# Introduction to DeepONet

Yaohua Zang

June 23, 2025

## 0.1 What is DeepONet? — The DeepONet Architecture

DeepONet is one of the foundational architectures in the family of **deep neural operators (DNOs)**, designed specifically to learn operators that map input functions to output functions. Unlike standard neural networks that map vectors to vectors, DeepONet learns a **function-to-function mapping** by cleverly separating the handling of input functions and spatial coordinates.

### 0.1.1 The Core Idea

The DeepONet architecture consists of two neural networks working in parallel:

- **Branch network**: Encodes the input function (e.g., a coefficient field like permeability or initial condition).
- **Trunk network**: Encodes the evaluation location (e.g., a spatial point $x$, or spatial-temporal point $(x, t)$) at which we want to evaluate the output solution.

Together, these two components produce an approximation of the solution operator:

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

### 0.1.2 Structure of DeepONet

The structure of the DeepONet architecture is illustrated in Figure 1. Let's break down each component:

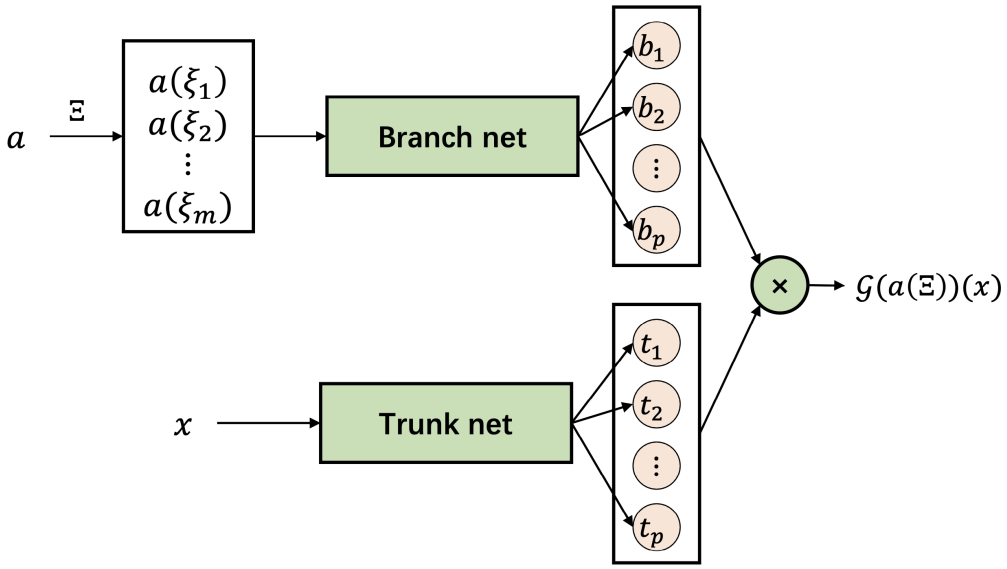


Figure 1: The architecture of DeepONet

### Branch Network — Representing the Input Function

The branch network takes a finite representation of the input function $a(x)$, which cannot be directly input into a neural network. There are two common approaches to obtain this representation:

- **Basis Expansion Representation**:
  - Represent the function as a sum of basis functions:
$$a(x) = \sum_i a_i \phi_i(x)$$
  - Use the first $m$ coefficients $(a_1, a_2, \ldots, a_m)$ as input to the branch network.
- **Sensor-based Discretization**:
  - Sample the function values at a set of predefined sensor points $\Xi = \{\xi_1, \xi_2, \ldots, \xi_m\}$:
$$a(\Xi) = (a(\xi_1), a(\xi_2), \ldots, a(\xi_m))$$
  - This vector of sampled values serves as the input to the branch network.

The branch network processes this input and outputs a set of $p$ intermediate terms:

$$b = (b_1(a), b_2(a), \ldots, b_p(a)).$$

### Trunk Network — Representing the Spatial Coordinates

The trunk network takes the evaluation point $x \in \Omega$ as input and also outputs $p$ intermediate terms:

$$t = (t_1(x), t_2(x), \ldots, t_p(x)).$$

These terms encode information about where in the domain the solution is being evaluated.

### Combining the Outputs of Branch Network and Trunk Network

Finally, the outputs of the branch and trunk networks are combined via an inner product, followed by the addition of a bias term:

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

where $b_k(a(\Xi))$ and $t_k(x)$ are the $k$-th components of the outputs from the branch and trunk networks, respectively, and $b_0$ is a bias term. This formulation allows the model to separately learn:

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

and then combine them dynamically at inference time to produce the desired solution at any location $x$.

## 0.2 How to Use DeepONet — The DeepONet Method

The DeepONet method is a **supervised learning approach** that learns the mapping from an input function (e.g., a coefficient or source term) to the corresponding solution of a PDE. The procedure involves four key steps:

### Step 1: Collecting Labeled Training Pairs

Since DeepONet is a data-driven model, the first and most essential step is to collect a dataset of input-output pairs:

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

- $a^{(i)}(\Xi)$: Discretized or feature-based representation of the input function for sample $i$.
- $u^{(i)}$: Corresponding solution of the PDE.

**How to get this data?**

- The input functions $a^{(i)}$ can be sampled randomly or from a predefined function space $\mathcal{A}$.
- The output solutions $u^{(i)}$ must be computed using accurate numerical methods (e.g., finite difference, finite element, spectral methods).

### Step 2: Building the DeepONet Model

To approximate the unknown operator $\mathcal{G}$, we use the DeepONet architecture $\mathcal{G}_\theta$, composed of two neural networks:

**Trunk Network:**

- Takes spatial coordinates $x \in \Omega$ as input.
- Outputs a vector $t(x) = (t_1(x), \ldots, t_p(x))$.
- Typically implemented using a Multilayer Perceptron (MLP).

**Branch Network:**

- Takes the discretized input function $a(\Xi)$ as input.
- Outputs a vector $b(a) = (b_1(a), \ldots, b_p(a))$.
- Implemented as:
  - MLP if $a(\Xi)$ is a low-dimensional vector.
  - CNN if $a(\Xi)$ is high-dimensional (e.g., image-like structure).

**Combined Output:** The DeepONet prediction at location $x$ is given by:

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

where

- $b_0$ is a learnable bias term.
- The model parameters $\theta$ include all weights in the trunk and branch networks, plus $b_0$.

## Step 3: Defining the Loss Function

To train DeepONet, we aim to minimize the discrepancy between its predictions and the true PDE solutions across all training samples and evaluation points. This is typically formulated as a **mean squared error (MSE)** loss:

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

- $\{x_j\}_{j=1}^{N_p}$: Set of collocation (evaluation) points in the domain.

The objective is to find optimal parameters:

$$\theta^* = \arg\min_\theta L(\theta)$$

**Note:** Other loss functions (e.g., relative error, custom physics-informed losses) may also be used depending on the application.

## Step 4: Training with Gradient Descent

The model is trained using a gradient-based optimization algorithm such as Stochastic Gradient Descent (SGD) or Adam. The parameters are updated iteratively as:

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

- $l_r$: Learning rate.
- $t$: Current iteration (also called an epoch).

Training continues until the loss converges to a satisfactory level or a maximum number of epochs is reached.

### Summary of DeepONet Usage

| Step | Description |
|---|---|
| Step 1 | Generate training data $(a^{(i)}, u^{(i)})$ via simulation or experiment |
| Step 2 | Build DeepONet with trunk and branch networks |
| Step 3 | Define a loss function based on prediction error |
| Step 4 | Train the model with gradient descent |

Table 1: Summary of the DeepONet method

# When Should We Use DeepONet? — Advantages and Limitations

DeepONet is a powerful tool for learning solution operators to PDEs, especially in scenarios where fast inference is required across many different inputs. However, like any method, it comes with both strengths and trade-offs.

## 0.2.1 Advantages of DeepONet

**Function-to-Function Learning**

- Unlike traditional neural networks that operate on fixed-size vectors, DeepONet is specifically designed to learn mappings between functions.
- This makes it suitable for solving families of PDEs with varying input functions (e.g., changing coefficients or source terms).

**Fast Inference After Training**

- Once trained, DeepONet can instantly predict the solution $u(x)$ for any new input function $a(x) \in \mathcal{A}$ without solving the PDE again.
- This is particularly valuable in scenarios where many queries need to be answered quickly, such as real-time simulation, control, or optimization.

## 0.2.2 Limitations of DeepONet

**Data Inefficiency**

- Training DeepONet requires a large number of labeled input-output pairs $(a, u)$.
- Since generating high-quality solutions $u$ often involves solving the PDE numerically (e.g., with finite element or spectral methods), this process can be computationally expensive and time-consuming.

**Poor Generalization to Unseen Inputs**

- DeepONet tends to perform poorly if the input function $a$ during testing lies outside the distribution of the training data.
- In other words, the method is not robust to out-of-distribution inputs, which limits its reliability in real-world settings unless the training set is very comprehensive.