# The DeepONet Method

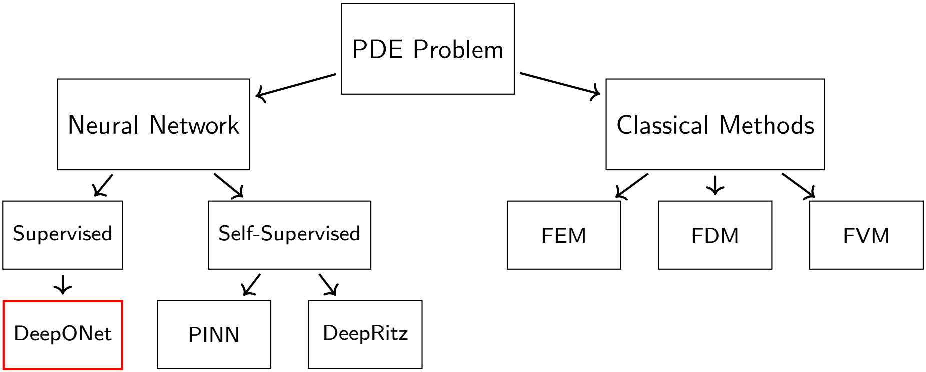
## Learning Nonlinear Operators for Differential Equations

Vincent Scholz & Dr. Yaohua Zang

Professorship for Data-driven Materials Modeling

June 18, 2025

# Overview



PDE Problem
Neural Network
Classical Methods
Supervised
Self-Supervised
FEM
FDM
FVM
DeepONet
PINN
DeepRitz

# Comparison: DNOs vs Solution Operator Networks

| Aspect | Solution Operator Network | Deep Neural Operator |
|---|---|---|
| **Input** | Discretized vector $\vec{a}$ | Function $a(x)$ |
| **Output** | Discretized vector $\vec{u}$ | Function $u(x)$ |
| **Mapping** | Vector-to-vector | **Function-to-function** |
| **Training Data** | Paired vectors $(\vec{a}, \vec{u})$ | Paired functions $(a(x), u(x))$ |
| **Flexibility** | Limited to specific discretizations | **Mesh-independent** |

**Solution Operator:** $\mathcal{K}_\theta(\vec{a}) : \vec{a} \in \mathbb{R}^m \to \vec{u} \in \mathbb{R}^n$

**DNO:** $\mathcal{G}_\theta(a) : a(x) \in \mathcal{A} \to u(x) \in \mathcal{U}$

# Motivation for later

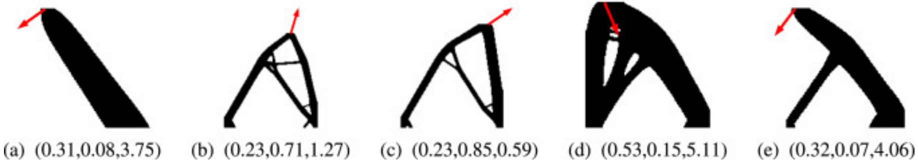



Figure: Goal: Predict stresses of different structures and loads.

# Table of Contents

# Table of Contents

# Introduction to DeepONet

- DeepONet is a foundational architecture in **deep neural operators**
- Learns operators that map input functions to output functions
- Unlike standard NN: learns **function-to-function mapping**
- **Key innovation**: separates handling of input functions and spatial coordinates

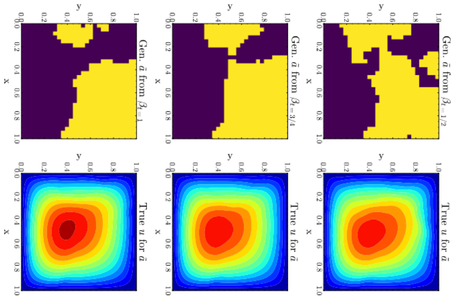



Figure: Darcy's flow function to function example.

# DeepONet Architecture Structure

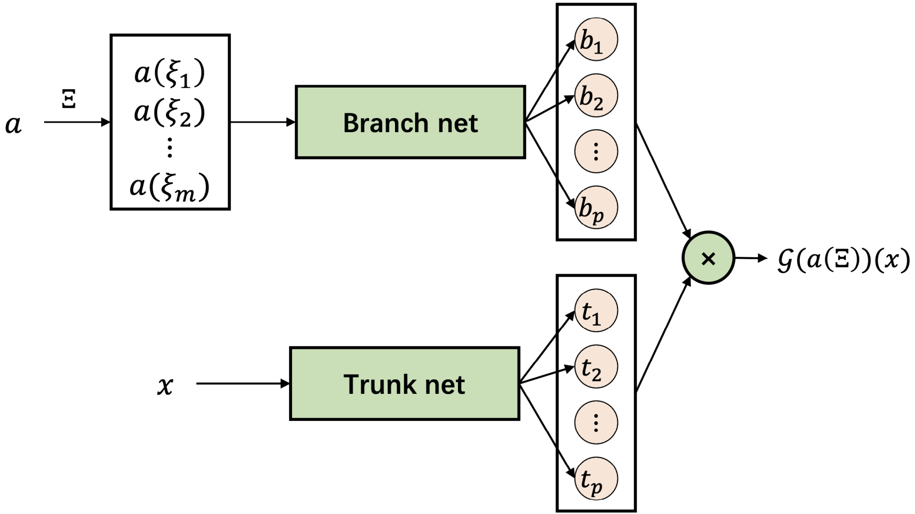



Figure: The architecture of a DeepONet.

# The Core Idea

## DeepONet Architecture

Two neural networks working in parallel:

- **Branch network**: Encodes the input function (e.g., coefficient field, initial condition)
- **Trunk network**: Encodes evaluation location (spatial point $x$ or $(x, t)$)

Together they produce:

$$\boxed{\mathcal{G}_\theta(a)(x) \approx u(x)}$$

# Branch Network - Representing Input Functions

## Challenge

Input function $a(x)$ cannot be directly input into neural network

**Two approaches for finite representation:**

1. **Basis Expansion Representation**:

$$a(x) = \sum_i a_i \phi_i(x)$$

   Use first $m$ coefficients $(a_1, a_2, \ldots, a_m)$

2. **Sensor-based Discretization**:

$$a(\Xi) = (a(\xi_1), a(\xi_2), \ldots, a(\xi_m))$$

   Sample at predefined sensor points $\Xi = \{\xi_1, \xi_2, \ldots, \xi_m\}$

**Output:** $p$ intermediate terms $b = (b_1(a), b_2(a), \ldots, b_p(a))$

# Trunk Network & Output Combination

## Trunk Network

- Input: evaluation point $x \in \Omega$
- Output: $p$ intermediate terms $t = (t_1(x), t_2(x), \dots, t_p(x))$
- Encodes spatial location information

## Combining Outputs

Final prediction via inner product + bias:

$$\boxed{\mathcal{G}_\theta(a(\Xi))(x) = b \odot t + b_0 = \sum_{k=1}^{p} b_k(a(\Xi)) \cdot t_k(x) + b_0}$$

**This formulation separately learns:**

- How **input function** affects solution (branch network)
- How **location** affects solution (trunk network)

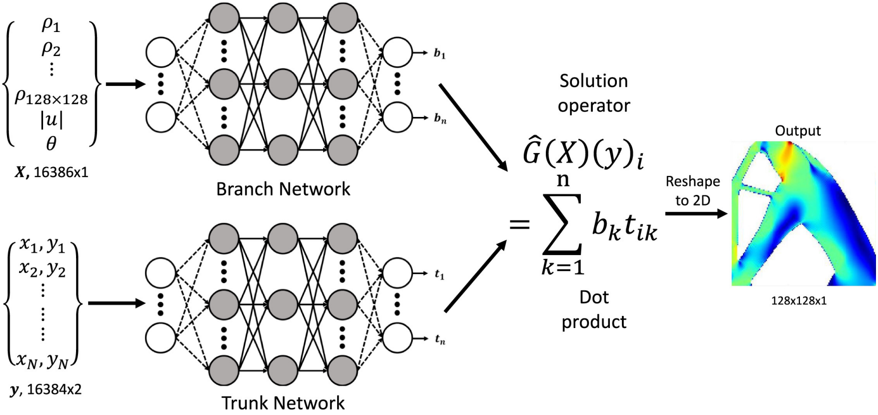



Figure: DeepONet for structures.

# Table of Contents

# DeepONet Method Overview

## Supervised Learning Approach

Learn mapping from input function to PDE solution

**Four Key Steps:**

1. Collect labeled training pairs → 2. Build DeepONet model → 3. Define loss function → 4. Train with gradient descent

# Step 1: Collecting Training Data

## Dataset Structure

$$\mathcal{D} = \left\{ \left( a^{(i)}(\Xi), u^{(i)} \right) \right\}_{i=1}^{N_{\text{data}}}$$

- $a^{(i)}(\Xi)$: Discretized input function for sample $i$
- $u^{(i)}$: Corresponding PDE solution

**How to obtain this data:**

- Input functions $a^{(i)}$ sampled from function space $\mathcal{A}$
- Output solutions $u^{(i)}$ computed using numerical methods:
  - Finite difference methods
  - Finite element methods
  - Spectral methods

# Step 2: Building the DeepONet Model

**Trunk Network:**

- Input: spatial coordinates $x \in \Omega$
- Output: vector $t(x) = (t_1(x), \ldots, t_p(x))$
- Implementation: MLP

**Branch Network:**

- Input: discretized $a(\Xi)$
- Output: vector $b(a) = (b_1(a), \ldots, b_p(a))$
- Implementation:
  - MLP (low-dim)
  - CNN (high-dim)

## Combined Output

$$\mathcal{G}_\theta(a(\Xi))(x) = \sum_{k=1}^{p} b_k(a(\Xi)) \cdot t_k(x) + b_0$$

# Step 3: Defining the Loss Function

## Mean Squared Error (MSE) Loss

$$L(\theta) = \frac{1}{N_{\text{data}}} \sum_{i=1}^{N_{\text{data}}} \sum_{j=1}^{N_p} \left| \mathcal{G}_\theta(a^{(i)}(\Xi))(x_j) - u^{(i)}(x_j) \right|^2$$

**Components:**

- $\{x_j\}_{j=1}^{N_p}$: Collocation (evaluation) points in domain
- Objective: find optimal parameters $\theta^* = \arg\min_\theta L(\theta)$

## Note

Other loss functions possible:

- Relative error
- Physics-informed losses
- Custom application-specific losses

# Step 4: Training with Gradient Descent

## Iterative Parameter Updates

$$\theta_{t+1} = \theta_t - l_r \nabla_\theta L(\theta_t)$$

- $l_r$: Learning rate
- $t$: Current iteration (epoch)

**Training continues until:**

- Loss converges to satisfactory level
- Maximum number of epochs reached

# Table of Contents

# DeepONet: Advantages and Limitations

**Overview**

DeepONet is powerful for learning PDE solution operators, especially when fast inference is required across many different inputs.

**Advantages**

- Function-to-function learning
- Fast inference after training
- Handles varying inputs

**Limitations**

- Data inefficiency
- Poor generalization
- Training cost

# Advantages of DeepONet

## [1] Function-to-Function Learning

- Unlike traditional NNs operating on fixed-size vectors
- Specifically designed for mappings between functions
- Suitable for PDE families with varying input functions

## [2] Fast Inference After Training

- Instantly predict solution $u(x)$ for new input $a(x) \in \mathcal{A}$
- No need to solve PDE again
- Valuable for: real-time simulation, control, optimization

# Training time / Prediction time

Table 9. Training and prediction time for all the learning cases.

| Case | Training time [s] (1 epoch) | | Prediction time [s] | |
|---|---|---|---|---|
| | DeepOnet | Ex-DeepOnet | DeepOnet | Ex-DeepOnet |
| Anti-derivative-Tanh | 2.040 | 2.274 | 1.98e−3 | 2.23e−3 |
| Anti-derivative-PI-Tanh | 2.707 | 3.149 | 2.02e−3 | 2.01e−3 |
| Darcy-Flow | 1.056 | 1.762 | 2.91e−3 | 2.88e−3 |
| Advection-PI | 2.564 | 2.632 | 1.13e−3 | 1.04e−3 |
| Pendulum | 3.891 | 5.376 | 1.36e−3 | 1.56e−3 |
| Pendulum-PI | 10.43 | 10.39 | 1.33e−3 | 1.11e−3 |
| Allen–Cahn | 19.05 | 41.61 | 3.17e−3 | 3.13e−3 |
| Burgers | 6.720 | 14.38 | 1.71e−3 | 1.65e−3 |
| Burgers-PI | 30.10 | 42.48 | 1.74e−3 | 1.66e−3 |

Figure: Table of training vs prediction times for multiple generic problems.

# Limitations of DeepONet

## [1] Data Inefficiency

- Requires large number of labeled pairs $(a, u)$
- Generating data $u$ computationally expensive
- Need high-quality numerical methods (FEM, spectral, etc.)

## [2] Poor Generalization to Unseen Inputs

- Performs poorly on out-of-distribution inputs
- Not robust if test input $a$ outside training distribution
- Limits reliability unless training set very comprehensive

# When to Choose DeepONet

## Ideal Use Cases

- Multiple queries on similar PDE families
- Real-time applications requiring fast inference
- Parameter studies and optimization
- Uncertainty quantification

## Consider Alternatives When

- Limited training data available
- High out-of-distribution robustness required
- Single-query scenarios
- Extremely high accuracy needed

# Conclusion

## Key Takeaways

- DeepONet learns function-to-function mappings for PDE operators
- Branch-trunk architecture separates input and spatial encoding
- Supervised learning approach requires substantial training data
- Excellent for fast inference, limited by generalization

## Future Directions

- Improved generalization techniques
- Physics-informed training
- Hybrid approaches combining DeepONet with traditional methods

**Thank you for your attention!**

# Physics-informed learning methods

- A hands-on introduction to Physics-Informed Neural Networks for solving partial differential equations with benchmark tests taken from astrophysics and plasma physics: This is a paper that provides an introduction to the application of PINN for solving partial differential equations (PDEs).
- DeepXDE: a library for scientific machine learning and physics-informed learning. It includes the following algorithm:
  - physics-informed neural network (PINN)
  - (physics-informed) deep operator network (DeepONet)
  - multifidelity neural network (MFNN)
- https://github.com/junbinhuang/DeepRitz: A Github repository for implementation of the Deep Ritz method and the Deep Galerkin method
- https://github.com/yaohua32/Physics-Driven-Deep-Learning-for-PDEs: A Github repository for Physics-Driven Deep Learning for PDEs and Inverse Problems. It includes the implementation of the following methods: PINN and ParticleWNN.

# Data-driven deep neural operators

- https://github.com/neuraloperator/neuraloperator: A Github repository for learning in infinite dimensions with neural operators.
- https://github.com/lu-group/deeponet-fno: A Github repository for implementation of DeepONet & FNO (with practical extensions).
- https://github.com/yaohua32/Physics-Driven-Deep-Learning-for-PDEs: A Github repository for Deep Neural Operator Frameworks for Solving Parametric PDEs. It includes the implementation of the following data-driven DNOs:
  - DeepONet
  - Fourier Neural Operator (FNO)
  - MultiONet