PDEasy

Lightweight PINN-PDE Solver for Research,

Balancing Abstraction and Flexibility for Algorithm Innovation

Author: Guanyu Pan

Version: 0.1.0

Date: 2025-02-18

Motivation (面向科研的 PINN-PDE 求解库)

- 现有的 PINN 库
  - 封装程度高
  - 面向工程部署 or 新手入门
  - 不易实现新的 idea
  - DeepXDE, …

- 面向科研工作者的？
  - 平衡封装程度与扩展性
  - 加速算法创新.
  - PDEasy

Key Features (PINN 求解流程化)

- PINN 求解流程规范为:

1. 定义超参数
2. 定义数据集
3. 定义网络模型
4. 定义 **PINN** 模型
5. **训练模型**
6. 评估与可视化.

1. 定义超参数
(定义域、采样点、网络结构等)

```
DOMAIN = (-1, 1, 0, 1)  # (x_min, x_max, t_min, t_max)
N_RES = 5000
N_BCS = 200
N_ICS = 200
N_ITERS = 20000
NN_LAYERS = [2] + [80]*4 + [1]
```

2. 定义数据集
(调用随机/网格采样, 或自定义采样方法)

```
class Dataset(Dataset1DT):
    def __init__(self, domain):...

    def custom_update(self, n_res=N_RES, n_bcs=N_BCS, n_ics=N_ICS):...
```

3. 定义网络模型
(调用已封装的网络, 或自定义网络)

```
class MLP(nn.Module):
    def __init__(...

    def forward(...
```

4. 定义 PINN 模型
(根据 PDE 重写方法, 提取数据集计算 loss)

```
class PINN(PINNForward):
    def __init__(self, network_solution, should_normalize=True):...

    def forward(self, data_dict):...

    def net_res(self, X):...

    def net_bcs(self, X):...

    def net_ics(self, X):...
```

5. 训练模型
(使用默认的训练方式, 或自定义新算法)

```
for it in range(N_ITERS):  # Training loop...
```

6. 评估与可视化
(调用已封装的评估与绘图函数, 快速绘图)

```
plot_loss_from_logger(logger, FIGURE_DIR, show=True)
plot_error_from_logger(logger, FIGURE_DIR, show=True)

plot_solution_from_data(...
```

Key Features (PINN 求解流程化)

- 流程中各个模块均适度封装, 兼顾扩展性:
  1. **网络模型**
  2. PINN 正反问题
  3. 可视化.

1. 网络模型, 统一接口

(仅需通过 list, 如 [2, 20, 20, 1], 即可调用, 激活函数、网络初始化方法也直接修改)

```
Example::
    >>> NN_LAYERS = [2, 20, 20, 1]
    >>> network = MLP(NN_LAYERS)
    >>> ...
    >>> network = ModifiedMLP(NN_LAYERS)
```

```
Example::
    >>> NN_LAYERS = [1, 100, 100, 1]
    >>> network = MFF1D(NN_LAYERS)
    >>> ...
```

```
Example::
    >>> NN_LAYERS = [2, 20, 20, 3]
    >>> network = MHN(NN_LAYERS)
```

- **MLP 类**
  - 普通的 **MLP**
  - 改进的 **Modified MLP**
- **Fourier Feature Network 类**
  - **MFF1D**: Multiscale Fourier Feature Model for 1D Space
  - **STFF1DT**: Spatio Temporal Fourier Feature Model for 1D Space and Time
  - **STMFF1DT**: Spatio Temporal Multiscale Fourier Feature Model for 1D Space and Time
  - **FF2D**: Fourier Feature Model for 2D Space
  - **MFF2D**: Multiscale Fourier Feature Model for 2D Space
  - **FF2DT**: Fourier Feature Model for 2D Space and Time
- **Multi-Head Network 类**
  - 最后两层分头输出的 **MHN**
- 后续会更新 KAN 等有效模型.

Key Features (PINN 求解流程化)

- 流程中各个模块均适度封装, 兼顾扩展性:
  1. 各种网络模型
  2. **PINN 正反问题**
  3. 可视化.

2. PINN 正反问题

以正问题基础框架为例
- **forward**
  - 从 **data_dict** 读取数据集
  - 通过 net_xxx 计算 loss
- **net_res**
  - 根据 PDE, 求偏导, 构造 loss
  - 调用 self.grad 即可求任意阶导数
- **net_bcs**
  - 根据边界条件改写
- **net_ics**
  - 根据初始条件改写.

```
class PINN(PINNForward):
    def __init__(self, network_solution, should_normalize=True):
        super().__init__(network_solution, should_normalize)

    def forward(self, data_dict):
        # 读取 data_dict 的数据
        X_res, X_bcs, X_ics = data_dict["X_res"], data_dict["X_bcs"], data_dict["X_ics"]

        # 计算 point-wise loss
        # 便于后续引入权重策略
        loss_dict = {}
        loss_dict['pw_loss_res'] = self.net_res(X_res) ** 2
        loss_dict['pw_loss_bcs'] = self.net_bcs(X_bcs) ** 2
        loss_dict['pw_loss_ics'] = self.net_ics(X_ics) ** 2

        return loss_dict

    def net_res(self, X):
        columns = self.split_X_columns_and_require_grad(X)
        x, t = columns
        u = self.net_sol([x, t])

        u_x = self.grad(u, x, 1)
        u_t = self.grad(u, t, 1)
        u_xx = self.grad(u, x, 2)
        res_pred = u_t + u * u_x - (0.01 / torch.pi) * u_xx
        return res_pred

    def net_bcs(self, X):
        u = self.net_sol(X)
        bcs_pred = u - 0
        return bcs_pred

    def net_ics(self, X):
        u = self.net_sol(X)
        ics_pred = u + torch.sin(torch.pi * X[:, [0]])
        return ics_pred
```

Key Features (PINN 求解流程化)

- 流程中各个模块均适度封装, 兼顾扩展性:

  1. 各种网络模型
  2. **PINN 正反问题**
  3. 可视化.

2. PINN 正反问题

- **data_dict**
  - 是以 Python 的字典数据结构存储的数据集. 直观地通过 key 调用
  - 增加采样方法, 直接读取和修改对应的 key-value 值即可.
- **self.grad**
  - 是已封装的求导方法
  - self.grad(u, x, 2) 即可求 u 对 x 的 2 阶导
  - 同理求 n 阶导

```python
class PINN(PINNForward):
    def __init__(self, network_solution, should_normalize=True):
        super().__init__(network_solution, should_normalize)

    def forward(self, data_dict):
        # 读取 data_dict 的数据
        X_res, X_bcs, X_ics = data_dict["X_res"], data_dict["X_bcs"], data_dict["X_ics"]

        # 计算 point-wise loss
        # 便于后续引入权重策略
        loss_dict = {}
        loss_dict['pw_loss_res'] = self.net_res(X_res) ** 2
        loss_dict['pw_loss_bcs'] = self.net_bcs(X_bcs) ** 2
        loss_dict['pw_loss_ics'] = self.net_ics(X_ics) ** 2

        return loss_dict

    def net_res(self, X):
        columns = self.split_X_columns_and_require_grad(X)
        x, t = columns
        u = self.net_sol([x, t])

        u_x = self.grad(u, x, 1)
        u_t = self.grad(u, t, 1)
        u_xx = self.grad(u, x, 2)
        res_pred = u_t + u * u_x - (0.01 / torch.pi) * u_xx
        return res_pred

    def net_bcs(self, X):
        u = self.net_sol(X)
        bcs_pred = u - 0
        return bcs_pred

    def net_ics(self, X):
        u = self.net_sol(X)
        ics_pred = u + torch.sin(torch.pi * X[:, [0]])
        return ics_pred
```

Key Features (PINN 求解流程化)

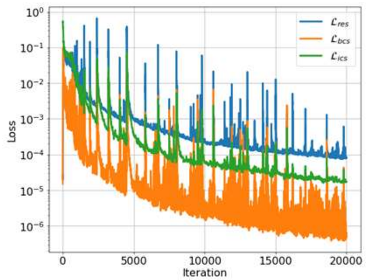
- 流程中各个模块均适度封装, 兼顾扩展性:

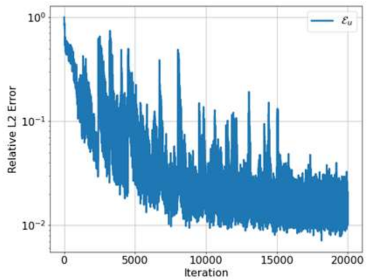
  1. 各种网络模型
  2. PINN 正反问题
  3. **3.** 可视化.

3. 可视化, 灵活调用

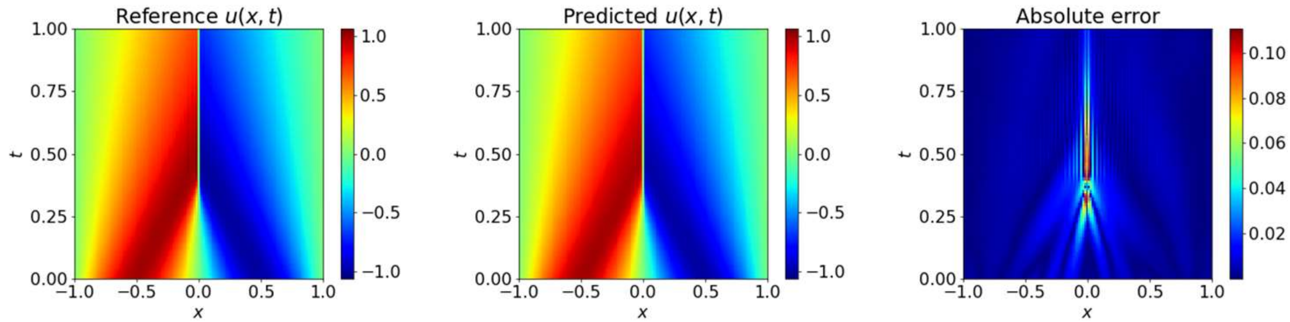
(最少仅需传入绘图数据, 有需要可以调整图标签、标题等, 都已封装到绘图函数中)

```
plot_loss_from_logger(logger, FIGURE_DIR, show=True)
plot_error_from_logger(logger, FIGURE_DIR, show=True)
```






```
plot_solution_from_data(
    FIGURE_DIR,
    x_grid=xx.reshape(u_shape),
    y_grid=tt.reshape(u_shape),
    sol=u.reshape(u_shape),
    sol_pred=u_pred.reshape(u_shape),

    x_label='$x$',
    y_label='$t$',

    x_ticks=np.linspace(-1, 1, 5),
    y_ticks=np.linspace(0, 1, 5),

    title_left=r'Reference $u(x,t)$',
    title_middle=r'Predicted $u(x,t)$',
    title_right=r'Absolute error'
)
```



Key Features (PINN 求解流程化)

- 简化偏导写法:
  - 在 PDEasy 中仅需调用 self.grad 即可直观求导 (左列)
  - 而原先需要调用 torch.autograd.grad, 并且对于多输入或多输出需要复杂提取操作 (右列)

```python
def net_res(self, X):
    columns = self.split_X_columns_and_require_grad(X)
    x, t = columns
    u = self.net_sol([x, t])

    u_x = self.grad(u, x, 1)
    u_t = self.grad(u, t, 1)
    u_xx = self.grad(u, x, 2)
    res_pred = u_t + u * u_x - (0.01 / torch.pi) * u_xx
    return res_pred
```

```python
def net_res(self, X):
    x, y, t = self.split_X_columns_and_require_grad(X)
    u, v, p = self.net_sol([x, y, t])

    u_t = self.grad(u, t, 1)
    u_x = self.grad(u, x, 1)
    u_y = self.grad(u, y, 1)
    p_x = self.grad(p, x, 1)
    u_xx = self.grad(u, x, 2)
    u_yy = self.grad(u, y, 2)

    v_t = self.grad(v, t, 1)
    v_x = self.grad(v, x, 1)
    v_y = self.grad(v, y, 1)
    p_y = self.grad(p, y, 1)
    v_xx = self.grad(v, x, 2)
    v_yy = self.grad(v, y, 2)

    nu = self.net_param(t, column_index=-1)

    # 控制方程
    u_res_pred = u_t + (u*u_x + v*u_y) + p_x - nu * (u_xx + u_yy)
    v_res_pred = v_t + (u*v_x + v*v_y) + p_y - nu * (v_xx + v_yy)
    return (u_res_pred, v_res_pred)
```

```python
def net_res(self, X):
    X.requires_grad_(True)
    u = self.net_u(X)

    # 求u的一阶导
    grad_u = self.grad(u, X)[0]
    u_x = grad_u[:, [0]]
    u_t = grad_u[:, [1]]
    # 求u的二阶导
    u_xx = self.grad(u_x, X)[0][:, [0]]

    res_pred = u_t + u * u_x - (0.01 / torch.pi) * u_xx

    return res_pred
```

```python
def net_f(self, X):
    X.requires_grad_(True)

    uvp = self.net_u(X)
    u = uvp[:, [0]]
    v = uvp[:, [1]]
    p = uvp[:, [2]]

    # 求u的时间导数 -------
    grad_u = self.grad(u, X)[0]
    u_t = grad_u[:, [-1]]
    # 求u的一阶导
    u_x = grad_u[:, [0]]
    u_y = grad_u[:, [1]]
    # 求u的二阶导
    u_xx = self.grad(u_x, X)[0][:, [0]]
    u_yy = self.grad(u_y, X)[0][:, [1]]

    # 求v的时间导数 -------
    grad_v = self.grad(v, X)[0]
    v_t = grad_v[:, [-1]]
    # 求v的一阶导
    v_x = grad_v[:, [0]]
    v_y = grad_v[:, [1]]
    # 求v的二阶导
    v_xx = self.grad(v_x, X)[0][:, [0]]
    v_yy = self.grad(v_y, X)[0][:, [1]]

    # 求p的一阶导
    grad_p = self.grad(p, X)[0]
    p_x = grad_p[:, [0]]
    p_y = grad_p[:, [1]]

    # net_param预测反演参数
    nu = self.net_param(X[:, [-1]])

    # 控制方程
    f_u_res_pred = u_t + (u*u_x + v*u_y) + p_x - nu * (u_xx + u_yy)
    f_v_res_pred = v_t + (u*v_x + v*v_y) + p_y - nu * (v_xx + v_yy)
```

Key Features (可以灵活扩展自己的 idea)

- 数据以 dict 数据结构流动, 轻松读取或修改:
  - **data_dict** 存储采样点
  - **loss_dict** 存储 loss
  - 采样类算法、权重类算法等, 均可在训练循环中调用 data_dict, loss_dict 实现

```
for it in range(N_ITERS):
    pinn.zero_grad()                                       # 清除梯度
    loss_dict = pinn(dataset.data_dict)                    # 计算 point-wise loss

    pw_loss_res = loss_dict["pw_loss_res"]                 # 提取 point-wise loss
    pw_loss_bcs = loss_dict["pw_loss_bcs"]
    pw_loss_ics = loss_dict["pw_loss_ics"]

    loss_res = torch.mean(pw_loss_res)                     # 计算 loss
    loss_bcs = torch.mean(pw_loss_bcs)
    loss_ics = torch.mean(pw_loss_ics)

    loss = loss_res + loss_bcs + loss_ics

    loss.backward()                                        # 反向传播
    optimizer.step()                                       # 更新网络参数
    scheduler.step(loss)                                   # 调整学习率
```

注: loss_dict 中存储的 loss 信息是 point-wise (逐点) 的, 也就是对应于 dataset 的每一个采样点, 很容易加入重采样和 loss 权重算法.

Key Features (可以灵活扩展自己的 idea)

- PINN 类提供了扩展接口, 方便作输出变换:
  - 通过 **net_sol_output_transform** 对网络输出的解作变换, 例如约束其满足边界和初始条件
  - 通过 **net_param_output_transform** 对网络输出的反演参数作变换, 例如尺度放缩或平移.

```python
class PINN(PINNForward):
    def __init__(self, network_solution, should_normalize=True):...

    def forward(self, data_dict):...

    def net_res(self, X):...

    def net_bcs(self, X):...

    def net_ics(self, X):...

    def net_sol_output_transform(self, X, u):
        # x^{2} \cos(\pi x) + t (1 - x^{2}) u
        x, t = X
        return x**2 * torch.cos(torch.pi * x) + t * (1 - x**2) * u

class PINN(PINNInverse):
    def __init__(self, network_solution, network_parameter, should_normalize=True):...

    def forward(self, data_dict):...

    def net_res(self, X):...

    def net_param_output_transform(self, X, parameter):
        # 作尺度变换
        lam_1, lam_2 = parameter
        lam_1 *= 1.
        lam_2 *= 0.1
        parameter = [lam_1, lam_2]
        return parameter
```

Key Features (全面的训练信息监控)

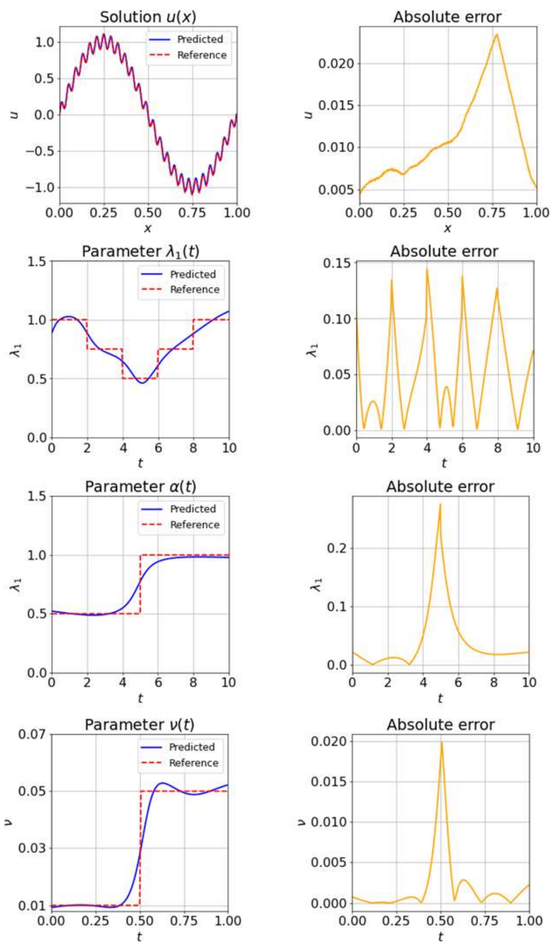
- Logger 类提供了自动化记录:
  - 自动打印. 通过预先给定的 log_keys, 在训练过程中传入 loss 和 error, 即可自动记录并打印
  - 快速绘图. 通过 logger 可直接绘图.

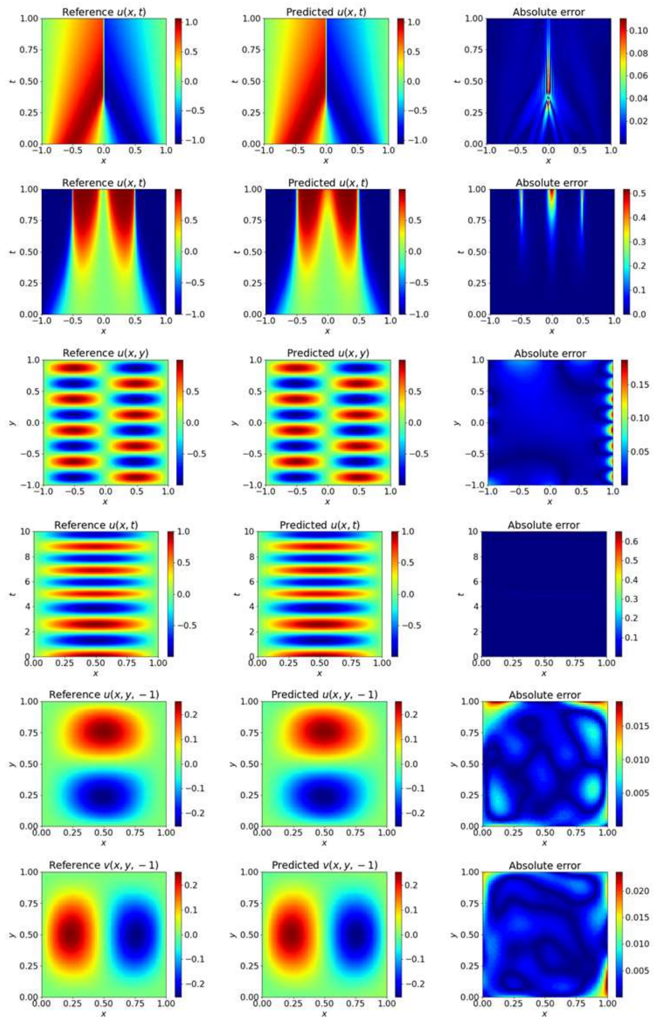
```
Iter #     0/20000 Time 0.2s     loss: 5.195e-01, loss_res: 5.488e-03, loss_bcs: 1.979e-03, loss_ics: 5.120e-01, error_u: 9.599e-01
Iter #   100/20000 Time 1.1s     loss: 1.544e-01, loss_res: 6.718e-02, loss_bcs: 6.282e-03, loss_ics: 8.097e-02, error_u: 5.176e-01
Iter #   200/20000 Time 2.1s     loss: 1.181e-01, loss_res: 4.705e-02, loss_bcs: 1.336e-03, loss_ics: 6.976e-02, error_u: 5.301e-01
Iter #   300/20000 Time 3.2s     loss: 1.098e-01, loss_res: 4.850e-02, loss_bcs: 7.681e-04, loss_ics: 6.058e-02, error_u: 4.622e-01
Iter #   400/20000 Time 4.3s     loss: 9.396e-02, loss_res: 3.596e-02, loss_bcs: 9.369e-04, loss_ics: 5.706e-02, error_u: 4.421e-01
Iter #   500/20000 Time 5.4s     loss: 8.505e-02, loss_res: 3.534e-02, loss_bcs: 6.640e-04, loss_ics: 4.905e-02, error_u: 4.402e-01
```

```python
log_keys = ['iter', 'loss', 'loss_res', 'loss_bcs', 'loss_ics', 'error_u']
logger = Logger(LOG_DIR, log_keys, num_iters=N_ITERS, print_interval=100)

# ---------------------------------
# --- 开始训练 打印并保存训练信息 ---
# ---------------------------------
best_loss = np.inf
for it in range(N_ITERS):
    ...

    error_u, _ = relative_error_of_solution(pinn, ref_data=(X, u), num_sample=500)

    logger.record(                              # 保存训练信息
        iter=it,                                # 每隔一定次数自动打印
        loss=loss.item(),
        loss_res=loss_res.item(),
        loss_bcs=loss_bcs.item(),
        loss_ics=loss_ics.item(),
        error_u=error_u
    )

    if it % 100 == 0:···

    if loss.item() < best_loss:                 # 保存最优模型···

logger.print_elapsed_time()
logger.save()
logger.load()
plot_loss_from_logger(logger, FIGURE_DIR, show=True)
plot_error_from_logger(logger, FIGURE_DIR, show=True)
```

Example (已实现 7 个算例)

- Forward problem 正问题
  - **Poisson**_Forward_1D
  - **Burgers**_Forward_1DT
  - **AllenCahn**_Forward_1DT
  - **Helmholtz**_Forward_2D
- Inverse problem 反问题
  - **Burgers**_Inverse_1DT
  - **SineGordon**_Inverse_1DT
  - **NavierStokes**_Inverse_2DT





Reference

- …

- …