Boundary Condition (BC) Implementation and Challenges in PINNs

Kristian Boroz

December 16, 2025

PINNs Loss Function and BCs

Boundary Conditions in the PINN Loss Function

- The total loss function minimized during training includes:
  - Physics Loss ($\mathcal{L}_{PDE}$)
  - Data Loss ($\mathcal{L}_{U}$)
  - Boundary/Initial Condition Loss ($\mathcal{L}_{BC/IC}$)
- The BC/IC loss measures the residual between $\mathcal{N}(\mathrm{X})$ and the boundary value $u_{BC}(\mathrm{X})$ at boundary points $\mathrm{X} \in \partial\Omega$.

Example: Loss with Dirichlet BC

1D Poisson Example

- PDE on $\Omega = (0, 1)$:
$$-u''(x) = f(x), \quad x \in (0, 1),$$
with Dirichlet BCs $u(0) = 0,\ u(1) = 0$.
- PINN loss:

$$\mathcal{L} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(-u''_{NN}(x_f^{(i)}) - f(x_f^{(i)})\right)^2 + \frac{1}{N_b} \sum_{j=1}^{N_b} \left(u_{NN}(x_b^{(j)}) - u_{BC}(x_b^{(j)})\right)^2.$$

Soft Constraints via Loss Minimization

Soft Constraints

- Penalize BC violations by including $\mathcal{L}_{BC/IC}$ in the total loss.
- **Pros:** Simple and general to implement.
- **Cons:** Creates a multi-term optimization problem, leading to residual boundary errors.

Example: Soft Dirichlet BC

Dirichlet BC as Penalty

- Same problem: $-u''(x) = 1$ on $(0, 1)$, with $u(0) = 0,\ u(1) = 0$.
- BC loss term for data points $x_b^{(1)} = 0$, $x_b^{(2)} = 1$:

$$\mathcal{L}_{BC} = \big(u_{NN}(0) - 0\big)^2 + \big(u_{NN}(1) - 0\big)^2.$$

- Total loss:

$$\mathcal{L} = \mathcal{L}_{PDE} + \lambda_{BC}\,\mathcal{L}_{BC}, \quad \lambda_{BC} > 0.$$

Hard Constraints: Concept and Transformation

Hard Constraints for Exact Imposition

- Enforced using a **mask function** (F_{mask}) applied to the output $\mathcal{N}(X)$.
- BC-compliant solution: $u = F_{mask}[\mathcal{N}(X)]$.
- Example: Custom `output_transform` in Diffusion-Reaction solvers enforces Dirichlet and initial BCs exactly.

Hard Constraints: Advantages and Drawbacks

Evaluation

- **Pros:** Exact BC satisfaction and faster convergence.
- **Cons:** Transformation F_{mask} must be problem-specific and carefully designed.
- Reduces needed derivative order, simplifying training when used effectively.

Dirichlet BC in DeepXDE

DeepXDE Implementation Overview

- BCs in DeepXDE are implemented through the `ICBC` module.
- Definition requires: geometry, boundary detection function (`onBoundary`), and constraint value.

Dirichlet BC

- Applied directly on the primary variable U: e.g. $U(-1) = 0$.
- No derivative calculations required; `model.predict` suffices for error evaluation.

Example: Dirichlet BC Formula

Dirichlet Condition

- Consider $u_t = u_{xx}$ on $(0, 1)$ with

$$u(0, t) = 0, \quad u(1, t) = 1.$$

- The Dirichlet BCs specify the value of $u(x, t)$ on the spatial boundary:

$$u(x, t)\big|_{x=0} = 0, \qquad u(x, t)\big|_{x=1} = 1.$$

Neumann BC in DeepXDE

Neumann BC

- Applies to the first derivative $\partial U/\partial X$ at the boundary.
- Requires Jacobian-based evaluation for computing gradients.
- Commonly used for flux continuity or gradient-based BC enforcement.

Example: Neumann BC Formula

Flux-Type Boundary

- Heat equation $u_t = \kappa u_{xx}$ on $(0, L)$ with insulated left end:

$$u_x(0,t) = 0.$$

- In a PINN, this BC term becomes

$$\mathcal{L}_N = \frac{1}{N_b}\sum_{j=1}^{N_b}\left(u_{NN,x}(0, t_b^{(j)}) - 0\right)^2,$$

where $u_{NN,x}$ is obtained via automatic differentiation.

Robin BC in DeepXDE

Robin BC

- Mixes Dirichlet and Neumann components.
- Written as $AU + B\frac{\partial U}{\partial X} = G$.
- Accepts constants or functional inputs for A, B, G.

Example: Robin BC Formula

Convective Boundary

- At $x = L$, heat equation with convection:

$$-ku_x(L,t) = h\big(u(L,t) - u_\infty\big).$$

- This can be written in Robin form as

$$Au(L,t) + Bu_x(L,t) = G$$

with $A = h,\ B = k,\ G = hu_\infty$.

Periodic BC in DeepXDE

Periodic BC

- Enforces continuity: $U(-1) = U(1)$.
- Useful for cyclic domains.
- Requires specifying spatial dimension (`component_x`) for periodicity.
- Works for U and its first derivative (not higher order).

Example: Periodic BC

Periodic Solution

- On domain $[-1,1]$, periodic BCs:

$$u(-1,t) = u(1,t), \qquad u_x(-1,t) = u_x(1,t).$$

- For a PINN, periodic BC loss:

$$\mathcal{L}_{per} = \frac{1}{N_b} \sum_{j=1}^{N_b} \left(u_{NN}(x_j^-) - u_{NN}(x_j^+)\right)^2 + \left(u_{NN,x}(x_j^-) - u_{NN,x}(x_j^+)\right)^2,$$

where x_j^- and x_j^+ are paired periodic points.

Automatic Differentiation for BCs

Automatic Differentiation (AD)

- Derivative-based BCs (Neumann, Robin, Operator) rely on AD.
- DeepXDE exposes gradient tools via its `grad` module.

Example: AD for Neumann BC

Computing Gradients

- Let $u_{NN}(x)$ be the network output.
- AD provides
$$u_{NN,x}(x) = \frac{\partial u_{NN}(x)}{\partial x},$$
which is evaluated at boundary points x_b and used in the Neumann loss
$$\mathcal{L}_N = \frac{1}{N_b} \sum_{j=1}^{N_b} \left(u_{NN,x}(x_b^{(j)}) - g(x_b^{(j)}) \right)^2.$$

Automatic Differentiation in PINNs

AD as the Engine Behind PDE Residuals

- For a network output $u_{NN}(X)$, AD builds a computational graph and applies the chain rule to compute
$$\frac{\partial u_{NN}}{\partial x_i}$$
exactly up to machine precision.
- Higher-order derivatives such as
$$\frac{\partial^2 u_{NN}}{\partial x_i \partial x_j}$$
are obtained by applying AD repeatedly on the same graph, enabling PDE residuals that depend on second derivatives.

AD in Practice: DeepXDE Interfaces

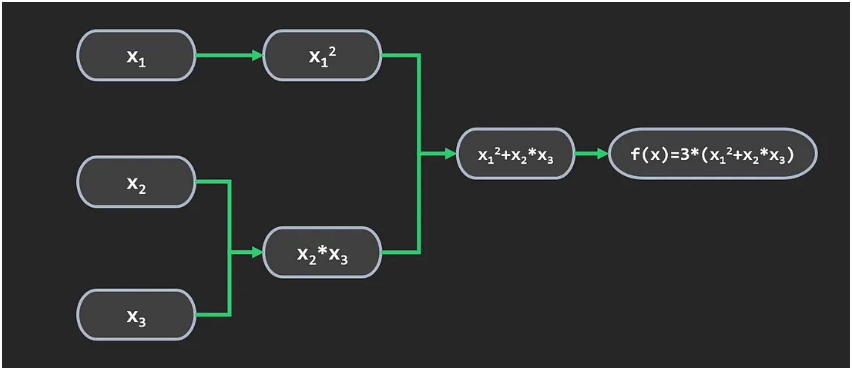
Jacobian and Hessian Calls

- DeepXDE wraps backend AD (`tf.gradients` / `torch.autograd.grad`) via `dde.grad.jacobian` and `dde.grad.hessian` to obtain Jacobians and Hessians used in PDE and BC residuals.
- In practice, calling

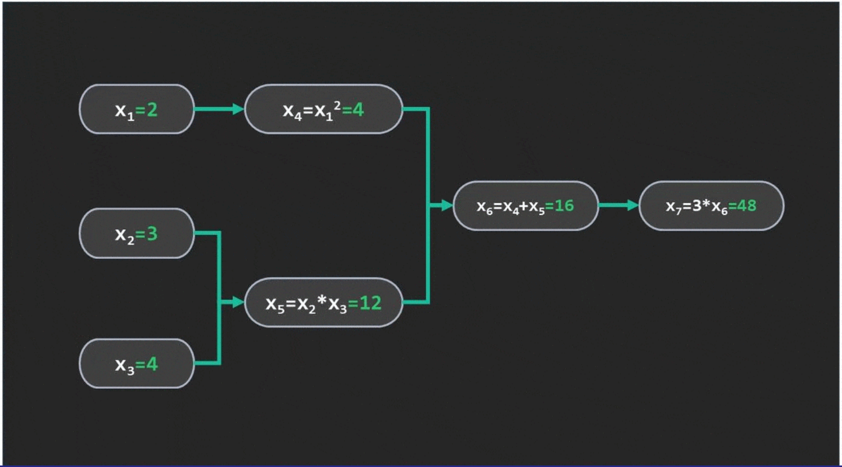
$$u_x = \texttt{dde.grad.jacobian}(u_{NN}, x), \quad u_{xx} = \texttt{dde.grad.hessian}(u_{NN}, x)$$

replaces manual finite differences and avoids truncation error in derivative-based BCs.

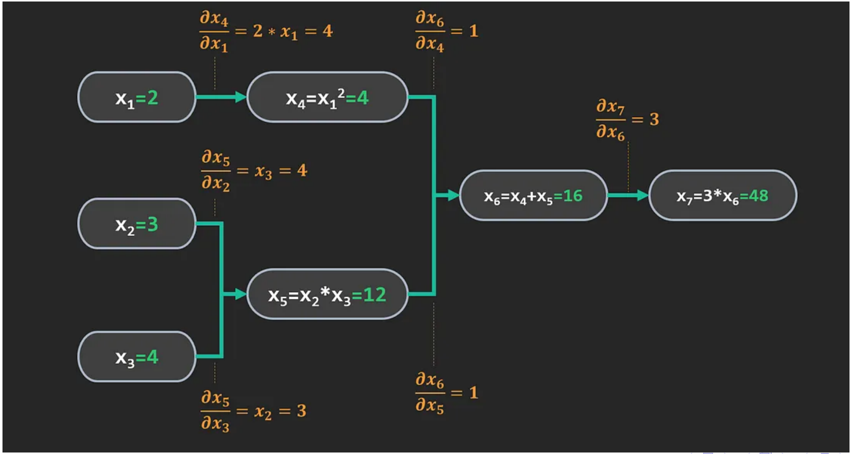
Automatic Differentiation



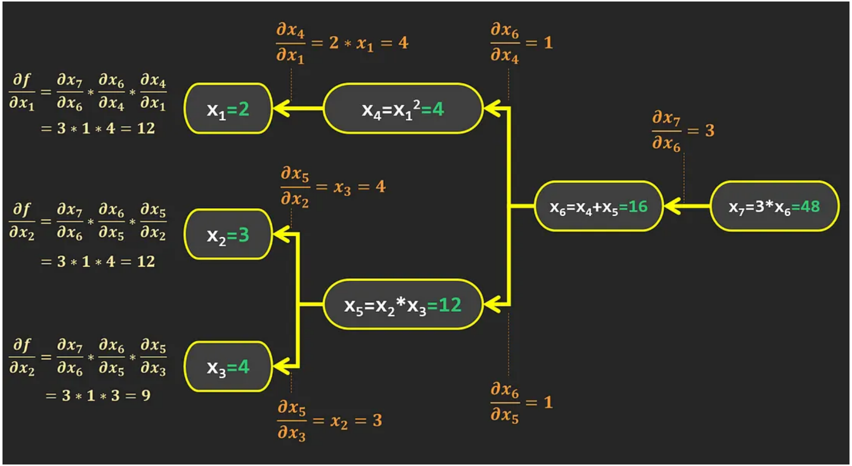
Automatic Differentiation



Automatic Differentiation



Automatic Differentiation



Derivatives: Jacobian and Hessian

- **Jacobian:** First-order derivatives, used in Neumann/Robin BCs.
- **Hessian:** Second-order derivatives, used for higher-order BCs.
- Higher-order derivatives are composed by chaining Jacobian and Hessian operators.

Example: Jacobian and Hessian

Second-Order PDE

- For $u : \mathbb{R}^d \to \mathbb{R}$, the Jacobian is

$$J_u(\mathrm{x}) = \nabla u(\mathrm{x}) = \begin{bmatrix} \partial_{x_1} u & \cdots & \partial_{x_d} u \end{bmatrix}.$$

- The Hessian is

$$H_u(\mathrm{x}) = \begin{bmatrix} \partial_{x_1 x_1} u & \cdots & \partial_{x_1 x_d} u \\ \vdots & \ddots & \vdots \\ \partial_{x_d x_1} u & \cdots & \partial_{x_d x_d} u \end{bmatrix},$$

and appears in PDEs like $\Delta u = \mathrm{tr}(H_u)$.

When Standard BCs Fail

Custom or Non-Standard Constraints

- Some problems require higher-order or nonlinear BCs.
- Standard classes (`DirichletBC`, `NeumannBC`, `RobinBC`) cannot handle these.

Example: Higher-Order BC

Euler-Bernoulli Beam

- Beam equation:

$$EI\, u''''(x) = q(x),$$

with clamped boundary at $x = 0$:

$$u(0) = 0, \quad u'(0) = 0.$$

- At a free end $x = L$, higher-order BCs may involve

$$u''(L) = 0, \quad u'''(L) = 0,$$

which standard BC classes cannot express directly.

The `OperatorBC` Class

- Enables defining arbitrary BCs using operators.
- Supports $U'' = 0$, $U''' = 0$, etc., as in the Euler beam equation.
- Can apply BCs at intermediate domain points.

Example: Operator-Type BC

Interior Constraint

- Suppose on $(0, 1)$ the solution must satisfy an interior constraint at x_c:

$$u''(x_c) = 0.$$

- An operator BC evaluates

$$\mathcal{B}[u](x_c) := u''(x_c),$$

and enforces $\mathcal{B}[u](x_c) = 0$ through a loss term

$$\mathcal{L}_{op} = \left(u''_{NN}(x_c)\right)^2.$$

Accessing the Network for BC Evaluation

Predict vs. Net Access

- `model.predict`: Uses NumPy data, no derivative access.
- `model.net`: Gives full access for gradient computation.

Example: Derivatives Need `net`

Evaluation Paths

- For $x \in \mathbb{R}$:
$$u_{NN}(x) = \texttt{model.net}(x),$$
and derivatives like $\partial_x u_{NN}(x)$ are computed via AD on the computational graph.
- Using `model.predict(x_np)` treats x as non-differentiable, so $\partial_x u_{NN}$ is not available.

Differentiable Input Tensors

- Derivative-based BCs (Neumann/Operator) require differentiable tensors.
- Use inputs with `requires_grad=True` (PyTorch).
- Simple Dirichlet BCs do not need this property.

Example: PyTorch Inputs for BCs

Gradient-Enabled Inputs

- Let $x \in \mathbb{R}^{N \times 1}$ be a tensor with `requires_grad=True`.
- Then

$$u = u_{NN}(x), \quad u_x = \frac{\partial u}{\partial x}$$

can be obtained via `torch.autograd.grad` and used to construct Neumann or Operator BC losses.

Evaluating BC Satisfaction

Residual Error via `.error()`

- `.error()` available for all BC types.
- Used post-training to assess boundary satisfaction.

Example: BC Residual

BC Error Metric

- For Dirichlet BC $u(x_b) = g(x_b)$ and collocation points $x_b^{(j)}$:

$$e_{BC} = \sqrt{\frac{1}{N_b}\sum_{j=1}^{N_b}\left(u_{NN}(x_b^{(j)}) - g(x_b^{(j)})\right)^2}.$$

- Values like $e_{BC} \approx 10^{-7}$–10^{-8} indicate very good but not exact satisfaction for soft BCs.

Interpreting Soft Constraint Errors

Residual Error in Practice

- Soft BCs minimize MSE of boundary residuals.
- Typically yield non-zero errors (10^{-7}–10^{-8}).
- Residual flexibility may hurt accuracy in complex PDEs (e.g., Burgers, Navier–Stokes).

Example: Soft vs. Hard BC Accuracy

Impact on Solution

- Let u^* be the exact solution and $\hat{u}$ the PINN solution.
- With soft BCs, boundary error

$$|\hat{u}(x_b) - u^*(x_b)| \approx 10^{-7}$$

  may still propagate into the interior solution for nonlinear PDEs.
- Hard BCs impose $\hat{u}(x_b) = u^*(x_b)$ exactly (up to machine precision), removing this source of error.