Meta Reinforcement Learning for Adaptive Control: 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 takes advantage of known, offline information for training, such as the system gain or time constant, yet efficiently controls novel systems in a completely model-free fashion. Our meta-RL agent has a recurrent structure that accumulates "context" for its current dynamics through a hidden state variable. This end-to-end architecture enables the agent to automatically adapt to changes in the process dynamics. Moreover, the same agent can be deployed on systems with previously unseen nonlinearities and timescales. In tests reported here, the meta-RL agent was trained entirely offline, yet produced excellent results in novel settings. A key design element is the ability to leverage model-based information offline during training, while maintaining a model-free policy structure for interacting with novel environments. To illustrate the approach, we take the actions proposed by the meta-RL agent to be changes to gains of a proportional-integral controller, resulting in a generalized, adaptive, closed-loop tuning strategy. 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

[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.

controller is trained using a distribution of different processes referred to as "tasks". We aim to use this framework to develop a "universal controller" which can quickly adapt to effectively control any process rather than a single task.

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

- A simplified and improved meta-RL algorithm;
- Completely new simulation studies, including industrially-relevant examples dealing with PID controllers and nonlinear dynamics; and
- A method of leveraging known 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.

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].

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 [29, 21, 30]. Our method differs in two significant ways: the training and deployment process is simplified with our meta-RL agent through its synthesized training over a large distribution of systems. Therefore, only a single 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 any system identification or prior knowledge of the underlying dynamics. 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 [31]. 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 [32].

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 [33]. Huisman et al. [34] 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 a 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. 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

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

[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.

In this problem, $h \sim p^\pi$ refers to a typical trajectory $h = (s_0, a_0, c_0, , \ldots, s_N, a_N, c_N)$ generated by the policy π with subsequent states distributed according to p. Within the space of all possible policies, we optimize over a parameterized subset whose members are denoted π_θ. We use θ as a generic term 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) [33], or a combination of both called *actor-critic* methods [35]. 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 [36, 37, 38]. 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:

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

where $\alpha > 0$ is a step-size parameter. Analytic expressions for such a gradient exist for both stochastic and deterministic policies [33, 37]. However, in practice, approximations are 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 [39]. Using the latest policy, whose weights we denote by θ_{old}, the surrogate objective function is defined as

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

The surrogate objective function $L_{\theta_{\text{old}}}$ computes the expected return of the optimization variable, policy π_θ, over the trajectories of the most recent policy. Hence the old policy $\pi_{\theta_{\text{old}}}$ is used 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

$$J(\pi) \le L_{\theta_{\text{old}}}(\theta) + C D_{\text{KL}}^{\max}(\pi_{\theta_{\text{old}}}, \pi)$$
$$\text{where} \quad D_{\text{KL}}^{\max}(\pi, \tilde{\pi}) = \max_s D_{\text{KL}}(\pi(s)\|\tilde{\pi}(s)),$$

and that minimizing this function over θ will decrease the true objective J [39]. In practice, TRPO minimizes $L_{\theta_{\text{old}}}$ 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:

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

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 'min' functions: when A is positive, the term inside the expectation becomes $\min\left(\frac{\pi_\theta(s)}{\pi_{\theta_{\text{old}}}(s)}, 1 + \epsilon\right) A_{\theta_{\text{old}}}(s, a)$, which puts a limit on how much the objective can increase; the case when A is negative is similar. Either way, the term inside the expectation can only increase 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 $A_{\theta_{\text{old}}}$, which is then used to approximate and optimize Equation (6) using gradient ascent.

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)

[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.

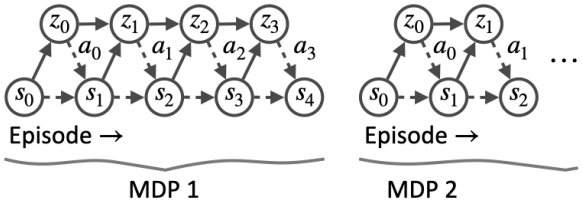

Figure 1: A diagram of the meta-RL agent's interactions with the task distribution $p(\mathcal{T})$.

[34]:

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

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 θ^* in Equation (7) is the optimal weight vector in (1) as a function of a sampled MDP $\mathcal{T}$ and the meta-weights $\boldsymbol{\Theta}$. 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 [40, 41]. Due to the shared structure among tasks in process control applications, we are interested in *context-based* meta-RL methods [42, 43, 44]. 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. [42]. We view Equation (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: $z_t = f_{\boldsymbol{\Theta}}(z_{t-1}, s_t)$. 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 $\boldsymbol{\Theta}$ is to quickly adapt a control policy for a 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; the RNN structure we use in practice is a gated recurrent network (GRU) [45]. A RNN is a special neural network structure for processing sequential data. Its basic form [46] is shown below:

$$z_t = \sigma\left(Wz_{t-1} + Ux_t + b\right) \tag{8}$$

$$o_t = Vz_t + c. \tag{9}$$

Here W, U, V, b, c are trainable weights, while x_t is some input to the network and o_t is the output. σ is a nonlinear function; in the case of the GRU structure used in the present work, a sigmoid function is used. A RNN can be thought of as a nonlinear state-space system that is optimized for some objective. The main point is that the characteristic feature of any type of RNN is the hidden state, which evolves alongside sequential input data.

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):

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

where k is the process gain, τ the time constant, θ is the time delay, and s is the Laplace variable (not to be confused with s_t, which represents the RL state at time step t). Such models are often good low-order approximations for the purposes of PI tuning [47]. The formulation in continuous time is tidy, but in practice we of course discretize Equation (10).

A PI controller has the form:

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

alternatively written as:

$$C(s) = k_p + k_i\frac{1}{s}, \tag{12}$$

where K_c, τ_I, k_p, and k_i are tuning parameters. The PI parameters in (11) are more commonly used and results are reported in terms of these parameters, however the parameter definitions in (12) are used to train the meta-RL agent due to the improved numerical stability gained by using an integral gain parameter rather than an integral time constant parameter in the RL state[4].

[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.

Prior work on RL for PI tuning suggests an update scheme of the form [28]:

$$[k_p, k_i] \leftarrow [k_p, k_i] + \alpha \nabla J([k_p, k_i]) \tag{13}$$

$$= [k_p, k_i] + \Delta[k_p, k_i] \tag{14}$$

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) 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 the velocity form of PI controllers or full PID controllers. The MDP state contains the current PI parameters as well as the proportional setpoint error and the integral setpoint error calculated from the beginning of an episode, t_0, to the current time step, t.

$$s_t = \left[k_p, k_i, e_t, \int_{t_0}^{t} e_\tau d\tau\right] \tag{15}$$

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 (16). The target trajectory is calculated by applying a first order filter to the setpoint signal. 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. An L1 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).

$$c_t = (y_{desired,t} - y_t)^2 + \beta|\Delta k_p| + \beta|\Delta k_i|, \tag{16}$$

$$Y_{desired}(s) = \frac{y_{sp}}{2\tau s + 1} e^{-\theta s} \tag{17}$$

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 novel "in distribution"[5] systems without any online training.
- The meta-RL agent is a single model that is trained once, offline, meaning one does not 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.

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 as 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

[5] "in-distribution" systems are defined as processes within the task distribution $p(\mathcal{T})$ the meta-RL agent was trained across. As shown in Section 4.5, nearly any system can be modified to be "in-distribution".

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

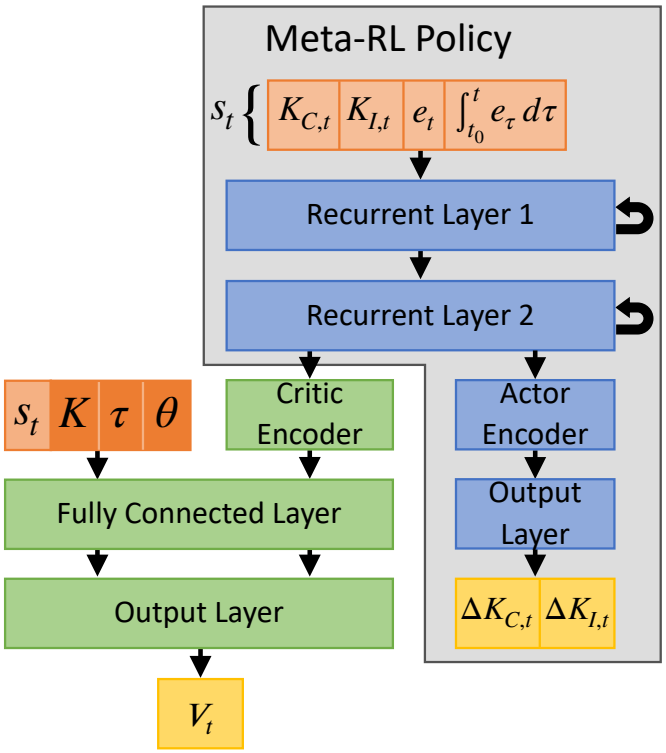

Figure 2: The structure of the RL agent. The control policy used online is shown in the grey box while the critic used during offline training is shown in green.

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 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 $\tau_I = 1.0$. The state of the system is randomly initialized near zero and the set point 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 on the order of milliseconds or hours).

Table 1 shows the distribution of FOPTD model parameters uniformly sampled from during training. In Section 4.5 we demonstrate how training across this range of parameters can be quite versatile in practice using data augmentation.

There are two main limitations to the size of the task distribution the meta-RL agent can effectively be trained across. Firstly, neural network training works best when the features they are trained on 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). Secondly, 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 set point change or disturbance would occur between time steps and not be 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.

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

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

Algorithm 1 shows the procedure used to train the meta-RL agent. The PPO algorithm is adopted from Open AI's "Spinning Up" implementation and modified to accomodate the inclusion of a recurrent neural network and distribution of control tasks [48].

[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.

Algorithm 1 Meta-RL Controller Training
Adapted from OpenAI's PPO implementation documentation.

Input: Initial meta-policy parameters Θ_0, initial value function parameters ϕ_0

for each training episode **do**
  Sample a batch of tasks $\mathcal{T}_{1:n} \sim p(\mathcal{T})$
  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_Θ on task $\mathcal{T}_i$
    Store h in $\mathcal{D}_k$
  end for
  Compute advantage estimates, $\hat{A}_t$ using generalized advantage estimation [49] 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:
  $\Theta_{k+1} = \arg\min_\Theta \frac{1}{|\mathcal{D}_k|T} \sum_{h\in\mathcal{D}_k} \sum_{t=0}^{T} \min(\frac{\pi_\Theta(a_s|s_t)}{\pi_{\Theta_k}(a_s|s_t)} A^{\pi_{\Theta_k}}(s_t, a_t), g$
  Update the value function to estimate the cost-to-go of an episode using gradient descent:
  $\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{h\in\mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t, \zeta_t) - \hat{R}_t)^2$
end for

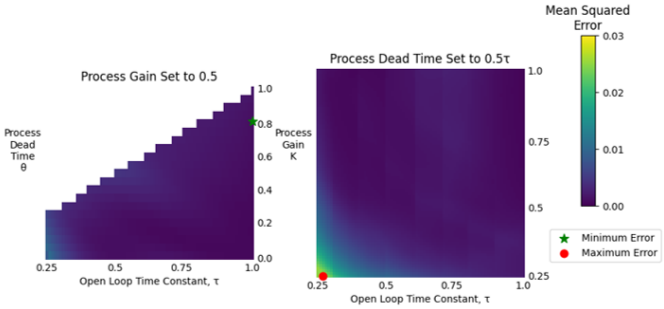

Figure 3: Mean squared error of FOPTD system outputs compared to their respective target trajectories after the meta-RL agent interacts with the system until its PI parameters converge. The horizontal lines in the left heatmap and vertical lines in the right heatmap are a result of the discretization of the time delay during simulations.

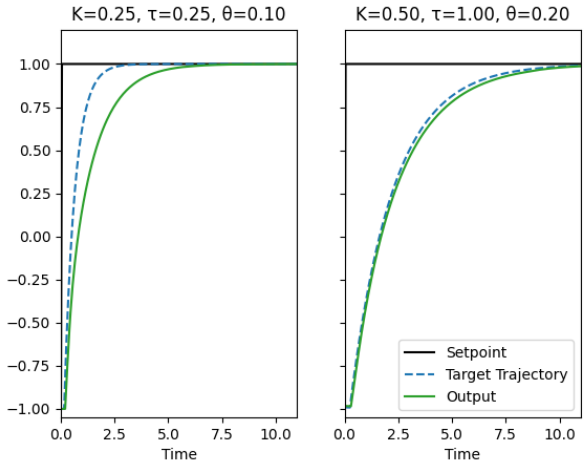

Figure 4: System output trajectories for a set point change from −1 to 1 using the meta-RL algorithm's PI tunings compared to the target trajectories. The worst-case (left) and best-case (right) are selected from the heatmaps in Figure 3. No noise is included in the simulation to more clearly illustrate the trajectories.

4. Experimental results

4.1. Asymptotic Performance of the Meta-RL Tuning Algorithm

Figure 3 shows the asymptotic performance of the meta-RL tuning algorithm as measured by the mean squared error from the target trajectory for a set point change from −1 to 1 and gives a cross-sectional view of how the model performs across the task distribution. There are three parameters which define the process dynamics so we hold k or the ratio $\frac{\tau}{\theta}$ constant so we can visualize the results in two dimensions.

Overall, the tuning algorithm is able to closely match the target output for any system from its distribution. Performance decreases slightly for systems where both the process gain and the time constant are small. A possible explanation for low performance in this region lies in the cost function formulation. The cost function in Equation (16) penalizes the agent proportional to $|\Delta k_p|$ and $|\Delta k_i|$. Systems with small process gains and time constants require the largest controller gains. An unintended effect of the cost function may be that it incentivizes the slight undertuning of such systems as the slight decrease in target trajectory tracking error is outweighed by the penalty incurred for further increasing the controller gains past a certain point within the finite time horizon of a training episode. In other words, the slight drop in performance may be a result of a slight misalignment of the meta-RL algorithm's objective from the control engineer's objective.

Figure 4 shows the performance in the worst-case and best-case scenarios based on target trajectory tracking performance selected from Figure 3. We see even in the worst-case scenario, the meta-RL algorithm's PI tunings provide desirable control performance.

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 converge to ±10% of their ultimate values. The convergence of the tunings is dependent on the excitation in the system. In our experiments, excitation was created by set point changes every 11 units of time. The convergence speed could be increased with more excitation (conversely, it can be slower with less excitation). 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 set point change).

Systems with large process gains and fast dynamics converge quickest, requiring just a single set point change (around 10

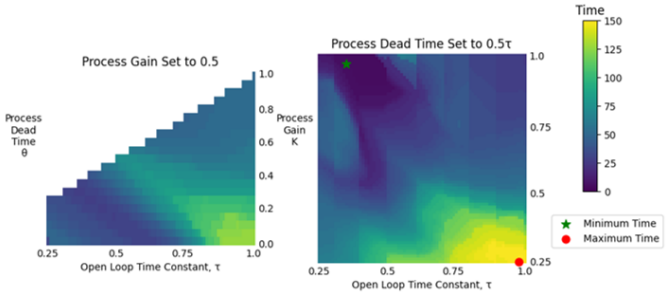

Figure 5: Online time required for both the k_p and k_i parameters to reach ±10% of their ultimate values.

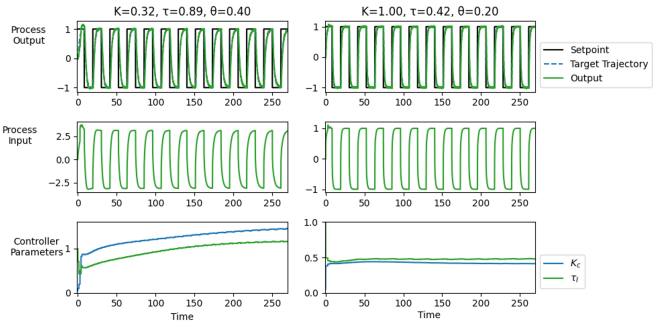

Figure 6: System output trajectories showing the converge of the controller's PI parameters over time. The worst-case (left) and best-case (right) are selected from the heatmaps in Figure 5.

units of time). Systems with small gains and slow dynamics take longer to converge, requiring 13 set point 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 set point 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 set point change. The performance continues to improve with time to more closely match the target trajectory.

4.3. Adaptive Control Using the Meta-RL Tuning Algorithm

A promising use case for a meta-RL tuning algorithm is adaptive process control. If the process dynamics change over time or if a process moves to a new operating region with different dynamics, this can be viewed as moving to a different region of the meta-RL agent's task distribution. Figure 7 shows the performance of the meta-RL tuning algorithm in response to significant changes to the process dynamics (τ ramping up from 0.4 to 1.0 or k having a step change from 0.5 to 1.0). In these examples, a forgetting factor, $\gamma = 0.99$, is applied to the meta-RL agent's hidden states at each time step as we empirically observed this to speed up adaptation without noticeably effecting performance. Equation (9) can be modified to show how the forgetting factor is incorporated:

$$z_t = \sigma(\gamma W z_{t-1} + U x_t + b) \tag{18}$$

We see the controller's parameters adapt to the changing sys-

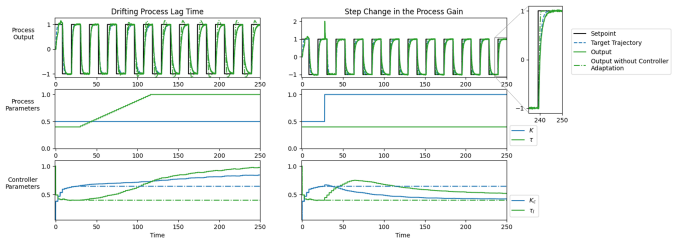

Figure 7: System output trajectories showing the response of the tuning algorithm to changes in the process dynamics. Dashed lines show the results if no adaptive tuning is performed after the initial tuning by the meta-RL algorithm.

tem dynamics with very little disturbance to the system output (aside from an unavoidable disturbance when the process gain is suddenly doubled).

4.4. Validation of the Meta-RL Algorithm's Internal Model Representation

Based on the simulation results in Sections 4.1 to 4.3 and the recurrent structure of the meta-RL algorithm, we expect the model's hidden states to encode information about the system dynamics which allow it to tailor its PI tunings to the specific system it is interacting with. To validate this theory, we perform principal component analysis (PCA) on the ultimate deep hidden states after interacting with different FOPTD processes. We perform PCA on hidden states taken from simulations with different process gains and time constants but a constant ratio $\frac{\tau}{\theta}$. At the end of the simulations, the model has had time to converge to the final PI parameters and we expect the only differences in the hidden states between different simulations to be related to the process gain and process time scale. Therefore, differences between hidden states of different systems are expected to reduce to two principal components (PCs) with very little loss of information.

Figure 8 confirms this hypothesis. Two components capture 98% of the variance in the ultimate deep hidden states. Looking at the PCA trends with respect to the process gain and time constant, we see the hidden states create a near-orthogonal grid based on these two parameters. The meta-RL model's hidden states allow it to create an internal representation of the process dynamics through closed-loop process data in a model-free manner.

In Figure 8 we also analyze how the deep hidden state evolves over time throughout a simulation. The hidden states are initialized with zeros at the start of every episode and, interestingly, this corresponds to a point in the principal component-space opposite the region of the principal component-space corresponding to systems with small gains and small time constants. This makes sense from a robustness perspective: the PI parameters for these systems are the largest and there is a greater risk in assuming a system has a small gain and small time constant rather than assuming a large gain and large time constant until more information can be collected. We see the meta-RL develops a very sensible internal representation of FOPTD systems.

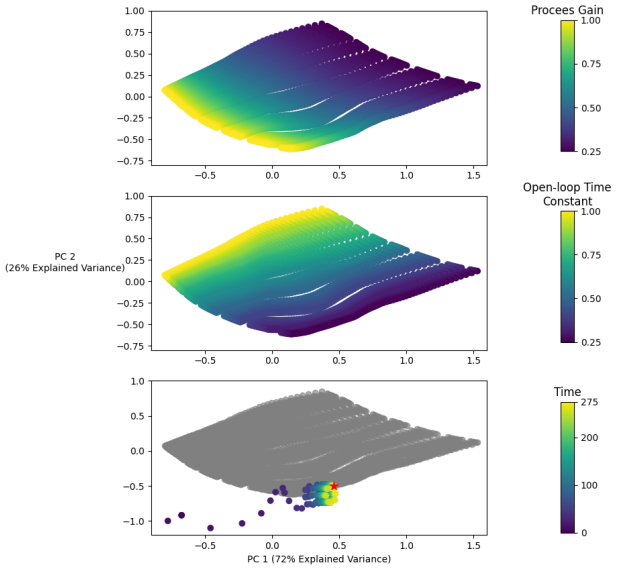

Figure 8: PCA results on the ultimate deep hidden states from the meta-RL model after interacting with systems with process gains and time constants ranging from 0.25 to 1.0 and a near-constant lag time to dead time ratio. The gaps in the grid are due to the discretization of the dead time used in simulations. The bottom plot illustrates the time evolution of the meta-RL model's deep hidden state through the low-dimensional principal component-space while interacting with a FOPTD system ($K = 0.75$, $\tau = 0.25$, $\theta = 0.20$).

4.5. A simulated two-tank environment

This example demonstrates an industrial testbed for the meta-RL agent. Using the same agent used in the previous results, we apply it to a simulated two-tank system. 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".
- We apply the meta-RL agent in novel operating regions not seen during training; the meta-RL agent was only trained on step changes centred at 0 with a magnitude of ±1.

We show how to apply the meta-RL agent to this novel environment despite all these practical differences. 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 reported in Lawrence et al. [28] but conveyed here for completeness.

4.5.1. Description of the dynamics

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 2.

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
$\tau_{p,\text{in,out},m}$	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 2: 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.

The system dynamics are based on Bernoulli's equation, $f_{\text{out}} \approx f_c \sqrt{2g\ell}$, and the conservation of fluid volume in the upper tank:

$$\frac{d}{dt}\left(\pi r_{\text{tank}}^2 \ell\right) = \pi r_{\text{tank}}^2 \dot{\ell} = f_{\text{in}} - f_{\text{out}}. \tag{19}$$

(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:

$$\tau_p \dot{p} + p = \bar{p} \tag{20}$$

$$\tau_{\text{in}} \dot{f}_{\text{in}} + f_{\text{in}} = f_{\max}\left(\frac{p}{100}\right) \tag{21}$$

$$\tau_{\text{out}} \dot{f}_{\text{out}} + f_{\text{out}} = \pi r_{\text{pipe}}^2 f_c \sqrt{2g\ell} \tag{22}$$

$$\pi r_{\text{tank}}^2 \dot{\ell} = f_{\text{in}} - f_{\text{out}} \tag{23}$$

$$\tau_m \dot{m} + m = \ell. \tag{24}$$

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

$$\bar{p} = \text{PI}_{\text{flow}}(\bar{f}_{\text{in}} - f_{\text{in}}) \tag{25}$$

$$\bar{f}_{\text{in}} = \text{PI}_{\text{level}}(\bar{\ell} - m). \tag{26}$$

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

Equation (25)–Equation (26) 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

An accurate first order approximation of the tank level dynamics around the operating region is:

$$G(s) = \frac{1.7}{55s + 1} e^{-13s} \tag{27}$$

However, assuming we have very limited information about the true dynamics of the system, we will use the following crude model to set up the meta-RL tuning algorithm:

$$\hat{G}(s) = \frac{1.2}{30s + 1} \tag{28}$$

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 apparent controller gain in Equation (28) 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 (28) 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 (28) 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 very non-conservative 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.

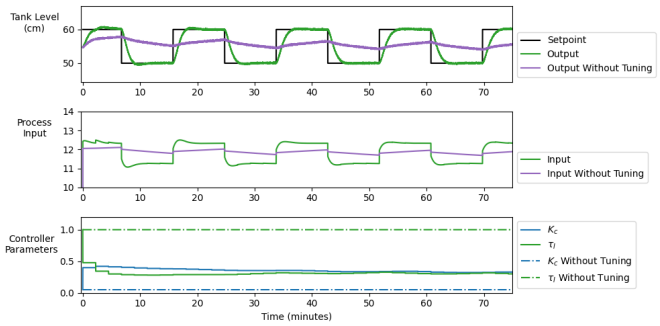

Figure 9: Performance of the meta-RL tuning algorithm for controlling the water level in a non-linear two-tank system. The performance without the meta-RL tuning is shown as a point of reference.

4.5.3. Results

Figure 9 shows the tuning performance of the meta-RL agent on the two-tank system. After just one set point 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.

From this case study, we see that the meta-RL algorithm can apply to a very large variety of processes. There is just one a caveat to the "model free" claim. 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 something we intend to work on.

5. Conclusion

In this work, we presented a meta-RL model capable of tuning fixed-structure controllers in closed-loop without explicit system identification and demonstrated our approach using PI controllers. The tuning algorithm can be used to help automate the initial tuning of controllers or maintenance of controllers by adaptively updating the controller parameters as process dynamics change over time. Assuming the magnitude of the process gain and time constant are known, the meta-RL tuning al-

gorithm can be applied to any system which can be reasonably approximated as a FOPTD system[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 $V_\phi(s)$, improves the training efficiency of the meta-RL model.

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.

[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 $p(\mathcal{T})$ used during training.

[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] 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.

[27] 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.

[28] N. P. Lawrence, M. G. Forbes, P. D. Loewen, D. G. McClement, J. U. Backstrom, R. B. Gopaluni, Deep reinforcement learning with shallow controllers: An experimental application to PID tuning, arXiv preprint arXiv:2111.07171 (2021).

[29] 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.

[30] 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.

[31] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, M. Hutter, Learning

quadrupedal locomotion over challenging terrain, Science robotics 5 (2020).
[32] 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.
[33] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.
[34] M. Huisman, J. N. van Rijn, A. Plaat, A survey of deep meta-learning, Artificial Intelligence Review (2021) 1–59.
[35] 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.
[36] 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).
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] 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.
[42] 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).
[43] 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.
[44] 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).
[45] 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).
[46] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.
[47] S. Skogestad, Simple analytic rules for model reduction and PID controller tuning, Journal of process control 13 (2003) 291–309.
[48] J. Achiam, Spinning Up in Deep Reinforcement Learning, 2018.
[49] J. Schulman, P. Moritz, S. Levine, M. Jordan, P. Abbeel, High-dimensional continuous control using generalized advantage estimation, 2018. arXiv:1506.02438.

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

*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.