Aalto University
School of Science

Master's Programme in Mathematics and Operations Research

# Physics-Informed Neural Operators for Optimal Control: A Comparative Study of DeepONet, Fourier, and Laplace Neural Operators

**Milana Begantsova**

Master's Thesis
2025

**Aalto University**
**School of Science**

**Author** Milana Begantsova

**Title** Physics-Informed Neural Operators for Optimal Control: A Comparative Study of DeepONet, Fourier, and Laplace Neural Operators

**Degree programme** Mathematics and Operations Research

**Major** Systems and Operations Research

**Supervisor** Prof. Fabricio Oliveira

**Advisor** MSc Oliver Lundqvist

**Date** 31 July 2025 **Number of pages** 86+9 **Language** English

**Abstract**
Neural operators have emerged as powerful surrogates for learning solution maps of differential equations. Yet, their application to optimal control remains underexplored. This thesis presents a unified control framework based on Physics-Informed Neural Operators (PINO) for solving parametric optimal control problems governed by ordinary differential equations. We investigate three prominent architectures - DeepONet, Fourier Neural Operator (FNO), and Laplace Neural Operator (LNO) - and train them using an unsupervised approach.

The framework is benchmarked on a suite of analytically solvable control problems, including linear-quadratic regulation, oscillatory forcing, polynomial tracking, and singular arc problems.

Our results reveal trade-offs across architectures: FNO achieves the fastest convergence, DeepONet excels in initial condition enforcement and prediction accuracy, while LNO provides competitive solutions on transient-rich problems. All experiments are reproducible, and the training pipeline is architecture-agnostic.

This work provides the first comparative study of neural operator architectures in a PINO-based optimal control setting and offers insights into their suitability for physics-constrained learning tasks.

**Keywords** Neural operators, Optimal control, Physics-informed learning, DeepONet, FNO, LNO

## Preface

This thesis is the final step of my Master's studies in Operations Research and Systems Analysis at Aalto University. Working on this project has been both challenging and rewarding, giving me the opportunity to dive deep into topics I truly enjoy - especially control theory and machine learning.

I would like to sincerely thank my supervisor, Prof. Oliveira, for his guidance and support throughout this work. I'm also deeply grateful to my advisor, Oliver, whose insightful feedback and well-structured codebase provided a strong foundation for this project.

To my partner - thank you for your endless patience, encouragement, and steady supply of fresh coffee. This thesis would not have been possible without you.

Otaniemi, 31 July 2025

Milana B.

# Contents

# Symbols and Abbreviations

## Symbols

| | |
|---|---|
| $t$ | time variable |
| $x(t) \in \mathbb{R}^n$ | system state at time $t$ |
| $u(t) \in \mathbb{R}^m$ | control input at time $t$ |
| $\dot{x}(t)$ | time derivative of state: $\frac{dx}{dt}$ |
| $J(u)$ | cost function to be minimized |
| $F(x, u, t)$ | running cost |
| $G(x)$ | terminal cost |
| $\mathcal{D}_\theta$ | neural operator with parameters $\theta$ |
| $\mathcal{R}$ | residual of the governing differential equation |
| $\lambda(t)$ | costate (adjoint) variable |
| $\mu$ | penalty weight for physics residual |
| $\Delta t$ | discretization step size |
| $N$ | number of time discretization points |

## Operators

| | |
|---|---|
| $\frac{d}{dt}$ | total derivative with respect to time |
| $\frac{\partial}{\partial t}$ | partial derivative with respect to time |
| $\\| \cdot \\|_{\mathcal{L}_2}$ | $\mathcal{L}_2$ norm (squared-integral over time domain) |
| $\mathcal{F}, \mathcal{F}^{-1}$ | Fourier transform and its inverse |
| $\mathcal{L}, \mathcal{F}^{-1}$ | Laplace transform and its inverse |
| $\mathcal{D}_\theta(u)$ | learned state trajectory for control input $u$ |
| $\arg\min$ | argument that minimizes an objective function |

## Abbreviations

| | |
|---|---|
| ODE | ordinary differential equation |
| PDE | partial differential equation |
| PINN | physics-informed neural network |
| NO | neural operator |
| PINO | physics-informed neural operator |
| FNO | Fourier neural operator |
| LNO | Laplace neural operator |

## 1 Introduction

Control problems are fundamental in engineering and applied sciences, whether you need to start and shut down power plants efficiently [1], fly aircraft along minimum-fuel trajectories [2], or optimize financial portfolios in continuous time [3]. In each case, differential equations driven by control inputs govern the system dynamics.

Classical theory - built on the Pontryagin Maximum Principle [4] and Hamilton-Jacobi-Bellman equations [5] - provides the necessary conditions for optimality. However, real-world systems introduce nonlinearities, constraints on states and controls, making closed-form solutions extremely challenging or impossible. To handle these complexities, practitioners commonly rely on numerical methods. Indirect methods, such as shooting methods [6], transform the boundary value problems into root-finding problems, but are sensitive to initial guesses. Direct transcription and collocation [7] discretize trajectories, converting the problem into large-scale nonlinear programs that, despite their effectiveness, require substantial computational effort and careful tuning as problem dimensionality grows.

Recent research studies how data-driven surrogate models can solve computational limitations and scalability issues of traditional numerical methods. Physics-informed neural networks (PINNs) and neural operators have emerged as particularly promising alternatives. PINNs enforce differential equations and boundary or initial conditions as soft constraints during training, enabling them to solve ordinary and partial differential equations effectively [8]. This led to further development of neural operators, designed explicitly to learn mappings between function spaces and effectively generalize solutions across parameterized PDE families.

Deep Operator Network (DeepONet) was one of the first neural operator architectures [9], which leveraged the universal approximation theorem to learn operators between infinite-dimensional function spaces. In particular, DeepONet successfully recovered solution maps for PDEs such as reaction-diffusion and Burgers equations. Subsequently, Fourier Neural Operator (FNO) [10] was introduced as an iterative architecture that represents the integral kernel as a convolution in the frequency domain. More recently, the Laplace Neural Operator (LNO) [11] was introduced as a possible alternative to FNO, which represents the integral kernel as a convolution in the Laplace domain.

The success in learning solution operators for PDEs showed that neural operators present significant potential as surrogate models in optimal control frameworks. Hwang et al. [12] demonstrated that autoencoder-based operators could approximate solution operators for Poisson source and Stokes boundary control problems. This success motivates exploration into whether alternative neural operator architectures can similarly serve as reliable surrogates in optimal control. More recently, Lundqvist and Oliveira [13] managed to use a DeepONet model trained solely in a physics-informed manner to solve a control problem constrained by diffusion-reaction PDE.

This work investigates explicitly several open questions: (i) how DeepONet, FNO, and LNO can be adapted for optimal control problems with various boundary or initial conditions; (ii) how differences in operator designs impact convergence, solution accuracy, and overall control performance; (iii) inherent limitations of neural operators as control surrogates; and (iv) whether neural operators can be effectively trained using only physics residuals, eliminating dependence on simulation-based datasets.

We adopt a physics-residual-based training approach, where model predictions are supervised exclusively by governing differential equations and boundary or initial conditions. This approach eliminates the need for costly simulations and significantly reduces memory usage and computational demands. We generate trajectories for some input controls to track signs of overfitting. We test if the approach suggested for DeepONet [13] is also generalizable for an alternative architecture.

This thesis systematically compares DeepONet, FNO, and LNO as surrogate models, providing explicit guidelines for architecture selection, hyperparameter tuning, and integration with solvers for practical PINO-based control applications. To thoroughly evaluate the learning behavior of each architecture, we test multiple classes of candidate control inputs (polynomial, linear, sine, etc), identifying the most effective inputs for accurate control-state mappings. We train all models using the Adam optimizer [14]. The architectures are tested on five benchmark optimal control problems with known analytical solutions, and their performance is evaluated using a relative $\mathcal{L}_2$ error between predicted and accurate trajectories. FNO is further tested on one-dimensional heat, diffusion-reaction, and Burgers constrained control problems.

The remainder of this thesis is organized as follows. Chapter 2 reviews the theory behind optimal control, PINOs, and existing neural operators. Chapter 3 describes our training framework of the neural operators and their integration into the optimal control problem. Chapter 4 details the benchmark control problems, evaluation metrics, and experimental setups. Chapter 5 presents results including convergence studies, error analyses, runtime comparisons for each architecture, sensitivity analysis, and solutions found for the control problems. Chapter 6 discusses practical guidelines, identified limitations, and future research directions. Chapter 7 summarizes contributions and provides our insights.

# 2 Background

In this chapter, we present the theoretical foundations necessary to understand the methods that we develop in later chapters. Section 2.1.1 defines the classical optimal control problem and reviews commonly used numerical solution methods. Section 2.2 introduces operator learning as a surrogate modeling approach to solve parametric differential equations. Section 2.2.3 describes physics-informed neural networks (PINNs) and explains how physics-based loss functions can be incorporated into operator training. Finally, Section 2.3 discusses how we can integrate neural operators into the optimal control problem, reviews relevant prior work, and motivates the unified PINO-control framework proposed in this thesis.

## 2.1 Optimal Control

### 2.1.1 Fundamental concepts

In optimal control, we seek a time–varying input $u(t)$ that drives the system from its initial state to meet boundary conditions while minimizing a given cost.

**Problem statement.** Let $x(t) \in \mathbb{R}^n$ denote the system state and $u(t) \in \mathbb{R}^m$ the control, over $t \in [0, T]$. Given

$$\dot{x}(t) = d\big(x(t),\, u(t)\big), \qquad x(0) = x_0, \tag{1}$$

We minimize the cost

$$J(u) = G\big(x(T)\big) + \int_0^T F\big(x(t),\, u(t),\, t\big)\,\mathrm{d}t, \tag{2}$$

where $d : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ defines the system *dynamics*, $F$ is the *running cost*, i.e., the instantaneous cost rate accumulated over time, $G$ is the *terminal cost*, evaluated at the final state, and, boundary conditions may impose constraints on the initial state $x(0) = x_0$ and on the terminal state $x(T)$ or control $u(T)$.

**Flight–control example.** Consider a glider whose state

$$\mathbf{x}(t) = \begin{bmatrix} h(t) \\ v(t) \\ \gamma(t) \\ x(t) \end{bmatrix},$$

collects altitude $h$, airspeed $v$, flight–path angle $\gamma$, and downrange coordinate $x$. The control $u(t) = C_L(t)$ is the lift coefficient. The dynamics

$$\dot{h} = v \sin\gamma, \quad \dot{v} = \frac{1}{m}(L(u) - D(v,h)), \quad \dot{\gamma} = \frac{1}{mv}(L(u)\cos\gamma - mg), \quad \dot{x} = v\cos\gamma,$$

incorporate lift $L$, drag $D$, and gravity $g$. Initial conditions are

$$x(0) = 0, \quad h(0) = h_0, \quad v(0) = v_0, \quad \gamma(0) = \gamma_0.$$

The objective is to maximize the final range $x(T)$ with free terminal time $T$ and penalizing control effort:

$$J(C_L) = -x(T) + \int_0^T (\alpha C_L(t)^2 + \beta D(v(t), h(t))r)dt.$$

We may impose terminal requirements such as

$$h(T) = h_f, \quad v(T) \geq v_f, \quad \gamma(T) \text{ free}, \quad T \text{ free}.$$

Figure 1 illustrates this problem setting.

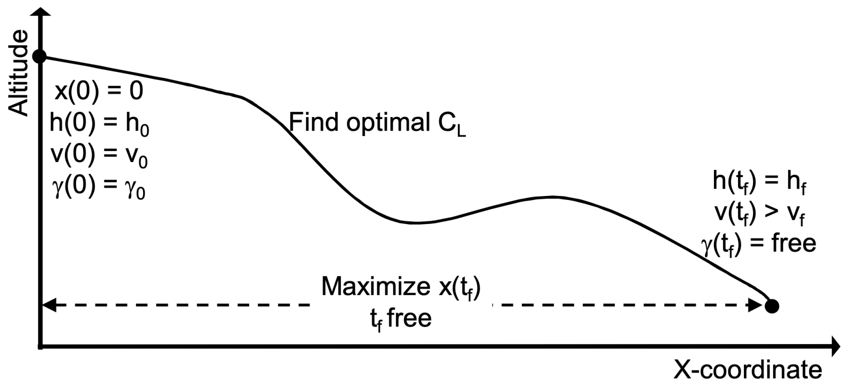


**Figure 1:** Flight–control example: initial conditions, control profile $C_L(t)$, and terminal objectives/constraints.

### 2.1.2 Existing methods

In practice, optimal-control problems, such as the flight-control example, are typically solved using numerical methods. These methods can be split into two main categories:

**Indirect methods** utilize Pontryagin's Minimum Principle (PMP) to derive optimality conditions [2], transforming the problem into a two-point boundary-value problem (TPBVP). PMP introduces costate variables $\lambda(t)$ and defines the Hamiltonian as:

$$H(x, u, \lambda, t) = F(x, u, t) + \lambda^\top d(x, u). \tag{3}$$

Optimal trajectories must satisfy:

$$\dot{x}(t) = \frac{\partial H}{\partial \lambda}, \quad \dot{\lambda}(t) = -\frac{\partial H}{\partial x}, \quad 0 = \frac{\partial H}{\partial u}, \quad \lambda(T) = \frac{\partial G}{\partial x(T)}. \tag{4}$$

Indirect methods typically employ shooting or multiple-shooting techniques, iteratively simulating state and costate trajectories to meet boundary conditions. However, they suffer from high sensitivity to initial guesses of costates, potentially causing divergence, and numerical difficulties due to nonlinearities, stiffness, and constraints on states and controls.

**Direct methods** [6, 15] discretize the optimal-control problem, converting it into a nonlinear programming (NLP) problem. The state variables $x$ and control inputs $u$ are defined at discrete time points $t_0, t_1, \ldots, t_N$. The objective function then takes the form:

$$\min_u G(x_N) + \sum_{i=0}^{N-1} F(x_i, u_i, t_i)\Delta t_i. \tag{5}$$

Additionally, discrete-time dynamics constraints are imposed:

$$x_{i+1} = x_i + \Phi\Big(x_i, u_i\Big)\Delta t_i, \tag{6}$$

where $\Phi$ represents the numerical integration or collocation approximation. Direct methods implicitly simulate trajectories through iterative evaluations of these discretized equations by NLP solvers (such as IPOPT [16] or SNOPT [17]), efficiently tackling large-scale problems. Nevertheless, direct methods face challenges, including computational and memory demands. Solution accuracy depends on the discretization grid density. It is also challenging to address free terminal-time conditions or intricate path constraints.

In summary, both indirect and direct methods fundamentally rely on numerical simulations. Practitioners select method based on specific computational requirements, desired solution accuracy, and the complexity of the problem.

## 2.2 Operator Learning

While classical numerical methods are effective, they often require repeated simulations and become computationally expensive - especially in iterative or real-time control settings. This motivates the development of data-driven surrogates that can emulate solver behavior without explicitly solving the underlying differential equations at each step.

### 2.2.1 Neural Operators

Neural operators can be a surrogate model that maps inputs - such as control signals, source terms, or boundary conditions - to the solution of *parametric differential equations*. Rather than solving each new instance from scratch, a neural operator learns the *solution operator* that maps function-valued inputs $u(t)$ to state trajectories $x(t)$ [18].

Formally, let $\mathcal{U}$ be a space of input functions (e.g., controls or forcing terms), and consider the parameterized initial-value problem

$$\dot{x}(t) = d\big(x(t), u(t)\big), \qquad x(0) = x_0, \tag{7}$$

where $x(t) \in \mathbb{R}^n$, $u \in \mathcal{U}$, and $d$ defines the system dynamics. The goal is to learn the mapping

$$\mathcal{D}(u)(t) = \hat{x}(t), \tag{8}$$

which assigns each control input $u$ to a predicted continuous trajectory $\hat{x}(t)$ defined on the targeted range $[0, T]$. This operator $\mathcal{D}$ can be approximated by a neural operator $\mathcal{D}_\theta$ with learnable parameters $\theta$.

A *neural operator* is a deep learning framework designed to approximate mappings between infinite-dimensional function spaces [18]. Unlike classical neural networks that map finite-dimensional vectors to outputs, neural operators generalize across functional inputs, enabling fast and differentiable predictions for a broad class of parametric equations without retraining.

### 2.2.2 Existing Architectures

Several neural operator architectures have been proposed, each introducing different inductive biases and trade-offs in approximation accuracy, training efficiency, and scalability.

**DeepONet** [9] is based on the universal approximation theorem for operators. It decomposes the learning task into two networks: a *branch* network that encodes the input function and a *trunk* network that encodes the evaluation coordinate locations. DeepONet is simple to implement and theoretically expressive, but may struggle with high-dimensional dynamics or long-range temporal dependencies.

**Fourier Neural Operator (FNO)** [10] learns operators in the frequency domain using spectral convolutions. It begins by projecting *the input* to *a higher-dimensional* latent space, then applies the fast Fourier transform (FFT), multiplies the frequency coefficients element-wise by learned complex weights, and finally transforms the signal back to the spatial domain. This architecture leverages translation invariance and the regular structure of uniform grids, yielding high efficiency on periodic or smoothly varying problems. Because FNO hinges on the FFT, however, it is less effective on irregular meshes or strongly non-periodic domains.

**Laplace Neural Operator (LNO)** [11] learns operator mappings in the Laplace domain using a pole-residue decomposition. It transforms inputs via the Laplace transform, applies learned transfer functions through a residue-pole parametrization, and inverts back to the domain of the output function. This approach is well-suited for systems where past inputs have lasting effects, such as those with decaying transients. In principle, working in the Laplace domain captures long-term dependencies and tends to produce stable solutions.

Beyond these, many other architectures have emerged to address irregular meshes, geometric structures, long-range dependencies, multiscale resolution, and implicit operator representations. Based on their designs, these methods can be grouped into operator families (see Figure 2). We briefly mention a few of the most interesting ones. One architecture - WaveletNO, similar to FNO and LNO, uses a wavelet transform to represent the kernel integral [19]. The Geometry-Informed Neural Operator (GINO)[20] leverages a Graph Neural Operator (GNO) [21] to map an irregular grid to a regular one, enabling FNO to be applied efficiently . The General Neural Operator Transformer (GNOT) employs a normalized attention (transformer) layer to handle multiple input functions and irregular meshes [22].

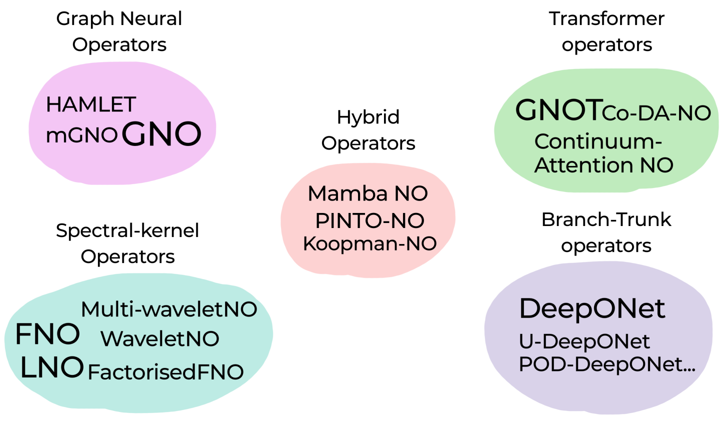


**Figure 2:** Each colored bubble groups representative models that share the same kernel-parameterisation strategy: *branch–trunk* (e.g. DeepONet variants), *spectral-kernel*, *transformer* (attention-based adaptive kernels), *graph*, and *hybrid/physics-informed*. New variants often blend ideas across families.

### 2.2.3 Physics-Informed Neural Networks (PINNs)

Physics-informed neural networks (PINNs) embed known physical laws - typically differential equations - directly into the loss function of a neural network [8]. Instead of relying on simulation data, a PINN minimizes the residual of the governing equations alongside any available data term.

A standard PINN approximates a solution with a feed-forward network whose parameters $\theta$ are trained by minimizing

$$\mathcal{L}_{\mathrm{PINN}}(\theta) = \mathcal{L}_{\mathrm{data}}(\theta) + \mu_{\mathrm{phys}}\,\mathcal{L}_{\mathrm{phys}}(\theta), \tag{9}$$

where $\mathcal{L}_{\mathrm{phys}}$ measures violation of the PDE/ODE, typically via automatic differentiation, and $\mathcal{L}_{data}$ is any metric that measures the difference between simulation and prediction. This in turn allows to encode physical dynamics without labeled data at every space–time point, make the model generalizable to unseen inputs while still obeying laws of the system, and enforce hard constraints such as initial or boundary conditions.

Combining PINNs with neural operators (NOs) yields the best of both worlds: NOs generalize over functional inputs, while residual penalties keep every predicted trajectory consistent with the system's governing equations. Given a neural operator $\mathcal{D}_\theta$ mapping a control $u \in \mathcal{U}$ to a predicted trajectory $\hat{x} = \mathcal{D}_\theta(u)$, we can train it by minimizing

$$\mathcal{L}_{\text{train}}(\theta) = \mu_{\text{data}}\, \mathcal{L}_{\text{data}} + \mu_{\text{phys}} \left\| \mathcal{R}(\hat{x}, u) \right\|^2, \tag{10}$$

where $\mathcal{R}$ denotes the *physics residual.* Given a differential equation of the form

$$\mathcal{F}(x(t), \dot{x}(t), u(t)) = 0,$$

and a prediction $\hat{x}(t) = \mathcal{D}_\theta(u)$ produced by a learned operator, the *residual* $\mathcal{R}(\hat{x}, u)(t)$ is defined as the violation of the differential equation by the predicted state:

$$\mathcal{R}(\hat{x}, u)(t) := \mathcal{F}(\hat{x}(t), \tfrac{d\hat{x}}{dt}(t), u(t)). \tag{11}$$

A zero residual implies exact satisfaction of the governing equation. The derivative $\frac{d\hat{x}}{dt}(t)$ can be computed using automatic differentiation [23] or numerical differentiation on a discrete grid.

**Toy residual example.** For the scalar ODE

$$\dot{x}(t) + x(t)^2 - u(t) = 0$$

the residual for the prediction $\hat{x}(t) = \mathcal{D}_\theta(u)$ is

$$\mathcal{R}(\hat{x}, u) = \frac{d\hat{x}}{dt}(t) + \hat{x}(t)^2 - u(t).$$

### 2.2.4 Training paradigms.

Recent works have evaluated different training paradigms combining supervision and physics priors.

**Physics-informed training (PINO)**: Li *et al.* [24] introduced the PINO framework, which trains neural operators purely using physics residuals. Their results showed that PINO can achieve high generalization ability to unseen data, outperforming supervised baselines when data is scarce.

**Self-supervised pretraining**: Madhavan *et al.* [25] proposed a transformer-based self-supervised learning approach for PDE solvers. The model is pretrained on large unlabeled datasets using masked or contrastive objectives, enabling it to learn a family of solution operators across varied PDE parameters. After fine-tuning with a small number of labeled trajectories, the model achieves faster convergence and significantly better generalization.

**Latent-space pretraining**: Wang & Wang [26] introduced a method where neural operators first learn to encode solution trajectories into a latent manifold using autoencoder-style pretraining. This latent-code pretraining on unlabeled data reduces complexity, and then is fine-tuned on downstream prediction tasks.

**Lower-dimensional PDE-slice pretraining**: Hemmasian & Farimani [27] suggest training neural operators on low-dimensional versions of the PDE (e.g., 1D/2D) before adapting to full-dimensional problems. This speeds up training and reduces overfitting when supervised data is limited.

## 2.3 Neural Operators in Optimal Control

Classical indirect and direct methods for optimal control become computationally expensive as problem dimensionality increases. This is critical in systems that demand high precision and solution accuracy. Neural networks have proven themselves to be universal function approximators, prompting many proposals for using them as surrogates in optimal-control settings.

Physics-informed neural networks (PINNs) have shown the ability to solve forward and inverse problems governed by differential equations [8, 28]. Building on this, Mowlavi and Nabi applied PINNs to control problems constrained by known differential equations [29]. Chen *et al.* (2019) trained a convex recurrent neural network that learns the system's temporal behavior and then finds the optimal control by solving a convex model-predictive-control problem [30]. Yin *et al.* (2023) proposed the Adjoint-Oriented Neural Network, a surrogate that learns the control, adjoint, and state to obtain the optimal solution via a single forward pass [31]. Schiassi *et al.* (2024) introduced Pontryagin Neural Networks for optimal-control problems with integral quadratic cost functions [32].

Hwang *et al.* (2021) [12] proposed a two-phase pipeline for operator-learning control. They employed a custom autoencoder specifically designed to control problems governed by partial differential equations. In phase 1, they trained an encoder for the input control $u$ and two decoders: one that maps to the system state and another that reconstructs the input control $\hat{u}_\theta$. In phase 2, they froze $\theta$ and treated the input control as a trainable variable, minimizing a combined loss consisting of the control objective and the relative trajectory error. Tested in both supervised and data-free modes, the method successfully approximated solutions to Poisson-, Stokes-, wave-, and Burgers-constrained problems. Although fast and as accurate as a numerical solver, it is trained for only a single optimal-control task.

Motivated by the limitations of autoencoder-based two-phase approach, Lundqvist and Oliveira (2025) [13] proposed a simpler algorithm. Given a differentiable neural operator, they use it directly in the optimization loop, treating the control $u$ as a trainable parameter that minimizes the objective. The key requirement is to include the physics residual - weighted appropriately - in the loss during control optimization to maintain feasibility. Their study focused only on DeepONet. This thesis is going to bridge the gap and extend their approach to other neural operator architectures and test on a wider set of benchmark problems.

More recently, Feng *et al.* (2025) introduced the Neural Adaptive Spectral Method (NASM), which learns the operator mapping from a problem instance to its solution and generalizes across related control tasks [33]. The authors criticize two-phase methods as computationally expensive and inconsistent: the input signal may fall outside of the trained distribution, causing convergence failure. NASM instead accepts the cost function, system dynamics, and initial conditions as input and directly outputs the candidate optimal control function. We found this approach refreshing, but it is still limited to a family of control problems.

We introduce a PINO-control framework that aims to support any neural operator architecture and apply it to any control problem. We used this framework to test DeepONet, FNO, and LNO on a set of various benchmark problems to address the weaknesses and strengths of the architectures. We test their robustness and discuss possible applicability to solve control problems. We show that Neural operators can be applied to more or less real-world problems in control problems (problems that are constrained by differential equations). By further showing the potential of the approach suggested by [13].

# 3 Methodological approach

## 3.1 General idea

Suppose we have a neural operator $\mathcal{D}_\theta$ trained to minimize the physics-residual

$$\mathcal{R}(x,u) = \dot{x} - d(x,u) \tag{12}$$

and enforce any boundary or initial conditions. In other words, $\mathcal{D}_\theta$ acts as a solution operator for the differential operator $d$, returning an approximate trajectory $\hat{x}$ for any input control function $u$.

To embed $\mathcal{D}_\theta$ into an optimal-control solver, we follow the direct transcription paradigm. We discretize the time interval $[0, m]$ into $N$ equal steps,

$$t_i = i\,\Delta, \quad i = 0, \dots, N, \quad \Delta = \frac{m}{N},$$

And recall that a standard direct method enforces (6):

$$x_{i+1} = x_i + \Phi(x_i, u_i)\,\Delta t_i,$$

Where $\Phi$ denotes a numerical integrator or collocation operator. Replacing $\Phi$ with our neural operator gives the following discrete control formulation:

$$\min_{X,U} \quad G(x_N) + \sum_{i=0}^{N-1} F(x_i, u_i, t_i)\,\Delta t_i, \tag{13}$$

$$\text{s.t.} \quad \{x_i\}_{i=0}^{N} = \mathcal{D}_\theta\left(\{u_i\}_{i=0}^{N}\right), \tag{14}$$

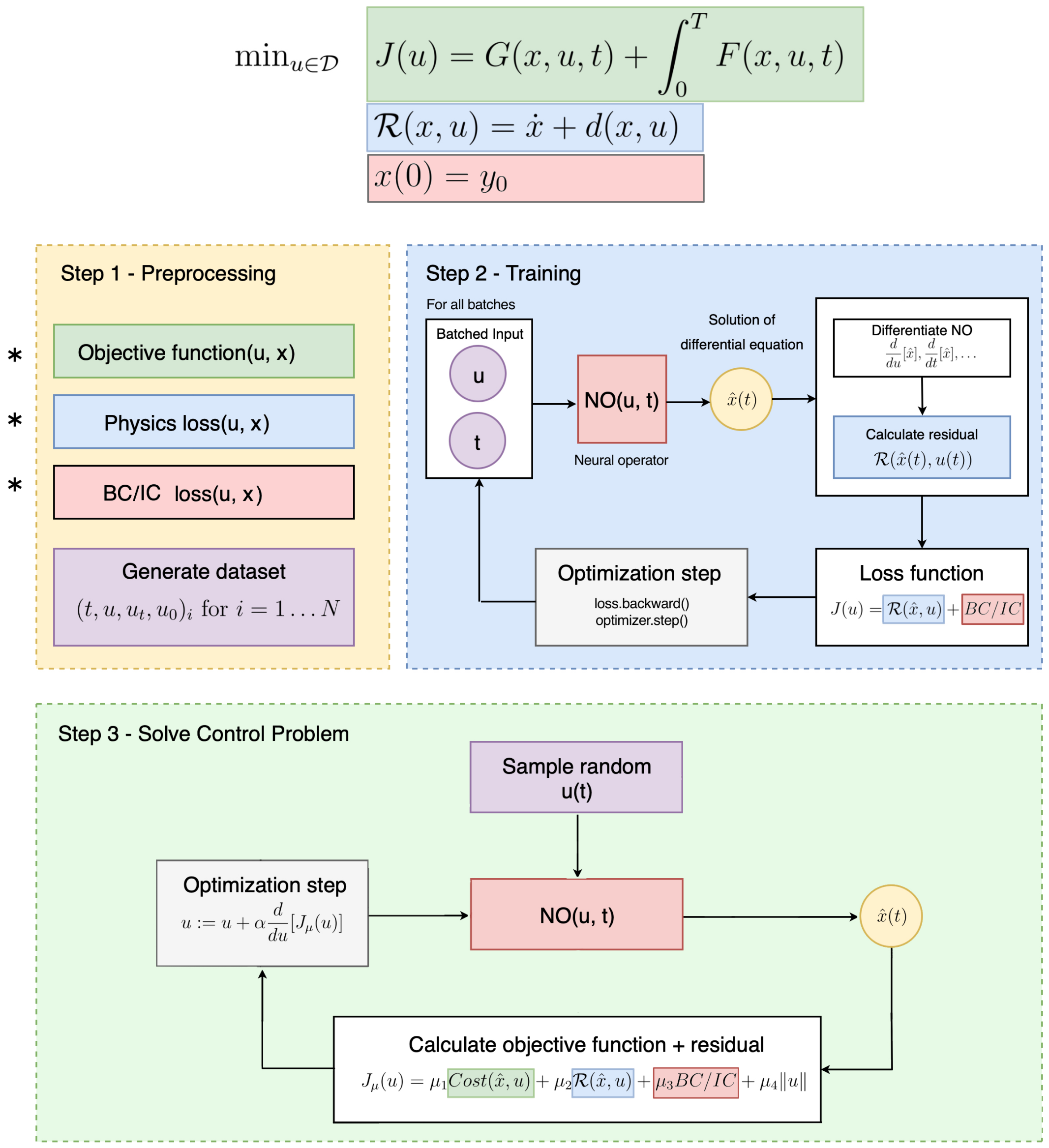
$$x_0 = y_0. \tag{15}$$

This formulation can have any additional boundary (BC), terminal (TC), or initial conditions (IC).

In practice, since $\mathcal{D}_\theta$ can be imperfect outside of its training distribution, it is advised to include a soft penalty on the physics residual. In the same fashion, we can include penalties on conditions posed on the trajectory $x$ inside the objective function. We also optionally add a term to discourage converging to overly noisy control functions. Given $\mu_i$ are the weights of the particular loss component, the control optimization problem can be formulated as follows:

$$\begin{aligned} \min_{U=\{u_i\}_{i=0}^{N}} \quad & \underbrace{\mu_1\Big[G(x_N) + \sum_{i=0}^{N-1} F(x_i,u_i,t_i)\,\Delta t\Big]}_{\substack{\text{original}\\ \text{cost}}} + \underbrace{\mu_2 \sum_{i=0}^{N-1} \|\mathcal{R}\|^2\,\Delta t}_{\text{physics penalty}} \\ & + \underbrace{\mu_3\|x_0 - y_0\|^2}_{\text{IC penalty}} + \underbrace{\mu_4 \sum_{i=0}^{N-1} \|u_{i+1} - u_i\|^2}_{\text{control smoothness}} \\ \text{s.t.} \quad & \{x_i\}_{i=0}^{N} = \mathcal{D}_\theta\Big(\{u_i\}_{i=0}^{N}\Big). \end{aligned} \tag{16}$$

The whole PINO-control framework is summarized in the Figure 3 as a 3-step process.



$\min_{u \in \mathcal{D}}$ $J(u) = G(x,u,t) + \int_0^T F(x,u,t)$
$\mathcal{R}(x,u) = \dot{x} + d(x,u)$
$x(0) = y_0$
Step 1 - Preprocessing
* Objective function(u, x)
* Physics loss(u, x)
* BC/IC loss(u, x)
Generate dataset
$(t, u, u_t, u_0)_i$ for $i = 1 \dots N$
Step 2 - Training
For all batches
Batched Input
u
t
NO(u, t)
Neural operator
Solution of differential equation
$\hat{x}(t)$
Differentiate NO
$\frac{d}{du}[\hat{x}], \frac{d}{dt}[\hat{x}], \dots$
Calculate residual
$\mathcal{R}(\hat{x}(t), u(t))$
Loss function
$J(u) = \mathcal{R}(\hat{x}, u) + BC/IC$
Optimization step
loss.backward()
optimizer.step()
Step 3 - Solve Control Problem
Sample random u(t)
NO(u, t)
$\hat{x}(t)$
Optimization step
$u := u + \alpha \frac{d}{du}[J_\mu(u)]$
Calculate objective function + residual
$J_\mu(u) = \mu_1 Cost(\hat{x}, u) + \mu_2 \mathcal{R}(\hat{x}, u) + \mu_3 BC/IC + \mu_4 \|u\|$

**Figure 3:** Overview of the PINO-control workflow. **(Step 1: Preprocessing)** Define the original cost $J(u)$, the physics residual $\mathcal{R}(x, u)$, and BC/IC losses; generate a training dataset of control–state tuples. **(Step 2: Training)** Batch $(u, t)$ through the neural operator to predict $\hat{x}(t)$, compute residuals and boundary/initial-condition penalties, assemble the composite loss, and update $\theta$ via backpropagation. **(Step 3: Control solve)** Initialize a candidate control $u(t)$, evaluate $\hat{x} = \mathcal{D}_\theta(u)$, form the control objective (original cost + physics/BC/IC penalties + control-smoothness term), and iteratively refine $u$ by gradient descent through the operator.

Step 3 is based on the gradient-based procedure introduced by Lundqvist and Oliveira [13].

**Algorithm 1** Gradient-based Optimization using a NO

```
Require: Pre-trained NO $\mathcal{G}_\theta$; discretized cost $\bar{J}(u)$
Ensure: Optimal control $u^*$
 1: Initialize $\{u_i\}_{i=0}^{I-1}$ (e.g. all zeros); set $t_i = i\,\Delta t$, so $u_i \equiv u(t_i)$
 2: while not converged do
 3:     $y_i \leftarrow \mathcal{G}_\theta\Big(\{u_k\}_{k=0}^{I-1}(t_i)\Big) \quad \forall i = 0, \dots, I-1$          ▷ Forward pass
 4:     $\dot{y}_i \leftarrow \frac{d}{dt}\,\mathcal{G}_\theta\Big(\{u_k\}_{k=0}^{I-1}(t_i)\Big) \quad \forall i = 0, \dots, I-1$          ▷ Time derivative via
    AutoDiff or Finite difference
 5:     $R_i \leftarrow \Big\|\dot{y}_i - f(y_i, u_i, t_i)\Big\|_2^2 \quad \forall i = 0, \dots, I-1$          ▷ Residuals
 6:     $\bar{J}_\mu\Big(\{u_i\}\Big) \leftarrow \bar{J}\Big(\{u_i\}\Big) + \frac{\mu}{I}\sum_{i=0}^{I-1} R_i$          ▷ Penalized cost
 7:     Compute $\frac{d\bar{J}_\mu}{du_i} \quad \forall i = 0, \dots, I-1$          ▷ Gradients via AutoDiff
 8:     $u_i \leftarrow \Phi\Big(u_i, \frac{d\bar{J}_\mu}{du_i}\Big) \quad \forall i = 0, \dots, I-1$          ▷ Update step
 9: end while
10: return $u^* \leftarrow \{u_i\}_{i=0}^{I-1}$
```

Algorithm 1 initializes a discretized control trajectory $\{u_i\}$, computes state predictions and their time-derivatives via automatic differentiation, assembles the residual penalty, and updates the controls using the map $\Phi$ until convergence [13].

In the next sections, we analyze the remaining stages of the PINO-Control framework and outline the neural-operator architectures we have selected to evaluate within it.

## 3.2 Preprocessing

Preprocessing consists of two separate parts: data generation and function definition. Data generation process is covered in full detail in Section 4.3.

First, we discretize the control problem on a uniform grid of $N + 1$ time points,

$$u = \{u_i\}_{i=0}^{N}, \quad x = \{x_i\}_{i=0}^{N}, \quad t_i = i\,\Delta t.$$

We then implement four core routines (see Figure 3, routines are denoted by $*$), each taking the discrete trajectories $u$, $x$, and any additional hyperparameters (denoted by `*args`):

- * `physics_loss`(u, x, *args) - computes $\|\mathcal{R}\|_{\mathcal{L}_2}^2$, which is the violation of physics residual, i.e.. In a perfect case, it should be $\approx 0$

* `boundary_loss`(u, x, *args) - Penalizes violation of terminal or other boundary conditions

* `initial_loss`(u, x, *args) - penalizes deviation of $x_0$ from a prescribed initial state

* `objective_function`(u, x, *args) - evaluates $G(x_N)+\sum_{i=0}^{N-1} F(x_i, u_i, t_i)\,\Delta t$.

Let us first consider how we treat the objective function. Any integral component - such as the running cost $\int_0^T F(x(t), u(t), t)dt$ - is discretized using the trapezoidal rule, which offers a good trade-off between accuracy and simplicity. $G(x_N)$, the terminal cost, is typically evaluated directly at the final state and added to the integral.

Boundary and initial loss functions can be written straightforwardly, as defined in (16). The physics loss function is generally harder to define, since depending on the residual $\mathcal{R}$, we may need to calculate derivatives of the output function $x$ with respect to any input variable.

Derivatives of $x$ can be obtained in two ways:

1. **Automatic differentiation:**
   If the neural operator is defined such that it accepts time $t$ as an explicit input, we can compute exact derivatives using automatic differentiation provided by modern frameworks (e.g. PyTorch, TensorFlow). These frameworks construct a computation graph and apply the chain rule to evaluate derivatives efficiently and accurately [23]:
$$\dot{x}(t_i) = \frac{\partial}{\partial t}\mathcal{D}_\theta\big(u(\cdot),\, t_i\big), \qquad \frac{\partial x(t_i)}{\partial u} = \frac{\partial}{\partial u}\mathcal{D}_\theta\big(u(\cdot),\, t_i\big). \tag{17}$$

2. **Numerical differentiation:**
   If $x$ is available only on a fixed grid $\{t_i\}$, we can employ finite-difference approximations:
$$\dot{x}(t_i) \approx \frac{x(t_{i+1}) - x(t_i)}{\Delta t}, \qquad \frac{\partial x(t_i)}{\partial u(t_i)} \approx \frac{x\big(u_i+\varepsilon, t_i\big) - x\big(u_i-\varepsilon, t_i\big)}{2\,\varepsilon}, \tag{18}$$
   where $\varepsilon \ll 1$ is a small perturbation.

If there are derivatives of $u$ present in the physics residual, they need to be generated along with you during the dataset generation step.

## 3.3 Training of the Neural Operator

Once the dataset is prepared and problem-specific loss components are defined, we train the neural operator using a standard learning loop adapted for physics-informed objectives.

The dataset consists of control input functions from various families (e.g., sine, polynomial, linear). We assume true trajectories are available only for a limited number of control functions - or not at all. We have datasets generated using different seeds. 20% of the training control functions have a trajectory, and the validation set has all of the trajectories. In this way, we can judge how well the trajectories from both are learned and see if there are signs of overfitting.

**Training Procedure.**

1. **Training phase.**
   Set the model to training mode (`model.train()`). For each batch $(u, t, \dots)$ in the training loader:

   (a) Perform a forward pass: $\hat{x} = \texttt{model}(u, t)$.

   (b) Evaluate the physics-informed loss:

   $$\ell = \ell_{\text{phys}}(u, \hat{x}, t) + \ell_{\text{init}}(u, \hat{x}, t) + \ell_{\text{bnd}}(u, \hat{x}, t)$$

   This follows the PINO framework that integrates PDE solvers via residual losses to reduce overfitting [24].

   (c) Backpropagate/update: `loss.backward(); optimizer.step()`.

   (d) Track average training loss per epoch.

2. **Validation phase.**
   Switch to evaluation mode (`model.eval()`). For each batch in the validation set:

   (a) Compare predicted trajectory with reference $x^{\text{true}}(t)$ using relative $\mathcal{L}_2$-error loss.

   $$\mathcal{L}_{data} = \frac{\|\hat{x} - x^{\text{true}}\|_{\mathcal{L}_2}}{\|x^{\text{true}}\|_{\mathcal{L}_2}} \tag{19}$$

   (b) Log metrics and optionally visualize the worst predictions on the test set.

3. **Checkpointing.**
   Update learning rate (scheduler) and save model/optimizer state.

If the model fails to learn the physics dynamics, there are alternative ways to estimate the gradients, namely spectral differentiation in Fourier space (accurate only if the signals are periodic), Savitzky–Golay differentiator [34], etc.

**Joint (Semi-supervised) Training.** For semi-supervised training, we can combine physics-informed loss with supervised data loss:

$$\ell_{\text{total}} = \lambda_{\text{phys}} \ell_{\text{phys}} + \lambda_{\text{data}} \ell_{\text{data}}$$

To prevent overfitting to limited supervised samples, we can employ *adaptive residual weighting* - which automatically balances different loss magnitudes - shown to enhance convergence and reduce overfitting in PINNs and NO frameworks [35].

**Curriculum: Pretraining + Finetuning.** Following recent practices in PDE operator learning, we can:

- *Pre-train* with physics residuals only (unsupervised) [25].
- *Finetune* on the few available trajectories, *continuing to include physics loss* to preserve generalization to yield better accuracy [25, 26].

The built framework allows employing both options, and we will apply them only in case the neural operator architecture is not capable to learn solely based on physics residuals.

Our objective is to verify that Algorithm 1 can be used not only with DeepONet exclusively. For this, we select benchmark control problems with known optimal solutions and repeat the experiments for each studied neural operator architecture on the same generated dataset.

## 3.4 Selected Neural Operator Architectures

This section presents the three neural operator architectures that we benchmark in this thesis: Deep Operator Network (DeepONet), Fourier Neural Operator (FNO), and Laplace Neural Operator (LNO). We describe their core mechanisms and explain their suitability for physics-informed training and control optimization. A more technical introduction to these architectures is available in Appendix A.

### 3.4.1 Deep Operator Network (DeepONet)

DeepONet [9] learns mappings between function spaces using a dual-branch neural network architecture. A *branch network* encodes input function values sampled at fixed sensor locations, while a *trunk network* encodes arbitrary query points (e.g., time or space). Their inner product yields the predicted output:

$$G(f)(x) \approx \sum_{k=1}^{p} b_k(f)\, t_k(x) + b_0,$$

Where $\mathbf{b}(u)$ and $\mathbf{t}(t)$ are the outputs of the branch and trunk networks, respectively.

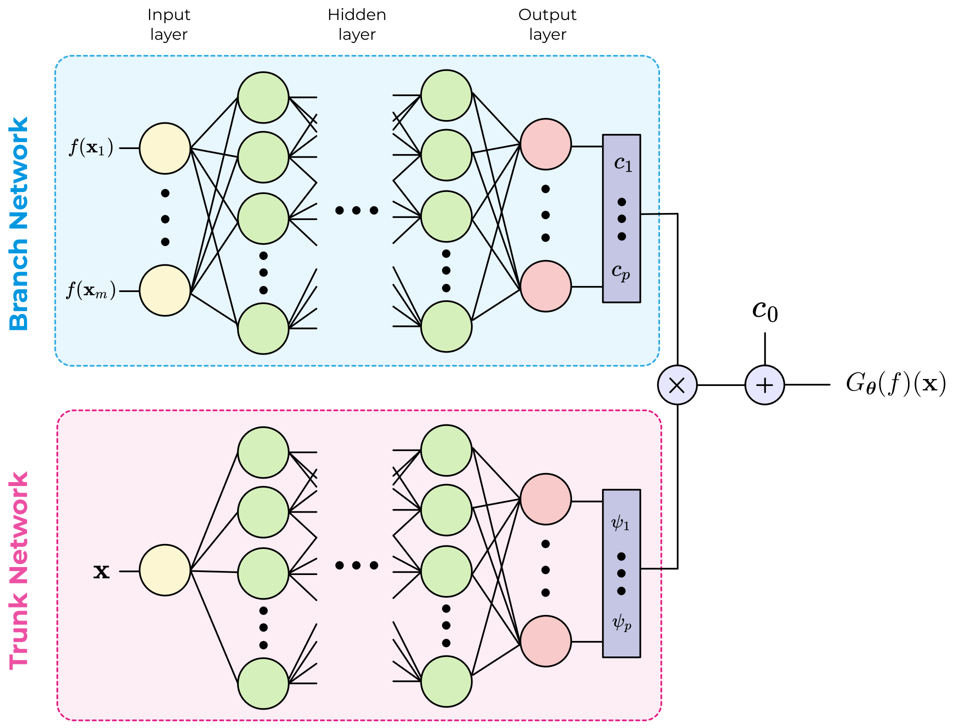


**Figure 4:** Illustration of the unstacked DeepONet architecture (adapted from [36]). Branch inputs: the function $f$ is sampled at fixed sensor locations $x_1, \ldots, x_m$. Trunk input: a query location $x \in \mathbb{R}^d$. The branch network encodes the sensor values into $\mathbf{b}$, the trunk network encodes $x$ into $\mathbf{t}$, and their inner product (plus bias) yields $G_\theta(f)(x)$.

Although BranchNet and TrunkNet can, in principle, adopt any architecture (e.g., CNNs, RNNs, graph neural networks), the original DeepONet authors used simple feed-forward neural networks (FNNs) for both (see Figure 4). This choice provides a good trade-off between expressive power and implementation efficiency. In our study, we use the original unstacked variant of DeepONet (see Appendix A for details).

Originally, DeepONet was trained in a purely data-driven manner by minimizing the mean-squared error loss. This approach required large datasets of paired input–output functions, which can be expensive to generate. To reduce reliance on simulation data, it was shown that DeepONet can incorporate a physics-informed loss by leveraging automatic differentiation through both BranchNet and TrunkNet [37]. This concise formulation makes DeepONet a powerful surrogate for solution operators in high-dimensional, parameterized PDEs. The process is straightforward since Deep-O-Net can produce values on the queried points, and time can be learned as a separate variable.

### 3.4.2 Fourier Neural Operator (FNO)

The *Fourier Neural Operator (FNO)* [10] extends the general neural operator iterative framework [21] by making two simplifications about the integral kernel (see Appendix A for details). With such simplifications integral kernel can be treated as a convolution and leverage the Fast Fourier Transform (FFT). This enabled efficient learning of solution operators to PDEs through truncated spectral representations.

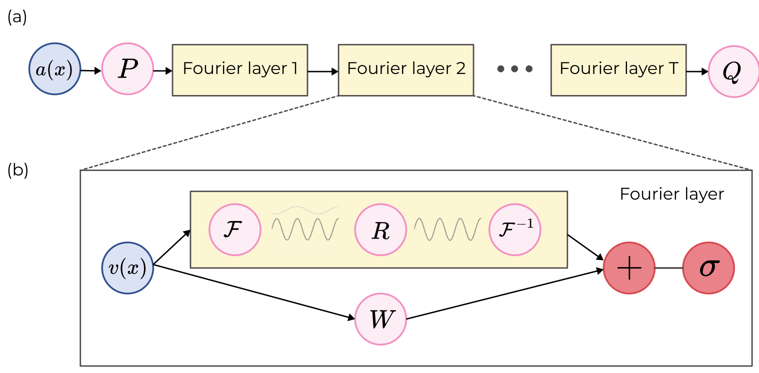


**Figure 5:** (a) Architecture of the Fourier Neural Operator (adapted from [10]): the input function $a(x)$ is lifted by a pointwise map $P$, passed through $T$ Fourier layers, and decoded by $Q$ to produce $u(x)$. (b) Detail of a single Fourier layer: the feature $v_t(x)$ is transformed by FFT $\mathcal{F}$, multiplied by learnable weights $\mathcal{R}_\phi$, transformed back via $\mathcal{F}^{-1}$, added to a local linear term $W v_t(x)$, and passed through a nonlinearity $\sigma$.

In the FNO, given an input field $a(x)$, we lift it to a latent representation $v_0(x) = P\,a(x)$ via a learnable projection $P$. Each Fourier layer (see Figure 5) then updates the latent field through the iterative rule:

$$v_{t+1}(x) = \sigma\Big(W\,v_t(x) + \mathcal{F}^{-1}\Big(\mathcal{R}_\phi \cdot \mathcal{F}\big(v_t\big)\Big)(x)\Big), \tag{20}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote the forward and inverse fast Fourier transforms; $W$ is a point-wise linear weight matrix; $\mathcal{R}_\phi$ is a learnable complex-valued filter that acts only on the lowest $k$ Fourier modes; $\sigma(\cdot)$ is a nonlinear activation function (e.g. GELU). Because the convolution is carried out in the spectral domain and restricted to a truncated set of modes, each layer costs only $\mathcal{O}\big(n \log n\big)$.

FNO performs well on smooth, high-dimensional, and periodic PDEs. However, it is restricted to fixed evaluation grids and does not support automatic differentiation in a straightforward way, because FNO is trained on a grid and uses FFT. Li *et al.* [24] proposed four schemes how we can treat gradients:

1. **Numerical differentiation.** Finite differences ($\mathcal{O}(n)$) or spectral differentiation ($\mathcal{O}(n \log n)$). Fast but less accurate on coarse or non-uniform meshes.

2. **Pointwise autodifferentiation.** Express $u(x)$ at arbitrary $x$ via its truncated Fourier series,
$$u(x) = Q\Big(\tfrac{1}{K} \sum_{|k|\leq K} \Big(R_k \, \mathcal{F}(v_{T-1})_k\Big) \, e^{\frac{i2\pi k\cdot x}{D}}\Big),$$
So derivatives are exact under autodiff. Accurate but memory- and compute-intensive. Notice that if $x$ is defined on a uniform grid, the derivative can be efficiently computed with the Fast Fourier transform.

3. **Function-wise (batched) differentiation.** Compute derivatives in Fourier space via the chain rule:
$$\frac{\partial u}{\partial x} = Q'\Big(v_T\Big)\mathcal{F}^{-1}\Big(i2\pi k \, \mathcal{F}(v_T)\Big).$$

4. **Fourier continuation.** Embed non-periodic domains into larger periodic ones (e.g. via zero-padding) to retain spectral accuracy, but evaluate PDE residuals only on the original domain.

In this thesis, we aim to create a general framework that uses any neural operator architecture to learn a parametric differential equation. Thus, this framework is going to be architecture invariant (meaning that the approach should work for any general operator).

Most existing physics-informed FNO implementations use numerical differentiation (e.g., central differences or spectral) and do not include implementation pointwise autodifferentiation or function-wise batched differentiation.

### 3.4.3 Laplace Neural Operator (LNO)

While the Fourier Neural Operator (FNO) [10] leverages the FFT to accelerate convolutional integral operators in the frequency domain, it suffers from three key limitations. First, the classical Fourier transform requires input functions to be integrable, and thus cannot represent signals such as $|x(t)|$ or model unstable dynamics. Second, Fourier-based layers capture only steady-state, periodic responses and omit any dependence on initial conditions. Third, non-periodic domains must be artificially periodized, often degrading accuracy near boundaries.

The Laplace transform addresses these issues by introducing an exponential decay factor. For a causal signal $x(t)$, its Laplace transform is
$$\mathcal{L}\{x\}(s) = \int_0^\infty x(t)\, e^{-st}\, \mathrm{d}t, \quad s = \sigma + i\omega,$$

Laplace transformation naturally encodes initial values via the differentiation properties:

$$\mathcal{L}\{\dot{x}(t)\} = s\,\mathcal{L}\{x(t)\} - x(0), \qquad \mathcal{L}\{\ddot{x}(t)\} = s^2\,\mathcal{L}\{x(t)\} - s\,x(0) - \dot{x}(0).$$

As a result, the Laplace domain simultaneously represents transient (through $\sigma$) and oscillatory (through $\omega$) behavior while embedding initial-condition information. Motivated by these advantages, the authors proposed replacing the Fourier layer with a *Laplace layer* (see Figure 6).

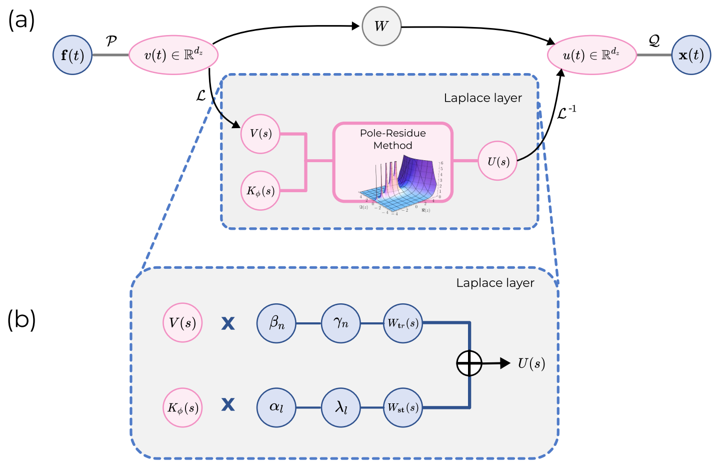


**Figure 6:** Schematic of the Laplace layer in LNO (adapted from [11]): the input function $a(t)$ is lifted via $\mathcal{P}$, convolved in the Laplace domain, and decoded by $Q$ to produce the output.

First, the input function $a(x)$ is lifted to a higher-dimensional feature field via a pointwise map $\mathcal{P}$, yielding $v(x) \in \mathbb{R}^{d_x}$, and then processed by

$$u(x) = \sigma\Big(\big(k(a;\phi) * v\big)(x)\Big) + W\,v(x),$$

where $\sigma$ is a nonlinear activation, $W$ a learnable linear transformation, and $k$ an integral kernel. The intermediate representation $u$ is finally decoded by $Q$ to produce the output $x(t)$. This overall pipeline mirrors that of FNO, but with a key modification in how the convolution is implemented. The Laplace-domain convolution is represented in pole-residue form:

$$K_\phi(s) = \sum_{n=1}^{N} \frac{\beta_n}{s - \mu_n},$$

with trainable poles $\mu_n$ and residues $\beta_n$. This formulation naturally models systems where responses decay over time, which is common in many real-world dynamical systems.

To date, research on a physics-informed LNO remains limited, since the architecture is relatively new. LNO also uses a fixed time grid during training, inheriting the same differentiability issues as FNO. In the original LNO study, the authors trained the model purely using a data-driven approach [11], without incorporating any physics-informed loss. Since the LNO architecture is similar to FNO, we employ the same numerical differentiation strategy.

# 4 Experimental Setup

## 4.1 Benchmark Problems

To evaluate the performance of neural operator architectures in optimal control settings, we select five benchmark problems with known analytical solutions.

### 4.1.1 Linear ODE

This classical linear-quadratic optimal control problem (21) is a standard benchmark in both theoretical and numerical optimal control literature, including Kirk [38], Bryson and Ho [2], and Vlassenbroeck and Van Dooren [39]. It serves as a canonical test case for evaluating convergence and accuracy in direct and indirect methods.

$$\begin{aligned} \min_u \quad & \frac{1}{2}\int_0^1 (x(t)^2 + u(t)^2)\,dt, && (21a)\\ \text{s.t.} \quad & \dot{x}(t) = -x(t) + u(t), && (21b)\\ & x(0) = 1. && (21c) \end{aligned}$$

The analytical solution is:

$$\begin{aligned} x^*(t) &= \frac{\sqrt{2}\cosh\left(\sqrt{2}(t-1)\right) - \sinh\left(\sqrt{2}(t-1)\right)}{\sqrt{2}\cosh(\sqrt{2}) + \sinh(\sqrt{2})}, && (22a)\\ u^*(t) &= \frac{\sinh\left(\sqrt{2}(t-1)\right)}{\sqrt{2}\cosh(\sqrt{2}) + \sinh(\sqrt{2})}. && (22b) \end{aligned}$$

This benchmark evaluates the capacity of neural operators to capture simple exponential decay patterns and stable feedback control laws. It also serves as a sanity check: if a model cannot learn this problem reliably, it is unlikely to perform well on more complex, nonlinear, or high-frequency tasks.

### 4.1.2 Oscillatory Forcing

The benchmark (23) features a periodic forcing term and appears in textbooks (e.g., Kirk [38]) as a test case for time-varying inputs.

$$\begin{aligned} \min_u \quad & \frac{1}{2}\int_0^1 \left(x(t)^2 + u(t)^2\right)\,dt, && (23a)\\ \text{s.t.} \quad & \dot{x}(t) = \cos(4\pi t) + u(t), && (23b)\\ & x(0) = 0, && (23c)\\ & x(1) = 0. && (23d) \end{aligned}$$

The analytical solution is:

$$x^*(t) = \frac{4\pi}{16\pi^2 + 1}\sin(4\pi t), \qquad u^*(t) = -\frac{1}{16\pi^2 + 1}\cos(4\pi t). \tag{24}$$

Derivation of analytical solution (24) is available in Appendix B.1.

This problem requires the model to represent periodic behaviour in both the state and control. In particular, spectral methods like FNO are expected to excel due to their efficient representation of periodic structures.

### 4.1.3 Polynomial Target Tracking

The quadratic tracking problem (25) appears in optimal control textbooks, such as Lewis et al. [40] and Kirk [38].

$$\begin{aligned} \min_{u} \quad & \int_0^1 \left((x(t) - t^2)^2 + u(t)^2\right) dt, && \text{(25a)} \\ \text{s.t.} \quad & \dot{x}(t) = u(t), && \text{(25b)} \\ & x(0) = 0. && \text{(25c)} \end{aligned}$$

The analytical solution is:

$$u^*(t) = 2t^2 - 2t + 1, \quad x^*(t) = \frac{2}{3}t^3 - t^2 + t. \tag{26}$$

Derivation of analytical solution (26) is available in Appendix B.2.

The problem is relatively simple, as it essentially requires learning an integral operator. However, its simplicity makes it easy for an expressive model to overfit the data rather than learn a generalizable mapping.

### 4.1.4 Nonlinear ODE

This nonlinear control problem (27) is widely used in benchmarking pseudospectral and direct collocation methods [41], as it combines bilinear and quadratic terms in both state and control.

$$\begin{aligned} \min_{u} \quad & -x(1), && \text{(27a)} \\ \text{s.t.} \quad & \dot{x}(t) = \frac{5}{2}\left(-x(t) + x(t)u(t) - u(t)^2\right), && \text{(27b)} \\ & x(0) = 1. && \text{(27c)} \end{aligned}$$

Its analytical solution is:

$$x^*(t) = \frac{4}{1 + 3\exp(5t/2)}, \quad u^*(t) = \frac{1}{2}x^*(t). \tag{28}$$

By introducing nonlinearity in the system dynamics and cost, this problem challenges neural operator models to represent more complex, nonlinear input–output maps, revealing their limitations and strengths in capturing non-affine behaviors.

### 4.1.5 Singular Arc Control Problem

The so-called “cliff problem“ (29) is the canonical singular arc benchmark, appearing in classical texts [2]. Its optimal control law is non-affine in $u$, requiring higher expressivity for surrogates.

$$\begin{aligned} \min_{u} \quad & \int_0^1 u(t)^2 \, dt, && \text{(29a)} \\ \text{s.t.} \quad & \dot{x}(t) = x(t)^2 + u(t), && \text{(29b)} \\ & x(0) = 1, \quad x(1) = 0. && \text{(29c)} \end{aligned}$$

The analytical solution is:

$$x^*(t) = \frac{1-t}{1+t}, \quad u^*(t) = -\frac{2}{(1+t)^2} - \left(\frac{1-t}{1+t}\right)^2. \tag{30}$$

Derivation of this analytical solution is available in Appendix B.3.

Singular arc problems are well known to be challenging for both classical and machine learning approaches, due to the presence of non-smooth and non-unique optimal controls. Successfully solving this problem indicates the ability of a model to handle non-standard optimality conditions and more intricate solution structures.

These benchmark problems are selected to provide a systematic and rigorous evaluation of neural operator architectures (DeepONet, FNO, LNO) within the PINO-control framework. Their diversity ensures a comprehensive test of model expressivity, ability to capture oscillatory dynamics, and robustness to nonlinear and singular behaviors. The availability of analytical solutions enables quantitative assessment of the found solutions.

The framework for the use of neural operators in optimal control - which we built - ensures that we can switch freely between all benchmark problems and that it can be easily extended.

## 4.2 Additional Benchmark PDEs

To further evaluate the performance and generalization capabilities of the best-performing neural operator architecture, we extend our benchmarking to more complex PDE-constrained optimal control problems. We specifically consider three PDEs - the Heat Equation, Diffusion-Reaction Equation, and Burgers' Equation - in one spatial dimension, arranged in order of increasing complexity. Our discussion is restricted to time-invariant control functions $u(x)$, due to the practical difficulty in generating qualitatively meaningful and time-varying controls $u(x,t)$ for neural operator training. Moreover, as our neural operators were limited to one-dimensional input representations (FNO1D, LNO1D), addressing temporal variations explicitly would require transitioning to higher-dimensional architectures such as FNO2D or LNO2D.

#### 4.2.1 Heat Equation (1D)

We consider the controlled heat equation described by:

$$\begin{aligned} \min_{u(x)} \quad & \frac{1}{2}|y(x,T) - y_{target}(x)|_2^2 + \frac{\rho}{2}|u(x)|_2^2, && (31a) \\ \text{s.t.} \quad & \frac{\partial y}{\partial t} = \nu \frac{\partial^2 y}{\partial x^2} + u(x), && (31b) \\ & y(x,0) = 0, \quad y(0,t) = y(1,t) = 0. && (31c) \end{aligned}$$

Here, $\nu$ represents the thermal diffusivity. The target terminal profiles tested include Gaussian distributions defined as:

$$y_{target}(x) = A \exp\left(-\frac{(x - x_0)^2}{2\sigma^2}\right) \tag{32}$$

With typical parameters $A = 0.5$, $x_0 = 0.5$, and $\sigma = 0.1$, we challenge the neural operator to learn smooth diffusive dynamics and optimal control inputs that achieve precise spatial profiles at the final time $T$.

#### 4.2.2 Diffusion-Reaction Equation (1D)

We then introduce nonlinear reaction terms, extending complexity:

$$\begin{aligned} \min_{u(x)} \quad & \frac{1}{2}|y(x,T) - y_{target}(x)|_2^2 + \frac{\rho}{2}|u(x)|_2^2, && (33a) \\ \text{s.t.} \quad & \frac{\partial y}{\partial t} = \nu \frac{\partial^2 y}{\partial x^2} - \alpha y^2 + u(x), && (33b) \\ & y(x,0) = 0, \quad y(0,t) = y(1,t) = 0. && (33c) \end{aligned}$$

Here, the reaction coefficient $\alpha$ introduces nonlinearity, significantly increasing the difficulty. We use more intricate targets, such as double Gaussian profiles defined by:

$$y_{target}(x) = A \exp\left(-\frac{(x - 0.3)^2}{2\sigma^2}\right) + A \exp\left(-\frac{(x - 0.7)^2}{2\sigma^2}\right), \tag{34}$$

with parameters $A = 1.0$ and $\sigma = 0.05$, requiring neural operators to capture nonlinear reaction dynamics accurately.

#### 4.2.3 Burgers' Equation (1D)

Finally, we test neural operators on the viscous Burgers' equation, widely recognized for its challenging nonlinear convective behavior and shock formation:

$$\min_{u(x)} \quad \frac{1}{2}|y(x,T) - y_{target}(x)|_2^2 + \frac{\rho}{2}|u(x)|_2^2, \tag{35a}$$

$$\text{s.t.} \quad \frac{\partial y}{\partial t} + y\frac{\partial y}{\partial x} = \nu\frac{\partial^2 y}{\partial x^2} + u(x), \tag{35b}$$

$$y(x,0) = 0, \quad y(0,t) = y(1,t) = 0. \tag{35c}$$

The selected terminal target is a steep shock profile defined by:

$$y_{target}(x) = \frac{0.1}{1 + \exp(-50(x - 0.5))} \tag{36}$$

This problem evaluates the neural operator's capacity to handle highly nonlinear phenomena and discontinuous solutions, thus offering an intensive assessment of model robustness and representational power.

## 4.3 Dataset

To train neural operators, we implement a general-purpose dataset class, which is able to generate functions of the different families. Each generated function optionally includes the corresponding ground-truth trajectory $x(t)$, computed by numerically solving the governing ODE. The class is flexible to generate both validation and training datasets.

### 4.3.1 Data generation

For each benchmark problem, we can select a list of function families. Then, for each function, we randomly draw a family class and, if the supervision is enabled, randomly select whether the true trajectory is going to be generated. The dataset generating class has the following function families available:

- **Gaussian Random Fields (GRF).** Sample white noise in the Fourier domain with a squared-exponential power spectral density (PSD), apply inverse FFT, normalize, and interpolate to $\{t_i\}$ [42]. This family is quite often used to train neural operators.

- **Linear functions.** Sample slope $a \sim \mathcal{U}(-2, 2)$, intercept $b \sim \mathcal{U}(-1, 1)$, then define $u(t_i) = a\,t_i + b$.

- **Sine waves.** Sample $f \sim \mathcal{U}(0.1, 10)$, $A \sim \mathcal{U}(0.5, 2)$, $\phi \sim \mathcal{U}(0, 2\pi)$, then define $u(t_i) = A\,\sin(2\pi f\,t_i + \phi)$.

- **Polynomial:** Sample degree $d \in \{3, \ldots, 7\}$, coefficients $c_k \sim \mathcal{U}(-3, 3)$, and define
$$u(t_i) = \sum_{k=0}^{d} c_k\,t_i^k.$$

- **Constant:** Draw $c \sim \mathcal{U}(-3, 3)$ and define $u$ as a constant function $u(t_i) = c$ for all $i$.

This process ensures that each family class (among selected ones) is presented nearly equally.

**Range projection.** Some of the differential equations require input controls to lie within specific bounds. Optionally, we apply a linear projection of each control $u(t)$ to the interval $[u_{\min}, u_{\max}]$. This is done via min-max normalization:

$$u_{\text{proj}}(t) = \begin{cases} u_{\min} + \left(\dfrac{u(t) - \min u(t)}{\max u(t) - \min u(t)}\right)(u_{\max} - u_{\min}), & \text{if } \max u(t) \neq \min u(t), \\ \text{random value in } [u_{\min}, u_{\max}], & \text{otherwise.} \end{cases} \tag{37}$$

This ensures that constant or nearly constant control functions are still projected into the target range by injecting mild random variation.

**Batch preparation.** Batches are prepared based on the neural operator architecture. LNO and FNO are trained on the fixed grid, and this is straightforward. The *batching process* includes: define a uniform grid $\{t_i\}$ in the specified domain and evaluate $u(t_i)$ at this uniform grid. Initial and boundary loss is calculated based on the first and last points of the predicted trajectory.

DeepONet is trained on the random $t$ grid in the specified range. The *batching process* includes: sampling evaluation points $\{\tau_j\}$ in specified domain; interpolating $u(t_i)$ to obtain values at $\tau_j$, forming $U_{\text{dom}}$, generate a vector of zeros $\tau_{\text{init}}$ to enforce initial condition at these points.

**Validation set.** To evaluate generalization, we generate a separate validation set using the same procedure. For each control $u^{(k)}(t)$, we compute the ground-truth trajectory $x^{(k)}(t)$ by solving:

$$\dot{x}(t) = d\Big(x(t), u^{(k)}(t)\Big), \qquad x(0) = x_0,$$

using `solve_ivp` from `SciPy` with method `"DOP853"` [43]. We found this method the most reliable and stable for our setup, but any other can be used.

**Dataset Reproducibility.** For each benchmark problem, we generate only one training and validation dataset with fixed seeds. The training set is generated with seed 1234 and the test with 42. The scripts for data generation are available in our *Github* repository [44].

### 4.3.2 Problem specific dataset

Each benchmark problem uses a tailored configuration of control function families and GRF/polynomial settings. Table 1 summarizes the parameters used.

**Table 1:** Dataset configuration for each benchmark problem with listed function families: `const.` - constant function, `lin.` - linear, `poly.` - polynomial, `grf` - GRF. Projection is done if the projection bounds are defined. The degree range is a setting for polynomial and GRF range for GRF function families, respectively.

| **Problem** | **Function Types** | **GRF Range** | **Deg. Range** | **Proj. Bound** |
|---|---|---|---|---|
| Linear | const., lin., poly, grf | [0.02, 0.5] | [3, 7] | – |
| Oscillatory | const., lin., poly, grf, sine | [0.02, 0.5] | [3, 7] | – |
| Poly. Track | const., lin., poly, grf, sine | [0.02, 0.5] | [3, 7] | – |
| Nonlinear | poly, grf | [0.05, 0.5] | [1, 5] | $[-1.5,\ 1.5]$ |
| Singular Arc | poly, grf | [0.05, 0.5] | [1, 5] | $[-3,\ -0.5]$ |

All training datasets consist of 100,000 training samples (with 20% supervised trajectories) and 10,000 validation functions, uniformly discretized over $m = 200$ time points. We chose to use 100,000 functions for the training since, at the early stages, we noticed that though DeepONet minimized physics loss, it was overfitting the training dataset. We discretized over $m = 200$ points because we are going to use numerical approximation of the gradients for the FNO and LNO. Increasing the number of time points generally improves the approximation quality, but for DeepONet, this comes at the cost of requiring a larger model and longer training time.

Polynomial tracking and oscillatory problems benefit from a large number of input families due to the simplicity of their governing differential equations. Thus we added `sine` function family with sine functions with frequency sampled from $(0.1,\ 10)$ and amplitude from $(0.5,\ 2)$.

Projection is applied *only* for the nonlinear problem (to stabilize physics loss due to the quadratic control term) and the singular arc problem (to avoid numerical solver failure from unbounded control values).

**Additional note:** For the singular arc problem, we generate an additional dataset for training the LNO architecture using only `sine` and `polynomial` function types. This variant uses sine functions with frequency sampled from $(0.1,\ 30)$ and amplitude from $(0.5,\ 2)$, while preserving the same ranges for polynomial coefficients and degrees. We generated this dataset replicating the procedure used by Cao *et al.* [11]

For the Heat and Diffusion-reaction problems, we switch to a semi-supervised learning approach. For 50% of input control functions, we generate corresponding trajectories using a `SciPy`-based inverse problem solver. However, for the Burgers' equation, this approach proved unreliable due to solver failures and instability. Instead, we trained the neural operator using only physics-informed losses, relying entirely on residual supervision. For all three PDEs, we sampled control functions from three families: Gaussian Random Fields (GRF), single sine waves, and composite “Fourier“ functions composed of multiple sine terms. In total, we used 10,000 samples for both the Heat and Diffusion-Reaction problems, and 100,000 samples for the Burgers’ problem due to the absence of supervised trajectories.

## 4.4 Evaluation Metrics

First, we evaluate how well the neural operator learns. In each epoch, we calculate physics and initial losses averaged across all batches. We define that loss converged once the physics loss has not improved for 100 epochs.

Since some of the input controls have a labeled trajectory on both the train and validation set, we can calculate *train* and *validation* error using average per sample relative $\mathcal{L}_2$ error mentioned earlier. In this way, we can ensure that the model does not overfit or underfit.

We also provide plots with cumulative execution time (for the training of the neural operator, excluding time spent for plotting) to compare architectures in terms of speed.

Once the neural operator is trained, we use it to solve a corresponding benchmark control problem. For each benchmark problem, we provide a plot that includes:

- Trajectory plots comparing analytical solutions $x^*(t)$ and predicted solution $\hat{x}(t)$.
- Solution to find $u$ using numerical solver (to ensure that the prediction for the found input is correct)
- Plot with optimal input control and found control $u$.

We are going to use this plot to judge the approximated solution qualitatively. To quantitatively assess the solutions, we also provide relative $\mathcal{L}_2$ errors between optimal and predicted input controls $u$ along with trajectory errors.

In addition to trajectory and control errors, we compute the true cost functional for each predicted solution and compare it to the ground-truth optimum $J(u^*)$. This allows us to assess whether the approximated control is near-optimal even if small deviations in $u$ exist.

## 4.5 Selected architectures

### 4.5.1 ODE Benchmark Problems

DeepONet was configured with five hidden layers of 200 neurons in both the branch and trunk networks. The input function was sampled at 200 uniformly spaced sensor points, and the output latent dimension was set to 200. We used `tanh` activation function.

The architecture for FNO1d is configured per problem to balance expressiveness and stability. Table 2 lists the spectral and decoder parameters used.

| **Problem** | **Modes** | **Width (Proj.)** | **Depth (Layers)** | **Decoder Dim** |
|---|---|---|---|---|
| Linear | 16 | 32 | 4 | 128 |
| Oscillatory | 16 | 32 | 4 | 64 |
| Poly. Tracking | 16 | 32 | 4 | 64 |
| Nonlinear | 16 | 64 | 5 | 128 |
| Singular Arc | 64 | 32 | 6 | 256 |

**Table 2:** FNO1d hyperparameters per benchmark problem.

The `width` parameter controls the size of the lifted feature space after the input linear layer. The number of `modes` determines how many low-frequency Fourier coefficients are retained in the spectral layers. The decoder consists of a two-layer MLP with a hidden dimension specified by `Decoder Dim`, and the output dimension is fixed to 1. After each Fourier layer, we use `GeLU` activation function.

We use at least four Fourier layers to maintain expressivity comparable to the Laplace layer used in LNO. For the nonlinear and singular arc problems, we increase the number of Fourier layers further, as these problems demand higher expressivity. Increasing only the projection and decoder dimensions was insufficient to ensure accurate learning for these cases.

Table 3 summarizes the hyperparameters of the LNO used for each benchmark problem.

**Table 3:** LNO1d hyperparameters used for each benchmark problem, where BN refers to batch-normalization.

| **Problem** | **Modes** | **Width (Proj.)** | **Depth** | **Decoder Dim** | **BN** |
|---|---|---|---|---|---|
| Linear | 8 | 4 | 1 | 64 | No |
| Oscillatory | 8 | 4 | 1 | 64 | No |
| Poly. Tracking | 8 | 4 | 1 | 64 | No |
| Nonlinear | 8 | 4 | 2 | 32 | Yes |
| Singular Arc | 32 | 16 | 1 | 128 | Yes |

Each LNO model uses at least one block composed of a pole-residue (PR) operator and pointwise convolution, followed by optional batch normalization and a SiLU activation (used throughout). For the nonlinear and singular arc problems, we increase expressivity via depth or width, and apply batch normalization to stabilize training. The decoder consists of a two- or three-layer MLP depending on model depth.

Our experiments suggest that (see Section 5) the expressivity of LNO was not enough to learn nonlinear and singular arc problems, so multiple architectural settings were tested, along with the extension of LNO to have more Laplace layers and a deeper decoder network.

All architectures are trained to minimize physics residual and initial condition violation. We do not include boundary violation during training of the neural operator, since this would complicate dataset generation (we would need to select only those inputs that lead to satisfied boundary conditions, while satisfying the differential equation with initial condition).

We adopted the DeepONet architecture from the original work introducing neural operators for control settings [13], as it was demonstrated to be sufficiently expressive for a wide range of problems. For LNO, we initially followed the configuration suggested by the original authors, which used a hidden layer of size 128 and a projection width of 4, with the number of modes adapted to the complexity of the target trajectories.

However, LNO failed to learn the dynamics of the **nonlinear** and **singular arc** problems (see Section 5). To address this, we performed a manual hyperparameter search to increase model expressivity, experimenting with deeper decoders, wider projection layers, and more Laplace (PR+Conv) blocks. During this process, we observed that increasing the width and number of modes often led to training instabilities, which we mitigated by incorporating batch normalization.

For FNO, we also conducted manual tuning of hyperparameters. The goal was to achieve convergence of the physics loss to a sufficiently low value while maintaining a compact model architecture.

**Optimizer and hyperparameters.** All models are trained using the *Adam optimizer* [14].

- **Learning rate:** $\eta_0 = 10^{-3}$ for LNO and FNO. $\eta_0 = 10^{-4}$ for DeepONet.
- **Scheduler:** We use a step scheduler for all problems. For DeepOnet, we halve the learning rate every 200 epochs. For FNO and LNO every 25-50, we decrease the learning rate with $\gamma = 0.9$
- **Batch size:** 128 samples per batch.

- **Epochs:** Up to 1000 for LNO and FNO, with early stopping based on physics loss that is calculated on the training set. Up to 2,000 for DeepONet.[1]

### 4.5.2 PDE Benchmark Problems

Previous work has demonstrated that DeepONet can successfully learn the diffusion-reaction equation and be applied in control settings [13]. However, its training time is prohibitively long (exceeding one week). We also attempted to apply LNO to these PDE problems, but - as with the nonlinear ODE case - it failed to learn the system dynamics under noisy inputs. As a result, we chose to restrict our final benchmarking to the FNO architecture, which, as will be shown, proved to be the most robust, reliable, efficient, and generalizable neural operator in our setting.

All models were trained on a spatiotemporal grid consisting of 64 spatial points and 100 time steps. All FNO models were configured with `modes=32`, `width=64`, and `hidden_layer=128`. The heat equation model used four Fourier layers, while both the diffusion-reaction and Burgers' equation models used five.

## 4.6 Baseline Comparison with Traditional Solver

To provide a direct baseline for model performance, we solve each benchmark optimal control problem using a traditional direct transcription method implemented with `CasADi` [45] and the `IPOPT` [16] nonlinear programming solver. Each problem is discretized using the same time grid ($N = 200$) as in the neural operator experiments. The resulting nonlinear program is solved to global optimality, yielding reference state and control trajectories.

For each problem, we compute the relative $\mathcal{L}_2$ errors in the optimal state $x$ and control $u$ by comparing the solver output to the analytical solution:

$$\text{Rel. Error}_x = \frac{\|x_{\text{opt}} - x^*\|_2}{\|x^*\|_2}, \qquad \text{Rel. Error}_u = \frac{\|u_{\text{opt}} - u^*\|_2}{\|u^*\|_2}. \tag{38}$$

We also report the value of the cost functional and the total solve time for each run. All baseline solutions and statistics are saved in the `baseline_solutions` directory, and the complete implementation scripts are available in the accompanying GitHub repository [44].

This baseline enables a quantitative comparison of neural operator-based solutions to those obtained with a state-of-the-art traditional solver, both in terms of accuracy and computational efficiency.

[1]We did not use validation loss for early stopping since it was not always correlated with physics loss convergence

## 4.7 Experiments with Control Problem

Given an input control $u$, the resulting state trajectory $x$ incurs several types of losses: the *physics* loss (residual of the governing dynamics), the *initial* loss (deviation at $t = 0$), the *boundary* loss (if terminal constraints are present), and the *objective* loss (cost functional to minimize). As described in Section 3, these constraints are imposed softly by penalizing their violations during optimization.

To promote well-behaved control profiles, we additionally include a *smoothness penalty* on the control $u$, defined as:

$$\mathcal{L}_{\text{smooth}} = \frac{1}{N-1} \sum_{i=1}^{N-1} (u_i - u_{i-1})^2 \tag{39}$$

This penalizes significant variations between adjacent time steps, encouraging temporally smooth control trajectories. We experimented with different weight settings to balance these losses in the total objective. Table 4 summarizes the tested weight ranges.

**Table 4:** Weight ranges tested for different loss components in control problem experiments.

| **Loss Component** | **Weight Range Tested** |
|---|---|
| Physics loss $w_{\text{phys}}$ | $\{1,\ 10,\ 100,\ 500\}$ |
| Initial condition loss $w_{\text{init}}$ | $\{1,\ 10\}$ |
| Boundary condition loss $w_{\text{bdy}}$ | $\{1,\ 10,\ 100,\ 500\}$ |
| Objective (cost) loss $w_{\text{obj}}$ | $\{0.1,\ 1,\ 10\}$ |
| Smoothness penalty $w_{\text{smooth}}$ | $\{0.0001,\ 0.001,\ 0.1, 1\}$ |

For all our experiments, we used a fixed learning rate $\eta = 0.001$, treating the control input $u$ as the sole trainable parameter. No learning rate scheduler was applied. For the nonlinear and singular arc problems, we enforced control bounds at each optimization step by projecting $u$ onto the target range using simple *clamping*, i.e., replacing out-of-bound values with the nearest bound.

We also experimented with a soft constraint approach by passing $u$ through a `tanh` activation function, which maps the values to the interval $[0, 1]$, and then linearly rescaling to the desired bounds. However, this method yielded results comparable to clamping, so we adopted the more straightforward clamping strategy throughout.

## 4.8 Sensitivity Analysis: Robustness to Control Objective Weight

To evaluate the robustness of trained neural operator models, we conduct a sensitivity analysis by varying the control objective weight $\rho$ in the cost functional for each benchmark problem. The goal is to determine whether the quality of the resulting control solutions is sensitive to changes in $\rho$ when using a model trained solely on the system dynamics. For each combination of benchmark problem and neural operator architecture, we perform the following steps:

1. **Model Loading.** Load the previously trained model checkpoint for each problem-architecture pair; the model weights are held fixed throughout.

2. **Control Objective Weight Sweep.** For each $\rho \in \{0.1, 1.0, 10.0\}$, redefine the cost functional to apply a stronger or weaker penalty to the control input $u(t)$, with the state penalty left unchanged.

3. **Control Optimization.** Solve the optimal control problem by optimizing $u(t)$ via gradient descent, using the trained neural operator as a surrogate for the system dynamics. For each setting, three independent runs are performed with different random seeds to average out optimizer stochasticity.

4. **Metrics Collected.** For each run, we record the relative $\mathcal{L}_2$ errors on the control and trajectory, the final value of the objective, and the optimization time.

This analysis covers all five benchmark problems (“linear“, “oscillatory“, “polynomial tracking“, “nonlinear“, “singular arc“) and all three architectures (DeepONet, FNO, LNO). In total, $5 \times 3 \times 3 \times 3 = 135$ combinations are tested.

The time grid and bounds matched those used in training and evaluation ($N = 200$ points). The initial guess for the control input was sampled from a uniform distribution for each run in all problems. Control optimization is run for up to 50,000 iterations with early stopping (patience of 1,000 epochs) and fixed learning rate ($\eta = 10^{-3}$). We note that the physics residual weight ($w_{\text{phys}}$) used during training is not varied in this analysis, as retraining neural operators for every possible configuration is computationally prohibitive. The focus here is on evaluating the generalization of *already trained* models to new control objective weights.

All experiments were designed for full reproducibility. We fixed the random seed used to generate both the training and validation datasets, re-using the same seed across every benchmark problem. The training pipeline logs key diagnostics - physics loss, initial-condition loss, and supervised trajectory error - at every epoch (see Section 5). All source code - including model definitions, training routines, evaluation scripts, and the complete sensitivity-analysis table - is released under the MIT License [46] and is available in the accompanying GitHub repository [44], ensuring that results can be replicated and extended. Finally, in terms of hardware, all experiments were run on a workstation equipped with an Intel Core i5-12600K CPU, an NVIDIA RTX A2000 GPU (12 GB), and 32 GB of RAM.

# 5 Results

This chapter presents both plots and numerical results from applying the trained neural operators to benchmark optimal control problems.

## 5.1 Problem: Linear

The architectures DeepONet, FNO, and LNO described in Section 4.5 successfully learned the governing dynamics of the system and satisfied the initial condition for the Linear ODE benchmark (see Figure 7).

In terms of convergence speed and physics loss, FNO performed best overall. LNO achieved the lowest physics loss but required more training epochs than FNO. DeepONet converged the slowest but achieved the most accurate satisfaction of the initial condition. Summary statistics are provided in Table 5.

**Table 5:** Training time, total epochs, and final loss values for each neural operator architecture.

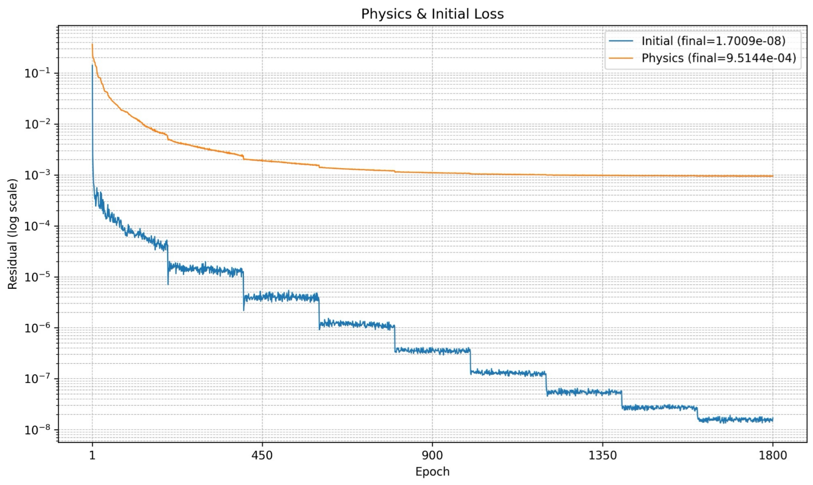
| Architecture | Training Time (min) | Epochs | Physics Loss | Initial Loss |
|---|---|---|---|---|
| DeepONet | 1267 | 1800 | $9.51 \cdot 10^{-4}$ | $1.70 \cdot 10^{-8}$ |
| FNO | 118 | 180 | $3.70 \cdot 10^{-4}$ | $2.53 \cdot 10^{-6}$ |
| LNO | 209 | 670 | $3.25 \cdot 10^{-4}$ | $1.23 \cdot 10^{-6}$ |

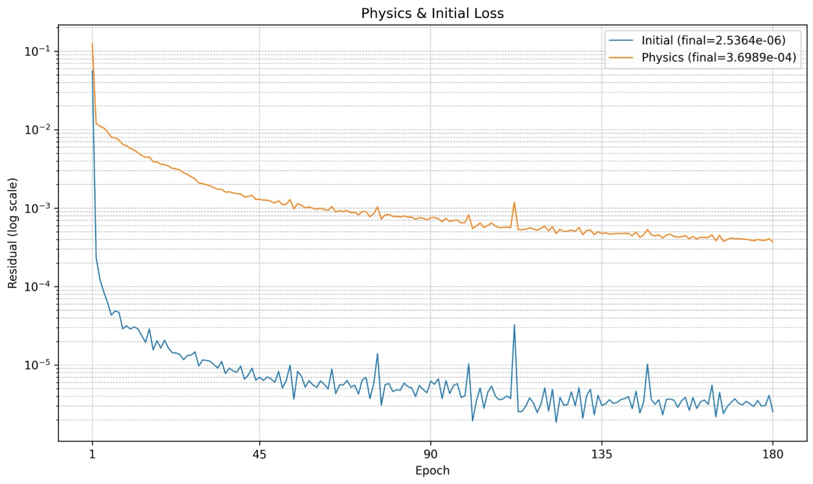
At each training epoch, we computed train and validation losses based on the relative error between predicted and ground-truth trajectories (see Figure 8). Note that although the models were trained using only the physics residual and initial condition, they generalize reasonably well to unseen trajectories.

The training loss converges smoothly in all models (except for LNO, which exhibited a temporary spike in the initial loss around epoch 180, possibly because of too large a learning rate). The validation loss exhibits oscillations - particularly for FNO and LNO - but we observe no signs of overfitting.

**Table 6:** Comparison of training and validation relative trajectory errors across three architectures.

| Architecture | Relative Train Error of $\hat{x}$ | Relative Test Error of $\hat{x}$ |
|---|---|---|
| DeepONet | $7.62 \cdot 10^{-4}$ | $6.14 \cdot 10^{-4}$ |
| FNO | $2.00 \cdot 10^{-3}$ | $2.67 \cdot 10^{-3}$ |
| LNO | $1.27 \cdot 10^{-3}$ | $1.09 \cdot 10^{-3}$ |

a) 

b) 

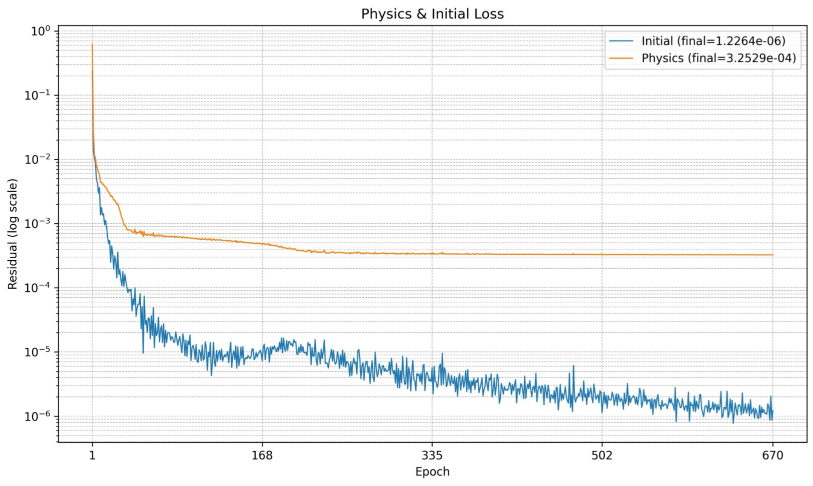
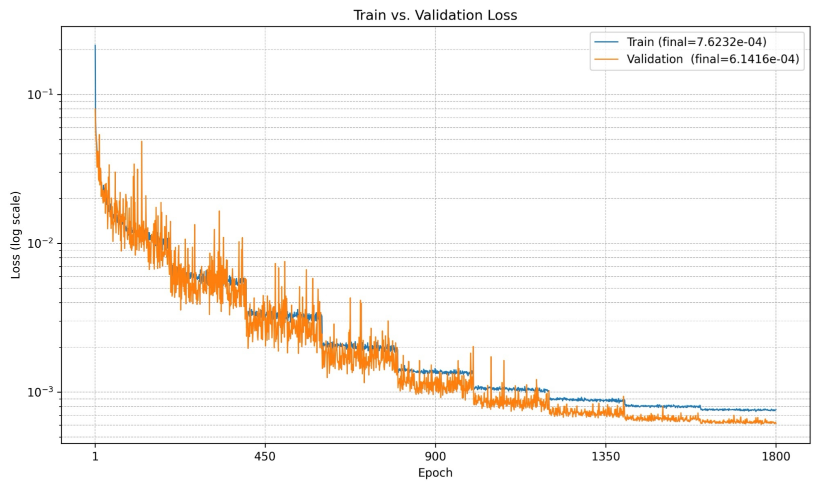
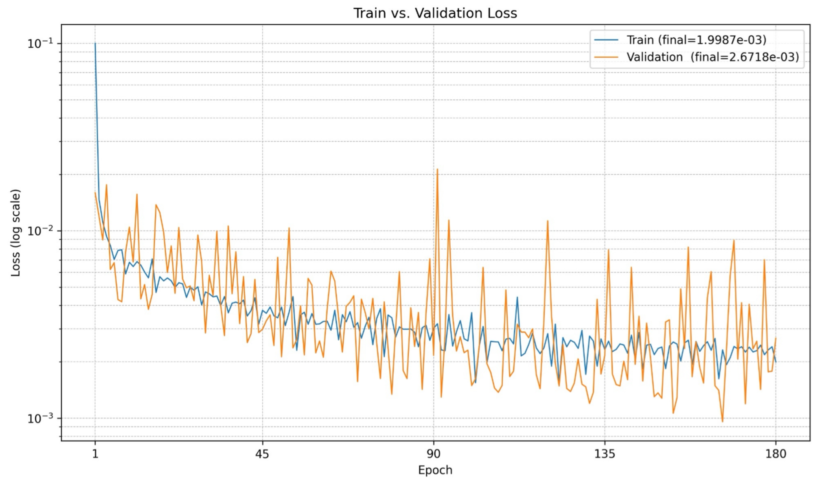
c) 

**Figure 7:** Convergence of physics and initial condition losses for the Linear ODE benchmark across three architectures: a) DeepONet, b) FNO, c) LNO

a)



b)



c)

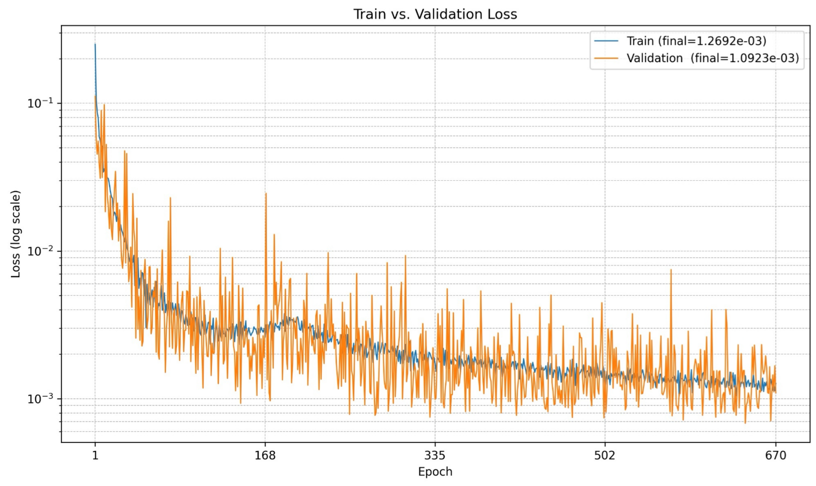


**Figure 8:** Relative trajectory errors (train and validation) for the Linear ODE benchmark across three architectures: a) DeepONet, b) FNO, c) LNO

In terms of trajectory prediction error, DeepONet performed best (see Table

6), likely due to its very low initial loss. This hypothesis is supported by LNO ranking second and FNO third in trajectory accuracy. The worst predictions on the test set are not worse than 1% relative error for all models.

After training the neural operators, we solved the associated control problem by tuning the weights for the physics residual, initial loss, control smoothness, and cost objective. The best-performing weight configurations for each model are shown in Table 7. As an initial guess, we choose a random vector sampled from a uniform distribution, since it has the richest gradient to start with. These weights are defined as in (16).

**Table 7:** Best-performing loss weights used in the control optimization phase.

| **Architecture** | $\mu_{\text{phys}}$ | $\mu_{\text{init}}$ | $\mu_{\text{smooth}}$ | $\mu_{\text{obj}}$ | $\mu_{\text{bound}}$ |
|---|---|---|---|---|---|
| DeepONet | 100 | 1 | 1 | 1 | 0 |
| FNO | 10 | 1 | 1 | 0 | 0 |
| LNO | 10 | 1 | 1 | 0 | 0 |

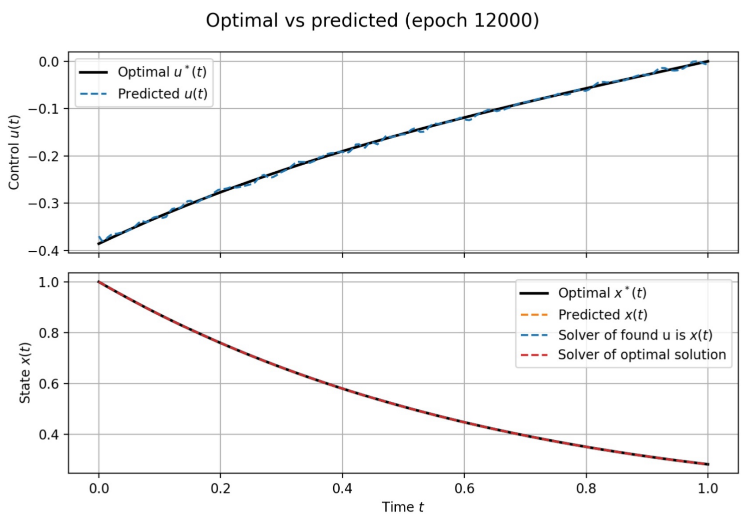
With these weights, we obtained the following candidate control functions, visualized in Figure 9. Among the candidate controls, LNO produced the closest match to the optimal input, followed by FNO, while DeepONet has more noise. The quantitative evaluation is summarized in Table 8.

**Table 8:** Performance of the optimized control $\tilde{u}$: cost value, optimization epochs, and control accuracy.

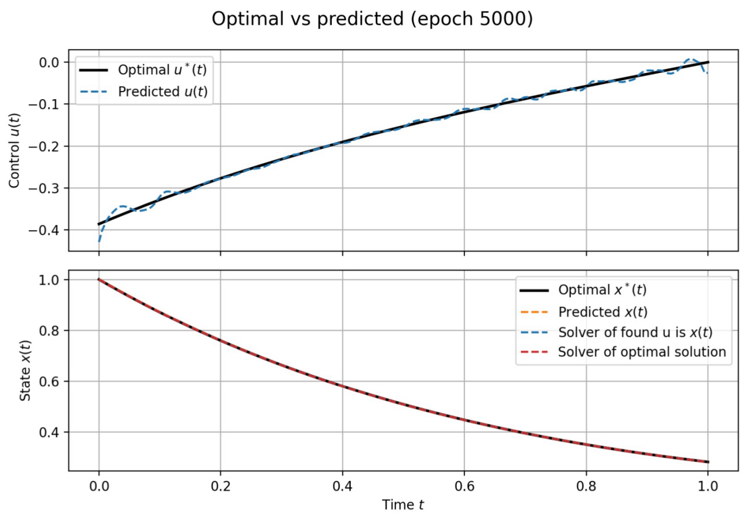
| **Architecture** | **Cost** $J(\tilde{u})$ | **Epochs** | **Relative Error of** $\tilde{u}$ | **Relative Error of** $\tilde{x}$ |
|---|---|---|---|---|
| DeepONet | 0.193 | 12000 | 0.0191 | 0.0005 |
| FNO | 0.193 | 5000 | 0.0348 | 0.0010 |
| LNO | 0.193 | 5000 | 0.0142 | 0.0014 |

With DeepONet, we needed twice the number of epochs as with FNO and LNO. All models were able to approximate the analytical solution with high accuracy. FNO and LNO are less accurate at the boundaries due to numerical approximation of the gradients. Solutions of DeepONet are slightly oscillatory along the whole time range. LNO produced the smoothest and accurate solution overall.

a)



b)



c)

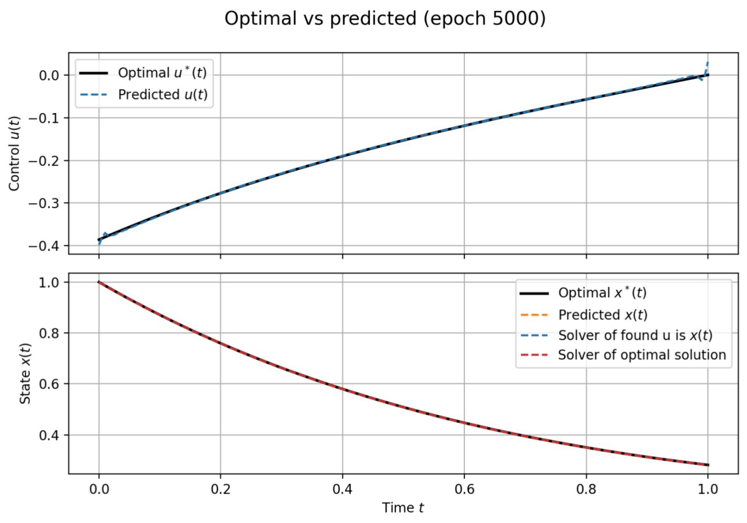


**Figure 9:** Predicted vs analytical control trajectories for the Linear ODE: a) DeepONet, b) FNO, c) LNO

## 5.2 Problem: Oscillatory Forcing

The neural operator architectures DeepONet, FNO, and LNO (see Section 4.5) were trained to model the dynamics of the oscillatory forcing problem. All models successfully minimized the physics and initial condition losses to a satisfactory level (see Figure 10). FNO and DeepONet demonstrated steady convergence for both losses. In contrast, LNO plateaued early on the physics loss and primarily focused on minimizing the initial loss. As expected, FNO performed the best on this oscillatory task due to its ability to efficiently learn periodic structures.

Table 9 summarizes the training performance. FNO achieved the best physics and initial loss in the shortest period of time.

**Table 9:** Training statistics for the oscillatory forcing benchmark.

| Architecture | Training Time (min) | Epochs | Physics Loss | Initial Loss |
|---|---|---|---|---|
| DeepONet | 1236 | 1580 | $3.49 \cdot 10^{-4}$ | $2.99 \cdot 10^{-6}$ |
| FNO | 60 | 770 | $1.18 \cdot 10^{-4}$ | $4.72 \cdot 10^{-7}$ |
| LNO | 67 | 1000 | $4.23 \cdot 10^{-4}$ | $8.66 \cdot 10^{-7}$ |

Figure 11 shows the relative training and validation trajectory errors. All models exhibited overfitting to the training dataset. DeepONet had not yet converged in terms of relative error, as its initial loss was still improving (as seen in Figure 10). LNO's validation loss oscillated significantly, likely due to fluctuations in the initial loss (since the physics loss had converged early). A possible reason for overfitting is insufficient diversity in the function families used during training.

FNO achieved the best overall performance in terms of both training and validation errors (see Table 10). While LNO had a slightly lower test error, its performance was unstable due to oscillations and lacked consistent convergence like FNO.

In edge cases, the worst LNO predictions still had relative errors below 2%, while FNO and DeepONet occasionally produced poor predictions for highly noisy input functions. See Figure 12 for the worst prediction of DeepONet.

**Table 10:** Relative trajectory errors for the Oscillatory problem.

| Architecture | Train Error $\mathcal{L}_2(\hat{x})$ | Test Error $\mathcal{L}_2(\hat{x})$ |
|---|---|---|
| DeepONet | $2.78 \cdot 10^{-3}$ | $6.49 \cdot 10^{-3}$ |
| FNO | $1.09 \cdot 10^{-3}$ | $2.78 \cdot 10^{-3}$ |
| LNO | $1.44 \cdot 10^{-3}$ | $1.75 \cdot 10^{-3}$ |

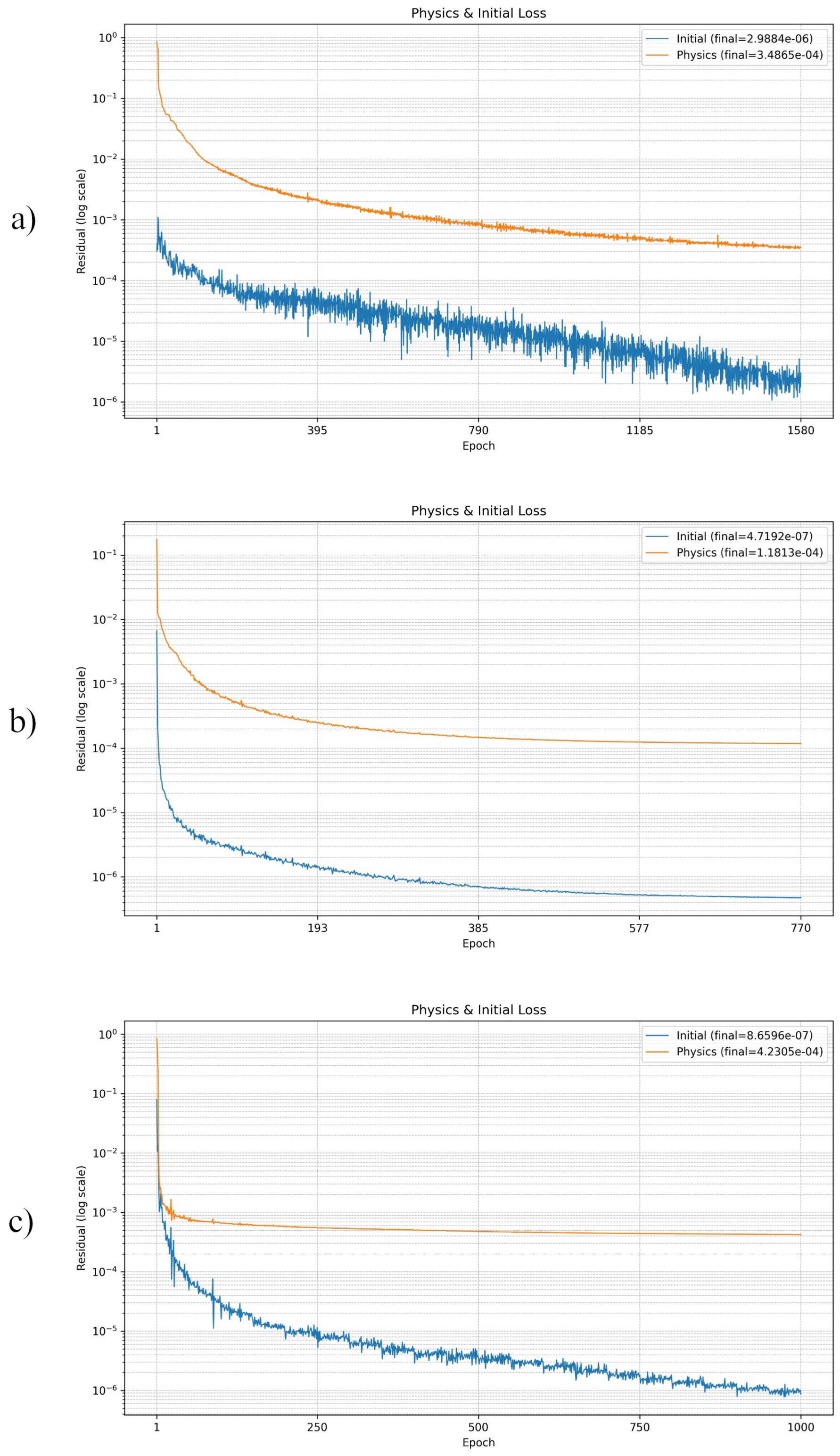
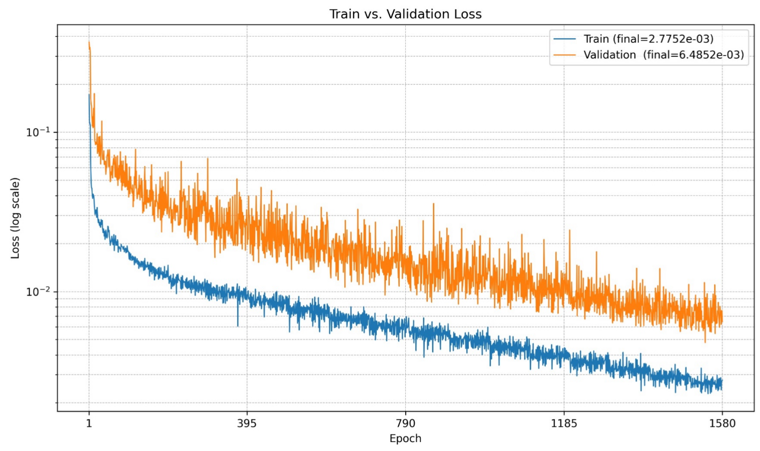
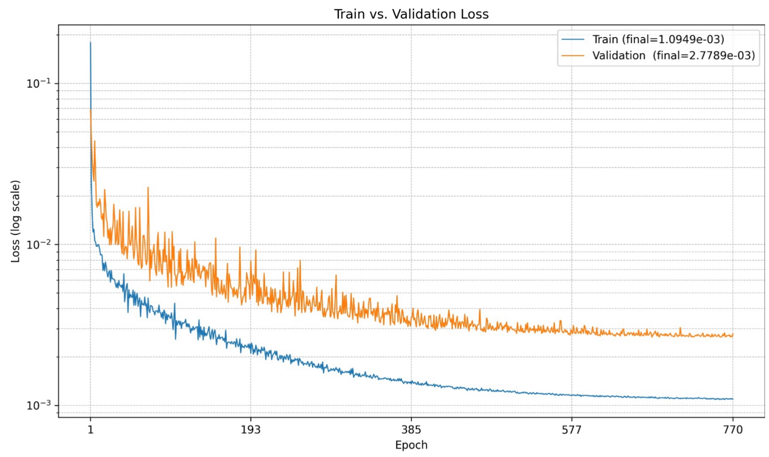


**Figure 10:** Physics and initial condition loss convergence for the oscillatory forcing benchmark: a) DeepONet, b) FNO, c) LNO

a)



b)



c)

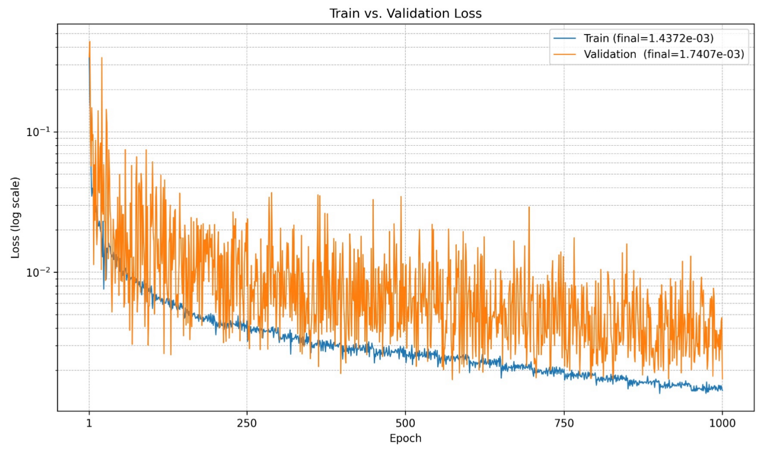


**Figure 11:** Relative trajectory errors (train/test) for the oscillatory forcing benchmark vs epochs: a) DeepONet, b) FNO, c) LNO

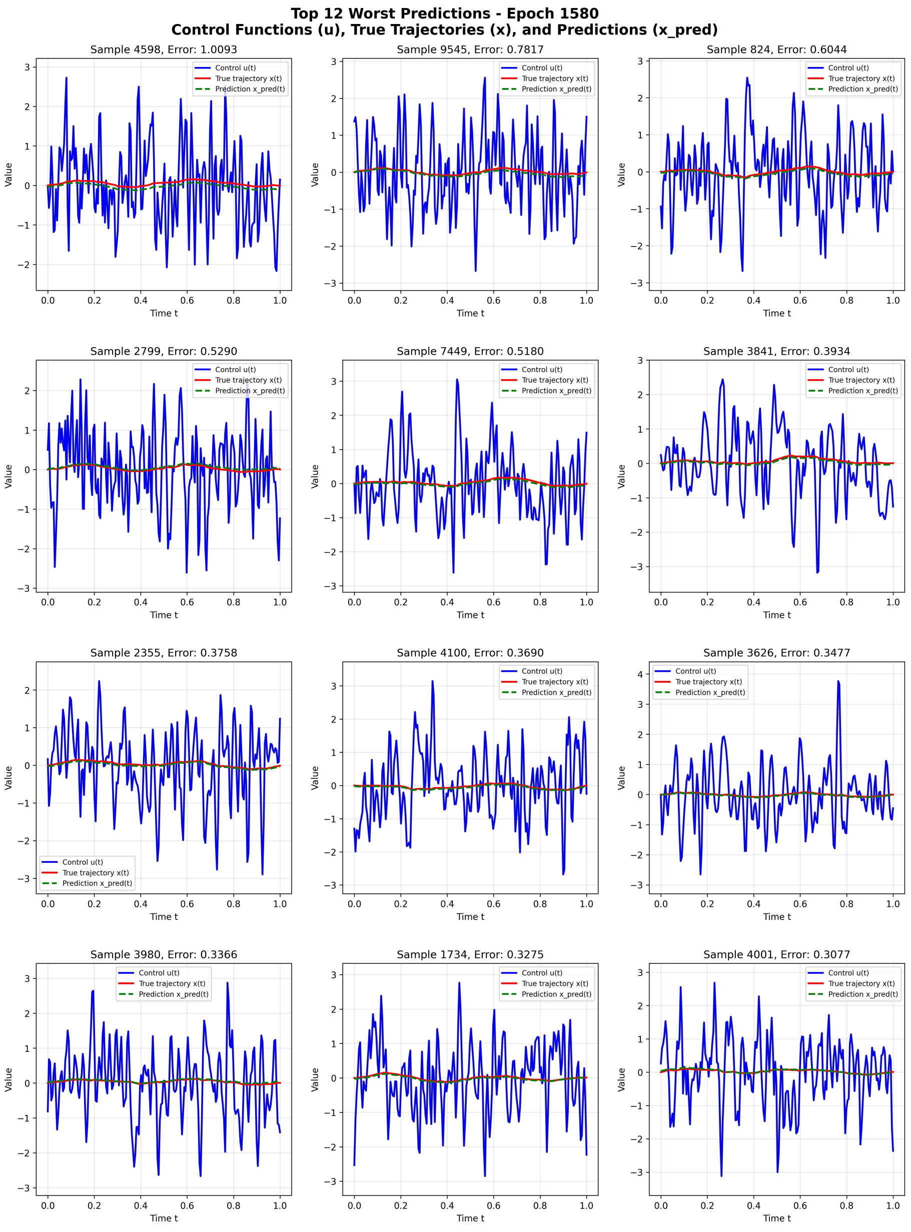


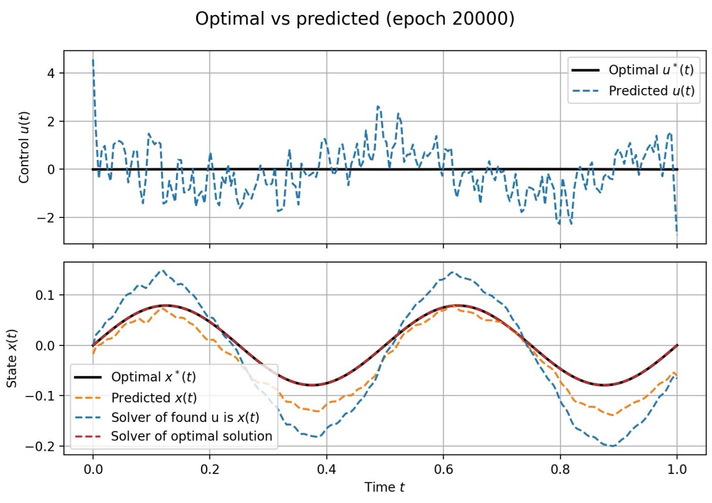
**Figure 12:** Worst predictions on the validation set of DeepONet for Oscillatory Benchmark

**Table 11:** Control optimization weights for the oscillatory forcing problem.

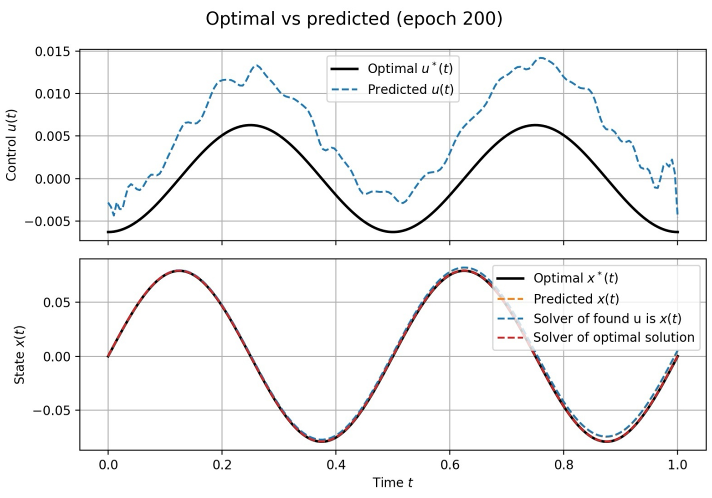
| **Architecture** | $\mu_{\text{phys}}$ | $\mu_{\text{obj}}$ | $\mu_{\text{init}}$ | $\mu_{\text{smooth}}$ | $\mu_{\text{bound}}$ |
|---|---|---|---|---|---|
| DeepONet | 100 | 1 | 1 | 0 | 100 |
| FNO | 10 | 1 | 1 | 0 | 0.1 |
| LNO | 10 | 1 | 1 | 0 | 0.1 |

Figure 13 shows the predicted control trajectories for each architecture. Among them, LNO produced the most accurate approximation of the analytical control.

a)



b)



c)

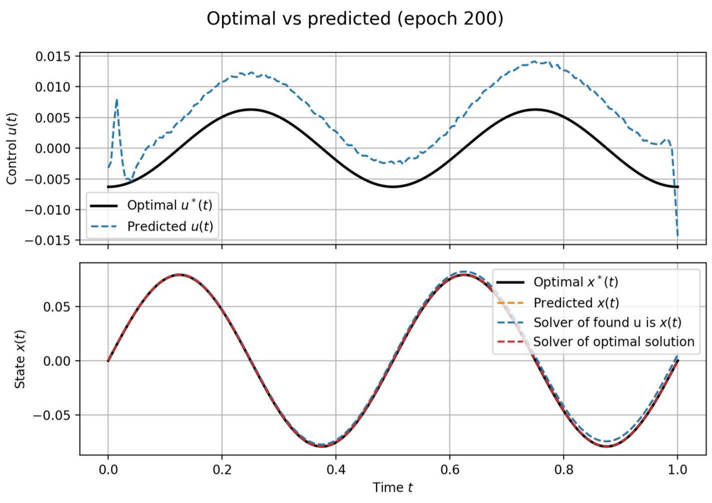
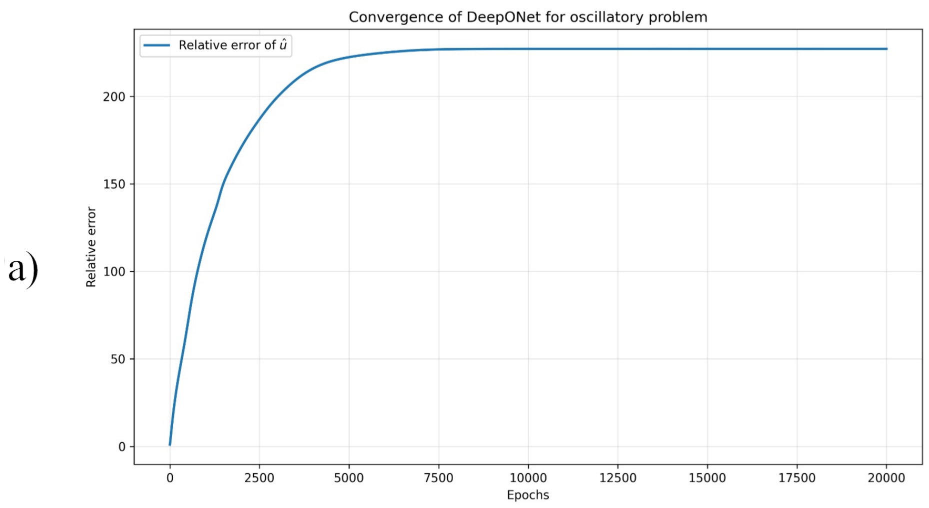


**Figure 13:** Predicted vs analytical control trajectories for the Oscillatory benchmark: a) DeepONet, b) FNO, c) LNO. DeepONet prediction of trajectory for the found control $u$ is incorrect (this was confirmed by solving an instance of the equation with `SciPy` solver)

From the plots, we observe that the optimal control input has an oscillatory form. Both FNO and LNO successfully captured this structure, including the correct phase. However, the accuracy of the predicted control suffered due to the small amplitude of the actual control signal. The numerical gradient approximations used in FNO and LNO introduced errors that significantly affected the result. The found input control is also less accurate near the borders again due to the approximation of the gradient.

The input control found by DeepONet is incorrect in phase and amplitude. Predicted trajectory for this control $\hat{u}$ deviates significantly from the `SciPy`-computed reference, whereas FNO and LNO are more accurate. Only the initial condition and period of the predicted trajectory are nearly identical.



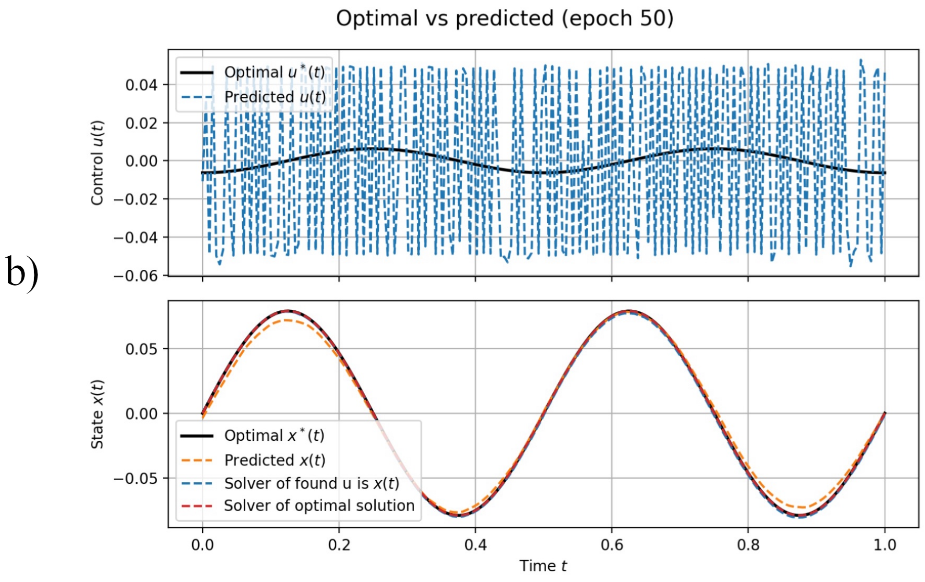


**Figure 14:** Poor performance of DeepONet on oscillatory forcing benchmark

During the optimization step, though DeepONet was trained to have a low physics and initial losses, the total loss was decreasing while the relative error of $\hat{u}$ was diverging each epoch (see Figure 14). Tuning of the weights in the loss function had no effect.

Random noise found by DeepONet on the 50th epoch had less relative error (see Figure 14), since at least it had a closer amplitude of the signal. Recall that DeepONet had highly inaccurate predictions on some noisy input functions (see Figure 12). This led to a highly inaccurate solution since intermediate solutions found in Algorithm 1 are all noisy.

We could potentially improve results by generating a dataset with smaller-amplitude signals. However, since the training was unsupervised and no assumptions about the actual control were made, DeepONet yielded the weakest performance in this setting. In contrast, LNO and FNO achieved near-optimal solutions.

Final performance metrics of the optimized control are reported in Table 12.

**Table 12:** Optimized control quality metrics for the oscillatory forcing problem.

| **Model** | **Cost** $J(\tilde{u})$ | **Epochs** | **Relative Error** $\mathcal{L}_2$ **of** $\tilde{u}$ | **Relative Error** $\mathcal{L}_2$ **of** $\tilde{x}$ |
|---|---|---|---|---|
| DeepONet | 2.02981 | 12000 | 454.27 | 1.4484 |
| FNO | 0.00156 | 200 | 1.2763 | 0.0516 |
| LNO | 0.00156 | 200 | 1.3008 | 0.0535 |

Both FNO and LNO found relatively the same solution, with FNO being slightly more accurate.

## 5.3 Problem: Polynomial Tracking

The neural operator architectures DeepONet, FNO, and LNO (see Section 4.5) were trained to model the dynamics of the polynomial tracking problem. All models successfully minimized both the physics and initial condition losses to a satisfactory level (see Figure 15). The convergence behavior is similar to that of the oscillatory forcing problem, which is expected, as the differential equation remains the same but without the forcing term.

The convergence curve of FNO is steady, except for a slight fluctuation around the 50th epoch. In contrast, both DeepONet and LNO exhibit oscillatory behavior in their loss curves.

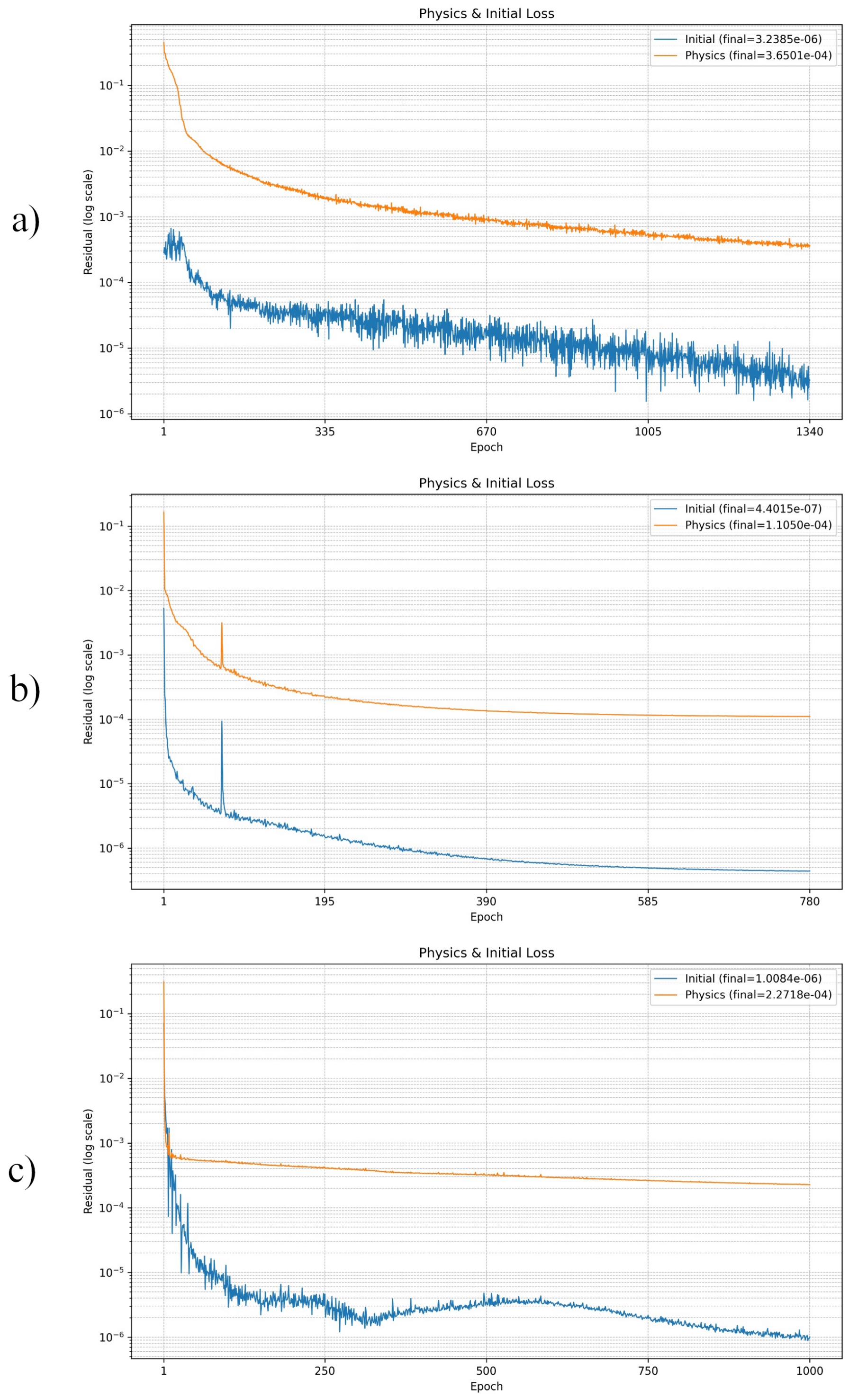
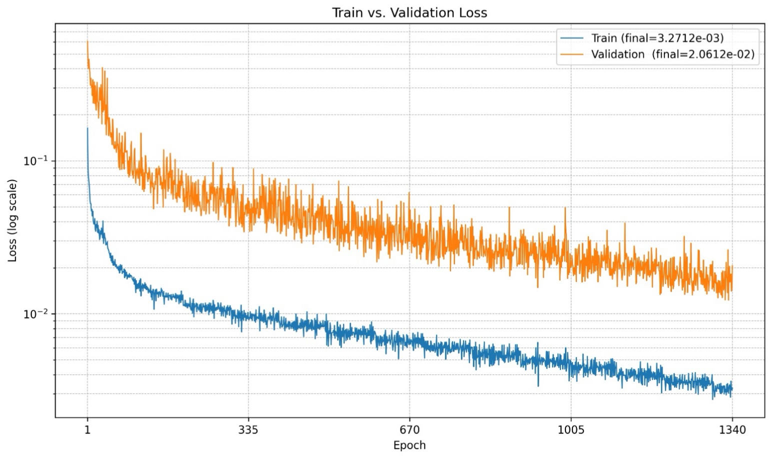
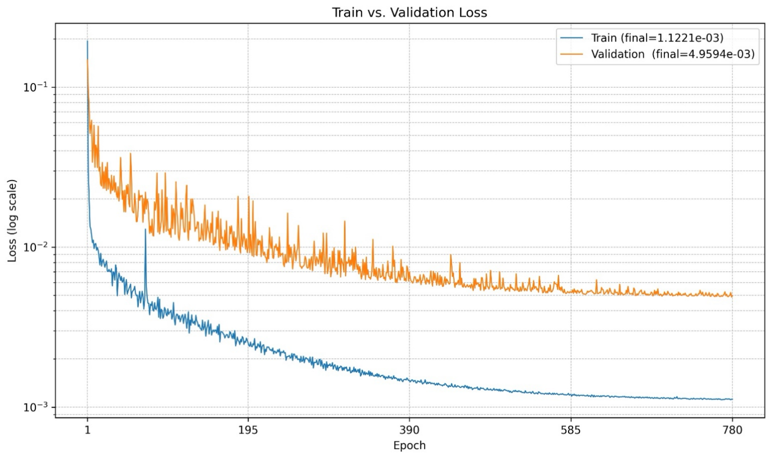


**Figure 15:** Convergence of physics and initial condition losses for the polynomial tracking benchmark: a) DeepONet, b) FNO, c) LNO

a)



b)



c)

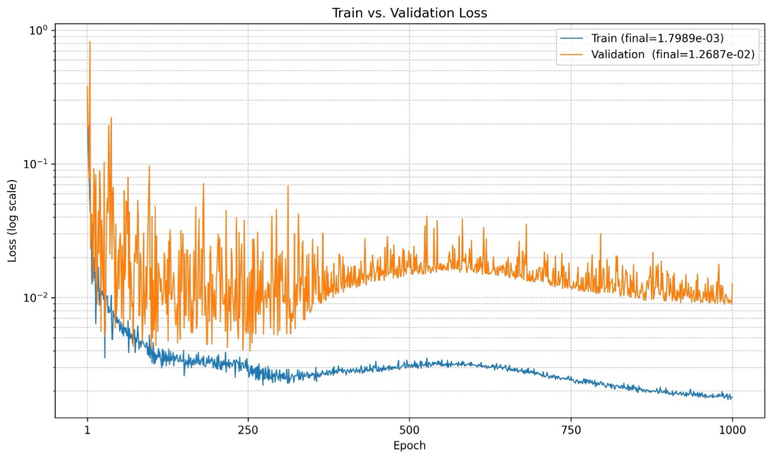


**Figure 16:** Relative trajectory errors (train/test) for the polynomial tracking benchmark across epochs: a) DeepONet, b) FNO, c) LNO

Table 13 summarizes the training performance. FNO achieved the best performance in terms of both losses and runtime, followed by LNO. DeepONet performed the worst among the three.

**Table 13:** Training statistics for the polynomial tracking benchmark.

| Architecture | Training Time (min) | Epochs | Physics Loss | Initial Loss |
|---|---|---|---|---|
| DeepONet | 1251 | 1340 | $3.65 \cdot 10^{-4}$ | $3.24 \cdot 10^{-6}$ |
| FNO | 72 | 780 | $1.11 \cdot 10^{-4}$ | $4.40 \cdot 10^{-7}$ |
| LNO | 79 | 1000 | $2.27 \cdot 10^{-4}$ | $1.01 \cdot 10^{-6}$ |

Figure 16 shows the relative training and validation trajectory errors. There are signs of overfitting, potentially due to a limited number of function families or a relatively small validation set.

As with the oscillatory forcing benchmark, the LNO exhibited highly oscillatory behavior in the initial loss curve. However, this time it converged to a local minimum.

Overall, FNO achieved the best performance in terms of the relative error of predicted trajectories. Both DeepONet and LNO showed significantly higher errors on the validation set.

**Table 14:** Relative trajectory errors for the polynomial tracking benchmark.

| Architecture | Train Error $\mathcal{L}_2(\hat{x})$ | Test Error $\mathcal{L}_2(\hat{x})$ |
|---|---|---|
| DeepONet | $3.27 \cdot 10^{-3}$ | $2.06 \cdot 10^{-2}$ |
| FNO | $1.22 \cdot 10^{-3}$ | $4.96 \cdot 10^{-3}$ |
| LNO | $1.80 \cdot 10^{-3}$ | $1.27 \cdot 10^{-2}$ |

DeepONet struggled particularly with noisy input functions, similar to its performance on the oscillatory forcing problem. FNO had difficulty with constant and noisy inputs, while LNO struggled most with constant functions and with sine inputs (see Figure 17). Both FNO and LNO performed badly on the constant function, possibly due to their inaccuracies in numerical gradient estimation.

The trained neural operators were then used to solve the associated control problem. The best-performing control optimization weights are listed in Table 15. The initial guess for the algorithm was a random vector. Initially, the output of all models was noisy, which required us to add some weight to the smoothness parameter.

**Table 15:** Control optimization weights for the polynomial tracking benchmark.

| Architecture | $\mu_{\text{phys}}$ | $\mu_{\text{obj}}$ | $\mu_{\text{init}}$ | $\mu_{\text{smooth}}$ | $\mu_{\text{bound}}$ |
|---|---|---|---|---|---|
| DeepONet | 200 | 1 | 1 | 2 | 0 |
| FNO | 100 | 1 | 1 | 1 | 0 |
| LNO | 100 | 1 | 1 | 1 | 0 |

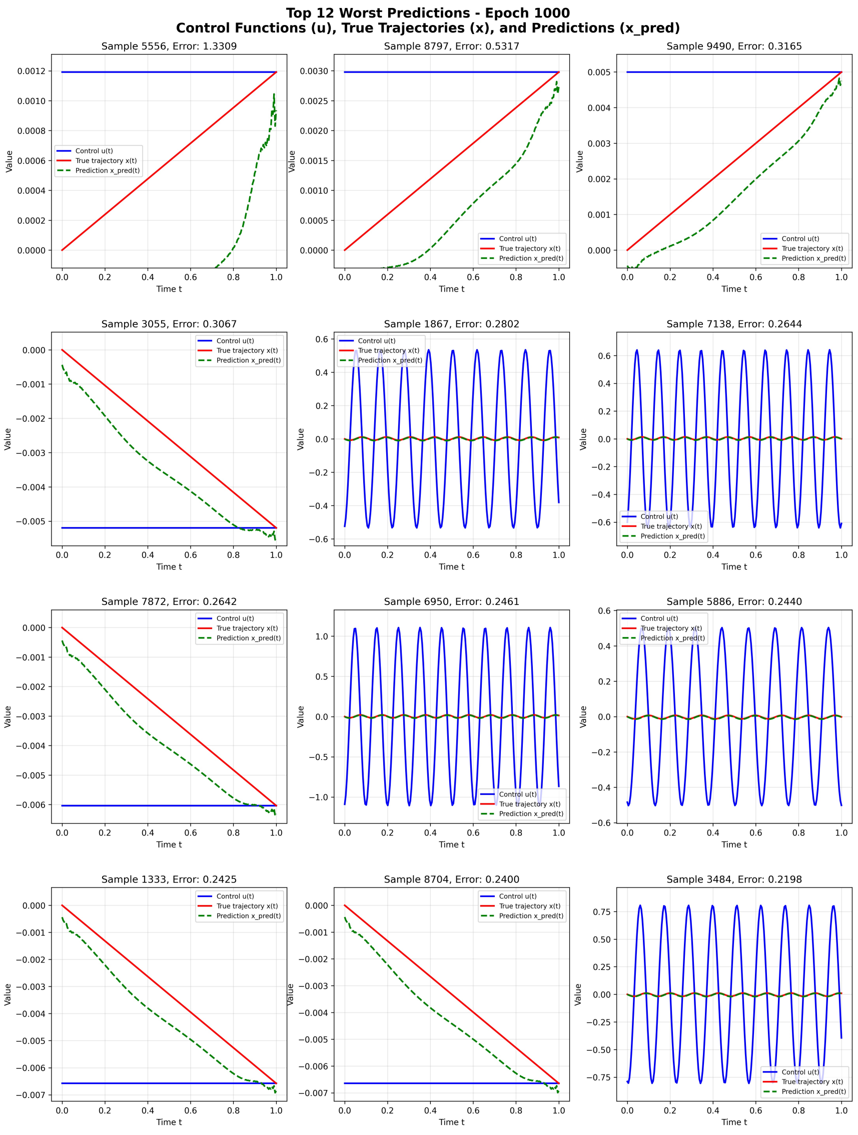


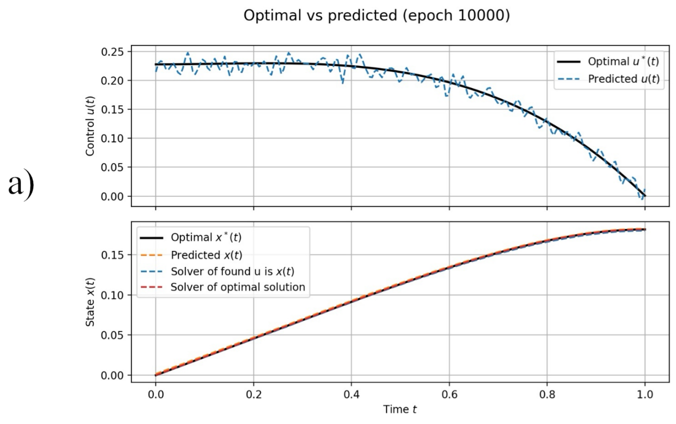
**Figure 17:** Worst predictions on the validation set for LNO for polynomial tracking benchmark.

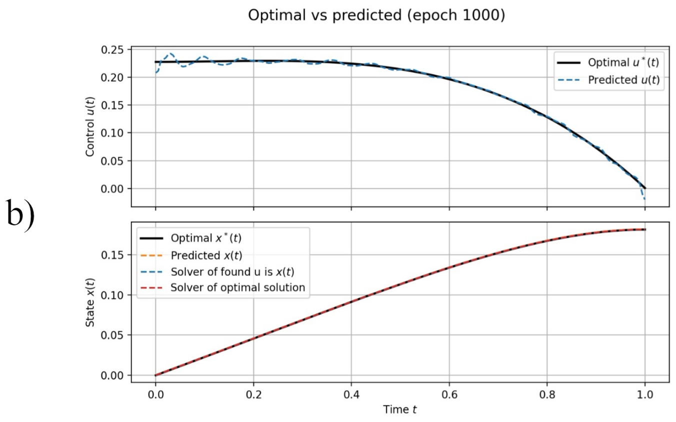
Figure 18 presents the predicted control trajectories for each model. FNO and LNO produced overall accurate control trajectories, except near the domain boundaries. DeepONet produced the noisiest solution, and the predicted trajectory slightly deviated from the one obtained using `SciPy` optimization on the recovered signal.

Final performance metrics for the optimized control problem are shown in Table 16. Both FNO and LNO yielded solutions close to optimal.

**Table 16:** Optimized control performance metrics for the polynomial tracking benchmark.

| Architecture | Cost $J(\tilde{u})$ | Epochs | Relative Error of $\tilde{u}$ | Relative Error of $\tilde{x}$ |
|---|---|---|---|---|
| DeepONet | 0.148 | 10000 | 0.0506 | 0.0099 |
| FNO | 0.149 | 1000 | 0.0222 | 0.0020 |
| LNO | 0.149 | 700 | 0.0327 | 0.0071 |





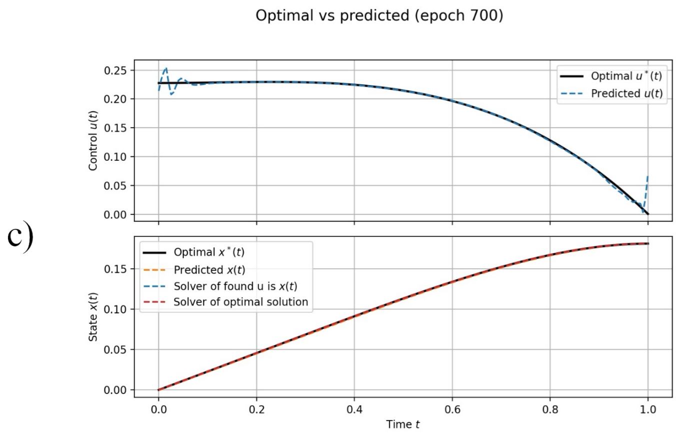


**Figure 18:** Predicted vs analytical control trajectories for the Polynomial Tracking benchmark: a) DeepONet, b) FNO, c) LNO

## 5.4 Problem: Nonlinear ODE

The neural operator architectures DeepONet, FNO, and LNO (see Section 4.5) were trained to model the dynamics of the nonlinear problem. DeepONet and FNO successfully learned the system dynamics to a sufficient degree, while LNO did not (see Figure 19).

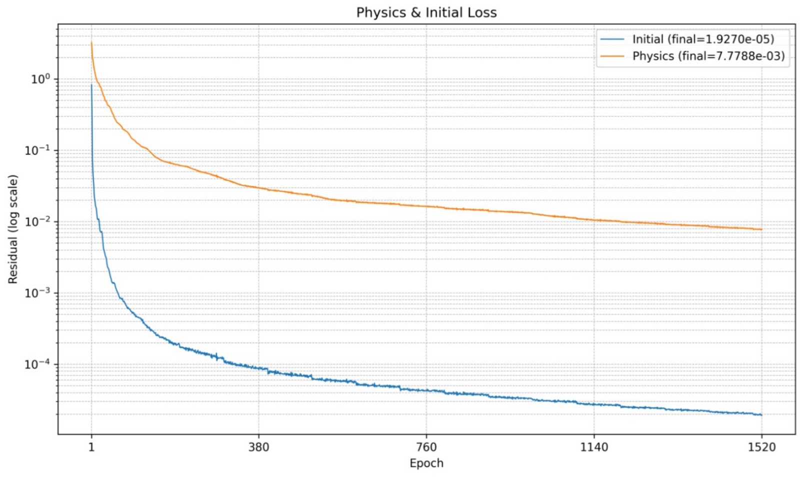
Table 17 summarizes the training performance. FNO achieved the lowest physics loss, likely due to the use of five Fourier layers instead of four. When preliminarily experimenting with its architecture, with four Fourier layers, the performance was similar to that of LNO.

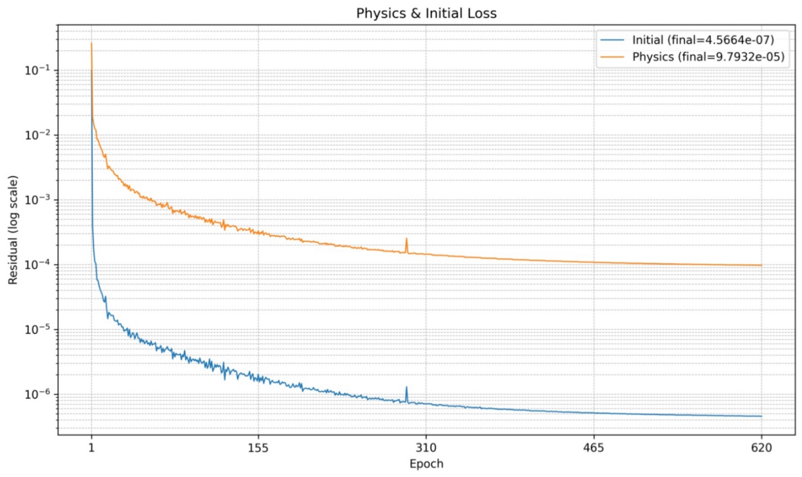
**Table 17:** Training statistics for the nonlinear ODE benchmark.

| Architecture | Training Time (min) | Epochs | Physics Loss | Initial Loss |
|---|---|---|---|---|
| DeepONet | 1308 | 1520 | $7.78 \cdot 10^{-3}$ | $1.93 \cdot 10^{-5}$ |
| FNO | 74 | 620 | $9.79 \cdot 10^{-5}$ | $4.57 \cdot 10^{-7}$ |
| LNO | 63 | 370 | $2.31 \cdot 10^{-2}$ | $2.10 \cdot 10^{-3}$ |

Figure 20 shows the relative trajectory errors on the training and validation sets. No signs of overfitting were observed. The validation error for LNO oscillates heavily, indicating instability. DeepONet had not fully converged by the end of training, and further training was infeasible due to prohibitive runtime.

Recall that LNO uses a Laplace layer, which was presented as an equivalent to four Fourier layers. We attempted to train a four-layer FNO, but its physics loss plateaued around 0.01–0.02, which was insufficient for solving the control problem. Increasing the number of Fourier layers to five significantly improved its performance. LNO similarly plateaued at this level, even after increasing expressivity and experimenting with two Laplace layers (which were not evaluated in the original paper [11]). In the end, we used two Laplace layers with eight trainable poles and residues, which yielded the best performance.

a)


b)


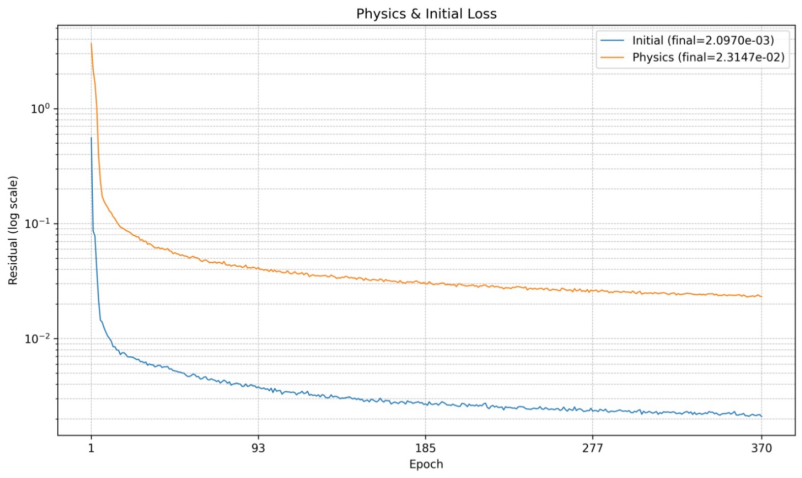
c)


**Figure 19:** Physics and initial condition loss convergence for the nonlinear ODE benchmark: a) DeepONet, b) FNO, c) LNO

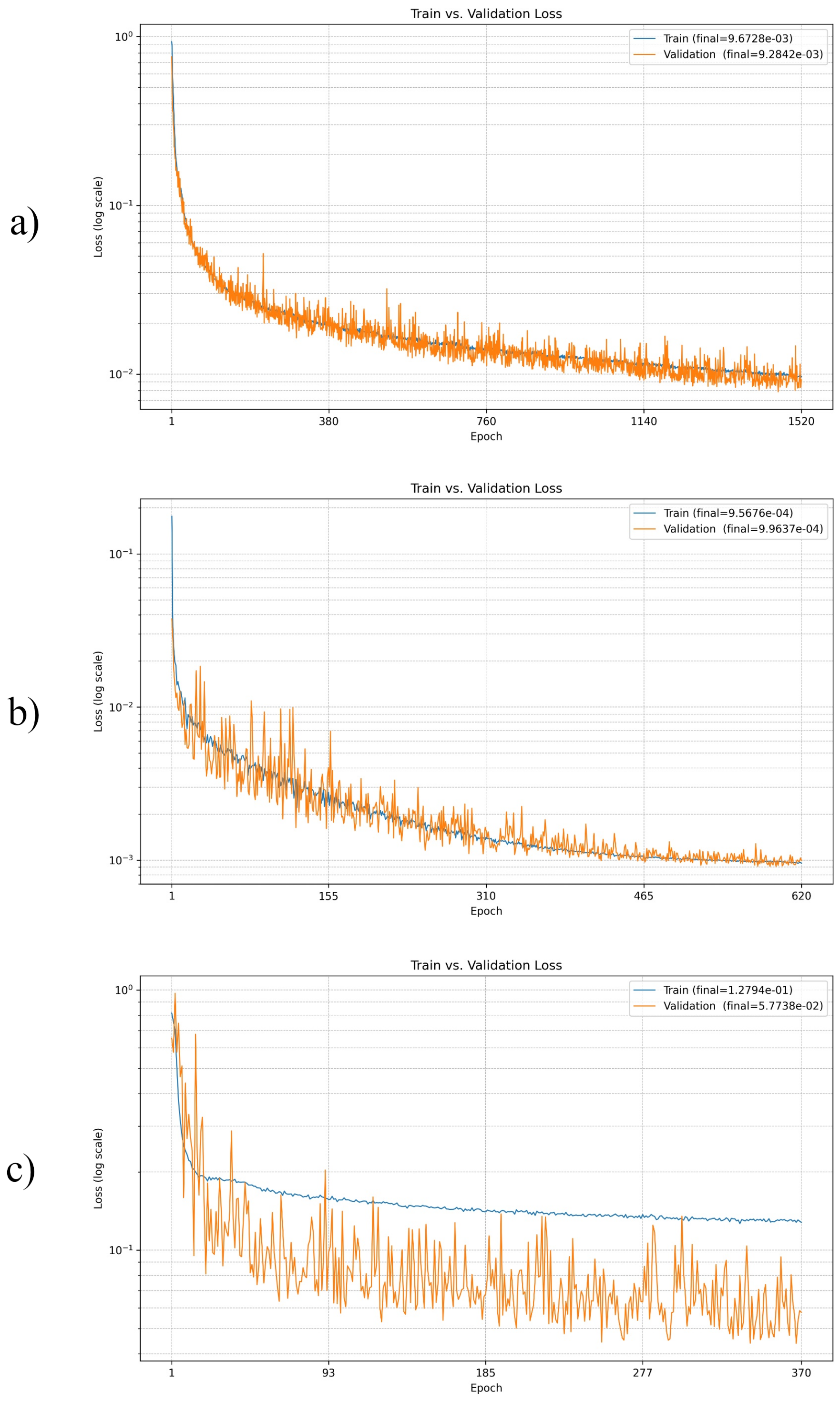


**Figure 20:** Relative trajectory errors (train/test) for the nonlinear ODE benchmark: a) DeepONet, b) FNO, c) LNO

Interestingly, the worst predictions by LNO on the test set are not completely inaccurate - they are generally only slightly off (see Figure 21).

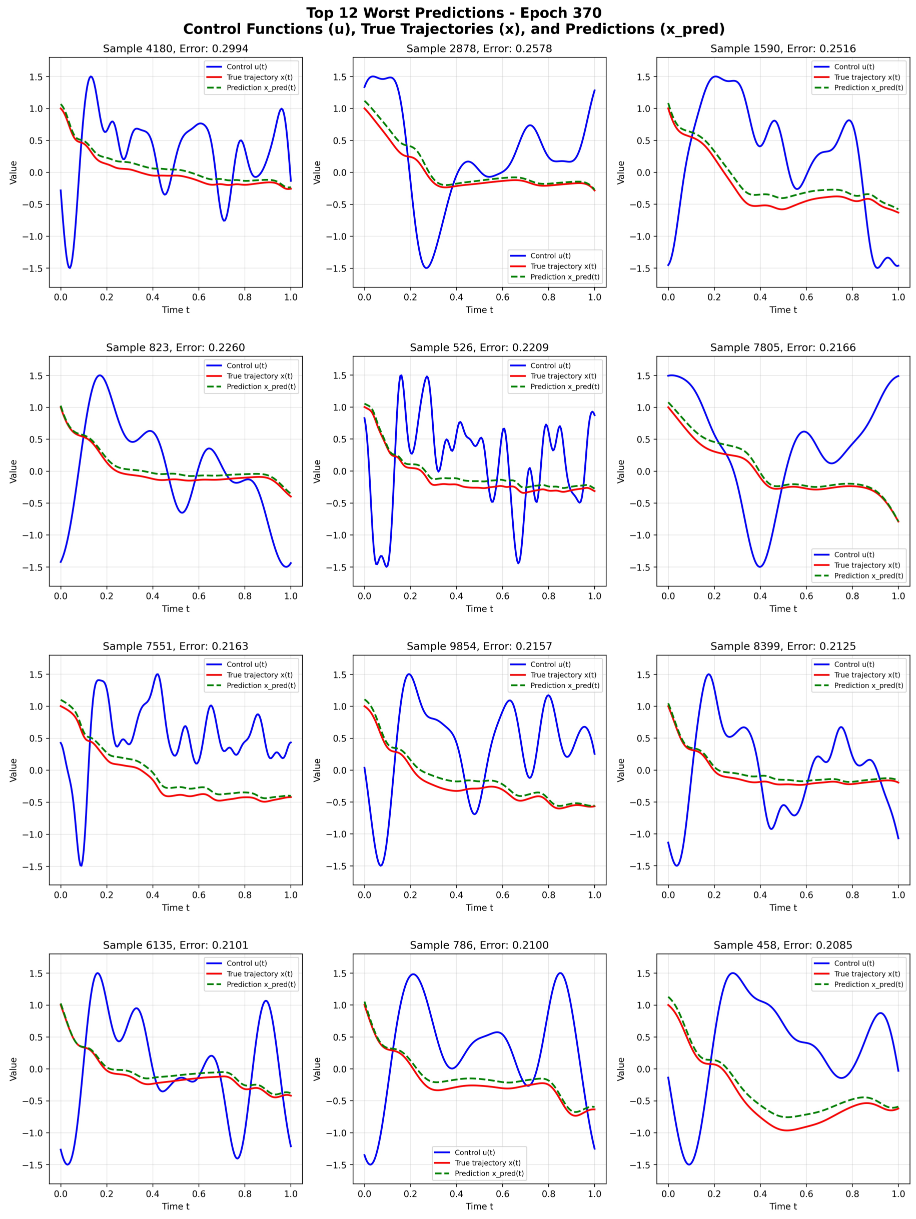


**Figure 21:** Worst-case validation predictions for LNO on the nonlinear ODE benchmark.

Table 18 summarizes the relative trajectory errors. FNO demonstrated the best generalization, followed by DeepONet. LNO performed significantly worse, with a substantial performance gap.

**Table 18:** Relative trajectory errors for the nonlinear ODE benchmark.

| **Architecture** | **Train Error $\mathcal{L}_2(\hat{x})$** | **Test Error $\mathcal{L}_2(\hat{x})$** |
|---|---|---|
| DeepONet | $9.67 \cdot 10^{-3}$ | $9.28 \cdot 10^{-3}$ |
| FNO | $9.56 \cdot 10^{-4}$ | $9.96 \cdot 10^{-4}$ |
| LNO | $1.28 \cdot 10^{-1}$ | $5.77 \cdot 10^{-2}$ |

We tried to fine-tune the trained LNO with available true trajectories, but this did not help to learn physics better and just enabled the model to memorize the trajectories and overfit the training set. For this particular framework (we have only 20% of data with labeled trajectories), we would not recommend a semi-supervised setting according to our experimental results.

The best-performing control optimization weights are listed in Table 19. LNO with any weight on physics loss could not converge to an optimal solution.

**Table 19:** Control optimization weights for the nonlinear ODE benchmark.

| **Architecture** | $\mu_{\text{phys}}$ | $\mu_{\text{obj}}$ | $\mu_{\text{init}}$ | $\mu_{\text{smooth}}$ | $\mu_{\text{bound}}$ |
|---|---|---|---|---|---|
| DeepONet | 150 | 1 | 1 | 1 | 0 |
| FNO | 5 | 1 | 1 | 0.1 | 0 |
| LNO | 100 | 1 | 1 | 1 | 0 |

Figure 22 shows the predicted control trajectories for DeepONet and FNO. FNO's control closely matches the analytical solution. DeepONet's trajectory is noisier but qualitatively accurate.

The candidate control predicted by LNO differs significantly from the analytical solution (Figure 22), which was expected since LNO failed to learn dynamics on noisy input families like GRF.

We hypothesize that LNO with one Laplace layer lacks sufficient capacity to model noisy, highly nonlinear input functions while simultaneously minimizing a nonlinear physics residual. Moreover, the exponential term in the Laplace layer, combined with numerical differentiation, may amplify prediction errors.

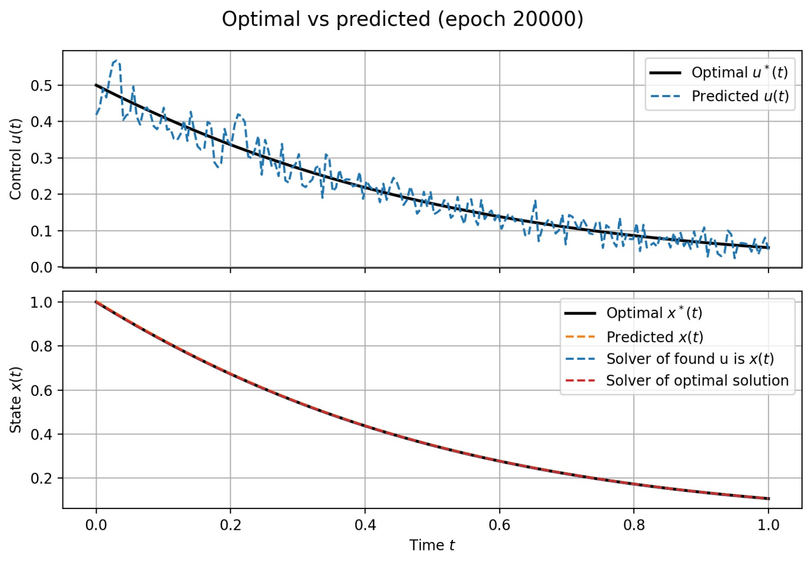
Final performance metrics for the optimized control solutions are shown in Table 20. LNO failed to converge to a feasible control trajectory.

**Table 20:** Optimized control quality metrics for the nonlinear ODE benchmark.

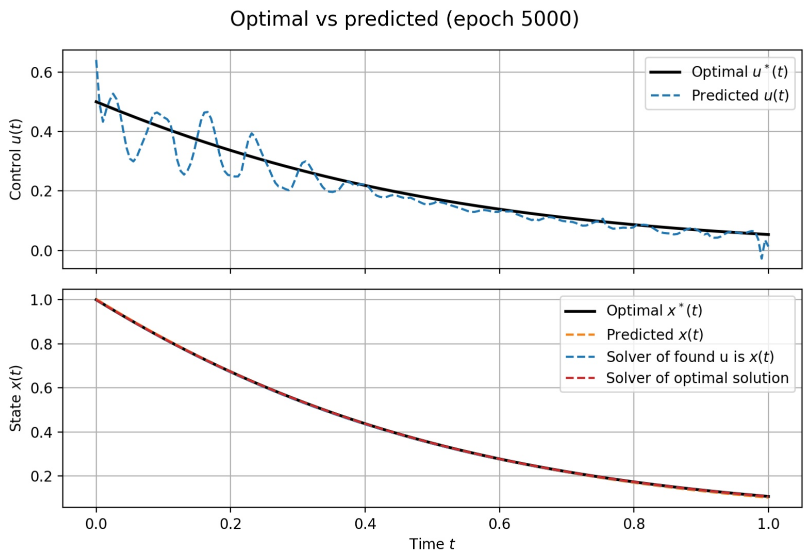
| **Architecture** | **Cost** $J(\tilde{u})$ | **Epochs** | **Relative Error of** $\tilde{u}$ | **Relative Error of** $\tilde{x}$ |
|---|---|---|---|---|
| DeepONet | -0.1070 | 20000 | 0.1362 | 0.0031 |
| FNO | -0.1011 | 5000 | 0.1745 | 0.0057 |
| LNO | -0.0282 | 10000 | 1.7671 | 0.2688 |

The most accurate solution was by DeepONet. But the quality of all found solutions degraded in comparison with solutions to other benchmark problems.

a)



b)



c)

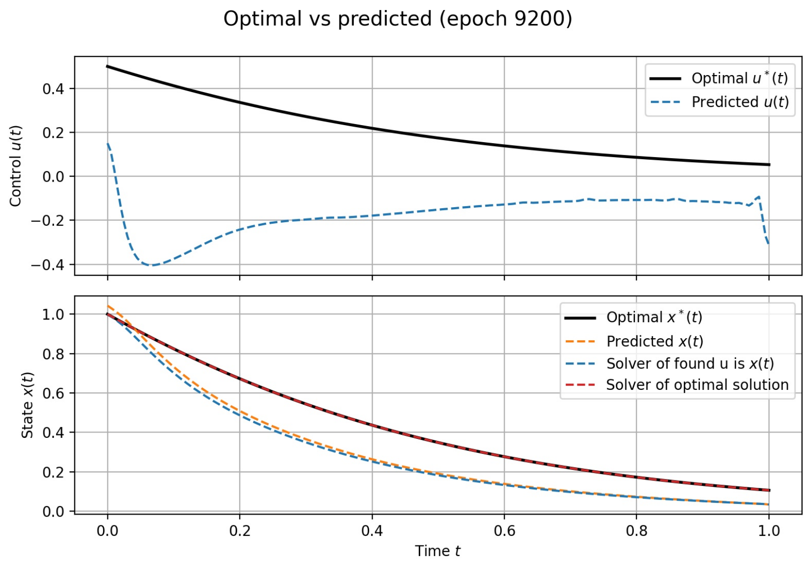


**Figure 22:** Control trajectory predictions for the nonlinear ODE benchmark: a) DeepONet, b) FNO, c) LNO

## 5.5 Problem: Singular Arc

This problem is interesting because, depending on the input control, the resulting trajectory can either grow exponentially, making it impossible for the numerical solver to find a solution, or have a slow downward trend.

We want our trajectory to satisfy both the initial and boundary conditions: $x(0) = 1$ and $x(1) = 0$. Ideally, a neural operator should approximate solutions to the differential equation within at least a specified range. We chose the input range $[-3, -0.5]$ for $u$. At the data generation step, whenever we were increasing the upper bound, the solver was failing for some of the inputs.

Just as the solver struggles with input controls that lead to exponentially growing trajectories, neural operators with selected architectures (see Section 4.5) also struggled to learn them. DeepONet and LNO failed to sufficiently learn the physics, while FNO managed to do so - but only after increasing its depth to 6 and significantly increasing both modes and width (see Figure 23).

Table 21 summarizes the training performance. Only FNO managed to learn the system's physics, while both DeepONet and LNO learned the physics somewhat poorly.

**Table 21:** Training statistics for the singular arc benchmark.

| Model | Training Time (min) | Epochs | Physics Loss | Initial Loss |
|---|---|---|---|---|
| DeepONet | 1249 | 1580 | $1.98 \cdot 10^{-2}$ | $1.57 \cdot 10^{-5}$ |
| FNO | 34 | 230 | $6.83 \cdot 10^{-4}$ | $1.41 \cdot 10^{-6}$ |
| LNO | 131 | 1000 | $1.12 \cdot 10^{-2}$ | $8.82 \cdot 10^{-5}$ |

Figure 24 shows the relative training and validation trajectory errors. No signs of overfitting were observed. Based on these errors, DeepONet had not yet converged. LNO also had potential for improvement.

Table 22 summarizes the training and validation relative errors of the predicted trajectories. In terms of relative error, DeepONet achieved the best performance. Interestingly, though FNO has the smallest physics loss, DeepONet predicted trajectories more accurately.

**Table 22:** Relative trajectory errors for the singular arc problem.

| Model | Train Error $\mathcal{L}_2(\hat{x})$ | Test Error $\mathcal{L}_2(\hat{x})$ |
|---|---|---|
| DeepONet | $8.99 \cdot 10^{-3}$ | $5.63 \cdot 10^{-3}$ |
| FNO | $1.33 \cdot 10^{-2}$ | $9.07 \cdot 10^{-3}$ |
| LNO | $8.02 \cdot 10^{-2}$ | $1.48 \cdot 10^{-2}$ |

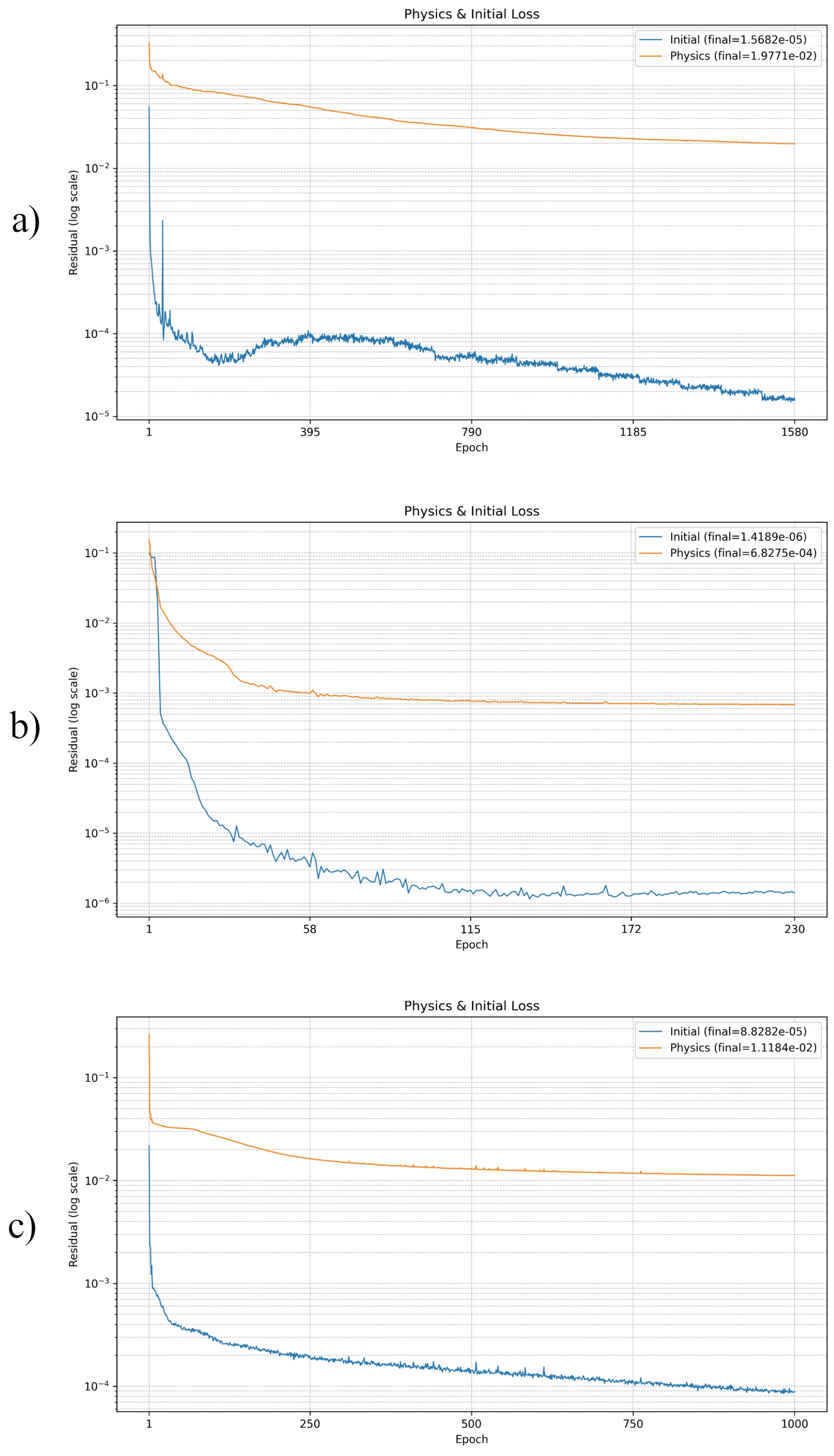
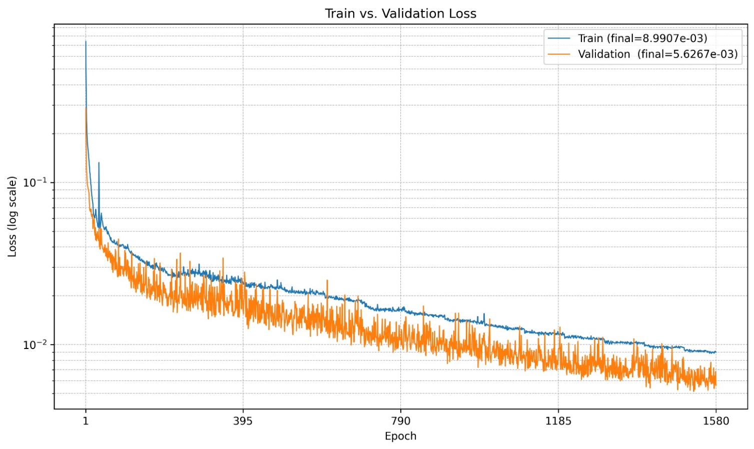
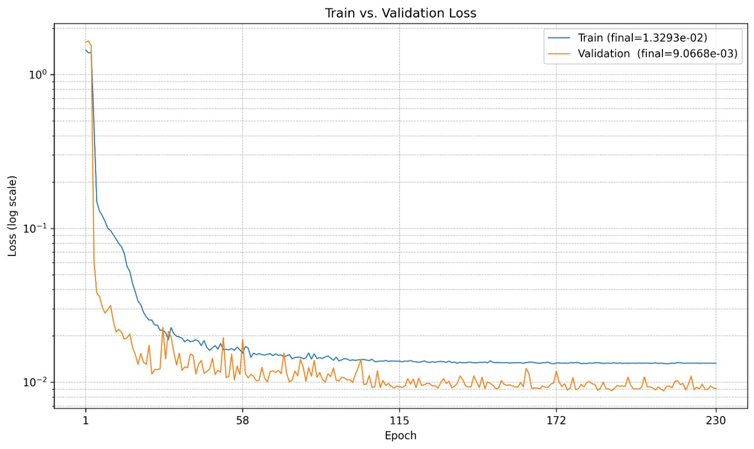


**Figure 23:** Physics and initial condition loss convergence for the singular arc benchmark: a) DeepONet, b) FNO, c) LNO

a)



b)



c)

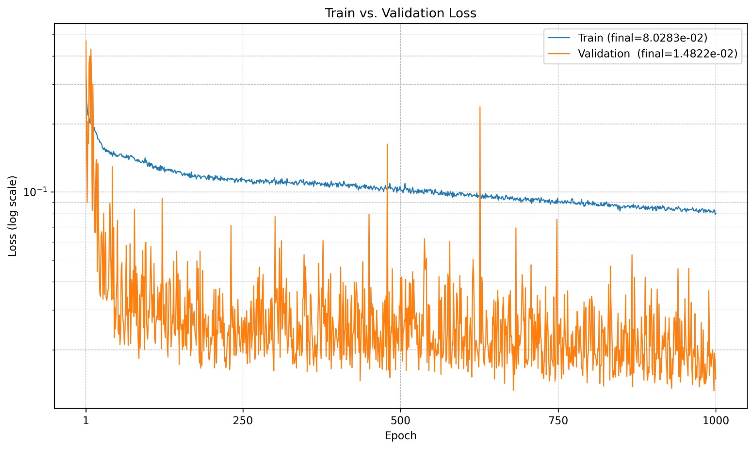
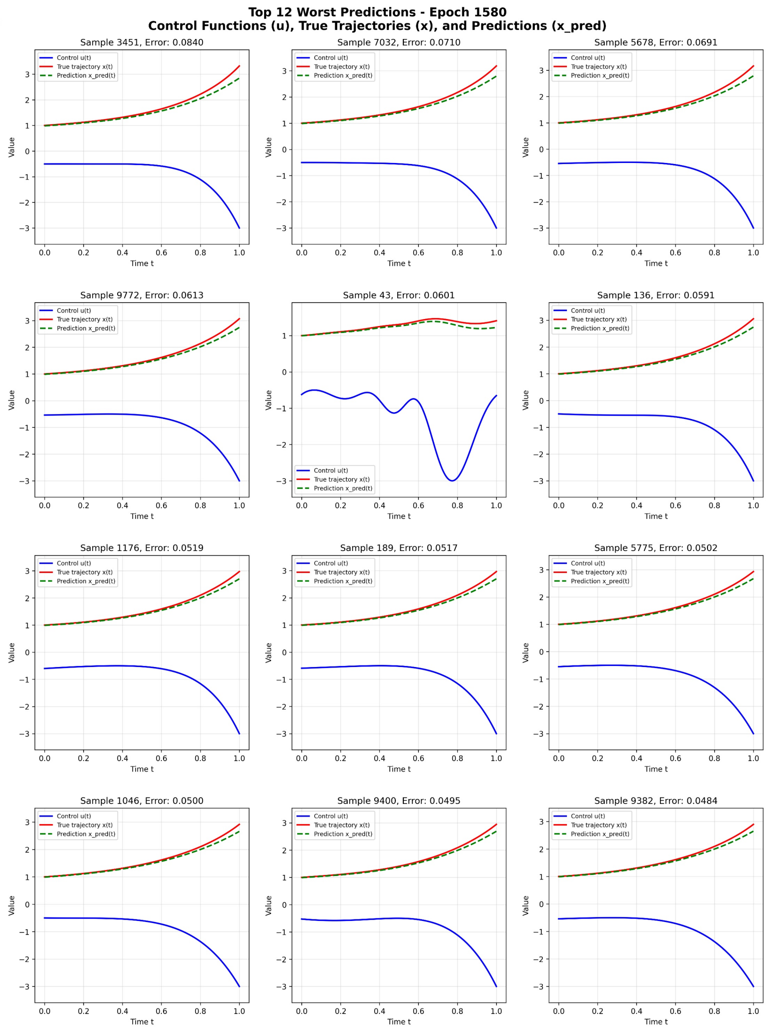


**Figure 24:** Relative trajectory errors (train/test) for the Singular Arc benchmark: a) DeepONet, b) FNO, c) LNO

There is a clear explanation for this. During training, models struggled most with trajectories like the one shown in Figure 25.

a)



b)

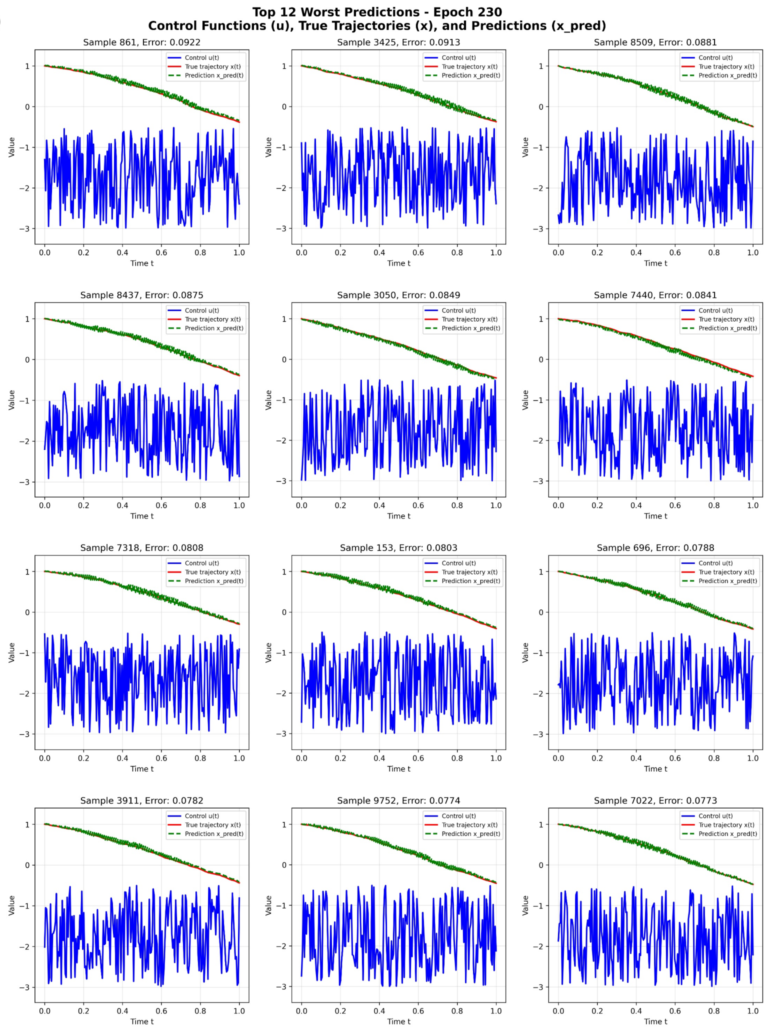


**Figure 25:** Worst predictions on the validation set for the Singular Arc problem of a) DeepONet, b)FNO

FNO also initially failed to predict these trajectories well, but eventually learned them at the cost of generating very noisy trajectories with noisy input controls. Thus, FNO learned the physics of the system but produced outputs that, while noisy, still followed the solution trend (see Figure 25). Thus, a small average physics loss in some cases does not lead to a low relative error of prediction on the validation set.

To give the LNO architecture a fair chance, we generated separate datasets specifically for it: one containing only polynomial functions, and another with a mixture of sine and polynomial functions. We used the sine activation function and kept the rest of the architecture unchanged. This configuration was chosen because authors in [11] trained the model exclusively on sine functions using this activation function. While the model was able to minimize the physics loss to $5.67 \cdot 10^{-5}$, it was impossible to use it in the optimization. Since initial and intermediate controls are noisy, LNO - which was trained only on smooth signals - can not generalize on them, and with every update, it diverges. The same problem happened with DeepONet when we were trying to solve the oscillatory forcing problem.

We then used the trained neural operators to solve the singular arc control problem (we used LNO trained on the original dataset). The best-performing control optimization weights are listed in Table 23.

**Table 23:** Control optimization weights for the Singular Arc problem.

| **Model** | $\mu_{\text{phys}}$ | $\mu_{\text{obj}}$ | $\mu_{\text{init}}$ | $\mu_{\text{smooth}}$ | $\mu_{\text{bound}}$ |
|---|---|---|---|---|---|
| DeepONet | 10 | 1 | 1 | 1 | 10 |
| FNO | 10 | 1 | 1 | 0 | 10 |
| LNO | 50 | 1 | 1 | 0 | 10 |

Figure 26 shows the predicted control trajectories. Among the models, FNO produced the closest match to the analytical control but failed to satisfy the boundary condition.

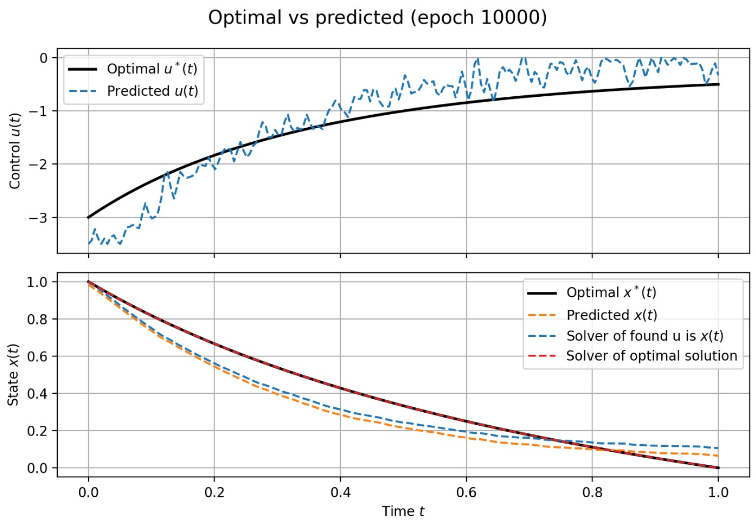
Interestingly, the control predicted by LNO closely resembles that of DeepONet but is less noisy. Even their predicted trajectories are similar, both violating the boundary condition in the same way. Recall that the physics loss of LNO and DeepONet were also comparable.

Final performance metrics for the optimized controls are shown in Table 24.

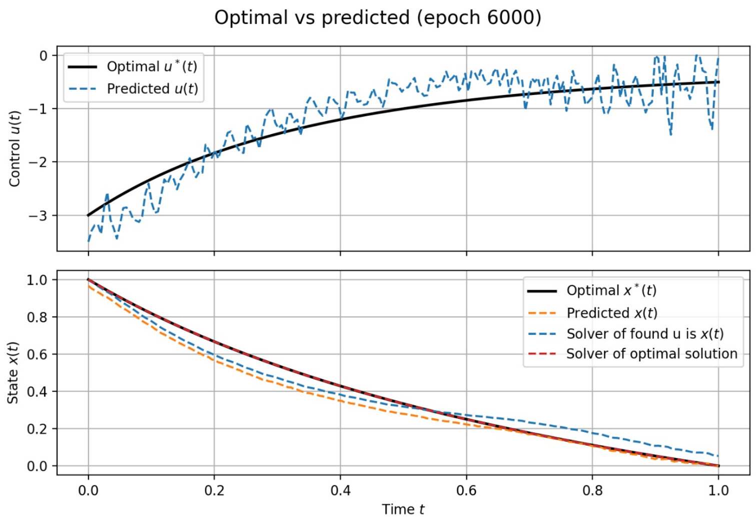
**Table 24:** Optimized control quality metrics for the Singular Arc problem.

| **Model** | **Cost** $J(\tilde{u})$ | **Epochs** | **Relative Error of** $\tilde{u}$ | **Relative Error of** $\tilde{x}$ |
|---|---|---|---|---|
| DeepONet | 2.173 | 10000 | 0.2873 | 0.1978 |
| FNO | 2.078 | 6000 | 0.2517 | 0.1241 |
| LNO | 2.183 | 5000 | 0.2550 | 0.1863 |

a)



b)



c)

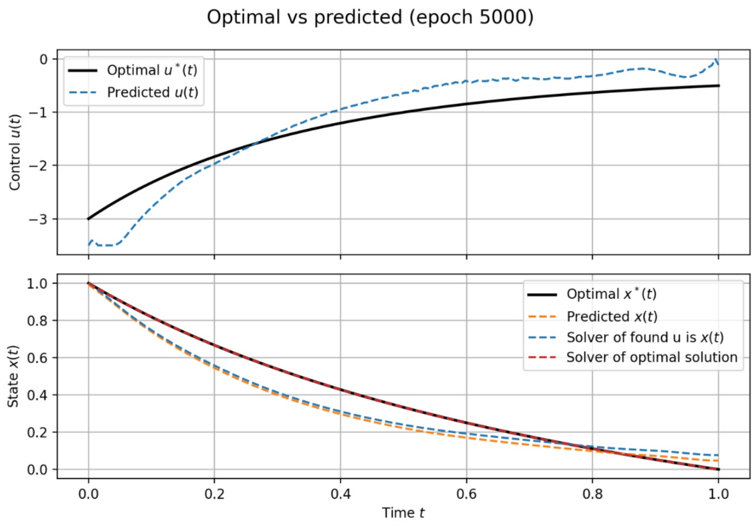


**Figure 26:** Predicted vs analytical control trajectories for the Singular Arc benchmark: a) DeepONet, b) FNO, c) LNO

## 5.6 Neural-operator performance and a classical IPOPT baseline

Table 25 reports all key metrics across models and problems. For each architecture–problem pair, we show: (i) total training time in GPU-minutes[2], (ii) final physics-residual loss, (iii) validation error on the predicted state trajectory, and (iv) relative $\mathcal{L}_2$-error of the optimized control. The best value per metric within each benchmark row is highlighted in bold.

**Table 25:** Unified performance comparison across problems and architectures. Bold indicates best performance per metric within each problem.

| Problem | Model | Train Time (min) | Physics Loss | Val. Error | Rel. Err. ($\tilde{u}$) |
|---|---|---|---|---|---|
| Linear ODE | DeepONet | 1267 | $9.51 \cdot 10^{-4}$ | $\mathbf{6.14 \cdot 10^{-4}}$ | $1.91 \cdot 10^{-2}$ |
|  | FNO | **118** | $\mathbf{3.70 \cdot 10^{-4}}$ | $2.67 \cdot 10^{-3}$ | $3.48 \cdot 10^{-2}$ |
|  | LNO | 209 | $3.25 \cdot 10^{-4}$ | $1.09 \cdot 10^{-3}$ | $\mathbf{1.42 \cdot 10^{-2}}$ |
| Oscillatory | DeepONet | 1236 | $3.49 \cdot 10^{-4}$ | $6.49 \cdot 10^{-3}$ | $4.54 \cdot 10^{0}$ |
|  | FNO | **60** | $\mathbf{1.18 \cdot 10^{-4}}$ | $2.78 \cdot 10^{-3}$ | $\mathbf{1.28 \cdot 10^{0}}$ |
|  | LNO | 67 | $4.23 \cdot 10^{-4}$ | $\mathbf{1.75 \cdot 10^{-3}}$ | $1.30 \cdot 10^{0}$ |
| Polynomial | DeepONet | 1251 | $3.65 \cdot 10^{-4}$ | $2.06 \cdot 10^{-2}$ | $5.06 \cdot 10^{-2}$ |
|  | FNO | **72** | $\mathbf{1.11 \cdot 10^{-4}}$ | $\mathbf{4.96 \cdot 10^{-3}}$ | $\mathbf{2.22 \cdot 10^{-2}}$ |
|  | LNO | 79 | $2.27 \cdot 10^{-4}$ | $1.27 \cdot 10^{-2}$ | $3.27 \cdot 10^{-2}$ |
| Nonlinear | DeepONet | 1308 | $7.78 \cdot 10^{-3}$ | $9.28 \cdot 10^{-3}$ | $\mathbf{1.36 \cdot 10^{-1}}$ |
|  | FNO | 74 | $\mathbf{9.79 \cdot 10^{-5}}$ | $\mathbf{9.96 \cdot 10^{-4}}$ | $1.75 \cdot 10^{-1}$ |
|  | LNO | **63** | $2.31 \cdot 10^{-2}$ | $5.77 \cdot 10^{-2}$ | $1.81 \cdot 10^{0}$ |
| Singular Arc | DeepONet | 1249 | $1.98 \cdot 10^{-2}$ | $\mathbf{5.63 \cdot 10^{-3}}$ | $2.87 \cdot 10^{-1}$ |
|  | FNO | **34** | $\mathbf{6.83 \cdot 10^{-4}}$ | $9.07 \cdot 10^{-3}$ | $\mathbf{2.52 \cdot 10^{-1}}$ |
|  | LNO | 131 | $1.12 \cdot 10^{-2}$ | $1.48 \cdot 10^{-2}$ | $2.55 \cdot 10^{-1}$ |

Across all models, FNO was the most successful in minimizing physics loss with the minimum time of training. For LNO, the most challenging problem was the nonlinear ODE, and for DeepONet, the oscillatory forcing. Though physics loss of trained DeepONet and LNO was large, they managed to predict almost the same solution as FNO.

Table 26 summarises the performance of a classical IPOPT direct-collocation solver on the same $N = 200$ grid used throughout this work. For each benchmark we list (i) the relative end-point errors in state and control, (ii) the final objective value $J^*$, and (iii) the CPU wall-clock time. These figures provide a concrete reference for the best accuracy and runtime attainable without a neural surrogate; the subsequent sections compare every PINO result against this baseline.

[2]Epoch time excludes validation and plotting.

**Table 26:** Classical IPOPT baseline ($N$=200). Relative errors are $\|x-x^*\|/\|x^*\|$ and $\|u-u^*\|/\|u^*\|$ (lower is better).

| Benchmark | Rel. Err. $x$ | Rel. Err. $u$ | Cost $J^*$ | Time [s] |
|---|---|---|---|---|
| Linear ODE | $1.21 \cdot 10^{-3}$ | $3.51 \cdot 10^{-3}$ | 0.193679 | 0.234 |
| Oscillatory forcing | $5.18 \cdot 10^{-2}$ | $1.76 \cdot 10^{-1}$ | 0.001553 | 0.018 |
| Polynomial tracking | $2.88 \cdot 10^{-3}$ | $2.16 \times 10^{-3}$ | 0.150206 | 0.016 |
| Nonlinear ODE | $5.17 \cdot 10^{-4}$ | $1.54 \cdot 10^{-2}$ | $-0.106544$ | 0.038 |
| Singular Arc | $1.98 \cdot 10^{-1}$ | $2.08 \cdot 10^{-1}$ | 1.863697 | 0.026 |

Compared to IPOPT, all neural operator models exhibit higher control and trajectory errors, with relative $\mathcal{L}_2$-errors often one to two orders of magnitude above the baseline. Runtime is also substantially greater - training takes minutes to hours, while IPOPT solves each instance in milliseconds. However, this cost is amortized: once trained, a neural operator can generalize across inputs without re-solving the OCP. In problems like the nonlinear and polynomial benchmarks, FNO achieved control accuracy within a factor of two from IPOPT, suggesting that physics-informed training can yield competitive surrogates when dynamics are smooth and well-resolved. As neural operator architectures continue to improve, and as better techniques for dataset generation and regularization emerge, their performance on more complex tasks is likely to close the gap. Importantly, a single trained model can be reused for fast inference on unseen inputs, making this approach promising for settings where repeated solves are required.

## 5.7 Sensitivity analysis

We conducted a sensitivity analysis with respect to the control cost weight $\rho \in \{0.1, 1, 10\}$, averaging results over three independent random seeds. Table 27 summarizes the relative error of the found control $\hat{u}$ (mean $\pm$ standard deviation) across all benchmark problems and architectures. Our goal was to see how the ratio of the physics loss weight and control objective affects the quality of the produced solutions.

The sensitivity analysis reveals clear trends in robustness across architectures. FNO is the most stable and generalizes well across all $\rho$. Its performance on the oscillatory forcing, nonlinear, and polynomial tracking problems is consistently strong. DeepONet performs well at moderate weights ($\rho = 1$), particularly in the linear, polynomial, and singular arc cases, but suffers at extreme values, suggesting that it needs the weight of physics loss to be much higher than that of the control objective to produce a feasible solution. LNO improves with higher $\rho$ on certain problems (e.g., polynomial and singular arc), but fails to generalize in the nonlinear setting, regardless of weight (recall that it has not managed to learn the physics of the system properly). Overall, FNO emerges as the most robust choice, while DeepONet and LNO display problem-specific sensitivity.

**Table 27:** Pilot sensitivity to control-physics ratio $\rho$. Values are mean $\pm$ std input control relative error $\|\hat{u} - u^*\|/\|u^*\|$ over three seeds (lower is better).

|  | Rho | $\rho = 0.1$ | $\rho = 1$ | $\rho = 10$ |
|---|---|---|---|---|
| Benchmark | Architecture |  |  |  |
| Linear ODE | DeepOnet | 6.170e-02 ± 3.6e-05 | 1.912e-02 ± 8.5e-06 | 5.306e-02 ± 4.9e-07 |
|  | FNO | 9.929e-02 ± 1.3e-04 | 3.667e-02 ± 4.3e-05 | 8.659e-02 ± 8.6e-06 |
|  | LNO | **4.541e-02 ± 2.5e-04** | **1.419e-02 ± 7.8e-05** | **2.193e-02 ± 2.8e-06** |
| Nonlinear ODE | DeepOnet | **1.267e-01 ± 2.1e-02** | **1.325e-01 ± 3.1e-03** | 3.124e-01 ± 8.9e-03 |
|  | FNO | 1.817e-01 ± 9.8e-06 | 1.816e-01 ± 1.2e-05 | 1.813e-01 ± 1.1e-05 |
|  | LNO | 2.044e+00 ± 2.3e-01 | 1.767e+00 ± 3.8e-05 | **1.493e+00 ± 3.3e-05** |
| Oscillatory forcing | DeepOnet | 4.326e+02 ± 1.8e+01 | 2.347e+02 ± 7.9e+00 | 9.419e+01 ± 5.4e+00 |
|  | FNO | **3.062e+00 ± 4.7e-03** | **1.277e+00 ± 1.1e-03** | 1.113e+00 ± 1.3e-05 |
|  | LNO | 3.412e+00 ± 6.5e-03 | 1.301e+00 ± 3.3e-04 | **1.102e+00 ± 1.5e-04** |
| Polynomial tracking | DeepOnet | 7.064e-02 ± 3.2e-05 | 5.053e-02 ± 1.3e-05 | 1.506e-01 ± 1.3e-06 |
|  | FNO | **4.849e-02 ± 1.1e-04** | **2.217e-02 ± 8.4e-05** | 4.925e-02 ± 6.8e-06 |
|  | LNO | 6.815e-02 ± 2.6e-04 | 3.287e-02 ± 2.5e-05 | **2.593e-02 ± 1.1e-05** |
| Singular arc | DeepOnet | 4.890e-01 ± 1.3e-07 | 2.873e-01 ± 1.2e-06 | 4.157e-01 ± 1.7e-08 |
|  | FNO | 5.018e-01 ± 3.4e-02 | **2.477e-01 ± 8.3e-03** | **3.172e-01 ± 4.2e-03** |
|  | LNO | **4.790e-01 ± 8.2e-06** | 2.565e-01 ± 1.0e-06 | 3.254e-01 ± 8.2e-07 |

## 5.8 Benchmarking PDE Problems

We additionally evaluated FNO on more challenging control problems governed by partial differential equations (PDEs) to demonstrate that neural operators can be applied to real-world nonlinear systems, not only to “toy“ examples. In this section, we present the resulting input-control solutions together with basic statistics on training time and performance. A more thorough study of FNO robustness and alternative models is left for future work.

Recall that the goal is to determine an input signal that drives the system from the initial state $y(x, 0) = 0$ to the desired target state while simultaneously minimizing control effort. The weights used for the individual loss terms during control optimization are given in Table 28.

**Table 28:** Loss weights used for each control problem.

| **Problem** | $\mu_{\text{obj}}$ | $\mu_{\text{phys}}$ | $\mu_{\text{bound}}$ | $\mu_{\text{init}}$ |
|---|---|---|---|---|
| Heat | 10 | 1 | 0.01 | 0.01 |
| Diffusion–Reaction | 1 | 1 | 0.01 | 0.01 |
| Burgers | 10 | 5 | 0.01 | 0.01 |

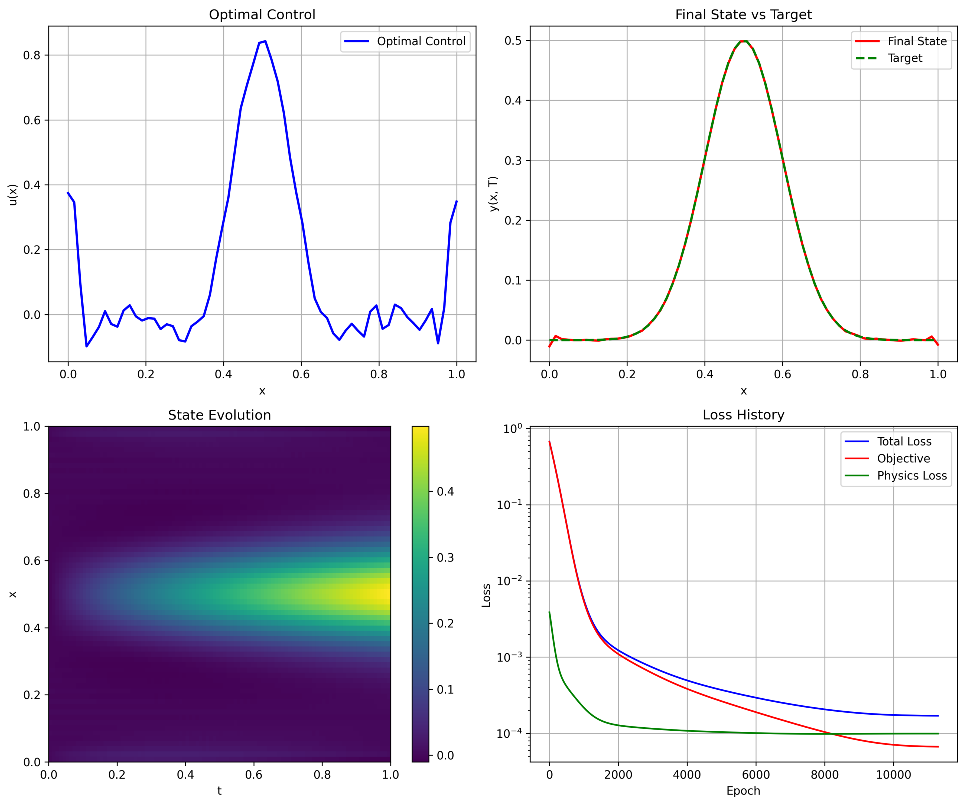


**Figure 27:** Heat (1D): candidate input control $u(x)$, final state $y(x, T)$, and target profile.

The solution produced by FNO for the heat equation is somewhat noisy near the boundaries (Figure 27), possibly due to inaccurate gradient approximations, yet it still drives the system close to the target state. For the diffusion–reaction problem, FNO achieved a highly accurate solution (Figure 28). The only shortcoming is that the peaks of the final state slightly deviate, which may reflect a lack of sufficiently high-amplitude input signals in the training set.

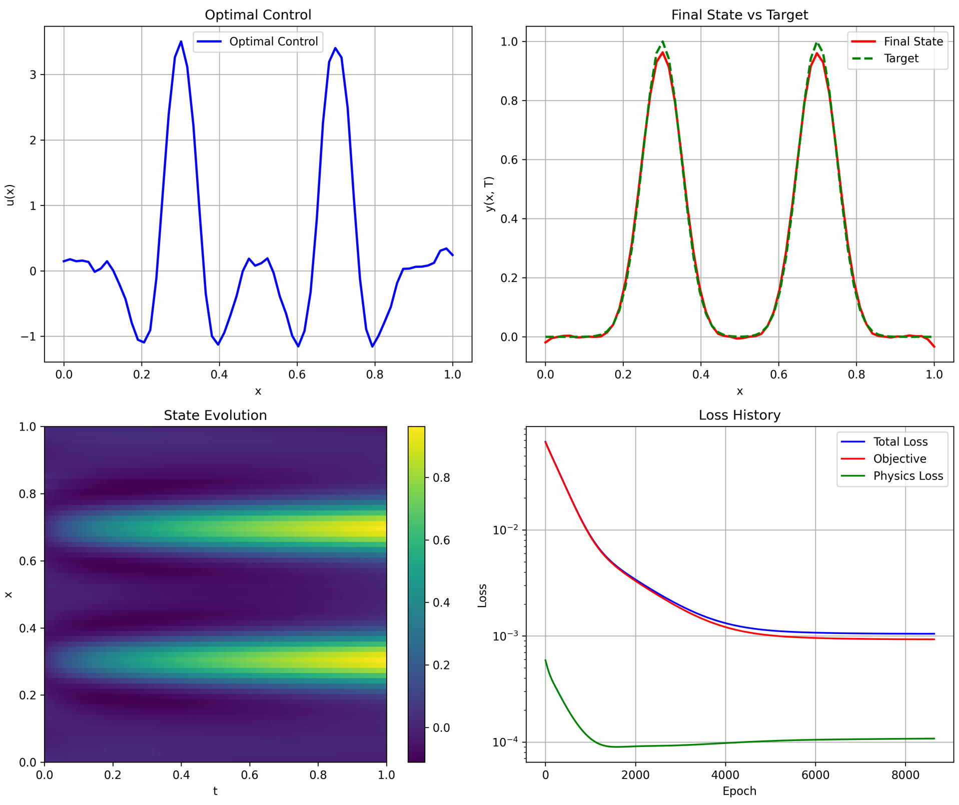


**Figure 28:** Diffusion–reaction (1D): candidate input control $u(x)$, final state $y(x, T)$, and target profile.

The solution for the 1-D Burgers equation is acceptable but somewhat noisy (Figure 29). This is likely because the model was trained solely on physics residuals. Nevertheless, FNO learned this demanding task without labeled trajectories, highlighting the promise of neural-operator architectures for real-world applications.

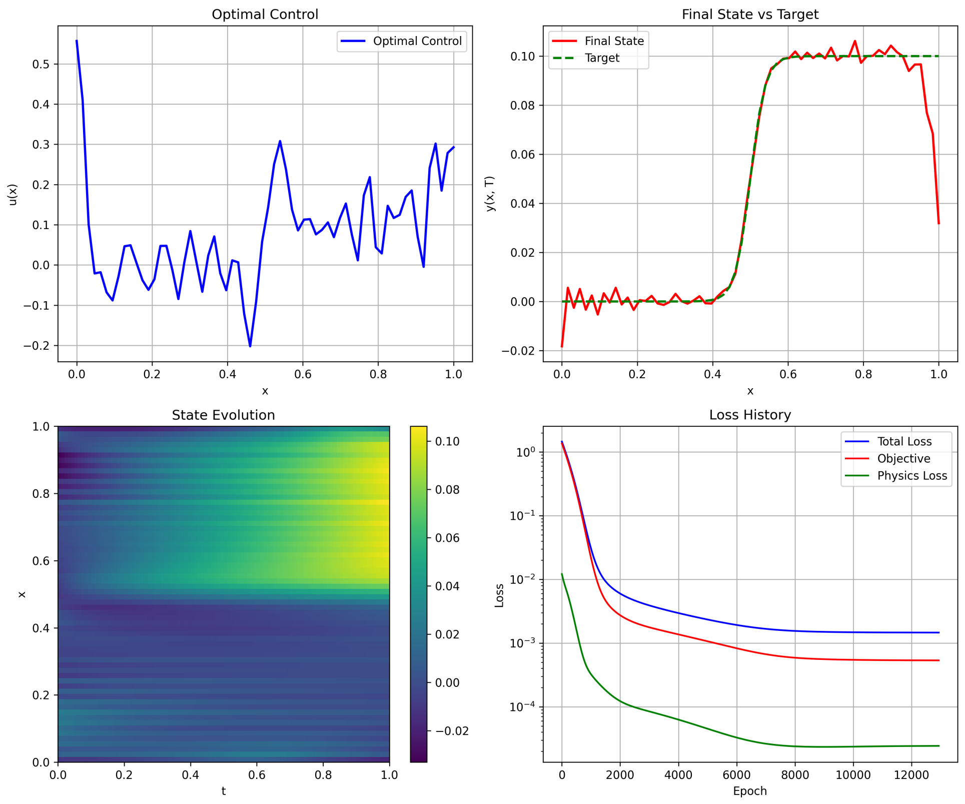


**Figure 29:** Burgers (1D): candidate input control $u(x)$, final state $y(x,T)$, and target profile.

Table 29 summarizes FNO training times and physics-loss values for each problem (validation loss is unavailable for Burgers because that model was trained unsupervised).

**Table 29:** Training performance of FNO on the PDE benchmarks.

| **Problem** | **Training Time (s)** | **Train $\mathcal{L}_{\text{phys}}$** | **Test $\mathcal{L}_{\text{phys}}$** |
|---|---|---|---|
| Heat | 145.46 | $5.98 \times 10^{-3} \pm 1.20 \times 10^{-3}$ | $6.63 \times 10^{-3} \pm 1.05 \times 10^{-3}$ |
| Diffusion–Reaction | 132.60 | $1.50 \times 10^{-3} \pm 2.34 \times 10^{-4}$ | $1.64 \times 10^{-3} \pm 2.58 \times 10^{-4}$ |
| Burgers | 505.89 | $7.90 \times 10^{-4} \pm 7.73 \times 10^{-5}$ | — |

Burgers required the longest training time because the dataset contained ten times more functions. However, training was still fast because no data-loss gradients had to be computed. We observed no signs of overfitting, and FNO reached a suitably small physics loss surprisingly quickly, suggesting that learning physically meaningful dynamics is easier than learning the toy-problem dynamics.

**Table 30:** Comparison of FNO-based optimization with a reference solver.

| **Problem** | **FNO Time (s)** | $\mathcal{L}_{\text{phys}}$ | FNO $J(u)$ | Solver $J^*(u)$ | **Solver Time (s)** |
|---|---|---|---|---|---|
| Heat | 44.46 | $9.92 \times 10^{-5}$ | $4.84 \times 10^{-5}$ | $1.09 \times 10^{-2}$ | 0.15 |
| Diffusion–Reaction | 36.89 | $1.08 \times 10^{-4}$ | $9.30 \times 10^{-4}$ | $6.40 \times 10^{-2}$ | 0.14 |
| Burgers | 57.05 | $2.44 \times 10^{-5}$ | $6.54 \times 10^{-5}$ | $2.80 \times 10^{-4}$ | 0.53 |

We used a traditional solver to find reference solutions to these benchmark problems in the same way as we did with toy problems 5.6. We formulated the problem in CasADi and solved it accordingly (the source code is available in our *GitHub* [44]). FNO outperformed the reference solver (Table 30), achieving a significantly lower objective value while preserving physical validity. Moreover, an FNO model trained on this problem can be reused with a different objective function, although careful tuning of the loss-component weights remains necessary.

## 6 Discussion

This section summarizes key findings and limitations of applying physics-informed neural operators to optimal control. We discuss (i) how different architectures adapt to boundary and initial conditions, (ii) the role of operator design in convergence and solution accuracy, (iii) limitations of neural operators as control surrogates, and (iv) the feasibility of training with only physics residuals. First, we compare architectures on the benchmark problems, and then we discuss FNO performance on PDEs.

In our framework, we enforce only *initial conditions* during training of the neural operator. This is done by including a soft penalty on the predicted state at $t = 0$, which is feasible for any input function. Enforcing boundary or terminal conditions during training would require us to pre-select only those input functions $u(t)$ that lead to trajectories ending at the correct terminal state. This would involve solving the differential equation for every input in the dataset, which is nearly equivalent to solving all possible equations and selecting the best. Hence, boundary conditions are deferred to the control-optimization stage, where they are added as a penalty in the optimization loss.

However, this design introduces limitations. In the singular-arc problem, many input signals led to trajectories that violated the terminal constraint, often growing exponentially. Once optimization enters such regions, it becomes hard to recover, and the loss fails to converge. While the physics-informed loss can still be minimized, the resulting controls may not produce feasible trajectories under the required terminal constraint. These particular signals were also hard for the neural operators to learn during training.

Architectural choices significantly affect convergence, physics loss, and the quality of the control solution. FNO had the highest capacity to fit the residual loss, achieving extremely low values (e.g. $9.79 \times 10^{-5}$) in some problems. However, this did not always translate into better control performance. In the singular-arc case, the FNO solution was as inaccurate as those from DeepONet and LNO, showing that minimizing residuals is not sufficient on its own.

DeepONet consistently enforced initial conditions better, thanks to its pointwise architecture and compatibility with automatic differentiation. However, its predictions were often noisy, and it failed to capture complex periodic dynamics, as seen in the oscillatory forcing problem. Its random initialization was more accurate at first, but with each iteration, it became worse due to its inability to generalize to such low-amplitude periodic signals. Though FNO and LNO were trained on the same dataset, they managed to find a nearly optimal solution.

LNO produced the smoothest trajectories across all problems and followed trends reliably even in difficult cases. However, it lacked the capacity to handle noisy inputs and nonlinear dynamics simultaneously. It worked well on smooth or linear problems, but it failed when both the inputs and the dynamics were complex. One potential explanation is the exponentiation step in LNO, which, combined with noisy gradient approximations (from finite differences), may amplify errors and prevent convergence.

Based on our experiments, we outline when each model is most appropriate:

- **DeepONet** is suitable when the goal is interpretability and generalization across uniform grids. It performs best when enforcing initial and terminal conditions is critical, and it serves as a reliable baseline.
- **FNO** is well-suited for fast training, periodic signals, and complex physics. Its flexibility in terms of modes and depth enables expressive modeling, though its predictions remain oscillatory.
- **LNO** is best when smooth trajectories are required. Its performance degrades under high input noise combined with nonlinear dynamics, making it less suitable for complex control problems.

We compared neural-operator-based control solutions to a traditional direct optimization method using IPOPT. IPOPT consistently achieved the lowest possible control cost and produced the most accurate trajectories across all toy benchmark problems. Neural operators frequently underperformed compared to IPOPT even when the residual loss was low. In particular, DeepONet and FNO occasionally produced highly oscillatory or noisy control signals, which would be unacceptable in precision-critical settings. Only LNO approached IPOPT-level smoothness in its output.

FNO demonstrated the most consistent behavior across all tested values of $\rho \in \{0.1, 1, 10\}$, maintaining a low relative error of the input signal $u$ (based on the physics loss) while successfully minimizing the control objective. This suggests that the dynamics learned by FNO were more stable and generalizable across control trade-offs. DeepONet was most effective at moderate cost weights ($\rho = 1$) but showed significant degradation when $\rho$ was either too small or too large. LNO's behaviour was problem-dependent: it improved on some problems at high $\rho$ but consistently failed to generalize on nonlinear dynamics, regardless of $\rho$. This highlights that physics-informed control performance is not only architecture-dependent but also highly sensitive to even single-parameter changes.

We observed that Algorithm 1 is highly sensitive to the initial guess of the control input. During each iteration, the control is updated and may take on a noisy shape that lies outside the distribution of functions learned by the neural operator. Since neural operators generalize by interpolating between functions seen during training, they may struggle to produce accurate predictions for such unseen or irregular inputs. Therefore, the training dataset must include both smooth and noisy control functions. If the neural operator fails to learn across this spectrum, the control-optimization process may diverge or converge to a suboptimal solution, as the model cannot reliably predict the system response for off-distribution controls.

The use of neural operators for control introduces several critical limitations. Control applications require precise gradient information. Models that rely on approximate gradients (such as FNO and LNO) risk learning consistent errors that compound during optimization. This can lead to poor or unsafe control trajectories, even if the residual loss appears low.

The approach is highly sensitive to the input-function family and dataset size. When training on noisy signals with limited data, the model may overfit edge cases and generalize poorly. We observed that models with larger residual losses could still outperform others on the actual control task. For example, LNO had a higher residual loss but produced better solutions than FNO on the singular-arc problem.

Results are extremely sensitive to the weights of the loss components. Small changes in the physics-loss or boundary-loss weights, or the random seed, caused large variability in results. This makes the method fragile and dependent on manual tuning. Often, the only way to select appropriate weights was to visualize the output trajectories and check their plausibility.

Finally, while neural operators offer fast inference and potential reusability, they must be validated rigorously before use in safety-critical systems. The learned dynamics are not guaranteed to be accurate, and optimization over poor dynamics may yield misleading results.

A central question in this work is whether neural operators can be trained using only physics residuals, without any trajectory supervision. Our experiments show that this is possible, but success depends heavily on the problem, the dataset, and the architecture. Residual-only training works better when the dynamics are relatively smooth, the dataset is sufficiently diverse, and the model has appropriate capacity.

It was surprising how much easier it was for FNO to solve systems constrained by PDEs than a comparatively simple ODE. This led us to think that neural operators perform better on physically meaningful systems. Solutions produced by FNO outperformed the traditional solver at the cost of short training times (about 10 minutes to train Burgers) and at most about 1 minute to converge to a solution (recall that we use early stopping with patience of 500 epochs). From our perspective, it has significant potential in more difficult scenarios. The following steps include developing a systematic method to generate meaningful time-variant input signals and extending it at least to 2-D versions of neural-operator architectures. We tried only FNO-1D and disregarded LNO-1D because it failed to learn these dynamics; we suspect there may be an issue in the official code implementation (the time grid is initialized internally) or errors in the example scripts [44]. In [11], they used the 2-D Burgers equation as a benchmark, but with LNO-2D, which is a very different variant and was trained in a supervised manner only on sine signals.

Several open directions remain:

- Conduct a systematic sensitivity analysis across hyperparameters, datasets, and control-problem performance.
- Evaluate newer neural-operator architectures that may combine expressivity with better smoothness or generalization (e.g., transformer-based variants).
- Explore cases in which models with larger residuals still produce acceptable or even superior control solutions, as seen in the singular arc.
- Develop principled methods to assess whether the training dataset is rich enough to learn a faithful physics surrogate without overfitting.
- Extend experiments to higher-dimensional systems. In all benchmark problems, our input signal $u$ was one-dimensional; the PDEs were also limited to simpler versions.
- Learn PDEs with varying parameters (e.g., diffusion coefficients), since we used only fixed values during training.

In summary, neural operators offer a promising direction for optimal control, especially when system dynamics are complex and simulation costs are high. However, their current use as control surrogates is limited by gradient reliability, dataset quality, architecture sensitivity, and lack of robustness. Further research is needed to make PINO-based control approaches reliable, interpretable, and safe in practice.

# 7 Conclusion

This thesis explored the application of physics-informed neural operators (PINO) to solving optimal control problems. Motivated by the limitations of traditional solvers in handling repeated or high-dimensional control tasks, we introduced a unified framework for PINO-based control that trains operator models using only physics residuals and initial condition penalties.

We implemented and compared three architectures - DeepONet, FNO, and LNO - on a selection of benchmark problems with known analytical solutions. Each model was trained using residual-only loss functions and evaluated through its use in the control optimization problem. Our experiments showed that residual-only training is feasible in practice, but fragile in the presence of noisy signals or insufficient data. FNO demonstrated the best trade-off between speed, expressivity, and robustness across varying control penalties. DeepONet was the most reliable at enforcing initial and boundary constraints due to its pointwise structure and compatibility with auto-differentiation. LNO produced the smoothest trajectories but failed to generalize to noisy inputs in nonlinear problems.

Compared to a baseline traditional solver (IPOPT) - with easy ODE constrained problems - PINO models struggled to match optimal cost and accuracy, especially in precision-sensitive problems. However, neural operators offer a compelling benefit: once trained, they can generalize across input functions and enable rapid control optimization at inference time since they accept changes in the control objective function. Sensitivity analysis further revealed that performance is highly dependent on loss weights. Changes in the ratio between the weight of physics loss and objective function loss led to high variations.

Benchmarking of FNO on PDEs in a control setting revealed that the performance of neural operators can be much better if the system that they are trying to learn is physically meaningful. We tried it only on FNO, but its solutions were consistently better than those of the traditional solver.

This work highlights both the promise and current limitations of using neural operators in optimal control. Future research should focus on improving robustness through hybrid training, architecture advancements, and systematic sensitivity analysis. Extending the framework to higher-dimensional or constrained systems and evaluating newer operator families may also expand the scope of PINO-based control in real-world applications.

## References

[1] N. Padhy. “Unit commitment-a bibliographical survey”. In: *IEEE Transactions on Power Systems* 19.2 (2004), pp. 1196–1205. DOI: 10.1109/TPWRS.2003.821611.

[2] A. E. Bryson and Y.-C. Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. 1st ed. Routledge, May 4, 2018. ISBN: 9781315137667. DOI: 10.1201/9781315137667.

[3] R. C. Merton. “Optimum consumption and portfolio rules in a continuous-time model”. In: *Journal of Economic Theory* 3.4 (Dec. 1971), pp. 373–413. ISSN: 00220531. DOI: 10.1016/0022-0531(71)90038-X.

[4] L. S. Pontryagin. *Mathematical Theory of Optimal Processes*. 1st ed. Routledge, May 3, 2018. ISBN: 9780203749319. DOI: 10.1201/9780203749319.

[5] R. Bellman. “On the Theory of Dynamic Programming”. In: *Proceedings of the National Academy of Sciences* 38.8 (Aug. 1952), pp. 716–719. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.38.8.716.

[6] J. T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Second. Society for Industrial and Applied Mathematics, Jan. 1, 2010. ISBN: 9780898718577. DOI: 10.1137/1.9780898718577.

[7] L. Biegler and V. Zavala. “Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization”. In: *Computers & Chemical Engineering* 33.3 (2009). Selected Papers from the 17th European Symposium on Computer Aided Process Engineering held in Bucharest, Romania, May 2007, pp. 575–582. ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2008.08.006.

[8] M. Raissi, P. Perdikaris, and G. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045.

[9] L. Lu et al. “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”. In: *Nature Machine Intelligence* 3.3 (Mar. 18, 2021), pp. 218–229. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00302-5.

[10] Z. Li et al. *Fourier Neural Operator for Parametric Partial Differential Equations*. 2021. arXiv: 2010.08895 [cs.LG]. URL: https://arxiv.org/abs/2010.08895.

[11] Q. Cao, S. Goswami, and G. E. Karniadakis. “Laplace neural operator for solving differential equations”. In: *Nature Machine Intelligence* 6.6 (June 24, 2024), pp. 631–640. ISSN: 2522-5839. DOI: 10.1038/s42256-024-00844-4.

[12] R. Hwang et al. “Solving PDE-Constrained Control Problems Using Operator Learning”. In: *Proceedings of the AAAI Conference on Artificial*

*Intelligence* 36 (June 2022), pp. 4504–4512. DOI: 10.1609/aaai.v36i4.20373.

[13] O. G. S. Lundqvist and F. Oliveira. *Was Residual Penalty and Neural Operators All We Needed for Solving Optimal Control Problems?* 2025. arXiv: 2506.04742 [math.OC]. URL: https://arxiv.org/abs/2506.04742.

[14] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[15] D. Jacobson. "Differential dynamic programming methods for solving bang-bang control problems". In: *IEEE Transactions on Automatic Control* 13.6 (Dec. 1968), pp. 661–675. ISSN: 0018-9286. DOI: 10.1109/TAC.1968.1099026.

[16] A. Wächter and L. T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical Programming* 106.1 (Mar. 2006), pp. 25–57. ISSN: 0025-5610, 1436-4646. DOI: 10.1007/s10107-004-0559-y.

[17] P. E. Gill, W. Murray, and M. A. Saunders. "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization". In: *SIAM Journal on Optimization* 12.4 (2002), pp. 979–1006. DOI: 10.1137/S1052623499350013.

[18] N. Kovachki et al. "Neural Operator: Learning Maps Between Function Spaces With Applications to PDEs". In: *Journal of Machine Learning Research* 24.89 (2023), pp. 1–97. ISSN: 1533-7928. URL: http://jmlr.org/papers/v24/21-1524.html.

[19] T. Tripura and S. Chakraborty. "Wavelet Neural Operator for solving parametric partial differential equations in computational mechanics problems". In: *Computer Methods in Applied Mechanics and Engineering* 404 (2023), p. 115783. ISSN: 0045-7825. URL: https://www.sciencedirect.com/science/article/pii/S0045782522007393.

[20] Z. Li et al. "Geometry-Informed Neural Operator for Large-Scale 3D PDEs". In: *Advances in Neural Information Processing Systems.* Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 35836–35854. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/70518ea42831f02afc3a2828993935ad-Paper-Conference.pdf.

[21] Z. Li et al. *Neural Operator: Graph Kernel Network for Partial Differential Equations.* 2020. arXiv: 2003.03485 [cs.LG]. URL: https://arxiv.org/abs/2003.03485.

[22] Z. Hao et al. "GNOT: A General Neural Operator Transformer for Operator Learning". In: *Proceedings of the 40th International Conference on Machine Learning.* Ed. by A. Krause et al. Vol. 202. Proceedings of Machine Learning Research. PMLR, June 2023, pp. 12556–12569. URL: https://proceedings.mlr.press/v202/hao23c.html.

[23] A. G. Baydin et al. "Automatic Differentiation in Machine Learning: a Survey". In: *Journal of Machine Learning Research* 18.153 (2018), pp. 1–43. URL: http://jmlr.org/papers/v18/17-468.html.

[24] Z. Li et al. "Physics-Informed Neural Operator for Learning Partial Differential Equations". In: *ACM / IMS J. Data Sci.* 1.3 (May 2024). DOI: 10.1145/3648506.

[25] V. Madhavan et al. *Self-supervised Pretraining for Partial Differential Equations*. 2024. arXiv: 2407.06209 [cs.LG]. URL: https://arxiv.org/abs/2407.06209.

[26] T. Wang and C. Wang. "Latent Neural Operator Pretraining for Solving Time-Dependent PDEs". In: *Neural Information Processing*. Ed. by M. Mahmud et al. Vol. 2282. Singapore: Springer Nature Singapore, 2025, pp. 163–178. ISBN: 9789819669486. DOI: 10.1007/978-981-96-6948-6_12.

[27] A. Hemmasian and A. B. Farimani. *Pretraining a Neural Operator in Lower Dimensions*. 2024. arXiv: 2407.17616 [cs.LG]. URL: https://arxiv.org/abs/2407.17616.

[28] M. Raissi. "Deep Hidden Physics Models: Deep Learning of Nonlinear Partial Differential Equations". In: *Journal of Machine Learning Research* 19.25 (2018), pp. 1–24. URL: http://jmlr.org/papers/v19/18-046.html.

[29] S. Mowlavi and S. Nabi. "Optimal control of PDEs using physics-informed neural networks". In: *Journal of Computational Physics* 473 (2023), p. 111731. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2022.111731.

[30] Y. Chen, Y. Shi, and B. Zhang. *Optimal Control Via Neural Networks: A Convex Approach*. 2019. arXiv: 1805.11835 [math.OC]. URL: https://arxiv.org/abs/1805.11835.

[31] P. Yin et al. "AONN: An Adjoint-Oriented Neural Network Method for All-At-Once Solutions of Parametric Optimal Control Problems". In: *SIAM Journal on Scientific Computing* 46.1 (2024), pp. C127–C153. DOI: 10.1137/22M154209X.

[32] E. Schiassi, F. Calabrò, and D. E. De Falco. "Pontryagin Neural Networks for the Class of Optimal Control Problems With Integral Quadratic Cost". In: *Aerospace Research Communications* Volume 2 - 2024 (2024). ISSN: 2813-6209. DOI: 10.3389/arc.2024.13151.

[33] M. Feng et al. "Optimal Control Operator Perspective and a Neural Adaptive Spectral Method". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 39 (Apr. 2025), pp. 14567–14575. DOI: 10.1609/aaai.v39i14.33596.

[34] A. Savitzky and M. J. E. Golay. "Smoothing and Differentiation of Data by Simplified Least Squares Procedures." In: *Analytical Chemistry* 36.8 (July 1, 1964), pp. 1627–1639. ISSN: 0003-2700, 1520-6882. DOI: 10.1021/ac60214a047.

[35] E. Torres, J. Schiefer, and M. Niepert. *Adaptive Physics-informed Neural Networks: A Survey*. 2025. arXiv: 2503.18181 [cs.LG]. URL: https://arxiv.org/abs/2503.18181.

[36] S. De et al. “Bi-fidelity modeling of uncertain and partially unknown systems using DeepONets”. In: *Comput. Mech.* 71.6 (Mar. 2023), pp. 1251–1267. ISSN: 0178-7675. DOI: 10.1007/s00466-023-02272-4.

[37] S. Goswami et al. “Physics-Informed Deep Neural Operator Networks”. In: *Machine Learning in Modeling and Simulation: Methods and Applications*. Ed. by T. Rabczuk and K.-J. Bathe. Cham: Springer International Publishing, 2023, pp. 219–254. ISBN: 978-3-031-36644-4. DOI: 10.1007/978-3-031-36644-4_6.

[38] D. E. Kirk. *Optimal control theory: an introduction*. Mineola, N.Y: Dover Publications, 2004. 452 pp. ISBN: 9780486434841. URL: https://dl.icdst.org/pdfs/files3/d0057acc9daf94f3e2b14d1498f2421b.pdf.

[39] J. Vlassenbroeck and R. Van Dooren. “A Chebyshev technique for solving nonlinear optimal control problems”. In: *IEEE Transactions on Automatic Control* 33.4 (1988), pp. 333–340. DOI: 10.1109/9.192187.

[40] F. L. Lewis, D. L. Vrabie, and V. L. Syrmos. *Optimal control*. 3rd ed. Hoboken: Wiley, 2012. 540 pp. ISBN: 9780470633496.

[41] Q. Gong, W. Kang, and I. Ross. “A pseudospectral method for the optimal control of constrained feedback linearizable systems”. In: *IEEE Transactions on Automatic Control* 51.7 (2006), pp. 1115–1129. DOI: 10.1109/TAC.2006.878570.

[42] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. URL: https://www.gaussianprocess.org/gpml/.

[43] P. Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature Methods* 17.3 (Mar. 2, 2020), pp. 261–272. ISSN: 1548-7091, 1548-7105. DOI: 10.1038/s41592-019-0686-2.

[44] M. Begantsova. *operator-learning-opt-control (Release 1.0.0) [Computer software]*. GitHub. July 29, 2025. URL: https://github.com/kankeinai/operator-learning-opt-control/releases/tag/v1.0.0.

[45] J. A. E. Andersson et al. “CasADi: a software framework for nonlinear optimization and optimal control”. In: *Mathematical Programming Computation* 11.1 (Mar. 2019), pp. 1–36. ISSN: 1867-2949, 1867-2957. DOI: 10.1007/s12532-018-0139-4.

[46] *MIT License*. https://opensource.org/licenses/MIT. Accessed: 2025-07-31.

[47] A. Rao. “A Survey of Numerical Methods for Optimal Control”. In: *Advances in the Astronautical Sciences* 135 (Jan. 2010).

# A Neural Operator architectures

## A.1 DeepONet

The DeepONet architecture is inspired by the Universal Approximation Theorem for operators [9], which ensures that a sufficiently wide single-hidden-layer network can approximate any continuous non-polynomial operator mapping functions to functions. We restate this theorem below for reference.

**Theorem 1 (Universal Approximation Theorem for Operator).** *Suppose that $\sigma$ is a continuous non-polynomial function, $X$ is a Banach Space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in $X$ and $\mathbb{R}^d$, respectively, $V$ is a compact set in $C(K_1)$, $G$ is a nonlinear continuous operator, which maps $V$ into $C(K_2)$. Then for any $\epsilon > 0$, there are positive integers $n$, $p$, $m$, constants $c_i^k$, $\xi_{ij}^k$, $\theta_i^k$, $\zeta_k \in \mathbb{R}$, $w_k \in \mathbb{R}^d$, $x_j \in K_1$, $i = 1, \dots, n$, $k = 1, \dots, p$, $j = 1, \dots, m$, such that*

$$\left| G(u)(y) - \sum_{k=1}^{p} c_i^k \sum_{i=1}^{n} \sigma \left( \sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right) \sigma(w_k \cdot y + \zeta_k) \right| < \epsilon \tag{1}$$

*holds for all $u \in V$ and $y \in K_2$.*

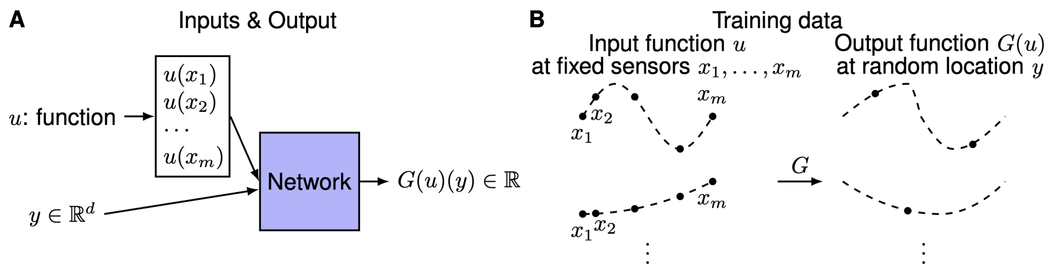


**Figure A1:** Illustration of the DeepONet operator learning paradigm (adapted from [9]).

Figure A1 illustrates Theorem 1 formulation, where a single network directly processes sampled function values and a query location to produce the scalar output.

In the original formulation, we can notice two networks. The *branch network* computes

$$\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k$$

to encode the sampled values of $u$, for each neuron $i$ in each branch $k$. The *trunk network* applies

$$\sigma(w_k \cdot y + \zeta_k)$$

To encode the evaluation location $y$.

DeepONet replaces each shallow module with a deeper feed-forward network. Denote

$$\mathbf{b} = \mathrm{BranchNet}(u; \theta_b) \in \mathbb{R}^p, \quad \mathbf{t} = \mathrm{TrunkNet}(y; \theta_t) \in \mathbb{R}^p.$$

Then the operator is approximated by their inner product plus a bias:

$$G(u)(y) \approx \sum_{k=1}^{p} b_k\, t_k + b_0.$$

There are two versions of the original DeepONet:

- **Unstacked:** A single branch network outputs the full vector $\mathbf{b}$ at once. This reduces memory usage and simplifies backpropagation.
- **Stacked:** $p$ separate branch networks, each producing one scalar $b_k$. This can increase representational flexibility but at a high computational cost.

Figure A2 compares the two implementation variants.

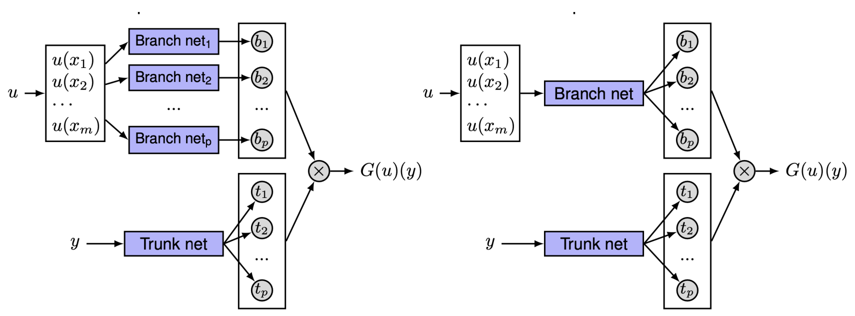


**Figure A2:** Two implementations of DeepONet's branch network (adapted from [9]). Left (unstacked): a single BranchNet outputs the full feature vector $\mathbf{b} \in \mathbb{R}^p$. Right (stacked): $p$ independent branch networks each produce one component $b_k$

## A.2 Fourier Neural Operator

The *neural operator* framework, proposed by [21], approximates mappings between infinite-dimensional function spaces using iterative architectures. The core idea is to construct a sequence of features.

$$v_0 \to v_1 \to \cdots \to v_T,$$

where each $v_t$, for $t = 0, \ldots, T$ takes values in $\mathbb{R}^{d_v}$.

Initially, the input function $a(x)$ is *lifted* to a higher-dimensional representation via a pointwise transformation $P$, yielding:

$$v_0(x) = P\big(a(x)\big) \in \mathbb{R}^{d_v},$$

Where $P$ is a pointwise linear (or affine) map. We then apply $T$ refinement steps:

$$v_{t+1}(x) = \sigma\Big(W\, v_t(x) + (\mathcal{K}(\phi)\, v_t)(x)\Big), \quad t = 0, \dots, T-1,$$

with $\sigma$ a nonlinear activation, $W \in \mathbb{R}^{d_v \times d_v}$ a learnable matrix, and the integral operator

$$(\mathcal{K}(\phi)\, v_t)(x) = \int_{\mathcal{D}} k_\phi\Big(x, y, a(x), a(y)\Big)\, v_t(y)\, \mathrm{d}y,$$

where $k_\phi : \mathbb{R}^{2(d+d_a)} \to \mathbb{R}^{d_v \times d_v}$ is a kernel network. Finally, a decoder $Q : \mathbb{R}^{d_v} \to \mathbb{R}^{d_u}$ produces

$$u(x) = Q\Big(v_T(x)\Big).$$

This framework was successful at generalizing neural networks to infinite-dimensional spaces. Though the integral operator is linear, such an architecture can still learn highly non-linear operators.

The Fourier Neural Operator (FNO) [10] makes two key simplifications to architecture described above:

1. *Kernel no longer depends on input function $a(x)$: $k_\phi(x, y, a(x), a(y)) \approx k_\phi(x, y)$.*

2. *Translation invariance: $k_\phi(x, y) = k_\phi(x - y)$,*

This allows the integral operator $\mathcal{K}$ to be implemented as a *convolution*, which can be computed efficiently in the Fourier domain. Specifically, the convolution becomes:

$$\mathcal{K}(\phi)\, v_t = \mathcal{F}^{-1}\Big(\mathcal{R}_\phi \cdot \mathcal{F}(v_t)\Big),$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the FFT and its inverse, and $\mathcal{R}_\phi$ is a learnable complex multiplier on the lowest $K$ Fourier modes (higher modes are truncated). This reduces per-layer cost from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ on a uniform grid of $n$ points.

The whole architecture can be seen in the Figure 5.

## A.3 Laplace Neural Operator

While the Fourier Neural Operator (FNO) [10] leverages FFT to accelerate convolutional integral operators in the frequency domain, it suffers from three key limitations. First, the classical Fourier transform requires input functions to be integrable, and thus cannot represent signals such as $|x(t)|$ or model unstable dynamics. Second, Fourier-based layers capture only steady-state, periodic responses and omit any dependence on initial conditions. Third, non-periodic domains must be artificially periodized, often degrading accuracy near boundaries.

The Laplace transform addresses these issues by introducing an exponential decay factor. For a causal signal $x(t)$, its Laplace transform

$$\mathcal{L}\{x\}(s) = \int_0^\infty x(t)\, e^{-st}\, \mathrm{d}t, \quad s = \sigma + i\omega,$$

converges under milder conditions than the Fourier transform and naturally encodes initial values via the differentiation properties:

$$\mathcal{L}\{\dot{x}(t)\} = s\,\mathcal{L}\{x(t)\} - x(0), \qquad \mathcal{L}\{\ddot{x}(t)\} = s^2\,\mathcal{L}\{x(t)\} - s\,x(0) - \dot{x}(0).$$

As a result, the Laplace domain simultaneously represents transient behavior (through $\sigma$) and oscillatory modes (through $\omega$) while embedding initial-condition information. Motivated by these advantages, the authors proposed replacing the Fourier layer with a *Laplace layer* (see Figure 6).

First, the input function $a(x)$ is lifted to a higher-dimensional feature field via a pointwise map $\mathcal{P}$, yielding $v(x) \in \mathbb{R}^{d_x}$, and then processed by

$$u(x) = \sigma\Big((k(a;\phi) * v)(x)\Big) + W\,v(x),$$

where $\sigma$ is a nonlinear activation, $W$ a learnable linear transformation, and $k$ an integral kernel. The intermediate representation $u$ is finally decoded by $Q$ to produce the output $x(t)$. This overall pipeline mirrors that of FNO, but with a key modification in how the convolution is implemented.

In LNO, the convolution is carried out in the Laplace domain:

$$U(s) = \mathcal{L}\Big((k(a;\phi) * v)(t)\Big) = K_\phi(s)\,V(s),$$

where $K_\phi(s) = \mathcal{L}\{k_\phi(t)\}$ and $V(s) = \mathcal{L}\{v(t)\}$. Crucially, $K_\phi(s)$ is parameterized in a *pole–residue* form:

$$K_\phi(s) = \sum_{n=1}^{N} \frac{\beta_n}{s - \mu_n},$$

with trainable poles $\{\mu_n\}$ and residues $\{\beta_n\}$.

One may also expand $v(t)$ in a Fourier series over $[0, T]$:

$$v(t) = \sum_{l=-\infty}^{\infty} \alpha_l\, e^{i\omega_l t}, \quad 0 \le t < T,$$

where $\omega_l = l\,\omega_1$ and $\alpha_l$ are the complex coefficients, giving

$$V(s) = \mathcal{L}\{v(t)\} = \sum_{l=-\infty}^{\infty} \frac{\alpha_l}{s - i\omega_l}.$$

Combining these expansions and expressing them in pole residue form (discussion is omitted since it is out of scope of the thesis), one obtains

$$U(s) = \sum_{n=1}^{N} \frac{\gamma_n}{s - \mu_n} + \sum_{l=-\infty}^{\infty} \frac{\lambda_l}{s - i\omega_l}.$$

After determining $\{\gamma_n\}$ and $\{\lambda_l\}$, the inverse Laplace transform yields

$$u_1(t) = \mathcal{L}^{-1}(U(s)) = \sum_{n=1}^{N} \gamma_n \, e^{\mu_n t} + \sum_{l=-\infty}^{\infty} \lambda_l \, e^{i\omega_l t},$$

which is then passed through $Q$ to recover the solution $x(t)$.

# B Analytical Solutions for Benchmark Problems

## B.1 Oscillatory Forcing (Problem 2) Derivation

Consider the problem

$$\min_u \frac{1}{2}\int_0^1 [x(t)^2 + u(t)^2]\, dt, \qquad \dot{x} = \cos(4\pi t) + u, \quad x(0) = 0,\, x(1) = 0$$

**Step 1: Pontryagin's Maximum Principle.**

Hamiltonian:

$$H = \frac{1}{2}(x^2 + u^2) + \lambda(\cos(4\pi t) + u)$$

Optimality condition:

$$\frac{\partial H}{\partial u} = u + \lambda = 0 \implies u^*(t) = -\lambda(t)$$

Costate dynamics:

$$\dot{\lambda}(t) = -\frac{\partial H}{\partial x} = -x(t)$$

State dynamics:

$$\dot{x}(t) = \cos(4\pi t) + u^*(t) = \cos(4\pi t) - \lambda(t)$$

**Step 2: Reduce to a Second-Order ODE.**

Differentiate the state equation:

$$\ddot{x}(t) = -4\pi \sin(4\pi t) - \dot{\lambda}(t)$$

But $\dot{\lambda}(t) = -x(t)$, so:

$$\ddot{x}(t) = -4\pi \sin(4\pi t) + x(t)$$
$$\implies \ddot{x}(t) - x(t) = -4\pi \sin(4\pi t)$$

**Step 3: General Solution.**

The homogeneous solution is $A\cosh t + B\sinh t$. A particular solution (by undetermined coefficients) is:

$$x_p(t) = C\sin(4\pi t)$$

Compute:

$$\ddot{x}_p - x_p = -16\pi^2 C\sin(4\pi t) - C\sin(4\pi t) = -(16\pi^2 + 1)C\sin(4\pi t)$$

Set equal to $-4\pi\sin(4\pi t)$ to solve for $C$:

$$-(16\pi^2+1)C = -4\pi \implies C = \frac{4\pi}{16\pi^2+1}$$

Full solution:

$$x^*(t) = A\cosh t + B\sinh t + \frac{4\pi}{16\pi^2+1}\sin(4\pi t)$$

**Step 4: Apply Boundary Conditions.**

- $x(0) = 0$: $A\cdot 1 + B\cdot 0 + 0 = 0 \implies A = 0$
- $x(1) = 0$: $B\sinh 1 + \frac{4\pi}{16\pi^2+1}\sin(4\pi) = 0$

But $\sin(4\pi) = 0$, so:

$$B\sinh 1 = 0 \implies B = 0$$

Thus, the unique solution is

$$x^*(t) = \frac{4\pi}{16\pi^2+1}\sin(4\pi t)$$

**Step 5: Compute the Optimal Control.**

Recall:

$$u^*(t) = \dot{x}^*(t) - \cos(4\pi t)$$

Compute the derivative:

$$\dot{x}^*(t) = \frac{4\pi}{16\pi^2+1}\cdot 4\pi\cos(4\pi t) = \frac{16\pi^2}{16\pi^2+1}\cos(4\pi t)$$

Thus,

$$u^*(t) = \frac{16\pi^2}{16\pi^2+1}\cos(4\pi t) - \cos(4\pi t) = -\frac{1}{16\pi^2+1}\cos(4\pi t)$$

**Step 6: Final Analytical Solution.**

$$\boxed{x^*(t) = \frac{4\pi}{16\pi^2+1}\sin(4\pi t), \qquad u^*(t) = -\frac{1}{16\pi^2+1}\cos(4\pi t)}$$

This solution satisfies all optimality and boundary conditions for the fixed-endpoint problem.

## B.2 Polynomial Target Tracking (Problem 3) Derivation

Consider

$$\min_u \int_0^1 \Big((x(t) - t^2)^2 + u(t)^2\Big)\, dt, \quad \dot{x} = u,\ x(0) = 0$$

**Step 1: Pontryagin's Principle.**

Hamiltonian:

$$H = (x - t^2)^2 + u^2 + \lambda u$$

Optimality:

$$\frac{\partial H}{\partial u} = 2u + \lambda = 0 \implies u^* = -\frac{1}{2}\lambda$$

Costate:

$$\dot{\lambda} = -\frac{\partial H}{\partial x} = -2(x - t^2)$$

State:

$$\dot{x} = u$$

**Step 2: Reduce to Second-Order ODE.**

Plug in $u^*$:

$$\dot{x} = -\frac{1}{2}\lambda, \quad \dot{\lambda} = -2(x - t^2)$$

Differentiate the state equation:

$$\ddot{x} = -\frac{1}{2}\dot{\lambda} = (x - t^2) \implies \ddot{x} - x = -t^2$$

**Step 3: General Solution.**

The homogeneous solution is $Ae^t + Be^{-t}$, but for $t \in [0, 1]$ and this system, a polynomial suffices:

$$x^*(t) = \alpha t^3 + \beta t^2 + \gamma t$$

Plug into $\ddot{x} - x = -t^2$ and solve for coefficients:

$$6\alpha t + 2\beta - (\alpha t^3 + \beta t^2 + \gamma t) = -t^2$$

Match powers and solve (or directly check the known solution).

Given the candidate

$$x^*(t) = \frac{2}{3}t^3 - t^2 + t$$

compute

$$\dot{x}^*(t) = 2t^2 - 2t + 1 = u^*(t)$$

**Step 4: Initial and Terminal Costate.**

With $x(0) = 0$, and terminal state unconstrained, the costate satisfies $\lambda(1) = 0$. The solution above satisfies all necessary conditions.

## B.3 Singular Arc ("Cliff") Problem (Problem 5) Derivation

Consider

$$\min_u \int_0^1 u(t)^2\,dt, \quad \dot{x}(t) = x(t)^2 + u(t),\ x(0) = 1,\ x(1) = 0$$

**Step 1: Pontryagin's Principle.**

Hamiltonian:

$$H = u^2 + \lambda(x^2 + u)$$

Optimality:

$$\frac{\partial H}{\partial u} = 2u + \lambda = 0 \implies u^* = -\frac{1}{2}\lambda$$

Costate:

$$\dot{\lambda} = -\frac{\partial H}{\partial x} = -2\lambda x$$

State:

$$\dot{x} = x^2 + u$$

**Step 2: Guess Solution by Substitution.**

The known analytical solution is (see Bryson & Ho, Rao [2, 47]):

$$x^*(t) = \frac{1-t}{1+t}$$

Compute

$$\dot{x}^*(t) = \frac{-2}{(1+t)^2}$$

Compute $x^*(t)^2 = \left(\frac{1-t}{1+t}\right)^2$

So,

$$u^*(t) = \dot{x}^*(t) - x^*(t)^2 = -\frac{2}{(1+t)^2} - \left(\frac{1-t}{1+t}\right)^2$$

**Step 3: Check Boundary Conditions.**

$$x^*(0) = 1, \quad x^*(1) = 0$$

This matches the stated initial and final states.