Preprints of the 21st IFAC World Congress (Virtual)
Berlin, Germany, July 12-17, 2020

Optimal PID and Antiwindup Control Design as a Reinforcement Learning Problem

Nathan P. Lawrence * **Gregory E. Stewart** ***
Philip D. Loewen * **Michael G. Forbes** ****
Johan U. Backstrom **** **R. Bhushan Gopaluni** **

* *Department of Mathematics, University of British Columbia, Vancouver, BC V6T 1Z2, Canada (e-mail: lawrence@math.ubc.ca, loew@math.ubc.ca).*
** *Department of Chemical and Biological Engineering, University of British Columbia, Vancouver, BC V6T 1Z3, Canada (e-mail: bhushan.gopaluni@ubc.ca)*
*** *Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC V6T 1Z4, Canada (e-mail: stewartg@ece.ubc.ca)*
**** *Honeywell Process Solutions, North Vancouver, BC V7J 3S4, Canada (e-mail: michael.forbes@honeywell.com, johan.backstrom@honeywell.com)*

Abstract: Deep reinforcement learning (DRL) has seen several successful applications to process control. Common methods rely on a deep neural network structure to model the controller or process. With increasingly complicated control structures, the closed-loop stability of such methods becomes less clear. In this work, we focus on the interpretability of DRL control methods. In particular, we view linear fixed-structure controllers as shallow neural networks embedded in the actor-critic framework. PID controllers guide our development due to their simplicity and acceptance in industrial practice. We then consider input saturation, leading to a simple nonlinear control structure. In order to effectively operate within the actuator limits we then incorporate a tuning parameter for anti-windup compensation. Finally, the simplicity of the controller allows for straightforward initialization. This makes our method inherently stabilizing, both during and after training, and amenable to known operational PID gains.

Keywords: neural networks, reinforcement learning, actor-critic networks, process control, PID control, anti-windup compensation

1. INTRODUCTION

The performance of model-based control methods such as model predictive control (MPC) or internal model control (IMC) relies on the accuracy of the available plant model. Inevitable changes in the plant over time result in increased plant-model uncertainty and decreased performance of the controllers. Model reidentification is costly and time-consuming, often making this procedure impractical and less frequent in industrial practice.

Reinforcement learning (RL) is a branch of machine learning in which the objective is to learn an optimal policy (controller) through interactions with a stochastic environment (Sutton and Barto, 2018). Only somewhat recently has RL been successfully applied in the process industry (Badgwell et al., 2018). The first successful implementations of RL methods in process control were developed in the early 2000s. For example, Lee and Lee (2001, 2008) utilize approximate dynamic programming (ADP) methods for optimal control of discrete-time nonlinear systems. While these results illustrate the applicability of RL in controlling discrete-time nonlinear processes, they are also limited to processes for which at least a partial model is available or can be derived through system identification.

Other approaches to RL-based control use a fixed control structure such as PID. With applications to process control, Brujeni et al. (2010) develop a model-free algorithm to dynamically assign the PID gains from a pre-defined collection derived from IMC. On the other hand, Berger and da Fonseca Neto (2013) dynamically tune a PID controller in continuous parameter space using the actor-critic method, where the actor is the PID controller; the approach is based on dual heuristic dynamic programming, where an identified model is assumed to be available. The actor-critic method is also employed in Sedighizadeh and Rezazadeh (2008), where the PID gains are the actions at each time-step.

In the recent work of Spielberg et al. (2019), an actor-critic architecture based on the deep deterministic policy gradient (DDPG) algorithm due to Lillicrap et al. (2015) is implemented to develop a model-free, input-output controller

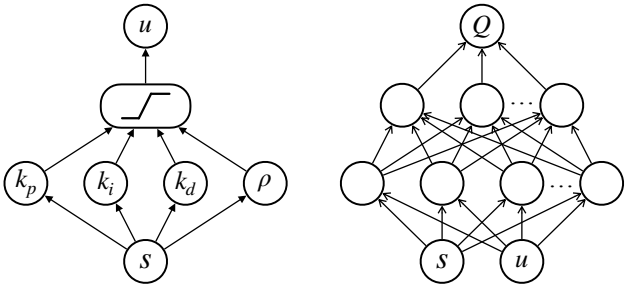

Fig. 1. The actor (PID controller) on the left is a linear combination of the state and the PID & anti-windup parameters followed by a nonlinear saturation function. The critic on the right is a deep neural network approximation of the Q-function whose inputs are the state-action pair generated by the actor.

for set-point tracking problems of discrete-time nonlinear processes. The actor and critic are both parameterized by ReLU deep neural networks (DNNs). At the end of training, the closed-loop system includes a plant together with a neural network as the nonlinear feedback controller. The neural network controller is a black-box in terms of its stabilizing properties. In contrast, PID controllers are widely used in industry due to their simplicity and interpretability. However, PID tuning is also known to be a challenging nonlinear design problem, making it an important and practical baseline for RL algorithms.

To this end, we present a simple interpretation of the actor-critic framework by expressing a PID controller as a shallow neural network (figure 1 illustrates the proposed framework). The PID gains are the weights of the actor network. The critic is the Q-function associated with the actor, and is parameterized by a DNN. We then extend our interpretation to include input saturation, making the actor a simple nonlinear controller. Input saturation can lead to integral windup; we therefore incorporate a new tuning parameter for anti-windup compensation. Finally, the simplicity of the actor network allows us to initialize training with hand-picked PID gains, for example, with SIMC (Skogestad, 2001). The actor is therefore initialized as an operational, interpretable, and industrially accepted controller that is then updated in an optimal direction after each roll-out (episode) in the plant. Although a PID controller is used here, the interpretation as a shallow neural network applies for any linear fixed-structure controller.

This paper is organized as follows: Section 2 provides a brief description of PID control and anti-windup compensation. Section 3 frames PID tuning in the actor-critic architecture and describes our methodology and algorithm. Finally, section 4 shows simulation results in tuning a PI controller as well as a PI controller with anti-windup compensation.

2. PID CONTROL AND INTEGRAL WINDUP

We use the parallel form of the PID controller:

$$u(t) = k_p e_y(t) + k_i \int_0^t e_y(\tau)d\tau + k_d \frac{d}{dt} e_y(t). \qquad (1)$$

Here $e_y(t) = \bar{y}(t) - y(t)$ is the difference between the output $y(t)$ and a given reference signal $\bar{y}(t)$. To implement the PID controller it is necessary to discretize in time. Let $\Delta t > 0$ be a fixed sampling time. Then define $I_y(t_n) = \sum_{i=1}^{n} e_y(t_i)\Delta t$, where $0 = t_0 < t_1 < \ldots < t_n$, and $D(t_n) = \frac{e_y(t_n) - e_y(t_{n-1})}{\Delta t}$. We then use u to refer to the discretized version of (1), written as follows

$$u(t_n) = k_p e_y(t_n) + k_i I_y(t_n) + k_d D(t_n). \qquad (2)$$

We note that the velocity form of a PID controller could also be used in the following sections. However, it is simpler and more common to explain our anti-windup strategy with the form of equation (2). Further, the velocity form is more sensitive to noise (exploration noise) added to the input because it gets carried over to subsequent time-steps.

Despite their simplicity, PID controllers are difficult to tune for a desired performance. Popular strategies for PID tuning include relay tuning (e.g., Åström and Hägglund (1984)) and IMC (e.g., Skogestad (2001)). This difficulty can be exacerbated when a PID controller is implemented on a physical plant due to the limitations of an actuator. In the next section, we describe how such limitations can be problematic, then introduce a practical and simple method for working within these constraints.

2.1 Anti-Windup Compensation

A controller can become saturated when it has maximum and minimum constraints on its control signal and is given a set-point or a disturbance that carries the control signal outside these limits. If the actuator constraints are given by two scalars $u_{\min} < u_{\max}$, then we define the saturation function to be

$$\mathrm{sat}(u) = \begin{cases} u_{\min}, & \text{if } u < u_{\min} \\ u, & \text{if } u_{\min} \le u \le u_{\max} \\ u_{\max}, & \text{if } u > u_{\max}. \end{cases} \qquad (3)$$

If saturation persists, the controller is then operating in open-loop and the integrator continues to accumulate error at a non-diminishing rate. That is, the integrator experiences *windup*. This creates a nonlinearity in the controller and can destabilize the closed-loop system. Methods for mitigating the effects of windup are referred to as *anti-windup* techniques. For a more detailed overview of the windup phenomenon and simple anti-windup techniques, the reader is referred to Åström and Rundqwist (1989).

In this paper, we focus on one of the earliest and most basic anti-windup methods called *back-calculation* (Fertik and Ross, 1967). Back-calculation works in discrete-time by feeding into the control signal a scaled sum of past deviations of the actuator signal from the unsaturated signal. The nonnegative scaling constant, ρ, governs how quickly the controller unsaturates (that is, returns to the interval $(u_{\min}, u_{\max})$). Precisely, we define $e_u(t) = \mathrm{sat}(u(t)) - u(t)$ and $I_u(t_n) = \sum_{i=1}^{n-1} e_u(t_i)\Delta t$, then we redefine the PID controller in (2) to be the following

$$u(t_n) = k_p e_y(t_n) + k_i I_y(t_n) + k_d D(t_n) + \rho I_u(t_n) \qquad (4)$$

From (3) it is clear that if the controller is operating within its constraints, then (4) emits the same control signal as (2). Otherwise, the difference $\mathrm{sat}(u) - u$ adds negative feedback to the controller if $u > u_{\max}$, or positive feedback if $u < u_{\min}$. Further, (4) agrees with (2) when

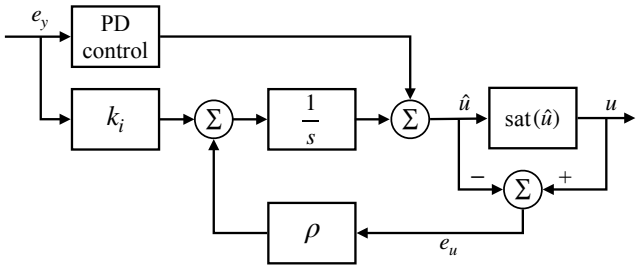

Fig. 2. The back-calculation scheme feeds the scaled difference between the saturated input signal and that suggested by a PID controller back into the integrator.

$\rho = 0$; therefore, the recovery time of the controller to the operating region $[u_{\min}, u_{\max}]$ is slower the closer ρ is to zero and more aggressive when ρ is large. A scheme of this approach is shown in figure 2 and the effect of the parameter ρ is shown in figure 7 in Section 4.

3. PID IN THE REINFORCEMENT LEARNING FRAMEWORK

Our method for PID tuning stems from the state-space representation of (4) followed by the input saturation:

$$\begin{bmatrix} e_y(t_n) \\ I_y(t_n) \\ D(t_n) \\ I_u(t_n) \end{bmatrix} = \begin{bmatrix} 0 & 0\ 0\ 0 \\ 0 & 1\ 0\ 0 \\ -1/\Delta t & 0\ 0\ 0 \\ 0 & 0\ 0\ 1 \end{bmatrix} \begin{bmatrix} e_y(t_{n-1}) \\ I_y(t_{n-1}) \\ D(t_{n-1}) \\ I_u(t_{n-1}) \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ \Delta t & 0 \\ 1/\Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} e_y(t_n) \\ e_u(t_n \end{bmatrix} \tag{5}$$

$$\hat{u}(t_n) = [k_p\ \ k_i\ \ k_d\ \ \rho] \begin{bmatrix} e_y(t_n) \\ I_y(t_n) \\ D(t_n) \\ I_u(t_n) \end{bmatrix} \tag{6}$$

$$u(t_n) = \text{sat}\big(\hat{u}(t_n)\big). \tag{7}$$

Equation (5) describes the computations necessary for implementing a PID controller in discrete time steps. On the other hand, (6) parameterizes the PID controller. We therefore take (6) and (7) to be a shallow neural network, where $[k_p\ k_i\ k_d\ \rho]$ is a vector of trainable weights and the saturation is a nonlinear activation.

In the next section we outline how RL can be used to train these weights without a process model. The overview of RL provided here is brief. For a thorough tutorial of RL in the context of process control the reader is referred to the paper by Spielberg et al. (2019). Further, the general DDPG algorithm we employ is introduced by Lillicrap et al. (2015).

3.1 Overview of Tuning Objective

The fundamental components of RL are the policy, the objective, and the environment. We assume the environment is modeled by a Markov decision process with action space $\mathcal{U}$ and state space $\mathcal{S}$. Therefore, the environment is modeled with an initial distribution $p(s_0)$ with a transition distribution $p(s_{n+1}|s_n, u_n)$, where $s_0, s_n, s_{n+1} \in \mathcal{S}$ and $u_n \in \mathcal{U}$. Here, we define $s_n = [e_y(t_n)\ I_y(t_n)\ D(t_n)\ I_u(t_n)]^T$ and $u_n \in \mathcal{U}$ refers to the saturated input signal given by (7) at time t_n. The vector of parameters in (6) is referred to as K. Formally, the PID controller with anti-windup compensation in (7) is given by the mapping $\mu(\cdot, K)\colon \mathcal{S} \to \mathcal{U}$ such that

$$u_n = \mu(s_n, K) \tag{8}$$

Each interaction the controller (8) has with the environment is scored with a scalar value called the *reward*. Reward is given by a function $r : \mathcal{S} \times \mathcal{U} \to \mathbb{R}$; we use r_n to refer to $r(s_n, u_n)$ when the corresponding state-action pair is clear. We use the notation $h \sim p^{\mu}(\cdot)$ to denote a trajectory $h = (s_1, u_1, r_1, \ldots, s_N, u_N, r_N)$ generated by the policy μ, where N is a random variable called the *terminal time*.

The goal of RL is to find a controller, namely, the weights K, that maximizes the expectation of future rewards over trajectories h:

$$J(\mu(\cdot, K)) = \mathbb{E}_{h\sim p^{\mu}(\cdot)}\left[\sum_{n=1}^{\infty} \gamma^{n-1} r(s_n, \mu(s_n, K))\middle| s_0\right] \tag{9}$$

where $s_0 \in \mathcal{S}$ is a starting state and $0 \leq \gamma \leq 1$ is a *discount* factor. Our strategy is to iteratively maximize J via stochastic gradient ascent, as maximizing J corresponds to finding the optimal PID gains. Optimizing this objective requires additional concepts, which we outline in the next section.

3.2 Controller Improvement

Equation (9) is referred to as the *value function* for policy μ. Closely related to the cost function is the *Q-function*, or *state-action value function*, which considers state-action pairs in the conditional expectation:

$$Q(s_n, u_n) = \mathbb{E}_{h\sim p^{\mu}(\cdot)}\left[\sum_{k=n}^{\infty} \gamma^{k-n} r(s_k, \mu(s_k, K))\middle| s_n, u_n\right] \tag{10}$$

Returning to our objective of maximizing (9), we employ the policy gradient theorem for deterministic policies (Silver et al., 2014):

$$\begin{aligned} &\nabla_K J(\mu(\cdot, K)) = \\ &\mathbb{E}_{h\sim p^{\mu}(\cdot)}\big[\nabla_u Q(s_n, u)|_{u=\mu(s_n,K)} \nabla_K \mu(s_n, K)\big]. \end{aligned} \tag{11}$$

We note that Eq. (9) is maximized only when the policy parameters K are optimal, which then leads to the update scheme

$$K \leftarrow K + \alpha \nabla_K J(\mu(\cdot, K)), \tag{12}$$

where $\alpha > 0$ is the learning rate.

3.3 Deep Reinforcement Learning

The optimization of J in line (12) relies on knowledge of the Q-function (10). We approximate Q iteratively using a deep neural network with training data from replay memory (RM). RM is a fixed-size collection of tuples of the form (s_n, u_n, s_{n+1}, r_n). Concretely, we write a parametrized Q-function, $Q(\cdot, \cdot, W_c) : \mathcal{S} \times \mathcal{U} \to \mathbb{R}$, where W_c is a collection of weights. This framework gives rise to a class of RL methods known as *actor-critic* methods. Precisely, the actor-critic methods utilize ideas from policy gradient methods and Q-learning with function approximation (Konda and

Tsitsiklis, 2000; Sutton et al., 2000). Here, the actor is the PID controller given by (8) and the critic is $Q(\cdot, \cdot, W_c)$.

3.4 Actor-Critic Initialization

An advantage of our approach is that the weights for the actor can be initialized with user-specified PID gains. For example, if a plant is operating with known gains k_p, k_i, and k_d, then these can be used to initialize the actor. The idea is that these gains will be updated by stochastic gradient ascent in the approximate direction leading to the greatest expected reward. The quality of the gain updates then relies on the quality of the Q-function used in (12). The Q-function is parameterized by a deep neural network and is therefore initialized randomly. Both the actor and critic parameters are updated after each roll-out with the environment. However, depending on the number of time-steps in each roll-out, this can lead to slow learning. Therefore, we continually update the critic during the roll-out using batch data from RM.

3.5 Connections to Gain Scheduling

As previously described, the actor (PID controller) is updated after each episode. We are, however, free to change the PID gains at each time-step. In fact, previous approaches to RL-based PID tuning such as Brujeni et al. (2010) and Sedighizadeh and Rezazadeh (2008) dynamically change the PID gains at each time-step. There are two main reasons for avoiding this whenever possible. First, the PID controller is designed for set-point tracking and is an inherently intelligent controller that simply needs to be improved subject to the user-defined objective (reward function); that is, it does not need to 'learn' how to track a set-point. Second, when the PID gains are free to change at each time-step, the policy essentially functions as a gain scheduler. This switching of the control law creates nonlinearity in the closed-loop, making the stability of the overall system more difficult to analyze. This is true even if all the gains or controllers involved are stabilizing (Stewart, 2012). See, for instance, example 1 of Malmborg et al. (1996).

Of course, gain scheduling is an important strategy for industrial control. The main point here is that RL-based controllers can inherit the same stability complications as gain scheduling. In the next subsection, we demonstrate the effect of updating the actor at different rates on a simple linear system.

4. SIMULATION RESULTS

In our examples we refer to several different versions of the DDPG algorithm which are differentiated based on how frequently the actor is updated: **V1** updates the actor at each time-step, while **V2** updates the actor at the end of each episode. See Appendix A for implementation details.

For our purposes, we define the reward function to be

$$r(s_n, u_n) = -\big(|e_y(t_n)|^p + \lambda |u_n|\big), \tag{13}$$

where $p \in \{1, 2\}$ and $\lambda \geq 0$ are fixed during training. An episode ends either after 200 time-steps or when the actor tracks the set-point for 10 time-steps consecutive time-steps.

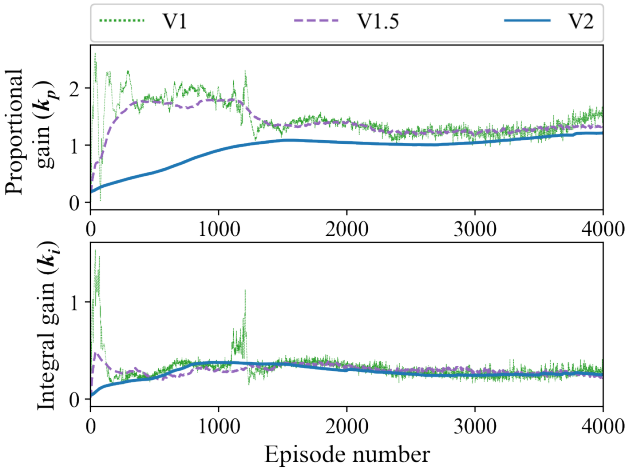

Fig. 3. (top) k_p parameter values at the end of each episode; (bottom) similarly, the k_i parameter values. Green corresponds to updating the PI parameters at each time-step; purple corresponds to an update every 10th time-step; blue corresponds to a single update per episode. The color scheme is consistent throughout the example.

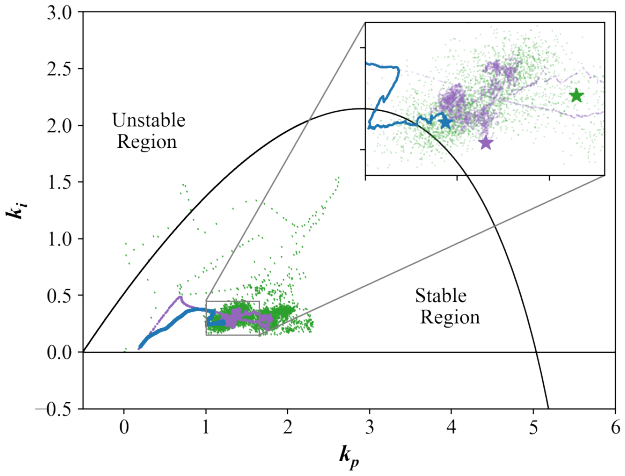

Fig. 4. A scatter plot of the data shown in figure 3. The black curve indicates the boundary of stability in the parameter plane. Stars show the $k_p - k_i$ coordinate at the end of training for its respective color.

4.1 Example 1

Consider the following continuous-time transfer function:

$$G(s) = \frac{2e^{-s}}{6s + 1}. \tag{14}$$

We discretize (14) with time-steps of 0.1 seconds. In this example, we initialize a PI controller with gains $k_p = 0.2, k_i = 0.05$. The following results are representative of other initial PI gains. Note, however, we cannot set $k_p = k_i = 0$ because this forces $\mu(\cdot, K) \equiv 0$ and the parameters will not change between updates.

In our experiments we implement algorithms **V1** and **V2**. We also consider "**V1.5**", in which the PID parameters are updated every tenth time-step. Note that the fundamental difference between **V1** and **V2** is that the former

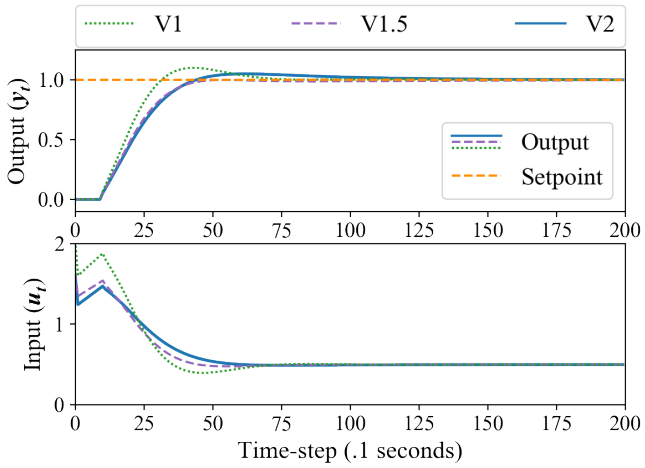

Fig. 5. (top) Output signal; (bottom) Input signal. The colors correspond to the respective final PI gains shown in figure 3.

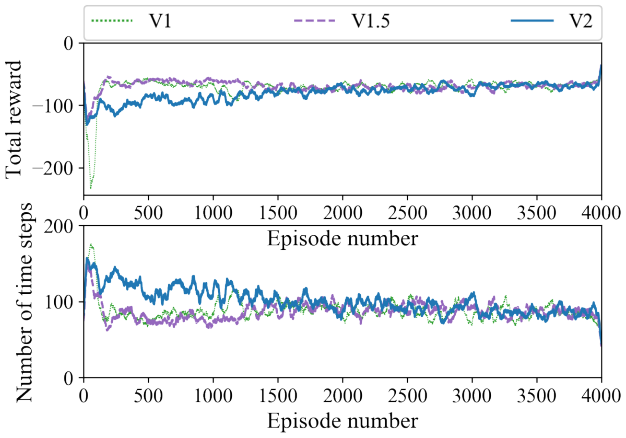

Fig. 6. (top) Moving average of total reward per episode; (bottom) Moving average of number of time-steps per episode before the PI controller tracked within 0.1 for 10 consecutive time-steps.

corresponds to an online implementation of the algorithm, while the latter can be seen as an offline version. **V1.5** represents a dwell time in the learning algorithm.

Figure 3 only shows the value of k_p, k_i at the end of each episode for each implementation. Nonetheless all three implementations reach approximately the same values; the final closed-loop step responses for each implementation are shown in figure 5.

Another way of visualizing the PID parameters is in the $k_p - k_i$ plane. We plot the boundary separating the stable and unstable regions using the parametric curve formulas due to Saeki (2007). Figure 4 is a scatter plot with the k_p, k_i value at the end of each episode along with the aforementioned boundary curve. We note that the stability regions refer to the closed-loop with (14) and a fixed $k_p - k_i$ point, rather than the nonlinear system induced by updating $k_p - k_i$ values online.

We see in figure 6 that all three implementations achieve similar levels of performance as measured by the reward function (13) ($\lambda = 0.50$). Although, **V1** and **V1.5** plateau

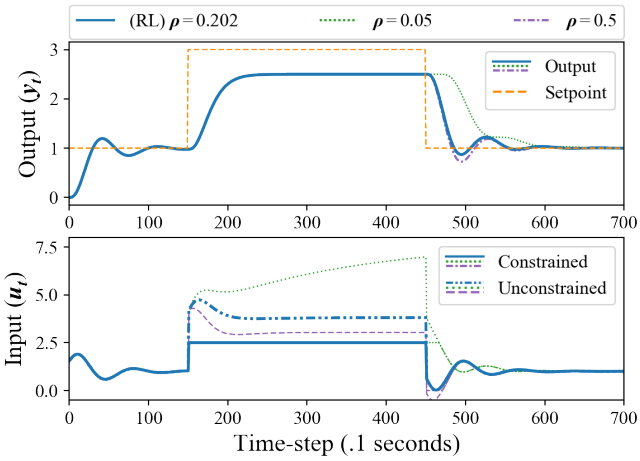

Fig. 7. (top) The output response corresponding to various values of ρ; (bottom) The colors correspond to the ρ values at the top, while the dashed lines show what the input signal would be without the actuator constraint.

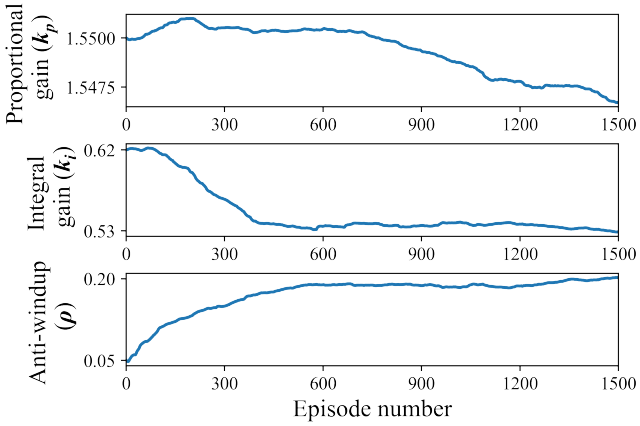

Fig. 8. (top) k_p value after each episode; (middle) k_i value; (bottom) ρ value.

sooner than **V2**, the initial dip in reward from **V1** can be explained by the sequence of unstable $k_p - k_i$ values around the boundary curve in figure 4.

Finally, we note that **V1** and **V1.5** reach their peak performances after approximately 25 minutes (real-time equivalent) of operation. In our experiments, the actor learning rate α had the most drastic effect on the convergence speed. Here, we show the results for a relatively small α (see Appendix A) to clearly capture the initial upheaval of the parameter updates as well as the long-term settling behavior. In principle, we could omit the latter aspect and simply stop the algorithm, for example, once the reward reaches a certain average threshold.

4.2 Example 2

In this example, we incorporate an anti-windup tuning parameter and employ algorithm **V2**. Consider the following transfer function:

$$G(s) = \frac{1}{(s+1)^3}. \tag{15}$$

In order to tune ρ, it is necessary to saturate the input: If the actor always operates within the actuator limits, then

$$\frac{\partial \mu}{\partial \rho} \equiv 0 \tag{16}$$

because $I_u \equiv 0$, meaning ρ will never be updated in (12). This can be understood from figure 7 at the bottom, as there is a non-zero difference between the dashed and solid lines only after the first step change (this corresponds to the difference shown in figure 2). Further, although (16) also holds for states s_n corresponding to input saturation, which therefore do not contribute to the update in (12), we still store them in RM for future policy updates.

In our experiment, the set-point is initialized to 1, then switches to 1.5 (plus a small amount of zero-mean Gaussian noise), then switches back to 1. The switches occur at varying time-steps. At the beginning of an episode, with 10% probability, the switches set-point is set to 3 instead of 1.5. Figure 7 shows a slower recovery time for smaller ρ and a more aggressive recovery for larger ρ values.

We emphasize that the actor can be initialized with hand-picked parameters. To illustrate this, we initialize k_p and k_i using the SIMC tuning rules due to Skogestad (2001). Figure 8 shows little change in the k_p parameter, while k_i and ρ adjust significantly, leading to a faster integral reset and smoother tracking than the initial parameters.

5. CONCLUSION

In this work, we relate well-known and simple control strategies to more recent methods in deep reinforcement learning. Our novel synthesis of PID and anti-windup compensation with the actor-critic framework provides a practical and interpretable framework for model-free, DRL-based control design with the goal of being implemented in a production control system. Recent works have employed actor-critic methods for process control using ReLU DNNs to express the controller; our work then establishes the simplest, nonlinear, stabilizing architecture for this framework. In particular, any linear control structure with actuator constraints may be used in place of a PID.

ACKNOWLEDGEMENTS

We would like to thank Profs. Benjamin Recht and Francesco Borrelli of University of California, Berkeley for insightful and stimulating conversations. We would also like to acknowledge the financial support from Natural Sciences and Engineering Research Council of Canada (NSERC) and Honeywell Connected Plant.

REFERENCES

Åström, K.J. and Hägglund, T. (1984). Automatic tuning of simple regulators with specifications on phase and amplitude margins. *Automatica*, 20(5), 645–651.

Åström, K.J. and Rundqwist, L. (1989). Integrator windup and how to avoid it. In *1989 American Control Conference*, 1693–1698. IEEE.

Badgwell, T.A., Lee, J.H., and Liu, K.H. (2018). Reinforcement learning–overview of recent progress and implications for process control. In *Computer Aided Chemical Engineering*, volume 44, 71–85. Elsevier.

Berger, M.A. and da Fonseca Neto, J.V. (2013). Neurodynamic programming approach for the PID controller adaptation. *IFAC Proceedings Volumes*, 46(11), 534–539.

Brujeni, L.A., Lee, J.M., and Shah, S.L. (2010). *Dynamic tuning of PI-controllers based on model-free reinforcement learning methods*. IEEE.

Fertik, H.A. and Ross, C.W. (1967). Direct digital control algorithm with anti-windup feature. *ISA transactions*, 6(4), 317.

Konda, V.R. and Tsitsiklis, J.N. (2000). Actor-critic algorithms. In *Proceedings of the Advances in Neural Information Processing Systems*, 1008–1014. Denver, USA.

Lee, J.M. and Lee, J.H. (2001). Neuro-dynamic programming method for mpc1. *IFAC Proceedings Volumes*, 34(25), 143–148.

Lee, J.M. and Lee, J.H. (2008). Value function-based approach to the scheduling of multiple controllers. *Journal of process control*, 18(6), 533–542.

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

Malmborg, J., Bernhardsson, B., and Åström, K.J. (1996). A stabilizing switching scheme for multi controller systems. *IFAC Proceedings Volumes*, 29(1), 2627–2632.

Saeki, M. (2007). Properties of stabilizing PID gain set in parameter space. *IEEE Transactions on Automatic Control*, 52(9), 1710–1715.

Sedighizadeh, M. and Rezazadeh, A. (2008). Adaptive PID controller based on reinforcement learning for wind turbine control. In *Proceedings of world academy of science, engineering and technology*, volume 27, 257–262. Citeseer.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning*. Beijing, China.

Skogestad, S. (2001). Probably the best simple PID tuning rules in the world. In *AIChE Annual Meeting, Reno, Nevada*, volume 77.

Spielberg, S., Tulsyan, A., Lawrence, N.P., Loewen, P.D., and Bhushan Gopaluni, R. (2019). Toward self-driving processes: A deep reinforcement learning approach to control. *AIChE Journal*, 65(10), e16689.

Stewart, G.E. (2012). A pragmatic approach to robust gain scheduling. *IFAC Proceedings Volumes*, 45(13), 355–362.

Sutton, R.S. and Barto, A.G. (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, R.S., McAllester, D.A., Singh, S.P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the Advances in Neural Information Processing Systems*, 1057–1063.

Appendix A. IMPLEMENTATION DETAILS

In example 1, we use the Adam optimizer to train the actor and critic. To demonstrate simpler optimization methods, we train the actor in example 2 using SGD with momentum (decay constant 0.75, learning rate decrease $O(1/\sqrt{n})$) and gradient clipping (when magnitude of gradient exceeds 1). Adam, RMSprop, and SGD all led to similar results in all examples. The actor and critic networks were trained using `TensorFlow` and the processes were simulated in discrete time with the Control Systems Library for `Python`. The hyperparameters in the DDPG algorithm used across all examples are as follows: Mini-batch size $M = 256$, RM size 10^5, discount factor $\gamma = 0.99$, initial learning rate for both actor and critic is 0.001. The critic is modeled by a 64×64 ReLU DNN. The saturation function in (7) can be modeled with $\mathrm{ReLU}(x) = \max\{0, x\}$:

$$\mathrm{sat}(u) = \mathrm{ReLU}\big(-\mathrm{ReLU}(u_{\max} - u) + u_{\max} - u_{\min}\big) + u_{\min}.$$