米兰理工大学 (POLITECNICO MILANO 1863)

工业与信息工程学院 (SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE)

不确定性量化下的物理信息深度学习综合库

项目报告

科学计算高级编程

作者:
Giulia Mescolini
Luca Sosta

教授:
Prof. Luca Formaggia

导师:
Prof. Andrea Manzoni
Prof. Stefano Pagani

学年:
2021-2022

摘要

摘要：在这个项目中，我们开发了一个新的库，实现了贝叶斯物理信息神经网络（B-PINNs）方法的多种变体，这是一个利用神经网络解决物理问题的框架，能够考虑来自偏微分方程的信息并量化预测的不确定性。

为了实施这一策略，我们从头开始设计了一个模块化且灵活的管道；事实上，在我们的设计中，一方面我们希望将 B-PINN 框架中涉及的网络任务的特性反映到类的层次结构中，另一方面我们旨在提出一种结构，能够在其中插入针对不同任务的各种功能。

事实上，在发布的版本中，我们提供了整个流程各个方面的不同变体，从数据管理到训练算法，其中我们实现了 Adam、哈密顿蒙特卡洛 (Hamiltonian Monte Carlo)、变分推断 (Variational Inference) 和斯坦变分梯度下降 (Stein Variational Gradient Descent)。

该库的设计也预留了进一步扩展的空间，因为代表脱离主骨架的不同组件的模块被构想为由少量特定案例的行为定义，并且对功能没有严格的限制，这一点在实现具有不同例程和结构或不同问题的算法时得到了验证。

最后，我们提出了一个结果展示，以突显该方法和库的主要特征：算法比较、在神经网络中引入 PDE 残差、高维度的可移植性以及通过微调模型参数可以获得的结果质量。

关键词： B-PINNs，不确定性量化，科学学习

目录

1 简介 (Introduction)

1.1 项目概述 (Project Overview)

机器学习技术在科学计算中的整合如今变得越来越流行。一个具有挑战性的场景是不确定性量化，神经网络的高效性有助于克服基于 PDE 求解器的传统方法的计算成本。

贝叶斯物理信息神经网络（B-PINNs）代表了一种解决此类问题的方法，这不仅归功于在学习过程中包含了微分方程的残差，还归功于能够重建网络输出分布而非单一预测的训练算法。

该方法最近在 [11] 和 [7] 等几篇论文中提出，是物理信息神经网络（PINNs）的演变，PINNs 是一种用于解决涉及偏微分方程问题的深度学习框架（[9], [10]）；B-PINNs 从贝叶斯统计理论中汲取灵感，增强了 PINNs 方法并使得引入不确定性量化成为可能。

在这个项目中，我们实现了一个用于贝叶斯物理信息机器学习的库，旨在实现模块化、灵活性并支持插件。在这个利基应用的最新技术中，通常只能找到应用于特定微分问题的特定贝叶斯训练方法的实现；因此，我们设定的目标是开发一个具有广泛功能的库。

考虑到项目目标，库中某些组件的设计起到了至关重要的作用。方法的主干被认为是一个刚性的模块化结构，构建方式使得所需应用程序的模块可以组装到其中而不影响骨架。每个模块都可以定义其自身的特征，这些特征数量很少且不对功能引入严格限制，因此很容易为我们感兴趣的不同方法或问题生成新模块。

库的特性随后通过一系列应用案例进行了强调，旨在分别展示主要的支柱：四种在结构和直觉上截然不同的训练算法的设置、将物理信息引入模型的贡献以及库向更高维度的可移植性。

对于具体应用中的方法性能，参数的微调起着至关重要的作用。尽管在项目中我们更关注库功能的横向扩展，但对于具有可持续计算时间和参数选择上具有有趣挑战的一种特定贝叶斯训练算法，我们强调了性能质量，开发了更多种类的测试用例并进行了精细的方法选项调整。

1.2 报告结构 (Report Structure)

本报告介绍了拟议实现的理论基础，在第 2 章中解释了 B-PINN 的构建模块。从第 2.1 节描述的神经网络基础和确定性训练算法开始，我们在第 2.2 节说明了物理信息机器学习背后的理论。然后，我们在第 2.3 节介绍了贝叶斯神经网络的学习机制，并在第 2.4 节说明了如何将它们与 PINNs 结合以获得 B-PINNs。

在第 3 章中，我们深入探讨了实现的非确定性训练算法。我们说明了它们背后的理论背景和程序，提出了问题并讨论了参数的选择，但没有深入技术实现细节（推迟到第 5 章）。每个小节专门介绍一种算法：第 3.1 节介绍哈密顿蒙特卡洛 (Hamiltonian Monte

Carlo)，第 3.2 节介绍变分推断 (Variational Inference)，第 3.3 节介绍斯坦变分梯度下降 (Stein Variational Gradient Descent)。

在第 4 章中，我们首先介绍并阐述了在环境（第 4.1 节）和库（第 4.2 节）方面的编码选择。然后，我们在第 4.3 节致力于介绍 Python 中的面向对象编程，重点关注项目中实现的特性。最后，第 4.4 节介绍了仓库的结构以及配置、数据和输出文件的内容。

在第 5 章中，我们说明了源代码：我们描述了三个可执行脚本，并专门用特定章节介绍了各个模块，这些模块的任务与 B-PINNs 方法的实现或后处理实用程序（如 UQ 和性能评估）相关。

在第 6 章中，我们展示了实现的库在一系列测试用例中的应用，这些测试用例具有不同的数据集和训练配置。应用展示的组织方式旨在分别突显库的主要特征：在第 6.1 节中，我们专注于同一任务上的训练算法比较。然后，在第 6.2 节中，我们展示了在阻尼谐振子问题上包含物理信息的力量。在第 6.3 节中，我们展示了一个 2D 维度的例子，最后在第 6.4 节中，我们展示了通过对使用 HMC 训练的模型进行大规模微调而在多个应用上获得的结果。

最后，在第 7 章中，我们对本项目进行了总结并提出了进一步发展的建议。

2 方法 (Methods)

2.1 神经网络概述 (An overview on Neural Networks)

2.1.1 核心结构 (Core Structure)

术语人工神经网络 (Artificial Neural Network, NN) 指的是一种受构成动物大脑的生物神经网络启发的计算系统。这些工具于 20 世纪下半叶首次引入，如今正见证着惊人的普及。

它们属于深度学习 (Deep Learning, DL) 的框架，而深度学习又是机器学习 (Machine Learning, ML) 的一个分支，它们能够以令人惊讶的完整方式处理信息。神经网络的主要优势在于，虽然经典的机器学习预测（例如线性回归）仅在特征（即输入或其手工组合）良好时才有效，但神经网络可以直接从数据中学习特征，因此属于特征学习 (Feature Learning)[1]的范畴。

神经元 (The neuron) 为了理解这些工具是如何工作的，我们首先介绍它们的基本组件：**神经元 (neurons)**。它们由处理单元组成，这些单元：

1. 接收来自输入数据或其他先前神经元输出的有限数量的信号 a_j 作为输入；
2. 计算它们与权重 $w_{i,j}$ 的加权和，其中 j 指代信号，i 标识神经元；
3. 通过添加偏置项 b_i 来减去一个阈值，这是一个相对于神经元本身的特殊权重；
4. 产生一个单一输出 z_i 并应用非线性激活函数 ϕ 获得 a_i。

激活函数 ϕ 必须是非线性的，这一点至关重要；否则，整个神经网络将仅仅是输入数据的高度因式分解的线性函数。

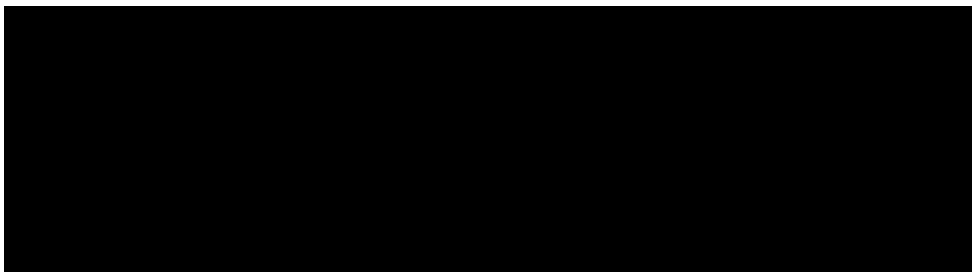

图 1: 神经元，神经网络的基本组件。

$$\mathbf{a}_1 = \sigma\left(\mathbf{W}_0^T \mathbf{a}_0 + \mathbf{b}_0\right), \quad \mathbf{W}_0 \in \mathbb{R}^{n \times m} \tag{1}$$

[1]在机器学习中，特征学习或表示学习是一组允许系统自动发现特征检测或分类所需的表示的技术。这取代了手动特征工程，并允许机器既学习特征又使用它们来执行特定任务。

全连接神经网络 (Fully Connected Neural Networks) 神经元堆叠在层中，而在全连接神经网络 (FCNN) 中（如本项目所涉及的那样），每个神经元都连接到下一层和上一层的所有神经元。现在，让我们定义神经网络 (NN) 的结构，这有助于理解已经描述过的从数据中学习特征的过程。如图 2 所示，三个主要部分是：

- **输入层 (Input Layer)**：接收 N_i 个输入。输入数据可以是例如域、图像、时间序列等；
- **隐藏层 (Hidden Layers)**：$L-1$ 层，每层有 N_h 个节点。它们的任务是找到合适的特征；
- **输出层 (Output Layer)**：根据所需的任务（例如回归、分类等），返回 N_o 个输出。

使用此符号，l 层中神经元 i 的输出仅取决于前几层的激活值 $\$a_j^{(l-1)}{}_{j=1}^{m}$，即：

$$a_i^{(l)} = \phi\left(\sum_{j=1}^{N_{l-1}} w_{j,i} a_j^{(l-1)} + b_i^{(l)}\right) \tag{2}$$

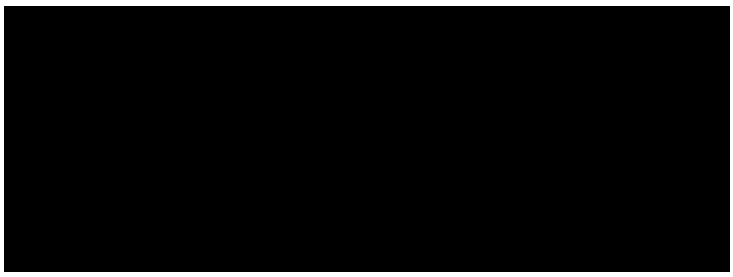

图 2: 神经网络的一般结构

激活函数 (Activation Functions) 在本节中，我们介绍激活函数 ϕ 的一些常见选择：

- **Sigmoid**：此函数处处平滑，但有一个弱点：当 $|x| \gg 1$ 时 $|\sigma'(x)| \ll 1$，因此梯度会消失 *(vanishing)*，对于深层神经网络学习可能很慢。
- **双曲正切 (Hyperbolic Tangent)**：Sigmoid 的平衡版本，它是可微的并且具有类似的 S 形。它的优点是将输入值推向 1 和 -1 而不是 1 和 0。
- **修正线性单元 (Rectified Linear Unit, ReLU)**：在这种情况下，对于 $x > 0$，梯度不会消失，因为导数为 1，但在 $x = 0$ 处函数不可微，且对于 $x < 0$ 导数为 0。
- **Leaky ReLU**：ReLU 的这个稍微修改的版本解决了 $x < 0$ 时 0 梯度的问题。

- **Swish**：ReLU 的另一种变体，根据实验，在许多具有挑战性的数据集上，它在更深的模型上往往比 ReLU 效果更好。

激活函数	解析表达式	激活函数的导数
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1-\sigma(x))$
Hyperbolic Tangent	$\tanh(x) = \frac{e^x-e^{-x}}{e^x+e^{-x}} = 2\sigma(2x) - 1$	$\tanh'(x) = 1 - \tanh^2(x)$
ReLU	$\mathrm{ReLU}(x) = \max\{0, x\}$	$\mathrm{ReLU}'(x) = \mathbb{1}_{\mathbb{R}_+}(x)$
Leaky ReLU	$\mathrm{LR}(x) = \max\{\alpha x, x\}$	$\mathrm{RL}'(x) = \alpha + (1-\alpha)\mathbb{1}_{\mathbb{R}_+}(x)$
Swish	$S(x) = x\sigma(x)$	$S'(x) = S(x) + \sigma(x)(1 - S(x))$

表 1: 神经网络的常见激活函数及其导数

我们神经网络的结构 复现 [11] 中提出的神经网络架构，在本项目中，我们构建了一个前馈全连接神经网络，包含 2 个具有相同节点数（通常为 16 或 50）的隐藏层，如图 3 所示。隐藏层选择的激活函数可以由用户指定；在提出的测试用例中，我们选择了 *swish* 函数。

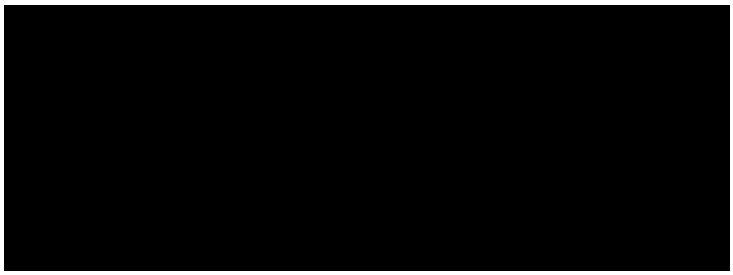

图 3: 具有 N_i 个输入、3 个隐藏层和 N_o 个输出的神经网络架构。

2.1.2 反向传播 (Backpropagation)

在机器学习中，神经网络学习正确权重的算法通常被称为反向传播 *(backpropagation)*。首先，让我们定义损失函数 $\mathcal{L}(f)$，它通过表达模型估计的输出 $\mathbf{y}^* = f(\mathbf{x})$ 与其所有数据的真实值 $\mathbf{y}$ 之间的距离，提供了神经网络性能的定量度量。最优解是通过最小化损失函数获得的，相对于所有权重和偏置：

$$w_{i,j}^{*,(l)}, b_i^{*,(l)} = \underset{w_{i,j}^{(l)}, b_i^{(l)}}{\operatorname{argmin}} \mathcal{L}(f) \tag{3}$$

为了执行此任务，我们首先必须选择优化器 *(optimizer)*，即优化算法。存在多种可能性，它们具有不同的内存消耗和达到收敛所需的时间。无论我们选择哪种算法，都需要计算

$\nabla\mathcal{L}$，特别是：

$$\frac{\partial\mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \left(\frac{\partial\mathcal{L}_n}{\partial\mathbf{W}_l}\right)_{i,j} \quad \frac{\partial\mathcal{L}_n}{\partial b_i^{(l)}} = \left(\frac{\partial\mathcal{L}_n}{\partial\mathbf{b}_l}\right)_i \quad \forall(i,j,l) \tag{4}$$

这意味着必须计算的导数数量级为 $\mathcal{O}(K^2L)$（假设为了简单起见，我们在 L 层中的每一层都有 K 个节点）；因此，找到一种有效的方式来联合计算导数变得至关重要。

通用符号 为了说明反向传播算法，让我们引入以下符号：

- **权重矩阵**：$\mathbf{W}_l$，其中 $\mathbf{W}_1 \in \mathbb{R}^{N_i \times N_h}$，$\mathbf{W}_l \in \mathbb{R}^{N_h \times N_h}\ \forall 2 \leq l \leq L-1$，$\mathbf{W}_L \in \mathbb{R}^{N_h \times N_o}$。
- **偏置向量**：$\mathbf{b}_l$，其中 $\mathbf{b}_l \in \mathbb{R}^{N_h}\ \forall 1 \leq l \leq L$，$\mathbf{b}_L \in \mathbb{R}^{N_o}$。
- **可训练参数**：$\boldsymbol{\theta}$ 通常用于表示所有层的权重和偏置的集合。

使用此符号，

$$\begin{aligned}
\mathbf{a}_1 &= f^{(1)}(\mathbf{a}_0) = \Phi(\mathbf{z}_0) = \Phi(\mathbf{W}_1^T\mathbf{a}_0 + \mathbf{b}_1) \\
\mathbf{a}_2 &= f^{(2)}(\mathbf{a}_1) = \Phi(\mathbf{z}_1) = \Phi(\mathbf{W}_2^T\mathbf{a}_1 + \mathbf{b}_2) \\
&\vdots \\
\mathbf{a}_L &= f^{(L)}(\mathbf{a}_{L-1}) = \Phi(\mathbf{z}_{L-1}) = \Phi(\mathbf{W}_L^T\mathbf{a}_{L-1} + \mathbf{b}_L)
\end{aligned} \tag{5}$$

因此，以更紧凑的形式书写，神经网络的输出为：

$$\mathbf{y}^* = f(\mathbf{x}) \text{ 其中} f = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)} \text{ 且} \mathbf{x} = \mathbf{a}_0,\ \mathbf{y}^* = \mathbf{a}_L \tag{6}$$

损失函数 (Loss Functions) 本工作中展示的所有测试用例均属于回归框架，因此我们的数据集通常由 N 对输入 $\$\ \mathrm{x}\ _{n=1}^{N}$ 和目标 $\$\ \mathrm{y}\ _{n=1}^{N}$ 向量序列组成。回归问题中最常用的损失函数是*均方误差 (Mean Squared Error, MSE)*，定义为：

$$\mathrm{MSE} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{y}_n - \mathbf{y}_n^*)^2$$

使用 MSE 作为*损失函数*并利用我们构建的模型（写在方程 4 中）对给定数据集进行预测，我们得到以下方程：

$$\mathcal{L}_{MSE} = \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}_n \text{ 其中} \mathcal{L}_n = (\mathbf{y}_n - f(\mathbf{x}_n))^2 = (\mathbf{y}_n - f^{(L)} \circ \cdots \circ f^{(1)}(\mathbf{x}_n))^2 \tag{7}$$

值得注意的是，损失可以看作是所有可训练参数 $\boldsymbol{\theta}$ 和输入 $\mathbf{x}$ 的函数，或者作为神经网络输出 $\mathbf{y}^*$ 的函数；后一种解释在处理误差的反向传播时将非常有用，因为我们可以划分来自模型本身和损失的贡献。在这两种情况下，目标 $\mathbf{y}$ 都被涉及，因为我们处于监督学习[2]的背景下，但它们是固定值。

[2]监督学习 (Supervised Learning, SL) 是一种机器学习范式，用于处理可用数据由标记示例组成的问题，这意味着每个数据点都包含特征（协变量）和相关联的标签。监督学习算法的目标是基于示例输入-输出对，学习一个将特征向量（输入）映射到标签（输出）的函数。

链式导数 (Chain Derivatives) 如上所述，我们感兴趣的量是方程 3 中的偏导数；我们将使用复合分数的导数计算的链式法则，并将每一层 l 的输出 $\mathbf{z}_l$ 和激活 $\mathbf{a}_l$ 定义为辅助量：

$$\mathbf{z}_l := \mathbf{W}_l^T \mathbf{a}_{l-1} + \mathbf{b}_l = \mathbf{W}_l^T \Phi(\mathbf{z}_{l-1}) + \mathbf{b}_l \tag{8}$$

按分量写入导数，我们得到：

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{N_l} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} a_i^{(l-1)} = \delta_j^{(l)} a_i^{(l-1)} \tag{9}$$

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \sum_{k=1}^{N_l} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)} \tag{10}$$

$\delta_j^{(l)}$ 表示由于层 l 的神经元 j 引起的误差，可以很容易地按以下方式计算：

2.1.3 优化器 (Optimizers)

对于一个模型来说，学习意味着解决寻找最小化方程 2 中所述损失函数的参数值的优化问题。由于神经网络通常包含数千个参数，确切的解无法以封闭形式获得，因此我们继续使用在 N 次迭代（也称为 *epochs*）中逐步减少损失函数的算法。优化器的选择既影响模型的准确性，也影响模型的学习速度。我们现在介绍一些用于经典神经网络的常见可用选择，这将有助于理解本项目后面介绍的用于贝叶斯神经网络的更复杂的优化器。

梯度下降 (Gradient Descent) 我们要参考的每一个优化算法（称为优化器），都使用简单的更新规则 $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \delta\boldsymbol{\theta}^{(t)}$ 更新 NN 的所有可训练参数，因此优化器之间的区别在于项 $\delta\boldsymbol{\theta}$。*梯度下降 (GD)* 算法源于最简单的想法：函数的梯度指向函数最大增加的方向，因此，为了最小化函数，我们在相反方向上迈出一步。所以，$\delta\boldsymbol{\theta} = -\gamma\mathcal{L}(\boldsymbol{heta})$，其中 γ 称为*学习率 (Learning Rate, LR)*，是 NN 的一个超参数[3]。

[H] 梯度下降 (GD) **Input:** 学习率 $\gamma > 0$，迭代次数 N_{epochs}，初始状态 $\boldsymbol{\theta}^{(0)}$

for $t = 1, \ldots, N_{epochs}$ **do**

    计算完整损失函数的梯度 $\nabla\mathcal{L}(\boldsymbol{heta}^{(t-1)})$; 更新可训练参数：$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t)} - \gamma\nabla\mathcal{L}(\boldsymbol{heta}^{(t-1)})$;

end

[H] 随机梯度下降 (SGD) **Input:** 学习率 $\gamma > 0$，迭代次数 N_{epochs}，初始状态 $\boldsymbol{\theta}^{(0)}$

for $t = 1, \ldots, N_{epochs}$ **do**

    均匀采样 $n \in \{1, 2, \ldots, N_{samples}\}$

    计算第 n 个样本上的损失梯度 $\nabla\mathcal{L}_n(\boldsymbol{\theta}^{(t-1)})$; 更新可训练参数：$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t)} - \gamma\nabla\mathcal{L}_n(\boldsymbol{\theta}^{(t-1)})$;

[3]关于参数和超参数的更详细解释将在第 2.1.4 小节中给出

end

[H] 自适应矩估计 (Adam) **Input:** $\alpha, \beta_1, \beta_2, \epsilon$，迭代次数 N_{epochs}，初始状态 $\boldsymbol{\theta}^{(0)}$，动量 $\mathbf{m}^{(0)}, \mathbf{v}^{(0)}$

for $t = 1, \ldots, N_{epochs}$ **do**

$mathbfm^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1 - \beta_1)$
$nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)})$;
$mathbfv^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1 - \beta_2)($
$nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)}))^2$ $\hat{\mathbf{m}} = (1 - \beta_1)^{-1}\mathbf{m}^{(t)}$, $\hat{\mathbf{v}} = (1 - \beta_2)^{-1}\mathbf{v}^{(t)}$; $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \alpha \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}} + \epsilon}$;

end

我们在算法 3 中报告了 Adam 的通用方案，使用以下符号：

- $\mathbf{m}, \mathbf{v}$ 是梯度的第一和第二矩，初始化为 $\mathbf{0}$；
- $\hat{\mathbf{m}}, \hat{\mathbf{v}}$ 是 $\mathbf{m}, \mathbf{v}$ 的无偏估计量；
- α, β_1, β_2 和 ϵ 是固定的超参数。

2.1.4 训练神经网络 (Training Neural Networks)

在本节中，我们介绍传统的管道，遵循该管道是为了提高文献中使用的大多数深度学习模型的性能。预测任务的成功取决于模型参数；事实上，人工神经网络被称为*参数模型*，因为它们在有限的一组参数内捕获有关其预测的所有信息，并且需要对其进行训练以选择最佳参数来表示我们要由模型学习的映射。首先，我们需要澄清参数之间的一些区别，因为一组参数在学习期间更新，而其他参数必须固定为一个值。因此我们区分：

- **可训练参数 (Trainable parameters)**：它们直接出现在损失评估中，通常梯度是针对这组参数获取的。其选择机制通常称为*训练 (training)*。在大多数情况下，它们是 NN 的权重和偏置的集合，但也可能有例外。例如，在处理逆问题（在专门章节中介绍）时，可能有其他称为 $\boldsymbol{\lambda}$ 的可训练参数，或者在迁移学习背景下，并非所有 NN 参数都是可训练的。
- **超参数 (Hyper parameters)**：它们是构建模型所需的所有其他参数；它们的选择称为*调优 (tuning)*，通常根据基于文献的知识手动执行或通过类似网格搜索的算法执行。一些相关的例子是所有的优化器参数（例如 GD 的学习率），epoch 的数量，NN 本身的架构……

数据处理 (Data Processing) 高质量的数据可以在模型性能中发挥关键作用。这就是为什么处理转换版本的数据可能很有用，这些数据通过初步的*预处理*获得，对 NN 最佳

域范围更友好，或者要求网络输出一个更容易重建的结果，然后需要通过一些后处理恢复为所需的结果。此外，在 ML 中，将数据集划分为训练 *(training)*、验证 *(validation)* 和测试 *(test)* 子集是一个重要的做法，因为对于模型性能的评估，使用已经见过并用于改进模型的数据是不公平的。这意味着我们不能使用整个数据集进行训练，而应该保留一部分用于以无偏的方式评估模型的优劣；这在 ML 中听起来可能是不利的，因为通常希望使用尽可能多的数据进行训练，但在我们的应用中，正如我们将在 2.2 节中陈述的那样，我们也依赖于来自偏微分方程的物理信息，而不是大量的数据，因此我们没有遇到数据短缺的问题。

学习工作流 (Learning Workflow) 在数据集的预处理之后，为了编写一个既能够准确又能泛化预测的 ML 模型来完成所需任务，应遵循一个精确的方法：

1. **训练阶段 (Training phase)**：更新 NN 的可训练参数以最小化训练数据集上的损失，在随后的 N_{epochs} 迭代期间，提高准确性 *(accuracy)*。这是优化器发挥作用的唯一阶段，它需要最繁重的计算（主要由于梯度）。

2. **验证阶段 (Validating phase)**：评估模型的通用性 *(generality)*，以确保模型没有在训练集上过拟合，即也学习了所提供数据中的噪声或特性。这使我们能够调整超参数以改进模型，同时也借助除损失之外的性能指标。

3. **测试阶段 (Testing phase)**：使用未见数据能够提供模型性能的真实无偏评估 *(evaluation)*。

逼近能力 (Approximation Power) 一个可能提出的问题可能是：神经网络能够逼近哪些函数？令人欣慰的答案是，当激活 ϕ 是类 sigmoid 时，由 Barron 的逼近结果给出。

定理 2.1. 给定一个函数$f : \mathbb{R}^D \to \mathbb{R}$, 令$\hat{f}$ 为其傅里叶变换。 (12)

如果 $\exists C > 0$ 使得 $\int_{\mathbb{R}^D} |\omega||\hat{f}(\omega)|d\omega \leq C$。那么 $\forall n \geq 1, \exists f_n, \{c_j\}_{j=1}^n \subset \mathbb{R}$，其中：$f_n(x) = \sum_{j=1}^n c_j \phi(x^T w_j + b_j) + c_0$ 使得 $\int_{|x|\leq r} |f(x) - f_n(x)|^2 dx \leq \frac{(2Cr)^2}{n}$

这等同于声明，只要使用足够数量的隐藏神经元，任何具有单个隐藏层和类 sigmoid 激活函数 ϕ 的 NN 都可以以任意小的误差（在 L^2 范数意义上）逼近紧集上的任何连续函数。Shekhtman 也获得了类似的逐点逼近误差结果；在这种情况下，激活函数是 ReLU。

2.2 物理信息神经网络 (Physics Informed Neural Networks)

在本节中，我们介绍物理信息神经网络 (PINN)，它代表了一个用于解决涉及偏微分方程问题的深度学习框架。它们在过去 5 年中被引入，在它们的发展中，Karniadakis、

Perdikaris 和 Raissi 是先驱（见 [9], [10]）。如今，这些工具甚至在学术界之外也得到了增强，如 NVIDIA Modulus 等解决方案所示。PINNs 由神经网络组成，这些神经网络经过训练以解决监督学习任务，同时遵守由常微分方程 (ODE) 或偏微分方程 (PDE) 控制的物理定律。在标准 ML 中，神经网络在“大数据”框架中很常见，当有大量观测值可用于训练我们的模型时，但在某些应用中，数据可能极其昂贵且难以评估，例如在医学领域。因此，利用我们对现象拥有的物理知识（以微分模型为代表）起着关键作用，因为它使我们即使在“小数据”体制下也能工作。

2.2.1 PINN 的结构 (Structure of a PINN)

我们现在说明物理信息神经网络的一般结构。给定一个通用的偏微分方程（线性/非线性，稳态/非稳态……），我们引入以下符号：

- Ω 是 $\mathbb{R}^n$ 中的有界域，$[0,T]$ 是时域；
- $\mathbf{x}$ 和 t 分别是空间和时间变量；$\mathbf{x} \in \Omega$, $t \in [0,T]$，用作输入；
- $\mathbf{u}(\mathbf{x},t)$ 和 $\mathbf{f}(\mathbf{x},t)$ 是解和参数场，通常可以依赖于空间和时间；
- $\mathcal{N}$ 是应用于 $\mathbf{u}$ 的完全已知的微分算子；
- $\boldsymbol{\lambda}$ 表示问题的物理参数，可能是未知的。

要解决的问题的解析表述因此为：

$$\begin{cases} \mathcal{N}(\mathbf{u}(\mathbf{x},t);\boldsymbol{\lambda}) = \mathbf{f}(\mathbf{x},t) & \text{在} \Omega \times [0,T] \\ + \text{边界条件} & \text{在} \partial\Omega \times [0,T] \\ + \text{初始条件} & \text{在} \Omega \times \{0\} \end{cases} \tag{13}$$

在图 4 中，我们展示了 PINNs 是如何工作的：经典的 NN 用于通过其前向传播对解 $\mathbf{u}$ 进行预测，然后在损失评估期间（示例中为 MSE）添加一个新的正则化项：表示为 MSE_R 的残差损失 *(residual loss)*。残差的评估不是由 NN 层进行的，而是通过对模型输出相对于输入使用自动微分[4]计算偏导数，然后在微分算子中组合来执行的。

[4]自动微分 (Automatic Differentiation, AD) 是一组评估由计算机程序指定的函数导数的技术。它利用了这样一个事实：无论多么复杂，每个计算机程序都会执行一系列算术运算和初等函数。

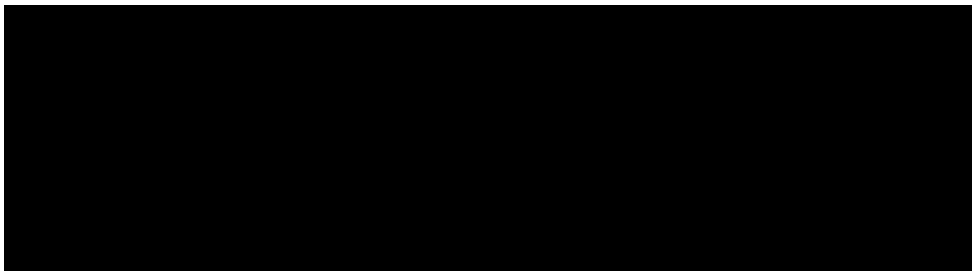

图 4: 物理信息神经网络的结构。

2.2.2 优化问题 (Optimization Problems)

在这类分析问题上引入 ML 模型允许我们将后者视为等同于解决优化问题。在我们的项目中，我们处理稳态 PDE 约束问题，例如：

$$\begin{cases} \min_{u} \mathcal{L}(u) = ||u - t||_{L^2} \\ s.t. \mathcal{N}(u; \boldsymbol{\lambda}) = f \text{ 在} \Omega \end{cases}$$

其中我们采用以下符号：

- $\mathcal{L}$ 要优化的量；
- u 最小化变量；
- t 我们想要学习的目标函数；
- f PDE 中的参数场。

离散问题 (Discrete Problem) 第一步是离散化 t 和 f 函数，因为我们将使用它们的稀疏测量。这些测量仅在某些点可用，表示为 $\mathbf{x}$，因此我们用向量 $\mathbf{t} = \mathbf{t}(\mathbf{x})$ 和 $\mathbf{f} = \mathbf{f}(\mathbf{x})$ 近似连续问题，最小化变量为 $\mathbf{u} = \mathbf{u}(\mathbf{x})$。离散化问题读作：

$$\begin{cases} \min_{u} \mathcal{L}(u) = \text{MSE}(\mathbf{u}, \mathbf{t}) \\ s.t. \mathcal{N}(\mathbf{u}; \boldsymbol{\lambda}) = \mathbf{f} \quad \forall \mathbf{x} \end{cases} \tag{14}$$

注意，L^2-范数现在被其离散对应物 MSE 替换，MSE 在观测集 $\mathbf{x}$ 上计算，扮演经验均值的角色。使用 NN 作为模型，我们可以根据参数 $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$ 重写方程 11 中的第一个方程，使用它作为最小化变量，结果为：

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{u}), \text{ 其中} \mathbf{u}(\mathbf{x}; \boldsymbol{\theta}) = \mathcal{NN}(\mathbf{x}; \boldsymbol{\theta})$$

最后，我们问题的优化部分仅依赖于 NN 的可训练参数。方程 11 的其余部分是微分约束；为了解决这个问题，我们模型的 PINN 部分发挥了作用。

微分残差 (Differential Residuals) 如果由函数 $\mathbf{u}$ 和参数场 $\mathbf{f}$ 给出的对是 PDE 的解，那么我们可以定义算子的残差为：

$$R := \mathcal{N}(\mathbf{u}; \boldsymbol{\lambda}) - \mathbf{f} \tag{15}$$

因此，我们可以通过损失函数对 $\mathbf{u}$ 和 $\mathbf{f}$ 的值施加 $R = 0$，使其通过我们的预测近似精确值，并通过在训练阶段将 $MSE(R, 0)$ 包含在 $\mathcal{L}$ 中，帮助模型学习合理接近正确函数的函数。边界和初始条件可以很容易地实现，只需考虑属于抛物线边界的域点 $\mathbf{x}$。关于损失、正则化的明确公式以及所需数据集的更多细节将在 2.2.3 小节和特定案例章节中提供。

正问题和逆问题 (Direct and Inverse Problems) 在处理 PINNs 时，我们可以考虑正 *(direct)* 或逆 *(inverse)* 问题，强调其差异很重要，因为我们需要通过引入 PINN 来处理正则化项。两者的区别基本上在于我们对某些参数 $\boldsymbol{\lambda}$（代表物理系数，如扩散系数）的了解：在正向设置中它们是已知的，而在逆向设置中它们需要从其他可用测量中估计。在正问题中，物理参数 $\boldsymbol{\lambda}$ 的值是已知的，因此可以将其忽略，就像它是微分算子 $\mathcal{N}$ 的一部分一样，引导我们得出以下公式：

$$\begin{cases} \min\limits_{\theta} \mathcal{L}(\mathbf{u}) = MSE(\mathbf{u}, \mathbf{t}) + R(\mathbf{u}) \\ \text{with } \mathbf{u}(\mathbf{x}; \boldsymbol{\theta}) = \mathcal{NN}(\mathbf{x}; \boldsymbol{\theta}) \end{cases} \tag{16}$$

在逆向设置中，模型物理参数 $\boldsymbol{\lambda}$ 是未知的，并由模型重建。在这种情况下，PINN 解决的优化问题是上述问题的推广：实际上，只需要将感兴趣的物理参数集 $\boldsymbol{\lambda}$ 包含在残差 R 中，导致：

$$\begin{cases} \min\limits_{\theta,\lambda} \mathcal{L}(\mathbf{u}; \boldsymbol{\lambda}) = MSE(\mathbf{u}, \mathbf{t}) + R(\mathbf{u}; \boldsymbol{\lambda}) \\ \text{with } \mathbf{u}(\mathbf{x}; \boldsymbol{\theta}) = \mathcal{NN}(\mathbf{x}; \boldsymbol{\theta}) \end{cases} \tag{17}$$

2.2.3 PINNs 的数据集 (Dataset for PINNs)

PINN 模型的训练和验证遵循与 2.1.4 小节中所述类似的时间表，但增加了一些特殊性，实际上计算前向传播是一项比随着残差插入所需的整个损失评估容易得多的任务。对于验证和测试，我们可以像以前一样毫无问题地依赖验证点 *(validation points)* 和测试点 *(test points)*，但我们可以在训练数据集 *(training dataset)* 中区分三个重要的子数据集：

- **拟合点 (Fitting Points)**：即域 Ω 内我们对解施加值的点（通过已知解析表达式获得，或例如从数值模拟中获得），并以经典训练集方式行事。这些点很少，因为

我们在小数据体制下工作，并且可能仅限于更容易进行 u_{ex} 实际测量的 Ω 的某些子域。这种情况下要添加的损失是：

$$\mathcal{L}_{fit} = \frac{1}{N_{fit}} \sum_{n=1}^{N_{fit}} (u(x_n, t_n) - u_{ex}(x_n, t_n))^2$$

- **配置点 (Collocation Points)**：是域 Ω 内的点，我们在这些点上通过要求最小化方程的残差来施加 PDE 约束。参考方程 12，我们的残差是：

$$\mathcal{R}(\mathbf{x}, t) = \mathcal{L}(\mathbf{u}(\mathbf{x}, t)) - \mathbf{f}(\mathbf{x}, t))$$

如使用图 4 中的 MSE 损失函数，我们的目标是最小化损失：

$$\mathcal{L}_{col} = \frac{1}{N_{col}} \sum_{n=1}^{N_{col}} \mathcal{R}(\mathbf{x}_n, t_n)^2$$

其中$(\mathbf{x}_n, t_n)_{n=1}^{N_{col}}$ 表示配置点集的元素。这些点是完全可用的，因为不需要目标函数，但它们在计算相对于输入的梯度的计算成本方面影响最大。

- **边界点 (Boundary Points)**：我们在其上施加精确值（当我们有 Dirichlet 边界条件时），或最小化描述 Neumann 条件的方程残差的点；取决于此，它们分别更类似于第一组或第二组。例如，对于 Γ_D 上的 $u = f$ 形式的 Dirichlet 边界条件，我们添加损失：

$$\mathcal{L}_{bnd} = \frac{1}{N_{Bnd}} \sum_{n=1}^{N_{Bnd}} (u(x_n, t_n) - f(x_n, t_n))^2$$

这些点来自物理方程，只涉及 f 而不涉及 u_{ex}，所以我们可以毫无顾虑地大量使用它们。初始条件也可以这样做。

损失的组合 (Combination of the Losses) 我们希望网络最小化的最终损失函数包括上述所有损失的影响，以及它们的加权组合。例如，对于 M 个损失 $\mathcal{L}_i$，我们有

$$\mathcal{L} = \sum_{i=1}^{M} q_i \mathcal{L}_i$$

2.3 贝叶斯神经网络 (Bayesian Neural Networks)

贝叶斯神经网络 (BNNs) 代表了一种在现代深度学习方法框架中引入不确定性量化 *(uncertainty quantification, UQ)* 的方法，深度学习方法构成了解决具有挑战性问题的极其强大的工具，但它们在预测的重要性上通常作为黑盒运行。

2.3.1 不确定性量化 (Uncertainty Quantification)

到目前为止，我们已经介绍了用神经网络重建函数的方法，这些方法无法提供关于预测置信度 *(confidence)* 的信息。对于 NN，我们确实无法量化预测的不确定性，因为我们从模型中只获得了解的一个样本，即对应于给定输入的网络输出，使用的是在先前介绍的方法最小化损失过程后选择的网络参数。主要思想是用合适的联合概率分布替换 $\boldsymbol{\theta} := \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$ 并改变我们执行前向传播的方式。对于验证和测试阶段，采样 N 个新的 $\boldsymbol{\theta}$ 实例，它们生成 N 个不同的独立预测，这些实际上是经典 NN 的输出，其参数每次都是集合中的成员。最后，有了输出的 N 个样本，我们获取相关的统计数据，并使用均值进行预测，使用方差进行 UQ。真正的负担发生在模型训练期间，因为必须为 BNN 拥有特设 *(ad hoc)* 算法；此讨论推迟到第 3 章，该章完全致力于描述该任务。

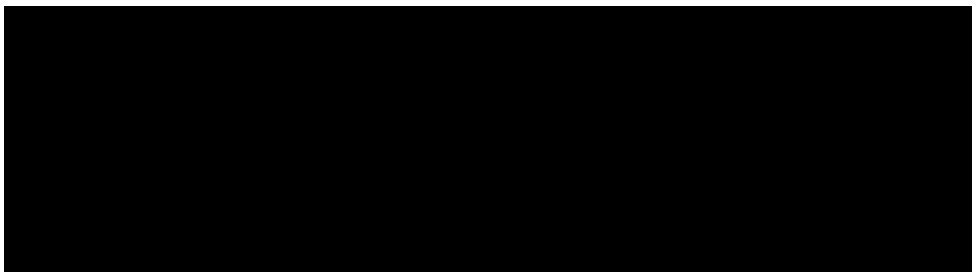

图 5: 经典神经网络 (NN) 和贝叶斯神经网络 (BNN) 之间的区别

2.3.2 贝叶斯框架 (Bayesian Framework)

BNN 框架基于将神经网络理论与贝叶斯推断相结合的思想。该方法所依赖的基本概念确实来自贝叶斯统计，它表达了先验信念影响后验信念的事实。这个想法可以形式化为以下关键定理：

定理 2.2 (贝叶斯定理). 给定两个事件A, B, 使得$\mathbb{P}(B) \neq 0$, A 给定B 的条件概率，表示为$\mathbb{P}(A|B)$, 可

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)mathbbP(A)}{mathbbP(B)}. \tag{18}$$

> 在使用 BNN 时，我们将把同一地定理应用于概率分布（表示为 p）的连续版本，并且感兴趣的分布将是网络参数 $\boldsymbol{\theta} := \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$ 的分布，因为从中我们可以推断输出的分布。

定理 2.3 (贝叶斯法则). 对于两个连续随机变量X 和Y, 让我们用f_X 和f_Y 表示它们的概率密度。那么

$$f_{X|Y=y}(x|y) = \frac{f_{Y|X=x}(y|x)f_Y(y)}{f_X(x)}, \quad \text{其中} f_{Y|X}(y|x)f_X(x) = f_{X,Y}(x,y) = f_{X|Y}(x|y)f_Y(y) \tag{19}$$

> 贝叶斯定理用于利用来自观察数据/物理的信息更新关于网络参数分布的先验知识。同时，事实证明我们可以建立概率分布和损失函数之间的直接关系：预测的良好质量确实可以通过以下方式表示：

1. 概率密度的高值；
2. 损失函数的低值。

因此，通过考虑它们在损失函数内的相反数并以此恢复到经典优化问题框架中，可以将概率密度插入到 2.1 节介绍的 NN 中。

2.3.3 似然分布 (Likelihood Distribution)

我们考虑这样一个场景：数据集 D 由外力测量值（表示为 D_f）和 PDE 解（内部表示为 D_u，边界上表示为 D_b）组成。正如在直接问题的情况下，通常不依赖域内的解的测量，而只依赖边界上的测量，因此在定义数据集 D 时区分这些点是很方便的，然后将其拆分为：

$$D = D_u \cup D_b \cup D_f.$$

> 考虑到可用测量受某些噪声影响是合理的，这是由于测量工具不可避免的缺陷。此外，原则上我们不应包括不同量测量之间的任何共同影响，因此我们假设 $\bar{\mathbf{u}}, \bar{\mathbf{f}}$ 是独立的高斯分布，以真实的隐藏值 $\mathbf{u}, \mathbf{f}$ 为中心：

2.3.4 先验分布 (Prior Distribution)

关于网络参数 $\boldsymbol{\theta}$ 分布的初始知识由 $p(\boldsymbol{\theta})$ 总结，这被称为先验 *(prior)*。NN 参数的一个经典选择是考虑 $\boldsymbol{\theta}$ 为先验 *(a priori)* 分布为多元正态分布。如 [11] 所述，贝叶斯学习框架中的一个常见选择是假设神经网络的每个权重和偏置之间相互独立且均值为零。对于方差，参考论文假设每个网络参数为 1，而在提出的实现中，方差设置为

$$\sigma_\theta^2 = \frac{50}{N_l}$$

> 其中 N_l 是每层的神经元数量。$N_l = 50$ 是参考论文中的选择，因此如果每层神经元数量相同，我们设置相同的方差，而在参数较少的网络情况下确保更大的灵活性（表现为更大的方差）。因此，

$$p(\boldsymbol{\theta}) = \prod_{i=1}^{N_\theta} \frac{1}{\sqrt{2\pi\sigma_\theta^2}} \exp\left(-\frac{(\boldsymbol{\theta}^{(i)} - 0)^2}{2\sigma_\theta^2}\right), \tag{20}$$

其中 N_θ 是网络参数的总数（同时考虑权重和偏置）。

2.3.5 后验分布 (Posterior Distribution)

贝叶斯定理能够从先验和似然重建量 $p(\boldsymbol{\theta}|D)$，这被称为 $\boldsymbol{\theta}$ 的*后验分布 (posterior distribution)*，代表在从数据获得信息后对参数分布的先验知识的更新。定理的表述如下：

$$p(\boldsymbol{\theta}|D) = \frac{p(D|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(D)}.$$

> $p(D)$ 代表数据集的分布，但其计算通常是不可行的。因此，我们仅利用贝叶斯定理得到比例关系：

$$p(\boldsymbol{\theta}|D) \propto p(D|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{21}$$

如上所述的后验重建随后被利用来采样一组$\boldsymbol{\theta}_i{}_{i=1}^{M}$，其中

$$\boldsymbol{\theta}_i \sim p(\boldsymbol{\theta}|D) \quad i = 1, \ldots, M$$

> 然而，使用神经网络直接计算后验通常是不可能的，因此涉及马尔可夫链蒙特卡洛或变分推断（如第 3 章中介绍的方法）的方法可用于直接从后验分布采样。重建了参数的后验分布后，它们随后用于重建神经网络的后验预测分布——即网络输出 $f_\theta(\mathbf{x})$ 的分布。实际上，通过生成对应于采样参数的神经网络输出，给定固定输入 $\mathbf{x}^*$ 的输出 f 的均值和方差的估计如下执行：

2.4 贝叶斯物理信息神经网络 (Bayesian Physics Informed Neural Networks)

上述讨论没有考虑任何来自物理的信息，而只是依赖于数据。贝叶斯物理信息神经网络 (B-PINNs) 方法包括向 BNN 模型添加知识，即问题解需要满足给定的偏微分方程和边界条件。该方法在主要的参考论文 [11], [7] 中实现。

2.4.1 物理似然 (Physical Likelihood)

它们代表了 PINNs 到贝叶斯框架的集成，因为 PDE 的约束作为*似然项的附加项*出现。在贝叶斯定理的表述中，来自偏微分定律的信息是存在的，给定了似然项的修改：

$$p(\boldsymbol{\theta}|D, R) \propto p(D, R|\boldsymbol{\theta})p(\boldsymbol{\theta}). \tag{22}$$

数据 $p(D, R|\boldsymbol{\theta})$ 的似然计算为

$$p(D, R|\boldsymbol{\theta}) = p(D|\boldsymbol{\theta})p(R|\boldsymbol{\theta}) \tag{23}$$

在方程 22 中，$p(D|\boldsymbol{\theta})$ 与非物理信息情况下的方程 15 中的量相同，而 $p(R|\boldsymbol{\theta})$ 以人工方式扮演相同的角色，但代表来自物理定律而非测量的信息。考虑到 2.2 节中已经定义的

配置点集$\mathbf{x}_i{}_{i=1}^{N_{col}}$，目标是要求在这些点上由神经网络近似的解满足偏微分方程。因此，人们可以像对拟合点所做的那样进行处理，同时强加解接近测量值，并且在 B-PINNs 框架中通常认为这部分似然也是高斯的：

$$p(R|\boldsymbol{\theta}) = \prod_{i=1}^{N_{col}} \frac{1}{\sqrt{2\pi\sigma_r^2}} \exp\left(-\frac{(\mathcal{R}(\tilde{\mathbf{u}}^{(i)}) - 0)^2}{2\sigma_r^2}\right). \tag{24}$$

在方程 23 中，$\mathcal{R}$ 如 2.2 节中那样表示残差，而 σ_r 扮演拟合情况下 σ_u 或 σ_f 的角色，并代表与物理知识相关的人工不确定性。模型近似现象越好，σ_r 就越小；还要注意，通过引入数据不确定性和物理知识不确定性值之间的差异，可以给似然的两个分量赋予不同的权重，从而确保最可靠的信息（即与之相关的不确定性较小的信息）权重更大，并在学习过程中更具决定性。

2.4.2 逆问题的先验 (Prior for Inverse Problems)

贝叶斯框架也可以应用于逆问题，通过考虑模型参数 $\boldsymbol{\lambda}$（遵循与 2.2 节相同的符号）遵循特定的概率分布。我们可以假设对 $\boldsymbol{\lambda}$ 有先验知识，并且可以合理地假设它独立于 $\boldsymbol{\theta}$，因为模型和网络参数之间没有关系。因此，包含两者信息的先验项 $p(\boldsymbol{\theta}, \boldsymbol{\lambda})$ 可以计算为

$$p(\boldsymbol{\theta}, \boldsymbol{\lambda}) = p(\boldsymbol{\theta})p(\boldsymbol{\lambda}).$$

> 有了上述先验的表达式，在直接问题情况下所做的相同讨论仍然成立，只要我们考虑到数据的似然也取决于固定的模型参数，并且 $p(D|\boldsymbol{\theta})$ 被 $p(D|\boldsymbol{\theta}, \boldsymbol{\lambda})$ 替换。因此，贝叶斯定理的表述读作

$$p(\boldsymbol{\theta}, \boldsymbol{\lambda}|D, R) = \frac{p(D, R|\boldsymbol{\theta}, \boldsymbol{\lambda})p(\boldsymbol{\theta})p(\boldsymbol{\lambda})}{p(D)p(R)}.$$

> 后者可以用所有上述简化重写为：

$$p(\boldsymbol{\theta}, \boldsymbol{\lambda}|D, R) \propto p(D|\boldsymbol{\theta}, \boldsymbol{\lambda})p(R|\boldsymbol{\theta}, \boldsymbol{\lambda})p(\boldsymbol{\theta})p(\boldsymbol{\lambda}) \tag{25}$$

我们获得了一个现成的比例关系。最后，对所有函数命名 $L = \log(p)$，我们得到：

$$Loss = L(D; \boldsymbol{\theta}, \boldsymbol{\lambda}) + L(R; \boldsymbol{\theta}, \boldsymbol{\lambda}) + L(\boldsymbol{\theta}) + L(\boldsymbol{\lambda}) = Loss_{data} + Loss_{pde} + Prior_{\theta} + Prior_{\lambda} \tag{26}$$

3 算法 (Algorithms)

如 2.4 节所述，贝叶斯机器学习的核心概念是从权重的后验分布 $p(\boldsymbol{\theta}|D)$（或逆问题中的 $p(\boldsymbol{\theta}, \boldsymbol{\lambda}|D)$）中采样。然而，后验的精确重建是不可行的，正如贝叶斯定理仅通过方程 20 中的比例关系进行近似所证明的那样。

在本章中，我们提出了三种实现的算法，它们代表了从合理近似真实分布的分布中采样 $\boldsymbol{\theta}$ 值的不同解决方案：

- **哈密顿蒙特卡洛 (Hamiltonian Monte Carlo, HMC)**，一种常见的马尔可夫链蒙特卡洛方法，利用哈密顿动力学的离散模拟和接受-拒绝步骤的校正；
- **变分推断 (Variational Inference, VI)**，旨在通过在一个比未知目标更简单的参数族中寻找最佳近似来重建目标分布的替代品；
- **斯坦变分梯度下降 (Stein Variational Gradient Descent, SVGD)**，它基于类似于 VI 的哲学，是梯度下降的对应物，它迭代地传输一组“粒子”以匹配目标。

我们将专门用一节来介绍它们，涵盖理论基础和算法机制。

对数后验 (Log-posterior) 在这些算法中，我们使用后验的对数进行操作，因此我们专门用这一节来计算稍后需要的这个量。根据乘积的对数规则，方程 21 中的关系变为：

$$L(\boldsymbol{\theta}) := \log(p(\boldsymbol{\theta}|D, R)) \propto \log(p(D, R|\boldsymbol{\theta})) + \log(p(\boldsymbol{\theta})) \tag{27}$$

通过将 22 和 15 代入方程 26，我们得到

$$L(\boldsymbol{\theta}) \propto \log(p(D_u|\boldsymbol{\theta})) + \log(p(D_b|\boldsymbol{\theta})) + \log(p(D_f|\boldsymbol{\theta})) + \log(p(R|\boldsymbol{\theta})) + \log(p(\boldsymbol{\theta})) \tag{28}$$

由于项的指数性质，上述关系可以简化为均方误差的总和。实际上，从 16–18 和 23 我们得到

$$\log(p(D_u|\boldsymbol{\theta})) \propto -\frac{1}{2\sigma_u^2}\sum_{i=1}^{N_u}(\tilde{\mathbf{u}}^{(i)} - \mathbf{u}^{(i)})^2 + \frac{N_u}{2}\log\left(\frac{1}{\sigma_u^2}\right), \tag{29}$$

$$\log(p(D_b|\boldsymbol{\theta})) \propto -\frac{1}{2\sigma_b^2}\sum_{i=1}^{N_b}(\tilde{\mathbf{u}}^{(i)} - \mathbf{u}^{(i)})^2 + \frac{N_b}{2}\log\left(\frac{1}{\sigma_b^2}\right), \tag{30}$$

$$\log(p(D_f|\boldsymbol{\theta})) \propto -\frac{1}{2\sigma_f^2}\sum_{i=1}^{N_f}(\tilde{\mathbf{f}}^{(i)} - \mathbf{f}^{(i)})^2 + \frac{N_f}{2}\log\left(\frac{1}{\sigma_f^2}\right), \tag{31}$$

$$\log(p(R|\boldsymbol{\theta})) \propto -\frac{1}{2\sigma_r^2}\sum_{i=1}^{N_{col}}(\mathcal{R}(\tilde{\mathbf{u}}^{(i)}) - 0)^2 + \frac{N_{col}}{2}\log\left(\frac{1}{\sigma_r^2}\right). \tag{32}$$

在 [3] 中，方程 27 被视为具有系数$\alpha_i{}_{i=1}^{5}$ $L(\boldsymbol{\theta}) \propto \alpha_1 \log(p(D_u|\boldsymbol{\theta})) + \alpha_2 \log(p(D_b|\boldsymbol{\theta})) + \alpha_3 \log(p(D_f|\boldsymbol{\theta})) + \alpha_4 \log(p(R|\boldsymbol{\theta})) + \alpha_5 \log(p(\boldsymbol{\theta}))$ 然而，在这项工作中，我们出于两个主要原因排除了这个选项：

1. 由于在应用贝叶斯定理时去除了分母，方程 20 不再代表概率分布，而只是一个近似。然而，在方程 27 中引入系数将意味着与贝叶斯定理的进一步分离。
2. 给予每一项不同权重的目标仍然可以通过保持与概率解释的一致性来实现，这要归功于用户对不确定性的选择。

3.1 哈密顿蒙特卡洛 (Hamiltonian Monte Carlo)

我们提出的第一种方法是哈密顿蒙特卡洛，我们将其简称为 HMC；通过整合蒙特卡洛技术和哈密顿动力学，它产生一系列随机样本，这些样本收敛到根据我们的目标概率分布进行分布。

3.1.1 理论基础 (Theoretical Foundations)

该算法是马尔可夫链蒙特卡洛 (MCMC) 方法家族中的经典，它通过构建一个以所需分布为平衡分布的马尔可夫链来实现从概率分布中采样。在 HMC 中，目的包括获得一系列随机样本，这些样本收敛到根据直接采样困难的目标概率分布进行分布；为此，我们首先使用数值积分模拟哈密顿动力学，然后通过接受-拒绝步骤进行校正，以减少离散化误差。

哈密顿动力学 (Hamiltonian Dynamics) 让我们考虑 27 中的后验对数，并将势能 *(potential energy)* 定义为它的相反数：

$$U(\boldsymbol{\theta}) := -L(\boldsymbol{\theta})$$

注意，这种在势能和（对数）后验之间符号上的关系是有意义的：此设置中的自然目标确实是最大化概率，并且通过遵循通常的物理惯例，最小化势能。然后，我们使用它来定义哈密顿函数，该函数将描述连续系统的动力学；注意，还添加了一个模仿动能的项：

$$H(\boldsymbol{\theta}, \mathbf{r}) := U(\boldsymbol{\theta}) + \frac{1}{2}\mathbf{r}^T\mathbf{M}^{-1}\mathbf{r}, \tag{33}$$

在上述公式中，$\mathbf{r}$ 是辅助动量变量，$\mathbf{M}$ 扮演质量矩阵的角色；通常设置 $\mathbf{M} = \mathbf{I}$，或对于某些 $\alpha > 0$，$\mathbf{M} = \alpha\mathbf{I}$。与方程 32 相关的哈密顿系统读作

$$\left\{ \frac{d\boldsymbol{\theta}}{dt} = \frac{dH}{d\mathbf{r}} = \mathbf{M}^{-1}\mathbf{r} \; \frac{d\mathbf{r}}{dt} = -\frac{dH}{d\boldsymbol{\theta}} = -\nabla U(\boldsymbol{\theta}). \right. \tag{34}$$

现在，我们在哈密顿动力学和概率之间建立联系：基本上，求解 33 中的系统对应于从联合分布生成 $(\boldsymbol{\theta}, \mathbf{r})$ 的样本

$$\pi(\boldsymbol{\theta}, \mathbf{r}) \sim \exp(-H(\boldsymbol{\theta}, \mathbf{r})) \tag{35}$$

并且上面的采样用于模拟从后验分布 $p(\boldsymbol{\theta}|D, R)$ 的采样，这 - 直到一个常数 - 与 $\exp(-U(\boldsymbol{\theta}))$ 重合。由于我们对 $\mathbf{r}$ 的样本不感兴趣，我们随后丢弃它们并只考虑 $\boldsymbol{\theta}$ 的样本，这遵循边缘分布 $p(\boldsymbol{\theta}|D, R)$。

Leap Frog 关于方程 33 的解，我们使用 *leap-frog* 方法进行数值处理，包括在 $\boldsymbol{\theta}$ 的后续更新之间引入 L 个额外的步骤，形式为：

$$\text{for } k = 1, \dots, L \quad \begin{cases} \mathbf{r}^{(k+\frac{1}{2})} = \mathbf{r}^{(k)} - \frac{\Delta t}{2} \nabla(U(\boldsymbol{\theta}^{(k)})) \\ \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} + \Delta t \mathbf{M}^{-1} \mathbf{r}^{(k+\frac{1}{2})} \\ \mathbf{r}^{(k+1)} = \mathbf{r}^{(k+\frac{1}{2})} - \frac{\Delta t}{2} \nabla(U \end{cases} \tag{36}$$

在方程 35 中，Δt 表示步长，它应该与 L 相关，L 表示每次更新执行的步数，正如我们将在 3.1.3 小节中说明的那样。

接受-拒绝 (Acceptance-Rejection) 为了减少离散化引入的误差，新值并不总是被接受，这种选择在随后的接受-拒绝步骤中受到控制。考虑两个随后的 $\boldsymbol{\theta}$ 值，为简单起见分别表示为 $\boldsymbol{\theta}^{(0)}$ 和 $\boldsymbol{\theta}^{(1)}$，我们计算 α：

$$\alpha := \min\left(1, \frac{\exp(-H(\boldsymbol{\theta}^{(1)}, \mathbf{r}^{(1)}))}{\exp(-H(\boldsymbol{\theta}^{(0)}, \mathbf{r}^{(0)}))}\right) = \min(1, \exp(-(H(\boldsymbol{\theta}^{(1)}, \mathbf{r}^{(1)}) - H(\boldsymbol{\theta}^{(0)}, \mathbf{r}^{(0)})))).$$

这个值是接受 $\boldsymbol{\theta}^{(1)}$ 的概率：如果 $\alpha \geq p$ 其中 $p \sim \mathcal{U}(0, 1)$，我们接受采样值 $\boldsymbol{\theta}^{(1)}$。定义 $h := H(\boldsymbol{\theta}_1, \mathbf{r}_1) - H(\boldsymbol{\theta}_0, \mathbf{r}_0)$，我们可以将接受率的表达式重写为：

$$\alpha = \min(1, \exp(-h)) \implies h < 0 \Rightarrow \alpha = 1$$

当 $h < 0$ 时，意味着哈密顿量已经减少，因此我们希望总是接受新参数。相比之下，当 $h > 0$ 时，哈密顿量变差，我们希望以随 h 递减的概率接受。整个过程输出一系列网络参数$\boldsymbol{\theta}_{i=1}^{(i)\,N}$；并非所有这些都被考虑在内，而是：

- 我们丢弃一些初始样本，引入 *burn-in*，以提高最终预测的质量；
- 我们引入 *stride*，并在最终的 $\boldsymbol{\theta}$ 列表中每隔 S 个考虑一个参数集。通过这种方式，我们减少了可能出现在后续样本之间的相关性，并改进了最终参数采样是独立且同分布的假设。

3.1.2 算法 (Algorithm)

我们现在报告已实现的完整算法：

[H] 哈密顿蒙特卡洛 (Hamiltonian Monte Carlo) **Input:** 初始值 $\boldsymbol{\theta}^{(0)}$, 总迭代次数 N, burn-in B, leap-frog 步数 L, stride S, 步长 Δt, 质量矩阵参数 α

for $k = 1, \ldots, N$ **do**
    // Leap-frog step
    sample $\mathbf{r}^{(k-1)} \sim \mathcal{N}(\mathbf{0}, \mathbf{M})$ set $(\boldsymbol{\theta}_0, \mathbf{r}_0) = (\boldsymbol{\theta}^{(k-1)}, \mathbf{r}^{(k-1)})$ **for** $i = 0, \ldots, (L-1)$ **do**
        $mathbf{r}_i = \mathbf{r}_i - \frac{\Delta t}{2} \nabla U(\boldsymbol{\theta}_i)$
        $boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \Delta t \mathbf{M}^{-1} \mathbf{r}_i$
        $mathbf{r}_{i+1} = \mathbf{r}_i - \frac{\Delta t}{2} \nabla U(\boldsymbol{\theta}_{i+1})$
    end
    // Acceptance-Rejection step
    sample $p \sim \mathcal{U}(0,1)$ $\alpha_{accept} = \min(1, \exp(-(H(\boldsymbol{\theta}_L, \mathbf{r}_L) - H(\boldsymbol{\theta}_0, \mathbf{r}_0))))$ **if** $p \geq \alpha_{accept}$ **then**
        $\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}_L$
    end
    else
        $\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)}$
    end
end
从集合$\boldsymbol{\theta}_{i=1}^{(i)\,N}$ 中，使用以下方法提取子集$\boldsymbol{\theta}^{*(N-B)/S}_{i\,i=1}$ // Burn-in: 丢弃前 B 个样本
    // Stride: 每 S 个取一个样本，从最后一个开始倒推

然后，使用权重集通过产生设置每个$\boldsymbol{\theta}^*_{i\,i}$ 元素作为网络参数的神经网络输出，来计算解 $\mathbf{u}$ 和参数场 $\mathbf{f}$ 的样本，并获得集合 $(\boldsymbol{\theta}^*_i)\_i$ 和 $(\boldsymbol{\theta}^*_i)\_i$。由于这些样本的统计数据，我们能够量化不确定性并在计算了样本的均值和标准差后为神经网络输出生成置信区间。

3.1.3 参数选择 (Parameters' choice)

epoch 数量 N 或 burn-in B 的选择已通过微调进行；我们将 N 设置为一个能够在合适的计算时间（HMC 训练模拟未超过 20 分钟）内以相当低的误差重建函数并保持接受率高于 80 对于 B，我们总是保留至少 50 另一方面，我们让 B 决定来自物理的信息进入对数后验的时间；事实上，我们注意到当网络输出远离其真实值时对其进行微分对学习过程是有害的。鉴于自动微分的计算负担，物理损失的贡献确实通常比拟合损失的贡献更重。因此，事实证明最优的策略是仅通过拟合点首先接近输出，然后让 PDE 发挥作用并改进重建——或重建缺乏拟合点的输出部分。在目前状态下，文献中没有关于固定 L 和 Δt 最佳值的策略。一个直接的想法是在它们之间建立反比关系，通过固定它们的乘积等于一个常数 K：

$$K = L\Delta t.$$

通过这种方式，我们确实可以固定行进的总“距离”，但在更多更小的步骤中覆盖它显然在计算上要求更高。因此，一旦选择了 K，要遵循的一般规则是尽可能减少 L 以支持更大的 Δt（这不影响计算时间），但保持相当高的接受率。大 Δt 的风险确实是性能下降，正如我们指出的那样，网络参数的更新需要渐进以防止转向新提议总是被拒绝的体制（由于方程 35 中梯度的爆炸）。

3.2 变分推断 (Variational Inference)

我们将在本节中说明另外两种属于不同框架的方法：变分贝叶斯方法，我们首先介绍变分推断 (VI) 方法。为了近似后验概率，变分推断类方法是蒙特卡洛采样方法的替代方案，用于对难以直接评估或采样的复杂分布进行统计推断采取完全贝叶斯方法。特别是，蒙特卡洛技术可能很慢并且难以达到收敛。在变分方法中，我们改变了视角，因为我们不是通过一组样本提供后验的间接近似，而是获得后验近似的局部最优、精确解析解，稍后可以从中生成样本。

3.2.1 理论基础 (Theoretical Foundations)

变分推断方法家族基于将问题（在这种情况下，是从分布中采样 $\boldsymbol{\theta}$）转化为确定性优化的策略，该优化用更简单的分布 $Q(\boldsymbol{\theta}|\zeta)$ 近似目标分布 $P(\boldsymbol{\theta}|D)$。

概率的距离 (Distance of Probabilities) 为了执行此步骤，我们需要确定概率测度近似的准确程度，因此需要一种捕获概率分布之间相似性的工具。我们使用目标分布和我们正在重建的分布之间的 Kullback–Leibler (KL) 散度：KL 散度是一个统计距离，让我们捕获一个概率分布与另一个的相似程度。**定义 3.1 (Kullback–Leibler 散度).** 给定一个测度空间 (X, μ) 和两个概率分布 P 和 Q，其密度分别为 $p(x)d\mu$ 和 $q(x)d\mu$，则 P 和 Q 之间的 KL 散度定义为

$$\mathcal{D}_{KL}(P||Q) = \int_X p(x) \log\left(\frac{p(x)}{q(x)}\right) d\mu$$

然后，该家族的算法试图最小化目标分布和近似分布之间的 KL 散度。在我们的 B-PINNs 背景下，我们感兴趣的是最小化 $\mathcal{D}_{KL}(Q(\boldsymbol{\theta}|\zeta)||P(\boldsymbol{\theta}|D))$ 的 ζ 值

$$\begin{aligned}
\zeta^* &= \text{argmin}[\mathcal{D}_{KL}[Q(\boldsymbol{\theta}|\zeta)||P(\boldsymbol{\theta}|D)]] = \text{argmin} \int Q(\boldsymbol{\theta}|\zeta) \log \frac{Q(\boldsymbol{\theta}|\zeta)P(D)}{P(\boldsymbol{\theta})P(D|\boldsymbol{\theta})} d\boldsymbol{\theta} = \\
&= \text{argmin} \int Q(\boldsymbol{\theta}|\zeta) \left[\log \frac{Q(\boldsymbol{\theta}|\zeta)}{P(\boldsymbol{\theta})} - \log P(D|\boldsymbol{\theta}) + \log P(D)\right] d\boldsymbol{\theta} = \\
&= \text{argmin}[\mathcal{D}_{KL}(Q(\boldsymbol{\theta}|\zeta)||P(\boldsymbol{\theta})) - \mathbb{E}_{Q(\boldsymbol{\theta})}[\log P(D|\boldsymbol{\theta})] + \text{constant}]
\end{aligned}$$

训练阶段 (Training Phase) 最小化问题随后可以概括为寻找 $\zeta^* = \operatorname{argmin} \mathcal{F}(D, \zeta)$，其中：

$$\mathcal{F}(D, \zeta) = \mathcal{D}_{KL}(Q(\boldsymbol{\theta}|\zeta)||P(\boldsymbol{\theta})) - \mathbb{E}_{Q(\boldsymbol{\theta})}[\log P(D|\boldsymbol{\theta})] \tag{37}$$

在 36 中，第一项依赖于先验，称为复杂性成本 *(complexity cost)*，而第二项是依赖于数据的经典似然成本 *(likelihood cost)*。我们的计算任务将是用其蒙特卡洛期望近似 $\mathcal{F}(D, \zeta)$ 并将其用作使用 Adam 优化的训练损失函数。作为辅助分布 Q，我们选择一个由一组参数描述的族，我们将遵循 [11] 中实施的常见做法用 $\zeta \in \mathbb{R}^{2d_\theta}$ 表示，并采用可分解的高斯分布

$$Q(\boldsymbol{\theta}, \zeta) = \prod_{i=1}^{d_\theta} q(\theta_i, \zeta_i^\mu, \zeta_i^\rho) \tag{38}$$

这也允许我们解析地计算算法所需的一些量。在方程 37 中，参数向量 ζ 具有对 $(\zeta_i^\mu, \zeta_i^\rho)$ 作为分量，其中 $i = 1, \ldots, d_\theta$。每一对表征一个网络参数的分布，该分布被取为彼此独立的单变量高斯分布，具有均值 ζ_i^μ 和标准差 $\log(1 + \exp(\zeta_i^\rho))$。注意我们选择使用量 ζ^ρ 而不是标准差，是为了摆脱必须为正量的约束，因为类梯度下降算法在参数存在于整个集合 $\mathbb{R}$ 中时工作最佳。

预测阶段 (Prediction Phase) 上述优化过程使我们能够恢复参数族中的最佳近似分布，我们将表示为 $Q(\zeta^*)$，我们可以像它是真实的后验分布一样从中采样。与 HMC 等算法不同，实际上，这次过程返回的是一个分布而不是来自一个分布的一组样本，这的一个明显结果是我们能够直接从输出分布生成任意数量的参数样本$\boldsymbol{\theta}_i^*{}_i$。再次通过设置集合的每个元素作为网络参数，我们就可以获得解 $(\boldsymbol{\theta}_i^*)\_i$ 和参数场 $(\boldsymbol{\theta}_i^*)\_i$ 的样本集，以重建它们的分布。

3.2.2 算法 (Algorithm)

我们现在报告已实现的完整算法：

[H] 变分推断 (Variational Inference) **Input:** 总 epoch 数 N, 优化 ζ 的 $\boldsymbol{\theta}$ 样本数 N_z, 所需样本数 M, 辅助参数初始值 ζ_i

for $k = 1, \ldots, N$ **do**

  独立地从 $\mathcal{N}(\mathbf{0}, \mathbf{I}_{d_\theta})$ 采样 $\mathrm{z}_{j=1}^{(j)\,N_z}$ 设置 $\boldsymbol{\theta}^{(j)} = \zeta^\mu + \log(1 + \exp(\zeta^\rho)) \odot \mathbf{z}^{(j)} \quad j =$**end** 独立地从 $1, \ldots, N_z$ $\mathcal{L}(\zeta) = \frac{1}{N_z} \sum_{j=1}^{N_z} [\log(Q(\boldsymbol{\theta}^{(j)}; \zeta)) - \log P(\boldsymbol{\theta}^{(j)}) - \log P(D|\boldsymbol{\theta}^{(j)})]$ 使用 Adam 优化器用梯度 $\nabla_\zeta \mathcal{L}(\zeta)$ 更新 ζ

3.2.3 参数选择 (Parameters' choice)

VI 方法与贝叶斯框架之外的算法（如 GD 或 Adam）在分布参数的训练部分具有一些共同特征；然后，它的最后一步保证进入概率框架，因为它输出是针对学习到的分

布采样的。这也反映在方法参数的选择上：epoch 数量 N 的选择应遵循与 Adam 和 GD 相同的原则，因为随着 epoch 数量的增加，学习到的参数会有所改进，并且对于预测仅使用学习到的参数的最终值。在此阶段，学习率 α 也应以类似于确定性算法的方式选择。另一方面，样本数量 M 不影响训练，因此不影响计算时间；对于像 SVGD 这样的方法，我们可以自由增加它以产生更显著的置信区间，而不会造成计算过载。

3.3 斯坦变分梯度下降 (Stein Variational Gradient Descent)

另一种通过不同策略最小化 KL 散度来近似后验的方法是斯坦变分梯度下降 (SVGD)。此方法可以被视为完全贝叶斯推断的梯度下降的自然对应物。

3.3.1 理论基础 (Theoretical Foundations)

该方法的理论基础与 VI 共享，因为潜在的主要思想仍然是通过最小化 KL 散度来学习后验。

解析推导 (Analytical Derivation) 该算法的解析推导在这种情况下比以前的情况要长，因此我们将简单地强调关键段落；更多细节可以在参考文献 [8] 中找到。

1. 辅助族 Q 这次由分布组成，这些分布通过随机变量 x 的平滑变换 T 获得，该随机变量从易处理的参考分布 q_0 中抽取：$z = T(x)$ 其中 $x \sim q_0$；

2. 我们寻找一种迭代算法来寻找最佳 T，这通过其微扰公式 $T(x) = x + \epsilon\phi(x)$ 处理，采用 $T_{n+1}(x) = T_n(x) + \epsilon\phi_n(x)$ 形式的方案；

3. 我们将此公式插入 KL 散度中，由于我们想要最小化此量，我们搜索最陡下降方向 ϕ，这与最大化 $-\nabla_\epsilon \mathcal{D}_{KL}(Q_T||P)|_{\epsilon=0}$ 一致；

4. 对于向量函数 $\phi(x)$，我们可以定义斯坦算子 $\mathcal{A}_p\phi(x) = \phi(x)\nabla_x \log p(x)^T + \nabla_x\phi(x)$。展开计算，我们得到：$-\nabla_\epsilon \mathcal{D}_{KL}(Q_T||P)|_{\epsilon=0} = \mathbb{E}_{x\sim Q_T}[\mathrm{tr}(\mathcal{A}_p(\phi(x)))]$；

5. 优化问题现在读作在 RKHS（具有核 k 的再生核希尔伯特空间）空间中最大化所谓的核化斯坦差异 $\mathbb{E}_{x\sim Q_T}[\mathrm{tr}(\mathcal{A}_p(\phi(x)))]$；

6. 此问题的解是已知的，它是 $\phi^*_{Q_T,P}(x) = \mathbb{E}_{x\sim Q_T}[k(x,\cdot)\nabla_x \log(p(x)) + \nabla_x k(x,\cdot)] = \mathbb{E}_{x\sim Q_T}[\mathcal{A}_p k(x)]$；

7. 最后，我们想用蒙特卡洛估计期望值，我们可以用从 q_0 采样的 N 个粒子来做，用步骤 2 中指示的公式迭代更新。

数值模拟 (Numerical Simulation) 应用此方法背后的直觉基本上是与有限数量的神经网络（我们将称为*粒子 (particles)*）一起工作，并传输，迭代后迭代，所有粒子的参数朝向目标分布，遵循最小化 KL 散度给出的方向。这些粒子代表由推导的方程 38 提供的蒙特卡洛期望计算的样本。在第一次迭代中，它们是从分布 q_0 中随机抽取的，每次都被用来计算斯坦算子。最后，给定增量，我们直接更新样本，就像它们是从 q_1 中抽取的一样，依此类推。

$$\mathbb{E}_{\boldsymbol{\theta}\sim Q}[k(\boldsymbol{\theta},\cdot)\nabla_{\boldsymbol{\theta}}\log p(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}}k(\boldsymbol{\theta},\cdot)], \tag{39}$$

在解析推导期间，我们从未提及 k 的含义；它是一个严格正定核，我们在实现中利用的一个合适例子是 RBF 核：

$$k(x,x') = \exp\left(-\frac{1}{h}||x-x'||^2\right) \tag{40}$$

其中 $h>0$ 为正带宽。注意，我们可以给方程 38 中的两个加数关联特定含义：

1. $k(\boldsymbol{\theta},\cdot)\nabla_{\boldsymbol{\theta}}\log p(\boldsymbol{\theta})$ 通过遵循平滑梯度方向将粒子驱向 p（或 $\log p$，即方程 26 的 $L(\boldsymbol{\theta})$）的高概率区域，该方向是所有点的梯度按核函数加权的加权和；

2. $\nabla_{\boldsymbol{\theta}}k(\boldsymbol{\theta},\cdot)$ 充当排斥力，防止所有点一起坍塌到同一点，即 $L(\boldsymbol{\theta})$ 的最大后验概率。事实上，考虑 RBF 核，我们有：

$$\nabla_x k(x,x') = \frac{2}{h}(x-x')k(x,x') \tag{41}$$

因此 x 被驱离 $k(x,x')$ 很大的邻居。如果我们让带宽 $h\to 0$，排斥项消失：这意味着参数更新减少为一组用于最大化 $\log p$ 的典型梯度上升的独立链，所有粒子都将坍塌到局部模式中。

参数的更新随后可以像梯度下降一样进行，引入一个也扮演学习率角色的项。此外，我们还可以实现类似于随机梯度下降（算法 2）的方法，这在处理大型数据集时很有用。

3.3.2 算法 (Algorithm)

我们现在报告已实现的完整算法：

[H] 斯坦变分梯度下降 (Stein Variational Gradient Descent) **Input:** 神经网络数量 N, 每个网络的参数初始值$\boldsymbol{\theta}_{i i=1}^{N}$, 总 epoch 数 E, 步长 ϵ

for $k=1,\dots,E$ **do**

  $\phi(\boldsymbol{\theta}_i) = \frac{1}{N}\sum_{j=1}^{N} k(\boldsymbol{\theta}_i,\boldsymbol{\theta}_j)\nabla_{\boldsymbol{\theta}_j}L(\boldsymbol{\theta}_j) + \nabla_{\boldsymbol{\theta}_j}k(\boldsymbol{\theta}_i,\boldsymbol{\theta}_j) \quad \forall i=1,\dots,N \quad \boldsymbol{\theta}_i = \boldsymbol{\theta}_i + \epsilon\phi(\boldsymbol{\theta}_i) \quad \forall i=1,\dots,N$

end

收集 E 个 epoch 后所有网络的参数$\boldsymbol{\theta}_{i=1}^{(i)\,N}$.

3.3.3 参数选择 (Parameters' choice)

在参数选择方面，SVGD 所需的决策工作量并不高，特别是与 HMC 相比。事实上，要设置的参数只有 N, E 和 ϵ，但调优的真正障碍在于 SVGD 是计算要求最高的算法，这个问题可以通过并行化粒子进行的计算来部分解决。epoch 数量 E 影响预测的质量，而粒子数量 N 可以决定 UQ 的质量；在 [8] 中建议 $E, N \to \infty$ 时收敛到目标分布，因此（显然）E 和 N 越大，分布的近似越好。另一方面，大的 E 和 N 需要更高的计算时间，此外还需要足够的 RAM 内存来存储所有粒子的参数。学习率 ϵ 在参数调优期间发挥了基础性作用，作为其他常见算法的学习率参数，因为它的选择能够防止网络的可训练参数因失控增长而爆炸为 nan。

4 代码概述 (Code Overview)

在本章中，我们首先介绍并阐述了在环境和库方面的编码选择。然后，我们专门用一节来介绍 Python 中的面向对象编程，重点关注项目中实现的特性。最后一部分介绍了仓库的结构以及配置、数据和输出文件的内容。

4.1 工作环境 (Working Environment)

项目中实现的代码 - 从头开始 - 包含在 GitLab 仓库 `PACS-BPINNS` 中，可以在这里找到，并附有 README。在附录 A 中，我们介绍了针对不同操作系统设置工作环境和下载所有必需包的步骤，这些包列在文件 `requirements.txt`[5] 中，并在 4.2 小节中更详细地介绍。

4.1.1 解释器 (Interpreter)

代码是用 Python 编写的，为了运行包含在可执行仓库中的脚本，用户需要 python 3.10.* 版本（从这里下载）。对版本的限制是由于 Python 3.10 中引入了 `match-case` 语句，该语句特别适合模仿 Java 或 C++ 中 `switch` 的行为，并且当我们每种情况需要排序不同的指令时，它保证了比条件语句链更好的可读性。

4.1.2 虚拟环境 (Virtual Environment)

我们选择的工作设置是在 Python 虚拟环境内编写代码，因为这种编程语言在很大程度上依赖于外部依赖项，并且版本更新频繁。实际上，通过选择为不同项目创建和管理单独的环境，我们实现了版本的安全管理：每个环境都可以使用不同版本的包依赖项和/或 Python，这确保了项目之间不会引起依赖冲突。此外，这种选择还可以简化其他程序员对结果的复现，因为他们可以安装所需的包，而不会与机器上已有的包产生冲突。此工具由模块 `virtualenv`（版本 20.14.*，可从这里下载）提供。另一种可能性是建立一个 Conda 环境，但既然我们只对这一个单一项目的环境感兴趣，我们更倾向于依赖这个 Python 的原生功能（Python 3.3 之后可用），而不需要任何第三方发行版。

4.2 库 (Libraries)

在代码中，利用了几个流行的 Python 库；我们现在列出所有这些库，深入探讨与实现相关的那些。

[5] 此文件列出了比 4.2.2 小节中安装和描述的更多的库，因为它已经包含了所介绍库的所有相关依赖项。

4.2.1 内置包 (Built-in Packages)

在这个项目中，我们也依赖于几个常见的 Python 模块来完成更简单的和辅助的任务。所有这些模块都属于 Python 运行时服务或属于 Python 标准库；因此，它们不会出现在 `requirements.txt` 中，而是直接随 Python 分发。

- `os`,提供使用操作系统相关功能的可移植方法的模块。我们主要利用子模块 `os.path`，用于操作文件路径。
- `shutil`，提供许多对文件和文件集合的高级操作的模块。
- `json`，用于处理 JSON 数据的模块，JSON 数据由文本组成，使用 JavaScript 对象表示法编写。
- `argparse`，使用户能够编写用户友好的命令行界面的模块。
- `warnings`，用于打印用户定义的警告的模块。
- `time`, `datetime`，提供与时间相关的函数和用于操作日期和时间的类的模块。

为了调试目的，我们依赖 `pdb`，它是一个定义 Python 程序交互式源代码调试器的模块。通过在代码的所需点插入 `import pdb; pdb.set_trace()`，用户可以中断进入调试器，然后通过键入 `continue` 继续运行直到下一个断点。

4.2.2 外部包 (External Packages)

由重要的开源项目和像 Google 和 Meta 这样的大型科技公司提供的量身定制的库的大量存在是 Python 语言的优势之一；在本节中，我们介绍在代码中利用的主要用于数据科学和 ML 任务的外部库。

Numpy `numpy` 是用于科学计算的最流行的 Python 包之一；它提供了一个多维数组对象 (`numpy.array`)、各种派生对象（如掩码数组和矩阵），以及各种用于数组快速操作（尤其是线性代数）的例程。我们强调 `numpy.array` 类型与 Python 原生 `list` 相比的一些特性：

- 与 Python 的 `list` 类型不同，它对存储对象的同质性有严格要求，并支持逐元素操作；
- 与 `list` 相比，`numpy.array` 更快，其元素在内存中连续存储；
- `numpy.array` 支持项赋值；此特性不被项目中使用的另一种结构共享：TensorFlow 的 `Tensor`（4.2.3 小节）。

Matplotlib `matplotlib.pyplot` 是 `matplotlib` 的用户友好接口，后者是 Python 中用于可视化的综合库。它由使 `matplotlib` 工作方式类似于 MATLAB 的函数集合组成。每个 `pyplot` 函数对图形进行一些操作：例如，它创建一个图形，在绘图区域中绘制一些线条，用标签、标题和图例装饰绘图……

Scipy `scipy.stats` 是 SciPy（另一个用于科学计算的常用库）中致力于统计的模块；它提供了大量的概率分布、摘要和频率统计、相关函数和统计测试。我们特别利用了子模块 `qmc`，它提供准蒙特卡洛生成器和相关的辅助函数。

Tqdm `tqdm` 是一个用于在迭代过程中显示进度条的模块。根据测试（参见文档），此功能增加的计算开销很低（每次迭代约 60ns），并且低于其他类似工具（如 `ProgressBar`）。此外，`tqdm` 使用智能算法来预测剩余时间并跳过不必要的迭代显示，这使得在大多数情况下开销可以忽略不计。

4.2.3 TensorFlow

`tensorflow (tf)` 是 NN 实现最相关的需求，它是由 Google Brain 团队在 2015 年开发的用于机器学习的免费开源软件库。多维元素数组在 TensorFlow 中通过 `tf.Tensor` 对象处理。此数据结构具有以下属性：

- 单一数据类型（例如 `float32`, `int32` 或 `string`）；
- 形状，表示每个 Tensor 轴的长度。

我们还强调了与 `numpy.array` 类型相比的两个根本区别：

- Tensor 有加速器支持，如 GPU 和 TPU；
- Tensor 不支持项赋值。

在代码中，我们使用 TensorFlow 版本 2.9.1，更具体地说，我们下载 `tensorflow-cpu` 库以避免在我们用来运行代码的机器上因没有 GPU 而产生的警告，但如果有兼容 CUDA 的 GPU 可用，通过选择 `tensorflow-gpu` 或 `tensorflow`（然后将检测 GPU 的存在），就能够在 GPU 上运行代码（更多详细信息请参见此处）。然而，对于本项目，考虑到所用神经网络的规模有限，所需的计算时间对于没有 GPU 的笔记本电脑来说总是可行的。

自动微分 (Automatic Differentiation) 自动微分已在 2.2 节中介绍。TensorFlow 可以自动计算模型中参数的梯度，这些梯度随后用于反向传播等算法。为此，框架必须跟踪对模型中的输入 `Tensor` 进行的操作顺序，然后计算相对于适当参数的梯度。为了跟踪操作，我们在 `tf.GradientTape` 内执行自动微分任务；这是一个上下文管理器，能够

记录其中的操作并计算相对于给定变量的梯度。在这种情况下，要求变量被 `tape` 监视 *(watched)*，默认情况下这只对可训练变量发生。因此，对于作为网络输入的任意张量，我们需要调用 `tape.watch()` 函数。

其他主要特性 TensorFlow 框架包含涉及 NN 训练的其他有用特性，这些特性是为 `Tensor` 数据结构和自动微分目的构建的，例如：

- **急切执行 (Eager execution)**：一种立即评估操作的模式，而不是将其添加到稍后执行的计算图中。通过这种类型的执行，可以通过调试器逐步检查代码，因为数据是在每一行代码中增加的，而不是稍后在计算图中增加。
- API 访问最常见的损失、指标、优化器和 `Tensor` 对象的数学运算，实现为与自动微分和梯度带兼容。
- **Keras 子模块**：`tensorflow.keras` 是 TensorFlow 的高级 API，提供了许多用于神经网络实现的工具，旨在用户友好。Keras 确实提供了一致且简单的 API，最大限度地减少了常见用例所需的用户操作次数。我们依靠它来轻松访问内置的激活和损失函数、参数初始化器、层和模型。

4.3 面向对象特性 (Object Oriented Features)

Python 属于面向对象编程 (OOP) 语言家族，代表一种基于“对象”概念的范式，对象可以包含数据和代码，而不是包含一系列要执行的计算步骤的“过程”。Python 提供了各种典型的 OOP 特性（如封装、继承、多态性……），但显然，与其他 OOP 语言一样，它甚至可以用于过程式编程。在本节中，我们介绍一些利用的模块和装饰器[6]，以便为出现在 B-PINNs 框架中的对象和数据结构提供直观的属性。稍后在第 5 章中，我们将指定我们在其中引入它们的类以及选择背后的原因，而在这里我们对所利用的主要功能进行总体概述。

4.3.1 继承 (Inheritance)

在代码中，您可以找到 OOP 继承概念的几个例子，即子类建立在父类之上：我们既有*单继承*的例子，其中子类继承一个超类的特征，也有*多重继承*的例子，其中一个类有多个超类并从所有超类继承。在处理继承时，我们经常重写方法，我们需要在子类和父类的方法之间选择引用哪一个。这也适用于所有方法，也适用于像构造函数这样的特殊方法。实际上，给定一个父类，在基于它构建子类时，我们可能希望为子类定义一个自定义构造函数，该构造函数将在实例化该类时被调用；否则，将调用父类的构造函数。

[6]在 Python 中，装饰器是一种设计模式，允许通过将函数包装在另一个函数中来修改函数的功能。

在其他情况下，比如我们的情况，我们可能希望为子类定义一个构造函数，同时也使用父类的构造函数。在这种情况下，我们依赖于 Python 内置函数 `super()`。

MRO 和 `super()` 函数 在处理继承时，`super()` 允许我们在子类中调用超类的方法，因此在访问已被重写的继承方法时，它变得至关重要。其主要用例是扩展继承方法的功能。从技术上讲，`super()` 返回一个代理对象，该对象将方法调用委托给父类。当继承树是一个复杂的结构时，必须处理对父类方法的调用，理解家谱，即*方法解析顺序 (Method Resolution Order, MRO)* 变得决定性。`super()` 根据 MRO 提供下一个方法，您可以在每个类的私有类属性 `__mro__` 中找到相关信息。MRO 包括按照子类声明中指示的顺序对父类进行排序并完成每个分支；当发现共同的祖先时，它们总是推迟到分支的末尾。该过程在“object”类处停止。在我们的上下文中，我们使用 `super()` 来检索父类的构造函数，我们将实例化父类对象所需的参数传递给它。基本上，对于构造函数，当我们调用 `super().__init__()` 时，它根据完整继承层次结构上下文中所使用的 MRO 算法提供下一个 `__init__()` 方法。`super()` 也可以接受两个参数，这两个参数决定了从树中的哪个点开始寻找方法：第一个是子类，第二个参数是作为该子类实例的对象。

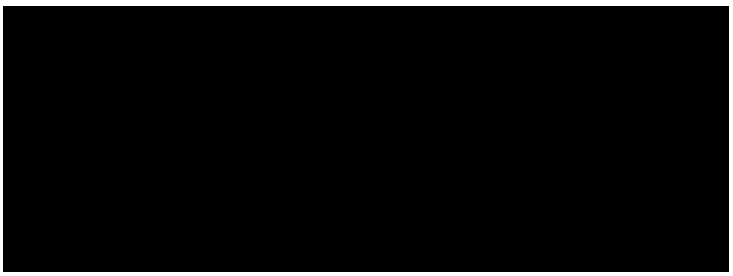

图 6: Python 中的单继承和多重继承

菱形继承 (Diamond inheritance) 在我们对 5.6 小节中类的实现中，我们依赖于类似于所谓的*菱形形状*的家谱：这包括一个类有两个共享其父类的父类，如图 7 所示，实际上，类 `Fourth` 同时继承自 `Second` 和 `Third`，而它们都继承自 `First`。使用上面显示的代码，`Fourth` 的实例具有来自直接父类的属性 `y` 和 `z` 以及来自“祖父母”类的 `x`。`Second` 的实例只有属性 `y` 和 `x`，而 `Third` 的实例只有属性 `z` 和 `x`。对 `super()` 的调用链用于激活所有祖先的 `__init()__`，在我们的案例中也用于将来自 5.2.3 小节中定义的结构的参数传递给祖先的构造函数。

4.3.2 抽象基类 (Abstract Base Class)

抽象类用于表示一般概念，可以用作具体类的基类。在我们的案例中，抽象概念例如是一个训练算法，其中训练管道对所有方法都是通用的，但采样新参数的配方根据特定算法而变化。因此，我们希望将所有通用特性存储在一个如果不被特定子类使用就无

法工作的抽象类中（详见 5.7 小节）。通过这种继承特性，我们在某种意义上将接口与实现分开，因为抽象基类只定义通用方法和属性，而它们的一部分实现则由具体子类处理，我们可以实例化这些子类的对象来处理任务。在 Python 中，模块 `abc` 提供了定义它们的基础设施。从这个模块中，我们专门导入 `ABC`，这是传递给类声明作为父类的内置类，传递给我们想要使其抽象的类。这禁止了抽象类的实例化。从 `abc` 中，我们还导入了装饰器 `@abstractmethod`，如果我们想要强制在子类中重写抽象类的方法，可以将其应用于该方法。这防止我们在并非所有抽象方法（也称为虚方法）都被重写的情况下创建具有抽象父类的类的实例。此特性有助于避免错误，并通过提供在子类中编写方法时要遵循的严格配方，使类层次结构更易于维护。

4.3.3 数据类 (Dataclass)

可能会发生这样的情况：我们有兴趣存储数据并同时利用 OOP 的一些特性，但我们希望有一个比传统类更具体、更轻量级的对象（例如 5.3.4 小节）。从 Python 3.7 开始，对于此任务我们可以依赖由 `dataclasses` 模块提供的 `@dataclass` 装饰器表征的数据类，它们由具有一些限制的 Python 类组成，主要用于包含数据（尽管对此没有严格限制）。它们允许用户用更少的代码定义类，因为例如，方法如 `__init__()` 和 `__repr__()` 会自动添加，而无需实现它们，并且所有方法都是类方法，因此在这个更简单的上下文中我们不需要类的通常多余的 `self` 语法。它们还提供了更多开箱即用的功能；例如，使用选项 `@dataclass(frozen=True)` 我们可以定义一个其实例不可变的类，并且使用诸如 `asdict()`, `astuple()` 等方法，数据类的对象可以转换为字典或元组，这是用于数据存储的其他自然数据结构。

4.3.4 迭代器 (Iterators)

在 Python 中，有可能将类创建为迭代器，这是一种可以遍历其所有值的对象；在我们的项目中，它们对于定义训练批次很有用（5.4.3 小节）。迭代器的实现示例如图 8 所示：

- `__init__()` 初始化预期为可迭代的数据属性；
- `__iter__()` 返回迭代器对象。在我们的示例中，这是在其本身上调用的 `Data` 对象，在实例上使用 `iter()`。我们将 `current_index` 初始化为零，以从第一个数据开始；
- `__next__()` 返回一次迭代后的下一个值。我们增加 `current_index` 属性以跟踪 `data` 中元素的当前索引。这个特殊方法在实例上调用 `next()` 时被调用。

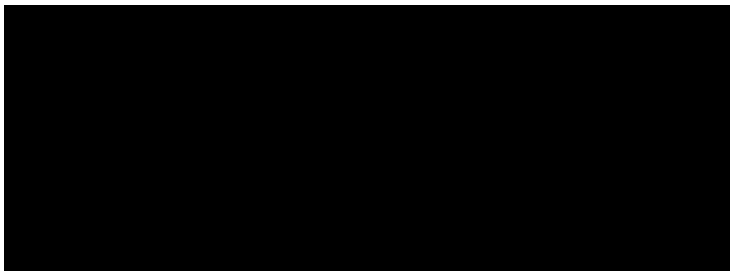

图 7: `__iter__()` 和 `__next__()` 特殊方法

4.3.5 属性 (Property)

属性可以被认为是使用属性的"Pythonic"方式，这在数据管理（5.3.4 小节）中大量存在，因为如 2.2 节所示，PINNs 的数据集是清晰表达的。属性的主要优势如下：

- 您可以像访问公共属性一样访问实例属性，同时使用中介（getter 和 setter）来验证新值并避免直接访问或修改数据；
- 通过使用 `@property`，您可以重用属性的名称以避免为 getter、setter 和 deleter 创建新名称，因此语法简洁易读。

在图 9 中，我们用一个基本示例展示了使用此装饰器定义自定义函数的语法。

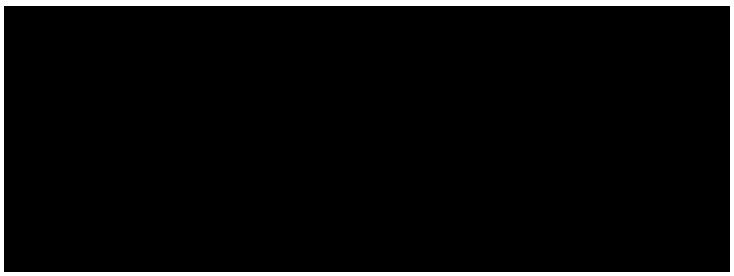

图 8: 使用 `@property` 装饰器的示例

4.4 仓库结构 (Repository Structure)

仓库的详细描述从本节开始，深入探讨文件夹 `config`, `data` 和 `outs` 的目的和内容。包含实际实现的文件夹 `src` 的介绍推迟到第 5 章。

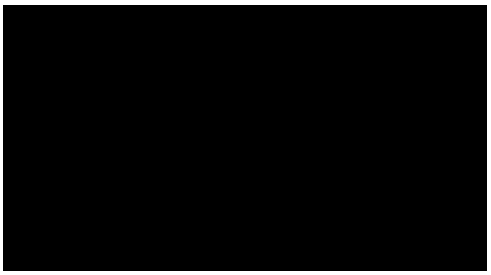

图 9: PACS-BPINNS 仓库

4.4.1 配置 - config 文件夹 (Configuration - config Folder)

此文件夹包含用于配置测试用例选项的 .json 文件，这些是主可执行脚本读取的第一批指令，包含有关算法参数、网络架构、数据集选择或生成以及各种实用程序的信息。这些文件分为两个主要的子文件夹：

- test_models，其文件是正在进行的测试用例；
- best_models，包含产生有趣结果的测试用例的选项。

除了成员文件夹的区别外，所有配置文件都具有相同的结构：它们存储字典的 json 字典，其中第一级键代表要选择的不同设置类别，而第二级键由要设置的变量名称组成。描述选项集群的第一级键是：

1. general，存储：
    - problem，带有要解决的问题名称的字符串（例如 Regression 用于函数插值，或物理信息任务的 PDE 名称）；
    - case_name，带有标识实验特定数据集名称的字符串；
    - method，包含要选择的训练算法的字符串；
    - init，为空字符串表示没有预训练，或包含所需预训练方法的名称。
2. architecture，包含有关网络架构的信息：
    - activation，带有所需激活函数的字符串；我们选择了 *swish* 函数；
    - n_layers，隐藏层数 L；
    - n_neurons，每隐藏层的神经元数 K_l。
3. losses，包含每个部分损失的布尔值，以包括在用于学习的总损失中；
    - data_u，表示解拟合的分量（方程 16）；

- `data_f`，表示参数场拟合的分量（方程 18）；
- `data_b`，表示边界损失的分量（方程 17）；
- `pde`，表示 PDE 残差的分量（方程 23）；
- `prior`，表示网络参数的先验分布（方程 19）。

4. `metrics`，包含与 `losses` 相同的键，以建立我们想要计算（无论我们是否将其用于学习）并绘制其历史记录的损失函数的哪个分量。

5. `num_points`，包含要使用的数据点数：

   - `sol`，用于解拟合点；
   - `par`，用于参数场拟合点；
   - `bnd`，用于边界点；
   - `pde`，用于配置点。

6. `uncertainty`，包含与 2.4 节中提到的噪声和不确定性相关的 std 值：

   - `sol`，解的 σ_u；
   - `par`，参数场的 σ_f；
   - `bnd`，边界数据的 σ_b；
   - `pde`，PDE 残差的 σ_r。

7. `utils`，包含：

   - `random_seed`，设置的随机种子；
   - `debug_flag`，布尔值，用于打印对调试任务有用的详细信息；
   - `save_flag`，布尔值，用于将正在进行的测试保存在 `outs` 中的特定文件夹中（参见 4.4.3 小节）；
   - `gen_flag`，布尔值，用于生成请求的数据集；仅当数据集尚未存在于 `data` 文件夹中时才需要将其设置为 true。

8. `{method_name}`[7]，存储训练算法参数的字典。

9. `{method_name}_0`[8]，存储用于预训练（可选）的方法参数的字典。

最后两个字典的键是方法名称；后两个字典的第二级键取决于所选的方法；对于项目中实现的算法，我们有：

[7] 字典的名称是实现的算法之一：`ADAM`, `HMC`, `VI`, `SVGD`.

1. `ADAM` (算法 3)

   - `epochs`，epoch 数 N；
   - `burn_in`，学习过程中物理损失的延迟；
   - `lr`，学习率 η。
   - `beta_1`，一阶矩估计的指数衰减率 β_1；
   - `beta_2`，二阶矩估计的指数衰减率 β_2；
   - `eps`，防止被零除的小数 ϵ；

2. `HMC` (算法 4)

   - `epochs`，样本总数 N；
   - `burn_in`，burn-in B 和学习过程中物理损失的延迟；
   - `skip`，在后续样本中跳过的样本数 S；
   - `HMC_L`，leap-frog 步数 L；
   - `HMC_dt`，时间步长 Δt；
   - `HMC_eta`，质量矩阵参数 α。

3. `VI` (算法 5)

   - `epochs`，epoch 数 N；
   - `burn_in`，学习过程中物理损失的延迟；
   - `samples`，所需的 $\boldsymbol{\theta}$ 样本数；
   - `alpha`，学习 $\boldsymbol{\zeta}$ 时的学习率。

4. `SVGD` (算法 6)

   - `epochs`，epoch 数 E；
   - `burn_in`，学习过程中物理损失的延迟；
   - `N`，粒子总数 N；
   - `h`，核定义中的带宽 h；
   - `eps`，步长 ϵ。

4.4.2 数据集 - data 文件夹 (Datasets - data Folder)

此文件夹包含为建议的测试用例生成的数据，存储在 `.npy` 文件（NumPy 数组文件）中。每次 4.4.1 小节中引用的 `gen_flag` 设置为 `True` 时，都会生成或覆盖当前测试用例的文件夹。如果不需要生成数据，而是从外部来源给出或已经创建，用户需要将 `gen_flag` 设置为 `False`，并且在第一种情况下，按照已创建数据的相同命名约定上传文件。如图 11 所示，数据按问题类型分隔在以任务命名的文件夹中（对于 `Regression` 案例），或物理信息问题中的微分方程（`Laplace1D`, `Oscillator`……）。然后，可以找到第二级文件夹，我们在其中存储该问题的特定数据集，以区分例如具有不同函数的 Laplace 1D 问题。

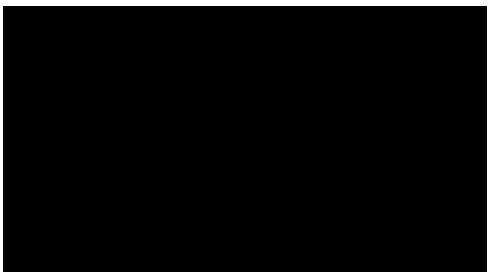

图 10: 文件夹 `data` 的层次结构。

在每个子文件夹内，我们最终找到数据，分成不同的文件：

- **边界数据 (Boundary data)**：`dom_bnd.npy`, `sol_bnd.npy` 用于边界的输入和解值；
- **配置数据 (Collocation data)**：`dom_pde.npy` 用于配置点的输入；
- **拟合数据 (Fitting data)**：`dom_sol.npy`, `sol_train.npy` 和 `dom_par.npy`, `par_train.npy` 用于解或参数场的输入和值；
- **测试数据 (Test data)**：`dom_test.npy`, `sol_test.npy`, `par_test.npy` 用于测试点的输入和值。

4.4.3 输出 - outs 文件夹 (Outputs - outs Folder)

此文件夹包含每个测试用例的结果和摘要统计信息，文件夹层次结构（见图 12）与 `data` 文件夹类似：第一级表示问题类型，第二级通过指定数据集来完全表征测试用例。还有一层缩进：在每个子文件夹内，每次将选项 `save_flag` 设置为 `True` 运行代码时，都会生成一个文件夹，其名称包含执行的日期和时间。`trash` 文件夹包含 `save_flag` 等于 `False` 时最新测试用例运行的输出，并且每次都会被清理。

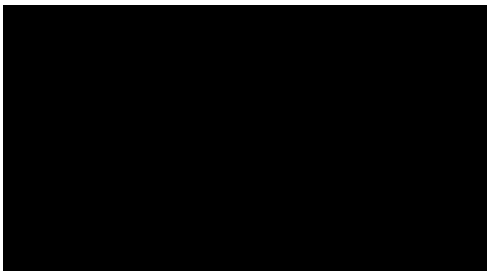

图 11: 文件夹 outs 的层次结构。

在每个测试用例文件夹内，有以下子文件夹：

1. log，存储 .txt 文件：

   - parameters.txt，重现 .json 配置文件；
   - errors.txt，重现每次运行后屏幕上的输出，包含解（及最终的参数场）的相对误差和不确定性量化，
   - keys.txt，包含学习过程中使用的损失分量；

   以及 .npy 文件：

   - loglikelihood.npy，对数后验值的历史记录（直到一个常数）；
   - posterior.npy，后验值的历史记录（直到一个常数）；

2. plot 存储图形输出。可以找到的文件有：

   - u_confidence.png 解的预测分布的均值，置信区间由均值 ± 标准差给出；
   - u_nn_samples.png 解的预测分布的几个样本的图；
   - f_confidence.png 参数场的预测分布的均值，置信区间由均值 ± 标准差给出；
   - f_nn_samples.png 参数场的预测分布的几个样本的图；
   - Mean Squared Error.png，MSE 的图；
   - Loss (Log-Likelihood).png，对数后验的图（直到一个常数）；

3. thetas,存储所有采样的网络参数。对于每个参数样本,创建一个名为 theta_(number) 的子文件夹，包含层数两倍的 .npy 文件：实际上每个文件存储每层权重或偏置的值；

4. values，包含：

- `sol_NN.npy` 解的预测分布的均值;
- `sol_std.npy` 解的预测分布的标准差;
- `par_NN.npy` 参数场的预测分布的均值;
- `par_std.npy` 参数场的预测分布的标准差;
- 子文件夹 `samples`，在 `.npy` 文件中包含对应于每个选定网络参数样本的解和参数场的值。

5 源代码 (Source Code)

在本节中，我们深入探讨源代码，它与配置、数据和输出分离，并存储在 `src` 文件夹中。该文件夹的概述如图 12 所示：它包括三个可执行脚本和特定模块，用于数据生成和预处理、B-PINNs 方法的实现、性能和 UQ 评估以及可视化和保存等后处理实用程序。在代码开发中，我们建立并遵循了三个通用准则：

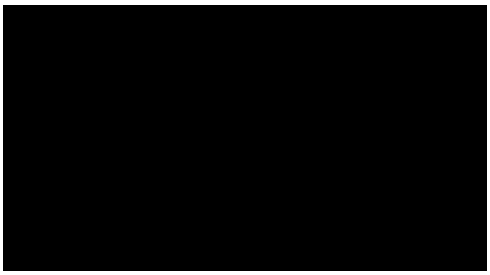

图 12: src 文件夹的层次结构。

5.1 I - 在类层次结构中反映理论框架

B-PINNs 的理论表明模型应具有模块化结构，因为 B-PINN 可以通过将模块相互叠加来获得：

- 从全连接神经网络开始；
- 添加非确定性训练算法以使其具有贝叶斯特性；
- 在损失中包含 PDE 以使其具有物理信息。

通过这种方式，可以选择组装哪些模块以实现模型的整个层次结构，例如 BNN 或 PINN，并将其与 BPINN 进行比较。上述三个组件通过归属到不同的构建块来突出显示：总体思路是从 `CoreNN` 类开始构建最终模型，其中包含基本的神经网络特征（初始化、前向传播步骤）；在此基础上，我们通过子类 `PhysNN` 引入物理信息，然后从 `PhysNN` 创建 `BayesNN`。此外，侧面子类致力于特定的功能，例如损失函数上的操作 (`LossNN`) 以及与不确定性量化指标计算相关的所有内容 (`PredNN`)。

5.2 II - 促进代码重用和扩展

由于项目的主要目标是为贝叶斯机器学习库奠定基础，我们注意选择足够灵活的架构，以便代码可以重复用于类似任务。一个直接的例子是努力实现训练算法（无论是贝叶斯还是确定性）的网络便捷应用：为了避免代码重复，我们设法将训练过程的主要结构共享在一个抽象类中，然后将差异限制在特定方法中，使其进入各种算法类，而无需

重复算法如何作用于网络或网络如何存储和升级参数。我们还开发了对不同训练算法通用的特定功能，例如网络参数集合的代数运算重载（使用 `Theta` 类），这对于 SVGD 实现中的可读紧凑代码至关重要。至于添加扩展，这种结构足够灵活，可以实现新的训练算法：这已在开发所提出的四种训练算法时得到评估。实际上，正如第 3 节所解释的，算法在执行步骤、参数选择和涉及的结构方面存在显著差异，但所提出的框架能够考虑每个算法的特殊性，避免了代码重复。

5.3 III - 抽象于单一应用

在这个项目中，我们提出了一些用于验证该方法的测试用例，旨在展示贝叶斯物理信息深度学习的主要特征，但代码首先是为了限制单一应用对核心结构的影响而设计的。根据应用而变化的是域的维度和输出空间（因此是解和参数场的组件数量）。代码执行直到误差计算和 UQ 阶段，对于所有输入和输出维度都是相同的。然后变化的是可视化阶段，为此我们必然需要根据维度调用不同的视觉策略。

5.4 主要文件 (Main Files)

如 `src` 文件夹中图 12 所示，用户可以找到 `.py` 可执行脚本。

- `main.py` 这是主可执行脚本，因为它执行使用 B-PINN 解决问题的任务。由于所有特定任务都在单独的脚本中执行，因此 `main` 的工作是按照下面解释的正确顺序调用它们。
- `main_data.py`；通过此脚本，用户可以生成新数据集而无需训练 B-PINN。新数据集存储在 `data` 文件夹中的新子文件夹中。
- `main_loader.py`；通过此脚本，用户可以加载先前保存的测试用例的输出。用户可以从终端读取可用的测试用例，并提示选择问题和数据集。选择完成后，所有图，即 `outs` 中 `plot` 文件夹的内容，都将被打开。

5.4.1 主要管道 (Main Pipeline)

`main.py` 脚本呈现了我们的工作管道，可以分为实现各种任务的几个部分：

1. **导入和设置 (Import and Setup)**：导入用于打印和路径管理的宏，然后加载 `src` 文件夹中的所有模块，并选择配置文件。
2. **参数创建 (Creation of Parameters)**：在这里，配置文件 (`.json`) 中的信息与命令行参数集成或更新，并全部存储在 `Param` 对象中。

3. **数据集创建 (Creation of Dataset)**：如果需要生成新数据集，则调用特定测试用例的数据生成器；否则，加载合适的数据集。然后执行训练所需的预处理。

4. **模型构建 (Model building)**：使用 `Param` 对象的参数初始化 `Equation` 和 `BayesNN` 对象。

5. **模型训练 (Model training)**：初始化优化算法并对先前创建的 `BayesNN` 进行训练（如果需要，还包括预训练），使用已经预处理的数据集。

6. **模型评估 (Model evaluation)**：使用测试集输入数据计算 UQ 的统计信息，然后通过与测试数据集真实值进行比较来评估模型性能。

7. **保存 (Saving)**：存储结果和相关测试用例的详细信息，生成如 4.4.3 小节所述的文件夹，或将所有内容临时存储到 `trash` 文件夹中。

8. **绘图 (Plotting)**：然后由 `Plotter` 类读取存储的结果并显示在屏幕上。

5.5 参数处理 (Parameters Handling)

测试用例选项信息的管理包含在 `setup` 模块中，该模块在其功能中提供了用于最终确定案例研究设置完整列表的类。此指令集的最终确定需要整合来自不同来源的信息，因为我们将与测试用例相关的细节（例如方法和模拟选项）与更具体地与数据集相关的细节（例如域和涉及的函数）分开了。此外，用户可以通过命令行参数直接指定选项，因此需要一种方法来更新它们。

5.5.1 配置文件 (Configuration files)

在主脚本开始时，我们加载来自不同来源的信息，如上所述，其中大部分是 4.4.1 小节中描述的配置文件。它们以 `json` 格式编写并导入到 Python 字典中。每个文件代表一个特定的测试用例，并使用来自指定 `data` 文件夹的数据，存储收集这些数据所需的参数。尽管大多数参数由配置文件提供并且必须在那里修改，但我们也提供了一种更快的方式，通过命令行参数对当前测试用例进行微小的更改。

5.5.2 命令行参数 (Command line arguments)

在代码中，我们包含了通过命令行指定每个测试用例和方法参数的可能性，以更新 `.json` 配置文件中已有的默认值。为了保证此功能，我们实现了 `Parser` 类：它继承自 Python 内置类 `ArgumentParser`，能够将命令行字符串解析为 Python 对象。此 类没有任何方法，因为它扮演一个信息容器的角色，我们不需要对其进行任何主动操作；因此，它只有一个构造函数，我们通过内置方法 `add_argument()` 添加命令行参数。

5.5.3 参数对象 (Param object)

`Param` 类旨在包含测试用例的所有用户定义选项的完整集合；它的构造函数确实需要配置文件内容和前面提到的 `Parser` 类处理的命令行参数。除了从这两个来源读取测试用例参数外，该类还负责将它们重新格式化为更实用的版本，例如将布尔值字符串转换为布尔值，或将预训练与训练选项分开。为了在此类的对象中全面收集方法详细信息，在此类中还包含了有关数据集的信息（网格、子域、函数...）；虽然测试用例信息直接存储在类属性中，但数据集信息存储在属性 `data_config` 中，其自定义 setter 能够将数据集对象的内容存储为 `datasets` 中以合适类属性形式呈现的内容。此对象唯一目的是将其传递给代码的所有高级特性和类，以向它们提供所有必要的信息和参数，以便进行所有处理。

5.6 数据生成 (Data Generation)

`setup` 模块中的 `DataGenerator` 类致力于生成 `data` 文件夹的内容：`.npy` 文件存储空间坐标以及解和参数场的精确值。所有文件的创建都是通过私有方法 `create_domains()` 执行的，该方法包括生成配置、拟合、边界和测试数据。

5.6.1 域创建 (Domain creation)

用于生成域的基本工具是私有方法 `create_domain()`，它在给定极值的情况下，可以在域上构建特定网格类型。可用的类型有：

- **均匀域 (Uniform domain)**，其中点布置在规则网格的角落；这种类型的网格已用于测试点，以确保在所有区域上的验证并生成清晰的图。但是，对于拟合和配置，这种网格类型不适合。

考虑在处理周期函数时，如果只要求少量点会发生什么：如果我们将点放置在网格上，我们可能会运气不好，得到对应于相同函数输出的输入，这表明网络认为函数是常数。相比之下，对于测试数据，我们通常没有这种风险——这不仅因为它们不用于学习，还因为它们通常数量非常大。

- **随机域 (Random domain)**，其中点是从域内部随机采样的。然而，正如图 14 所强调的，这种选择不能保证域的良好覆盖，例如在回归任务中，域中的孔洞可能会对预测质量产生影响。这种限制可以通过实现以下伪随机点集来克服；
- **Sobol 域 (Sobol domain)**，它由通过准蒙特卡洛 (QMC) 方法家族中的常用技术生成的纯确定性采样点组成，这有助于在不使结构复杂化的情况下提高蒙特卡洛估计器的收敛性。这种点集的特点是其低差异性，在不深入其数学定义的情况下，低差异性是点均衡分布的度量。

为了定义 Sobol 点，我们首先需要引入低差异点的一般类别，即 t, m, d 网格和 t, d 序列，我们在 d 维空间中定义它们。

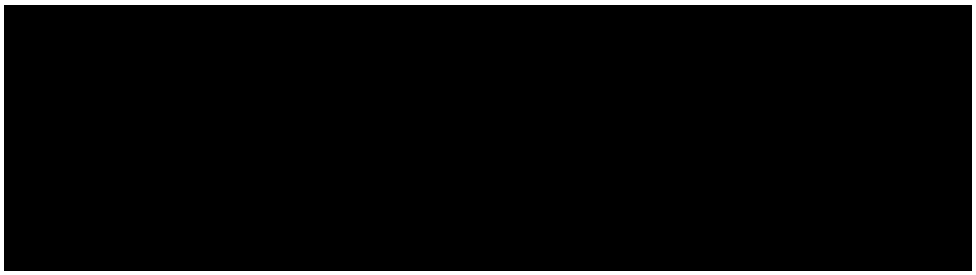

图 13: 随机采样和 Sobol 点集的比较。

Sobol 点集是基数 $b = 2$ 的 (t, d) 序列的特殊类型；注意 Sobol 序列是针对维度 n（2 的幂）的样本优化的，如果不能遵守此选择，它们会失去平衡特性。实际上，为了在代码中生成点序列，我们依赖于 `scipy.stats` 的 `qmc` 模块，它提供了一个 Sobol 序列的生成器，生成在单位超立方体中。在提供的绘图选项中，我们选择了 `random_base2()`，它安全地绘制 $n = 2^m$ 个点，保证了序列的平衡特性。然后，我们添加了序列的正确边界（这些边界未包含在样本中），并将样本从单位超立方体重新缩放到我们的一般域。

5.6.2 多域创建 (Multi-domain creation)

对于拟合数据，表示测量值，我们包含了将点定位到域的某些特定部分的可能性，以便模拟实际测量情况，其中定位数据受限于定位传感器的可行性。事实上，在实际域中，可能存在一些无法到达的区域，例如它们可能位于边界附近。具有局部化数据的测试用例也可以突出显示来自对底层 PDE 知识的贡献，该知识在数据不可用的区域中是可用的。请注意，在我们对配置点的实现中，我们假设 PDE 在整个域中都成立，因此不包括多域的创建；但是，如果我们要表示仅在域的一部分中成立的物理定律，后者仍然可以通过微小更改来实现。子域中的数据是通过私有方法 `create_multidomain()` 生成的，该方法从数据集配置文件中读取每个子域的极值，在子域之间平均分配点数，并通过迭代调用前面提到的方法 `create_domain()` 来创建子域列表。然而，应该考虑一个额外的细节：我们希望以这样的顺序定位最终数据结构中的点，即当点数少于可用分辨率时，子集中的点在子域之间均匀分布，例如避免仅覆盖第一个区域。为此，我们实现了辅助私有方法 `merge_2points()`，它将两个点序列交替合并为一个。

5.6.3 边界点 (Boundary Points)

边界上数据的生成需要不同的策略，因为它由一个维度比物理维度少一个的域组成，该域存在于 R^d 中。实现依赖于将 d 维域投影到边界的每个边缘上的想法：显然，此策略仅限于矩形域。

5.6.4 数据配置 (Data Configuration)

此模块致力于数据集的配置：其类中包含的信息在某种程度上与配置文件中的信息互补，因为这里关注的不是训练方法，而是生成测试用例所需的所有信息。在这里，您可以找到生成独立数据集的所有规范，例如计算域、网格和分辨率选项、PDE 中出现的系数以及问题中涉及的函数的表达式。信息以层次结构的方式组织在类中，其规范级别逐渐递增，如 Laplace 案例图 14 所示。此文件夹中提出的所有类都只旨在存储数据，因此我们使用装饰器 `@dataclass`（在 4.3.3 小节中介绍）对它们进行装饰。

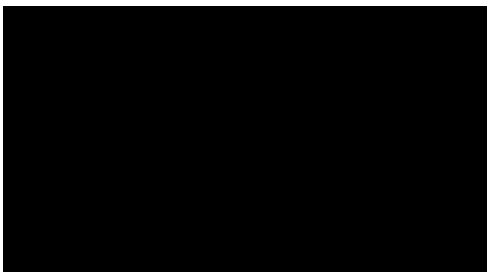

图 14: 数据集配置类的继承树。

模板 (Templates) 此处提出的模板负责固定结构，其细节可以在不同的测试用例中指定。我们创建了一个抽象基类 `Data_Config`，它充当各种数据集配置类的蓝图，所有这些类都共享域和分辨率定义等属性。然后，我们为分析的各种问题（`Laplace`, `Oscillator` 等）实现了子模板，其中包含指定的问题名称和问题参数的字典。从这些模板中，我们获得了表示问题和分析维度的最终模板，为空间输入、解和参数场分配了正确的维度。

配置 (Config) 为每个问题和维度的规范构建模板后，我们为特定测试用例生成了子类，这些子类可以对应于所涉及函数的特定选择或 PDE 系数的特定值。子模块的每个类都有一个 `values` 属性，返回一个包含解和参数场的 lambda 表达式的字典。在属性中，我们突出显示 `domains`，它包含数据和解和参数场数据集中子域的规范。这些类的实例表示数据集配置，然后将其设置到 `Param` 的 `data_config` 中。

5.7 预处理 (Pre-Processing)

除了数据生成，`setup` 模块还负责另外两项与数据集相关的任务，这些任务能够按顺序组装训练 PINNs 所需的关节数据集。一方面，我们确实需要组装用于训练网络的实际数据集，这需要根据 2.2 节中描述的类别进行转换和组织。这些任务在 `data_creation` 中执行。另一方面，为了进一步改进训练数据，我们还可以将它们组织成训练批次；此功能在 `data_batcher` 中实现。

5.7.1 数据集创建 (Dataset creation)

`Dataset` 类致力于管理 `data` 文件夹中通过 `data_generation` 生成的数据，这些数据代表用于组装网络所需数据集的库存材料。为了准备它们供模型使用，例如，数据可以通过任意转换进行预处理，从而提高模型性能，或者如果我们要使用模糊的量测量训练网络，可以通过添加一些噪声来修改数据。因此，我们引入了真实数据和合成数据之间的差异，即用于模型训练的转换数据。所有这些转换都由 `Dataset` 类的方法执行，最终代表训练和测试过程可访问的数据集。

数据选择 (Data selection) 在配置文件中或通过命令行，用户可以指定用于训练的数据量（特别是用于配置、拟合和边界）：如果所需数量 N 小于最大可用域分辨率，我们需要从 `data` 文件夹中的文件选择数据点。请注意，在选择器属性 (`data_pde`, `data_bnd`, `data_sol`, `data_par`) 中，我们只取完整数据结构的前 N 个元素的一个切片，因为在生成过程中，我们已经确保了存储机制能够通过这种直接切片数组的方式获得整个域的代表性样本。

噪声添加 (Addition of noise) 存储在 `data` 文件夹中的解和参数场的值对应于它们的精确值；通过这种选择，在噪声水平变化的实验中，我们不需要重新生成整个数据集。当用户要求使用带噪声的数据时，它会通过命令行或配置文件指定噪声水平（高斯噪声的标准差）：私有方法 `add_noise()` 的任务是生成具有所需噪声水平的数据集，从输入中读取。

5.7.2 数据归一化 (Data normalization)

公共方法 `normalize_dataset()` 和 `denormalize_dataset()` 负责归一化转换，这对于模型性能至关重要。实际上，在验证阶段，我们注意到从非归一化数据中出现了不准确的结果，因为即使在回归任务中，网络在尝试同时重建两个不同值范围的函数时也表现出困难。因此，我们通过在归一化版本的数据上训练模型，暂时将它们带到相同的数量级；采用的策略是通过减去均值并除以标准差 σ 来归一化解和参数场的值：

$$u_{norm} = \frac{u - \mu_u}{\sigma_u} \quad f_{norm} = \frac{f - \mu_f}{\sigma_f}$$

归一化统计量 μ 和 σ 是在读取 `data` 文件夹中的数值后计算的，然后存储在字典类属性 `norm_coeff` 中。然后利用它来反归一化预测，将其恢复到真实范围。归一化对 PDE 进入模型也有影响：首先，我们需要以归一化形式重写 PDE，因为网络只看到归一化数据，因此需要学习一个适合的缩放物理定律，该定律由转换后的参数集 A_{norm} 描述。

5.7.3 数据批处理 (Data Batching)

在机器学习模型中加载数据集时，通常将其分成批次。这种策略包括不一次性将整个数据集加载到内存中，而是一次只加载一个样本集。由此带来的优势在于速度：事实上，当处理大型数据集时，使用整个数据集评估损失在计算上可能是昂贵的，而多次评估损失但每次使用较少数据可能会更好。因此，我们在代码中实现了用批次训练的可能性：`BatcherDataset` 类执行此任务，因为它扮演一个数据集的角色，具有在 `data_creation` 中提到的数据集的相同属性，代表按所需批次数量 N_{batch} 划分和组织训练数据。在类构造函数中，从长度为 N_{tot} 的数据集开始，我们为每种类型的训练点构建大小为 $\lceil N_{tot}/N_{batch} \rceil$ 的批处理器；每个类别都由一个 `Batcher` 对象表示，它是一个存储数据值和原始数据集索引的类。后者存储在实现随机分割 N_{tot} 索引到批次的 `BatcherIndex` 类的实例中。我们通过调用 `next()` 方法遍历所有批次：由于此方法在 4.3 小节中声明，调用对象的 `__next__()` 方法，因此在所有三个批处理相关类中，您都可以找到此私有方法的重写。实际上，即使在每个训练步骤开始时我们只调用 `BatcherDataset` 对象的 `next()`，该方法也依赖于 `Batcher` 类的 `next()` 方法，而 `Batcher` 又会内部调用 `BatcherIndex` 上的 `next()`。

5.8 方程 (Equations)

模块 `equations` 致力于计算物理信息机器学习上下文中所 需 的 PDE 残差；正如 2.2 节所示，这些量有助于对数似然的值，并且由于我们处理的是微分方程，因此需要工具来区分网络重建的函数。在 `equation` 模块中，我们实现了两个主要功能：

- 一个模块，能够评估标量、向量和张量上的微分算子，然后用于计算 PDE 残差的方法中；
- 分析方程的特定类，其实例被设计为网络的类属性，特别将放置在致力于物理信息的部分 (`PhysNN`) 下。

5.8.1 算子 (Operators)

`Operators` 类包含微分算子的静态方法和用于张量操作的辅助静态方法，主要旨在以可移植的方式重塑张量。我们尝试依赖已编码的基本算子来实现算子；我们从 `gradient_scalar` 的基本实现开始，它计算标量函数的梯度。从这个操作中，我们能够实现 `gradient_vector`，它使用列表推导对每个函数分量调用标量版本。对于向量的散度，我们将其计算为表示其梯度的张量的迹，而张量的散度则在列表推导中调用向量版本。最后，标量的拉普拉斯算子是通过梯度的散度获得的，而向量的拉普拉斯算子我们再次依赖标量版本和列表推导。

算子	版本	解析表达式	输入形状	输出形状
梯度	标量	$\nabla u = [\frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_{N_{in}}}]^T$	$N_s \times 1$	$N_s \times N_{in}$
	向量	$\nabla \mathbf{u} = [\nabla u_1, \ldots, \nabla u_{N_{out}}]^T$	$N_s \times N_{out}$	$N_s \times N_{in} \times N_{out}$
散度	向量	$\mathrm{div}\mathbf{u} = \frac{\partial u_1}{\partial x_1} + \cdots + \frac{\partial u_{N_{in}}}{\partial x_{N_{in}}}$	$N_s \times N_{in}$	$N_s \times 1$
	张量	$\mathrm{div}\mathbf{U} = [\mathrm{div}\mathbf{U}_1, \ldots, \mathrm{div}\mathbf{U}_{N_{out}}]^T$	$N_s \times N_{in} \times N_{out}$	$N_s \times N_{out}$
拉普拉斯	标量	$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \cdots + \frac{\partial^2 u}{\partial x_{N_{in}}^2}$	$N_s \times 1$	$N_s \times 1$
	向量	$\Delta \mathbf{u} = [\Delta u_1, \ldots, \Delta u_{N_{out}}]^T$	$N_s \times N_{out}$	$N_s \times N_{out}$

表 2: 类 `Operators` 提供的微分算子概述。

在表 2 中可以找到可用算子的摘要，并指定了输入和输出形状。采用的符号如下：

- N_s 是函数被评估的样本数；
- N_{in} 是输入空间的维度；
- N_{out} 是输出空间的维度；
- u 表示标量函数；
- $\mathbf{u}$ 表示向量值函数；
- $\mathbf{U}$ 表示张量值函数。

5.8.2 抽象方程 (Abstract Equation)

`Equation` 类是一个抽象基类，代表了生成各种 PDE 约束的子类的通用蓝图（图 15）。此类包含存储问题物理维度和其系数信息的属性，以及一个抽象方法 `comp_residual()`，用于计算方程的残差，并在每个子类中重写。

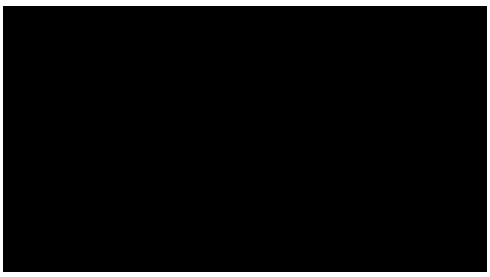

图 15: 方程类的继承树。

5.8.3 已实现问题 (Problems Implemented)

我们现在深入探讨已实现的问题：第一个是回归应用案例，不涉及任何微分模型，而后面的案例代表了我们同时拥有数据和潜在 PDE 知识的物理现象。

回归 (Regression) `Regression` 子类的对象在涉及简单回归任务（即重建插值一些给定点的函数）的测试用例中实例化。在这种情况下，不需要计算 PDE 的残差，因此只保留了关于维度的属性；`comp_residual()` 方法只会引发异常，以突出显示在回归任务中（错误地）调用它时出现的错误。

拉普拉斯 (Laplace) `Laplace` 子类实现了椭圆泊松/拉普拉斯 PDE 残差的计算，其扩散系数为 $\mu \in \mathbb{R}$，读作：

$$-\mu \Delta u = f$$

然而，网络重建的函数必须满足的方程与描述物理问题的方程略有不同，因为它必须在数据集归一化后进行调整，如 5.4.1 小节所述。归一化信息存储在类属性 `norm_coeff` 中，它包含归一化系数，这些系数在预处理期间通过数据集中的信息强制到方程对象中。然后，在有了归一化系数之后，`comp_residual()` 方法计算归一化函数的方程残差，读作：

$$-\Delta u_{norm} = f_{norm} + \frac{\sigma_u}{\mu \sigma_f} f_{norm}$$

振荡器 (Oscillator) `Oscillator` 子类实现了阻尼谐振子的残差计算。谐振子是一个系统，其质量为 m，当其偏离平衡位置时，会受到一个与质量位移 x 成比例的恢复力 $F = -kx$ 的作用，其中 $k > 0$ 是一个常数，例如可以代表弹簧的弹性常数。回忆牛顿第二定律，其中 $F = ma = m\frac{d^2x}{dt^2}$，我们得到：

$$m\frac{d^2x}{dt^2} + kx = 0$$

微分方程 41 的解描述了一个周期性运动，以恒定振幅 A 正弦式重复：

$$x(t) = A\cos(\omega t + \phi)$$

其中 $\omega = \sqrt{\frac{k}{m}}$ 和 ϕ 称为相位，决定了正弦波的起点。周期和频率由质量 m 和力常数 k 决定，而振幅 A 和相位 ϕ 由起始位置和速度决定。系统也可能受到摩擦力，摩擦力与速度 $\frac{dx}{dt}$ 成比例；在这种情况下，我们得到了阻尼谐振子，它是代码中实现的应用之一。将摩擦常数设为 c，力平衡方程读作：

$$F = -kx - c\frac{dx}{dt}$$

再次，牛顿第二定律使我们能够将 42 重写为

$$m\frac{d^2x}{dt^2} + 2\delta\frac{dx}{dt} + \omega^2 x = 0$$

其中 $\delta = \frac{c}{2m}$。在这种情况下，振荡的振幅随时间衰减，位置 x 的演变读作：

$$x(t) = Ae^{-\delta t}\sin(\psi t)$$

其中 ψ 通过关系 $\psi = \sqrt{\omega^2 - \delta^2}$ 与 ω 和 δ 相关。这个问题与之前的问题不同，因为它是一个时间相关的 ODE，所以它可能看起来比之前的 PDE 案例更简单，但它仍然是展示 PINN 强大应用的一个非常有用的方式。事实上，这个问题的一个明显配置是将时域分成两部分：一部分有数据测量，另一部分没有任何线索，然后试图通过问题背后的物理知识来重建这部分，如 6.2 节所示。

5.9 模型 (Model)

在本节中，我们介绍 `networks` 模块的内容，该模块致力于所有与神经网络功能相关的内容。所采用的策略是在父类 `CoreNN` 之上构建子类，`CoreNN` 只包含基本的神经网络功能：初始化、权重和偏置的存储、给定参数的前向传播步骤。在此基础上，我们构建了子类 `PhysNN`，其主要作用是包含来自物理的信息和归一化常数（这些常数反过来可以是问题特定的）。从 `PhysNN` 派生出两个子类：`LossNN`，用于损失和损失梯度计算；`PredNN`，其方法是用于预测和测试的功能，因为它包含不确定性量化和误差计算函数。我们在主脚本中实例化对象的类是 `BayesNN`，它结合了所有祖先类的特性并管理训练期间的损失历史。上面描述的继承链如图 16 所示，并与 4.3 小节中描述的工具一起工作。除了网络类之外，此模块还包含一个 `Theta` 类，旨在处理神经网络参数（权重和偏置），并包含对 `Theta` 对象的运算重载和算法中使用的少量循环方法。

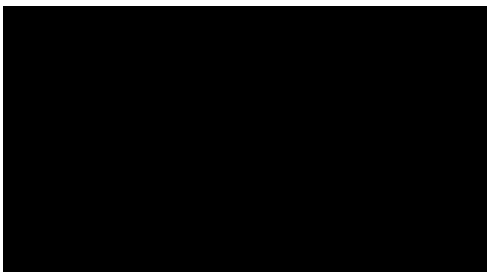

图 16: 网络类的继承树。

5.9.1 参数对象 - Theta (Parameters object - Theta)

此类用于生成代表 NN 参数 $\boldsymbol{\theta} := \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$ 的对象。将特定类用于它们的选择基本上是为了通过重载它们的循环代数运算来提高代码可读性。事实上，在训练算法中，

通常需要对 NN 参数执行代数运算，但 TensorFlow 中默认表示它们的数据结构不允许简单使用它们。此结构是不同形状张量的列表，其长度是层数的两倍；列表中每个元素分别是一个张量，包含每层的权重或偏置。注意，对于张量，最常见的代数运算已被重载，因此例如，您可以通过使用 + 运算符对张量进行逐元素求和。但对于张量列表，+ 运算符不会执行所需任务，因为 + 运算符已被重载以表示列表连接。重载主要算子的优点中，我们特别提到：

- 通过允许使用熟悉的算子，提高代码可读性；
- 保证类对象与内置类型和其他用户定义类型行为一致；
- 简化代码编写，尤其是在需要对复杂数据类型进行大量操作时；
- 通过实现链中定义的一个算子来增强代码重用。

类构造函数和属性 (Class constructor and properties) 构造函数要求 NN 参数，这些参数旨在通过 TensorFlow 的约定表示为张量列表。然后它们存储在属性 `values` 中。该类有两个属性：`weights`，包含权重自定义 getter 的实现，它从列表 `values` 中提取从第一个开始，步长为二的元素。属性 `biases` 表示相同，但对于偏置，它提取列表其余的元素。

运算重载 (Operation Overloading) 在 Python 中，有针对算子重载的特定方法名称，这些方法名称对于每个算子都是唯一的。因此，重载包括重写具有表 3 中所示名称的私有方法。当两个操作数都是 `Theta` 对象，或者其中一个操作数是标量整数或浮点数时，已经实现了求和、减法、乘法和除法。我们可以根据操作将我们重新定义的函数分为以下类别：

- **经典二元运算** $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$：返回一个 `Theta` 对象，其值是对一对 `Theta` 对象执行逐元素运算的结果；
- **多态二元运算** $(\boldsymbol{\theta}, n)$：作为 `Theta` 对象工作，所有条目都等于 n，作为第二个参数传入；
- **逆向多态二元运算** $(n, \boldsymbol{\theta})$：与之前相同，但更改参数位置。这是因为在重载时，如果二元运算的第一个参数无法处理它，就会引发错误，但可以通过第二个参数中的特殊方法捕获；
- **一元运算** $(\boldsymbol{\theta})$：实现很大程度上取决于任务。例如，通过标量 -1 的乘法实现相反数，幂运算应用于每个列表元素的 TensorFlow 幂函数。

此外，我们还实现了以下公共方法：

- `ssum()`，计算 `self.values` 中所有条目的平方和；
- `size()`，计算 `self.values` 中所有条目的数量；
- `copy()`，返回一个 `Theta` 对象的副本，其值是当前对象值的独立副本；
- `normal()`，创建一个带有随机正态初始化值的 `Theta` 对象。

在表 3 中，我们报告了所有已重载的运算。

运算	方法	符号	输入	输出
取反	`__neg__`		$\boldsymbol{\theta}$	$-\boldsymbol{\theta}$
加法	`__add__`, `__radd__`	$+$	$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1 + \boldsymbol{\theta}_2$
减法	`__sub__`, `__rsub__`	-	$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2$
乘法	`__mul__`, `__rmul__`	*	$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1 * \boldsymbol{\theta}_2$
除法	`__truediv__`, `__rtruediv__`	/	$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1 / \boldsymbol{\theta}_2$
幂	`__pow__`	**	$\boldsymbol{\theta}, 2$ $\boldsymbol{\theta}, 0.5$ $\boldsymbol{\theta}, -1$	$\boldsymbol{\theta}^2$ $\sqrt{\boldsymbol{\theta}}$ $1/\boldsymbol{\theta}$
类方法	`__len__` `__str__`	`len` `print`	$\boldsymbol{\theta}$ $\boldsymbol{\theta}$	$\text{len}(\boldsymbol{\theta})$ $\text{print}(\boldsymbol{\theta})$

表 3: `Theta` 对象的重载运算。

5.9.2 参数和前向传播 - CoreNN (Parameters and forward pass - CoreNN)

此类旨在：

- 存储与神经网络架构相关的属性；
- 存储和设置网络参数；
- 存储模型，即 `tf.keras.Sequential` 类的实例，该实例将线性层堆栈分组到一个模型中；
- 使用给定种子初始化模型；
- 执行前向传播步骤 `output = model(inputs)`。

构造函数和属性 (Constructor and properties) 该类的构造函数需要 par，即 Param 类的对象，用于检索有关输入/输出维度和网络架构的信息。它还通过调用私有方法 build_NN 初始化模型，并计算网络参数向量的维度。对于网络参数，我们引入了属性 nn_params，以便能够为它们定义自定义 getter 和 setter，以弥补模型和算法中网络参数表示之间的差距。getter 遍历 model.layers 列表，并在每层调用 TensorFlow 内置函数 get_weights() 将权重和偏置存储到两个单独的列表中。然后，将这两个列表合并为一个 thetas，其中权重和偏置交替存储。属性 setter 也执行相同的操作。最后，thetas 列表转换为 Thetas 对象，这是算法使用的数据结构。

主要方法 (Main methods) 私有方法 build_NN()，在给定种子的情况下，以随机正态初始化权重（与贝叶斯理论中权重的先验信息一致，详见 2.3 小节）和零初始化偏置（这是 NN 的常见做法）的方式初始化全连接神经网络。用于构建网络的构建块是继承自 tf.keras.layers 的类的实例，tf.keras.layers 是 TensorFlow 通用层蓝图；特别是，我们利用 Input 层和 Dense 层用于隐藏层和输出层。此方法具有公共接口 initialize_NN()，它通过传递种子作为输入来简单地调用它。公共方法 forward() 执行前向传播：它接受 Numpy 数组输入，将其转换为 Tensor，并向用户返回模型对其应用的 结果。此操作旨在逐元素执行。

5.9.3 物理强制 (Physics enforcement - PhysNN)

此类继承自 CoreNN，旨在包含处理物理信息的任务所需的所有部分，例如：

- 存储所选方程类（不是 Equation，而是一个子类）的实例。此对象代表物理信息进入问题，因为它包含计算偏微分方程残差的方法；
- 存储逆问题标志和 PDE 的 $\boldsymbol{\lambda}$ 参数，以供第 7 节中描述的未来开发；
- 在属性中存储问题归一化系数；
- 将网络输出分割为解和参数场。

构造函数和属性 (Constructor and properties) 此类构造函数需要 par（如 CoreNN 中）和 equation（即对应于测试用例的类对象）两个参数。我们在此上下文中使用 super() 来检索父类 CoreNN 的构造函数，我们将所需的 par 参数传递给它。在构造函数中，我们还为测试用例方程构建了一个对象，并将其存储在属性 pinn 中，同时存储逆问题标志并初始化归一化系数为 None。norm_coeff 属性表示解和参数场的归一化系数。

主要方法 (Main methods) 此类包含对父类 `forward` 方法的重写。我们再次使用 `super` 函数来检索父类 `forward`；然后，通过分离第一和最后 N_{out_sol} 个分量将其输出分割为解和参数场，其中 N_{out_sol} 是解的维度。这对于 `Regression` 也非常有用，因为它创建一个空的参数场并允许像所有其他情况一样解决问题。

5.9.4 训练功能 - LossNN (Training functionalities - LossNN)

此类继承自 `PhysNN`，旨在：

- 分别计算损失函数的所有组件；
- 根据测试特定的选项组合组件；
- 还计算损失函数中未出现的其他指标；
- 计算损失函数相对于网络可训练参数的梯度。

构造函数 (Constructor) 构造函数依赖于父类 `PhysNN` 的构造函数，采用与 `PhysNN` 相同的方法。属性 `vars` 是不确定性的字典，用于计算损失。然后，它将度量和键存储在属性中，这些属性是需要分别包含在损失计算中以进行历史监控和学习的对数后验的组件。实际上，只有在键中声明的组件对参数更新有影响，而度量中的组件不进入学习过程，只是被绘制出来。

损失组件评估 (Loss components evaluation) 在描述各种方法之前，重要的是要强调，损失函数和梯度评估所需的所有计算都必须以 TensorFlow `GradientTape` 中可以自动微分的方式编写（在 4.2.3 小节中介绍）。我们实现了一个私有方法 `mse()` 来计算适用于我们操作的张量的 MSE 式分数。当此方法在输出和目标之间的差异上调用时，它执行以下操作。输入张量与零张量之间的 MSE 计算作为真实值（事实上，差异的期望目标是零）。此方法附带了装饰器 `@staticmethod`，使该方法绑定到整个类而不是其实例。由于张量的 MSE 计算是一个脱离类的通用任务，因此允许用户在没有 `self` 参数的情况下调用该方法是合理的。`mse()` 函数的输出随后被 `normal_loglikelihood()` 利用，这是一个私有静态方法，它是计算具有相同结构项 28-31 的蓝图。然后，我们实现了 `loss_data()`，这是一个用于损失的拟合和边界组件的辅助私有方法。它包括计算目标和输出之间的 MSE 以及对数后验分量，并将用作我们需要调用 `mse()` 和 `normal_loglikelihood()` 的情况的通用蓝图。有了这个方法，我们可以在不同的数据集上调用它，使用私有方法 `loss_data_u()`、`loss_data_f()`、`loss_data_b()` 分别作用于解、参数场和边界数据。另一方面，`loss_residual()` 包含物理损失的计算，其中包括从 `pinn` 中检索偏微分方程残差。我们在 `tf.GradientTape` 内部执行前向传播步骤，让 `tape` 监视配置数据集。先验组件由 `loss_prior` 管理，它本质上执行与 `loss_data` 相同的任务（输出 MSE 和

对数后验分量）以用于先验分布。请注意，在这种情况下，高斯分布的 std 不是从 par 中读取的，而是从 CoreNN 继承的属性。

主要方法 (Main methods) 我们使用字典组织损失组件的存储：此任务分配给私有方法 compute_loss()，该方法返回一个字典，其键值对是上面介绍的组件的名称以及对应的 MSE 和损失值。方法 loss_total 和 metric_total 都依赖于 compute_loss() 的评估，并分别对 keys 和 metrics 属性中编写的所有组件求和。此外，损失只包含对数似然的总和，而指标还存储后验值和每个单独的组件。最后，公共方法 grad_loss() 的任务是计算损失函数相对于网络可训练参数的梯度。操作在注册了 tape 内的可训练参数后，在 tf.GradientTape 内部执行。

5.9.5 预测阶段 - PredNN (Prediction phase - PredNN)

此类继承自 PhysNN，旨在：

- 使用给定参数和输入计算样本；
- 存储 NN 参数的样本；
- 执行并显示误差计算；
- 计算并显示 UQ 的统计信息。

构造函数 (Constructor) 此类的构造函数依赖于父类 PhysNN 的构造函数，采用与 LossNN 描述的相同机制。然后，它初始化一个空的 thetas 列表来存储 NN 参数的样本。

主要方法 (Main methods) 该类具有计算输出样本的私有方法：compute_sample() 是一个私有方法，它在给定一个 NN 参数样本的情况下，通过依赖 PhysNN 的 forward() 执行一次前向传播。然后，它通过执行操作 $\tilde{u} = u_{norm} \cdot \sigma_u + \mu_u$ 返回解和参数场的去归一化输出。predict() 私有方法随后计算给定一组 NN 参数样本的 N_{thetas} 输出样本列表。首先它检查请求的样本数量是否可用，然后它通过迭代调用 compute_sample 辅助方法。它返回两个单独的列表，一个用于解，一个用于参数场，它们的元素是形状为 (n_sample, n_out_sol) 和 (n_sample, n_out_par) 的 tf.Tensor。此方法随后在公共方法 draw_samples() 中使用，该方法通过调用 predict 并将其转换为 numpy 类型，从而绘制给定输入的解和参数场的样本。输出是一个字典，其键是 solution 和 parametric field。predict 方法也被 mean_and_std() 调用，这是一个公共方法，它使用辅助方法 statistics 计算输出样本的均值和标准差。表示每个坐标处的均值/标准差的长度为 n_samples 的度量数组随后存储在字典 function_confidence 中。用于误差和 UQ

计算的私有方法是 compute_errors 和 compute_UQ(); 对于误差，计算解和参数场，我们比较验证点处的函数值与 NN 预测的分布均值。误差以离散 L^2 范数 (MSE) 计算，然后归一化以查看其百分比。在 UQ 的情况下，我们返回一个包含重建函数不确定性的标量量化字典。存储的指标是解和参数场上的均值和最大标准差。上述私有方法随后由 test_errors() 调用，这是一个致力于创建存储误差和 UQ 字典的公共方法。至于误差与用户的通信，私有辅助方法 disp_UQ() 和 disp_err() 分别在终端上显示 UQ 和误差。然后，公共方法 show_errors() 在终端上打印解和参数场的误差和 UQ。

5.9.6 最终封装 - BayesNN (Final wrapper - BayesNN)

此类继承自 LossNN 和 PredNN。它是我们在主脚本中实例化对象并旨在实现以下目的的类:

- 在两个字典中包含所有训练历史;
- 将新的损失值附加到历史中;
- 作为 Algorithm 的子类传递，如 5.7 小节所述进行训练。

构造函数 (Constructor) 此类的构造函数需要两个参数: par，即 5.2.3 小节中描述的 Param 类的实例，以及 equation，即 5.5 小节中描述的对应对象。此类具有两个父类 (LossNN 和 PredNN)，因此属于多重继承框架，如图 7 所示。使用 super() 方法，我们访问父类的构造函数，并向其传递参数 par 和 equation。这些参数随后将传递给层次结构中更高级别的类 (PhysNN，然后是 CoreNN)；这些类随后仅在它们自己的构造函数中利用已描述的选定信息，并根据封装原则构建它们自己的属性。构造函数参数随后存储在属性 constructors 中，以确保可以轻松生成 BayesNN 派生对象的副本（如 5.7.6 小节中 SVGD 算法的粒子）。

主要方法 (Main methods) 该类的主要目的是提供其他结构的封装，以便在可执行脚本中实例化一个综合类的对象。实际上，它作为参数传递给算法类，我们将在其中存储一个由算法提供的数据训练的 BayesNN 实例。尽管如此，此类还包含两个方法: initialize_losses()，一个初始化 MSE 和对数似然的空字典的私有方法，以及 loss_step()，一个将新损失附加到损失历史的公共方法。