

Physics-Informed Neural Networks (PINNs)

Outline

- 1 Introduction
- 2 Deep Neural Networks
- 3 Automatic Differentiation
- 4 Physics-Informed Neural Networks
- 5 Approximation Theory and Error Analysis
- 6 Comparison: PINNs vs FEM
- 7 Advanced Topics
- 8 Summary

Introduction: From Deep Learning to PDEs

- Deep learning has achieved remarkable success in:
 - Computer vision
 - Natural language processing

Introduction: From Deep Learning to PDEs

- Deep learning has achieved remarkable success in:
 - Computer vision
 - Natural language processing
- **Recent emergence:** Using deep learning to solve **Partial Differential Equations** (PDEs)

Introduction: From Deep Learning to PDEs

- Deep learning has achieved remarkable success in:
 - Computer vision
 - Natural language processing
- **Recent emergence:** Using deep learning to solve **Partial Differential Equations** (PDEs)
- **Key advantage:** Replace traditional mesh-based numerical methods with neural networks

Introduction: From Deep Learning to PDEs

- Deep learning has achieved remarkable success in:
 - Computer vision
 - Natural language processing
- **Recent emergence:** Using deep learning to solve **Partial Differential Equations** (PDEs)
- **Key advantage:** Replace traditional mesh-based numerical methods with neural networks
- Neural networks approximate the solution to a PDE

Scientific Machine Learning (SciML)

Traditional Approaches:

- Finite Difference Method (FDM)
- Finite Element Method (FEM)
- Mesh-based methods
- Fixed domain geometry

Deep Learning Approach:

- Mesh-free approach
- Automatic differentiation
- Potentially breaks curse of dimensionality
- Flexible domain handling

Scientific Machine Learning (SciML)

Traditional Approaches:

- Finite Difference Method (FDM)
- Finite Element Method (FEM)
- Mesh-based methods
- Fixed domain geometry

Deep Learning Approach:

- Mesh-free approach
- Automatic differentiation
- Potentially breaks curse of dimensionality
- Flexible domain handling

Physics-Informed Neural Networks (PINNs): Embed PDE constraints directly into the neural network loss function using automatic differentiation.

Feed-Forward Neural Networks (FNN)

Definition

An L -layer neural network $\mathcal{N}^L(\mathbf{x}) : \mathbb{R}^{d_{in}} \rightarrow \mathbb{R}^{d_{out}}$ with N_ℓ neurons in layer ℓ :

Input layer: $\mathcal{N}^0(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^{d_{in}}$

Hidden layers: $\mathcal{N}^\ell(\mathbf{x}) = \sigma(W^\ell \mathcal{N}^{\ell-1}(\mathbf{x}) + \mathbf{b}^\ell) \in \mathbb{R}^{N_\ell}$ for $1 \leq \ell \leq L-1$

Output layer: $\mathcal{N}^L(\mathbf{x}) = W^L \mathcal{N}^{L-1}(\mathbf{x}) + \mathbf{b}^L \in \mathbb{R}^{d_{out}}$

where $W^\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ and $\mathbf{b}^\ell \in \mathbb{R}^{N_\ell}$ are weights and biases.

Common activation functions:

- Sigmoid: $\sigma(x) = 1/(1 + e^{-x})$
- Tanh: $\sigma(x) = \tanh(x)$
- ReLU: $\sigma(x) = \max\{x, 0\}$

Why Automatic Differentiation?

Four methods to compute derivatives:

- 1 Hand-coded analytical derivatives
- 2 Finite difference approximations
- 3 Symbolic differentiation
- 4 **Automatic Differentiation (AD)** \Leftarrow Used in PINNs

Efficiency comparison:

- **Finite differences:** Requires $d_{in} + 1$ forward passes to compute all partial derivatives
- **AD (backpropagation):** Requires only **1 forward pass + 1 backward pass**

Why Automatic Differentiation?

Four methods to compute derivatives:

- 1 Hand-coded analytical derivatives
- 2 Finite difference approximations
- 3 Symbolic differentiation
- 4 **Automatic Differentiation (AD)** \Leftarrow Used in PINNs

Efficiency comparison:

- **Finite differences:** Requires $d_{in} + 1$ forward passes to compute all partial derivatives
- **AD (backpropagation):** Requires only **1 forward pass + 1 backward pass**
- AD is much more efficient for high-dimensional problems!

Key principle: AD applies the chain rule repeatedly to compute derivatives of composite functions.

Automatic Differentiation: Example

Simple network: $v = -2x_1 + 3x_2 + 0.5$, $h = \tanh v$, $y = 2h - 1$

Forward Pass (at $x_1 = 2, x_2 = 1$):

- $x_1 = 2, x_2 = 1$
- $v = -4 + 3 + 0.5 = -0.5$
- $h = \tanh(-0.5) \approx -0.462$
- $y = 2(-0.462) - 1 = -1.924$

Backward Pass (chain rule):

- $\frac{\partial y}{\partial y} = 1$
- $\frac{\partial y}{\partial h} = 2$
- $\frac{\partial y}{\partial v} = 2 \cdot \text{sech}^2(v) \approx 1.573$
- $\frac{\partial y}{\partial x_1} = 1.573 \cdot (-2) = -3.146$
- $\frac{\partial y}{\partial x_2} = 1.573 \cdot 3 = 4.719$

Key insight: All partial derivatives computed efficiently in two passes!

PDE Formulation

Consider a general PDE parameterized by λ :

$$\mathbf{f} \left(\mathbf{x}; \frac{\partial u}{\partial x_1}, \dots, \frac{\partial^2 u}{\partial x_i \partial x_j}, \dots; \lambda \right) = 0, \quad \mathbf{x} \in \Omega \quad (1)$$

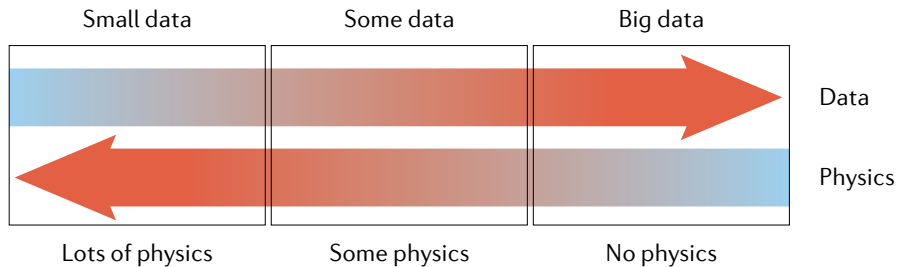
Boundary conditions:

$$B(u, \mathbf{x}) = 0, \quad \mathbf{x} \in \partial\Omega \quad (2)$$

Where:

- $u(\mathbf{x})$ is the solution
- $\mathbf{x} = (x_1, \dots, x_d)$ is in domain $\Omega \subset \mathbb{R}^d$
- B can be Dirichlet, Neumann, Robin, or periodic
- Time t is treated as a special component of \mathbf{x}

Data vs Physics



PINN Algorithm: Overview

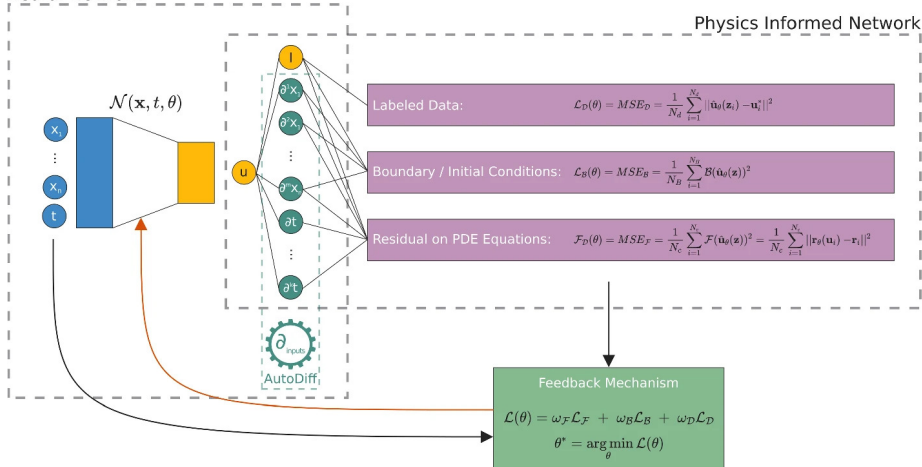
[PINN Algorithm for Solving PDEs]

- 1 **Construct** a neural network $\hat{u}(\mathbf{x}; \theta)$ as surrogate of $u(\mathbf{x})$
- 2 **Specify** training sets \mathcal{T}_f (domain points) and \mathcal{T}_b (boundary points)
- 3 **Define** loss function: weighted sum of PDE and BC residuals
- 4 **Train** the network by minimizing the loss

Key innovation: Use automatic differentiation to compute derivatives of \hat{u} directly!

PINNs: scheme

Neural Network



Loss Function

Define loss as weighted sum of residuals:

$$L(\boldsymbol{\theta}; \mathcal{T}) = w_f L_f(\boldsymbol{\theta}; \mathcal{T}_f) + w_b L_b(\boldsymbol{\theta}; \mathcal{T}_b) \quad (3)$$

PDE residual loss:

$$L_f(\boldsymbol{\theta}; \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} \left\| \mathbf{f} \left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \dots; \lambda \right) \right\|_2^2 \quad (4)$$

Boundary condition residual loss:

$$L_b(\boldsymbol{\theta}; \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x} \in \mathcal{T}_b} \|B(\hat{u}, \mathbf{x})\|_2^2 \quad (5)$$

Training: Minimize $L(\boldsymbol{\theta}; \mathcal{T})$ using gradient-based optimizers (Adam, L-BFGS)

Advantages of PINNs

- ➊ **Mesh-free:** No mesh generation required
- ➋ **Automatic differentiation:** No truncation or quadrature errors in derivatives
- ➌ **Forward AND inverse problems:** Same algorithm for both, minimal code change
- ➍ **General applicability:** Works for various PDE types:
 - Parabolic and elliptic PDEs
 - Integro-differential equations (IDEs)
 - Fractional differential equations (FDEs)
 - Stochastic differential equations (SDEs)
- ➎ **Complex domains:** Via constructive solid geometry (CSG)
- ➏ **High-dimensional problems:** Potentially avoids curse of dimensionality

Can Neural Networks Approximate PDE Solutions?

Theorem (Pinkus (1999))

If $\sigma \in C^m(\mathbb{R})$ is not a polynomial, then single hidden layer neural networks are dense in $C^{m_1, \dots, m_s}(\mathbb{R}^d)$.

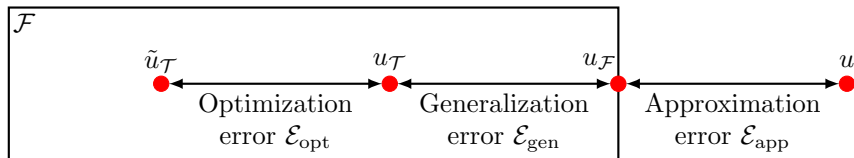
Interpretation:

- Neural networks with enough neurons can simultaneously approximate a function AND its partial derivatives
- This is a **fundamental theoretical justification** for PINNs

Limitation:

- Theorem is non-constructive (doesn't say how many neurons needed)
- Neural networks in practice have limited size

Error Decomposition



Total error decomposition:

$$E := \|\tilde{u}_T - u\| \leq \underbrace{\|\tilde{u}_T - u_T\|}_{\text{Opt. error}} + \underbrace{\|u_T - u_F\|}_{\text{Gen. error}} + \underbrace{\|u_F - u\|}_{\text{App. error}} \quad (6)$$

Error components:

- **Approximation error E_{app} :** How well the function family \mathcal{F} can approximate u
- **Generalization error E_{gen} :** Depends on number of residual points and network capacity
- **Optimization error E_{opt} :** From not achieving global minimum

Error Analysis: Key Points

- **Approximation error:** Smaller networks \Rightarrow larger approximation errors
- **Generalization error:** Larger networks \Rightarrow may lead to overfitting (bias-variance tradeoff)
- **No current error bounds:** Unlike FEM, we don't have guaranteed error estimates for PINNs
- **Non-uniqueness:** PINNs may converge to different solutions from different initializations
 - Common practice: Train multiple times, choose best
- **Hyperparameter tuning needed:**
 - Network size (depth and width)
 - Number and distribution of residual points
 - Learning rate
 - Loss weights w_f, w_b

PINNs vs Finite Element Method (FEM)

| Aspect | PINN | FEM |
|------------------|-----------------------------|--------------------------|
| Basis function | Neural network (nonlinear) | Polynomial (linear) |
| Parameters | Weights and biases | Point values |
| Training points | Scattered (mesh-free) | Mesh points |
| PDE embedding | Loss function | Algebraic system |
| Parameter solver | Gradient-based optimizer | Linear solver |
| Errors | $E_{app}, E_{gen}, E_{opt}$ | Approximation/quadrature |
| Error bounds | Not available yet | Partially available |

Fundamental difference: PINNs use **nonlinear** function approximation, whereas FEM uses **linear** approximation.

PINNs for Integro-Differential Equations (IDEs)

Problem: Solve IDE with integral term

$$\frac{dy}{dx} + y(x) = \int_0^x e^{t-x} y(t) dt \quad (7)$$

PINN approach:

- ① Use AD for analytical derivatives: $\frac{\partial \hat{u}}{\partial x}$
- ② Approximate integrals numerically: Gaussian quadrature

Discretization example:

$$\int_0^x e^{t-x} y(t) dt \approx \sum_{i=1}^n w_i e^{t_i(x)-x} y(t_i(x)) \quad (8)$$

This introduces a new error component: **discretization error** E_{dis}

Extension: Same approach works for FDEs and SDEs

Inverse Problems with PINNs

Problem: Identify unknown parameters λ given measurements

$$\mathbf{f}\left(\mathbf{x}; \frac{\partial u}{\partial x_1}, \dots; \lambda\right) = 0, \quad u(\mathbf{x}) = g_m(\mathbf{x}) \text{ (measurements)} \quad (9)$$

Solution approach: Add extra loss term for measurement data

$$L(\boldsymbol{\theta}, \lambda; \mathcal{T}) = w_f L_f(\boldsymbol{\theta}, \lambda; \mathcal{T}_f) + w_b L_b(\boldsymbol{\theta}, \lambda; \mathcal{T}_b) + w_i L_i(\boldsymbol{\theta}, \lambda; \mathcal{T}_i) \quad (10)$$

where

$$L_i(\boldsymbol{\theta}, \lambda; \mathcal{T}_i) = \frac{1}{|\mathcal{T}_i|} \sum_{\mathbf{x} \in \mathcal{T}_i} \|I(\hat{u}, \mathbf{x})\|_2^2 \quad (11)$$

Optimize: Both $\boldsymbol{\theta}$ and λ simultaneously

$$\boldsymbol{\theta}^*, \lambda^* = \arg \min_{\boldsymbol{\theta}, \lambda} L(\boldsymbol{\theta}, \lambda; \mathcal{T}) \quad (12)$$

Key advantage: Same framework for forward and inverse problems!

Residual-Based Adaptive Refinement (RAR)

Motivation: For PDEs with steep gradients, uniform sampling is inefficient

Idea: Add more residual points where PDE residual is large
[RAR Algorithm]

- 1 Select initial residual points and train for limited iterations
- 2 Estimate mean PDE residual by Monte Carlo sampling
- 3 If mean residual $> E_0$: Add m points with largest residuals
- 4 Repeat until convergence

Mean residual:

$$E_r = \frac{1}{V} \int_{\Omega} \left| \mathbf{f} \left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \dots; \lambda \right) \right| d\mathbf{x} \quad (13)$$

Benefit: Better capture of discontinuities and sharp fronts (e.g., Burgers equation)

Residual Points and Mini-Batch Training

Three strategies for selecting residual points:

- ➊ **Fixed points:** Specify at beginning, never change
 - Grid points or random points
- ➋ **Random resampling:** Select different points each iteration
 - Adds stochasticity
- ➌ **Adaptive refinement:** RAR method
 - Focus on high residual regions

Mini-batch training:

- For very large number of residual points: use mini-batches
- Each iteration uses subset of points
- Reduces computational cost per iteration
- Strategy (2) is special case of mini-batch training

Hard vs Soft Boundary Conditions

Soft constraints (used in PINNs):

- Boundary conditions enforced through loss function term L_b
- Flexible and general
- Works for complex domains
- Any type of BC: Dirichlet, Neumann, Robin, periodic

Hard constraints (alternative):

- Manually construct network to satisfy BC automatically
- Example: For $u(0) = u(1) = 0$, use $\hat{u}(x) = x(x-1)\mathcal{N}(x)$
- Advantages: Automatically satisfies BC, reduces degrees of freedom
- Limitations: Only for simple, homogeneous BCs

Best practice: Use soft constraints for generality, hard constraints when applicable

Summary: Key Takeaways

① **PINNs:** Embed PDEs into neural network loss using automatic differentiation

② **Advantages:**

- Mesh-free, highly flexible
- Forward and inverse problems unified
- Extends to many PDE types

③ **Theory:** Theoretical foundation via universal approximation

- Approximation, generalization, optimization errors identified
- No error bounds available yet

④ **Training challenges:**

- Hyperparameter tuning needed
- Non-convex optimization
- RAR method improves efficiency

⑤ **Emerging field:** Scientific Machine Learning

- Bridges classical numerical methods and deep learning
- Libraries like DeepXDE democratize implementation