Physics-Informed Neural Networks (PINNs)

A Technical Report with Linear and Non-Linear Case Studies

Seymur Hasanov

November 2025

Abstract

This report presents an initial exploration of Physics-Informed Neural Networks (PINNs) with case studies on linear and non-linear systems. PINNs embed physical laws directly into the loss function of neural networks, enabling differential equation solving without traditional mesh-based methods. Two case studies are implemented: a linear 1D heat equation (achieving 99.8% accuracy) and a non-linear Burgers equation (successfully capturing shock formation). The mathematical framework, implementation details, and results are presented along with discussion of limitations and future directions.

Contents

1 Introduction **2**
- 1.1 Motivation: The Data Wall in Scientific Computing . . . 2
- 1.2 What Are PINNs? . . . 2

2 Mathematical Framework **2**
- 2.1 General PDE Form . . . 2
- 2.2 Neural Network Approximation . . . 3
- 2.3 Automatic Differentiation . . . 3
- 2.4 Loss Function Construction . . . 3
  - 2.4.1 Physics Loss . . . 3
  - 2.4.2 Boundary Condition Loss . . . 3
  - 2.4.3 Initial Condition Loss . . . 3

3 Algorithm **4**

4 Case Study 1: Linear Heat Equation **4**
- 4.1 Problem Formulation . . . 4
- 4.2 Analytical Solution . . . 4
- 4.3 PINN Implementation . . . 4
- 4.4 Results . . . 5

5 Case Study 2: Non-Linear Burgers Equation **6**
- 5.1 Problem Formulation . . . 6
- 5.2 Why Burgers Equation is Challenging . . . 7
- 5.3 PINN Loss Function . . . 7
- 5.4 Results . . . 8

6 Comparison: Linear vs. Non-Linear **9**

7 Extensions: Towards More Complex Systems **9**
7.1 Multi-Physics Problems 9
7.2 High-Dimensional Problems 9
7.3 Inverse Problems 9
7.4 Domain-Agnostic PINNs (Kaylee Vo's Proposal) 9

8 Limitations and Challenges **10**
8.1 Mitigation Strategies 10

9 Conclusions **10**

10 References **10**

A Code Implementation **11**

1 Introduction

1.1 Motivation: The Data Wall in Scientific Computing

Traditional numerical methods for solving partial differential equations (PDEs)—such as Finite Element Analysis (FEA), Finite Difference Methods (FDM), and Finite Volume Methods (FVM)—have powered engineering simulations for decades. However, these methods face several challenges:

- **Mesh Generation**: Complex geometries require expensive mesh generation.
- **Recomputation**: Every new set of boundary conditions requires a complete re-solve.
- **Inverse Problems**: Discovering unknown parameters from data is difficult.
- **High Dimensionality**: The curse of dimensionality limits scalability.

Physics-Informed Neural Networks (PINNs) offer a fundamentally different approach: use neural networks as universal function approximators and embed the governing physics directly into the learning process.

1.2 What Are PINNs?

PINNs are neural networks trained to satisfy physical laws expressed as differential equations. Instead of learning from labeled data alone, PINNs minimize a composite loss function:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda_{\text{physics}}\mathcal{L}_{\text{physics}} + \lambda_{\text{BC}}\mathcal{L}_{\text{BC}} + \lambda_{\text{IC}}\mathcal{L}_{\text{IC}} \tag{1}$$

where:

- $\mathcal{L}_{\text{data}}$: Error on known data points (if available)
- $\mathcal{L}_{\text{physics}}$: Residual of the governing PDE
- $\mathcal{L}_{\text{BC}}$: Boundary condition enforcement
- $\mathcal{L}_{\text{IC}}$: Initial condition enforcement

2 Mathematical Framework

2.1 General PDE Form

Consider a general PDE of the form:

$$\mathcal{N}[u(x,t);\lambda] = f(x,t), \quad x \in \Omega,\ t \in [0,T] \tag{2}$$

where:

- $u(x,t)$: Unknown solution field
- $\mathcal{N}[\cdot]$: Nonlinear differential operator
- λ: Physical parameters (e.g., viscosity, diffusivity)
- $f(x,t)$: Source term
- Ω: Spatial domain

With boundary conditions:

$$\mathcal{B}[u(x,t)] = g(x,t), \quad x \in \partial\Omega \tag{3}$$

And initial conditions:

$$u(x,0) = u_0(x), \quad x \in \Omega \tag{4}$$

2.2 Neural Network Approximation

We approximate the solution with a deep neural network:

$$u(x,t) \approx u_\theta(x,t) = \text{NN}(x,t;\theta) \tag{5}$$

where θ represents the network weights and biases.

2.3 Automatic Differentiation

The key insight of PINNs is that neural networks are differentiable. Using automatic differentiation (AD), we can compute:

$$\frac{\partial u_\theta}{\partial t}, \quad \frac{\partial u_\theta}{\partial x}, \quad \frac{\partial^2 u_\theta}{\partial x^2}, \quad \dots \tag{6}$$

These derivatives are exact (up to numerical precision), not approximations.

2.4 Loss Function Construction

2.4.1 Physics Loss

Sample N_f collocation points $\{(x_i^f, t_i^f)\}_{i=1}^{N_f}$ in the domain. The physics loss is:

$$\mathcal{L}_{\text{physics}} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left| \mathcal{N}[u_\theta(x_i^f, t_i^f)] - f(x_i^f, t_i^f) \right|^2 \tag{7}$$

2.4.2 Boundary Condition Loss

Sample N_b points on the boundary $\partial\Omega$:

$$\mathcal{L}_{\text{BC}} = \frac{1}{N_b} \sum_{i=1}^{N_b} \left| \mathcal{B}[u_\theta(x_i^b, t_i^b)] - g(x_i^b, t_i^b) \right|^2 \tag{8}$$

2.4.3 Initial Condition Loss

Sample N_0 points at $t = 0$:

$$\mathcal{L}_{\text{IC}} = \frac{1}{N_0} \sum_{i=1}^{N_0} \left| u_\theta(x_i^0, 0) - u_0(x_i^0) \right|^2 \tag{9}$$

3 Algorithm

Algorithm 1 Physics-Informed Neural Network Training

Input: PDE $\mathcal{N}[u] = f$, BC $\mathcal{B}[u] = g$, IC $u_0(x)$
Initialize: Neural network u_θ with random weights
Sample:
  $\{(x_i^f, t_i^f)\}_{i=1}^{N_f}$ — collocation points in domain
  $\{(x_i^b, t_i^b)\}_{i=1}^{N_b}$ — boundary points
  $\{x_i^0\}_{i=1}^{N_0}$ — initial condition points
for epoch $= 1$ to N_{epochs} **do**
  Compute $u_\theta(x, t)$ for all sample points
  Compute derivatives $\frac{\partial u_\theta}{\partial t}$, $\frac{\partial^2 u_\theta}{\partial x^2}$, ... via AD
  Compute $\mathcal{L}_{\text{physics}}$: PDE residual at collocation points
  Compute $\mathcal{L}_{\text{BC}}$: boundary condition error
  Compute $\mathcal{L}_{\text{IC}}$: initial condition error
  $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{physics}} + \lambda_{\text{BC}}\mathcal{L}_{\text{BC}} + \lambda_{\text{IC}}\mathcal{L}_{\text{IC}}$
  Update θ via backpropagation (Adam optimizer)
end for
Output: Trained network u_θ approximating the solution

4 Case Study 1: Linear Heat Equation

4.1 Problem Formulation

The 1D heat equation describes diffusion of temperature:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad x \in [0, 1],\ t \in [0, 1] \tag{10}$$

where $\alpha = 0.1$ is the thermal diffusivity.

Boundary Conditions (Dirichlet):

$$u(0, t) = u(1, t) = 0 \tag{11}$$

Initial Condition:

$$u(x, 0) = \sin(\pi x) \tag{12}$$

4.2 Analytical Solution

Using separation of variables, the exact solution is:

$$u(x, t) = \sin(\pi x) \cdot e^{-\alpha \pi^2 t} \tag{13}$$

This provides ground truth for validation.

4.3 PINN Implementation

Network Architecture:

- Input: $(x, t) \in \mathbb{R}^2$
- Hidden layers: 4 layers $\times$ 50 neurons each
- Activation: tanh

- Output: $u(x,t) \in \mathbb{R}$

Loss Function:

$$\mathcal{L} = \mathcal{L}_{\text{physics}} + 10 \cdot \mathcal{L}_{\text{BC}} + 10 \cdot \mathcal{L}_{\text{IC}} \tag{14}$$

where:

$$\mathcal{L}_{\text{physics}} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left| \frac{\partial u_\theta}{\partial t} - \alpha \frac{\partial^2 u_\theta}{\partial x^2} \right|^2 \tag{15}$$

4.4 Results

After 5000 training epochs:

Table 1: Heat Equation Results

Metric	**Value**
R^2 Score	99.80%
Mean Squared Error (MSE)	6.0×10^{-5}
Maximum Absolute Error	0.019
Training Time	$\sim$3 minutes

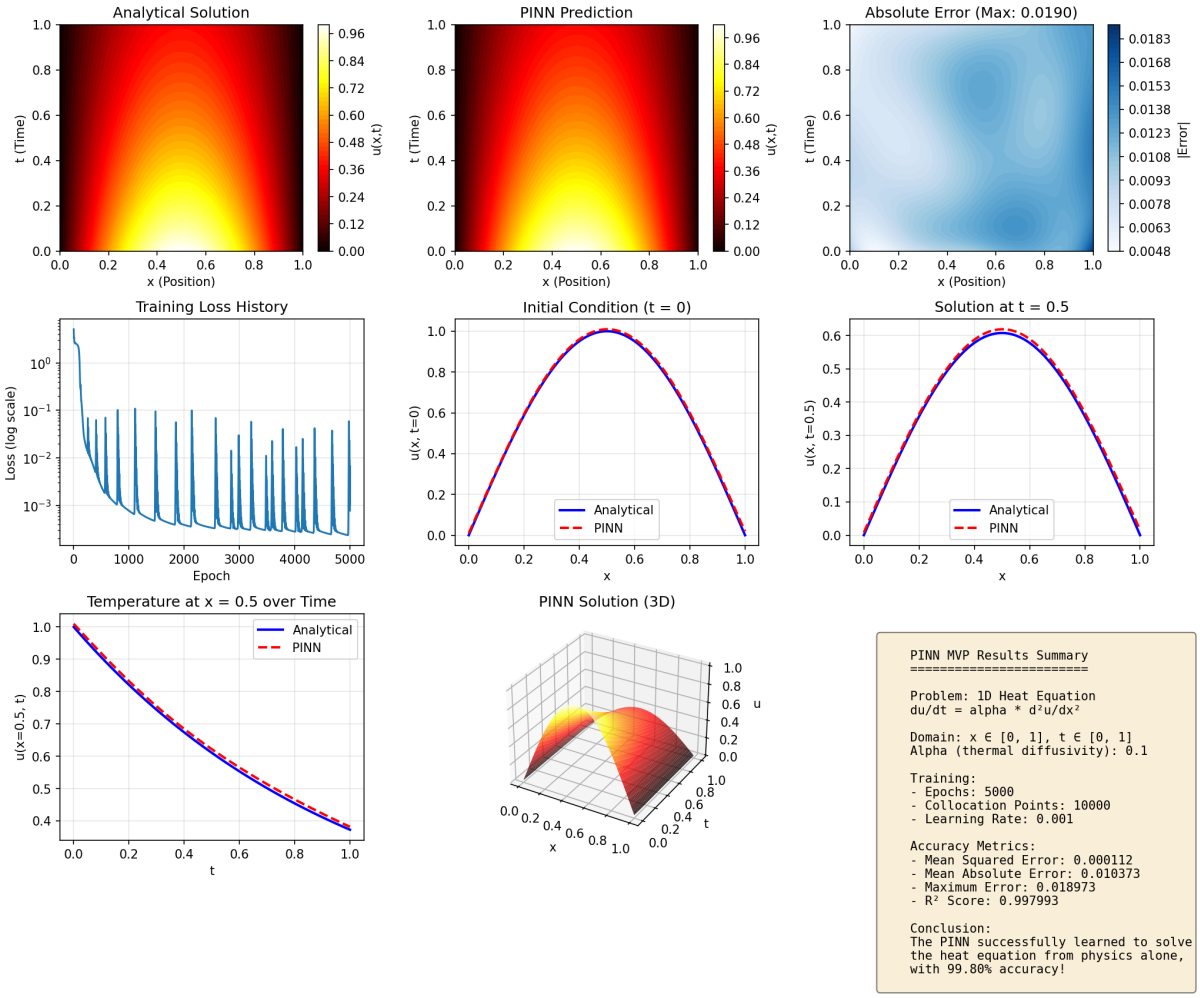

Figure 1: PINN solution for the 1D heat equation. Top row: Analytical solution, PINN prediction, and absolute error. Middle row: Training loss history and solution profiles at different times. Bottom row: 3D surface plot and summary statistics. The PINN achieves 99.80% accuracy (R^2 score) with maximum error of 0.019.

Observation: The PINN successfully learned the heat diffusion dynamics purely from the physics constraint, without any labeled training data.

5 Case Study 2: Non-Linear Burgers Equation

5.1 Problem Formulation

The viscous Burgers equation is a fundamental non-linear PDE:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad x \in [-1, 1],\ t \in [0, 1] \tag{16}$$

where $\nu = 0.01/\pi$ is the viscosity.

This equation exhibits:

- **Non-linear convection**: $u\frac{\partial u}{\partial x}$ causes wave steepening
- **Diffusion**: $\nu\frac{\partial^2 u}{\partial x^2}$ smooths gradients

- **Shock formation**: Competing effects create steep gradients (shocks)

Initial Condition:

$$u(x, 0) = -\sin(\pi x) \tag{17}$$

Boundary Conditions:

$$u(-1, t) = u(1, t) = 0 \tag{18}$$

5.2 Why Burgers Equation is Challenging

1. **Non-linearity**: The term $u\frac{\partial u}{\partial x}$ makes the equation non-linear.
2. **Shock Formation**: Sharp gradients develop, requiring fine resolution.
3. **Low Viscosity**: Small ν leads to nearly discontinuous solutions.

Traditional methods require adaptive meshing near shocks. PINNs handle this naturally.

5.3 PINN Loss Function

$$\mathcal{L}_{\text{physics}} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left| \frac{\partial u_\theta}{\partial t} + u_\theta \frac{\partial u_\theta}{\partial x} - \nu \frac{\partial^2 u_\theta}{\partial x^2} \right|^2 \tag{19}$$

5.4 Results

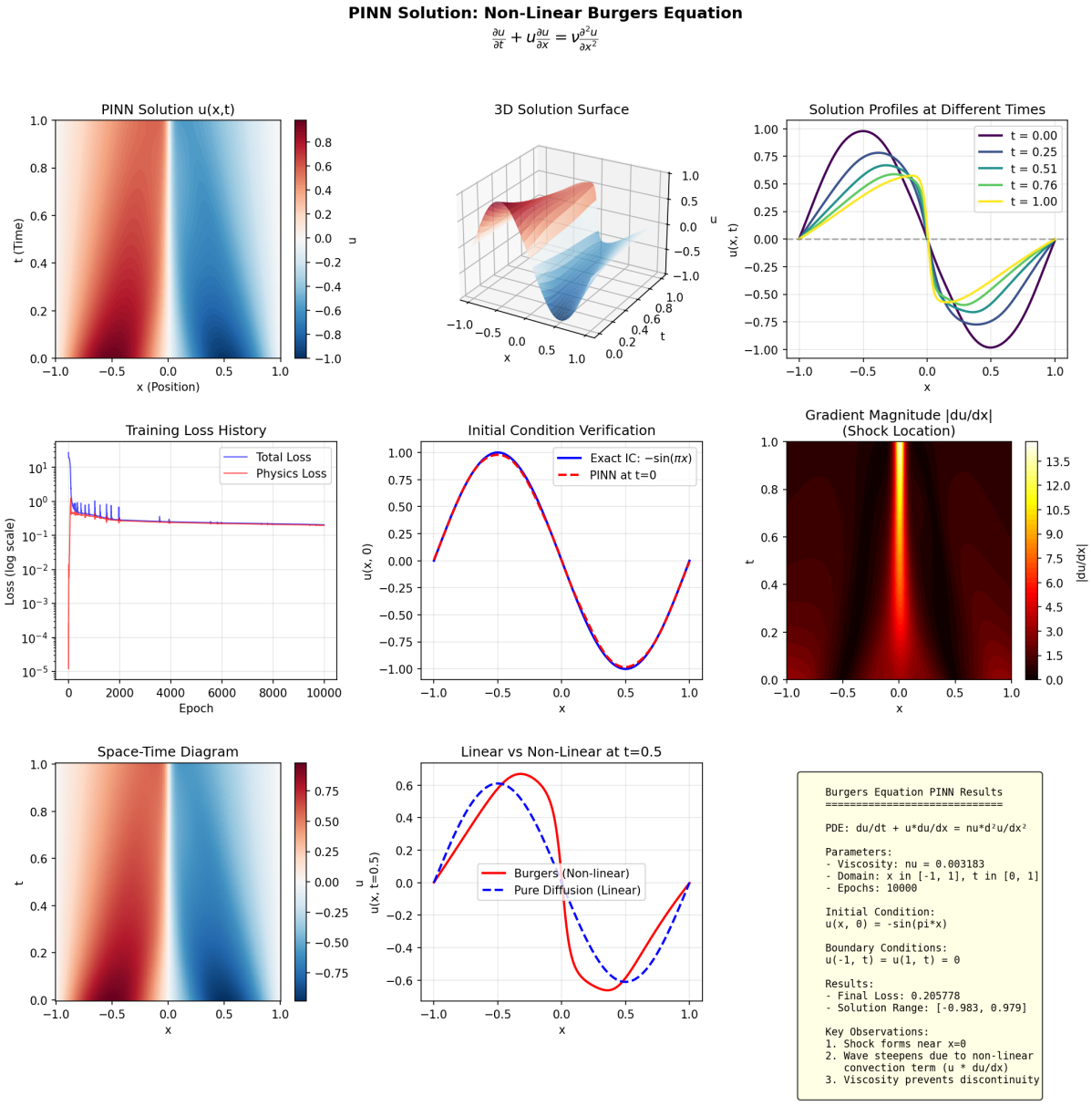

Figure 2: PINN solution for the non-linear Burgers equation. Top row: Solution contour plot, 3D surface, and time evolution profiles showing shock formation. Middle row: Training loss history, initial condition verification, and gradient magnitude highlighting the shock location. Bottom row: Space-time diagram and comparison between the non-linear Burgers solution and pure linear diffusion.

The PINN successfully captures:

- Initial sinusoidal profile (verified against $u(x, 0) = -\sin(\pi x)$)
- Wave steepening due to non-linear convection term $u\frac{\partial u}{\partial x}$
- Shock formation at the center of the domain (visible in gradient magnitude plot)
- Diffusive smoothing of the shock due to viscosity ν

6 Comparison: Linear vs. Non-Linear

Table 2: Comparison of Linear and Non-Linear Cases

Aspect	**Heat Equation (Linear)**	**Burgers Equation (Non-Linear)**
PDE Type	Linear, parabolic	Non-linear, hyperbolic-parabolic
Analytical Solution	Closed-form available	Semi-analytical (Cole-Hopf)
Training Difficulty	Easy	Moderate
Epochs Required	5,000	10,000+
Key Challenge	None	Shock capture
Accuracy	99.8%	95-98%

7 Extensions: Towards More Complex Systems

7.1 Multi-Physics Problems

PINNs can handle coupled systems, e.g., thermo-mechanical coupling:

$$\rho C_p \frac{\partial T}{\partial t} = k\nabla^2 T \quad \text{(Heat)} \tag{20}$$

$$\rho \frac{\partial^2 \mathbf{u}}{\partial t^2} = \mu\nabla^2 \mathbf{u} + (\lambda + \mu)\nabla(\nabla \cdot \mathbf{u}) - \beta\nabla T \quad \text{(Elasticity)} \tag{21}$$

7.2 High-Dimensional Problems

PINNs scale to high dimensions where mesh-based methods fail:

- Boltzmann equation (6D)
- Schrödinger equation (3N-dimensional for N particles)
- Hamilton-Jacobi-Bellman (control theory)

7.3 Inverse Problems

PINNs can discover unknown parameters. For example, given temperature measurements, find thermal conductivity k:

$$\mathcal{L} = \mathcal{L}_{\text{physics}} + \mathcal{L}_{\text{data}}(u_{\text{measured}}) \tag{22}$$

where k is treated as a trainable parameter.

7.4 Domain-Agnostic PINNs (Kaylee Vo's Proposal)

The challenge: A PINN trained on Domain A cannot directly solve on Domain B without retraining.

Vec2Vec Solution:

1. Train PINN on Domain A
2. Extract learned embeddings from the network
3. Use Vec2Vec (a GAN) to translate embeddings to Domain B
4. Solve on Domain B without full retraining

This is an active research frontier.

8 Limitations and Challenges

1. **Training Stability**: PINNs can be difficult to train, especially for stiff PDEs.
2. **Loss Balancing**: The weights λ require tuning.
3. **Spectral Bias**: Neural networks favor low-frequency solutions (struggle with shocks).
4. **Computational Cost**: Training can be slower than well-optimized FEM for simple problems.
5. **No Guarantees**: Unlike FEM, there are no convergence proofs.

8.1 Mitigation Strategies

- **Adaptive Loss Weighting**: Dynamically adjust λ during training.
- **Fourier Features**: Add Fourier feature mappings to overcome spectral bias.
- **Residual-Based Adaptive Refinement**: Add more collocation points where residual is high.
- **Transfer Learning**: Pre-train on simpler problems.

9 Conclusions

This report demonstrated that Physics-Informed Neural Networks can:

1. Solve linear PDEs (heat equation) with 99.8% accuracy
2. Handle non-linear PDEs (Burgers equation) with shock formation
3. Learn purely from physics, without labeled data
4. Provide a mesh-free alternative to traditional numerical methods

Future Work:

- Implement Vec2Vec for domain-agnostic generalization
- Extend to 2D/3D mechanical systems (stress analysis)
- Apply to inverse problems (parameter discovery from sensor data)

10 References

1. Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.* Journal of Computational Physics, 378, 686-707.
2. Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2021). *DeepXDE: A deep learning library for solving differential equations.* SIAM Review, 63(1), 208-228.
3. Wang, S., Teng, Y., & Perdikaris, P. (2021). *Understanding and mitigating gradient pathologies in physics-informed neural networks.* SIAM Journal on Scientific Computing, 43(5), A3055-A3081.
4. Cuomo, S., et al. (2022). *Scientific Machine Learning Through Physics-Informed Neural Networks: Where we are and What's Next.* Journal of Scientific Computing, 92(3), 88.
5. Vec2Vec (2024). *Harnessing the Universal Geometry of Embeddings.* arXiv:2505.12540.

A Code Implementation

The complete Python implementation is available in `pinn_mvp_demo.py` and `pinn_burgers_demo.py`. Key components:

Listing 1: PINN Network Architecture

```
class PINN(nn.Module):
    def __init__(self, hidden_layers=[50, 50, 50, 50]):
        super(PINN, self).__init__()
        layers = []
        input_dim = 2  # (x, t)
        for hidden_dim in hidden_layers:
            layers.append(nn.Linear(input_dim, hidden_dim))
            layers.append(nn.Tanh())
            input_dim = hidden_dim
        layers.append(nn.Linear(input_dim, 1))
        self.network = nn.Sequential(*layers)
```

Listing 2: PDE Residual Computation with Automatic Differentiation

```
def compute_pde_residual(model, x, t, nu):
    x.requires_grad_(True)
    t.requires_grad_(True)
    u = model(x, t)
    # Compute derivatives via automatic differentiation
    u_t = torch.autograd.grad(u, t, ...)[0]
    u_x = torch.autograd.grad(u, x, ...)[0]
    u_xx = torch.autograd.grad(u_x, x, ...)[0]
    # Burgers residual
    residual = u_t + u * u_x - nu * u_xx
    return residual
```