HiNPS

发行版本 *0.1*

YipZLF

2022 年 08 月 19 日

Contents

HiNPS, High-performance Neural Network PDE Solver, 是一个基于神经网络方法求解偏微分方程的软件。

友情提示

本项目依然在活跃开发中。This project is under active developtment.

CHAPTER 1

快速入门

下面的例子将帮助你快速进行 HiNPS 的安装和运行，我们还将通过一个例子向你展示 HiNPS 求解偏微分方程的过程。

1.1 安装

首先从 GitHub 获取我们的源代码

```
git clone https://github.com/chaoyanggroup/hinps
cd hinps
```

我们推荐使用 `conda` 进行环境配置。创建一个名为 `hinps` 的 python3.8 环境。

```
conda activate
conda create -n hinps python==3.8
```

请根据你的安装环境首先安装PyTorch(版本要求 >=1.10)。需要使用 GPU 则请将 cudatoolkit=11.3 改为你的 CUDA 版本号。

```
conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch
```

如果只需要在 CPU 上运行，请使用

```
conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

接下来准备 MPI。你需要首先在你的机器上安装 MPI，集群上可以通过 `module avail` 检查是否存在已安装好的 MPI，如果有的话请加载。

如果没有我们推荐使用 OpenMPI。

```
conda install openmpi mpi4py
```

随后安装 mpi4py

```
conda install mpi4py
```

1.1.1 安装 HiNPS

在 HiNPS 文件夹下，可以通过 setup.py 进行安装。

```
python setup.py install
```

如果上述过程出错了，可以手动安装以下依赖包

```
pip install sympy matplotlib pandas scipy
```

进入 python，检查 HiNPS 版本从而检查安装是否正确

```
python
import hinps
hinps.__version__   #输出应为当前版本号(0.1)
```

1.2 第一个算例

下面我们通过一个算例解释如何使用 HiNPS 进行偏微分方程的求解。我们选择的是一个定义在 $T \times \Omega$ 上的热传导问题，其中 $T = [0, 1], \Omega = [0, 1]^3$。给定狄利克雷边界条件和初始温度条件。我们需要编写运行脚本的代码来表示下述的问题。

$$\frac{\partial u}{\partial t} - \Delta u = r, \quad on \ \Omega$$

$$u = r_d, \quad on \ \partial\Omega$$

$$u_{|t=0} = r_i, \quad on \ \partial\Omega$$

首先我们进行初始化，在这里我们会解析来自命令行的参数，返回的 args 则包括了这些参数。

```
import hinps as np
...
args = hp.init()
```

1.2.1 定义求解区域

准备好求解区域的定义。我们把求解区域的定义分成两部分，一部分是空间区域的定义。在这个例子中我们选择的是一个边长平行于坐标轴的方块形空间区域，所以只需要定义上界 `sup` 和下界 `inf` 即可。

```
geo = hp.geometry.Box(sup=np.array([1., 1., 1.]), inf=np.array([0., 0., 0.]), dim=3)
```

TIPS: 更多形状、参数详细定义请参考 hinps.geometry

接下来定义时间维度。如果你的问题是诸如对稳态的求解，与时间无关，那么可以跳过这一步。我们将时间维度的信息存在 `Dataset` 数据结构中，所以你首先需要用新建的几何区域 `geo` 来构造一个 `Dataset`。然后再用 `add_time.range` 声明你需要加上一个时间维度，传入你关心的时间上界和下界。

```
train_dataset = hp.data.DecomposedTrainDataset(geometry=geo)
train_dataset.add_time_range(time_inf=0, time_sup=1)
train_dataset.decompose_init(time_dims=1)
```

当前我们默认所有的训练任务都需要区域分解，所以我们需要使用 `DecomposedTrainDataset`，并且在声明时间维度后需要现式地初始化区域分解信息 `decompose_init()`

1.2.2 定义问题

接下来我们定义问题。问题分为两部分，方程本身与边界条件。方程本身可以使用我们内置的 `pde` 模块。当然我们也支持自定义 PDE，请参考 hinps.pde

```
heat = hp.pde.Heat(a=1, b=None, dim=3, device=args.device)
```

另外，我们使用SymPy作为我们符号运算的工具，经过 `hinps.expression.Function` 的封装，可以变为可调用的函数。在这里你可以根据你习惯的数学表达式写法描述你的函数。接下来得到的 `ground` 则是一个输入为 4 维，输出为 1 维的函数。

```
from sympy import *
t, x1, x2, x3 = symbols('t x1 x2 x3')
u = (x1**3 - 3 * x1) * (x2**3 - 3 * x2) * (x3**3 - 3 * x3) * exp(-1 * t * (x1 + x2 +␣
↪x3))
ground = hp.expression.Function(input=[t, x1, x2, x3], output=[u])
# y = ground(x), x.shape == [...,4], y.shape==[..., 1]
```

在 HiNPS 中，我们认为 PDE 是定义在求解区域内部的一种“约束”，所以我们构造一个 `Constraint` 的对象，并且将其加入到 `train_dataset` 中以表示这个约束。它表示作用于集合空间的内部（以区别于“边界”），类型是”internal”。

”internal” 类型的约束是等式约束的形式，左端项 (lhs, left-hand side) 与我们神经网络推理的结果有关，我们给定一个 heat.lhs；右端项 (rhs, right-hand side) 是我们已知的函数（例如热源、振动源等等）有关。

我们还需要指定这个约束的权重（默认为 1），针对这个约束，我们要采样的数目 `sample_size`。为了方便调试，我们可以给这个约束一个名字，例如’Internal’。

```
train_dataset.add_constraint(
    hp.data.Constraint(
        type='internal',
        on_boundary=False,
        lhs_handle=heat.lhs,
        rhs_handle=rhs, # hp.expression.Function
        weight=1.,
    ),
    sample_size=2048,
    name='Internal')
```

我们再定义边界条件。类型为”dirichlet”，也是一种等式约束，但是等式的左边为待求的函数，右边才是我们已知的函数。

我们可以用 filter 参数来指定该约束作用于边界的哪些部分，如果不指定，则默认该约束作用域整个边界。

```
train_dataset.add_constraint(
    hp.data.Constraint(
        on_boundary=True,
        type='dirichlet',
        filter=None,  #hp.expression.Function(input=[x1, x2, x3], output=[Abs(x1 -␣
↪inf[1]) < 1e-8]),
        rhs_handle=r_b, # hp.expression.Function
        weight=1.),
    sample_size=256,
    name='Boundary',
```

接下来我们定义初始条件。关于时间的约束我们需要指定作用的时间范围，对于初始条件，我们让这个范围的上界和下界都等于 TIME_INF（初始时刻）就可以了。该参数功能类似边界条件的 `filter`。

```
train_dataset.add_constraint(
    hp.data.Constraint(
        type='dirichlet',
        time_span=(TIME_INF, TIME_INF),
        on_boundary=False,
        rhs_handle=r_i, # hp.expression.Function
        weight=1.,
    ),
    sample_size=256,
    name='Initial Temperature')
```

为了验证我们创建 `val_datset`

```
val_dataset = hp.data.DecomposedValDataset(
    train_dataset=train_dataset, geometry=geo, size=200, handle=ground, device=args.
↪device)
```

1.2.3 定义神经网络和求解器

构建网络和优化器。

```
network = hp.model.Net(
    hidden_size=args.hidden_size,
    input_sup=torch.tensor([[train_dataset.time_sup, 1., 1., 1.]],
                           device=args.device),
    input_inf=torch.tensor([[train_dataset.time_inf, 0, 0, 0.]],
                           device=args.device),
    block_num=args.block_num,
    input_dim=4,
    output_dim=1,
    device=args.device)

optimizer = torch.optim.LBFGS(
    network.parameters(),
    lr=1,
    max_iter=100,
    tolerance_grad=1e-16,
    tolerance_change=1e-16,
    line_search_fn='strong_wolfe')
```

创建求解器，调用`.solve()`即可开始 PINN 的流程。

```
solver = hp.train.TDPINNSolver(
    train_dataset=train_dataset,
    val_dataset=val_dataset,
    geometry=geo,
    network=network,
    optimizer=optimizer,
    outer_max_iter=20,
    inner_max_iter=40,
    loss_p=2)

solver.solve()
```

CHAPTER 2

几何区域

偏微分方程都是定义在一定空间区域上，本模块的功能是表示和管理空间区域。边界条件，也就是边界上的约束，也是偏微分方程的一个重要部分，所以我们还需要维护空间区域的边界。

整体来看，本模块的实现逻辑是：

- 描述几何图形的核心是 SDF, Signed Distance Function。

SDF 的定义

对于一个封闭几何区域 $\Omega \in \mathcal{R}^d$, 我们构造一个函数 $F : \mathcal{R}^d \to \mathcal{R}$。满足：

- 如果 $F(\mathbf{x}) > 0$，则 $\mathbf{x} \in \Omega / \partial\Omega$，$\mathbf{x}$ 在 Ω 内
- 如果 $F(\mathbf{x}) < 0$，则 $\mathbf{x} \in \overline{\Omega}$，$\mathbf{x}$ 在 Ω 外
- 如果 $F(\mathbf{x}) = 0$，则 $\mathbf{x} \in \partial\Omega$，$\mathbf{x}$ 在 Ω 边界上。

我们称 F 是 Ω 的一个 SDF。

```
- 例如，对于球来说，SDF可以定义为其到球心距离减去半径；
```

- `Geometry` 需要维护采样空间 (sample space)，提供给 `Sampler` 用于采样。
- 每个几何图形都维护了边界 `Boundary` 的信息。边界本质上是低一个维度的几何图形，需要维护其法向信息，其他大体逻辑与 `Geometry` 相似。

2.1 Geometry 几何区域

我们以 d 维的 Box 为例子，介绍我们如何表示一个有解析表达式的几何区域。

2.1.1 定义形状

形状用 SDF 来表示。sdf 方法接受的输入是采样点的数据 Tensor，大小必须为 [⋯,d]，即最后一维长度为 d。返回的数据大小为 [⋯,1]。

我们需要把这个几何区域的解析表达式写成一个函数句柄，保存为 self._sdf，从而给 sdf 方法来调用。对于 Box 来说，需要的参数就是每个维度可取值的上界 sup 和下界 inf。

注意

考虑到浮点数精度，我们在代码中不使用 SDF(x)==0 这一判断，而是使用 abs(SDF(x))< EPS。

2.1.2 定义边界 Boundary

我们需要创建边界对象：给出边界需要的参数，具体见 Boundary。然后保存到 self._bd 中。

这一部分的目的是采样的时候我们可以遍历每个边界，并在其上进行采样。

为了采样合理，我们用 MC 方法估算了每个边界的“面积”，用于分配在不同边界上面的采样数目。

2.1.3 原空间与采样空间的映射

当原空间（origin_space，定义 PDE 的空间）和采样空间不一致时，Geometry 还需要维护这两空间之间的映射。例如：对于平面直角坐标系中的圆，其空间 $\Omega = \{x | \|x\|_2 \leq r\}$；我们可以将其转换为极坐标，定义其采样空间为 $[0, r] \times [0, 2\pi]$

Box 的可以不需要这样的映射，所以

- to_sample_space 是直接返回上界和下界
- to_origin_space 是一个恒等映射

filter_sample 这一方法的目的是，我们允许原空间到采样空间的映射不是一一对应的，为了简化代码的实现

- 例如：我们可以把一个半径为 r 的圆映射到 $[-r, r] \times [-r, r]$，如果我们在采样空间进行均匀采样后，得到的点只有 $\frac{\pi}{4}$ 会落在原来的圆内。

我们在 filter_sample 中调用 sdf 可以将不在圆内的点筛掉。如果输入是一个尺寸为 [⋯,N,d] 的数据 Tensor，返回则是一个尺寸为 [⋯,M,d] 的 Tensor，其中 M ≤ N。

提示

在采样空间的分布与原空间的分布不一定是一样的，以上述圆为例子，在采样空间的均匀采样并不是在原空间的均匀采样。

相关加权的采样逻辑需要在 `Sampler` 中实现。

2.2 Boundary 边界

我们定义边界形状与 `Geometry` 略有不同，另外需要定义边界法向。

2.2.1 定义形状

我们要求边界都是可参数化的，需要给出参数范围和参数的映射函数（只需用 `SymPy` 给出数学表达式，使用 `hinps.expression.Function` 转换为可调用的 Python 函数）。

- 如果是一个不与坐标轴平行的正方形 ABCD，我们可以用 $(a, b) \in [0, 1] \times [0, 1]$ 作为参数，通过映射 $g(a, b) = a * \overrightarrow{AB} + b * \overrightarrow{AD}$ 得到这个正方形内的点。

2.2.2 原空间与采样空间的映射

采样空间就是该边界的参数空间。

2.2.3 定义法向

我们需要在定义边界的时候给出法向的函数，他是一个可调用的 Python 函数。输入为 [⋯,d]，输出也为 [⋯,d]，而且输出的每一个法向应该是单位向量。

注意

我们维护的是**外法向**。

Coming up⋯

```
STL文件定义区域
区域分解
```

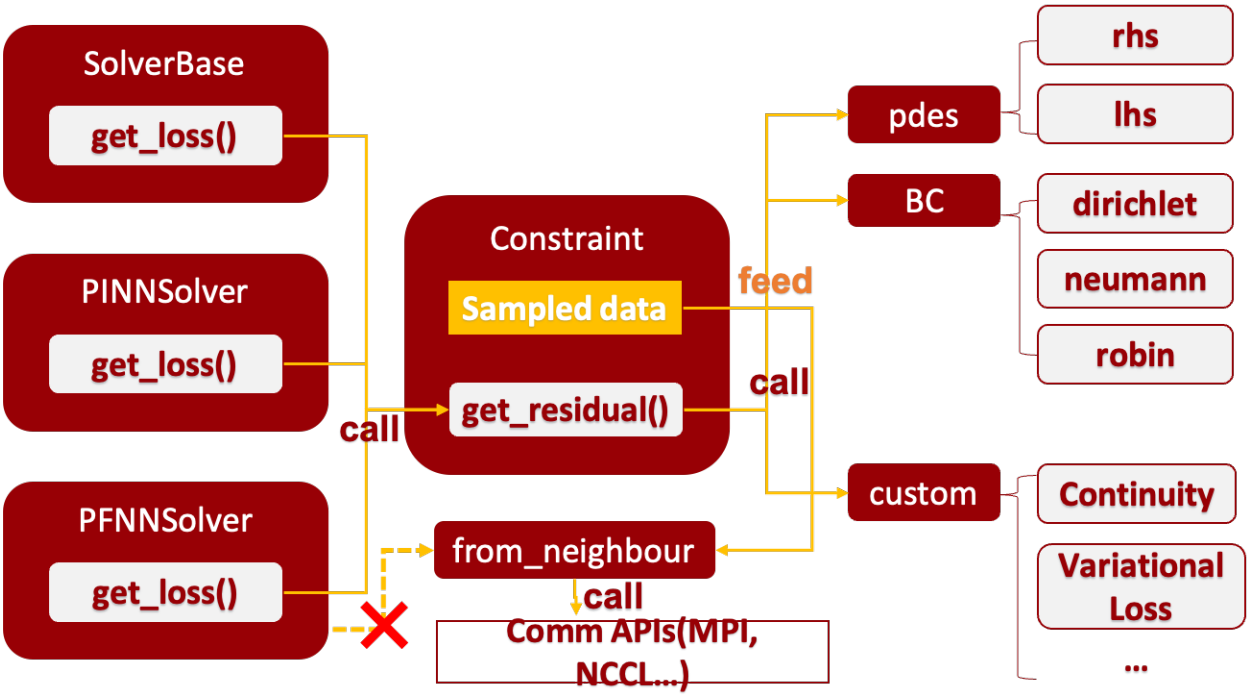
CHAPTER 3

数据与采样

本模块负责与数据的获得、存储和使用的部分。在 HiNPS 中，我们认为：

- 每一个约束都应该以一个 `Constraint` 对象来表示
- 所有的约束都保存在一个 `Dataset` 对象中在训练过程中
- `Dataset` 对象中会根据需要生成 `Sampler` 并调用采样逻辑获得数据
- 约束会通过某种**计算**得到一个值作为损失函数的一部分，所以为了进行这个计算 `Constraint` 对象存储了采样得到的数据，并且定义了这个计算。

3.1 Constraint 约束



这一部分内容可以见源代码 `data/constraint.py` 中。`Constraint` 提供了几种基本的约束形式，这几种基本约束形式我们都假设表示为等式约束，需要给出等式左边和右边的函数句柄，然后该约束分别计算两边的结果然后作差作为残差 (residual)。

该对象存储了具体采样得到的数据。

- `x`: (S,N,D)，采样得到的数据，S 为子区域数目，如果无区域分解则默认为 1；N 为采样数目；D 为输入数据的维度。
- `x_normal`: 这一项是在输入数据点计算得到的法向单位向量。
- `value`: 是在 `x` 上通过等式右边函数句柄进行计算得到的函数值。

3.1.1 约束表达式

为了表示该约束的计算，主要参数如下：

- `type`:str, 表示该约束的类型，[‘dirichlet’，‘neumann’，‘internal’，‘from_neighbour’，‘custom’]
  - `internal`: 指的是施加在求解区域内部的约束，一般为 PDE
  - `from_neighbour`: 指的是在区域分解过程中需要从其他相邻区域获得的边界信息（该类型的约束不需要用户指定）
  - `custom`: 自定义的约束类型，用户需要自己实现计算函数

- rhs_handle: expression.Function, 右侧函数的函数句柄
- lhs_handle: expression.Function, 左侧函数的函数句柄
  - 若为 None，则认为只需要直接用神经网络推理得到 pred

具体的计算发生在 get_residual 中。目前我们默认需要由调用者传入一个 network 用来推理。

狄利克雷（第一类）边界条件

在 get_residual 中进行正向推理计算，然后与 value（等式约束的右侧函数值）作差。

备注： value 是在进行采样的时候就算出来的，所以不需要在此处进行计算。

纽曼（第二类）边界条件

需要在构建 Constraint 的时候传入参数：

- grad_dims: List, 指的是需要计算梯度的维度序号，例如：
  - [1,2,3] 指的是在 3D 含时问题时对空间维度求法向
  - [0] 指的是对时间维度求“法向”，即对时间求微分。

然后再 get_residual 中我们会计算正向并且求导得到在采样点的梯度在法向上的投影。

$$\frac{\partial u}{\partial \mathbf{n}} := \nabla_x u(x;\theta) \cdot \mathbf{n}_\mathbf{x}$$

然后与等式右侧的函数值 value 作差。

PDE 约束（internal）

PDE 的左侧是比较复杂的含微分的表达式。我们需要创建约束 Constraint 时明确给出左侧函数的表达式。见*pde 模块*。

它的接口要求是接受 u 作为待解的函数值（神经网络推理的结果），x 作为采样点，然后计算返回偏微分方程的左侧值。

如果 x 的维度是 [S,N,InputDim]，u 的维度应当为 [S,N,OutputDim]，等式的右侧函数句柄应当返回一个 [S,N,OutputDim]，而该左侧的函数句柄 lhs_handle 应当返回一个 [S,N,OutputDim]（与右侧函数句柄范围的数据规模一样）

3.1.2 作用范围

表示该约束在时空上的作用范围，我们有以下主要参数：

- `on_boundary`: bool, 默认为 True. 用于指定当前约束是否是施加在边界上的
- `filter`: expression.Function, 指定当前约束使用的空间范围，
- `time_span`: tuple(2)，指定当前约束使用的时间范围，如果为 None 则默认为全过程，如果两个值相等则认为是一个时刻（例如初始时刻）

3.1.3 自定义你自己的约束

备注： 撰写中…

3.2 Dataset 数据集

3.3 Sampler 采样器

目前采样器支持的种类有只有 GridSample, UniformSample 两种。我们需要给出参数有：

- `inf`,`sup`: 采样范围：我们当前支持在方块区域中采样，只需要给出每个维度的上界 (sup) 和下界 (inf)，例如：
  - `inf=List([0,0,0,0]), sup=List([10,2,2,2])`
- `size`: 采样数据量
- `device`: 采样数据存放的位置, [`cpu`,`cuda:i`,…]

构建完采样器后调用`.sample()`方法可以进行采样。

3.3.1 自定义采样逻辑

我们支持自定义重要性采样、自适应采样等采样策略。实现应当保证接口一致性，在 `__init__` 中初始化该采样方法需要的数据；`.sample()`方法执行采样逻辑。

备注： 撰写中…

Coming up…

区域分解

CHAPTER 4

方程与约束

为了表示方程和各种各样的约束，我们在 HiNPS 中实现了 `Constraint` 类。

4.1 偏微分方程

Coming up…

偏微分方程
交换边界信息

CHAPTER 5

并行与分布式

Coming up…

硬件映射，通信组织方式

CHAPTER 6

热传导

CHAPTER 7

弹簧振子系统

CHAPTER 8

HiNPS API 文档

在HiNPS 的仓库中我们将算例均存与demo中，在其中根据不同文件夹的名字可以找到不同的算例而 HiNPS 的核心代码均存在hinps目录中。

```
./hinps
├── const.py
├── data
├── distributed
├── expression
├── geometry
├── __init__.py
├── kernels
├── model
├── pdes
├── train
├── unitest
└── utils
```

8.1 约束

8.2 geometry

8.3 数据与采样

8.4 其他

8.4.1 日志 `logging`

在 hinps 中我们采用 logging 包封装了输出信息，并且默认输出模式为 logging。输出由低到高分为如下表。级别低于输出模式的日志都会被输出。层级越高的日志输出应当越少。

在 hinps 中为了适应分布式的情况，我们默认每次日志打印都由 Rank 0，也就是主进程来打印。如果需要检查每个进程的情况，需要给出参数 `force=True`。

8.4.2 符号表达式与计算

8.4.3 计时