Meta-Reinforcement Learning for the Tuning of PI Controllers: An Offline Approach

Daniel G. McClement[a], Nathan P. Lawrence[b], Johan U. Backström[c], Philip D. Loewen[⋆b], Michael G. Forbes[d], R. Bhushan Gopaluni[⋆a]

[a] *Department of Chemical and Biological Engineering, University of British Columbia, Vancouver, BC Canada*
[b] *Department of Mathematics, University of British Columbia, Vancouver BC, Canada*
[c] *Backstrom Systems Engineering Ltd.*
[d] *Honeywell Process Solutions, North Vancouver, BC Canada*
[⋆] *Authors provided equal supervision.*

Abstract

Meta-learning is a branch of machine learning which trains neural network models to synthesize a wide variety of data in order to rapidly solve new problems. In process control, many systems have similar and well-understood dynamics, which suggests it is feasible to create a generalizable controller through meta-learning. In this work, we formulate a meta reinforcement learning (meta-RL) control strategy that can be used to tune proportional-integral controllers. Our meta-RL agent has a recurrent structure that accumulates "context" to learn a system's dynamics through a hidden state variable in closed-loop. This architecture enables the agent to automatically adapt to changes in the process dynamics. In tests reported here, the meta-RL agent was trained entirely offline on first order plus time delay systems, and produced excellent results on novel systems drawn from the same distribution of process dynamics used for training. A key design element is the ability to leverage model-based information offline during training in simulated environments while maintaining a model-free policy structure for interacting with novel processes where there is uncertainty regarding the true process dynamics. Meta-learning is a promising approach for constructing sample-efficient intelligent controllers.

Keywords: Meta-learning, deep learning, reinforcement learning, adaptive control, process control, PID control

1. Introduction

Reinforcement learning (RL) is a branch of machine learning that formulates a goal-oriented "policy" for taking actions in a stochastic environment [1]. This general framework has attracted the interest of the process control community [2]. For example, one can consider feedback control problems without the need for a process model in this setting. Despite its

appeal, an overarching challenge in RL is its need for a significant amount of data to learn a useful policy.

Meta-learning, or "learning to learn", is an active area of research in which the objective is to learn an underlying structure governing a distribution of possible tasks [3]. In process control applications, meta-learning is appealing because many systems have similar dynamics or a known structure, which suggests training over a distribution could improve the sample efficiency[1] when learning any single task. Moreover, extensive online learning is impractical for training over a large number of systems; by focusing on learning a underlying structure for the tasks, we can more readily adapt to a new system.

This paper proposes a method for improving the *online* sample efficiency of RL agents. Our approach is to train a "meta" RL agent *offline* by exposing it to a broad distribution of different dynamics. The agent synthesizes its experience from different environments to quickly learn an optimal policy for its present environment. The training is performed completely offline and the result is a single RL agent that can quickly adapt its policy to a new environment in a model-free fashion.

We apply this general method to the industrially-relevant problem of autonomous controller tuning. We show how our trained agent can adaptively fine-tune proportional-integral (PI) controller parameters when the underlying dynamics drift or are not contained in the distribution used for training. We apply the same agent to novel dynamics featuring nonlinearities and different time scales. Moreover, perhaps the most appealing consequence of this method is that it removes the need to accommodate a training algorithm on a system-by-system basis – for example, through extensive online training or transfer learning, hyperparameter tuning, or system identification – because the adaptive policy is pre-computed and represented in a single model.

1.1. Contributions

In this work, we propose the use of meta-reinforcement learning (meta-RL) for process control applications. We create a recurrent neural network (RNN) based policy. The hidden state of the RNN serves as an encoding of the system dynamics, which provides the network with "context" for its policy. The controller is trained using a distribution of different processes referred to as "tasks". We use this framework to develop a versatile controller which can quickly adapt to effectively control any process from a prescribed distribution of processes rather than a single task.

This paper extends McClement et al. [4] with the following additional contributions:

[1] How efficient a machine learning model is at learning from data; a high sample efficiency means a model can effectively learn from small amounts of data.

- A simplified and improved meta-RL algorithm: while [4] required online training, the meta-RL agent in this work is trained entirely offline in advance.
- Completely new simulation studies, including industrially-relevant examples dealing with PI controllers and non-linear dynamics; and
- A method of leveraging known, model-based system information offline for the purposes of training, with model-free online deployment.

This framework addresses key priorities in industrial process control, particularly:

- Initial tuning and commissioning of a PID controller, and
- Adaptive updates of the PID controller as the process changes over time.
- Scalable maintenance of PID controllers across many different systems without case-by-case tuning.

This paper is organized as follows: In Section 2 we summarize key concepts from RL and meta-RL; in Section 3 we describe our algorithm for meta-RL and its practical implementation for process control applications. We demonstrate our approach through numerical examples in Section 4, and conclude in Section 5.

1.2. Related work

We review some related work at the intersection of RL and process control. For a more thorough overview the reader is referred to the survey papers by Shin et al. [5], Lee et al. [6], or the tutorial-style papers by Nian et al. [2], Spielberg et al. [7].

Some initial studies by Hoskins and Himmelblau [8], Kaisare et al. [9], Lee and Lee [10], Lee and Wong [11] in the 1990s and 2000s demonstrated the appeal of reinforcement learning and approximate dynamic programming for process control applications. More recently, there has been significant interest in deep RL methods for process control [12, 13, 14, 15, 16, 17, 18].

Spielberg et al. [7] adapted the deep deterministic policy gradient (DDPG) algorithm for setpoint tracking problems in a model-free fashion. Meanwhile, Wang et al. [19] developed a deep RL algorithm based on proximal policy optimization [20]. Petsagkourakis et al. [21] use transfer learning to adapt a policy developed in simulation to novel systems. Variations of DDPG, such as twin-delayed DDPG (TD3) [22] or a Monte-Carlo based strategy, have also shown promising results in complex control tasks [23, 24]. Other approaches to RL-based control utilize a fixed controller structure such as PID [25, 26, 27, 28]; some of these are applied to a physical system [29, 30, 31].

This present work differs significantly from the approaches mentioned so far. Other approaches to more sample-efficient RL in process control utilize apprenticeship learning,

transfer learning, or model-based strategies augmented with deep RL algorithms [32, 21, 33]. Our method differs in two significant ways. First, the training and deployment process is simplified with our meta-RL agent through its synthesized training over a large distribution of systems. Therefore, only one model needs to be trained, rather than training models on a system-by-system basis. Second, the meta-RL agent in our framework does not rely on precise system identification and only a crude understanding of the process dynamics is required. By training across a distribution of process dynamics, the meta-RL agent learns to control a wide variety of processes with no online or task-specific training required. Although the meta-RL agent is trained in simulation, the key to our approach is that the policy only utilizes process data, and thus achieves efficient model-free control on novel dynamics. A similar concept has been reported in the robotics literature where a robust policy for a single agent is trained offline, leveraging "privileged" information about the system dynamics [34]. Most similar to this present work is a paper in the field of robotics where a recurrent PPO policy was trained with randomized dynamics to improve the adaptation from simulated environments to real ones [35].

2. Background

2.1. Reinforcement learning

In this section, we give a brief overview of deep RL and highlight some popular meta-RL methods. We refer the reader to Nian et al. [2], Spielberg et al. [7], for tutorial overviews of deep RL with applications to process control. We use the standard RL terminology that can be found in Sutton and Barto [36]. Huisman et al. [37] gives a unified survey of deep meta-learning.

The RL framework consists of an *agent* and an *environment*. For each *state* $s_t \in \mathcal{S}$ (the state-space) the agent encounters, it takes some *action* $a_t \in \mathcal{A}$ (the action-space), leading to a new state s_{t+1}. The action is chosen according to a conditional probability distribution π called a *policy*; we denote this relationship by $a_t \sim \pi(a_t|s_t)$. Although the system dynamics are not necessarily known, we assume they can be described as a Markov decision process (MDP) with initial distribution $p(s_0)$ and transition probability $p(s_{t+1}|s_t, a_t)$. A state-space model in control is a special case of an MDP, where the states are the usual (minimal realization) vector that characterizes the system, while the actions are the control inputs. However, the present formulation is more general, as we will demonstrate in later sections. At each time step, a bounded scalar *cost*[2] $c_t = c(s_t, a_t)$ is evaluated. The cost function describes the desirability of a state-action pair: defining it is a key part of the design process.

[2]In RL literature, the objective is a maximization problem in terms of a *reward* function. Equivalently, we will formulate a minimization problem in terms of a *cost* function.

The overall objective, however, is the expected long-term cost. In terms of a user-specified discount factor $0 < \gamma < 1$, the optimization problem of interest becomes

$$\begin{aligned} &\text{minimize} \quad J(\psi) = \mathbb{E}_{h\sim p^{\pi_\psi}(\cdot)}\left[\sum_{t=1}^{\infty}\gamma^{t-1}c(s_t,\pi_\psi(s_t))\middle| s_0\right] \\ &\text{over all} \quad \psi \in \mathbb{R}^n. \end{aligned} \tag{1}$$

In this formulation, h denotes an infinite-horizon trajectory $h = (s_0, a_0, c_0, \ldots, s_N, a_N, c_N, \ldots)$ and the notation $h \sim p^\pi$ indicates that the policy π induces a probability distribution p^π on the set of such trajectories. Within the space of all possible policies, we optimize over a parameterized subset whose members are denoted π_ψ. We use ψ as a generic term for a vector of parameters: in our application, the individual parameters are weights in a neural network.

Common approaches to solving Problem (1) involve techniques based on Q-learning (value-based methods) and the policy gradient theorem (policy-based methods) [36], or a combination of both called *actor-critic* methods [38]. Closely-related functions to J are the Q-function (state-action value function) and value function, respectively:

$$Q(s_t, a_t) = \mathbb{E}_{h\sim p^{\pi}(\cdot)}\left[\sum_{k=t}^{\infty}\gamma^{t-1}c(s_k,a_k)\middle| s_t, a_t\right] \tag{2}$$

$$V(s_t) = \mathbb{E}_{h\sim p^{\pi}(\cdot)}\left[\sum_{k=t}^{\infty}\gamma^{t-1}c(s_k,a_k)\middle| s_t\right]. \tag{3}$$

The *advantage function* is then $A(s,a) = Q(s,a) - V(s)$. These functions help form the basis for deep RL algorithms, that is, algorithms that use deep neural networks to solve RL tasks. Deep neural networks are a flexible form of function approximators, well-suited for learning complex control laws. Moreover, function approximation methods make RL problems tractable in continuous state and action spaces [39, 40, 41]. Without them, discretization of the state and action spaces is necessary, accentuating the "curse of dimensionality"[3].

A standard approach to solving Problem (1) uses gradient descent:

$$\psi \leftarrow \psi - \alpha\nabla J(\psi), \tag{4}$$

where $\alpha > 0$ is a step-size parameter. Analytic expressions for such a gradient exist for both stochastic and deterministic policies [36, 40]. However, in practice, approximations are

[3]The "curse of dimensionality" refers to data sets having exponentially larger "sample spaces" as the number of features grows. The larger sample space requires exponentially more training data to learn from, reducing the sample efficiency.

necessary. Therefore, it is of practical interest to formulate a "surrogate" objective that can be used to decrease the true objective given in (1).

Trust region policy optimization (TRPO) is an on-policy method for decreasing J with each policy update [42]. Using the latest policy, whose weights we denote by ψ_{old}, the surrogate objective function is defined as

$$L_{\psi_{\text{old}}}(\psi) = \mathbb{E}_{h\sim p^{\pi_{\psi_{\text{old}}}}(\cdot)}\left[\frac{\pi_\psi(s)}{\pi_{\psi_{\text{old}}}(s)} A_{\psi_{\text{old}}}(s,a)\right] \tag{5}$$

The surrogate objective function $L_{\psi_{\text{old}}}$ quantifies the advantage of the optimization variable, policy π_ψ, over the trajectories of the most recent policy, using the old policy $\pi_{\psi_{\text{old}}}$ as an importance sampling estimator. The keys behind the derivation of TRPO are twofold: 1) There exists a non-trivial step-size that will improve the true objective J; 2) In order to decrease the true objective, one must place a constraint on the "difference" between policies between update iterations. We use the Kullback-Leibler (KL) divergence, defined for generic probability densities p and q by $D_{\text{KL}}(p||q) = \mathbb{E}_{x\sim p}\left[\log\left(\frac{p(x)}{q(x)}\right)\right]$. The principal result is that there is constant C such that

$$\begin{aligned} J(\pi) &\leq L_{\psi_{\text{old}}}(\psi) + C D_{\text{KL}}^{\max}(\pi_{\psi_{\text{old}}}, \pi) \\ &\text{where} \quad D_{\text{KL}}^{\max}(\pi, \tilde{\pi}) = \max_s D_{\text{KL}}(\pi(s)||\tilde{\pi}(s)), \end{aligned}$$

and that minimizing this function over ψ will decrease the true objective J [42]. In practice, TRPO minimizes $L_{\psi_{\text{old}}}$ subject to a hard constraint on $D_{\text{KL}}^{\max}$ between policy iterates. Regardless of this hard constraint, the optimization problem is solved using natural policy gradients, which requires computing the Hessian of the KL-divergence with respect to the policy parameters. Thus, the main disadvantage of TRPO is its scalability due to its computational burden.

Proximal policy optimization (PPO) is a first-order approximation of TRPO [20]. The main idea behind PPO is to modify the surrogate loss function in Equation (5) such that parameter updates using stochastic gradient descent do not drastically change the policy probability density. The new surrogate objective function is the following:

$$L_{\psi_{\text{old}}}^{\text{PPO}}(\psi) \quad = \quad \mathbb{E}_{h\sim p^{\pi_{\psi_{\text{old}}}}(\cdot)}\left[\max\left\{\frac{\pi_\psi(s)}{\pi_{\psi_{\text{old}}}(s)} A_{\psi_{\text{old}}}(s,a),\ \text{sat}\left(\frac{\pi_\psi(s)}{\pi_{\psi_{\text{old}}}(s)}; 1, \epsilon\right) A_{\psi_{\text{old}}}(s,a)\right\}\right] \tag{6}$$

where $\text{sat}(u; 1, \epsilon) = u$ if $-\epsilon < u - 1 < \epsilon$ and $\text{sat}(u; 1, \epsilon) = 1 + \epsilon\frac{u}{|u|}$ otherwise. Despite being somewhat complicated, the intuition for Equation (6) is understood through cases inside the 'max' functions: when A is positive, the term inside the expectation becomes $\max\left(\frac{\pi_\psi(s)}{\pi_{\psi_{\text{old}}}(s)}, 1-\epsilon\right) A_{\psi_{\text{old}}}(s,a)$, which puts a limit on how much the objective can decrease;

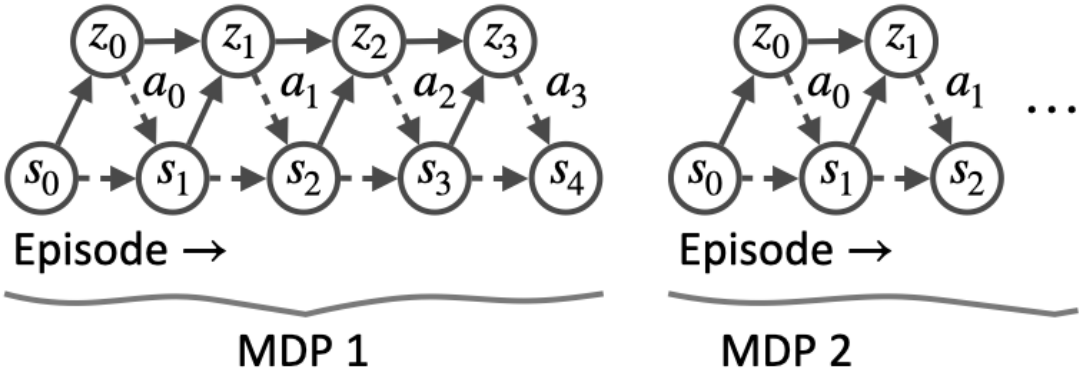


Figure 1: A diagram of the meta-RL agent's interactions with the task distribution $p(\mathcal{T})$.

the case when A is negative is similar. Either way, the term inside the expectation can only decrease by making actions more or less likely, depending on if the advantage is positive or negative, respectively. Moreover, the saturation limits how much the new policy can deviate from the old one. Trajectories with π_{old} are used to approximate $A_{\psi_{\text{old}}}$, which is then used to approximate and optimize Equation (6) using gradient descent.

2.2. Meta Reinforcement Learning

While the algorithms mentioned above can achieve impressive results in a wide range of domains, they are designed to be applied to a single MDP. In contrast, meta-RL aims to generalize agents to a distribution of MDPs. Formally, a single MDP can be characterized by a tuple $\mathcal{T} = (\mathcal{S}, \mathcal{A}, p, c, \gamma)$; in contrast, meta-RL tackles an optimization problem over a distribution $p_{\text{meta}}(\mathcal{T})$ of MDPs. Therefore, in the meta-RL terminology, a "task" is simply all the components comprising a single RL problem. The problem of interest in the meta-RL setting is a generalization of the standard RL objective in Problem (1) [37]:

$$\begin{aligned} &\text{minimize} \quad J_{\text{meta}}(\boldsymbol{\Psi}) = \mathbb{E}_{\mathcal{T}\sim p_{\text{meta}}(\mathcal{T})}\left[J(\psi^*(\mathcal{T}, \boldsymbol{\Psi}))\right] \\ &\text{over all} \quad \boldsymbol{\Psi} \in \mathbb{R}^n. \end{aligned} \tag{7}$$

Crucially, in the context of process control, meta-RL does *not* aim to find a single controller that performs well across different plants. Note that ψ^* in Equation (7) is the optimal weight vector in (1) as a function of a sampled MDP $\mathcal{T}$ and the meta-weights $\boldsymbol{\Psi}$. Meta-RL agents aims to simultaneously learn the underlying structure characterizing different plants and the corresponding optimal control strategy under its cost function. The practical benefit is that this enables RL agents to quickly adapt to novel environments.

There are two components to meta-learning algorithms: the models (e.g., actor-critic networks) that solve a given task, and a set of meta-parameters that learn how to update the model [43, 44]. Due to the shared structure among tasks in process control applications, we are interested in *context-based* meta-RL methods [45, 46, 47]. These approaches learn a latent representation of each task, enabling the agent to simultaneously learn the context and the policy for a given task.

Our method is similar to Duan et al. [45]. We treat the problem in line (7) as a single RL problem. For each MDP $\mathcal{T} \sim p(\mathcal{T})$, the meta-RL agent has a maximum number of time steps, T, to interact with the environment, called an *episode*. As each episode progresses, the RL agent has an internal hidden state z_t which evolves with each time step through the MDP based on the RL states the agent observes: $z_t = f_{\mathbf{\Psi}}(z_{t-1}, s_t)$. The RL agent conditions its actions on both s_t and z_t. An illustration of this concept is shown in Figure 1. Therefore, the purpose of the meta-parameters $\mathbf{\Psi}$ is to quickly adapt a control policy for an MDP $\mathcal{T} \sim p(\mathcal{T})$ by solving for a suitable set of MDP-specific parameters encoded by z_t. This is why this approach is described as meta-RL; rather than training a reinforcement learning agent to control a process, we are training a meta-reinforcement learning agent to find a suitable set of parameters for a reinforcement learning agent which can control a process. The advantage of training a meta-RL agent is that the final model is capable of controlling every MDP across the task distribution $p(\mathcal{T})$ whereas a regular RL agent could only be optimized for a single task $\mathcal{T}$.

Clearly, the key component of the above framework is the hidden state. This is generated with a recurrent neural network (RNN), which we briefly describe in a simplified form. An RNN is a special neural network structure for processing sequential data. Its basic form [48] is shown below:

$$z_t = f\left(W z_{t-1} + U x_t + b\right) \tag{8}$$

$$o_t = V z_t + c. \tag{9}$$

Here W, U, V, b, c are trainable weights, while x_t is some input to the network and o_t is the output. f is a nonlinear function. An RNN can be thought of as a nonlinear state-space system that is optimized for some objective. The characteristic feature of any type of RNN is the hidden state, which evolves alongside sequential input data. The simple RNN formulation in Equations 8 and 9 is prone to vanishing or exploding gradients. In practice, we mitigate these problems by using a more sophisticated form of recurrent layer called the gated recurrent unit (GRU). GRUs use trainable information gates to control the updates to a layer's hidden state which help avoid vanishing gradients, the reader is referred to [49] for further information on the GRU architecture.

3. Meta-RL for process control

We apply the meta-RL framework to the problem of tuning proportional-integral (PI) controllers. The formulation can be applied to any fixed-structure controller, but due to their prevalence, we focus on PI controllers as a practical illustration.

3.1. Tasks, states, actions, costs

The systems of interest are first-order plus time delay (FOPTD): their transfer functions have the form

$$G(s) = \frac{K}{\tau s + 1} e^{-\theta s}, \tag{10}$$

where K is the process gain, τ is the time constant, θ is the time delay, and s is the Laplace variable (not to be confused with s_t, the RL state at time step t). Such models are often good low-order approximations for the purposes of PI tuning [50]. The formulation in continuous time is tidy, but in practice we of course discretize Equation (10).

A PI controller has the form

$$C(s) = K_c \left(1 + \frac{1}{\tau_I s}\right), \tag{11}$$

where K_c and τ_I are constant tuning parameters. In our numerical work, we used $k_p = K_c$ and $k_i = K_c/\tau_I$ instead of K_c and τ_I in the RL state to improve the numerical stability of the computations[4].

Prior work on RL for PI tuning suggests an update scheme of the form [31]:

$$[k_p, k_i] \leftarrow [k_p, k_i] + \alpha \nabla J([k_p, k_i]) \tag{12}$$

$$= [k_p, k_i] + \Delta[k_p, k_i] \tag{13}$$

where the RL policy is directly parameterized as a PI controller. Therefore, in the meta-RL context, we take the actions to be changes to the PI parameters $\Delta[k_p, k_i]$.

For simplicity, the MDP state (s_t) used by the RL agent to select its actions (updates to the PI parameters) is based on the standard form of the PI controller. In practice, different flavours of fixed-structure controllers can be used, including PI controllers in velocity form and full PID controllers. The MDP state at time step t contains the PI parameters, the proportional setpoint error and the integrated setpoint error from the beginning of the episode, t_0:

$$s_t = \left[k_p, k_i, e_t, \int_{t_0}^{t} e_\tau d\tau\right]. \tag{14}$$

[4]The inverse relationship between τ_I and the controller output can cause instability early in offline training, if a poorly trained meta-RL model sets $\tau_I \approx 0$. No similar stability concerns arise when using k_i.

The RL agent is trained to minimize its discounted future cost interacting with different tasks. The cost function used to train the meta-RL agent is the squared error from a target trajectory, shown in Equation (15). The target trajectory is calculated by applying a first order filter to the setpoint signal[5]. The time constant of this filter is set to the desired closed-loop time constant, τ_{cl}. A target closed-loop time constant of 2τ is chosen for robustness and smooth control action, though other choices for τ_{cl} could be made by a control practitioner, such as setting τ_{cl} to the process dead time [50]. An L^1 regularization penalty $\beta > 0$ on the agent's actions is also added to the cost function to encourage sparsity in the meta-RL agent's output and help the tuning algorithm converge to a constant set of PI parameters (rather than acting as a non-linear feedback controller and constantly changing the controller parameters in response to the current state of the system).

$$c_t = (y_{desired,t} - y_t)^2 + \beta_1 |\Delta k_p| + \beta_2 |\Delta k_i|, \tag{15}$$

$$Y_{desired}(s) = \frac{y_{sp}}{2\tau s + 1} e^{-\theta s} \tag{16}$$

Comparing the RL state definition to the RL cost definition, we see similar trajectories through different MDPs will receive very different costs depending on the underlying system dynamics in the particular tasks being controlled. In order for the meta-RL agent to perform well on a new task, it needs to perform implicit system identification to generate an internal representation of the system dynamics.

The advantages of this meta-RL scheme for PI tuning are summarized as follows:

- Tuning is performed in closed-loop and without explicit system identification.
- Tuning is performed automatically even as the underlying system changes.
- The agent can be deployed on new systems within the task distribution $p(\mathcal{T})$ without any online training. Further, as shown in Section 4.5, nearly any system can be modified to be "in-distribution" in this sense.
- The meta-RL agent is a single model that is trained once, offline, so there is no need to specify hyperparameters on a task-by-task basis.
- The meta-RL agent's cost function is conditioned on the process dynamics and will produce consistent closed-loop control behaviour on different systems.

[5]The filtered setpoint signal is only used to calculate the meta-RL agent's cost function. The PI controller itself does not use this filtered setpoint signal; the controller uses the unfiltered setpoint signal when calculating control actions.

This approach is not limited to PI tuning. It can also be applied to other scenarios where the model *structure* is known. The agent then learns to behave near-optimally inside each task in the training distribution, bypassing the need to identify model parameters and only train on that instance of the dynamics.

3.2. RL Agent Structure

The structure of the meta-RL agent is shown in Figure 2. The grey box shows the "actor", i.e., the part of the agent used online for controller tuning. Through interacting with a system and observing the RL states at each time step, the agent's recurrent layers create an embedding (hidden state) which encodes information needed to tune the PI parameters, including information about the system dynamics and the uncertainty associated with this information. These embeddings essentially represent process-specific RL parameters which are updated as the meta-RL agent's knowledge of the process dynamics changes. Two fully connected layers use these embeddings to recommend adjustments to the controller's PI parameters. The inclusion of recurrent layers is essential for the meta-RL agent's performance. Having a hidden state carried between time steps equips the agent with memory and enables the agent to learn a representation of the process dynamics. A traditional feedforward RL network would be unable to differentiate between different tasks and would perform significantly worse. This concept is demonstrated in McClement et al. [4].

Outside of the grey box are additional parts of the meta-RL agent which are only used during offline training. The "critic" (shown in green) is trained to calculate the value (an estimate of the agent's discounted future cost in the current MDP given the current RL state). This value function is used to train the meta-RL actor through gradient descent using Equation (6).

A unique strategy we use to improve the training efficiency of the meta-RL agent is to give the critic network access to "privileged information", defined as any additional information outside the RL state and denoted by ζ. In addition to the RL state, the critic conditions its estimates of the value function on the true process parameters (K, τ, and θ), as well as the deep hidden state[6] of the actor. Knowledge of a task's process dynamics, as well as knowledge of the actor's internal representation of the process dynamics through its hidden state, allows the controller to more accurately estimate the value function, which improves the quality of the surrogate objective function used to train the actor. Equipping the critic with this information also allows it to operate as a simpler feedforward neural network rather than a recurrent network like the actor.

The privileged information given to the critic network may at first appear to conflict

[6]The deep hidden state is the hidden state of the second (i.e. "deeper") recurrent layer in the meta-RL agent.

with the advantages of the proposed meta-RL tuning method, since the critic requires the true system parameters and much simpler tuning methods for PI controllers exist if such information is known. However, this information is only required during *offline* training. The meta-RL agent is trained on simulated systems with known process dynamics, but the end result of this training procedure is a meta-RL agent that can be used to tune PI parameters for a real process *online* with no task-specific training or knowledge of the process dynamics. The portion of the meta-RL agent operating online contained in the grey box only requires RL state information – process data – at each time step.

3.3. Training Algorithm

The meta-RL agent is trained by uniformly sampling K, τ, and θ to create a FOPTD system and initializing a PI controller with $K_c = 0.05$ and $\tau_I = 1.0$. These initial PI tunings were selected because they result in a very slow control response for any possible system from the meta-RL training distribution. The ultimate performance of the PI controller can then be attributed to the meta-RL agent's tuning rather than a good initialization of the controller's parameters. The state of the system is randomly initialized near zero and the setpoint is switched between 1 and -1 every 11 units of time. The meta-RL agent has no inherent time scale and so we keep the units of time general to highlight the applicability of the proposed PI tuning algorithm to both fast and slow processes (allowing time constants whose orders range from milliseconds to hours).

Section 3.3 shows the ranges from which the FOPTD model parameters are uniformly sampled during training. In Section 4.5 we demonstrate how data augmentation extends the applicability of training across this range of parameters.

There are two main limitations to the size of the task distribution the meta-RL agent can effectively be trained across. First, neural network training works best when the features of interest have a consistent scale. However, for different systems, suitable k_p and k_i parameters can vary by orders of magnitude. It becomes very difficult to train a neural network to effectively process inputs with significantly varying magnitudes (k_p and k_i are part of the RL state) as well as produce outputs which vary by orders of magnitude (Δk_p and Δk_i are the RL actions). Second, the time scale of the distribution of systems must be reasonably bounded so there exists a sampling time[7] for the meta-RL agent to use which is appropriate for every system it interacts with. A large MDP time step on systems with fast dynamics will not allow the meta-RL agent to effectively learn the process dynamics. The transient response to any setpoint change or disturbance would occur between time steps and not be

[7]The sampling time referenced in this work is the sampling time for updates to the RL state (which is also the time increment between updates to the controller gains). This is *not* the same as the controller sampling time used to update the control action.

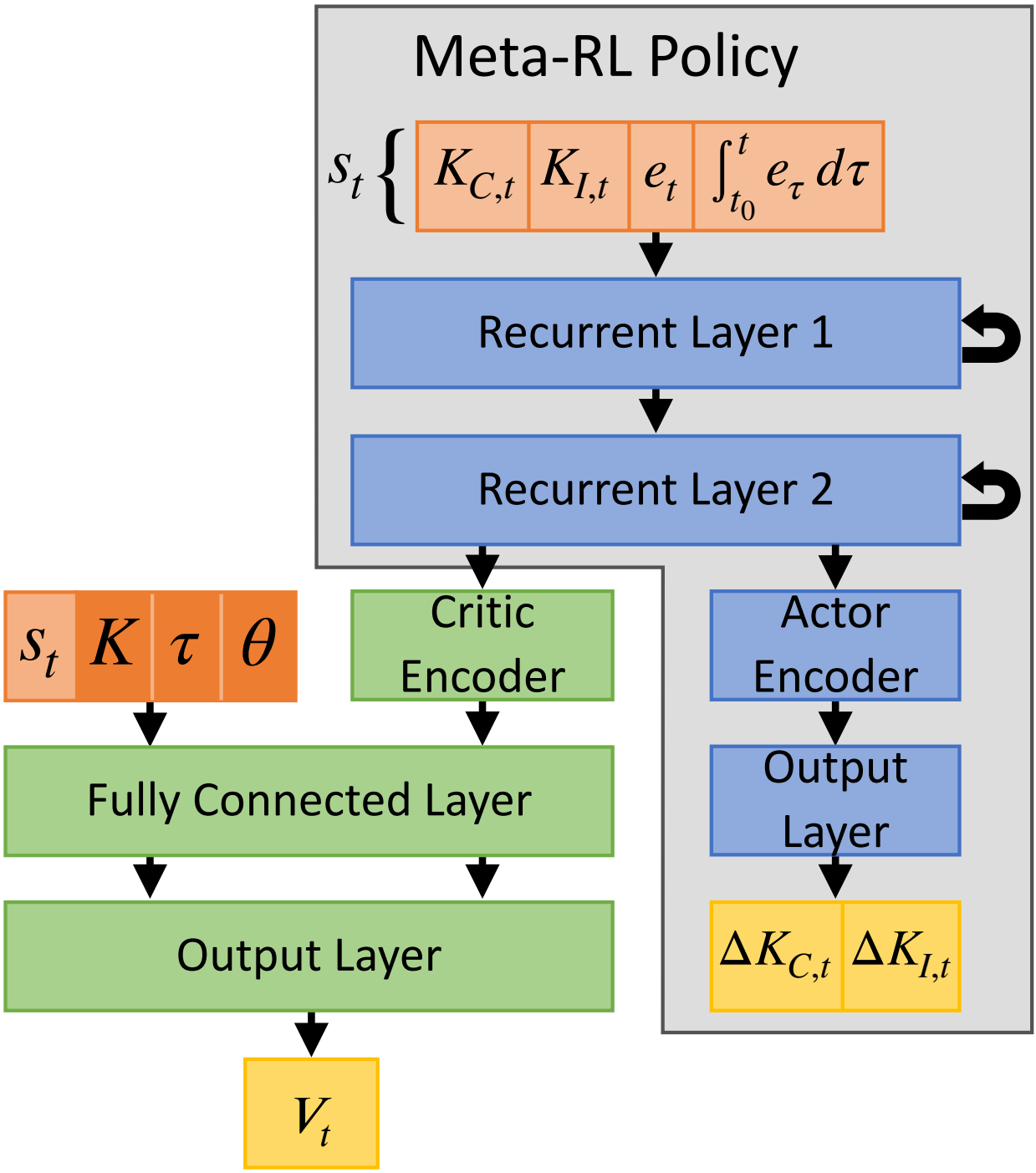


Figure 2: The structure of the RL agent. The control policy used online is shown in the grey box while the critic used during offline training is shown in green.

visible to the neural network. On the other hand, a small sampling time on systems with slow dynamics will cause transient system responses to stretch across many time steps. Recurrent neural networks struggle to learn relationships in data occurring over very long sequences, so the ability for the network to identify systems with slow dynamics is reduced if the time step is too small.

Model Parameter	K	τ	$\frac{\theta}{\tau}$
Minimum	0.25	0.25	0
Maximum	1.0	1.0	1.0

Table 1: The range of model parameters used to train the meta-RL agent.

Algorithm 1 shows the procedure used to train the meta-RL agent. Simulations and model training were performed in Python 3 using the PyTorch machine learning library [51]. We started with the PPO algorithm as implemented in Open AI's "Spinning Up" [52] and modified it to accommodate a recurrent neural network and a distribution of control tasks.

Algorithm 1 Meta-RL Controller Training
Adapted from the documentation of OpenAI's PPO [52]

Input: Initial meta-policy parameters Ψ_0, initial value function parameters ϕ_0

for each training episode **do**
  Sample a batch of n tasks $\mathcal{T}_i \sim p(\mathcal{T})$, $i \in \{1, \ldots, n\}$
  Initialize a buffer to hold state transition data $\mathcal{D}_k$
  for each $\mathcal{T}_i$ **do**
    Collect a trajectory h using the current meta-policy π_Ψ on task $\mathcal{T}_i$
    Store h in $\mathcal{D}_k$
  end for
  Compute advantage estimates $\widehat{A}_t$ using generalized advantage estimation [53] and the current value function V_ϕ.
  Divide trajectories into sequences of the desired length, l, for backpropagation through time.
  Update the policy by minimizing the PPO-Clip objective using gradient descent:

$$\Psi_{k+1} = \arg\min_{\Psi} \frac{1}{|\mathcal{D}_k|T} \sum_{h \in \mathcal{D}_k} \sum_{t=0}^{T} \max\Big(\frac{\pi_\Psi(a_s|s_t)}{\pi_{\Psi_k}(a_s|s_t)} A^{\psi_{\Theta_k}}(s_t, a_t), \; \mathrm{sat}(\epsilon, A^{\psi_{\Theta_k}}(s_t, a_t)) \Big)$$

  Update the value function to estimate the cost-to-go of an episode using gradient descent:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{h \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t, \zeta_t) - \widehat{R}_t)^2$$

end for

4. Experimental results

4.1. Asymptotic Performance of the Meta-RL Tuning Algorithm

Figure 3 depicts the asymptotic performance of the meta-RL tuning method. The intervals of K, τ, and θ/τ in Table 1 define a 3D box in which each point corresponds to a different FOPTD system. After using the meta-RL agent to generate a PI controller for every such system, we could apply a setpoint step from -1 to 1, observe the closed-loop response (see Figure 4), and compute its mean-squared deviation from the target trajectory in Equation (16). The results could, in principle, be used to produce a solid 3D heatmap. Figure 3 shows two heatmaps sliced from this solid. In the left subplot, K is held constant at 0.5, while τ and θ vary on the horizontal and vertical axes. On the right, $\frac{\theta}{\tau}$ is held constant at 0.5, while τ and K vary on the horizontal and vertical axes. The dark color dominating both images corresponds to a very small mean-squared error. Lighter shades in the right

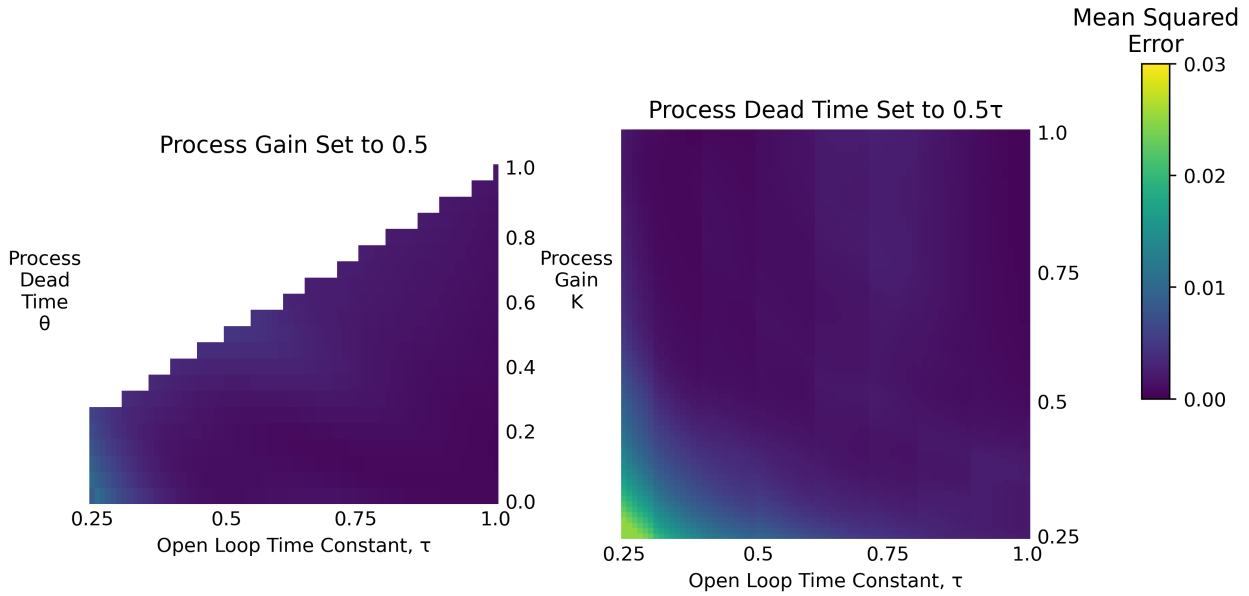


Figure 3: Tracking errors for various FOPTD systems under meta-RL supervision. Each point corresponds to a unique combination of (K_c, τ, θ). Its color shows the corresponding closed-loop system's mean-squared deviation from the target response after the meta-RL agent has acted long enough for the PI parameters to stabilize.

subplot indicate larger MSE values for systems where both K_c and τ are small. The system for which the MSE is largest is labelled with a red dot. Overall, every system in the given region of parameter space is well-controlled by the meta-RL tuning algorithm.

Figure 4 depicts the performance in the worst-case and best-case scenarios based on target trajectory tracking performance selected from Figure 3. In the best-case, the mean squared error between the meta-RL's control trajectory and the target trajectory is 0.0004 while in the worst-case the mean squared error is 0.0300. Even in the worst-case scenario, the meta-RL algorithm's PI tunings provide desirable control performance. Table 2 compares the meta-RL agent's PI tunings in the worst-case and best-case scenario to the PI tunings calculated using the improved SIMC PI tuning method [54], which provides near-optimal tunings for FOPTD systems. In the best-case (which from Figure 3 we see is similar to the performance across most of the task distribution), there is a 2.99% difference between the meta-RL PI tunings and the SIMC tunings.

Table 2: Comparison between the meta-RL agent's PI tunings and those calculated using the SIMC method [54].

	K_c		τ_I	
System	Meta-RL	SIMC	Meta-RL	SIMC
Best-case ($k = 0.5$, $\tau = 1.0$, $\theta = 0.2$)	0.876	0.909	1.042	1.067
Worst-case ($k = 0.25$, $\tau = 0.25$, $\theta = 0.1$)	1.227	1.667	0.367	0.283

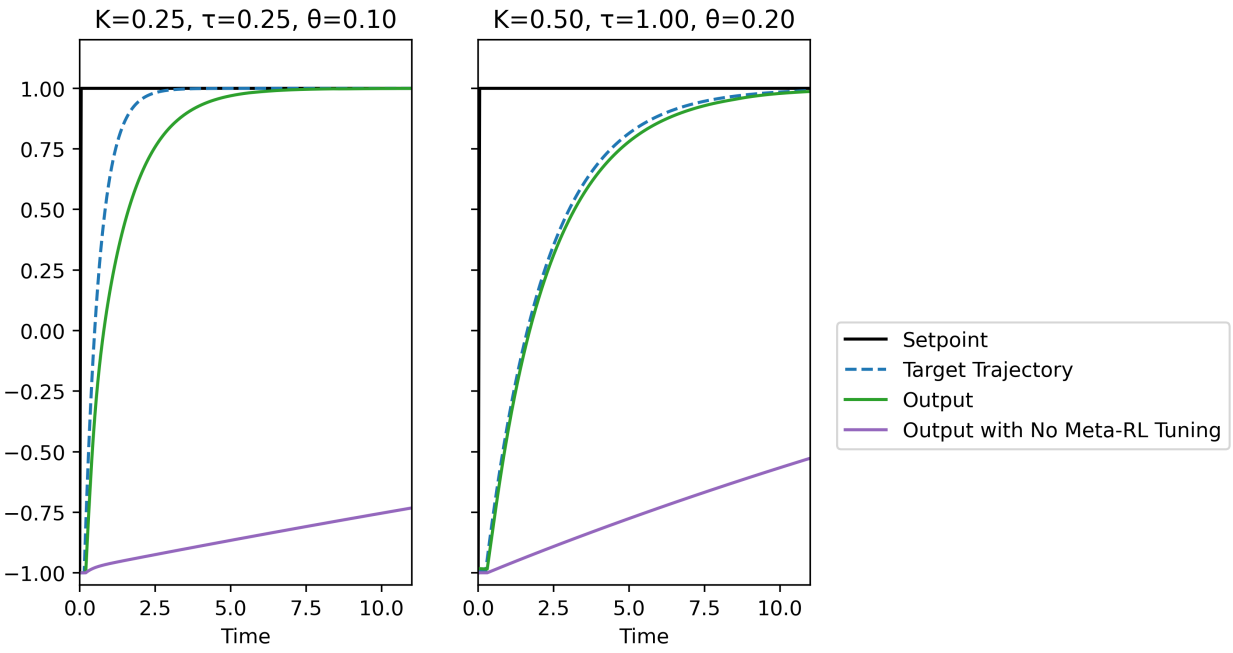


Figure 4: System output trajectories for a setpoint change from −1 to 1 using the meta-RL algorithm's PI tunings compared to the target trajectories. The worst-case (left) and best-case (right) are selected from the heatmaps in Figure 3. A trajectory using the initial PI tunings is also shown for comparison.

4.2. Online Sample Efficiency of the Meta-RL Tuning Algorithm

Section 4.1 showed the asymptotic performance of the meta-RL PI tunings. Another important consideration is the online sample efficiency of the PI tuning; how fast do the controller parameters converge? Figure 5 shows the time for both controller parameters to arrive within 10% of their ultimate values. The convergence of the tunings depends on the excitation in the system. In our experiments, excitation was created by setpoint changes every 11 units of time. The convergence speed could be increased with more excitation (or decreased with less). The meta-RL agent uses a sampling time of 2.75 units of time (i.e. the PI parameters are updated every 2.75 units of time; 4 times for each setpoint change).

Systems with large process gains and fast dynamics converge quickest, requiring just a single setpoint change (around 10 units of time). Systems with small gains and slow dynamics take longer to converge, requiring 13 setpoint changes to converge (around 140 units of time).

Figure 6 shows the performance in the worst-case and best-case scenarios based on convergence time selected from Figure 5. Requiring over 13 setpoint changes to near convergence sounds undesirable, however from Figure 6 we see even in this worst-case scenario, reasonable PI tunings are reached after a single setpoint change. The performance continues to improve with time to more closely match the target trajectory.

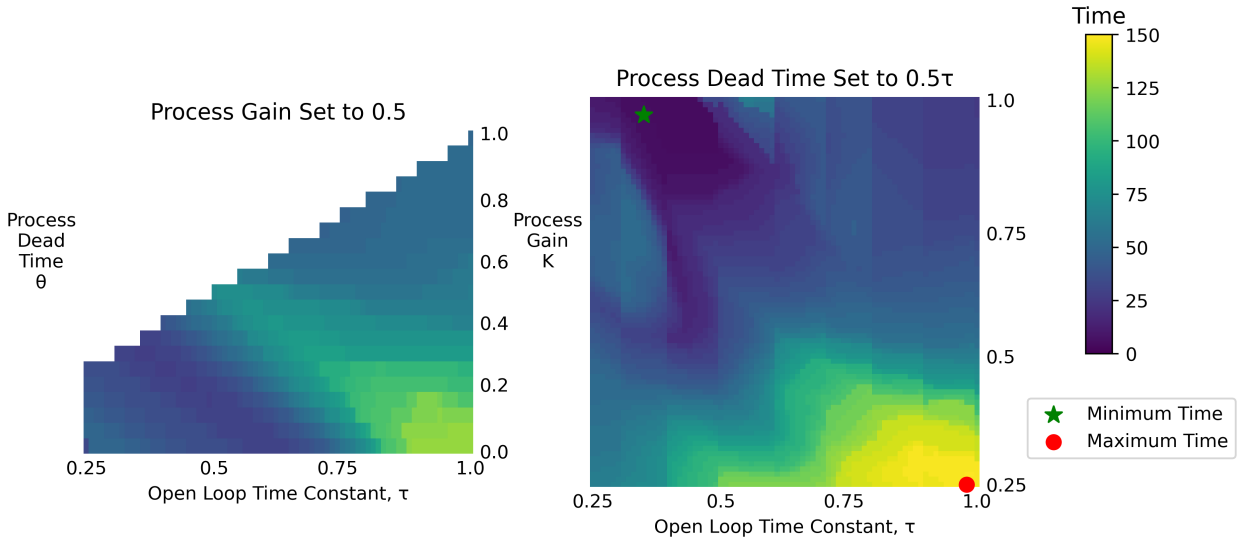


Figure 5: Online time required for both the k_p and k_i parameters to reach $\pm 10\%$ of their ultimate values.

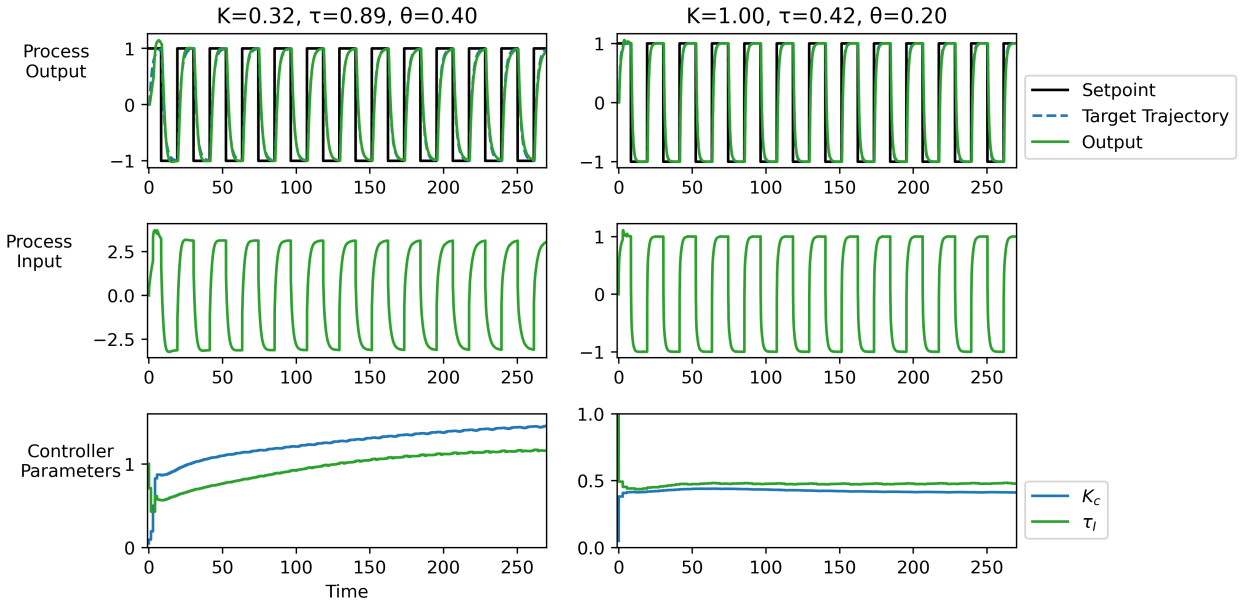


Figure 6: System output trajectories showing the convergence of the controller's PI parameters over time. The worst-case (left) and best-case (right) are selected from the heatmaps in Figure 5.

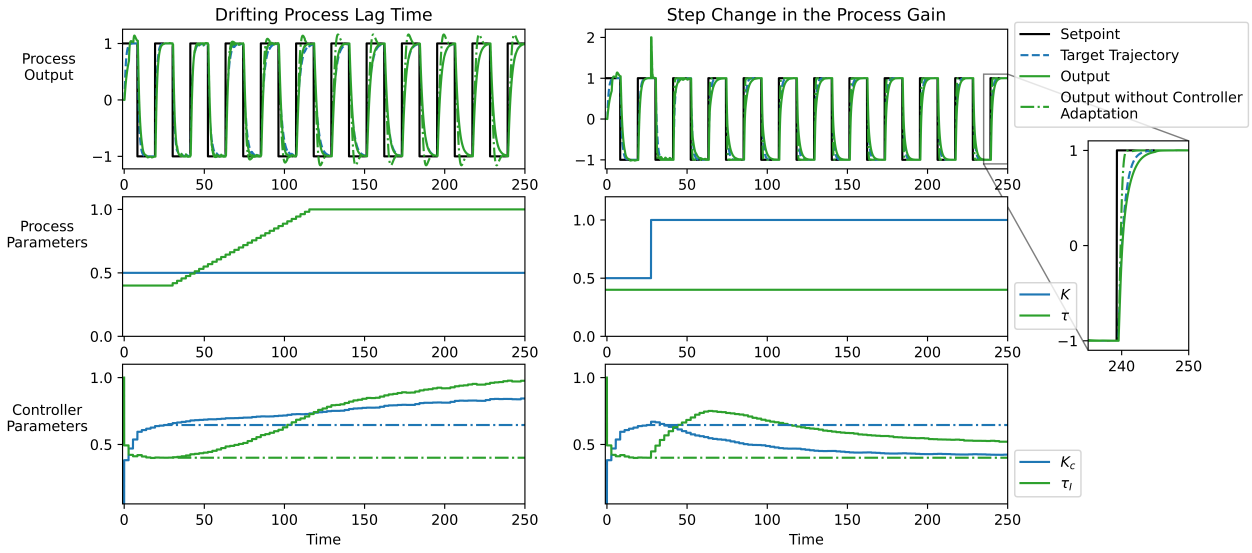


Figure 7: Meta-RL tuning with changing process dynamics. Solid lines show the controller parameters and process output with meta-RL in continuous operation. Fixing K_c and τ_I at the the values initially produced by the meta-RL algorithm produces the dash-dot lines shown for comparison.

4.3. Adaptive Control Using the Meta-RL Tuning Algorithm

In continuous operation, the proposed meta-RL tuning algorithm adapts effectively to changes in process dynamics. Such a change can be viewed as a move to a different location in the meta-RL agent's task distribution. Two sample scenarios involving significant changes to the process dynamics produce the results shown in Figure 7. In the first, τ ramps up from 0.4 to 1.0; in the second K steps up from 0.5 to 1.0. In both cases, a forgetting factor, $\gamma = 0.99$, is applied to the meta-RL agent's hidden states at each time step. (This speeds up adaptation without noticeably affecting performance.) Equation (9) can be modified to show how the forgetting factor is incorporated:

$$z_t = f(\gamma W z_{t-1} + U x_t + b) \tag{17}$$

The controller's parameters adapt to the changing system dynamics with very little disturbance to the system output (aside from an unavoidable disturbance when the process gain is suddenly doubled). In the case where the process time constant drifts, the meta-RL's adaptive tuning achieves a mean squared error of 0.006 when tracking the target trajectory through a setpoint change—a 100-fold improvement over the mean squared error of 0.0673 when there is no meta-RL adaptation. For the step change in the process gain, the meta-RL adaptive tuning achieves a mean squared error of 0.0032 while without adaptive tuning the mean squared error is 0.0290 (9 times larger).

4.4. Internal Model Representation

The good simulation results in Sections 4.1 to 4.3 suggest that information about the dynamics of the particular system being controlled must somehow be embedded in method. To validate the hypothesis that the deep hidden states encode relevant information, we apply principal component analysis (PCA) to the ultimate deep hidden states of a trained model. We collect hidden state trajectories from simulations with different process gains and time constants but a constant ratio τ/θ. At the end of the simulations, the model has had time to converge to the final PI parameters and we expect the hidden states to differ primarily because of differences in the gain and time scale involved. Therefore, we expect differences between hidden states associated with different systems to be captured by two principal components (PCs) with very little loss of information.

Figure 8 confirms this hypothesis. Two orthogonal components capture 98% of the variance in the ultimate 100-dimensional deep hidden states. Projecting the hidden state into the plane spanned by these components, we show the process gain and time constant associated with each observation. The hidden states create a near-orthogonal grid based on these two parameters, whose variations act in complementary directions. Evidently the meta-RL model's hidden states constitute an internal representation of the process dynamics derived from closed-loop process data in a model-free manner.

The bottom subplot in Figure 8 shows how the deep hidden state evolves over time during a simulation involving a particular FOPTD system. The hidden states are initialized with zeros at the start of every episode. This corresponds to a point in lower left corner of the projected principal-component space, which the top subplot associates with systems having large process gains. This association is established during the process of training the meta-RL agent, and it admits a sensible interpretation. The system has "learned" to approach an unfamiliar process by assuming it has high gain. This leads to small control moves, which are appropriate until more information can be observed and incorporated into the controller design. The deep hidden state moves to a final point whose projection is highlighted in the figure: comparing the heatmaps in earlier subplots confirms that this point is associated with the correct values of $K = 0.75$ and $\tau = 0.25$.

4.5. A simulated two-tank environment

The agent used above also performs well on a simulated version of the practical two-tank control system detailed in Lawrence et al. [31]. Notable features of this example are the following:

- The two-tank dynamics are nonlinear and slower than the FOPTD systems used for training the meta-RL agent. That is, the two-tank system is "out of distribution".

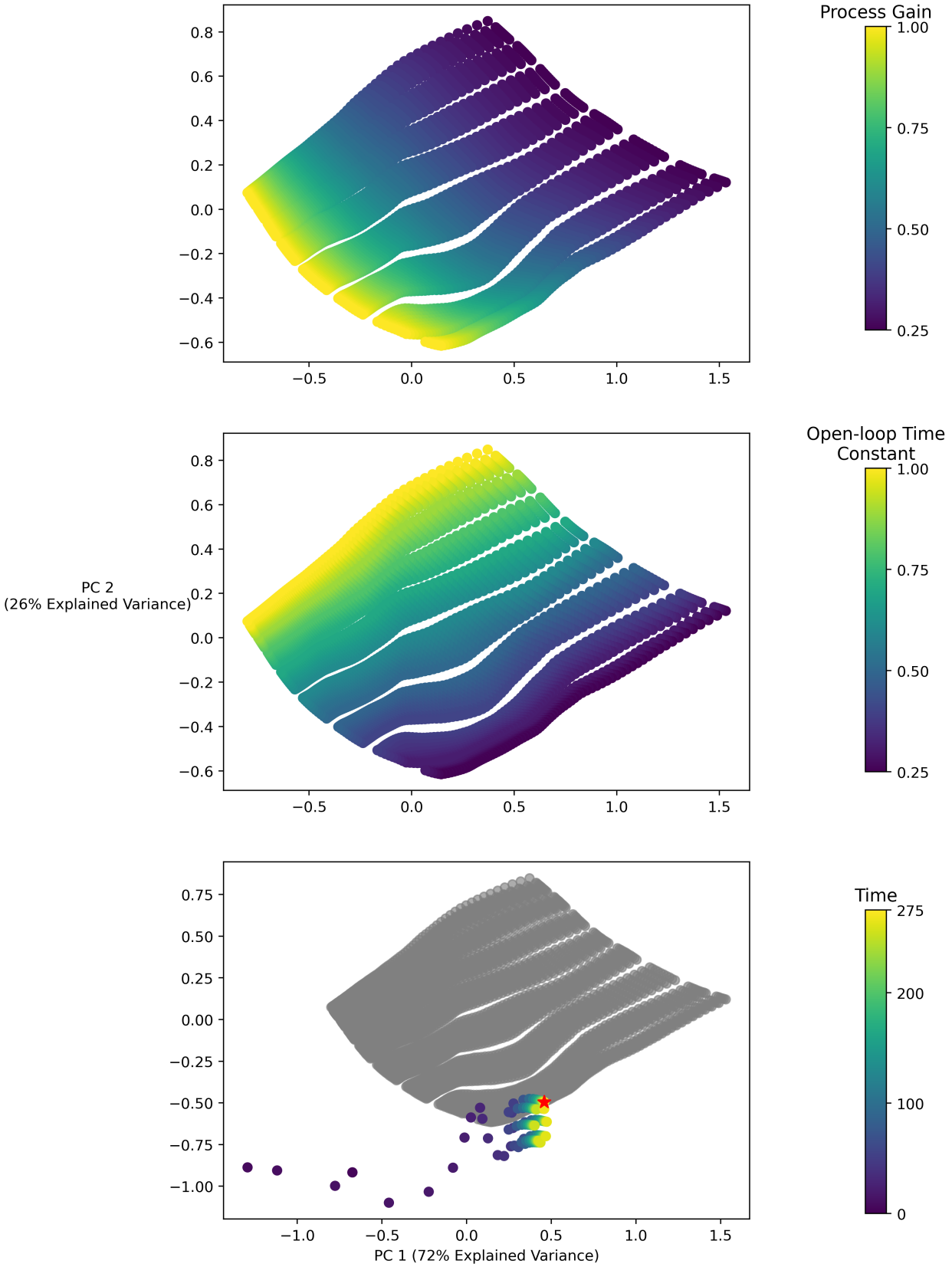


Figure 8: PCA projections of the ultimate deep hidden states from the meta-RL model after interacting with various systems. (Process gains and time constants range from 0.25 to 1.0; the ratio of lag time to dead time is approximately constant.) The top two subplots color each projected point using the process gain and time constant of the underlying system, respectively. The bottom subplot tracks the evolution of the meta-RL model's deep hidden state while interacting with the specific system where $K = 0.75$, $\tau = 0.25$, $\theta = 0.20$.

- The required operating regimes were not anticipated during training. Indeed, the meta-RL agent was only trained on step changes of ± 1 starting from 0.

We show how to apply the meta-RL agent to this novel environment despite these apparent obstacles. This simulated environment is a reasonable surrogate for a real apparatus: it is nonlinear, has a cascaded structure (for pump and flow control), and the pump, flow, level measurements are realizable through the use of filters. These dynamics are detailed in [31] but summarized here for completeness.

4.5.1. Dynamics of the two-tank system

Symbol	Value or unit	Description
r_{tank}	1.2065 (length)	Tank radius
r_{pipe}	0.125 (length)	Outflow pipe radius
$f_{\max}$	80 (volume/time)	Maximum flow
f_c	0.61	Flow coefficient
$\tau_{p,\text{in},\text{out},m}$	0.1, 0.1, 0.1, 0.2 (time)	Time constants
g	(length/time^2)	Gravitational constant
ℓ	length	Tank level
m	length	Filtered tank level
f_{in}	volume/time	Inflow
f_{out}	volume/time	Outflow
p	%	Pump speed
$\bar{p}$	%	Desired pump speed

Table 3: Parameters and variables for the two-tank system. "Tank" here refers to the upper tank. The four time constants refer to the pump speed, inflow, outflow, and measured level, respectively. Length is in decimeters (dm), time is in minutes, volume is in liters. The tank height in our simulation is 12.192 dm.

We consider the problem of controlling the liquid level in an upper tank, positioned vertically above a second tank that serves as a reservoir. Water drains from the tank into the reservoir through an outflow pipe, and is replenished by water from the reservoir delivered by a pump whose flow rate is our manipulated variable. More precisely, two PI controllers are in operation: For a desired level, one PI controller outputs the desired flow rate based on level tracking error. This flow rate is then used as a reference signal for the second PI controller, whose output is the pump speed. The first is referred to as the "level controller" and the second as the "flow controller". System parameters, values, and descriptions are given in Table 3.

The system dynamics are based on Bernoulli's equation, $f_{\text{out}} \approx f_c\sqrt{2g\ell}$, and the conser-

vation of fluid volume in the upper tank:

$$\frac{d}{dt}\left(\pi r_{\text{tank}}^2 \ell\right) = \pi r_{\text{tank}}^2 \dot{\ell} = f_{\text{in}} - f_{\text{out}}. \tag{18}$$

(We use dot notation to represent differentiation with respect to time.) Our application involves four filtered signals, with time constants τ_p for the pump, τ_{in} for changes in the inflow, τ_{out} for the outflow, and τ_m for the measured level dynamics. We therefore have the following system of differential equations describing the pump, flows, level, and measured level:

$$\tau_p \dot{p} + p = \bar{p} \tag{19}$$

$$\tau_{\text{in}} \dot{f}_{\text{in}} + f_{\text{in}} = f_{\max}\left(\frac{p}{100}\right) \tag{20}$$

$$\tau_{\text{out}} \dot{f}_{\text{out}} + f_{\text{out}} = \pi r_{\text{pipe}}^2 f_c \sqrt{2g\ell} \tag{21}$$

$$\pi r_{\text{tank}}^2 \dot{\ell} = f_{\text{in}} - f_{\text{out}} \tag{22}$$

$$\tau_m \dot{m} + m = \ell. \tag{23}$$

To track a desired level[8] $\bar{\ell}$, we can employ level and flow controllers by including the following equations:

$$\bar{p} = \text{PI}_{\text{flow}}(\bar{f}_{\text{in}} - f_{\text{in}}) \tag{24}$$

$$\bar{f}_{\text{in}} = \text{PI}_{\text{level}}(\bar{\ell} - m). \tag{25}$$

Equations (24)–(25) use shorthand for PI controllers taking the error signals $\bar{f}_{\text{in}} - f_{\text{in}}$ and $\bar{\ell} - m$, respectively. For our purposes, PI_{flow} is fixed and a part of the environment, while PI_{level} is the tunable controller.

This mathematical description is given to provide intuition for our control system. For the following results, we emphasize that the meta-RL agent was not trained on data from this environment, yet it iteratively fine-tunes the controller PI_{level}.

4.5.2. Adapting the Meta-RL Model to the Two Tank System

While the two-tank system is nonlinear, an accurate first order approximation of the dynamics relating the level in the tank to the pump flow rate setpoint is:

$$G(s) = \frac{1.7}{55s + 1} e^{-13s} \tag{26}$$

For a realistic example of how the meta-RL tuning algorithm can be used, we assume

[8] Barred variables are used to denote setpoints. For example, $\bar{\ell}$ represents the tank level setpoint.

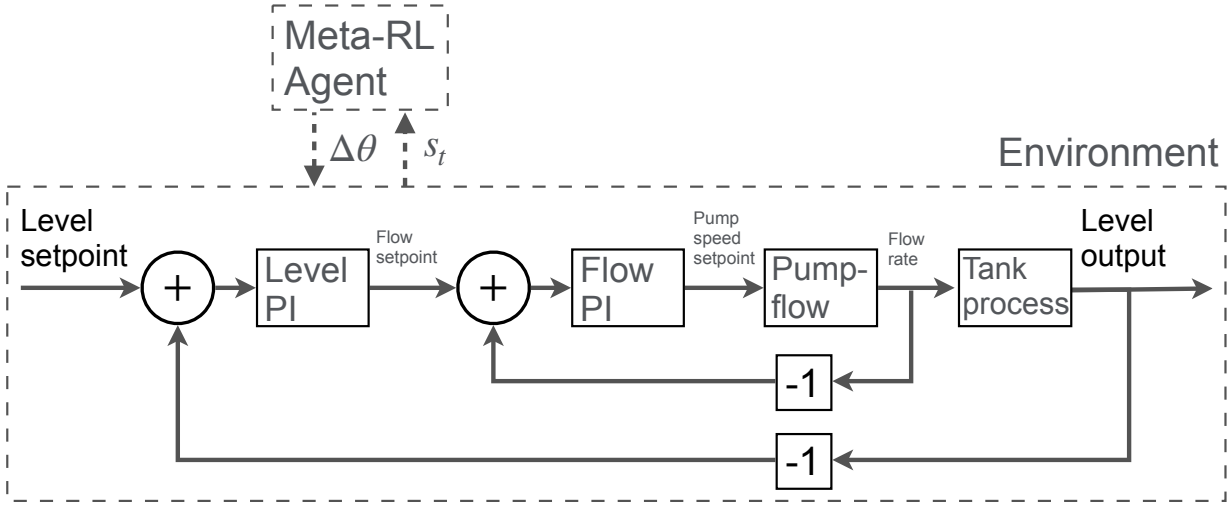


Figure 9: A schematic of the simulated nonlinear two-tank control system corresponding to Equations (19) to (25). The "pump-flow" block is modeled by Equations (19) to (20); The "tank process" is modeled by Equations (21) to (23); the level and flow controllers in Equations (24) and (25) output flow setpoints and pump speed setpoints, respectively. The meta-RL agent generates incremental changes to the PI parameters of the level controller.

only a crude approximation of the process' dynamics is available. The following crude model will be used to set up the meta-RL tuning algorithm:

$$\widehat{G}(s) = \frac{1.2}{30s + 1} \tag{27}$$

Crucially, the meta-RL agent is still interacting with the full two-tank system, given by Equations (19) to (25), and illustrated in Figure 9. This model is simply used for data processing purposes, as we demonstrate below. Equation (26) is only used to facilitate discussion about how a crude model compares to an accurate for meta-RL adaptation.

Our objective is to use the meta-RL algorithm to control the tank level around the operating region of 50–60 cm. First, we need to augment the process data to match the data distribution used to train the model (centered at 0, ranging from -1 to 1). To do this, we first apply a constant control action to bring the tank level into the desired operating region ($u = 12$ liter/min). Next, all process data has the mean (55 cm) subtracted and is scaled down by a factor of 10. This brings the data the meta-RL agent observes into alignment with its training distribution. Scaling the data also has the effect of decreasing the gain in the apparent process model (27) to 0.12.

Next, we adjust the controller gain. The meta-RL algorithm is equipped to handle systems with process gains ranging from 0.25-1.0. By scaling the controller's output by $\frac{0.5}{0.12}$, we geometrically centre the model in Equation (27) to appear to the meta-RL agent as a system with $k = 0.5$. If the estimated process gain used to set up the meta-RL agent is incorrect

by any factor between 0.5× to 2.0×, the true process gain will still fall within the task distribution. In this case, the true process gain of 1.7 appears as a process gain 0.71 to the meta-RL agent.

Next, we select an appropriate sampling time. By picking a slow sampling time, the tank's dynamics appear faster from the perspective of the meta-RL agent. To geometrically center the time constant in Equation (27) to the meta-RL's task distribution, we set the sampling time to every $\frac{30}{0.5} = 60$ seconds. The true time constant of 55 seconds then appears as a time constant of 0.92 to the meta-RL agent.

Through data augmentation, controller gain adjustment, and sampling time adjustment, the meta-RL agent's task distribution can be adapted to many "out-of-distribution" systems as long as the *magnitudes* of each parameter can be estimated.

Alternatively, if a meta-RL agent is being created for a particular application where there is a very coarse understanding of the process dynamics, the agent could be trained across a wide distribution of possible process dynamics to avoid the need for data augmentation and directly deploy the meta-RL agent on the system as in the previous examples in Sections 4.1, 4.2, 4.3. However, the advantage of direct deployment without data augmentation comes at the expense of training a meta-RL agent from scratch. Both these meta-RL approaches avoid the disadvantages of conventional RL methods: the need for very accurate estimates of the process dynamics or additional online fine-tuning to deal with plant-model mismatch.

4.5.3. Results

Figure 10 shows the tuning performance of the meta-RL agent on the two-tank system. After just one setpoint change, the meta-RL agent is able to find reasonable PI parameters for the system, demonstrating it is effective not just on true FOPTD systems, but also on nonlinear systems which can be approximated with FOPTD models. This example also contextualizes the sample efficiency of the meta-RL algorithm by providing an example with real units of time. For a system with a time constant around 1 minute and a dead time of around 13 seconds, it takes around 4 minutes for the PI parameters to nearly converge. Figure 10 also shows the performance of the meta-RL agent when there is noise of ±1 cm added to the tank level measurements. Despite the meta-RL agent not being trained on systems with noise, we see the agent's performance is not significantly affected by this change.

This case study shows that the meta-RL algorithm can apply to a very large variety of processes. While a process model is not needed for the meta-RL algorithm to work, the *magnitude* of the process gain and time constant must be known so the process data can be properly augmented. The task of scaling the gains and process dynamics needs to be automated for successful industry acceptance and this is an area for future work.

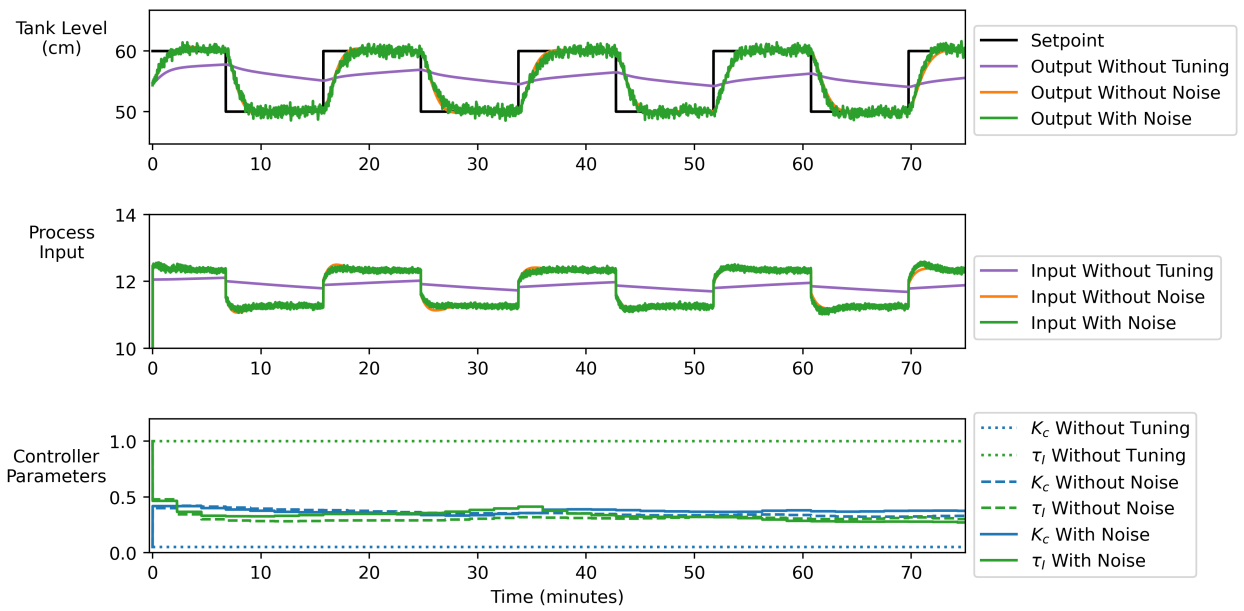


Figure 10: Performance of the meta-RL tuning algorithm for controlling the water level in a non-linear two-tank system both with and without measurement noise. The performance without the meta-RL tuning is also shown as a point of reference.

5. Conclusion

This work presents a meta-RL approach to tuning fixed-structure controllers in closed-loop without explicit system identification and demonstrates the approach using PI controllers. The method algorithm can be used to automate the initial tuning of controllers and, in continuous operation, to adaptively update controller parameters as process dynamics change over time. Assuming the magnitude of the process gain and time constant are known, the meta-RL tuning algorithm can be applied to any system which can be reasonably approximated as FOPTD[9].

A major challenge of applying RL to industrial process control is sample efficiency. The meta-RL model presented in this work addresses this problem by training a model to control a large distribution of possible systems offline in advance. The meta-RL model is then able to tune fixed-structure process controllers online with no process-specific training and no process model. There are two key design considerations which enable this performance. First is the inclusion of a hidden state in the RL agent, giving the meta-RL agent a memory it uses to learn internal representations of the process dynamics through process data. Second is constructing a value function which uses extra information in addition to the RL state. Conditioning the value function on this additional information, $V_\phi(s, \zeta)$ as opposed to $V_\phi(s)$,

[9]The present work focuses on FOPTD systems with $\tau > \theta$, however the results could be extended to dead-time dominant systems by expanding the task distribution $p(\mathcal{T})$.

improves the training efficiency of the meta-RL model.

Industrial priorities suggest further investigation of the promising meta-RL framework presented here. For example, the training procedure should incorporate noisy process data and process disturbances of the sort often seen in real-world settings. The methods stability should also be investigated more deeply. (Using RL methods to produce PI parameters rather than to provide direct control inputs has the advantage of allowing access to known stability criteria. This is clearly relevant in practice, and may also suggest a feasible approach to future theoretical work.) The versatility of the meta-RL algorithm could also be improved by adding derivative action and extending the task distribution to incorporate a greater diversity of processes, including integrating processes and processes with higher order dynamics. Moreover, the task distribution could be extended to encompass both different process dynamics and different control objectives – a complex process may require fast control for certain control loops and slower, smoother control for others. Finally, to add value to industry outside of continuous online tuning, we suggest exploring whether the meta-RL agent can be trained to identify when PID controllers should be retuned and what perturbation is needed for controller tuning (without relying on external sources of excitation, such as setpoint changes).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and Honeywell Connected Plant.

References

[1] R. S. Sutton, Learning to predict by the methods of temporal differences, Machine learning 3 (1988) 9–44.

[2] R. Nian, J. Liu, B. Huang, A review on reinforcement learning: Introduction and applications in industrial process control, Computers & Chemical Engineering (2020) 106886.

[3] C. Finn, P. Abbeel, S. Levine, Model-agnostic meta-learning for fast adaptation of deep networks, arXiv preprint arXiv:1703.03400 (2017).

[4] D. G. McClement, N. P. Lawrence, P. D. Loewen, M. G. Forbes, J. U. Backström, R. B. Gopaluni, A meta-reinforcement learning approach to process control, IFAC-PapersOnLine 54 (2021) 685–692.

[5] J. Shin, T. A. Badgwell, K.-H. Liu, J. H. Lee, Reinforcement Learning – Overview of recent progress and implications for process control, Computers & Chemical Engineering 127 (2019) 282–294. doi:10.1016/j.compchemeng.2019.05.029.

[6] J. H. Lee, J. Shin, M. J. Realff, Machine learning: Overview of the recent progresses and implications for the process systems engineering field, Computers & Chemical Engineering 114 (2018) 111–121.

[7] S. Spielberg, A. Tulsyan, N. P. Lawrence, P. D. Loewen, R. B. Gopaluni, Toward self-driving processes: A deep reinforcement learning approach to control, AIChE Journal (2019). doi:10.1002/aic.16689.

[8] J. Hoskins, D. Himmelblau, Process control via artificial neural networks and reinforcement learning, Computers & Chemical Engineering 16 (1992) 241–251. doi:10.1016/0098-1354(92)80045-B.

[9] N. S. Kaisare, J. M. Lee, J. H. Lee, Simulation based strategy for nonlinear optimal control: application to a microbial cell reactor, International Journal of Robust and Nonlinear Control 13 (2003) 347–363.

[10] J. M. Lee, J. H. Lee, Value function-based approach to the scheduling of multiple controllers, Journal of Process Control 18 (2008) 533–542. doi:10.1016/j.jprocont.2007.10.016.

[11] J. H. Lee, W. Wong, Approximate dynamic programming approach for process control, Journal of Process Control 20 (2010) 1038–1048.

[12] M. M. Noel, B. J. Pandian, Control of a nonlinear liquid level system using a new artificial neural network based reinforcement learning approach, Applied Soft Computing 23 (2014) 444–451. doi:10.1016/j.asoc.2014.06.037.

[13] S. Syafiie, F. Tadeo, E. Martinez, T. Alvarez, Model-free control based on reinforcement learning for a wastewater treatment problem, Applied Soft Computing 11 (2011) 73–82. doi:10.1016/j.asoc.2009.10.018.

[14] Y. Ma, W. Zhu, M. G. Benton, J. Romagnoli, Continuous control of a polymerization system with deep reinforcement learning, Journal of Process Control 75 (2019) 40–47. doi:10.1016/j.jprocont.2018.11.004.

[15] Y. Cui, L. Zhu, M. Fujisaki, H. Kanokogi, T. Matsubara, Factorial Kernel Dynamic Policy Programming for Vinyl Acetate Monomer Plant Model Control, in: 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE), IEEE, Munich, 2018, pp. 304–309. doi:10.1109/COASE.2018.8560593.

[16] Y. Ge, S. Li, P. Chang, An approximate dynamic programming method for the optimal control of Alkai-Surfactant-Polymer flooding, Journal of Process Control 64 (2018) 15–26. doi:10.1016/j.jprocont.2018.01.010.

[17] B. J. Pandian, M. M. Noel, Control of a bioreactor using a new partially supervised reinforcement learning algorithm, Journal of Process Control 69 (2018) 16–29. doi:10.1016/j.jprocont.2018.07.013.

[18] O. Dogru, N. Wieczorek, K. Velswamy, F. Ibrahim, B. Huang, Online reinforcement learning for a continuous space system with experimental validation, Journal of Process Control 104 (2021) 86–100. doi:10.1016/j.jprocont.2021.06.004.

[19] Y. Wang, K. Velswamy, B. Huang, A novel approach to feedback control with deep reinforcement learning, IFAC-PapersOnLine 51 (2018) 31–36.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, arXiv preprint arXiv:1707.06347 (2017).

[21] P. Petsagkourakis, I. O. Sandoval, E. Bradford, D. Zhang, E. A. del Rio-Chanona, Reinforcement learning for batch bioprocess optimization, Computers & Chemical Engineering 133 (2020) 106649.

[22] S. Fujimoto, H. Van Hoof, D. Meger, Addressing function approximation error in actor-critic methods, arXiv preprint arXiv:1802.09477 (2018).

[23] T. Joshi, S. Makker, H. Kodamana, H. Kandath, Application of twin delayed deep deterministic policy gradient learning for the control of transesterification process, arXiv:2102.13012 [cs, eess] (2021). URL: http://arxiv.org/abs/2102.13012. arXiv:2102.13012.

[24] H. Yoo, B. Kim, J. W. Kim, J. H. Lee, Reinforcement learning based optimal control of batch processes using Monte-Carlo deep deterministic policy gradient with phase segmentation, Computers & Chemical Engineering 144 (2021) 107133. doi:10.1016/j.compchemeng.2020.107133.

[25] M. Sedighizadeh, A. Rezazadeh, Adaptive PID controller based on reinforcement learning for wind turbine control, in: Proceedings of world academy of science, engineering and technology, volume 27, Citeseer, 2008, pp. 257–262.

[26] W. J. Shipman, L. C. Coetzee, Reinforcement Learning and Deep Neural Networks for PI Controller Tuning, IFAC-PapersOnLine 52 (2019) 111–116. doi:10.1016/j.ifacol.2019.09.173.

[27] A. R. Kumar, P. J. Ramadge, DiffLoop: Tuning PID controllers by differentiating through the feedback loop, in: 2021 55th Annual Conference on Information Sciences and Systems (CISS), IEEE, 2021, pp. 1–6.

[28] A. I. Lakhani, M. A. Chowdhury, Q. Lu, Stability-preserving automatic tuning of PID control with reinforcement learning, arXiv preprint arXiv:2112.15187 (2021).

[29] I. Carlucho, M. De Paula, S. A. Villar, G. G. Acosta, Incremental q-learning strategy for adaptive PID control of mobile robots, Expert Systems with Applications 80 (2017) 183–199.

[30] O. Dogru, K. Velswamy, F. Ibrahim, Y. Wu, A. S. Sundaramoorthy, B. Huang, S. Xu, M. Nixon, N. Bell, Reinforcement learning approach to autonomous PID tuning, Computers & Chemical Engineering 161 (2022) 107760.

[31] N. P. Lawrence, M. G. Forbes, P. D. Loewen, D. G. McClement, J. U. Backström, R. B. Gopaluni, Deep reinforcement learning with shallow controllers: An experimental application to PID tuning, Control Engineering Practice 121 (2022) 105046.

[32] M. R. Mowbray, R. Smith, E. A. Del Rio-Chanona, D. Zhang, Using process data to generate an optimal control policy via apprenticeship and reinforcement learning, AIChE Journal (2021). doi:10.1002/aic.17306.

[33] Y. Bao, Y. Zhu, F. Qian, A Deep Reinforcement Learning Approach to Improve the Learning Performance in Process Control, Industrial & Engineering Chemistry Research (2021) acs.iecr.0c05678. doi:10.1021/acs.iecr.0c05678.

[34] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, M. Hutter, Learning quadrupedal locomotion over challenging terrain, Science robotics 5 (2020).

[35] J. Siekmann, S. Valluri, J. Dao, L. Bermillo, H. Duan, A. Fern, J. W. Hurst, Learning memory-based control for human-scale bipedal locomotion, CoRR abs/2006.02402 (2020). URL: https://arxiv.org/abs/2006.02402. arXiv:2006.02402.

[36] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[37] M. Huisman, J. N. van Rijn, A. Plaat, A survey of deep meta-learning, Artificial Intelligence Review (2021) 1–59.

[38] V. R. Konda, J. N. Tsitsiklis, Actor-critic algorithms, in: Proceedings of the Advances in Neural Information Processing Systems, Denver, USA, 2000, pp. 1008–1014.

[39] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, D. Wierstra, Continuous control with deep reinforcement learning, arXiv Preprint, arXiv:1509.02971 (2015).

[40] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, M. Riedmiller, Deterministic policy gradient algorithms, in: International conference on machine learning, PMLR, 2014, pp. 387–395.

[41] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in: Proceedings of the Advances in Neural Information Processing Systems, 2000, pp. 1057–1063.

[42] J. Schulman, S. Levine, P. Abbeel, M. Jordan, P. Moritz, Trust region policy optimization, in: International conference on machine learning, PMLR, 2015, pp. 1889–1897.

[43] S. Bengio, Y. Bengio, J. Cloutier, J. Gescei, On the optimization of a synaptic learning rule, in: Optimality in Biological and Artificial Networks?, Routledge, 2013, pp. 281–303.

[44] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, N. De Freitas, Learning to learn by gradient descent by gradient descent, in: Advances in neural information processing systems, 2016, pp. 3981–3989.

[45] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, P. Abbeel, RL^2: Fast reinforcement learning via slow reinforcement learning, arXiv preprint arXiv:1611.02779 (2016).

[46] K. Rakelly, A. Zhou, D. Quillen, C. Finn, S. Levine, Efficient off-policy meta-reinforcement learning via probabilistic context variables, in: International conference on machine learning, 2019, pp. 5331–5340.

[47] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, M. Botvinick, Learning to reinforcement learn, arXiv preprint arXiv:1611.05763 (2016).

[48] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[49] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, arXiv preprint arXiv:1406.1078 (2014).

[50] S. Skogestad, Simple analytic rules for model reduction and PID controller tuning, Journal of process control 13 (2003) 291–309.

[51] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[52] J. Achiam, Spinning Up in Deep Reinforcement Learning, 2018.

[53] J. Schulman, P. Moritz, S. Levine, M. Jordan, P. Abbeel, High-dimensional continuous control using generalized advantage estimation, 2018. arXiv:1506.02438.

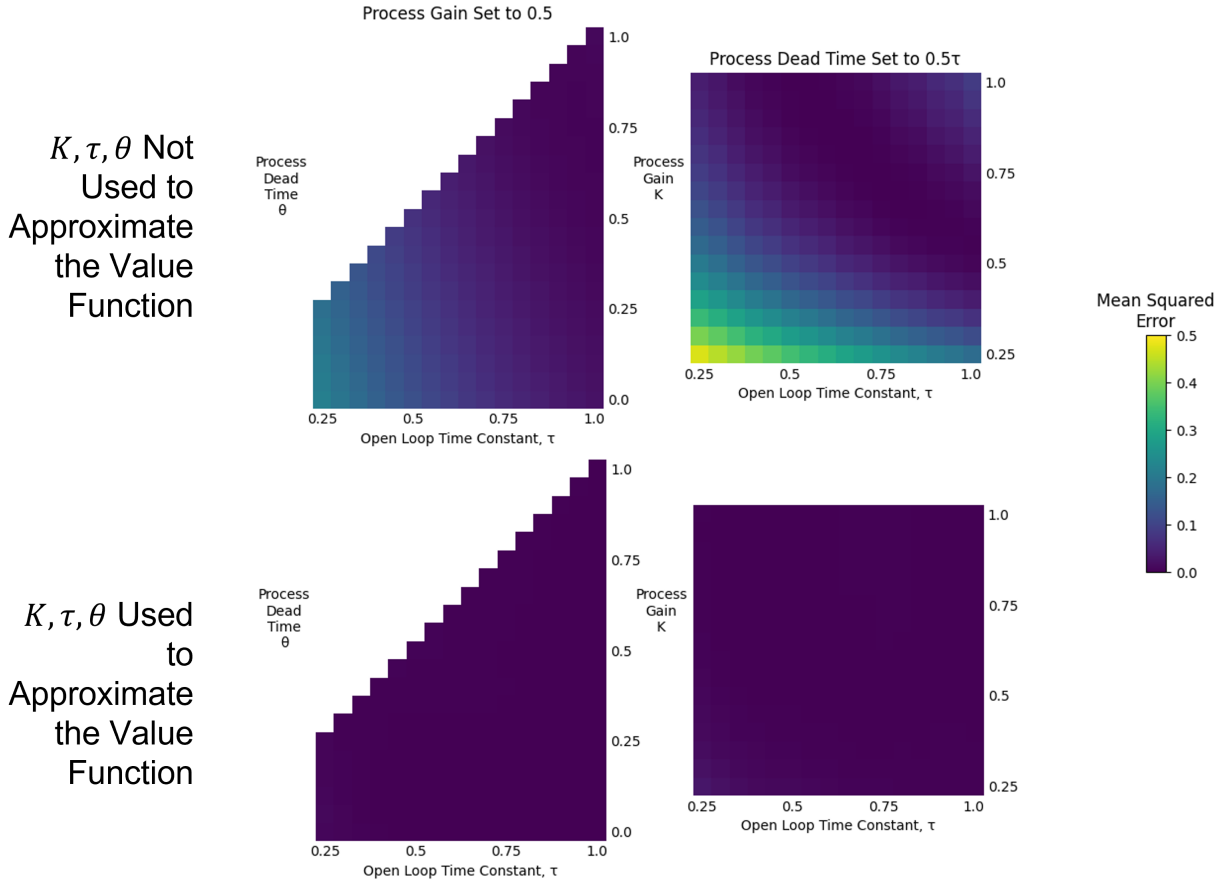
[54] C. Grimholt, S. Skogestad, The improved SIMC method for PI controller tuning, 2012. URL: http://npcw17.imm.dtu.dk/Proceedings/Session%207%20Control%20Theory/The%20improved%20SIMC%20method%20for%20PI%20controller%20tuning.pdf.

Appendix: Meta-RL Implementation Details

A.1: Hyperparameters used to train the meta-RL network.

Hidden layer size	100
Recurrent cell type	GRU
Activation function for feedforward layers	Leaky-ReLU
Optimizer	Adam
Initial learning rate	3×10^{-4}
Episode length	40 steps (110 time units)
Sequence length for backpropagation	40 steps
Training episodes per epoch	300
Epochs	2500
Discount factor*	0.99
GAE $\lambda^\star$	0.95
Policy iterations*	Up to 20
Value iterations*	40
Maximum KL divergence*	0.015
Regularization penalty on Δk_p, β_1	0.5
Regularization penalty on Δk_i, β_2	0.5

*These hyperparameters are specific to PPO or RL more generally. The reader is referred to the original PPO paper by Schulman et al. [20] for further explanation of these hyperparameters.



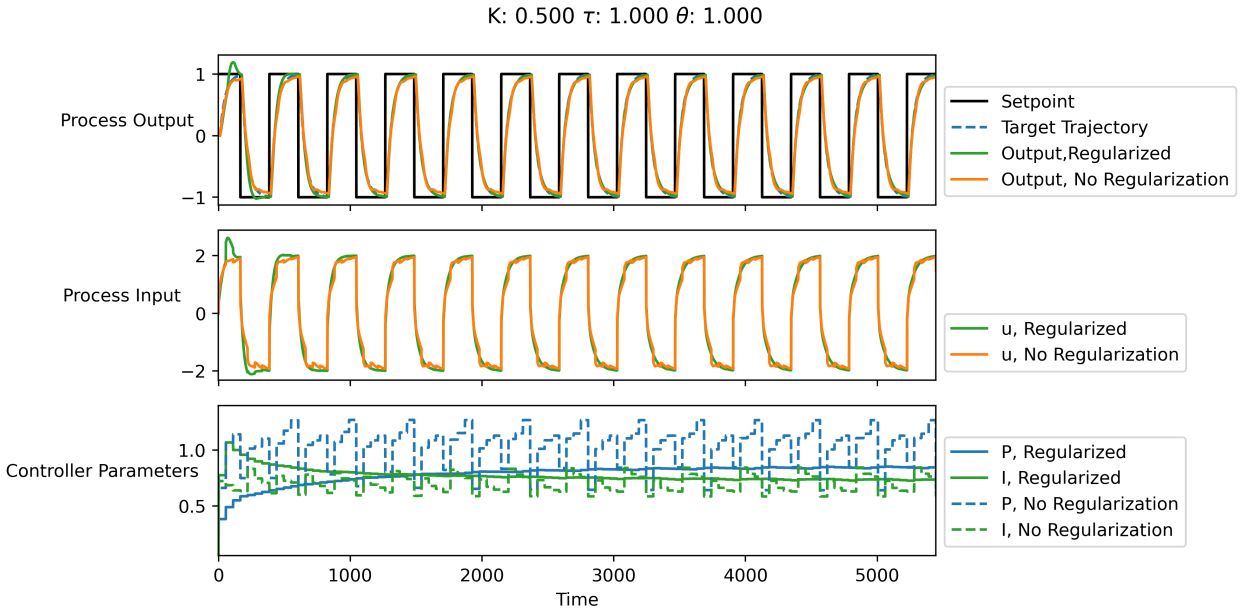
A.1: Performance of the meta-RL agent when the value function is trained without information about the process dynamics (top) vs. with this information (bottom).

Comparison of the Meta-RL Agent's Performance With and Without Information About the Process Dynamics During Training

In Section 3.2, additional information outside of the RL state s_t used to train the value function is introduced. To investigate whether this additional information influenced the meta-RL agent's performance, an ablation study is conducted. The meta-RL agent's value function is trained with and without information about the process dynamics for an equal number of epochs. The performance of each meta-RL agent is presented in Figure A1. The meta-RL agent's performance is significantly better when the value function is trained with this additional information. The meta-RL agent's worst-case setpoint tracking error as measured by the mean squared error from the target trajectory for a step change from -1 to 1 is 0.467 without this information compared to 0.030 with this information.

Sample Trajectories With and Without Regularization in the Reward Function

The cost in Equation (15) includes penalty terms proportional to the size of the parameter updates proposed by the meta-RL agent. These regularization terms aid in the convergence



A.2: Performance of the meta-RL agent with and without cost regularization. For regularized trajectories, $\beta_1 = \beta_2 = 0.5$ in Equation (15); trajectories with no regularization have $\beta_1 = \beta_2 = 0$. The system has $K = 0.5$, $\tau = 1.0$, $\theta = 1.0$.

of the PI parameters. Sample trajectories produced by the meta-RL agent trained with and without this regularization are shown for comparison.

Declaration of interests

☐The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Daniel McClement reports financial support was provided by Honeywell International Inc.

CRediT authorship contribution statement

Daniel McClement:
Conceptualization, Methodology, Formal analysis, Investigation, Software, Visualization, Writing - Original Draft, Writing - Review & Editing;

Nathan Lawrence:
Conceptualization, Software, Writing - Original Draft, Writing - Review & Editing;

Johan Backstrom:
Writing - Review & Editing;

Philip Loewen:
Project administration, Supervision, Writing - Review & Editing;

Michael Forbes:
Funding acquisition, Supervision, Writing - Review & Editing;

Bhushan Gopaluni:
Project administration, Funding acquisition, Supervision, Writing - Review & Editing.