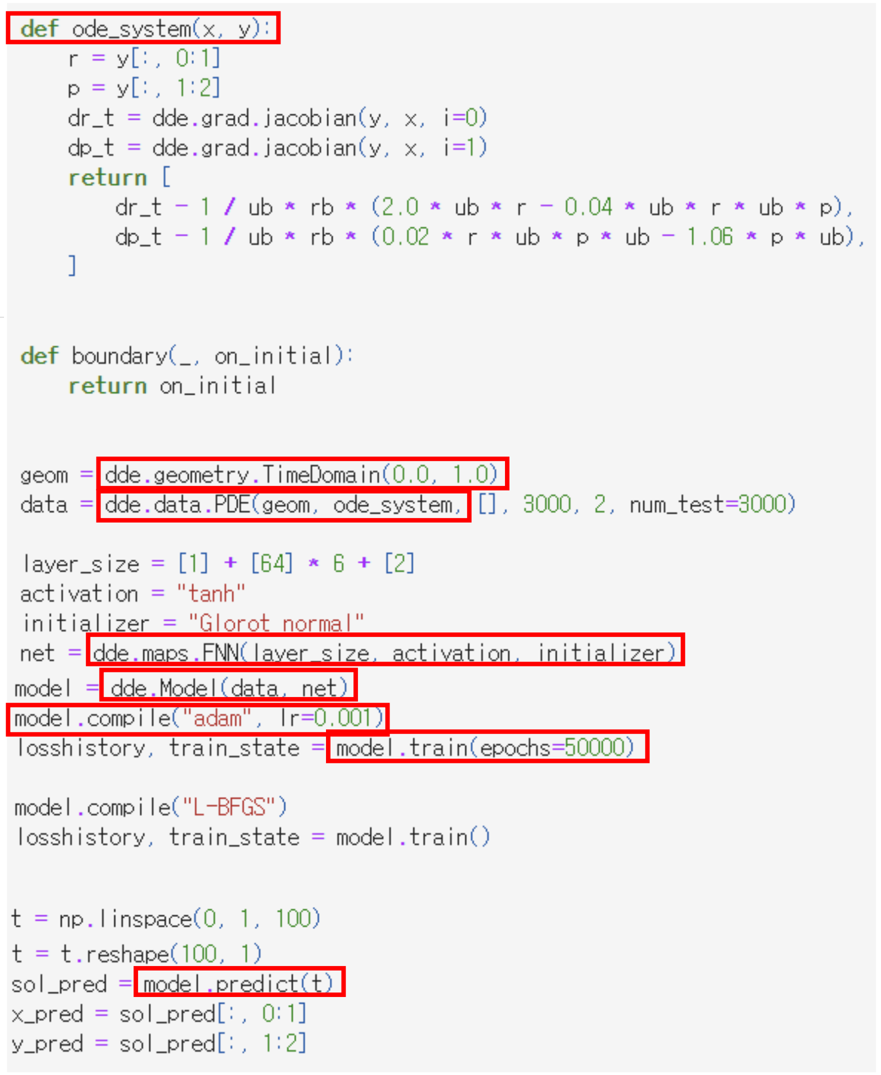
- Other dependencies
  - Matplotlib
  - NumPy
  - scikit-learn
  - scikit-optimize
  - SciPy

Workflow



Generate Train data using numerical ODE solvers such as `solve_ivp` or `ode_int`

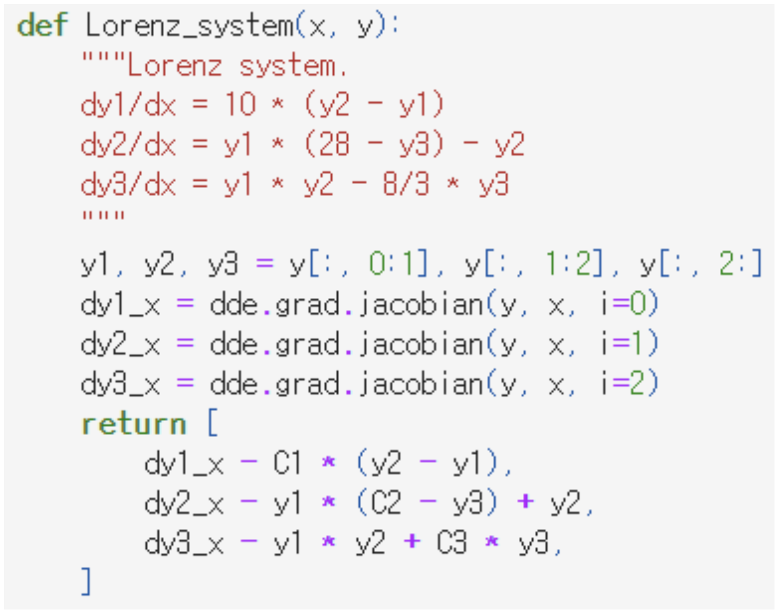
Let's take a look with a specific example



```
def ode_system(x, y):
    r = y[:, 0:1]
    p = y[:, 1:2]
    dr_t = dde.grad.jacobian(y, x, i=0)
    dp_t = dde.grad.jacobian(y, x, i=1)
    return [
        dr_t - 1 / ub * rb * (2.0 * ub * r - 0.04 * ub * r * ub * p),
        dp_t - 1 / ub * rb * (0.02 * r * ub * p * ub - 1.06 * p * ub),
    ]


def boundary(_, on_initial):
    return on_initial


geom = dde.geometry.TimeDomain(0.0, 1.0)
data = dde.data.PDE(geom, ode_system, [], 3000, 2, num_test=3000)

layer_size = [1] + [64] * 6 + [2]
activation = "tanh"
initializer = "Glorot normal"
net = dde.maps.FNN(layer_size, activation, initializer)
model = dde.Model(data, net)
model.compile("adam", lr=0.001)
losshistory, train_state = model.train(epochs=50000)

model.compile("L-BFGS")
losshistory, train_state = model.train()


t = np.linspace(0, 1, 100)
t = t.reshape(100, 1)
sol_pred = model.predict(t)
x_pred = sol_pred[:, 0:1]
y_pred = sol_pred[:, 1:2]
```

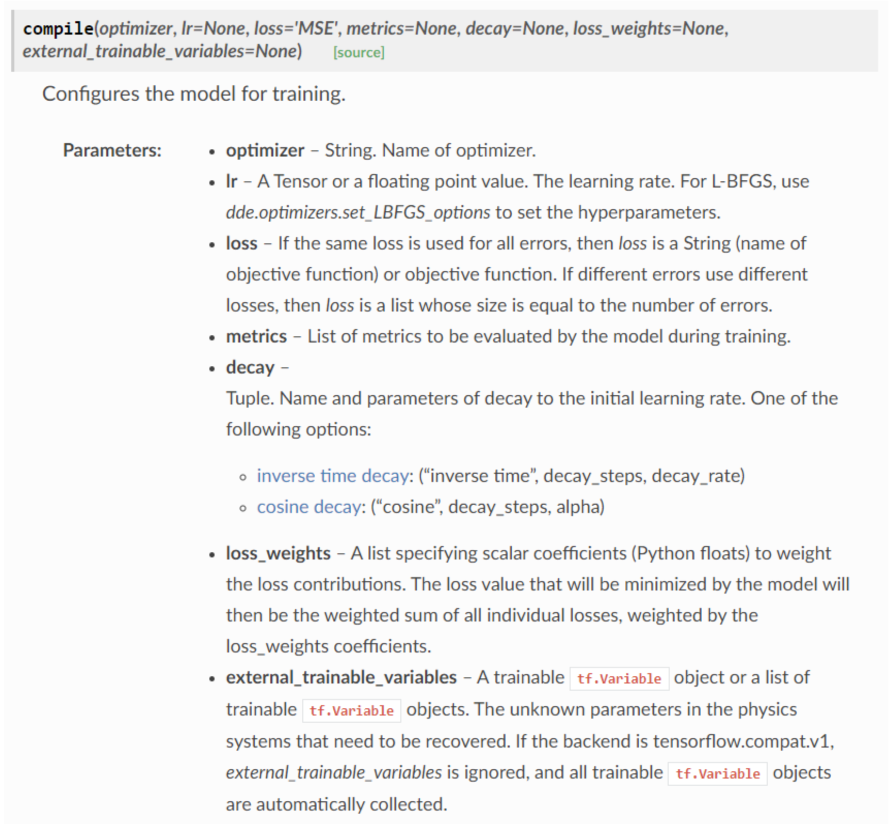
How to specify DE: example

- **Inverse Lorenz model**



```
def Lorenz_system(x, y):
    """Lorenz system.
    dy1/dx = 10 * (y2 - y1)
    dy2/dx = y1 * (28 - y3) - y2
    dy3/dx = y1 * y2 - 8/3 * y3
    """
    y1, y2, y3 = y[:, 0:1], y[:, 1:2], y[:, 2:]
    dy1_x = dde.grad.jacobian(y, x, i=0)
    dy2_x = dde.grad.jacobian(y, x, i=1)
    dy3_x = dde.grad.jacobian(y, x, i=2)
    return [
        dy1_x - C1 * (y2 - y1),
        dy2_x - y1 * (C2 - y3) + y2,
        dy3_x - y1 * y2 + C3 * y3,
    ]
```

Documentation https://deepxde.readthedocs.io/en/latest/index.html



`compile`(*optimizer, lr=None, loss='MSE', metrics=None, decay=None, loss_weights=None, external_trainable_variables=None*) [source]

Configures the model for training.

Parameters:

- **optimizer** – String. Name of optimizer.
- **lr** – A Tensor or a floating point value. The learning rate. For L-BFGS, use *dde.optimizers.set_LBFGS_options* to set the hyperparameters.
- **loss** – If the same loss is used for all errors, then *loss* is a String (name of objective function) or objective function. If different errors use different losses, then *loss* is a list whose size is equal to the number of errors.
- **metrics** – List of metrics to be evaluated by the model during training.
- **decay** –
  Tuple. Name and parameters of decay to the initial learning rate. One of the following options:
  - inverse time decay: ("inverse time", decay_steps, decay_rate)
  - cosine decay: ("cosine", decay_steps, alpha)
- **loss_weights** – A list specifying scalar coefficients (Python floats) to weight the loss contributions. The loss value that will be minimized by the model will then be the weighted sum of all individual losses, weighted by the loss_weights coefficients.
- **external_trainable_variables** – A trainable `tf.Variable` object or a list of trainable `tf.Variable` objects. The unknown parameters in the physics systems that need to be recovered. If the backend is tensorflow.compat.v1, *external_trainable_variables* is ignored, and all trainable `tf.Variable` objects are automatically collected.