MOX-Report No. 31/2025

HypeRL: Parameter-Informed Reinforcement Learning for Parametric PDEs

Botteghi, N.; Fresca, S.; Guo, M.; Manzoni, A.

MOX, Dipartimento di Matematica
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

mox-dmat@polimi.it https://mox.polimi.it

HypeRL: Parameter-Informed Reinforcement Learning for Parametric PDEs

Nicolò Botteghi[a], Stefania Fresca[b], Mengwu Guo[c], Andrea Manzoni[b]

[a] *Mathematics of Imaging and AI, University of Twente, Enschede, Netherlands*
[b] *MOX – Department of Mathematics, Politecnico di Milano, Piazza Leonardo da Vinci 32, I-20133, Milano, Italy*
[c] *Centre for Mathematical Sciences, Lund University, Lund, Sweden*

Abstract

In this work, we devise a new, general-purpose reinforcement learning strategy for the optimal control of parametric partial differential equations (PDEs). Such problems frequently arise in applied sciences and engineering and entail a significant complexity when control and/or state variables are distributed in high-dimensional space or depend on varying parameters. Traditional numerical methods, relying on either iterative minimization algorithms – exploiting, e.g., the solution of the adjoint problem – or dynamic programming – also involving the solution of the Hamilton-Jacobi-Bellman (HJB) equation – while reliable, often become computationally infeasible. Indeed, in either way, the optimal control problem has to be solved for each instance of the parameters, and this is out of reach when dealing with high-dimensional time-dependent and parametric PDEs. In this paper, we propose *HypeRL*, a deep reinforcement learning (DRL) framework to overcome the limitations shown by traditional methods. HypeRL aims at approximating the optimal control policy directly, bypassing the need to numerically solve the HJB equation explicitly for all possible states and parameters, or solving an adjoint problem within an iterative optimization loop for each parameter instance. Specifically, we employ an actor-critic DRL approach to learn an optimal feedback control strategy that can generalize across the range of variation of the parameters. To effectively learn such optimal control laws for different instances of the parameters, encoding the parameter information into the DRL policy and value function neural networks (NNs) is essential. To do so, HypeRL uses two additional NNs, often called *hypernetworks*, to learn the weights and biases of the value function and the policy NNs. In this way, HypeRL effectively embeds the parametric information into the value function and policy NNs. We validate the proposed approach on two PDE-constrained optimal control benchmarks, namely a 1D Kuramoto-Sivashinsky equation with in-domain control and on a 2D Navier-Stokes equations with boundary control, by showing that the knowledge of the PDE parameters and how this information is encoded, i.e., via a hypernetwork, is an essential ingredient for learning parameter-dependent control policies that can generalize effectively to unseen scenarios and for improving the sample efficiency of such policies.

1. Introduction

Many complex, distributed dynamical systems can be modeled through a set of parametrized partial differential equations (PDEs) and their optimal control (OC) represents a crucial challenge in many engineering and science applications, going way beyond the single, direct simulation of these systems. OC allows the integration of active control mechanisms into a control system and its most common application in addressing such problems involves determining optimal closed-loop controls that minimize a specified objective functional [1]. To solve a PDE-constrained OC problem, one possibility is to rely on the Hamilton-Jacobi-Bellman (HJB) equation. However, the HJB is not easily tractable and is usually computationally expensive for high-dimensional and large-time horizon control problems. Another option is to locally solve the OC problem by exploiting the Pontryagin Maximum Principle (PMP). However, PMP involves the backward solution (in time) of the adjoint problem with the same dimension of the state equation. Hence, to find the OC law one should solve both the state and the adjoint equation repeatedly – forward and backward in time, respectively – in the whole space-time domain. For high-dimensional problems, storage and computational requirements make the PMP becoming quickly prohibitive. Traditional OC theory may present non-negligible shortcomings [2], which are even more severe when the PDE parameters vary and the OC problem has to be solved for each new instance of the parameters.

In this respect, reinforcement learning (RL) [3] is emerging as a new paradigm to address the solution of PDE-constrained OC problems and has been shown to outperform other OC strategies when the system's states are high-dimensional, noisy, or only partial measurements are available. RL avoids to solve the HJB or the adjoint equations explicitly, which would be untractable for extremely complex problems. Unlike the aforementioned OC approaches, RL aims to solve control problems by learning an OC law (the *policy*), while interacting with the dynamical system (the *environment*). RL assumes no prior knowledge of the system, thus yielding broadly applicable control approaches. Deep reinforcement learning (DRL) is the extension of RL using deep neural networks (NNs) to represent value functions and policies [4, 5, 6]. DRL has shown outstanding capabilities in complex control problems such as games [7, 8, 9, 10, 11, 12], simulated and real-world robotics [13, 14, 15, 16, 17, 18, 19], and recently PDEs, with particular emphasis on fluid dynamics [20, 21, 22, 23, 24, 25, 26, 27].

Despite its success, DRL still suffers from two major drawbacks, namely *(i)* the *sample inefficiency*, making the DRL algorithms extremely data hungry, and *(ii)* the *limited generalization* of the control strategies to changes in the environments. Tackling these two challenges is crucial for advancing DRL towards large-scale and real-world problems. These limitations are especially severe in the context of control of parametric PDEs, where obtaining (state) measurements is challenging due to the computational complexity of the (forward) PDE models. Moreover, little to no attempt has yet been made to devise DRL algorithms capable of handling changes in the systems' dynamics resulting from variations in known PDE parameters. This means that for any new configuration of the system, the optimization problem must be solved from scratch. Additionally, while DRL generally decreases the computational complexity of traditional methods for the solution of OC problems, these algorithms still require huge training times and large amounts of data, i.e., we need the repeated evaluation of the solution to the system state equations. Consequently, applying DRL algorithms to address single problem scenarios would not be entirely justified.

Improving sample efficiency and generalization in DRL has been a key focus of recent research. Examples of such approaches are *imitation learning* [28, 29], where expert data are used to pre-train the control policies and to speed-up the (policy-)optimization process, *transfer learning* [30, 31], where an optimal policy is transferred to a new environment with little or no retraining, and *unsupervised representation learning* [32, 33], where unsupervised learning techniques are exploited to learn compact representations of the data. Representation learning has been shown to improve the generalization of control policies to new environments and scenarios [33]. Eventually, another prominent approach for enhancing the generalization capabilities of DRL agents is *meta learning* [34, 35], where DRL policies are specifically built and optimized for adapting to new scenarios. However, these approaches have yet to be developed to tackle the challenging problem of controlling parametric PDEs, leaving a large gap for research and developments in the field.

Hypernetworks [36] are a class of NNs that provide the parameters, i.e., the weights and biases, of other NNs, often referred to as main or primary networks. Hypernetworks have shown promising results in a variety of deep learning problems, including continual learning, causal inference, transfer learning, weight pruning, uncertainty quantification, zero-shot learning, natural language processing, and recently DRL [37]. Indeed, hypernetworks are capable of enhancing the flexibility, expressivity, and performance of deep learning-based architecture, opening new doors for the development of novel and more advanced architectures. Hypernetworks in DRL were first used in [38] to learn the parameters of the value function or of the policy NNs. Enhancing DRL with hypernetworks has been done in the context of meta RL, zero-shot RL, and continual RL [39, 40, 41] for improving the performance of RL agents. However, to the best of our knowledge, no one has tackled the problem of controlling parametric PDEs with hypernetworks and DRL so far.

In this paper, we propose a novel parameter-informed DRL framework, namely HypeRL, for the efficient solution of parameter-dependent PDE-constrained OC problems by addressing the two aforementioned limitations, namely sample efficiency and generalization. In particular, with reference to Figure 1, we exploit the knowledge of PDE parameters $\boldsymbol{\mu}$ to learn a parametrization of the control policy (and value function), dependent on the parameters of the PDE, by means of a hypernetwork $h(\boldsymbol{\mu}; \boldsymbol{\theta}_{h_\pi})$ taking as input the PDE parameters $\boldsymbol{\mu}$ and providing as output the weights and biases $\boldsymbol{\theta}_\pi$ of the policy $\pi(\boldsymbol{y}; \boldsymbol{\theta}_\pi)$ (and value function). In contrast with the simple and widely-used concatenation of the parameters μ to the PDE state $\boldsymbol{y}$, this parametrization allows for learning OC strategies with less data and that can better adapt to unseen instances of the PDE parameters, within and without the training range, i.e., interpolation and extrapolation, respectively. Additionally, we show that the knowledge of the PDE parameters is crucial for learning OC strategies, and that parameter-informed DRL outperforms parameter-unaware DRL.

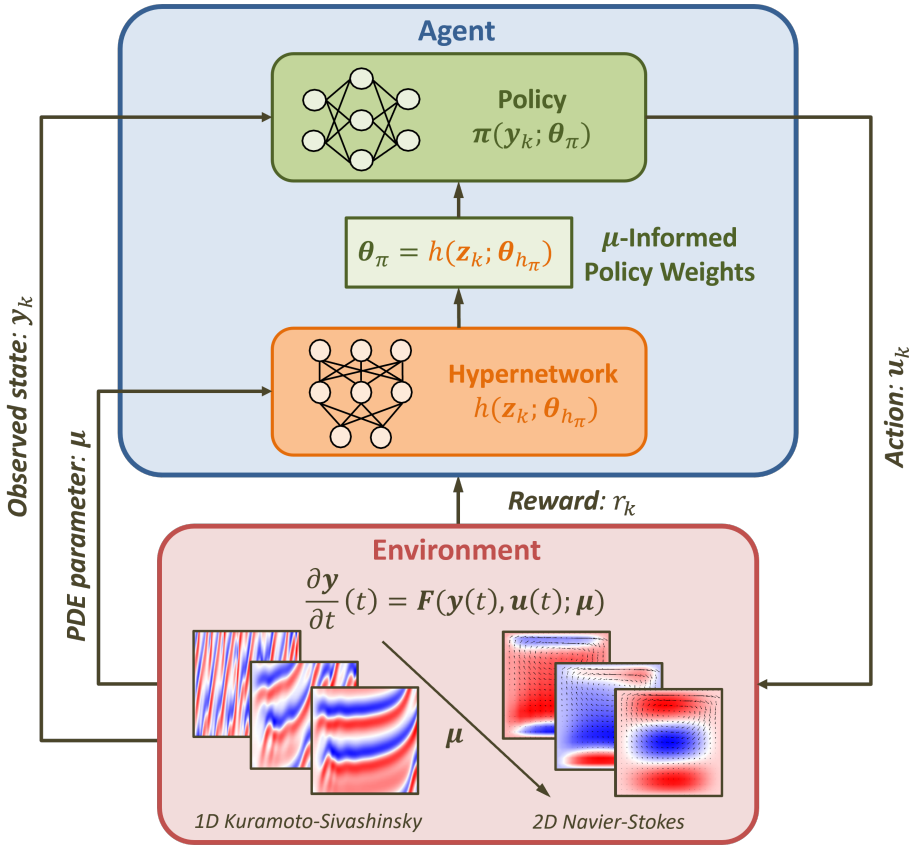


Figure 1: HypeRL for parametric PDE-constrained OC. We rely on a hypernetwork $h(\boldsymbol{\mu};\boldsymbol{\theta}_{h_\pi})$ to learn, from the PDE parameters $\boldsymbol{\mu}$, the weights and biases of the policy (and value function) neural network.

The paper is organized as follows: in Section 2, we introduce the building blocks of our framework, namely OC, RL, and hypernetworks. In Section 3, we describe our parameter-informed HypeRL framework combining DRL with hypernetworks, and in Section 4 we show the results and discuss the findings. Some conclusions are finally reported in Section 5.

2. Preliminaries

In this section, we frame the class of optimal control (OC) problems we aim at solving, and we introduce the building blocks – namely reinforcement learning (RL) and hypernetworks – that will be combined, at a later stage, to obtain the new HypeRL framework for OC of parametric partial differential equations (PDEs).

2.1. Optimal Control Problems

Along the paper we consider time-continuous OC problems under the following form [1]:

$$\begin{aligned} &\min_{\boldsymbol{y}\in\mathcal{Y},\ \boldsymbol{u}\in\mathcal{U}_{\text{ad}}} && J(\boldsymbol{y},\boldsymbol{u}) \\ &\text{subject to} && \frac{\partial \boldsymbol{y}}{\partial t}(t) = \boldsymbol{F}(\boldsymbol{y}(t),\boldsymbol{u}(t);\boldsymbol{\mu}) \quad \text{for } t\in[t_0,t_f]\,,\ \text{with } \boldsymbol{y}(t_0)=\boldsymbol{y}_0\,, \end{aligned} \tag{1}$$

in which $\boldsymbol{y} : [t_0,t_f] \to \mathbb{R}^{|\boldsymbol{y}|}$ denotes the state solution in a function space $\mathcal{Y} = \mathbb{R}^{|\boldsymbol{y}|} \otimes C^1[t_0,t_f]$, and $\boldsymbol{u} : [t_0,t_f] \to U_{\text{ad}} \subset \mathbb{R}^{|\boldsymbol{u}|}$ denotes the control input functions in an admissible space $\mathcal{U}_{\text{ad}} = U_{\text{ad}} \otimes C[t_0,t_f]$, U_{ad} being the admissible set of input control vectors at each time instance. The state equation represents a semi-discretized form of a nonlinear time-dependent PDE characterized by a parameter vector $\boldsymbol{\mu} \in \mathcal{M} \subset \mathbb{R}^{|\boldsymbol{\mu}|}$,

$\mathcal{M}$ being the compact parameter space, $\boldsymbol{F}$ is continuous in $\boldsymbol{u}$ and Lipschitz continuous in $\boldsymbol{y}$ (with a Lipschitz constant independent of $\boldsymbol{u}$)[1], $J : \mathcal{Y} \times \mathcal{U}_{\text{ad}} \to \mathbb{R}$ is a cost functional, and $\boldsymbol{y_0}$ gives a prescribed initial condition.

In this work, we consider cost functionals of the following form:

$$J(\boldsymbol{y}, \boldsymbol{u}) = \int_{t_0}^{t_f} L(\boldsymbol{y}(\tau), \boldsymbol{u}(\tau), \tau) \, \mathrm{d}\tau \,, \tag{2}$$

in which

$$L : \mathbb{R}^{|\boldsymbol{y}|} \times U_{\text{ad}} \times [t_0, t_f] \to \mathbb{R}, \quad (\boldsymbol{x}, \boldsymbol{z}, \tau) \mapsto \frac{1}{2}||\boldsymbol{x} - \boldsymbol{y}_{\text{ref}}(\tau)||^2 + \frac{\alpha}{2}||\boldsymbol{z} - \boldsymbol{u}_{\text{ref}}(\tau)||^2 \,, \tag{3}$$

where α is a scalar coefficient balancing the contribution of the two terms, $\boldsymbol{y}_{\text{ref}}$ and $\boldsymbol{u}_{\text{ref}}$ are the reference values of the state and of the control, respectively, and $\| \cdot \|_2$ represents the Euclidean norm.

The OC problem in (1) can be addressed by introducing the value function $V : \mathbb{R}^{|\boldsymbol{y}|} \times [t_0, t_f] \to \mathbb{R}$ as follows:

$$V(\boldsymbol{y}(t), t) := \min_{\boldsymbol{u} \in U_{\text{ad}} \otimes C[t, t_f]} \int_t^{t_f} L(\boldsymbol{y}(\tau), \boldsymbol{u}(\tau), \tau) \, \mathrm{d}\tau \,. \tag{4}$$

Note that the state equation is satisfied over the time interval $[t, t_f]$ with the state starting with $\boldsymbol{y}(t)$. To note, such a value function is the solution to the Hamilton-Jacobi-Bellman (HJB) [43, 44] equation:

$$-\frac{\partial V(\boldsymbol{y}, t)}{\partial t} = \min_{\boldsymbol{u}(t) \in U_{\text{ad}}} \left\{ L(\boldsymbol{y}, \boldsymbol{u}(t), t) + \boldsymbol{F}(\boldsymbol{y}, \boldsymbol{u}(t); \boldsymbol{\mu})^T \frac{\partial V(\boldsymbol{y}, t)}{\partial \boldsymbol{y}} \right\}, \quad (\boldsymbol{y}, t) \in \mathbb{R}^{|\boldsymbol{y}|} \times [t_0, t_f] \,, \tag{5}$$

which provides, at least in principle, the solution of the OC problem (1), that is, $\min J = V(\boldsymbol{y}_0, t_0)$.

The HJB equation is typically a high-dimensional PDE as $|\boldsymbol{y}|$ is often large. Moreover, due to the dependency of $\boldsymbol{F}(\cdot)$ on the (many) parameters $\boldsymbol{\mu}$, the HJB equation has to be solved for each sampling location of the parameters, because each sample defines an individual optimal control problem for the specific $\boldsymbol{\mu}$. Therefore, (5) is not easily tractable and may be computationally prohibitive for high-dimensional OC problems constrained by PDEs. This trait is commonly known as the *curse of dimensionality* [3]. We also note that an alternative approach to solving the constrained OC problem (1) uses the Karush-Kuhn-Tucker (KKT) optimality conditions [45, 46] via the introduction of a Lagrange multiplier. However, also in this case the iterative nature of the optimization methods (like, e.g., gradient-based, Newton, quasi-Newton, sequential quadratic programming) makes the numerical solution of each PDE-constrained optimization problem usually very hard. In the case of multiple OC problems, for different parameter values, the overall computational cost would be therefore prohibitive.

Remark 1. *To solve* (5) *one possibility is to rely on traditional numerical methods, e.g., the finite element method. In particular, in the latter case, the semi-discretized form of the HJB equation reads as follows. Let* $V(\boldsymbol{y}, t)$ *be linearly approximated by a set of basis functions on* $\mathbb{R}^{|\boldsymbol{y}|}$*, i.e.,* $V(\boldsymbol{y}, t) \approx \sum_{j=1}^{|\boldsymbol{v}|} v_j(t)\phi_j(\boldsymbol{y})$*, in which* ϕ_j *is the* j*-th basis function, and* $\boldsymbol{v}(t) = \{v_1(t), \cdots, v_{|\boldsymbol{v}|}(t)\}^T$ *collects the expansion coefficients. Using the same test functions as the basis functions, the Galerkin scheme gives a semi-discretized form of the HJB equation as follows: for* $t \in [t_0, t_f]$ *and* $i = 1, \cdots, |\boldsymbol{v}|$*,*

$$\begin{aligned} -\sum_j \frac{dv_j(t)}{dt} \int \phi_i(\boldsymbol{y})\phi_j(\boldsymbol{y}) \, d\boldsymbol{y} = \\ \int \phi_i(\boldsymbol{y}) \min_{\boldsymbol{u}(t) \in U_{ad}} \left\{ L(\boldsymbol{y}, \boldsymbol{u}(t), t) + \boldsymbol{F}(\boldsymbol{y}, \boldsymbol{u}(t); \boldsymbol{\mu})^T \left(\sum_j v_j(t) \frac{d\phi_j(\boldsymbol{y})}{d\boldsymbol{y}} \right) \right\} \, d\boldsymbol{y} \,, \end{aligned} \tag{6}$$

which has to be solved for $\boldsymbol{v}(t)$*. We highlight that the solution of* (6) *entails several issues which need to be addressed. Indeed, the value function is expressed in terms of basis functions depending on the state solution; thus requiring* $\boldsymbol{y}$ *to be known at each time instance. Moreover, handling the right-hand side of* (6)*, that is the minimum in the variational formulation, requires special treatment as, for example, solving local optimization problems at quadrature points as in [47]. As a result, the HJB equation is not easily tractable and may become extremely computationally expensive for high-dimensional and large-time horizon control problems.*

[1] We refer, e.g., to the Picard–Lindelöf theorem [42].

2.2. From Dynamic Programming to Reinforcement Learning

Solving the HJB equation in (5) requires reformulating the problem in a time-discrete framework. Algorithms solving the fully-discretized HJB are typically referred to as dynamic programming (DP) approaches [3, 48]. DP methods can handle all kinds of hybrid systems, even with non-differentiable dynamics, and stochastic OC problems [49]. Examples of DP algorithms are policy iteration and value iteration [3]. DP utilizes the Markov Decision Process (MDP) as underlying mathematical framework in order to account for stochastic systems' dynamics and tackle a broader class of stochastic OC problems [48].

A MDP is a tuple $\langle Y, U_{\text{ad}}, \boldsymbol{T}, R \rangle$ where $Y \subset \mathbb{R}^{|\boldsymbol{y}|}$ is the set of observable states, $U_{\text{ad}} \subset \mathbb{R}^{|\boldsymbol{u}|}$ is the set of admissible actions, $\boldsymbol{T} : Y \times Y \times U_{\text{ad}} \longrightarrow [0,1]^{|\boldsymbol{y}|}$ such that $(\boldsymbol{y}_{k+1}, \boldsymbol{y}_k, \boldsymbol{u}_k) \longmapsto \boldsymbol{T}(\boldsymbol{y}_{k+1}, \boldsymbol{y}_k, \boldsymbol{u}_k)$ is the transition function, and $R : Y \times U_{\text{ad}} \longrightarrow \mathbb{R}$ such that $(\boldsymbol{y}_k, \boldsymbol{u}_k) \longmapsto R(\boldsymbol{y}_k, \boldsymbol{u}_k)$ denotes the reward function. The transition function $(\boldsymbol{y}_{k+1}, \boldsymbol{y}_k, \boldsymbol{u}_k) \longmapsto \boldsymbol{T}(\boldsymbol{y}_{k+1}, \boldsymbol{y}_k, \boldsymbol{u}_k)$ describes the probability of reaching state $\boldsymbol{y}_{k+1}$ from state $\boldsymbol{y}_k$ while taking action $\boldsymbol{u}_k$,

$$p(\mathrm{Y}_{k+1} = \boldsymbol{y}_{k+1} | \mathrm{Y}_k = \boldsymbol{y}_k, \mathrm{U}_k = \boldsymbol{u}_k)\,, \tag{7}$$

fulfilling

$$\sum_{\boldsymbol{y}_{k+1} \in Y} p(\mathrm{Y}_{k+1} = \boldsymbol{y}_{k+1} | \mathrm{Y}_k = \boldsymbol{y}_k, \mathrm{U}_k = \boldsymbol{u}_k) = 1, \qquad \forall \;\; \boldsymbol{y}_k \in Y, \boldsymbol{u}_k \in U_{\text{ad}}. \tag{8}$$

It is worth mentioning that a deterministic transition function $\boldsymbol{y}_{k+1} = \boldsymbol{T}(\boldsymbol{y}_k, \boldsymbol{u}_k)$ is a special case arising when $p(\boldsymbol{y}_{k+1} | \boldsymbol{y}_k, \boldsymbol{u}_k) = 1$ and that the reward function is the fully-discretized counterpart of the running cost $L(\cdot)$, but with opposite sign. Therefore, the value function in DP problems is typically written as a maximization problem over the possible controls rather than a minimization one:

$$V(\boldsymbol{y}_k) = \max_{\boldsymbol{u} \in U_{\text{ad}}} \mathbb{E}\big[R(\mathrm{Y}_k, \mathrm{U}_k) + V(\mathrm{Y}_{k+1}) | \mathrm{Y}_k = \boldsymbol{y}_k, \mathrm{U}_k = \boldsymbol{u}_k\big]\,, \tag{9}$$

where we indicate with $\mathbb{E}[\cdot]$ the expected value of a random variable, and $\mathrm{Y}_{k+1} \sim T(\boldsymbol{y}_{k+1}, \boldsymbol{y}_k, \boldsymbol{u}_k)$. Equation (9) can be seen as the fully-discretized HJB in stochastic settings.

The key idea of DP is to estimate the value function $V(\cdot)$ using the perfectly-known environment dynamics $R(\cdot)$ and $\boldsymbol{T}(\cdot)$ and then use it to structure the search for good control strategies. Despite their success, DP algorithms *(i)* require perfect knowledge of $R(\cdot)$ and $\boldsymbol{T}(\cdot)$ – despite they are, in many scenarios, often not known exactly or extremely expensive to compute, especially when the dimensionality of the state is very high – and *(ii)* are unpractical to use in problems with large number of states due to the need of solving the HJB for all possible states. These two drawbacks drastically limit the application of DP algorithms to complex and large-scale problems such as those arising in the OC of parametric PDEs.

Reinforcement learning (RL) [3] is a promising machine learning approach to solve sequential decision-making problems through a trial-and-error process. Unlike DP, RL does not try to solve the HJB for all possible states; rather, it aims at deriving OC laws from *(i)* measurements of the system – often referred to as observations, and *(ii)* reward samples, without direct knowledge of $\boldsymbol{T}(\cdot)$ and $R(\cdot)$ [3, 50]. In RL, we can identify two main entities: the *agent* and the *environment* (see Figure 1). The agent aims to find the best strategy to solve a given task by interacting with an unknown environment. Similarly to DP, the optimality of the strategy learned by the agent is defined by a task-dependent reward function. In particular, we can use RL to solve OC problems, such as the one in Equation (1), by learning an OC law from data without the need to explicitly solve the HJB equation (either in continuous or in discrete settings) for all possible states. Similarly to DP, we can rely on MDPs to mathematically formulate the RL problem. The goal of any RL algorithm is to find the optimal policy (control law) maximizing the expected cumulative return G_k:

$$G_k = r_k + \gamma r_{k+1} + \gamma^2 r_{k+2} + \cdots = \sum_{j=0}^{H} \gamma^j r_{k+j}\,, \tag{10}$$

where the control horizon[2] H is defined as $H = (t_f - t_0)/\Delta t$, the subscript k denotes the time-step, $r_k = R(\boldsymbol{y}_k, \boldsymbol{u}_k)$ is the instantaneous reward received by the agent at time-step k, and γ is a discount factor balancing the contribution of present and future rewards, where $0 \leq \gamma \leq 1$. It is worth mentioning that the

[2]The control horizon can be either finite or infinite.

expected return G_k is the analogous of the cost functional $J(\cdot)$ in Equation (2) in a fully-discretized and discounted setting.

Almost all RL algorithms revolve around estimating the value function without any knowledge of the true transition and reward functions, i.e., without explicitly solving the HJB for all possible states. Thus, the value function is learned from state-action-reward trajectories, i.e., from data. Starting from an initial estimate of the value function and of the optimal policy, RL algorithms iteratively improve these estimates until the value function and the optimal policy are found. We can rewrite the value function $V(\cdot)$ using the expected return G_k:

$$V(\boldsymbol{y}_k) = \mathbb{E}_{\pi^*}[G_k|\mathrm{Y}_k = \boldsymbol{y}_k], \tag{11}$$

where $\mathbb{E}_{\pi^*}[\cdot|\mathrm{Y}_k = \boldsymbol{y}_k]$ denotes the conditional expectation of a random variable if the agent follows the optimal policy π^* on a time-step of length H, given the starting value $\mathrm{Y}_k = \boldsymbol{y}_k$. In general, a control policy π can be either stochastic, i.e., $\pi : Y \times U_{\mathrm{ad}} \to [0,1]^{|\boldsymbol{u}|}$, or deterministic, i.e., $\pi : Y \to U_{\mathrm{ad}}$.

Similarly, we can define the action value function $Q : Y \times U_{\mathrm{ad}} \to \mathbb{R}$, as the value of taking action $\boldsymbol{u}_k$ at a certain state $\boldsymbol{y}_k$:

$$Q(\boldsymbol{y}_k, \boldsymbol{u}_k) = \mathbb{E}_{\pi^*}[G_k|\mathrm{Y}_k = \boldsymbol{y}_k, \mathrm{U}_k = \boldsymbol{u}_k]. \tag{12}$$

It is worth mentioning that there exists a direct relation between the value function $V(\boldsymbol{y}_k)$ and the action-value function $Q(\boldsymbol{y}_k, \boldsymbol{u}_k)$. In particular, we can write:

$$V(\boldsymbol{y}_k) = \max_{\boldsymbol{u}_k \in U_{\mathrm{ad}}} Q(\boldsymbol{y}_k, \boldsymbol{u}_k). \tag{13}$$

RL algorithms are usually classified as *model-based* or *model-free* methods [51]. In this context, the keyword *model* indicates whether the agent relies (model-based) or not (model-free) on an environment model, often built from the interaction data, to learn the value function and the policy. Another important distinction can be found between *online* and *offline* approaches. Online RL aims at learning the optimal policy while interacting with the environment. Conversely, offline RL aims to learn the policy offline given a fixed dataset of trajectories. While online methods better embody the interactive nature of RL, in many (safety-critical) applications it is not possible to apply random actions to explore the environment and offline approaches are preferred. Eventually, we can distinguish among *value-based*, *policy-based*, and *actor-critic* algorithms [3]. Value-based algorithms rely only on the estimation of the (action) value function and derive the optimal policy by greedily selecting the action with the highest value at each time-step. Examples of value-based algorithms are Q-learning [52] and its extensions relying on deep neural networks [11, 7, 12, 53]. Second, policy-based algorithms directly optimize the parameters of the policies with the aim of maximizing the return G_k via the policy gradient [3]. One of the first and most famous policy-based algorithms is REINFORCE [54]. Third, actor-critic algorithms learn value function and policy at the same time. The keyword *actor* refers to the policy acting on the environment, while *critic* refers to the value function assessing the quality of the policy. Examples are deep deterministic policy gradient (DDPG) [55], proximal policy optimization (PPO) [56], and soft actor-critic (SAC) [57]. Eventually, we can identify *on-policy* and *off-policy* methods. On-policy approaches utilize the same policy for exploration and exploitation. Therefore, they often optimize a stochastic policy that can either explore the environment, but also exploit good rewards. An example of on-policy approach is PPO. On the other side, off-policy algorithms maintain two distinct policies for exploration and exploitation, making it possible to reuse the interaction data to update the models. Examples of off-policy algorithms are DDPG and SAC.

2.2.1. Twin-Delayed Deep Deterministic Policy Gradient

In our numerical experiments, we utilize a model-free, online, off-policy, and actor-critic approach, namely Twin-Delayed Deep Deterministic Policy Gradient (TD3) [58]. However, our method can be directly used by any other RL algorithm. TD3 learns a deterministic policy $\pi(\cdot)$, i.e., the actor, and the action-value function $Q(\cdot)$, i.e., the critic. The actor and the critic are parametrized by means of two DNNs of parameters $\boldsymbol{\theta}_Q$ and $\boldsymbol{\theta}_\pi$, respectively. We indicate the parametrized policy with $\pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi)$ and the action-value function with $Q(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_Q)$. TD3 can handle continuous[3] state and action spaces, making it a suitable candidate for controlling parametric PDEs using smooth control strategies.

[3]Space and time are discretized but each variable can assume any continuous value in the admissible ranges.

To learn the optimal action-value function[4], TD3 relies on temporal-difference (TD) learning [3]. In particular, starting from the definition of the action-value function in Equation (12), we can write:

$$\begin{aligned} Q(\boldsymbol{y}_k, \boldsymbol{u}_k) &= \mathbb{E}_\pi[\Sigma_{j=0}^{H} \gamma^j r_{k+j} | \mathrm{Y}_k = \boldsymbol{y}_k, \mathrm{U}_k = \boldsymbol{u}_k] \\ &= \mathbb{E}_\pi[r_k + \Sigma_{j=1}^{H} \gamma^j r_{k+j} | \mathrm{Y}_k = \boldsymbol{y}_k, \mathrm{U}_k = \boldsymbol{u}_k] \\ &= \mathbb{E}_\pi[r_k + \gamma \max_{\boldsymbol{u}_k \in U_{\mathrm{ad}}} Q(\mathrm{Y}_{k+1}, \cdot) | \mathrm{Y}_k = \boldsymbol{y}_k, \mathrm{U}_k = \boldsymbol{u}_k], \end{aligned} \tag{14}$$

where we use *bootstrapping* to express the value of a state-action pair $Q(\boldsymbol{y}_k, \boldsymbol{u}_k)$ by using in the update rule (15) the estimate of the action-value function $Q(\boldsymbol{y}_{k+1}, \cdot)$ instead of observed returns from complete trajectories. The expectation is now with respect to a generic policy π, which may be far from the optimal policy π^*. Using TD learning, we can iteratively update the estimate of the action-value function as:

$$Q(\boldsymbol{y}_k, \boldsymbol{u}_k) \leftarrow Q(\boldsymbol{y}_k, \boldsymbol{u}_k) + \alpha \Big(r_k + \gamma \max_{\boldsymbol{u}_k \in U_{\mathrm{ad}}} Q(\boldsymbol{y}_{k+1}, \cdot) - Q(\boldsymbol{y}_k, \boldsymbol{u}_k) \Big), \tag{15}$$

where α is the learning rate. However, in the case of a continuous action space the update rule in Equation (15) cannot be used directly. Indeed, while for discrete action spaces evaluating the maximum of the Q-value for all the possible actions is straightforward, for continuous actions the bootstrap of the target Q-value would require solving an (expensive) optimization problem over the entire action space. Therefore, for continuous actions the following update rule is commonly used:

$$Q(\boldsymbol{y}_k, \boldsymbol{u}_k) \leftarrow Q(\boldsymbol{y}_k, \boldsymbol{u}_k) + \alpha \Big(r_k + \gamma Q(\boldsymbol{y}_{k+1}, \boldsymbol{u}_{k+1}) - Q(\boldsymbol{y}_k, \boldsymbol{u}_k) \Big), \tag{16}$$

where $\boldsymbol{u}_{k+1} = \pi(\boldsymbol{y}_{k+1})$ is selected accordingly to the current estimate of the "optimal" policy.

TD3 relies on a memory buffer $\mathcal{D}$ to store the interaction data $(\boldsymbol{y}_k, \boldsymbol{u}_k, r_k, \boldsymbol{y}_{k+1})$ for all time-steps k. Given a randomly-sampled batch of interaction tuples, we can employ Equation (16) as a loss function for updating the parameters $\boldsymbol{\theta}_Q$ of the action-value function $Q(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_Q)$ as:

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}_Q) &= \mathbb{E}_{\boldsymbol{y}_k, \boldsymbol{u}_k, \boldsymbol{y}_{k+1}, r_k \sim \mathcal{D}}[(r_k + \gamma \bar{Q}(\boldsymbol{y}_{k+1}, \boldsymbol{u}_{k+1}; \boldsymbol{\theta}_{\bar{Q}}) - Q(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_Q))^2] \\ &= \mathbb{E}_{\boldsymbol{y}_k, \boldsymbol{u}_k, \boldsymbol{y}_{k+1}, r_k \sim \mathcal{D}}[(\underbrace{r_k + \gamma \bar{Q}(\boldsymbol{y}_{k+1}, \bar{\pi}(\boldsymbol{y}_{k+1}; \boldsymbol{\theta}_{\bar{\pi}}) + \boldsymbol{\epsilon}; \boldsymbol{\theta}_{\bar{Q}})}_{\text{target value}} - Q(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_Q))^2], \end{aligned} \tag{17}$$

where the so-called target networks $\bar{Q}(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{\bar{Q}})$ and $\bar{\pi}(\boldsymbol{y}_k; \boldsymbol{\theta}_{\bar{\pi}})$ are copies of $Q(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_Q)$ and $\pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi)$, respectively, with frozen parameters, i.e., they are not updated in the backpropagation step to improve the stability of the training. We indicate with $\boldsymbol{\epsilon} \sim \mathrm{clip}(\mathcal{N}(\boldsymbol{0}, \bar{\boldsymbol{\sigma}}), -c, c)$ the noise added to estimate the action value in the interval $[-c, c]$ around the target action. To reduce the problem of overestimation of the target Q-values, TD3 estimates two independent action-value functions, namely $Q_1(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{Q_1})$ and $Q_2(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{Q_2})$, and two target action-value functions $\bar{Q}_1(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{\bar{Q}_1})$ and $\bar{Q}_2(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{\bar{Q}_2})$, and computes the target value for regression (see (17)) as:

$$\underbrace{r_k + \gamma \min_{i=1,2} \bar{Q}_i(\boldsymbol{y}_{k+1}, \boldsymbol{u}_{k+1}; \boldsymbol{\theta}_{\bar{Q}_i})}_{\text{target value}}. \tag{18}$$

The action-value function $Q_1(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{Q_1})$ is used to update the parameters of the deterministic policy $\pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi)$ according to the deterministic policy gradient theorem [59]. In particular, the gradient of the critic guides the improvements of the actor and the policy parameters are updated to ascend the action-value function:

$$\mathcal{L}(\boldsymbol{\theta}_\pi) = \mathbb{E}_{\boldsymbol{y}_k \sim \mathcal{D}}[-\nabla_{\boldsymbol{u}_k} Q_1(\boldsymbol{y}_k, \pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi); \boldsymbol{\theta}_{Q_1})]. \tag{19}$$

[4]Because we do not directly solve the HJB equation to obtain the value function, its initial estimate may be far from the true one. Therefore, we highlight the keyword "optimal" to distinguish the true value function from the estimated one.

The target networks, parametrized by $\boldsymbol{\theta}_{\bar{Q}_1}$, $\boldsymbol{\theta}_{\bar{Q}_2}$, and $\boldsymbol{\theta}_{\bar{\pi}}$, respectively, are updated with a slower frequency than the actor and the critic according to:

$$\begin{aligned} \boldsymbol{\theta}_{\bar{Q}_1} &= \rho\boldsymbol{\theta}_{Q_1} + (1-\rho)\boldsymbol{\theta}_{\bar{Q}_1}\,, \\ \boldsymbol{\theta}_{\bar{Q}_2} &= \rho\boldsymbol{\theta}_{Q_2} + (1-\rho)\boldsymbol{\theta}_{\bar{Q}_1}\,, \\ \boldsymbol{\theta}_{\bar{\pi}} &= \rho\boldsymbol{\theta}_{\pi} + (1-\rho)\boldsymbol{\theta}_{\bar{\pi}}\,, \end{aligned} \tag{20}$$

where ρ is a constant factor determining the speed of the updates of the target parameters.

2.3. Hypernetworks

A hypernetwork [36] is a neural network (NN) that generates the weights and biases of another NN, often referred to as main or primary network. Formally, a hypernetwork $h : Z \subset \mathbb{R}^{|\boldsymbol{z}|} \to \mathbb{R}^{|\boldsymbol{\theta}_f|}$ learns the parameters $\boldsymbol{\theta}_f$ of the main network $f : X \subset \mathbb{R}^{|\boldsymbol{x}|} \to W \subset \mathbb{R}^{|\boldsymbol{w}|}$. If we consider a standard supervised learning regression task, and we assume the availability of a dataset of N input-output pairs $[(\boldsymbol{x}^{(1)}, \boldsymbol{w}^{(1)}), \cdots, (\boldsymbol{x}^{(N)}, \boldsymbol{w}^{(N)})]$ and hypernetwork inputs $[\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(N)}]$, we can write:

$$\begin{aligned} \boldsymbol{\theta}_f &= h(\boldsymbol{z}^{(i)}; \boldsymbol{\theta}_h)\,, \\ \hat{\boldsymbol{w}}^{(i)} &= f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_f)\,, \end{aligned} \tag{21}$$

where $\boldsymbol{z}$, often called context vector, can be a task-conditioned, data-conditioned, or noise-conditioned input [37], and $\boldsymbol{\theta}_h$ denotes the set of parameters of the hypernetwork $h(\cdot)$. The parameters of the two networks $\boldsymbol{\theta}_h$ and $\boldsymbol{\theta}_f$ can be updated jointly by minimizing a prescribed loss function. In the specific case of a regression task, the loss function is usually the mean-squared error between the target and predicted values:

$$\mathcal{L}(\boldsymbol{\theta}_f, \boldsymbol{\theta}_h) = \sum_{i=1}^{N} ||\boldsymbol{w}^{(i)} - \hat{\boldsymbol{w}}^{(i)}||_2^2. \tag{22}$$

3. Methodology for HypeRL

Given all the elements introduced in the previous section, we are now ready to set our proposed strategy to address OC of parametric PDEs through RL.

3.1. Problem Settings

We cast the parametric PDE-constrained OC problem, introduced in Equation (1), as RL problem, where an agent, i.e., the controller, aims to learn the OC strategy by interacting with an unknown environment, i.e., the PDE state. The RL environment is defined by a transition and reward functions:

$$\begin{aligned} \boldsymbol{y}_{k+1} &= \boldsymbol{T}(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\mu})\,, \\ r_k &= R(\boldsymbol{y}_k, \boldsymbol{u}_k)\,, \end{aligned} \tag{23}$$

where the transition function is assumed to be deterministic and dependent on the PDE parameters $\boldsymbol{\mu}$. The transition function corresponds to the fully-discretized form of the state equation (see Equation (1)) obtained by introducing a suitable time-integration scheme over a partition of $[t_0, t_f]$ made by N_t time-steps $\{t_k\}_{k=0}^{N_t}$ such that the step size is $\Delta t = (t_f - t_0)/N_t$. For example, by using an explicit Runge-Kutta scheme, we obtain:

$$\boldsymbol{y}_{k+1} = \underbrace{\boldsymbol{y}_k + \Delta t \Phi(t_k, \boldsymbol{y}_k, \boldsymbol{u}_k; \Delta t, \boldsymbol{F}, \boldsymbol{\mu})}_{:= T(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\mu})}, \qquad k = 0, \ldots, N_t - 1, \tag{24}$$

where $\boldsymbol{y}_k \approx \boldsymbol{y}(t_k)$ and $\boldsymbol{u}_k \approx \boldsymbol{u}(t_k)$, and Φ denotes the integration method's increment function related to the state equation. The reward function corresponds to the fully-discretized counterpart of the running cost in Equation (3) with opposite sign. Starting from (3), in the following, our goal is to steer the PDE state to a reference state while tracking a reference control signal. Therefore, we utilize the following reward function:

$$R(\boldsymbol{y}_k, \boldsymbol{u}_k) = -\frac{1}{2}||\boldsymbol{y}_k - \boldsymbol{y}_{\text{ref}}||_2^2 - \frac{\alpha}{2}||\boldsymbol{u}_k - \boldsymbol{u}_{\text{ref}}||_2^2 = -\frac{1}{2}c_1 - \frac{\alpha}{2}c_2\,, \tag{25}$$

where $\boldsymbol{y}_{\text{ref}}$ and $\boldsymbol{u}_{\text{ref}}$ indicate the reference values for the state and the control input, and α is a scalar positive coefficient balancing the contribution of the two terms.

With reference to Algorithm 1, the agent-environment interaction scheme is organized in two loops: the outer loop indicates the training episode; instead, the inner loop refers to the number of time-steps for each episode. At the beginning of each new episode we sample a random initial condition and a PDE parameters value $\boldsymbol{\mu}$ to obtain the initial state. The agent utilizes the state at the current time-step and the PDE parameters to select a control input. The control input is fed to the transition and reward models to obtain the state at the next time-step and the reward. The tuples $(\boldsymbol{y}_k, \boldsymbol{u}_k, r_k, \boldsymbol{y}_{k+1}, \boldsymbol{\mu})$ collected at each time-step k of the interaction are used to train the policy (and value function) deep NN (DNN) to maximize the expected cumulative reward at each episode.

Algorithm 1 Episodic RL for control of parametric PDEs

```
Initialize policy and value function DNN parameters θ_π and θ_Q
for e = 1 : E_max do
    Sample an initial condition and PDE parameter μ
    Get initial measurement y_k
    for k = 1 : K_max do
        Sample action from a policy u_k ~ π(y_k, μ; θ_π) + ε, where ε ~ N(0, σ)
        Observe reward r_k = R(y_k, u_k) and new state y_{k+1} = T(y_k, u_k; μ)
        if train models then
            Update policy and value function using the tuple (y_k, u_k, r_k, y_{k+1}, μ)
        end if
    end for
end for
```

Differently from the majority of the literature on RL for OC of PDEs, we focus on devising a control strategy that is adaptable to changes of the systems' dynamics deriving from variations of known parameters $\boldsymbol{\mu}$. We assume the agent to be able to observe the PDE state and the parameters $\boldsymbol{\mu}$. In this setting, we present a novel RL framework that can efficiently learn control policies for parametric PDEs from limited samples and that can generalize to new, unseen instances of the PDE parameters $\boldsymbol{\mu}$ (see Figure 1). In contrast with the widely-used concatenation of information in the agent's state, to enhance the sample-efficiency and generalization capabilities of RL agents, we propose a parameter-informed HypeRL architecture relying on hypernetworks (see Section 2.3). Hypernetworks allow us to express the weights and biases of the value function and policy as functions of the PDE parameters $\boldsymbol{\mu}$ (see Figure 2). This new paradigm for encoding the information of the PDE parameters drastically improves the performance of the RL agent in terms of total cumulative reward, sample efficiency, and generalization with respect to traditional RL approaches.

3.2. HypeRL TD3

In this work, we enhance the TD3 algorithm (see Section 2.2.1) with hypernetworks. However, our method can be easily and directly applied to other RL algorithms, such as PPO and SAC. Analogously to the TD3 algorithm, we rely on the estimation of two action-value functions $Q_1(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{Q_1})$ and $Q_2(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{Q_2})$ and a policy $\pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi)$ by means of DNNs. However, the parameters of the main networks are now learned using three hypernetworks $h_{Q_1}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{Q_1}})$, $h_{Q_2}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{Q_2}})$, and $h_\pi(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_\pi})$:

$$\begin{aligned} \boldsymbol{\theta}_{Q_1} &= h_{Q_1}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{Q_1}}), \\ \boldsymbol{\theta}_{Q_2} &= h_{Q_2}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{Q_2}}), \\ \boldsymbol{\theta}_\pi &= h_\pi(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_\pi}), \end{aligned} \tag{26}$$

where $\boldsymbol{z}_k$ indicates the hypernetwork input. With reference to Figure 2, we propose two variations of HypeRL: (i) we utilize the PDE parameter vector $\boldsymbol{\mu}$ as input to the hypernetworks, i.e., $\boldsymbol{z}_k = \boldsymbol{\mu}$ and (ii) we utilize state and PDE parameters as hypernetwork input, i.e., $\boldsymbol{z}_k = [\boldsymbol{y}_k, \boldsymbol{\mu}]$. In this way, we are able to learn a representation of PDE parameters or of states and PDE parameter vector and encode this information in the

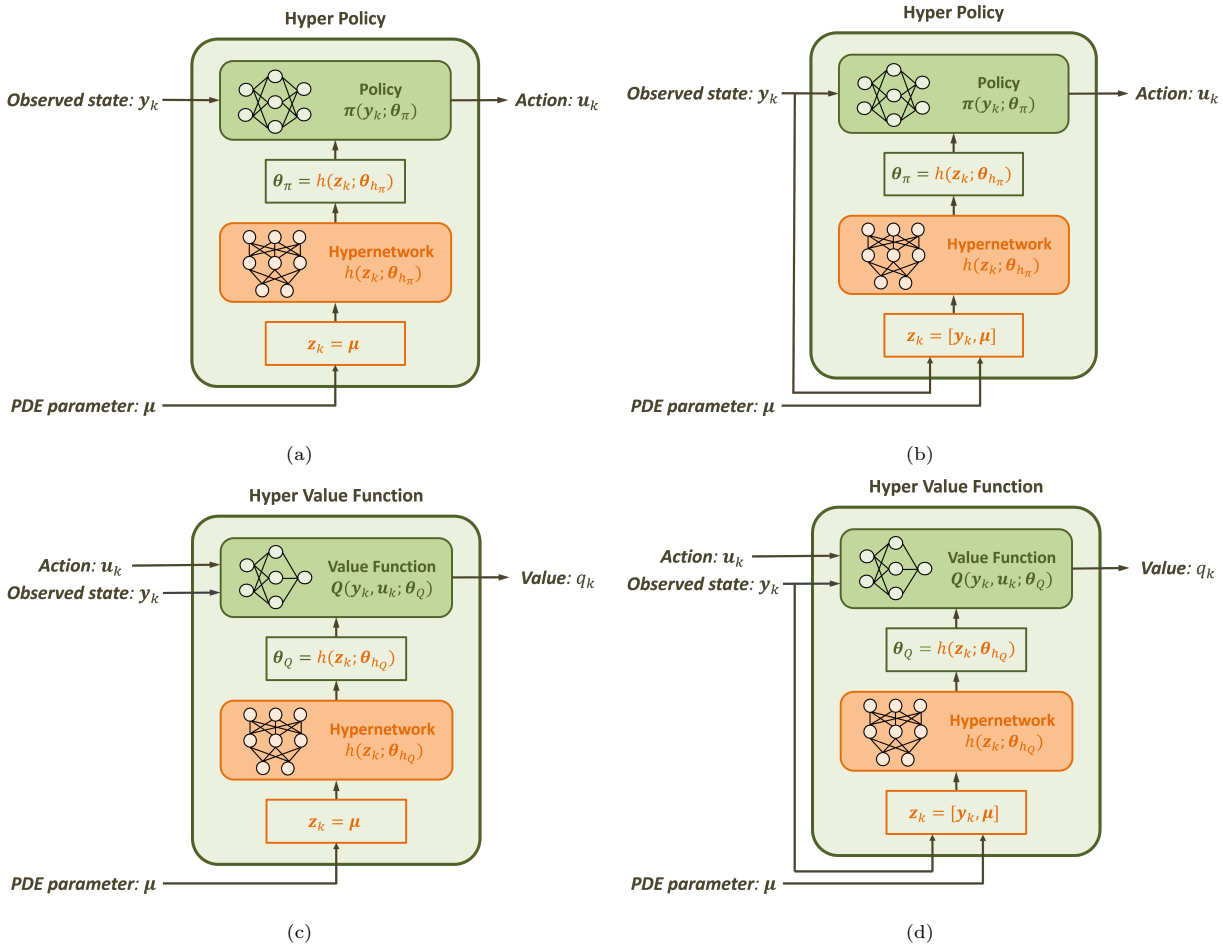


Figure 2: Hyper policy and hyper value function architectures. In Figure 2a and 2c only the PDE parameters are used as input to the hypernetwork, i.e., $\boldsymbol{z}_k = \boldsymbol{\mu}$, while in Figure 2b and 2d the PDE state and parameters are used as input to the hypernetwork, i.e., $\boldsymbol{z}_k = [\boldsymbol{y}_k, \boldsymbol{\mu}]$.

parameters of the main networks. We can then obtain the actions from the policy as:

$$\begin{aligned} \boldsymbol{\theta}_\pi &= h_\pi(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_\pi}) \,, \\ \boldsymbol{u}_k &= \pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi) \,, \end{aligned} \tag{27}$$

and analogously, we can obtain the Q-values as:

$$\begin{aligned} \boldsymbol{\theta}_{Q_i} &= h_{Q_i}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{Q_i}}) \,, \\ q_{i,t} &= Q_i(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{Q_i}) \,, \end{aligned} \tag{28}$$

where $q_{i,t}$ is the predicted Q-value by the action-value function $Q_i(\cdot, \cdot)$.

The hypernetwork parameters are jointly optimized with the main network parameters by simply allowing the gradient of the TD3 training objectives[5] (see in Equation (17) and (19)) to flow through the hypernetworks:

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}_{Q_i}, \boldsymbol{\theta}_{h_{Q_i}}) &= \mathbb{E}_{\boldsymbol{y}_k, \boldsymbol{u}_k, r_k, \boldsymbol{y}_{k+1} \sim \mathcal{D}}[(r_k + \gamma \min_{i=1,2} \bar{Q}_i(\boldsymbol{y}_{k+1}, \boldsymbol{u}_{k+1}; \boldsymbol{\theta}_{\bar{Q}_i}) - Q_i(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{Q_i}))^2] \,, \\ \mathcal{L}(\boldsymbol{\theta}_\pi, \boldsymbol{\theta}_{h_\pi}) &= \mathbb{E}_{\boldsymbol{y}_k \sim \mathcal{D}}[-\nabla_{\boldsymbol{u}_k} Q_1(\boldsymbol{y}_k, \pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi); \boldsymbol{\theta}_{Q_1})]. \end{aligned} \tag{29}$$

[5]We do not need to change the TD3 working principles and losses.

In Algorithm 2, we show the HypeRL TD3 pseudo code, where we highlight in blue the changes to the original TD3 algorithm. It is worth highlighting that our method can be plugged in any RL algorithm as we did not change their working principles.

Algorithm 2 Parameter-Informed HypeRL TD3

Initialize $Q_1(\boldsymbol{y}, \boldsymbol{u}; \boldsymbol{\theta}_{Q_1})$, $Q_2(\boldsymbol{y}, \boldsymbol{u}; \boldsymbol{\theta}_{Q_2})$, and $\pi(\boldsymbol{y}; \boldsymbol{\theta}_\pi)$
Initialize hypernetworks $h_{Q_1}(\boldsymbol{z}; \boldsymbol{\theta}_{h_{Q_1}})$, $h_{Q_2}(\boldsymbol{z}; \boldsymbol{\theta}_{h_{Q_2}})$, and $h_\pi(\boldsymbol{z}; \boldsymbol{\theta}_{h_\pi})$ with random parameters $\boldsymbol{\theta}_{h_{Q_1}}, \boldsymbol{\theta}_{h_{Q_2}}, \boldsymbol{\theta}_{h_\pi}$
Initialize target hypernetworks $\boldsymbol{\theta}_{h_{\bar{Q}_1}} \leftarrow \boldsymbol{\theta}_{h_{Q_1}}$, $\boldsymbol{\theta}_{h_{\bar{Q}_2}} \leftarrow \boldsymbol{\theta}_{h_{Q_2}}$, $\boldsymbol{\theta}_{h_{\bar{\pi}}} \leftarrow \boldsymbol{\theta}_{h_\pi}$
Initialize memory buffer $\mathcal{D}$
for $e = 1 : E_{\max}$ **do**
  Initialize the system and get initial measurement $\boldsymbol{y}_k, \boldsymbol{\mu}$
  for $t = 1 : T_{\max}$ **do**
    Set $\boldsymbol{z}_k = \boldsymbol{\mu}$ or $\boldsymbol{z}_k = [\boldsymbol{y}_k, \boldsymbol{\mu}]$
    Sample policy parameters $\boldsymbol{\theta}_\pi = h_\pi(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_\pi})$
    Sample action $\boldsymbol{u}_k \sim \pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi) + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\sigma})$
    Observe reward r_k and new state $\boldsymbol{y}_{k+1}$
    Store tuple $(\boldsymbol{y}_k, \boldsymbol{u}_k, r_k, \boldsymbol{y}_{k+1}, \boldsymbol{\mu})$ in $\mathcal{D}$

    if train models **then**
      Sample mini-batch $(\boldsymbol{y}_k, \boldsymbol{u}, r, \boldsymbol{y}_{k+1}, \boldsymbol{\mu})$ from $\mathcal{D}$
      Set $\boldsymbol{z}_{k+1} = \boldsymbol{\mu}$ or $\boldsymbol{z}_{k+1} = [\boldsymbol{y}_{k+1}, \boldsymbol{\mu}]$
      Sample target policy parameters $\boldsymbol{\theta}_{\bar{\pi}} = h_{\bar{\pi}}(\boldsymbol{z}_{k+1}; \boldsymbol{\theta}_{h_{\bar{\pi}}})$
      $\boldsymbol{u}_{k+1} \leftarrow \bar{\pi}(\boldsymbol{y}_{k+1}; \boldsymbol{\theta}_{\bar{\pi}}) + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \text{clip}(\mathcal{N}(\boldsymbol{0}, \bar{\boldsymbol{\sigma}}), -c, c)$
      Sample target value functions parameters $\boldsymbol{\theta}_{\bar{Q}_1} = h_{\bar{Q}_1}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{\bar{Q}_1}})$ and $\boldsymbol{\theta}_{\bar{Q}_2} = h_{\bar{Q}_2}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{\bar{Q}_2}})$
      $q^k \leftarrow r_k + \gamma \min_{i=1,2} \bar{Q}(\boldsymbol{y}_{k+1}, \boldsymbol{u}_{k+1}; \boldsymbol{\theta}_{\bar{Q}_i})$
      Sample value functions parameters $\boldsymbol{\theta}_{Q_1} = h_{\bar{Q}_1}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{Q_1}})$ and $\boldsymbol{\theta}_{Q_2} = h_{Q_2}(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_{Q_2}})$
      Update critic parameters and hypernetworks parameters according:
      $\mathcal{L}(\boldsymbol{\theta}_{Q_i}, \boldsymbol{\theta}_{h_{Q_i}}) = \mathbb{E}_{\boldsymbol{y}_k, \boldsymbol{u}_k, r_k, \boldsymbol{y}_{k+1} \sim \mathcal{D}}[(q_k - Q_i(\boldsymbol{y}_k, \boldsymbol{u}_k; \boldsymbol{\theta}_{Q_i}))^2]$ with $i \in \{1, 2\}$
      if train actor **then**
        Sample policy parameters $\boldsymbol{\theta}_\pi = h_\pi(\boldsymbol{z}_k; \boldsymbol{\theta}_{h_\pi})$
        Update policy parameters and hypernetworks parameters according to:
        $\mathcal{L}(\boldsymbol{\theta}_\pi, \boldsymbol{\theta}_{h_\pi}) = \mathbb{E}_{\boldsymbol{y}_k \sim \mathcal{D}}[-\nabla_{\boldsymbol{u}_k} Q_1(\boldsymbol{y}_k, \pi(\boldsymbol{y}_k; \boldsymbol{\theta}_\pi); \boldsymbol{\theta}_{Q_1})$
        Update target networks by updating the hypernetworks parameters:
        $\boldsymbol{\theta}_{h_{\bar{Q}_1}} = \rho \boldsymbol{\theta}_{h_{Q_1}} + (1 - \rho) \boldsymbol{\theta}_{h_{\bar{Q}_1}}$
        $\boldsymbol{\theta}_{h_{\bar{Q}_2}} = \rho \boldsymbol{\theta}_{h_{Q_2}} + (1 - \rho) \boldsymbol{\theta}_{h_{\bar{Q}_2}}$
        $\boldsymbol{\theta}_{h_{\bar{\pi}}} = \rho \boldsymbol{\theta}_{h_\pi} + (1 - \rho) \boldsymbol{\theta}_{h_{\bar{\pi}}}$
      end if

    end if
  end for
end for

4. Numerical Experiments

We validate our proposed approach on two control baselines, namely (i) a parametric Kuramoto-Sivashinsky PDE with distributed in-domain actuators, and (ii) a 2D Navier-Stokes equation with boundary control.

4.1. 1D Kuramoto-Sivashinsky equation

The Kuramoto-Sivashinsky (KS) equation is a nonlinear 1D PDE describing pattern and instability in fluid dynamics, plasma physics, and combustion, e.g., the diffusive-thermal instabilities in a laminar flame front [60]. Similarly to [25, 27], we write the KS PDE with state $y(x, t) = y$ and forcing term $u(x, t) = u$

with the addition of a parametric cosine term, breaking the spatial symmetries of the equation and making the search for the optimal control policy more challenging:

$$\begin{aligned} \frac{\partial y}{\partial t} + y\frac{\partial y}{\partial x} + \frac{\partial^2 y}{\partial x^2} + \frac{\partial^4 y}{\partial x^4} + \mu \cos\left(\frac{4\pi x}{L}\right) &= u\,, \\ u &= \sum_{i=1}^{N_a} a_i \psi(x, m_i)\,, \\ \psi(x, m_i) &= \frac{1}{2}\exp\left(-\left(\frac{x - m_i}{\sigma}\right)^2\right), \end{aligned} \tag{30}$$

where u is the control input function with $a_i \in [-1, 1]$, $\psi(x, c_i)$ is a Gaussian kernel of mean c_i and standard deviation $\sigma = 0.8$, $\mu \in [-0.25, 0.25]$ is the parameter of interest of the system, and $\mathcal{D} = [0, 22]$ is the spatial domain with periodic boundary conditions. Small values of μ generates chaotic solutions, while large values of μ periodic solutions, making it a very challenging test case and requiring strong generalization capabilities of the control policies. To numerically solve the PDE, we discretize the spatial domain with $N_x = 64$. We assume to have $N_a = 8$ equally-spaced actuators. The state of the agent is set equal to the (discretized) PDE state (although we are not limited to that). Similarly to [25, 27], we utilize the reward function (see Equation (25)) with $\boldsymbol{y}_{\text{ref}} = \mathbf{0}$, $\boldsymbol{u}_{ref} = \mathbf{0}$, and $\alpha = 0.1$. The choice of $\alpha = 0.1$ is dictated by the need for balancing the contribution of the state cost c_1 and the action cost c_2. In particular, in our experiments we prioritize steering the system to the reference state over the minimization of the injected energy. We train the control policies by randomly sampling a value of the parameter μ at the beginning of each training episode $\in [-0.225, -0.2, -0.175, \ldots, -0.05, -0.025, 0.0, 0.025, 0.05, \ldots, 0.175, 0.2, 0.225]$. To test the generalization abilities of the policies, we evaluate the agents on unseen and randomly-sampled parameters $\in [-0.25, 0.25]$. We compare our two variants of HypeRL TD3 agent (see Section 3) with (i) the standard TD3 agent with state $\boldsymbol{y}_k$ augmented by the parameter μ, and (ii) the TD3 algorithms without access to the parameter μ.

In Table 1 and 2, we show the training and evaluation rewards collected by the different agents over 5 different random seeds, where we highlight in bold the highest scores in three different phases of the training/evaluation. We consider the average cumulative reward of 5 independent runs over the whole training/evaluation episodes, the average cumulative rewards collected after 500/10 episodes until the end, and the average cumulative rewards collected after 1000/20 episodes until the end. The evaluation is performed by randomly-sampling 10 different instances of the parameter μ and then record the agents performance.

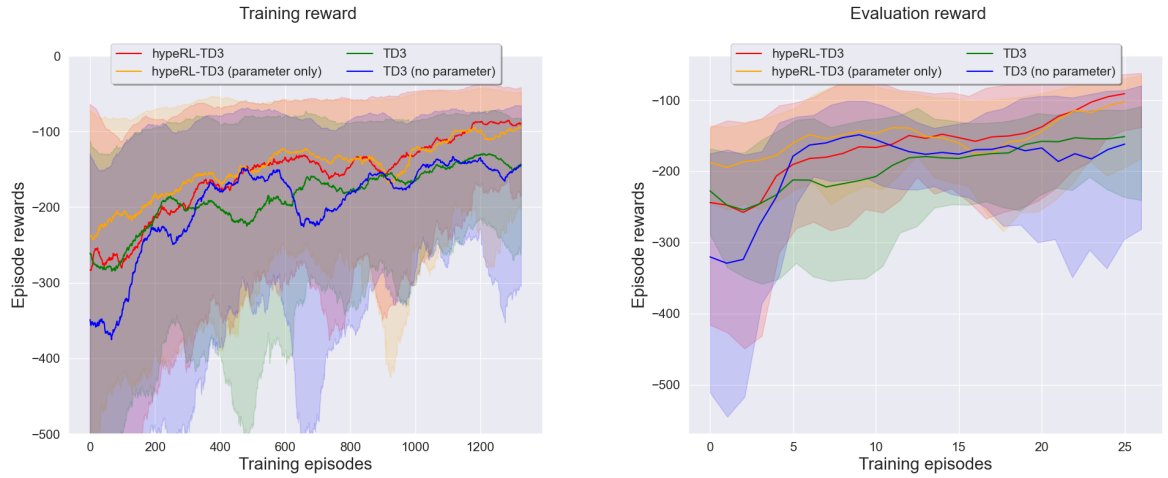
Cumulative reward	mean ± std	mean ± std (> 500 ep.)	mean ± std (> 1000 ep.)
HypeRL-TD3	-160.06 ± 81.35	-130.56 ± 53.71	$\mathbf{-112.66} \pm 48.29$
HypeRL-TD3 (μ only)	$\mathbf{-149.09} \pm 79.01$	$\mathbf{-129.65} \pm 69.19$	-118.21 ± 76.41
TD3	-186.59 ± 72.56	-168.28 ± 61.39	-148.12 ± 42.04
TD3 (no μ)	-197.98 ± 97.52	-165.62 ± 63.06	-150.08 ± 49.58

Table 1: Mean and standard deviation of the cumulative reward over training collected by the different algorithms. The results report the average performance over 5 different random seeds.

Cumulative reward	mean ± std	mean ± std (> 10 ep.)	mean ± std (> 20 ep.)
HypeRL-TD3	-163.68 ± 62.35	$\mathbf{-136.95} \pm 37.63$	$\mathbf{-114.28} \pm 32.70$
HypeRL-TD3 (μ only)	$\mathbf{-150.09} \pm 37.40$	-138.21 ± 34.84	-122.98 ± 37.88
TD3	-189.67 ± 43.88	-174.59 ± 30.75	-154.58 ± 17.97
TD3 (no μ)	-193.29 ± 77.11	-165.16 ± 32.39	-164.49 ± 42.13

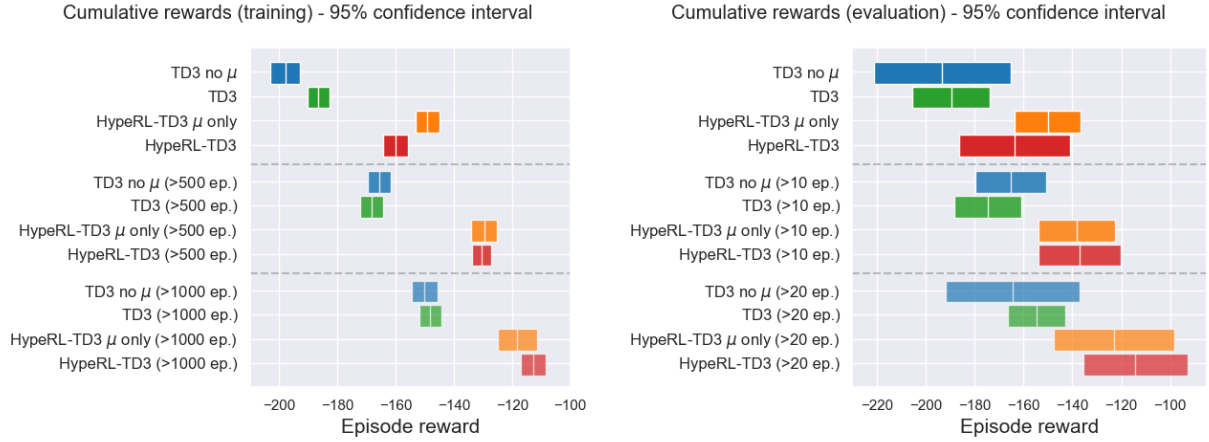
Table 2: Mean and standard deviation of the cumulative reward over evaluation collected by the different algorithms. The results report the average performance over 5 different random seeds.

The results show that the hypernetwork-based algorithms, namely HypeRL-TD3 and HypeRL-TD3 (μ only), outperform the TD3 agents not only in later stages of the training but also in early ones, showing that the way the information about the PDE parameter is encoded is crucial for sample efficiency and generalization of



(a) (b)

Figure 3: Training and evaluation results. The solid line represents the mean and the shaded area the standard deviation over 5 different random seeds.



(a) (b)

Figure 4: 95% confidence intervals of the training and evaluation reward for the different phases of training and evaluation. This metric, suggested in [61], allows to assess the reliability of the results accounting for the stochastic nature of the RL experiments.

the RL algorithms. HypeRL-TD3 (μ only) is the algorithm capable of achieving the highest average training rewards, followed by HypeRL-TD3, TD3, and TD3 (no μ). However, if we only consider later stages of the training HypeRL-TD3 is the best performing agent. We see similar trends in the evaluation results.

In Figure 3 and Figure B.10, we show the mean and the standard deviation of the cumulative reward, state cost, and action cost during training and evaluation, respectively. It is possible to notice the superior performance of the hypernetwork-based agents in early and later stages of the training and during evaluation with unseen values of the parameter μ. Additionally, due to the stochastic nature of the experiments and to improve the reliability of the results, we compute and show in Figure 4 the 95% confidence intervals, as suggested in [61]. The 95% confidence intervals are aligned and consistent with the average training and evaluation results and again highlights the higher sample efficiency and generalization capabilities of the hypeRL agents.

Eventually, in Figure 5 we show examples of controlled solution of the KS PDE. The KS PDE is evolved for 100 time-steps before turning the controllers ON. In particular, we highlights two different test cases: (a) interpolation regime with $\mu = 0.062$ (unseen but withing the training range), and (b) extrapolation regime

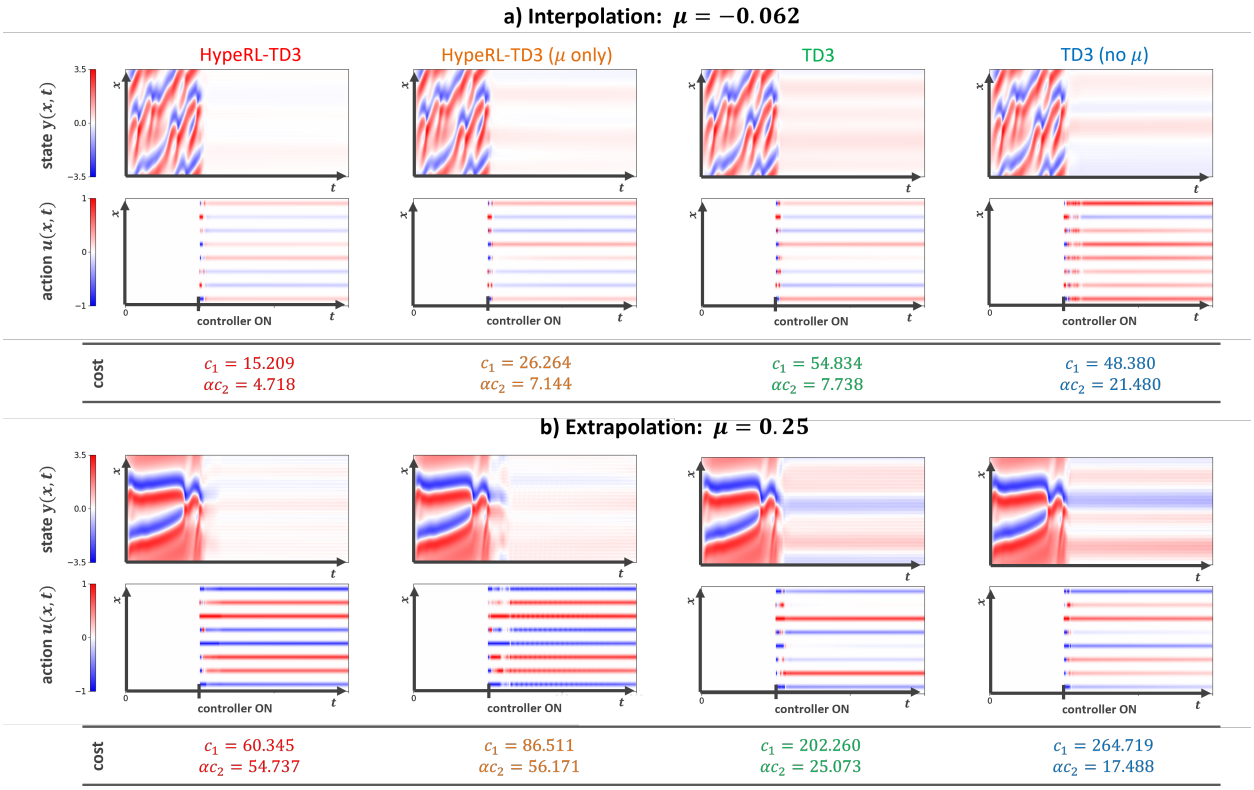


Figure 5: Controlled solutions for $\mu = -0.062$ and $\mu = 0.25$.

with $\mu = 0.25$ (unseen and outside the training range). We also report the state and action costs of the solutions after enabling the controllers. The hypeRL agents are capable of quickly steering the state of the KS PDE very close to the reference value with very little control effort in interpolation and extrapolation regimes, even when the PDE dynamics is very different. On the other side, the TD3 agents (with and without access to the PDE parameter μ) struggle to achieve good state tracking, especially in extrapolation regimes. In both interpolation and extrapolation the hypeRL agents improve the results with respect to the TD3 agents, showing that learning the main networks weights' with a PDE-parameter dependent hypernetwork is the key ingredient for improving generalization and sample efficiency of RL algorithms.

4.2. 2D Navier-Stokes equations

As a second test case, we consider a 2D parametric incompressible Navier-Stokes (NS) equations. The NS equations are in fluid dynamics with applications in different fields of engineering and science [62]:

$$\begin{aligned} \nabla \cdot \boldsymbol{y} &= 0 \\ \frac{\partial \boldsymbol{y}}{\partial t} + \boldsymbol{y} \cdot \nabla \boldsymbol{y} &= -\frac{1}{\rho} \nabla p + \mu \nabla^2 \boldsymbol{y}\,, \end{aligned} \tag{31}$$

where $\boldsymbol{y} = (y_1, y_2) : \mathcal{X} \times [t_0, t_f] \to \mathbb{R}^2$ represents the 2D velocity field, $\mathcal{X} = [0,1] \times [0,1]$ is the spatial domain, μ is the kinematic viscosity, ρ is the fluid density, and p is the pressure field. To numerically solve the PDE, we discretize the spatial domain in $21 \times 21 \times 2 = 882$ dimensions. The control is applied from one of the boundaries of the domain $\boldsymbol{u}(x_1 = x, x_2 = 0, t) = u_k, \forall x \in [0,1]$, where x_1, x_2 represent the spatial variables in the 2D domain. We use Dirichlet conditions for all the other boundaries. Similarly to the KS, we utilize the reward function in Equation (25) where $\boldsymbol{y}_{\text{ref}}$ is obtained when applying the controls $u = 3 - 5t$ for $t \in [t_0 = 0, t_f = T]$, $u_{\text{ref}} = 2.0$, $T = 0.2$, and $\alpha = 0.01$. It is worth mentioning that the reference solution is the one provided by PDEControlGym with a default and fixed value of the kinematic viscosity μ and it is kept the same for all the difference instances of the parameter. The initial velocity field and the initial pressure are sampled from a uniform distribution $\in (-5, 5)$.

We extend the implementation of PDEControlGym [63] to include a variations of the kinematic viscosity μ, our parameter of interest. Changing the kinematic viscosity is an effective way to change the Reynolds number ($Re \propto 1/\mu$). In particular, we vary $\mu \in [0.01, 0.025, 0.05, 0.075, 0.1]$ during training, while we evaluate the different agents for $\mu \in [0.009, 0.12]$. The agents observe the (discretized) velocity field $\boldsymbol{y}_k$ at different time-steps and predict the 1D control action u_k. We use the same main network and hypernetwork architectures of the KS case. However, due to the high-dimensionality of the state ($21 \times 21 \times 2$), we utilize a convolutional encoder to reduce the state dimensionality to 20 dimensions before feeding it to the RL networks. We discuss the architecture of the convolutional encoder in Appendix Appendix A. Again, we compare our two variants of HypeRL TD3 agent (see Section 3) with (i) the standard TD3 agent with state $\boldsymbol{y}_k$ augmented by the parameter μ, and (ii) the TD3 algorithms without access to the parameter μ.

In Table 3 and 4, we show the training and evaluation rewards collected by the different agents, where we highlight in bold the highest scores in three different phases of the training/evaluation. We consider the average cumulative reward over the whole training/evaluation episodes, the cumulative rewards collected after 250/40 episodes until the end, and the cumulative rewards collected after 450/80 episodes until the end. Even in the NS case, the HypeRL agents still outperform the TD3 agents. However, because the control

Cumulative reward	mean $\pm$ std	mean $\pm$ std ($>$ 250 ep.)	mean $\pm$ std ($>$ 450 ep.)
HypeRL-TD3	$\mathbf{-13.50} \pm 4.00$	$\mathbf{-13.47} \pm 4.10$	$\mathbf{-13.07} \pm 3.85$
HypeRL-TD3 (μ only)	-13.64 ± 4.02	-13.57 ± 4.09	-13.16 ± 3.83
TD3	-13.80 ± 4.10	-13.68 ± 4.18	-13.24 ± 3.94
TD3 (no μ)	-13.97 ± 4.04	-13.88 ± 4.13	-13.41 ± 3.84

Table 3: Mean and standard deviation of the cumulative reward over training collected by the different algorithms.

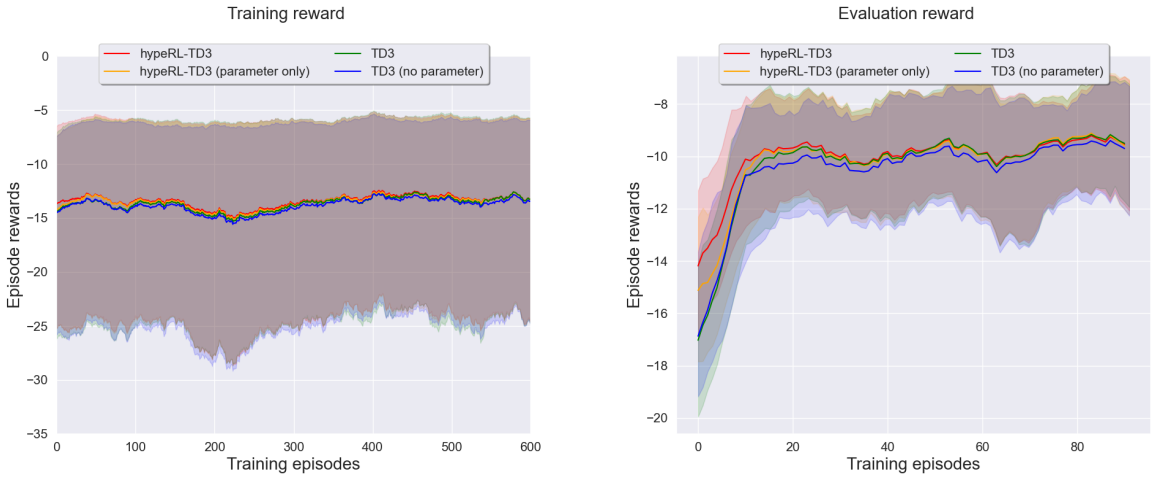
Cumulative reward	mean $\pm$ std	mean $\pm$ std ($>$ 40 ep.)	mean $\pm$ std ($>$ 80 ep.)
HypeRL-TD3	$\mathbf{-10.21} \pm 1.62$	-9.69 ± 0.89	-9.45 ± 0.88
HypeRL-TD3 (μ only)	-10.37 ± 1.88	-9.70 ± 0.93	-9.41 ± 0.02
TD3	-10.58 ± 2.36	$\mathbf{-9.68} \pm 0.90$	$\mathbf{-9.40} \pm 0.87$
TD3 (no μ)	-10.77 ± 2.24	-9.90 ± 0.91	-9.62 ± 0.89

Table 4: Mean and standard deviation of the cumulative reward over evaluation collected by the different algorithms.

is only applied on one boundary, the system is less "controllable" than the KS PDE and the difference in performance among the different agents is smaller. For example, if we only consider the later stages of the training, the TD3 agent is capable of achieving similar performance to the HypeRL agents in evaluation, while still inferior in early training/evaluation stages.

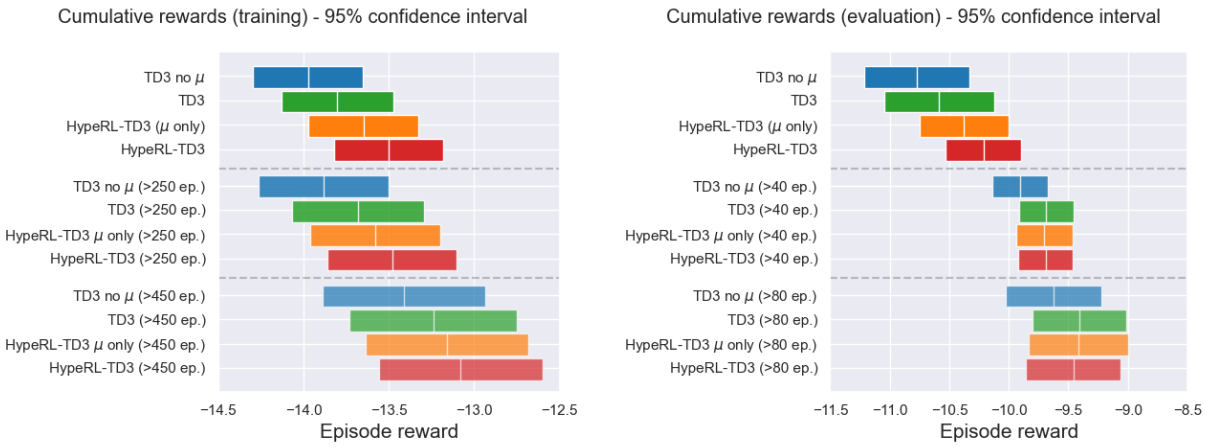
In Figure 6, B.11, and 7, we show average rewards, state and action costs, and 95% confidence bounds of the rewards collected by the different agents. Similarly to the KS case, the results, especially in early stages of the training (see Figure 6 (d)), show the superior performance of the parameter-informed HypeRL-TD3 agents. Again, the agent without access to the parameter μ achieves the worst performance. The analysis of the confidence interval is consistent with the previous results and show very clearly that, especially in early stages of the training, the HypeRL agents can learn good policies with limited training time.

Eventually, we show controlled solutions for $t = t_f$ and $\mu = 0.077$ (unseen value inside the training range) in Figure 8 and $\mu = 0.11$ (unseen value outside the training range) in Figure 9 and we report the state cost c_1, and action cost αc_2. Analogously to the KS case, the TD3 agent without access to the PDE parameter μ is the worst-performing method in interpolation and extrapolation regimes. In the two examples shown, the HypeRL-TD3 with only access to the parameter is the best performing agent, closely followed by HypeRL-TD3, and TD3 with knowledge of μ. It is worth highlighting that the differences in performance among the agents is less pronounced than in the KS case. This happens for two main reasons: (i) the control only affects the system through one boundary, leaving the agent less capabilities to freely steer the PDE towards the desired state, and (ii) the reference state and control used in the reward function in Equation (25), provided by PDEControlGym, were generated for the case of $\mu = 0.1$, making the control problem extremely complex for different values of the parameters.



(a) (b)

Figure 6: Training and evaluation results. The solid line represents the mean and the shaded area the standard deviation over 5 different random seeds.



(a) (b)

Figure 7: 95% confidence intervals of the training and evaluation reward for the different phases of training and evaluation. This metric, suggested in [61], allows to assess the reliability of the results accounting for the stochastic nature of the RL experiments.

5. Conclusion

In this paper, we proposed a novel approach for optimal control of parametric PDEs using RL and hypernetworks that we named HypeRL. In particular, due to the high computational cost of solving PDEs, we focused on developing a framework for enhancing the performance of RL algorithms in terms of sample efficiency and generalization to unseen instances of the PDE parameters when trained on a limited number of instances of such parameters. We tested the capabilities of HypeRL on two challenging control problems of chaotic and parametric PDEs, namely a 1D Kuramoto Sivashinsky and a 2D Navier Stokes. We showed that knowledge of the PDE parameters and how this information is encoded, i.e., via hypernetorks, is an essential ingredient for sample-efficient learning of control policies that can generalize effectively.

Acknowledgments

AM and SF are members of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM) and acknowledge the project "Dipartimento di Eccellenza" 2023-2027, funded by MUR. SF

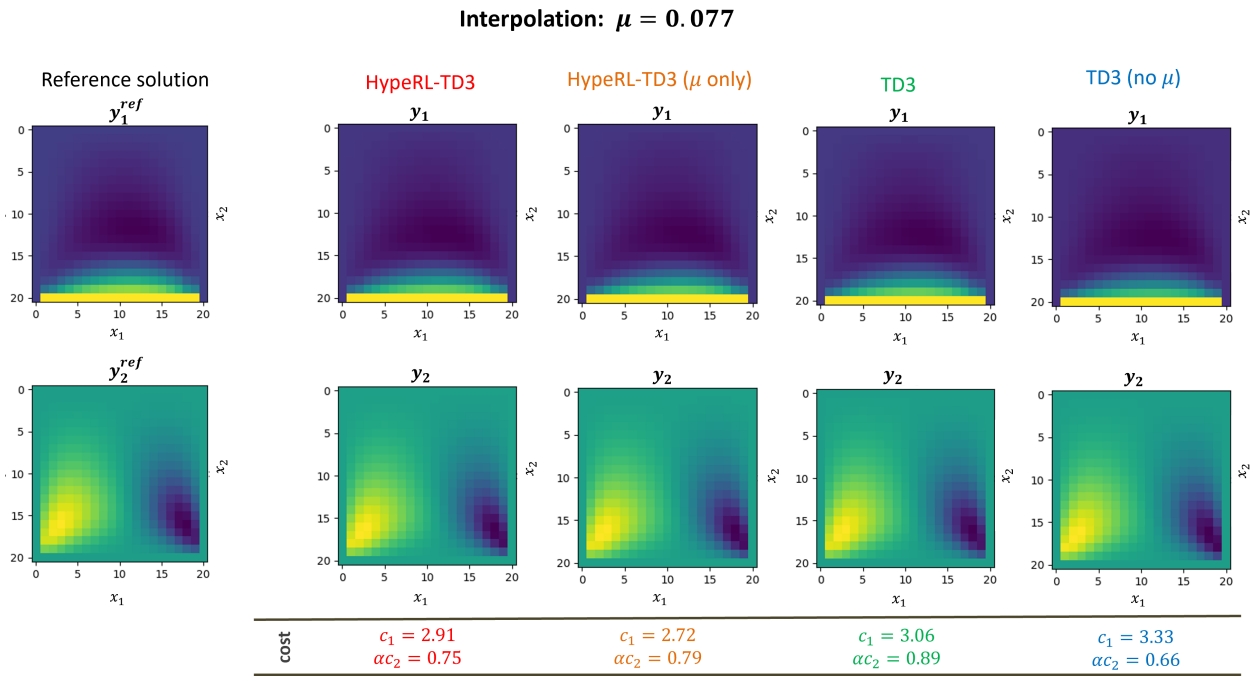


Figure 8: Controlled solutions $\boldsymbol{y}(t = t_f)$ for $\mu = 0.077$.

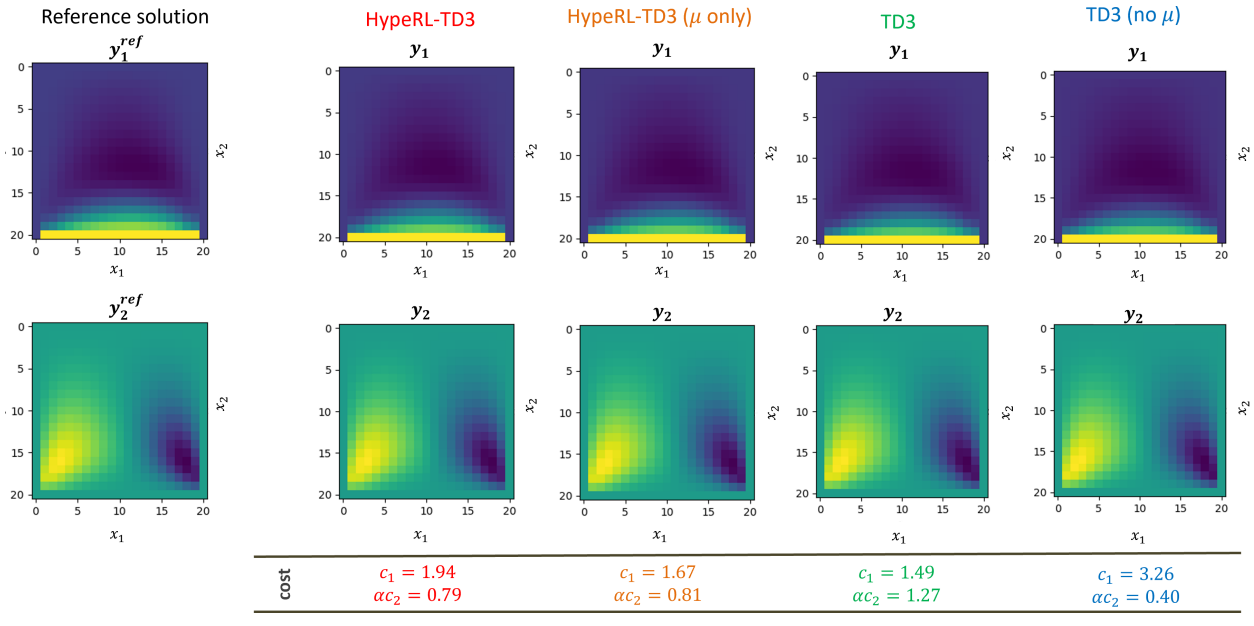


Figure 9: Controlled solutions $\boldsymbol{y}(t = t_f)$ for $\mu = 0.11$.

acknowledges the Istituto Nazionale di Alta Matematica (INdAM) for the financial support received through the "Concorso a n. 45 mensilità di borse di studio per l'estero per l'a.a. 2023-2024". AM acknowledges the PRIN 2022 Project "Numerical approximation of uncertainty quantification problems for PDEs by multi-fidelity methods (UQ-FLY)" (No. 202222PACR), funded by the European Union - NextGenerationEU, and the Project "Reduced Order Modeling and Deep Learning for the real- time approximation of PDEs (DREAM)" (Starting Grant No. FIS00003154), funded by the Italian Science Fund (FIS) - Ministero dell'Università e della Ricerca. AM and SF acknowledge the project FAIR (Future Artificial Intelligence Research), funded by the NextGenerationEU program within the PNRR-PE-AI scheme (M4C2, Investment 1.3, Line on Artificial Intelligence).

References

[1] Andrea Manzoni, Alfio Quarteroni, and Sandro Salsa. *Optimal control of partial differential equations*. Springer, 2021.

[2] S. S. Collis, R. D. Joslin, A. Seifert, and V. Theofilis. Issues in active flow control: theory, control, simulation, and experiment. *Progress in Aerospace Sciences*, 40(4-5):237–289, 2004.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[4] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[5] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[6] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[8] Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, et al. Mastering complex control in moba games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 6672–6679, 2020.

[9] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*, 2019.

[10] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[12] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[13] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[14] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[15] Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

[16] Fangyi Zhang, Jürgen Leitner, Michael Milford, Ben Upcroft, and Peter Corke. Towards vision-based deep reinforcement learning for robotic motion control. In *Australasian Conference on Robotics and Automation 2015*. Australian Robotics and Automation Association (ARAA), 2015.

[17] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

[18] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *2020 IEEE symposium series on computational intelligence (SSCI)*, pages 737–744. IEEE, 2020.

[19] Nicolò Botteghi, Khaled Alaa, Mannes Poel, Beril Sirmacek, Christoph Brune, Abeje Mersha, and Stefano Stramigioli. Low dimensional state representation learning with robotics priors in continuous action spaces. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 190–197. IEEE, 2021.

[20] Gerben Beintema, Alessandro Corbetta, Luca Biferale, and Federico Toschi. Controlling rayleigh–benard convection via reinforcement learning. *Journal of Turbulence*, 21(9-10):585–605, 2020.

[21] Michele Buzzicotti, Luca Biferale, Fabio Bonaccorso, Patricio Clark di Leoni, and Kristian Gustavsson. Optimal control of point-to-point navigation in turbulent time dependent flows using reinforcement learning. Springer, 2020.

[22] Dixia Fan, Liu Yang, Zhicheng Wang, Michael S Triantafyllou, and George Em Karniadakis. Reinforcement learning for bluff body active flow control in experiments and simulations. *Proceedings of the National Academy of Sciences*, 117(42):26091–26098, 2020.

[23] Jean Rabault and Alexander Kuhnle. Accelerating deep reinforcement learning strategies of flow control through a multi-environment approach. *Physics of Fluids*, 31(9), 2019.

[24] Chengwei Xia, Junjie Zhang, Eric C Kerrigan, and Georgios Rigas. Active flow control for bluff body drag reduction using reinforcement learning with partial measurements. *Journal of Fluid Mechanics*, 981:A17, 2024.

[25] Sebastian Peitz, Jan Stenner, Vikas Chidananda, Oliver Wallscheid, Steven L Brunton, and Kunihiko Taira. Distributed control of partial differential equations using convolutional reinforcement learning. *arXiv preprint arXiv:2301.10737*, 2023.

[26] Nicholas Zolman, Urban Fasel, J Nathan Kutz, and Steven L Brunton. Sindy-rl: Interpretable and efficient model-based reinforcement learning. *arXiv preprint arXiv:2403.09110*, 2024.

[27] Nicolò Botteghi and Urban Fasel. Parametric pde control with deep reinforcement learning and differentiable l0-sparse polynomial policies. *arXiv preprint arXiv:2403.15267*, 2024.

[28] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Comput. Surv.*, 50(2), 2017.

[29] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4950–4957, 2018.

[30] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.

[31] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):1–40, 2016.

[32] Timothee Lesort, Natalia Diaz-Rodriguez, Jean-Franois Goudou, and David Filliat. State representation learning for control: An overview. *Neural Networks*, 108:379–392, 2018.

[33] Nicolò Botteghi, Mannes Poel, and Christoph Brune. Unsupervised representation learning in deep reinforcement learning: A review. *arXiv preprint arXiv:2208.14226*, 2022.

[34] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):5149–5169, 2021.

[35] Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial intelligence review*, 18:77–95, 2002.

[36] David Ha, Andrew Dai, and Quoc V. Le. Hypernetworks, 2016.

[37] Vinod Kumar Chauhan, Jiandong Zhou, Ping Lu, Soheila Molaei, and David A Clifton. A brief review of hypernetworks in deep learning. *arXiv preprint arXiv:2306.06955*, 2023.

[38] Elad Sarafian, Shai Keynan, and Sarit Kraus. Recomposing the reinforcement learning building blocks with hypernetworks. In *International Conference on Machine Learning*, pages 9301–9312. PMLR, 2021.

[39] Jacob Beck, Matthew Thomas Jackson, Risto Vuorio, and Shimon Whiteson. Hypernetworks in meta-reinforcement learning. In *Conference on Robot Learning*, pages 1478–1487. PMLR, 2023.

[40] Sahand Rezaei-Shoshtari, Charlotte Morissette, Francois R Hogan, Gregory Dudek, and David Meger. Hypernetworks for zero-shot transfer in reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 9579–9587, 2023.

[41] Yizhou Huang, Kevin Xie, Homanga Bharadhwaj, and Florian Shkurti. Continual model-based reinforcement learning with hypernetworks. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 799–805. IEEE, 2021.

[42] E. Lindelöf. Sur l'application de la méthode des approximations successives aux équations différentielles ordinaires du premier ordre. *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, 118:454–7, 1894.

[43] Donald E Kirk. *Optimal control theory: an introduction*. Courier Corporation, 2004.

[44] Wendell H. Fleming and H.M. Soner. *Controlled Markov processes and viscosity solutions*. Springer New York, NY, 2006.

[45] W. Karush. Minima of functions of several variables with inequalities as side constraints. *M.Sc. Thesis*, 1939.

[46] H. W. Kuhn and A. W. Tucker. Nonlinear programming. *Proceedings of 2nd Berkeley Symposium*, pages 481–492, 1951.

[47] Maurizio Falcone and Roberto Ferretti. *Semi-Lagrangian approximation schemes for linear and Hamilton-Jacobi equations*. 2014.

[48] Dimitri Bertsekas. *Dynamic programming and optimal control: Volume I*, volume 4. Athena scientific, 2012.

[49] Moritz Diehl and Sébastien Gros. Numerical optimal control. *Optimization in Engineering Center (OPTEC)*, 2011.

[50] Frank L Lewis and Draguna Vrabie. Reinforcement learning and adaptive dynamic programming for feedback control. *IEEE circuits and systems magazine*, 9(3):32–50, 2009.

[51] Chayan Banerjee, Kien Nguyen, Clinton Fookes, and Maziar Raissi. A survey on physics informed reinforcement learning: Review and open problems. *arXiv preprint arXiv:2309.01909*, 2023.

[52] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[53] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[54] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[55] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[56] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[57] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[58] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[59] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

[60] Nikolai A Kudryashov. Exact solutions of the generalized kuramoto-sivashinsky equation. *Physics Letters A*, 147(5-6):287–291, 1990.

[61] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems*, 34:29304–29320, 2021.

[62] Roger Temam. *Navier–Stokes equations: theory and numerical analysis*, volume 343. American Mathematical Society, 2024.

[63] Luke Bhan, Yuexin Bian, Miroslav Krstic, and Yuanyuan Shi. Pde control gym: A benchmark for data-driven boundary control of partial differential equations, 2024.

Appendix A. Neural Network Architectures

Appendix A.1. HypeRL Architecture

The main policy and value function networks are composed of one input layer of dimension equal to the state dimension $|\boldsymbol{y}_k|$, and state and action dimension $|\boldsymbol{y}_k| + |\boldsymbol{u}_k|$, respectively, one hidden layer with 256 neurons, and an output layer of dimension equal to the control action dimension $|\boldsymbol{u}_k|$, and dimension 1, respectively. These weights and biases are learned by two hypernetworks with inputs the PDE parameter $\boldsymbol{\mu}$ or state and PDE parameter $\boldsymbol{y}_k$ and $\boldsymbol{\mu}$. The hypernetworks use the same architecture and weight initialization proposed in [38].

Appendix A.2. TD3 Architecture

The TD3 algorithms use the default and widely-used architectures introduced in [58]. The architectures are analogous to the main policy and value function described in Section Appendix A.1 but with an additional hidden layer of dimension 256. The input to these networks are state and PDE parameter, and state, action, and PDE parameter, respectively. The TD3 without parameter has the very same architecture but without access to the PDE parameter.

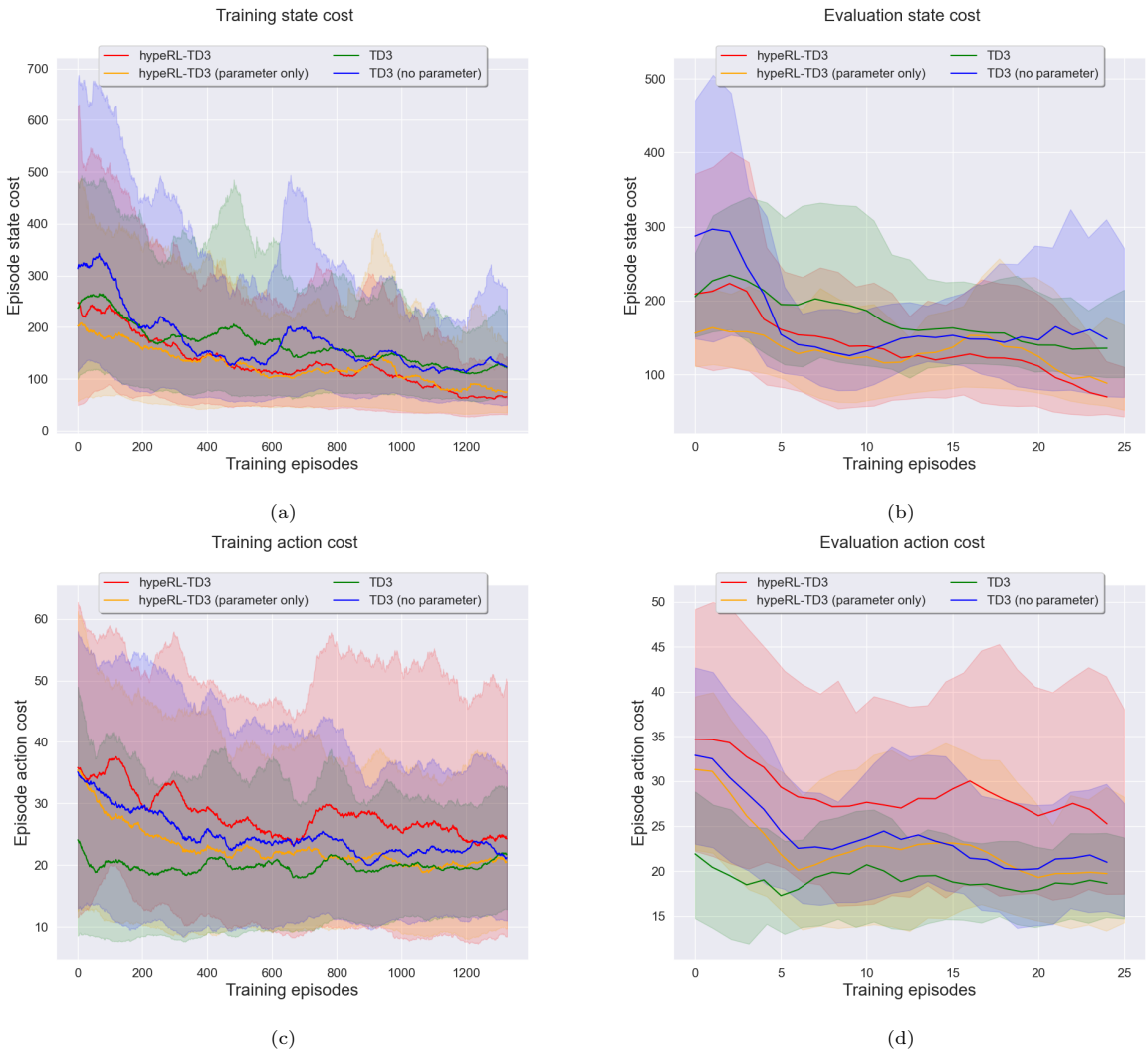


Figure B.10: Training and evaluation results. The solid line represents the mean and the shaded area the standard deviation over 5 different random seeds.

Appendix A.3. Convolutional Encoder 2D Navier-Stokes Equations

Due to the high dimensionality of the state ($21 \times 21 \times 2 = 882$ dimensions) in the NS test case, we utilize a convolutional encoder to reduce the dimensionality of the state to 20 dimensions before feeding it to the RL networks. The convolutional encoder is composed of 4 2D convolutional layers with 32 output channels, kernel size of 3×3, and stride length equal to 1 (with the exception of the first layer that uses stride length equal to 2). After the second and forth convolutional layer, we use a batch-normalization layer. The features are then flattened and fed to two fully-connected layers to reduce the state dimension to 20. A similar architecture was employed in [33].

Appendix B. Additional Results

Appendix B.1. 1D Kuramoto-Sivashinsky PDE

In Figure B.10, we show the state and action costs over training and evaluation for the KS PDE.

Appendix B.2. 2D Navier-Stokes Equations

In Figure B.11, we show the state and action costs over training and evaluation for the NS equations.

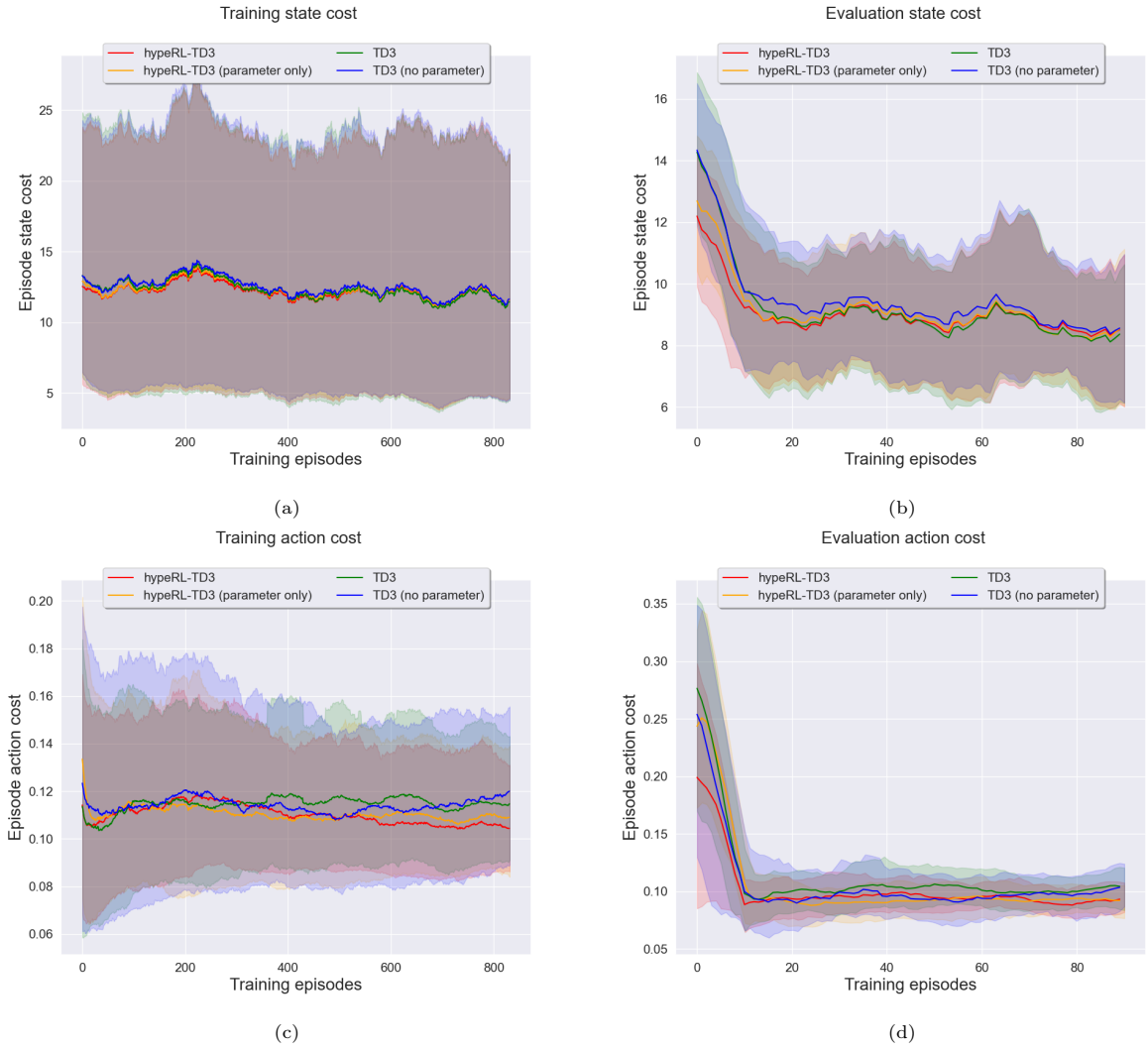


Figure B.11: Training and evaluation results. The solid line represents the mean and the shaded area the standard deviation over 5 different random seeds.

MOX Technical Reports, last issues

Dipartimento di Matematica
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

30/2025 Rosafalco, L.; Conti, P.; Manzoni, A.; Mariani, S.; Frangi, A.
Online learning in bifurcating dynamic systems via SINDy and Kalman filtering

29/2025 Centofanti, E.; Ziarelli, G.; Parolini, N.; Scacchi, S.; Verani, M. ; Pavarino, L. F.
Learning cardiac activation and repolarization times with operator learning

28/2025 Ciaramella, G.; Gander, M.J.; Mazzieri, I.
Discontinuous Galerkin time integration for second-order differential problems: formulations, analysis, and analogies

27/2025 Antonietti P.F.; Artoni, A.; Ciaramella, G.; Mazzieri, I.
A review of discontinuous Galerkin time-stepping methods for wave propagation problems

24/2025 Bartsch, J.; Borzi, A.; Ciaramella, G.; Reichle, J.
Adjoint-based optimal control of jump-diffusion processes

22/2025 Leimer Saglio, C. B.; Pagani, S.; Antonietti P. F.
A p-adaptive polytopal discontinuous Galerkin method for high-order approximation of brain electrophysiology

23/2025 Antonietti, P. F.; Caldana, M.; Mazzieri, I.; Re Fraschini, A.
MAGNET: an open-source library for mesh agglomeration by Graph Neural Networks

21/2025 Caldera, L., Masci, C., Cappozzo, A., Forlani, M., Antonelli, B., Leoni, O., Ieva, F.
Uncovering mortality patterns and hospital effects in COVID-19 heart failure patients: a novel Multilevel logistic cluster-weighted modeling approach

20/2025 Botti, M.; Prada, D.; Scotti, A.; Visinoni, M.
Fully-Mixed Virtual Element Method for the Biot Problem

19/2025 Bortolotti, T.; Wang, Y. X. R.; Tong, X.; Menafoglio, A.; Vantini, S.; Sesia, M.
Noise-Adaptive Conformal Classification with Marginal Coverage