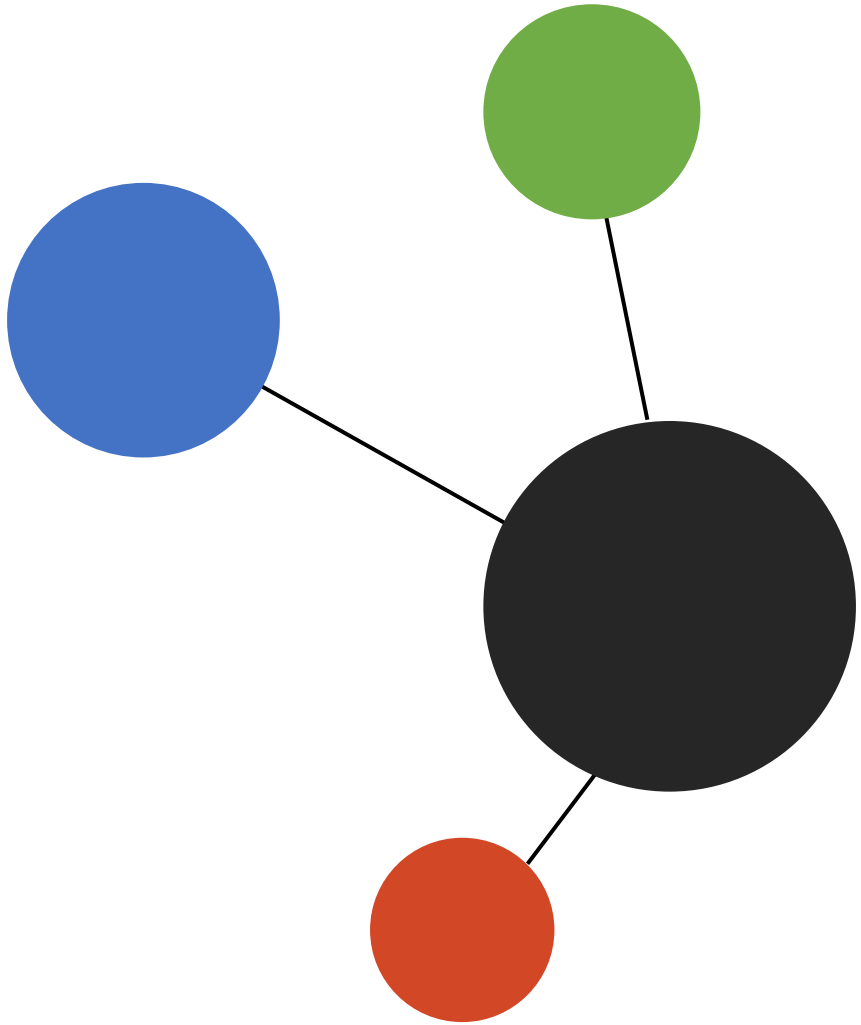
Physics-Informed Neural Networks (PINNs)Basics

By,
Faiq Shahbaz



Problem Setting: What Are We Solving?

- We want to solve a PDE of the general form:

$$u_t + \mathcal{N}[u] = 0$$

- unknown function (e.g. velocity, temperature).
- t: time, x: space.
- $\mathcal{N}[u]$:some (possibly nonlinear) expression involving u and its derivatives (for example, $uu_x - \nu u_{xx}$).
- We also have:
  - Initial condition (IC): $u(0, x) = u_0(x)$
  - Boundary conditions (BC): $u(t, x) = g(t, x)$ on the boundary of the spatial domain.

Core Idea of a PINN

- Use a neural network to approximate the solution:

$$u_\theta(t,x) \approx u(t,x)$$

- Train this neural network to satisfy:
  - Data (initial and boundary conditions).
  - The PDE (physics) inside the domain.

- Define a total loss function:

$$\text{Loss}(\theta) = \text{Loss}_{\text{data}}(\theta) + \text{Loss}_{\text{physics}}(\theta)$$

- $\text{Loss}_{\text{data}}$:mismatch at IC and BC points.
- $\text{Loss}_{\text{physics}}$:mismatch of the PDE residual (how much we violate the equation).

What Is Automatic Differentiation (AD)

- Automatic differentiation (AD) is how frameworks (PyTorch, TensorFlow, etc.) compute derivatives automatically.
- When you compute a function, the library builds a **computation graph**:
  - Nodes = intermediate values (results of operations).
  - Edges = operations (add, multiply, tanh, etc.).
- Each operation has a known derivative.
- AD runs a **backward pass** over this graph using the **chain rule** to compute derivatives:
  - With respect to parameters (weights) → standard backprop.
  - With respect to inputs (like t and x) → needed for u_t, u_x, u_{xx}, etc.

Computation Graph Example

- Consider:

$$f(x) = x^2 + 3x$$

- Break into intermediate nodes:
  - Node 0 (input): x
  - Node 1: a = x^2
  - Node 2: b = 3x
  - Node 3 (output): c = a + b (so f(x) = c)
- Local derivatives:

$$\frac{\partial a}{\partial x} = 2x, \frac{\partial b}{\partial x} = 3, \frac{\partial c}{\partial a} = 1, \frac{\partial c}{\partial b} = 1$$

- Total derivative via chain rule:

$$\frac{dc}{dx} = \frac{\partial c}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial c}{\partial b}\frac{\partial b}{\partial x}$$

$$\frac{dc}{dx} = 1 \cdot 2x + 1 \cdot 3 = 2x + 3$$

Tiny Neural Network Example

- Define a very small neural network:

$$z = w_1 t + w_2 x + b$$

$$u = \tanh(z)$$

- Here:
  - Inputs: t and x
  - Parameters: w_1, w_2, b
  - Output: u(t, x)
- Local derivatives of the li

$$\frac{\partial z}{\partial t} = w_1, \frac{\partial z}{\partial x} = w_2, \frac{\partial z}{\partial w_1} = t, \frac{\partial z}{\partial w_2} = x, \frac{\partial z}{\partial b} = 1$$

Activation Derivative

- We have: $u = \tanh(z)$
- Known calculus result:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z)$$

- Therefore:

$$\frac{\partial u}{\partial z} = 1 - \tanh^2(z)$$

- Since $u = \tanh(z)$,we can write:

$$\tanh^2(z) = u^2$$

- So:

$$\frac{\partial u}{\partial z} = 1 - u^2$$

Derivatives from the Tiny Network

- Using chain rule for input derivatives:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial z}\frac{\partial z}{\partial t} = (1 - u^2)\, w_1$$

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial z}\frac{\partial z}{\partial x} = (1 - u^2)\, w_2$$

- Derivatives w.r.t. parameters (like in backprop):

$$\frac{\partial u}{\partial w_1} = \frac{\partial u}{\partial z}\frac{\partial z}{\partial w_1} = (1 - u^2)\, t$$

$$\frac{\partial u}{\partial w_2} = (1 - u^2)\, x$$

$$\frac{\partial u}{\partial b} = (1 - u^2)\, 1 = 1 - u^2$$

- AD in frameworks computes these automatically from the computation graph.

Designing the Neural Network in a PINN

- We design a neural network NN_θ to approximate the solution:

$$u_\theta(t, x) = \mathrm{NN}_\theta(t, x)$$

- Inputs:
  - All independent variables, e.g. (t, x) for 1D time-dependent problems.
- Outputs:
  - The unknown field(s), e.g. scalar u or vector (u, v, p) for systems.
- Typical architecture (for PINNs):
  - 4–10 hidden layers.
  - 20–100 neurons per layer.
  - Smooth activation functions: commonly tanh or sin.

Types of Training Points

- **Initial condition (IC) points**:
  - Points on t = 0, with known values:
  - $u(0, x_i) = u_0(x_i)$
- **Boundary condition (BC) points**:
  - Points on spatial boundaries, e.g. x = -1 and x = 1.
  - Values given by the boundary condition function g(t, x).
- **Collocation (interior) points**:
  - Randomly sampled points inside the space–time domain.
  - No target u value is given.
  - Used to enforce that the PDE residual is close to zero.

Example PDE: Burgers' Equation

- Burgers' equation:

$$u_t + uu_x - \nu u_{xx} = 0$$

- Domain:

$$x \in [-1,1], t \in [0,1]$$

- Viscosity:

$$\nu = \frac{0.01}{\pi}$$

- Initial condition:

$$u(0, x) = -\sin(\pi x)$$

- Boundary conditions:

$$u(t, 1) = 0, \qquad u(t, -1) = 0$$

PINN Approximation for Burgers

- Define neural network solution:

$$u_\theta(t,x) = \mathrm{NN}_\theta(t,x)$$

- At any point (t, x):
  - Network outputs an approximation of u(t, x).
- To enforce the PDE, we need:

$$u_t(t,x) = \frac{\partial u_\theta}{\partial t}(t,x)$$

$$u_x(t,x) = \frac{\partial u_\theta}{\partial x}(t,x)$$

$$u_{xx}(t,x) = \frac{\partial^2 u_\theta}{\partial x^2}(t,x)$$

- These derivatives are computed using automatic differentiation.

PDE Residual Using AD

- At collocation point (t, x):
  - Compute $u = u_\theta(t, x)$.
  - Use AD to compute:

$$u_t = \frac{\partial u_\theta}{\partial t}(t, x)$$

$$u_x = \frac{\partial u_\theta}{\partial x}(t, x)$$

$$u_{xx} = \frac{\partial^2 u_\theta}{\partial x^2}(t, x)$$

- Define the PDE residual:

$$f_\theta(t, x) = u_t + u, u_x - \nu u_{xx}$$

- If the PDE is satisfied exactly, then:

$$f_\theta(t, x) = 0$$

Loss Terms for Burgers' PINN

- **Initial condition loss** (Loss_{IC}):
  - For IC points
$$\text{Loss}_{\text{IC}} = \frac{1}{N_0}\sum_{i=1}^{N_0}\left(u_\theta(0, x_i) - (-\sin(\pi x_i))\right)^2$$
- **Boundary condition loss** (Loss_{BC}):
  - For boundary points
$$\text{Loss}_{\text{BC}} = \frac{1}{N_b}\sum_{j=1}^{N_b}\left(u_\theta(t_j, -1)\right)^2 + \frac{1}{N_b}\sum_{j=1}^{N_b}\left(u_\theta(t_j, 1)\right)^2$$
- **PDE residual loss** (Loss_{PDE}):
  - For collocation points (t_k, x_k):
  - $\text{Loss}_{\text{PDE}} = \frac{1}{N_f}\sum_{k=1}^{N_f} f_\theta(t_k, x_k)^2$
- **Total loss**:
  - $\text{Loss}(\theta) = \text{Loss}_{\text{IC}} + \text{Loss}_{\text{BC}} + \text{Loss}_{\text{PDE}}$

Training Pipeline (Algorithm View)

- **Step 1:** Choose neural network architecture.
  - Input size = 2 (t, x), output size = 1 (u).
- **Step 2:** Initialize network parameters θ randomly.
- **Step 3:** Sample three sets of points:
  - IC points: $(0, x_i)\ with\ u_0(x_i) = -\sin(\pi x_i)$.
  - BC points: $(t_j, -1)\ and\ (t_j, 1)$with value 0.
  - Collocation points: (t_k, x_k)randomly in [0, 1] ×[-1, 1].
- **Step 4:** For each training iteration:
  - Compute u_θ at IC and BC points and evaluate $\mathrm{Loss}_{\mathrm{IC}}\ and\ \mathrm{Loss}_{\mathrm{BC}}$.
  - At collocation points, use AD to compute $u_t, u_x, u_{xx}\ and\ then\ f_\theta(t_k, x_k)$.
  - Compute $\mathrm{Loss}_{\mathrm{PDE}}$ from residuals.
  - Sum to get total $\mathrm{Loss}(\theta)$.
  - Use backpropagation (AD) to compute $\nabla_\theta \mathrm{Loss}$.
  - Update θ with an optimizer (e.g. gradient descent, Adam).