PINNs 教程

Physics-Informed Neural Networks 理论与实现

王扬
浙江理工大学

前言

本文是一份关于 Physics-Informed Neural Networks（PINNs）的中文教程，旨在系统介绍 PINNs 的基本理论框架及其在偏微分方程（PDE）正问题与反问题中的应用方法。

本文面向具有偏微分方程基础及一定深度学习背景的研究生和相关研究人员，力求在数学严谨性与可读性之间取得平衡，强调物理建模思想、数学表达与数值实现之间的对应关系。

全文结构如下：首先介绍 PINNs 的基础理论框架，其后将结合具体算例与代码实现，展示 PINNs 在实际问题中的应用方式。

目录

第 1 章　Physics-Informed Neural Networks（PINNs）基础理论

本章系统介绍 Physics-Informed Neural Networks（PINNs）的理论基础，重点阐述其在偏微分方程（Partial Differential Equations, PDEs）正问题与反问题中的统一建模框架。内容严格基于一般形式的偏微分方程，并详细说明神经网络近似、物理约束、损失函数构造以及参数反演的数学含义，为后续数值实现提供完整、严谨的理论支撑。

1.1　一般形式的偏微分方程描述

考虑定义在区域 $\Omega \subset \mathbb{R}^d$ 上的一般偏微分方程

$$\mathcal{N}[u(\boldsymbol{x});\, \boldsymbol{\lambda}] = 0, \quad \boldsymbol{x} \in \Omega, \tag{1}$$

并满足边界条件或初始条件

$$\mathcal{B}[u(\boldsymbol{x})] = 0, \quad \boldsymbol{x} \in \partial\Omega. \tag{2}$$

其中：

- $u(\boldsymbol{x})$ 表示待求解的未知物理场（如温度、位移、波函数等）；
- $\boldsymbol{x} \in \mathbb{R}^d$ 表示自变量，可能包含空间坐标与时间变量；
- $\mathcal{N}(\cdot)$ 为给定的微分算子，通常包含一阶或高阶偏导数；
- $\mathcal{B}(\cdot)$ 为边界条件或初始条件算子；
- $\boldsymbol{\lambda}$ 表示 PDE 中的物理参数，可以是常数、向量，甚至是空间相关的未知函数。

对于时空问题，时间变量 t 被视为输入向量 $\boldsymbol{x}$ 的一个分量，此时边界条件算子 $\mathcal{B}$ 同时包含：

- 初始条件（$t = 0$）；
- 空间边界条件（$\boldsymbol{x} \in \partial\Omega$）。

1.2　PINNs 的核心思想

PINNs 的基本思想可以概括为：

使用神经网络作为未知场函数的连续近似，并通过损失函数将偏微分方程、边界条件和观测数据作为软约束直接嵌入训练过程。

与传统数值方法不同，PINNs 不需要对微分算子进行显式离散，而是借助自动微分（Automatic Differentiation, AD）精确计算神经网络关于输入变量的导数，从而在连续意义下构造 PDE 残差。

1.3 PINNs 求解 PDE 正问题

在正问题中，物理参数 $\boldsymbol{\lambda}$ 是已知的，目标是求解满足方程 (1) 和条件 (2) 的未知函数 $u(\boldsymbol{x})$。

1.3.1 神经网络近似

引入一个带参数的神经网络

$$u_{\boldsymbol{\theta}}(\boldsymbol{x}) \approx u(\boldsymbol{x}),$$

其中 $\boldsymbol{\theta}$ 为网络的可训练参数。该网络在整个定义域 Ω 上提供一个连续、可微的函数表示。

1.3.2 配置点与损失函数构造

在 PINNs 中，通常在定义域内部选取一组配置点

$$\mathcal{T}_f = \{\boldsymbol{x}_f^{(i)}\}_{i=1}^{N_f} \subset \Omega,$$

在边界上选取一组边界点

$$\mathcal{T}_b = \{\boldsymbol{x}_b^{(i)}\}_{i=1}^{N_b} \subset \partial\Omega.$$

据此构造损失函数

$$\mathcal{L}(\boldsymbol{\theta}) = w_f \mathcal{L}_f(\boldsymbol{\theta}) + w_b \mathcal{L}_b(\boldsymbol{\theta}), \tag{3}$$

其中

$$\mathcal{L}_f(\boldsymbol{\theta}) = \frac{1}{|\mathcal{T}_f|} \sum_{\boldsymbol{x}_f \in \mathcal{T}_f} \left\| \mathcal{N}[u_{\boldsymbol{\theta}}(\boldsymbol{x}_f); \boldsymbol{\lambda}] \right\|^2, \tag{4}$$

$$\mathcal{L}_b(\boldsymbol{\theta}) = \frac{1}{|\mathcal{T}_b|} \sum_{\boldsymbol{x}_b \in \mathcal{T}_b} \left\| \mathcal{B}[u_{\boldsymbol{\theta}}(\boldsymbol{x}_b)] \right\|^2. \tag{5}$$

其中：

- $\mathcal{L}_f$ 为 PDE 残差损失；
- $\mathcal{L}_b$ 为边界条件损失；
- w_f, w_b 为权重系数，用于平衡不同约束的重要性。

1.3.3 参数优化

通过最小化损失函数

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}),$$

即可得到满足 PDE 与边界条件的近似解 $u_{\boldsymbol{\theta}^*}(\boldsymbol{x})$。损失函数中涉及的导数通过自动微分计算，参数更新通常采用 Adam 等一阶优化算法。

1.4 PINNs 求解 PDE 反问题

在反问题中，物理参数 $\boldsymbol{\lambda}$ 是未知的，需要借助有限的观测数据进行反演。

1.4.1 观测数据约束

设在测量点集合

$$\mathcal{T}_m = \{\boldsymbol{x}_m^{(i)}\}_{i=1}^{N_m} \subset \Omega$$

上获得观测数据，其约束形式可写为

$$\mathcal{M}[u_{\boldsymbol{\theta}}, \boldsymbol{x}_m] = 0. \tag{6}$$

1.4.2 联合损失函数

将 PDE、边界条件与观测数据统一纳入损失函数：

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}) = w_f \mathcal{L}_f + w_b \mathcal{L}_b + w_m \mathcal{L}_m, \tag{7}$$

其中

$$\mathcal{L}_m = \frac{1}{|\mathcal{T}_m|} \sum_{\boldsymbol{x}_m \in \mathcal{T}_m} \left\| \mathcal{M}[u_{\boldsymbol{\theta}}, \boldsymbol{x}_m] \right\|^2. \tag{8}$$

通过联合优化

$$(\boldsymbol{\theta}^*, \boldsymbol{\lambda}^*) = \arg\min_{\boldsymbol{\theta}, \boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}),$$

即可同时得到未知解与未知物理参数的反演结果。

第 2 章　基于 PyTorch 的 PINNs 正问题与反问题数值实现

本章通过一个简单的一维 Poisson 方程示例，系统介绍如何使用 PyTorch 从零实现 Physics-Informed Neural Networks（PINNs），并分别求解偏微分方程的正问题与反问题。所有实现均不依赖 DeepXDE 等高层框架，力求逻辑清晰、步骤可复现，适合作为 PINNs 的入门示例。

2.1　模型问题说明

考虑定义在一维空间区间

$$\Omega = [-1, 1]$$

上的 Poisson 方程

$$\frac{\mathrm{d}^2 u(x)}{\mathrm{d}x^2} = q(x), \quad x \in [-1, 1], \tag{9}$$

并满足齐次 Dirichlet 边界条件

$$u(-1) = 0, \quad u(1) = 0. \tag{10}$$

其中：

- x 表示一维空间自变量；
- $u(x)$ 为未知解函数；
- $\dfrac{\mathrm{d}^2 u}{\mathrm{d}x^2}$ 表示解函数的二阶空间导数；
- $q(x)$ 为源项函数，用于描述系统内部的激励分布。

该方程可作为多种物理模型的简化原型，例如一维稳态热传导或静电势问题。

2.1.1　构造解析解用于验证

为验证 PINNs 的数值求解能力，本文人为构造解析解

$$u_{\text{true}}(x) = \sin(\pi x), \tag{11}$$

其对应的源项函数为

$$q_{\text{true}}(x) = \frac{\mathrm{d}^2 u_{\text{true}}}{\mathrm{d}x^2} = -\pi^2 \sin(\pi x). \tag{12}$$

该构造保证了解函数与边界条件完全一致，可作为数值结果的参考解。

2.2 PINNs 求解正问题

2.2.1 正问题的定义

在正问题中，源项函数 $q(x)$ 被视为已知函数，目标是求解满足 PDE 约束与边界条件的未知解函数 $u(x)$。该过程不依赖任何观测数据，完全由物理方程驱动。

2.2.2 神经网络近似

使用全连接神经网络 $u_{\boldsymbol{\theta}}(x)$ 近似解函数 $u(x)$。由于神经网络在参数空间中是可微的，其关于输入变量的导数可以通过自动微分机制精确计算。

2.2.3 损失函数构造

通过自动微分计算二阶导数，定义 PDE 残差

$$r(x) := \frac{\mathrm{d}^2 u_{\boldsymbol{\theta}}}{\mathrm{d}x^2}(x) - q(x). \tag{13}$$

正问题的损失函数由 PDE 残差项与边界条件项组成：

$$\mathcal{L} = \mathcal{L}_{\mathrm{pde}} + \mathcal{L}_{\mathrm{bc}}, \tag{14}$$

其中

$$\mathcal{L}_{\mathrm{pde}} = \frac{1}{N_f} \sum_{x_f} |r(x_f)|^2, \tag{15}$$

$$\mathcal{L}_{\mathrm{bc}} = \frac{1}{2}\big(|u_{\boldsymbol{\theta}}(-1)|^2 + |u_{\boldsymbol{\theta}}(1)|^2\big). \tag{16}$$

2.2.4 PyTorch 实现（正问题）

```
# ------------------------------
# 正问题 PINNs 完整示例
# ------------------------------

import torch
import torch.nn as nn
```

```
import numpy as np
import matplotlib.pyplot as plt
import math

# 1. 定义全连接神经网络
class FCN(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.layers = nn.ModuleList()
        for i in range(len(layers)-1):
            self.layers.append(nn.Linear(layers[i], layers[i+1]))
        self.activation = torch.tanh

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = self.activation(layer(x))
        return self.layers[-1](x)

# 2. 实例化网络
net_u = FCN([1, 20, 20, 20, 1])  # 输入1维，输出1维，隐藏层20个神经元

# 3. 定义 PDE 残差函数
def pde_residual(x):
    x.requires_grad_(True)
    u = net_u(x)

    # 计算一阶导数
    u_x = torch.autograd.grad(
        u, x, grad_outputs=torch.ones_like(u),
        create_graph=True
    )[0]

    # 计算二阶导数
    u_xx = torch.autograd.grad(
        u_x, x, grad_outputs=torch.ones_like(u_x),
        create_graph=True
    )[0]
```

```
    # 已知源项函数 q(x) = -pi^2 sin(pi x)
    q = -math.pi**2 * torch.sin(math.pi * x)
    return u_xx - q

# 4. 训练数据: PDE 内部点 + 边界点
N_f = 100
x_f = torch.linspace(-1, 1, N_f).view(-1, 1)
x_bc = torch.tensor([[-1.0], [1.0]])

# 5. 定义优化器
optimizer = torch.optim.Adam(net_u.parameters(), lr=1e-3)

# 6. 训练循环
for epoch in range(5000):
    optimizer.zero_grad()

    # PDE 残差损失
    loss_pde = torch.mean(pde_residual(x_f)**2)
    # 边界条件损失
    loss_bc = torch.mean(net_u(x_bc)**2)

    # 总损失
    loss = loss_pde + loss_bc
    loss.backward()
    optimizer.step()

    if epoch % 500 == 0:
        print(f"Epoch {epoch}, Loss {loss.item():.3e}")

# 7. 结果可视化
x_test = torch.linspace(-1, 1, 200).view(-1, 1)
u_pred = net_u(x_test).detach().numpy()
u_true = np.sin(np.pi * x_test.numpy())

plt.plot(x_test, u_true, label="Exact")
plt.plot(x_test, u_pred, "--", label="PINN")
```

```
plt.xlabel("x")
plt.ylabel("u(x)")
plt.legend()
plt.title("PINNs Solution of the Forward Problem")
plt.show()
```

说明

- 网络 net_u 用于近似解函数 $u(x)$；
- PDE 残差通过自动微分计算二阶导得到；
- 损失函数由 PDE 残差和边界条件组成；
- 训练过程中不需要任何观测数据，仅依赖物理方程。

2.3 PINNs 求解反问题

2.3.1 反问题的定义

在反问题中，源项函数 $q(x)$ 为未知函数，需要与解函数 $u(x)$ 一并从有限观测数据中反演。假设在测量点 $\{x_i\}_{i=1}^{N_m}$ 上可获得

$$u(x_i) = u^{\mathrm{obs}}(x_i). \tag{17}$$

2.3.2 双网络建模

分别使用两个神经网络（也可以使用具有多个输出的单个神经网络）：

- $u_{\boldsymbol{\theta}}(x)$ 近似解函数；
- $q_{\boldsymbol{\phi}}(x)$ 近似未知源项。

PDE 残差定义为

$$r(x) := \frac{\mathrm{d}^2 u_{\boldsymbol{\theta}}}{\mathrm{d}x^2}(x) - q_{\boldsymbol{\phi}}(x). \tag{18}$$

2.3.3 反问题损失函数

$$\mathcal{L} = \mathcal{L}_{\text{pde}} + \mathcal{L}_{\text{bc}} + \mathcal{L}_{\text{data}}, \tag{19}$$

其中

$$\mathcal{L}_{\text{data}} = \frac{1}{N_m} \sum_{i=1}^{N_m} |u_{\boldsymbol{\theta}}(x_i) - u^{\text{obs}}(x_i)|^2. \tag{20}$$

2.3.4 PyTorch 实现（反问题）

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import math


# ------------------------------
# 定义全连接网络 FCN
# 输入 1 维 x, 输出 2 维 [u,q]
# ------------------------------
class FCN(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.layers = nn.ModuleList()
        for i in range(len(layers) - 1):
            self.layers.append(nn.Linear(layers[i], layers[i + 1]))
        self.activation = torch.tanh

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = self.activation(layer(x))
        return self.layers[-1](x)  # 输出 2 维: u,q


# ------------------------------
# 实例化网络
```

```
# ------------------------------
net = FCN([1, 50, 50, 50, 2])


# ------------------------------
# 训练数据生成
# ------------------------------
def gen_traindata(num):
    xvals = np.linspace(-1, 1, num).reshape(num, 1)
    uvals = np.sin(np.pi * xvals)
    return xvals, uvals


# PDE 内部点
N_f = 200
x_f = torch.linspace(-1, 1, N_f).view(-1, 1)
# 观测数据点
N_m = 20
x_m, u_m = gen_traindata(N_m)
x_m = torch.tensor(x_m, dtype=torch.float32)
u_m = torch.tensor(u_m, dtype=torch.float32)

# PDE 点与观测点合并
x_pde = torch.cat([x_f, x_m], dim=0)

# 边界点
x_bc = torch.tensor([[-1.0], [1.0]], dtype=torch.float32)


# ------------------------------
# PDE 残差函数
# ------------------------------
def pde_residual(x):
    x.requires_grad_(True)
    y = net(x)
    u = y[:, 0:1]
    q = y[:, 1:2]
```

```
    u_x = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u),
                              create_graph=True)[0]
    u_xx = torch.autograd.grad(u_x, x, grad_outputs=torch.ones_like(u_x),
                               create_graph=True)[0]

    # 使用 -u_xx + q，与解析解符号一致
    return -u_xx + q


# ------------------------------
# 优化器和训练循环
# ------------------------------
optimizer = torch.optim.Adam(net.parameters(), lr=1e-4)
num_epochs = 20000

for epoch in range(num_epochs):
    optimizer.zero_grad()

    # PDE 残差损失
    res = pde_residual(x_pde)
    loss_pde = torch.mean(res ** 2)

    # 边界条件损失
    u_bc = net(x_bc)[:, 0:1]
    loss_bc = torch.mean(u_bc ** 2)

    # 数据损失
    u_pred_m = net(x_m)[:, 0:1]
    loss_data = torch.mean((u_pred_m - u_m) ** 2)

    # 总损失，使用自定义权重系数
    loss = 1.0 * loss_pde + 100.0 * loss_bc + 1000.0 * loss_data
    loss.backward()
    optimizer.step()

    if epoch % 2000 == 0:
```

```
        print(
            f"Epoch {epoch}, total loss {loss.item():.3e}, "
            f"PDE {loss_pde.item():.3e},"
            f" BC {loss_bc.item():.3e}, "
            f"observation data {loss_data.item():.3e}")

# ------------------------------
# 预测和可视化
# ------------------------------
x_test = torch.linspace(-1, 1, 500).view(-1, 1)
y_pred = net(x_test).detach().numpy()
u_pred = y_pred[:, 0:1]
q_pred = y_pred[:, 1:2]

x_test_np = x_test.numpy()
u_true = np.sin(np.pi * x_test_np)
q_true = -np.pi ** 2 * np.sin(np.pi * x_test_np)

# L2 相对误差
l2_u = np.linalg.norm(u_true - u_pred) / np.linalg.norm(u_true)
l2_q = np.linalg.norm(q_true - q_pred) / np.linalg.norm(q_true)
print(f"L2 relative error u: {l2_u:.3e}, q: {l2_q:.3e}")

# 绘图
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(x_test_np, u_true, "-", label="u_true")
plt.plot(x_test_np, u_pred, "--", label="u_NN")
plt.xlabel("x")
plt.ylabel("u(x)")
plt.legend()
plt.title("u(x)")

plt.subplot(1, 2, 2)
plt.plot(x_test_np, q_true, "-", label="q_true")
plt.plot(x_test_np, q_pred, "--", label="q_NN")
plt.xlabel("x")
```

```
plt.ylabel("q(x)")
plt.legend()
plt.title("q(x)")
plt.tight_layout()
plt.show()
```

说明

- 正问题中 $q(x)$ 已知；反问题中 $q(x)$ 是未知函数，需要用神经网络拟合；
- 反问题损失函数增加了数据拟合项 $\mathcal{L}_{data}$，以确保网络能匹配观测值；
- 使用单个或者两个网络拟合 $u(x)$ 和 $q(x)$，网络参数通过联合优化求解反问题，是典型的受物理约束的非线性反演问题。

2.4 正问题与反问题的对比

- 正问题中仅优化解函数网络；
- 反问题中需同时优化解函数与未知参数函数；
- 反问题本质上是受物理约束的非线性反演问题。

2.5 本章小结

本章以一维 Poisson 方程为例，系统介绍了 PINNs 在正问题与反问题中的实现流程。通过 PyTorch 的自动微分机制，可以自然地将 PDE 约束嵌入神经网络训练过程，为后续复杂物理模型与高维问题的研究奠定基础。

推荐阅读文献

1. Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2021). **DeepXDE: A Deep Learning Library for Solving Differential Equations**. Journal of Computational Physics, 429, 109926.
介绍 *DeepXDE* 框架及其在 *PDE* 正/反问题中的应用。

2. Jagtap, A. D., Kawaguchi, K., & Karniadakis, G. E. (2020). **Adaptive activation functions accelerate convergence in deep and physics-informed neural networks**. Journal of Computational Physics, 404, 109136.
提出自适应激活函数以加快深度神经网络和 *PINNs* 的训练收敛速度。

3. Yu, L., Lin, Z., Meng, X., & Karniadakis, G. E. (2021). **Self-adaptive physics-informed neural networks**. Journal of Computational Physics, 426, 109951.
介绍自适应 *PINNs* 框架，通过动态调整损失权重提高训练稳定性。

4. Wang, S., Teng, Y., & Perdikaris, P. (2021). **When and why PINNs fail to train: A neural tangent kernel perspective**. Journal of Computational Physics, 449, 110768.
从神经切线核视角分析 *PINNs* 训练失败的原因。

5. Zhuang, X., Chen, C., & Karniadakis, G. E. (2022). **A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks**. Computer Methods in Applied Mechanics and Engineering, 394, 114901.
系统研究 *PINNs* 中自适应采样策略的效果。

6. Gao, Y., Sun, W., Zhang, J., & Karniadakis, G. E. (2021). **Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems**. Journal of Computational Physics, 442, 110440.
提出梯度增强 *PINNs* 方法，提高正问题和反问题求解精度。

7. Wang, S., Teng, Y., & Perdikaris, P. (2022). **On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks**. Journal of Computational Physics, 469, 111543.
分析傅里叶特征网络的特征偏差及其在多尺度 *PDE* 问题中的应用。