Chemical Engineering Research and Design 232 (2026) 654–697

Contents lists available at ScienceDirect

Chemical Engineering Research and Design

journal homepage: www.elsevier.com/locate/cherd

Constraint-informed reinforcement learning in feedback control of nonlinear processes: A tutorial overview and a new design

Arthur Khodaverdian [a], Xiaodong Cui [a], Panagiotis D. Christofides [a,b,*]

[a] Department of Chemical and Biomolecular Engineering, University of California, Los Angeles, CA 90095-1592, USA
[b] Department of Electrical and Computer Engineering, University of California, Los Angeles, CA 90095-1592, USA

ARTICLE INFO

Keywords:
Reinforcement learning
Model predictive control
Lyapunov stability
Nonlinear systems
Chemical processes

ABSTRACT

Process control systems operate with strict constraints that are necessary for optimal and safe process operation. To ensure the satisfaction of these constraints, nonlinear constrained optimal control techniques such as model predictive control are typically deployed; however, these iterative schemes are computationally expensive, limiting the scalability of the framework. Alternatively, neural network-based control methods provide a lightweight approach to approximate constrained optimal control, but lack closed-loop stability guarantees. Although methods exist that inject differentiable but non-trainable operations such as quadratic programming problems into neural networks in the form of implicit layers, such methods ensure constraint satisfaction but fail to enable constraint-informed training. Under strict constraints, these layers yield zero gradients in the violation direction, which nullifies the neural network controller's ability to learn the impact of constraint violations on its objective. Additionally, these layers reintroduce iterative algorithms, and thus the computational feasibility concerns.

This work explores a lightweight means of ensuring that stability guarantees are inherently satisfied by neural network-based control policies while also allowing for approximate constraint violation impacts to be learned. Whereas constraint satisfaction can be ensured by external projection, the resulting effective policy would be suboptimal relative to a policy that is trained to optimize the objective while strictly satisfying constraints. Quadratic programming layers allow for constraints to be enforced inside neural networks as a differentiable layer, thereby allowing for constrained optimization during training, but suffer from zero gradients when constraint violations such as control actuator bounds are encountered during the forward pass. We propose an alternative approach that uses the Heaviside step function with a softplus-based straight-through estimator to yield a gating mechanism that enforces a Lyapunov-based stability constraint through internal control fallback. This fallback-based gating mechanism allows for non-zero gradients during constraint violations and, under sufficient constraint satisfaction, exhibits minimal mismatch between the forward and backward pass. Additionally, since the constraint enforcement is done through a simple gate, there is no significant computational impact as would be true in the quadratic programming layer approach. Our approach allows for the network to both enforce the constraint internally while also learning via gradients how this constraint enforcement impacts the objective and consequently how to adjust weights to optimize the objective while satisfying the constraints.

To demonstrate this, Actor-Critic reinforcement learning is used with the proposed gating mechanism integrated into the Actor. A quadratic programming-based reference is designed for comparison alongside a standard feedforward neural network as the baseline to demonstrate the impact of the architecture on the resulting closed-loop behavior. Additionally, a model predictive controller and fallback proportional controller are used for comparison of cost-optimality and computational efficiency during closed-loop simulation. The controllers are applied to a benchmark nonlinear chemical process example to demonstrate their feasibility and relative performance. Results show that the gate-based constraint enforcement achieves closed-loop performance that closely matches or exceeds that of the non-CENNC and model predictive control baseline while consistently exceeding that of the quadratic programming-based reference and fallback controller with minimal impact on computation time.

Overall, this work demonstrates that, relative to quadratic programming-based approaches, a straight-through estimator matched with strict gating allows for efficient inference, stable training, and generalizable

* Corresponding author.
E-mail addresses: tailsmsn@g.ucla.edu (A. Khodaverdian), cuixiaodong@g.ucla.edu (X. Cui), pdc@seas.ucla.edu (P.D. Christofides).

https://doi.org/10.1016/j.cherd.2026.07.018
Received 22 June 2026; Received in revised form 3 July 2026; Accepted 6 July 2026
Available online 19 July 2026

performance. The presented framework simplifies the approach to stability guarantees by leveraging Lyapunov-stability theory in a manner that enables the framework to be applied to large-scale systems with minimal training and inference time computational impacts relative to unconstrained methods.

1. Introduction

As the need for advanced control systems grows across various domains, Model Predictive Control (MPC) has remained a uniquely well-suited solution to general constrained optimal control problems. MPC as a framework operates through internal simulation of process dynamics through a mathematical model of the process which is used for evaluation and refinement of a control action trajectory until constraint and quality margins are met. This design allows it to find local optima that optimize the desired objective and satisfy constraints simultaneously as opposed to frameworks that may need to sacrifice optimality of the objective for constraint satisfaction (Qin and Badgwell, 2003). The consequence of this broad capability is suboptimal scalability. The nonlinear constrained optimal control problems that MPC solves are NP-hard problems. Even in the linear convex case, where known global optima can be found through various approaches, the computational time complexity of MPC remains roughly $\mathcal{O}\left(n^4\right)$ (Pardalos and Vavasis, 1991; Peng et al., 2024); however, MPC is not restrained to a single nonlinear solver. Yet, even for computationally efficient methods such as the Sequential Least Squares Programming (SLSQP) method, which breaks the problem down into least squares subproblems, it is found that the subproblems alone have time complexities of $\mathcal{O}\left(n^2\right)$ or $\mathcal{O}\left(n^3\right)$ (Gill et al., 1979; Kraft, 1988). Thus, it is observed that MPC as a framework suffers from suboptimal scaling laws due to this computationally intensive iterative scheme.

To counteract this problem, there has been much research done on the topic of improving MPC. Although the scope of the research is broad, the key topics can be categorized into either approaches that reduce the problem's effective scale or approaches that reduce the solver's effective computational burden. Reducing the scale of the problem only proportionally shifts the scale of problems that can be solved on a given set of hardware and software; it only alleviates the problem and does not solve it Tomasetto et al. (2025), Alora et al. (2023) and Zarrouki et al. (2024). As such, these approaches are suitable for systems where scale is only a marginally limiting factor. As for reducing the solver's effective computational burden, this approach has the most potential as the theoretical scope in which a true solution to the issue would arise, but existing research has not yet achieved this goal. Further, the interpretability of these works and their corresponding general applicability are limited by the often conceptually heavy groundwork. Additionally, such works vary on the means used to quantify the impact of the change, which limits the ability to make statements on their impact on scaling laws (Meng et al., 2024; Yaren and Kizir, 2025; Adabag et al., 2024).

One particularly interesting category of works, which can be considered a part of the scale reduction research, is the use of neural networks (NNs) as the model of the process dynamics. Unlike typical methods of reducing model complexity, NNs operate as a data-driven approach that can estimate high-dimensional phenomena through low-dimensional regression. In fact, NNs can approximate continuous functions on a compact domain to an arbitrary degree of precision, as is shown by the Universal Approximation Theorem (Hornik et al., 1989). As a result, the computational complexity is linear and it has potential to model complex processes with high precision. This is what motivates research into the use of NNs as the model inside MPC, and the results have been promising. Not only do NNs generally operate with a significant reduction in computation time, but they also can sometimes improve the accuracy of the model relative to first-principles-based approaches (Wu et al., 2019; Gordon et al., 2024; Patel et al., 2025; Alsmeier et al., 2024; Ren et al., 2022; Macmurray and Himmelblau, 1995). Sadly, because the NN is nested within the MPC framework, the end result is still an iterative scheme with polynomial-time complexity.

Upon further consideration, one can motivate a broadening of the scope for NN applications in control beyond a process dynamics modeler. In recent years, various studies have demonstrated that NNs are viable as controllers themselves; however, the research is scattered across various fields and processes and thus is lacking in details regarding the scalability and strict constraint satisfaction properties of the network (Lucia and Karg, 2018; Bonzanini et al., 2020; Ahn et al., 2022; Gonzalez et al., 2024). In our prior works, these issues were considered and it was found that, in regard to the scalability concerns, methods such as Imitation Learning (IL) demonstrate limitations at scale that require careful selection of the operating region and informed sampling techniques to reduce the burden of densely sampling high-dimensional states (Khodaverdian et al., 2025b, 2026). Notably, because the high-quality data in imitation learning is derived from an MPC, the computation time problems persist in the form of slow data generation. To combat this, we further researched the use of Reinforcement Learning (RL) methods, which alleviated the data generation speed problems but complicated the design process (Khodaverdian et al., 2025a). As for the strict constraint satisfaction properties of the network, all of these works achieved this through external constraint satisfaction utilizing a fallback controller designed to satisfy the Lyapunov-based stability constraint. While this is sufficient to achieve the desired stability properties, it distances the control behavior from the optimal constrained behavior demonstrated by MPC. Several works have explored modifications of the RL approach through means such as physics-informed RL (Wang and Wu, 2024b), or control Lyapunov-barrier function approaches (Wang and Wu, 2024a), but these approaches only achieve guarantees in the theoretical limit where the optimal policy is extracted. Such approaches that aim to achieve desired behavior through modification of the loss functions alone should rely on assumptions about the network or training loop that are not guaranteed to hold true in practice.

Motivated by these problems, this work explores optimal design techniques for building scalable NNs. In particular, the issue of gradient stability is explored from a first-principles basis to motivate techniques that theoretically aid in maintaining stable training for deep networks. In addition, this work expands on the external constraint enforcement concept by exploring an internal constraint enforcement mechanism that allows for the network to strictly satisfy the Lyapunov-based stability constraint internally. While several techniques exist that enable internal constraint enforcement, it is demonstrated that these come with caveats during training due to their impact on gradients, whereas we propose a lightweight and simple framework that preserves potentially useful gradient signals through approximate means such as straight-through-estimation. This work proposes a novel strict constraint enforcing neural network controller (CENNC) framework called the Alpha-Gate CENNC which utilizes Lyapunov-based stability theory and Straight-Through-Estimation techniques to guarantee that the control action generated by the neural network will satisfy the desired stability constraint. By formulating fallback-based constraint handling as an `if-else` constraint, a Heaviside function formulation can be built with approximated gradients thereby allowing for training on an otherwise non-differentiable function. Unlike existing CENNC methods, the Alpha-Gate formulation yields gradients during backpropagation that can teach the network how to improve the objective through improved constraint satisfaction. The resulting constraint enforcing neural network controller is applied to a benchmark chemical process example and contrasted with an alternative internal constraint method, external constraint method, and IL method to demonstrate the impact on performance, generalizability, and speed.

2. Preliminaries

2.1. Notation

The transpose of a vector x, set of real numbers, set difference, functions, and piecewise-constant functions with period Δ are denoted by $x^\top$, $\mathbb{R}$, $\Omega_1 \backslash \Omega_2$, $f(\cdot)$, and $S(\Delta)$, respectively, where both f and S are arbitrary denotations. The initial timeframe (i.e., where $t = 0$) is denoted by t_0, whereas arbitrary reference timeframes are denoted by t_k.

2.2. Class of systems

This paper considers systems described by nonlinear first-order ordinary differential equations (ODEs) of the form:

$$\dot{x} = F(x,u) = f(x) + g(x)^\top u \tag{1}$$

For these systems, the state vector $x = \left[x_1, x_2, \dots, x_n\right]^\top \in \mathbb{R}^n$ is the vector representation of all relevant state variables for the process. These states are assumed to be measured at fixed sampling intervals of length Δ, as is standard for state feedback control. Similarly, the control input vector $u = \left[u_1, u_2, \dots, u_m\right]^\top \in \mathbb{R}^m$ is the vector representation of all relevant variables that are applied as control actions. Unlike the system states, the control actions are bounded as a means to account for physical limits that real actuators would face. The lower ($u_{i,\min}$) and upper ($u_{i,\max}$) bounds on the control actions form the boundary of the set of valid control actions, defined as:

$$U := \left\{ u \in \mathbb{R}^m \middle| \begin{matrix} u = \left[u_1, u_2, \dots, u_m\right]^\top \\ u_{i,\min} \le u_i \le u_{i,\max} \\ \forall\ i = 1, 2, \dots, m \end{matrix} \right\} \subset \mathbb{R}^m \tag{2}$$

Additionally, we utilize the deviation variable form of the system as a means to render the origin of the open-loop system (i.e., a system of the form shown in Eq. (1) where $u_i = 0\ \forall\ i = 1, 2, \dots, m$) as a steady state, thus ensuring $f(0) = 0$ without loss of generality. In other words, we treat $F(0,0) = 0$. We further assume that systems satisfying Eq. (1) have sufficiently smooth vector functions for $f(\cdot)$ and $g(\cdot)$.

2.3. Stabilizability assumption

The core assumptions that ensure stabilizability are collectively referred to as the stabilizability assumption. The stabilizability assumption consists of two main assumptions. The first is that we assume the existence of a sufficiently smooth explicit feedback control law that renders the origin of the system described by Eq. (1) exponentially stable. This controller is referred to as the reference stabilizing controller, or reference controller, as a shorthand.

$$\Phi : D \to U$$
$$u(x) = \Phi(x) \tag{3}$$

The second is the assumption that there exists a sufficiently smooth Lyapunov function $V(x)$ which, when applied to the closed-loop system utilizing the reference controller for all x bounded by an open neighborhood near the origin, denoted D, satisfies the following inequalities:

$$c_1 |x|^2 \le V(x) \le c_2 |x|^2 \tag{4a}$$

$$\frac{\partial V(x)}{\partial x} F(x, \Phi(x)) \le -c_3 |x|^2 \tag{4b}$$

$$\left| \frac{\partial V(x)}{\partial x} \right| \le c_4 |x| \tag{4c}$$

$$c_i > 0\ \forall\ i \in \{1,2,3,4\} \tag{4d}$$

The remaining parts of the stabilizability assumption are derived from assumptions made thus far. While the following holds true for $\Phi(x) \in U$ due to the sufficiently smooth assumption, the properties can be applied more broadly to any admissible input $u \in U$. The admissible input set U is a closed and bounded rectangular box in $\mathbb{R}^m$, hence compact. When we consider states that lie in a compact subset of D, such as the level set $\Omega_\rho := \{x \mid V(x) \le \rho\} \subset D$, then both $x \in \Omega_\rho$ and $u \in U$ are within compact sets, and thus $F(x,u)$ is bounded for all $x \in \Omega_\rho$ and $u \in U$. Because $f(x)$ and $g(x)$ are sufficiently smooth, then $F(x,u)$ is also sufficiently smooth for each fixed $u \in U$. Finally, the sufficiently smooth properties imply Lipschitz continuity for $F(x,u)$, $V(x)$, $\frac{\partial V(x)}{\partial x}$, and thus $\frac{\partial V(x)}{\partial x} F(x,u)$. Thus, the stabilizability assumption implies the existence of positive constants M_F, L_x, L'_x that ensure, for all $x, x' \in \Omega_\rho$ and $u \in U$, that the following inequalities are satisfied:

$$\left| F(x',u) - F(x,u) \right| \le L_x |x - x'| \tag{5a}$$

$$|F(x,u)| \le M_F \tag{5b}$$

$$\left| \frac{\partial V(x)}{\partial x} F(x,u) - \frac{\partial V(x')}{\partial x} F(x',u) \right| \le L'_x |x - x'| \tag{5c}$$

2.4. Lyapunov-based model predictive control

The stabilizability assumption can be applied to MPC to yield the Lyapunov-based MPC (LMPC) that solves for optimal control while ensuring closed-loop stability within Ω_ρ (Mhaskar et al., 2006):

$$\mathcal{J} = \min_{u \in S(\Delta)} \int_{t_k}^{t_k + N\Delta} L(\tilde{x}(t), u(t))\ \mathrm{d}t \tag{6a}$$

$$\text{s.t.} \quad \dot{\tilde{x}}(t) = F(\tilde{x}(t), u(t)) \tag{6b}$$

$$u \in U,\ \forall\ t \in [t_k, t_k + N\Delta) \tag{6c}$$

$$\tilde{x}(t_k) = x(t_k) \tag{6d}$$

$$\dot{V}(\tilde{x}(t_k), u(t_k)) \le \dot{V}(\tilde{x}(t_k), \Phi(\tilde{x}(t_k))) \tag{6e}$$

In this formulation, the optimization takes place over a horizon of length $N\Delta$, with Δ denoting the controller's sampling period and N the horizon's sampling step count. This formulation uses sample-and-hold control, as continuous-time control is infeasible for real-world processes. For simplicity, the sampling interval for both the controllers and state measurements is treated as equivalent. Eq. (6a) denotes an arbitrary cost as a function of the control actions and estimated states over the horizon. Eq. (6b) represents the process dynamics, which are used for numerical integration during optimization to predict how the states of the closed-loop system evolve over the horizon. Eq. (6d) enforces an initialization step where the sensor readings are used as a ground truth initial state, and Eq. (6c) enforces the control bounds. Eq. (6e) denotes the stability constraint; the implementation of the stabilizability assumption. This constraint ensures that the closed-loop system is at least as stabilizing as it is under the reference controller. An alternative form is provided:

$$\dot{V}(\tilde{x}(t_k), u(t_k)) \le -\alpha V(\tilde{x}(t_k)) \tag{7}$$

This formulation uses the properties of the Lyapunov function from Section 2.3 as opposed to using the reference controller. Using a positive constant α to control the strength of this constraint, this formulation allows for stability guarantees without needing to directly apply the reference controller.

Remark 1. Eq. (6e) is only applied at t_k because this formulation is a receding horizon LMPC, where only the first control input from the solution is applied. After applying the first solution for one sampling interval, the LMPC problem is re-solved. This approach relaxes the constraints of the optimization problem, allowing for faster solutions, but comes at the cost of marginally reduced accuracy of the cost-optimal trajectory.

3. Neural network-based control

3.1. Imitation learning-based control

Imitation learning is a category of training methods for neural networks with the goal of yielding outputs that match the demonstrated behavior of some existing expert policy. Typically, this is done by collecting data for various states under the control of the expert. By doing this, a dataset of state–action pairs can be built over time. Despite the simplistic nature of this category of machine learning, there are several existing implementations with varying strengths and weaknesses.

3.1.1. Behavioral Cloning (BC)

The simplest form of IL is referred to as BC. In BC, supervised learning is used to train the neural network. Supervised learning is a training method where the goal is to teach the network to function as a mapping from the input data to the desired output data using coupled input–output pairs. In the context of control, the state–action pairs suffice as input–output pairs and thus the model is trained with any technique that would drive the model towards mimicking the control action distribution for the provided states. For deterministic continuous-time systems, a simple implementation of this would be the mean squared error (MSE) or Euclidean (L_2) norm between the neural network policy and the expert policy, defined as follows:

$$\mathrm{MSE}\,(x) = \frac{1}{N}\sum_{i=1}^{N}\left(x_i - x_i^{pred}\right)^2 \tag{8}$$

$$\mathrm{L}_2\,(x) = \sqrt{\sum_{i=1}^{N}\left(x_i - x_i^{pred}\right)^2} \tag{9}$$

Here, N denotes the number of samples in the batch that the loss will be averaged over. While BC is a simple and powerful tool for producing a seemingly good controller, in practice, it has a critical weakness in the form of distribution shift. Specifically, BC suffers from covariate shift caused by the mismatch between its training dataset's distribution and the real-world distribution of inputs. In order for the training distribution to perfectly match the true distribution, all possible states must be sampled for the dataset. Real systems operate with continuous state and action spaces, and as such, it is impossible to sample all state–action pairs in a finite dataset. As a result, the training dataset will inevitably have a different input/state distribution relative to reality. While this problem diminishes as the sampled initial states approach a densely sampled uniform distribution, the unavoidable problem is that the performance can degrade arbitrarily in out-of-distribution (OOD) states. As a result, even with small but non-zero average imitation error (measured under the expert's state distribution, denoted ϵ), the expected difference between the performance of the expert and the neural network is tightly upper-bounded by $T^2\epsilon$, a term that scales quadratically with the length of the closed-loop trajectory (Ross and Bagnell, 2010). Such behavior is of great concern, as it not only poses a limit to the generalizability of Behavioral Cloning methods when sampled trajectories are sparsely generated, but it also limits the generalizability under dense sampling if the trajectory length is exceedingly long.

3.1.2. Dataset Aggregation (DAgger)

To counteract the limitations of traditional Behavioral Cloning techniques, various methods have been devised with the goal of approaching near-linear scaling of the expected performance difference with trajectory length. A notable algorithm that achieves this is DAgger (Ross et al., 2011). DAgger solves the issue by involving the neural network in the generation of the dataset. Instead of using purely expert rollouts, DAgger generates a modified policy by mixing the current policy with the expert policy via $\pi_i = \beta_i\pi^* + \left(1-\beta_i\right)\hat{\pi}_i$ with a weight satisfying $\lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N}\beta_i = 0$ where N denotes the number of iterations of the DAgger algorithm. This modified policy (π_i) is used in place of the expert (π^*) in deciding the exploration of states, but the expert policy action is still stored for each visited state. After adding these state–action pairs to the dataset and retraining the policy, the newly updated policy can iteratively be refined over N loops. Of the N policies this yields, only the best is kept. As a result of this process, the scaling of the expected trajectory cost difference is now relative to $uT\epsilon$ instead of $T^2\epsilon$ where u denotes the upper bound on the extra cumulative cost over the remaining steps if the policy deviates from the expert. While this alleviates the quadratic scaling problem, it is still a method that relies on high volume querying of the expert which, in the case of Lyapunov-based model predictive control, is itself a limiting factor on the scalability of the neural-network based control design.

3.1.3. Fixed expert dataset approaches

Some approaches to IL only require an initial set of expert data and will learn a policy through means that do not require additional expert rollouts. Inverse Reinforcement Learning (IRL) is a nested training method which trains the policy alongside a reward function. A reward function is trained by evaluating the policy's state–action visitation frequency relative to the experts distribution. Using this reward function as a basis for a reinforcement learning loop yields a policy that gradually approaches the expert.

Generative Adversarial Imitation Learning (GAIL) is a sequential training loop that trains a discriminator alongside the policy. The discriminator is trained with the goal of properly differentiating between the expert's trajectory and the policy's trajectory. The policy is in turn trained with the goal of confusing the discriminator. By training both sequentially, the idea is that the end result will yield a policy that is indistinguishable from the expert.

3.1.4. Limitations of imitation learning

Due to its simplistic design, BC functions as a benchmark for the weaknesses inherent in imitation learning methods. Conceptually, the goal of functioning as an imitation leads to poor generalization. As discussed previously, there is the compounding risk associated with drifting into OOD states that leads to difficulties in stable long trajectory rollouts without simultaneously teaching the model how to escape these regions by itself. DAgger was designed to mitigate this problem, and both IRL and GAIL mitigate it through their explorative training loops, but all 3 methods only alleviate this problem for states near the expert distribution; significantly OOD states will still suffer from unpredictable behavior for all methods (Lee et al., 2025; Zhou and Li, 2024; Han et al., 2026). IRL in particular suffers from the risk of suboptimal policies, where the learned reward function takes a form that allows for similar state distributions to form without the resulting trajectories achieving the desired task in its entirety. In other words, local minima are achieved instead of roughly global minima as desired (Ablett, 2025). This highlights a fundamental flaw in imitation learning; one can imitate without understanding.

Because these methods rely on the existence of some expert demonstrations, either as raw data to process or through indirect data via observation, they are hindered in their ability to scale. The issues mentioned previously emphasize the importance of densely sampling the state distributions that are encountered in reality, but such dense sampling poses issues as dimensionality scales or as the expert's computational complexity scales, as both result in increasingly high time requirements for sampling sufficient amounts of data. Even in the abundance of data, the end result will only ever match the performance of the expert.

The goal of control systems is to optimally execute the desired task, and the expert may not necessarily satisfy this requirement for real systems as even optimal control methods are limited both in their ability to consistently reach global optima of nonlinear problems and in their inherent inaccuracy in using mathematical models to approximate physical systems. To this extent, it is desirable for neural networks to

not simply mimic the expert, but to understand the control problem itself.

Because IL methods struggle in regards to understanding what they aim to mimic, especially when distribution shift is present, there is a heightened risk of causal confusion (de Haan et al., 2019). Behavioral Cloning and DAgger are the most susceptible to this issue, as their training method is purely regression-based. IRL and GAIL are training loops that involve exploration of the environment via the policy, and as such, they are able to implicitly understand the causal effects of the environment, mitigating the causal confusion risk; however, this is limited to the scope of mimicking the expert. In other words, while these methods may truly understand the true causal relationship, they do not necessarily understand the desired task as is most clearly seen by IRL's tendency to "reward-hack" (train such that the loss is small but the end result is not as desired). Such problems are only made worse in observation-based imitation learning when unobserved inputs are made available to the expert but not to the policy, which reintroduces the causal confusion problem to these methods (Ruan et al., 2023). Similarly, observed but irrelevant signals give the GAIL discriminator avenues to derail the training objective from reaching the expert behavior (Zolna et al., 2020). The broader trend resulting from these limitations can be seen in existing research focused on continual learning methods for large language models where it is seen that supervised-fine-tuning, often implemented using behavioral cloning, results in models that memorize rather than generalize and fail to retain information upon further fine-tuning.

We can see from the strengths of IRL and GAIL over DAgger and BC that the ability to explore the environment and learn from it results in models that are more robust to minor deviations. The results of LLM-scale research corroborate this trend, and further show how methods such as on-policy reinforcement learning are capable of several desirable strengths including robustness against OOD performance collapse, generalizability of learned information, resistance to catastrophic forgetting, and the ability to improve the baseline model's abilities (Wolfe, 2026). As such, reinforcement learning is uniquely positioned as an appealing training architecture for scalable control-oriented neural networks.

3.2. Reinforcement Learning (RL)-based control

In the context of process control, RL is a class of training methods for controllers that teaches through experience. More generally, an Agent is used to interact with an Environment to yield a Reward, some quantifiable outcome that is desired. The goal of RL is to train the agent to maximize the expected discounted cumulative reward based on its interactions with the environment. The interactions themselves are dictated by the agent's Policy, which is a functional mapping from observed states (s) to actions (a). Applying actions to the environment causes a transition in the state, which yields an observed future state (s') and a corresponding scalar reward (r). This results in a 4-element experience tuple (s, a, r, s'). Using this data, we can quantify the quality of trajectories of arbitrary length through the discounted cumulative reward:

$$G(S_t, A_t) = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \qquad \gamma \in [0, 1) \tag{10}$$

In this formulation, starting at some reference time t, the reward following k+1th transition is defined as $R_{t+k+1} := r(S_{t+k}, A_{t+k})$ while γ denotes the discount factor. Although the systems considered in this work are continuous-time time-invariant systems, it is not possible to control a real system in a continuous-time fashion. Instead, the control is done in a sample-and-hold fashion, where the control action is held constant for a fixed duration before being recalculated. As a result, it is beneficial to reformulate the control problem as a discretized continuous-time process. In fact, because of this discretized formulation, traditional RL methods operate under a tabular nature where value functions, defined based on the discounted cumulative reward expression, are built using look-up tables containing all (s, a) pairs. One such value function is the state-value function which evaluates a policy's discounted cumulative reward across its resulting closed-loop trajectory. If we consider a form of the reward function that is for the case of a fixed policy, i.e., $R^{\pi}_{t+k+1} := r\left(S_{t+k}, \pi\left(S_{t+k}\right)\right)$, then the state-value function can be defined as:

$$V^{\pi}(s) = G\left(S_t, \pi\left(S_t\right)\right) = \sum_{k=0}^{\infty} \gamma^k R^{\pi}_{t+k+1} \tag{11}$$

It is often simpler to work with a recursive form of this equation, referred to as the Bellman Equation for the state-value function. The recursive value term is often referred to as a bootstrapped term:

$$V^{\pi}(s) = r\left(s, \pi\left(s\right)\right) + \gamma V^{\pi}\left(s'\right) \tag{12}$$

For an optimal policy, this equation is referred to as the Bellman optimality equation, defined as:

$$V^{*}(s) = \max_{a}\left[r\left(s, a\right) + \gamma V^{*}\left(s'\right)\right] \tag{13}$$

which allows for greedy extraction of the optimal policy via:

$$\pi(s) = \arg\max_{a}\left[r\left(s, a\right) + \gamma V^{*}\left(s'\right)\right] \tag{14}$$

This recursive form of the state-value function is especially convenient in the sense that it can be used as an iterative numerical method for approximating the state-value function in the case where the state-value function is not known. In particular, value iteration uses Eq. (13) in an iterative scheme defined by

$$V_{i+1}(s) = \max_{a}\left[r\left(s, a\right) + \gamma V_i\left(s'\right)\right] \tag{15}$$

where some current guess of the state-value function denoted by V_i is iteratively updated over all states and the corresponding best-valued action, which requires all actions to be considered. This scheme monotonically converges to the optimal state-value function, allowing for the use of Eq. (14) to extract the corresponding optimal policy. There is an alternative method called policy iteration which utilizes Eq. (12) to iteratively update the state-value function for the current policy until convergence, followed by Eq. (14) to extract the corresponding policy update. This entire process is done iteratively, until both the state-value function and the policy converge to their optimal forms, which is also monotonic.

Both methods suffer from three critical limitations. First, all states need to be considered in order for the monotonic property to hold. As a tabular method, the entire look-up table for the value function needs to be updated for each iteration, which is a resource intensive task. Second, both methods require the use of the max or arg max operation which both require the processing of all valid actions due to the discrete formulation. Third, both methods are model-informed, as the calculation of the transition to the future state s' and the corresponding reward r require full knowledge of the process. The need to process all states and actions is not possible for continuous state–action spaces that are observed in reality, making these methods ill-suited for process control cases. The need for a perfect process model is also unrealistic for complex process control cases, and as such it is often desirable to either utilize data-driven models or to use model-free approaches where process data can be used directly without a need to model the transitions.

Remark 2. While in Eq. (10) it is often the case that $\gamma \in [0, 1)$ because of the favorable properties this yields for G such as remaining finite over infinite horizons, it is not always true. In fact, there are cases such as known finite horizon problems where $\gamma = 1$ can be favorable if the intent is to not over-favor early actions.

3.2.1. Temporal-Difference (TD) Learning

Beyond the traditional Dynamic Programming or Monte-Carlo methods, there exists another subcategory of training methods in RL called TD-Learning. TD-Learning functions like a hybrid of the two, where trajectories are used to build the target, much like Monte-Carlo methods, but instead of requiring full trajectories, only a desired number of steps are needed with the remaining terms being bootstrapped, as is done in DP methods. While TD methods are not inherently model-free, they are well suited for model-free applications with the caveat that the state-value function is no longer sufficient for greedy policy improvement because the model is no longer assumed to be known and thus evaluation of all actions with the state-value function becomes impossible due to the lack of knowledge on transitions. To avoid this, the deterministic Bellman relation form of the action-value function is defined as follows:

$$Q^{\pi}(s,a) = r(s,a) + \gamma V^{\pi}\left(s'\right) \tag{16}$$

In other words, the action-value function first considers the value of some action (a) at the current state (s) followed by the value of all future actions executed via the current policy. Policy improvement is now possible through greedy extraction from the action-value function.

Further, training via RL can be done with either on-policy or off-policy methods. These training types are defined through their use of two categories of policies. The target policy is the policy being evaluated or improved, whereas the behavioral policy is the policy whose actions are used for data/experience generation. The distinction between off-policy and on-policy is that on-policy is the case where the target and behavioral policy are the same, whereas off-policy allows for different policies. SARSA is one such algorithm that is on-policy, whereas Q-learning is an algorithm that is off-policy as shown in Eqs. (17) and (18), respectively:

$$Q_{i+1}(s,a) = Q_i(s,a) + \alpha\left[r + \gamma Q_i\left(s',a'\right) - Q_i(s,a)\right] \tag{17}$$

$$Q_{i+1}(s,a) = Q_i(s,a) + \alpha\left[r + \gamma \max_{a'} Q_i\left(s',a'\right) - Q_i(s,a)\right] \tag{18}$$

Although the two seem similar, the distinction comes from the greediness of the target policy. SARSA stands for State-Action-Reward-State-Action, which represents the structure of its experience tuple where the future action is also sampled, allowing for the bootstrapped term to use the behavioral policy's action, thereby making the target policy the behavioral policy compared to Q-learning's purely-greedy target-policy. This distinction results in significant impacts on the behavior of their resulting policies. Q-learning's max operation directs the performance improvements by exploiting the greedy policy for the current value function estimate, whereas on-policy methods like SARSA are guided by an exploratory behavioral policy. In other words, the target policy for Q-learning being different from the behavioral policy allows it to exploit good-performing policies regardless of what is done for exploration. As a result, SARSA often results in conservative policies due to the need to optimize around exploration behavior such as ϵ-greediness. Consequently, Q-learning risks overestimation of its decision quality due to the maximization bias caused by the max operator on noisy value function estimates, which can hinder training, especially in environments where high quality comes with a risk of consequences, such as the Cliff Walk example shown in Fig. 1.

Cliff Walk is a toy problem where a 2D grid contains a cliff region in the bottom row, excluding the corner grids. The goal of the policy is to move from a fixed starting grid in the bottom left corner to the goal in the bottom right corner in single grid steps. One formulation of the problem assigns a reward of -1 to all state transitions besides the stepping into the cliff, which is assigned a reward of -100 and causes the state to return to the starting point (Sutton and Barto, 2020). For this formulation with ϵ-greedy exploration with $\epsilon = 0.1$, SARSA eventually converges to a policy that yields roughly -25 reward on average, while Q-learning converged to -50. One might conclude from this that the SARSA policy is more optimal than the Q-learning policy, but in reality, SARSA learned the conservative path instead of the optimal path that Q-learning did. The reason the episodic rewards are better in SARSA is because exploration is done using the ϵ-greedy policy, and so the occasional randomness in the exploration would cause a policy following the optimal path to wander off the cliff. This example demonstrates that, ultimately, SARSA learns a policy that is optimal during exploration whereas Q-learning learns a policy that is truly optimal. Both methods still suffer from the computational infeasibility of greedy policy extraction in continuous state–action spaces which motivates the use of gradient-based methods.

3.2.2. Twin delayed deep deterministic policy gradient (TD3)

Unlike value-based methods, policy-gradient methods allow the policy to be parameterized. These parameters are then modified through gradient methods to achieve the desired behavior. In other words, the goal of value-based methods is to optimize the value function and then greedily (via arg max) derive the policy, whereas the goal of policy-gradient methods is to optimize the policy directly based on the value function in an iterative process. In its simplest form, there is Vanilla Policy Gradient (VPG). VPG is typically formulated for stochastic policies where the goal is to maximize the expected discounted return under the current policy and current value-function estimate. This can be formulated as the product of the trajectories discounted cumulative reward weighed by the probability of the trajectory under the current policy summed over all possible trajectories as follows:

$$J(\theta) = \sum_{\tau}\left(\prod_{t}\pi_{\theta}\left(a_t^{\tau} \mid s_t^{\tau}\right)\right)V_{\tau}^{\pi} \tag{19}$$

Here, the sum over all unique trajectories τ is represented through the true state-value function for the given trajectory V_{τ}^{π} and with trajectory probability given by the product of the probability that the stochastic policy whose learnable parameters are denoted by θ yields the actions a^{τ} that it did at every state s^{τ} along the trajectory with time-index t. Through the fact that:

$$\nabla_{\theta}\log f(\theta) = \frac{\nabla_{\theta} f(\theta)}{f(\theta)} \rightarrow \nabla_{\theta} f(\theta) = f(\theta)\nabla_{\theta}\log f(\theta) \tag{20}$$

and via the chain rule:

$$\nabla_{\theta}\prod_{t} g_t = \left(\prod_{t} g_t\right)\sum_{t}\frac{\nabla_{\theta} g_t}{g_t} \tag{21}$$

the corresponding gradient of the objective is:

$$\nabla_{\theta} J(\theta) = \sum_{\tau}\left(\nabla_{\theta}\prod_{t}\pi_{\theta}\left(a_t^{\tau} \mid s_t^{\tau}\right)\right)V_{\tau}^{\pi} \tag{22}$$

$$= \sum_{\tau}\left(\left(\prod_{t}\pi_{\theta}\left(a_t^{\tau} \mid s_t^{\tau}\right)\right)\sum_{t}\frac{\nabla_{\theta}\pi_{\theta}\left(a_t^{\tau} \mid s_t^{\tau}\right)}{\pi_{\theta}\left(a_t^{\tau} \mid s_t^{\tau}\right)}\right)V_{\tau}^{\pi} \tag{23}$$

$$= \sum_{\tau}\left(\left(\prod_{t}\pi_{\theta}\left(a_t^{\tau} \mid s_t^{\tau}\right)\right)\sum_{t}\nabla_{\theta}\log\pi_{\theta}\left(a_t^{\tau} \mid s_t^{\tau}\right)\right)V_{\tau}^{\pi} \tag{24}$$

This formulation is the baseline for REINFORCE, where the sampled return is involved to make an unbiased estimate of the policy gradient. Although unbiased, this estimator can have high variance because the return may not isolate whether or not a particular action was better or worse than expected for its state. This motivates alternatives that improve variance without impacting bias. For a given trajectory, we can denote the current value function as $\hat{V}_i^{\pi}(s)$ and approximate this gradient for use in gradient ascent. Since an estimate is used, it is often advantageous to use the advantage function form, where instead of increasing the probability of actions that give high return trajectories, the probability of actions that give better than expected returns are increased. Notably, because the state-value function is only a function of the state, and these methods are on-policy, the bias is unchanged in this advantage formulation based on the estimated function. Thus, while the exact formulation of the advantage estimate can vary based

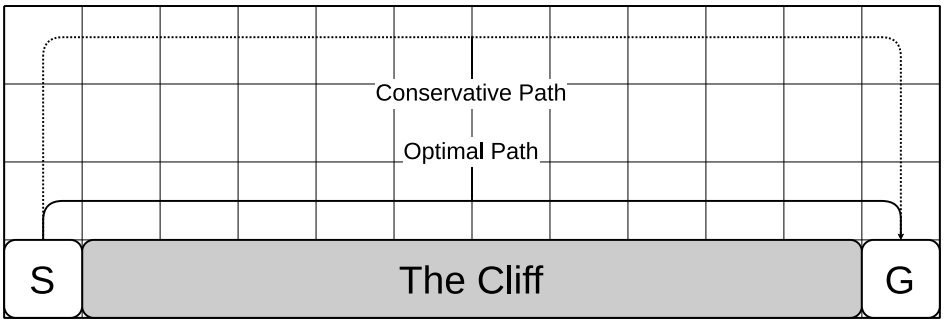

Fig. 1. Cliff Walk toy problem. Entering 'The Cliff' will reset the state to the start, denoted by 'S'. Episodes end upon entering 'G', the goal.

on the design, VPG tends to use the following mini-batch based loss for gradient descent:

$$\mathcal{L}_{VPG} = -\frac{1}{N}\sum_{\tau=1}^{N}\sum_{t}\log\pi_\theta\left(a_t^\tau \mid s_t^\tau\right)\hat{A}_t^\pi \tag{25}$$

where $\hat{A}_i^\pi$ is the advantage function defined using $\hat{V}_i^\pi(s)$ and N is the number of samples in the mini-batch.

In practice, VPG benefits from a separate training loop to improve on these estimate value functions, which motivates Actor-Critic methods where the policy is modeled by the Actor which is trained with respect to the Critic, which models the value function and itself is trained based on the Actor. This design aims to improve the policy and value function estimate in a self-improving loop. Further, the gradient based iterative parameter optimization directly motivates the use of NNs as the estimate for the policy or value functions due to their inherent use of these gradient methods during training and their broad modeling abilities. Finally, it can be seen that the existence of stochasticity in the policy complicates the relationship between the objective, the loss function, and their corresponding gradients, hence the earlier emphasis on focusing on deterministic systems. To build on this further, we consider deterministic frameworks that utilize gradient-based methods, involve NNs, and are off-policy to make reuse of data (which can be scarce in process control contexts) viable during training. This leads to one of the original gradient based Actor-Critic methods, Deep Deterministic Policy Gradient (DDPG). DDPG avoids the greedy policy of Q-learning by instead training a target policy network to approximately maximize the action-value function through gradient ascent (Lillicrap et al., 2016). As a result, the training of the action-value function approximation can be achieved through a neural network referred to as the Critic using the following loss function:

$$\mathcal{L}_{\text{Critic}} = \text{MSE}\left[Q_\phi(s,a), \left(r + \gamma\left(1-d\right)Q_\phi(s', \pi_\theta\left(s'\right))\right)\right] \tag{26}$$

As was true for VPG, θ denotes the learnable parameters of the policy and ϕ denotes the learnable parameters of the Critic network. DDPG improves on this further, by taking into account the instability of training a neural network with respect to a target using its own weights. The solution is to use a second neural network as a dedicated target network which gets updated via Polyak averaging, as follows:

$$\phi' = \tau\phi + (1-\tau)\phi' \tag{27}$$

$$\theta' = \tau\theta + (1-\tau)\theta' \tag{28}$$

Additionally, as a 'Deep' approach, the neural networks are used to represent the Actor and Critic which aim to approximate the policy and value function, respectively. To train such networks, a dataset called the replay buffer is built using the experience replays defined earlier:

$$D = \left\{\left(s_k, a_k, r_k, s_k'\right)\right\}_{k=1}^{N} \tag{29}$$

where k denotes the index of the sample within the replay buffer and N denotes the number of samples currently stored in the replay buffer. Notably, in the case of off-policy methods like TD3 and DDPG, the actions stored in this buffer come from various policies such as from the Actor as it is being trained. Because there is no ϵ-greedy policy available during exploration, the exploration phase of DDPG utilizes the Actor with additive noise. Exploration itself is a process that can be sped up using a simulated model for bulk data generation, but is best done when applied to a physical system to capture real-world dynamics and noise as is modeled in Fig. 2.

TD3 takes additional measures to stabilize training beyond what is done in DDPG. It adds clipped noise to the target policy during the construction of the Critic target for the loss calculation of the Critic, adds a second Critic with its own target network, and adds a delay to the updates for the target networks and Actor (Fujimoto et al., 2018). The goal of the second Critic is to counteract overestimation of the action-value function. Because the Actor is trained to maximize the Critic, there remains an analogous effect to the maximization bias present in Q-learning because the policy can exploit inaccuracies in the learned action-value function by prioritizing actions in regions where the function is overestimated. Unlike tabular Q-learning, which converges to the optimal action-value function in the limit under standard assumptions, deep actor-critic based approaches operate in continuous spaces with function approximations, and so they do not inherit these convergence guarantees. Instead, the overestimation in the Critic causes the Actor to yield more actions in the overestimated region, and because the Actor is used for exploration, the replay buffer can begin to over-sample these actions/regions, which the Critic will then train off of and reinforce. The result is a feedback loop of continued exploitation that makes training brittle and risks policy collapse or divergence. Similar to how Dual Q-learning handles the maximization bias, by using the minimum of two estimates of the action-value function, TD3 lessens the possibility of overestimation as it is unlikely for the Actor to exploit both networks simultaneously. The remaining modifications follow similar logic. The policy noise effectively smoothens the Critic, making it less susceptible to jagged peaks, and the delay further lessens the moving target problem. Pseudocode for the TD3 algorithm is provided in Algorithm 1.

3.3. Scalable, stable, and safe neural network design

While RL methods such as TD3 are good options for model-free off-policy reinforcement learning, which enables a broad range of designs, NN training on such frameworks has been shown to be brittle as was seen in prior works (Khodaverdian et al., 2025a). Beyond the careful hyperparameter consideration that is needed for such training methods, the NN itself must be designed in such a way that does not excessively compound on this brittleness. To understand the root of this instability and to design systems that are both scalable to deeper and wider layers, one must take a first-principles approach to training.

3.3.1. Gradient descent

Despite the wide variety of frameworks and training methods available, a fundamental component of NN training that is consistent in most

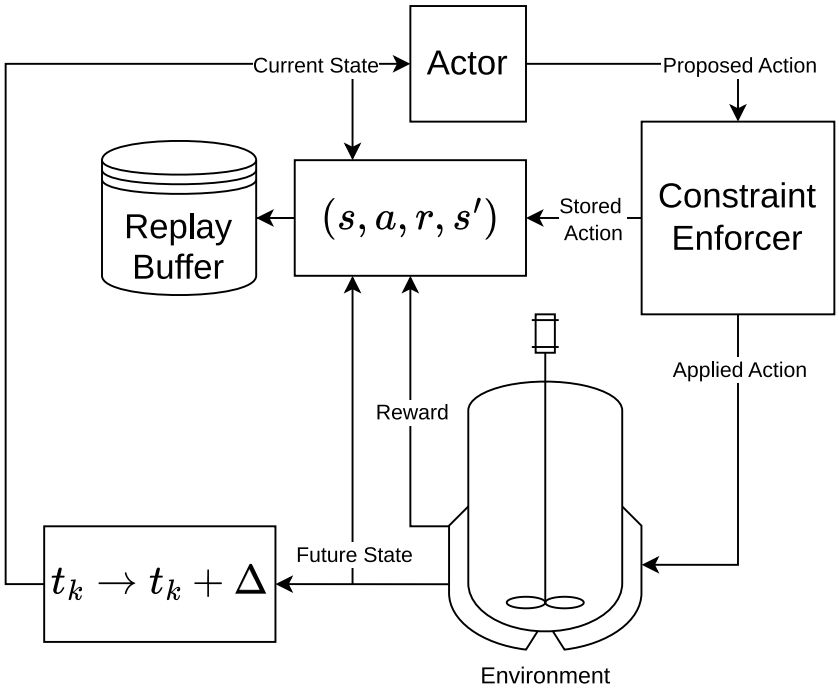

Fig. 2. Reinforcement Learning exploration phase diagram under action constrained closed-loop rollout.

works is the use of gradient descent. In principle, gradient descent is a method of calculating gradients sequentially by utilizing the chain rule. As a result, weights and bias terms in early layers of a NN can be thought of through a series of multiplications of gradients from deeper nodes in the network. While the process is convenient and allows for significant computational improvements, it suffers from a fundamental problem; vanishing/exploding gradients. As more layers are introduced, more multiplication must occur to calculate the gradients for shallower weights. If enough terms have excessively small or high values, the result is that these shallow gradients can collapse towards zero or diverge.

The issue of exploding gradients is of lesser concern as most commonly used activation functions have gradients that are bounded by 1. Commonly used activation functions include `ReLU`, `Sigmoid`, and `Tanh`, which are all defined below along with their corresponding derivatives:

$$\texttt{ReLU}(x) = \max(0, x) \qquad \frac{\partial \texttt{ReLU}}{\partial x} = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{30}$$

$$\texttt{Sigmoid}(x) = \frac{1}{1 + \exp(-x)} \qquad \frac{\partial \texttt{Sigmoid}}{\partial x} = \frac{\exp(x)}{(1 + \exp(x))^2} \tag{31}$$

$$\texttt{Tanh}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \qquad \frac{\partial \texttt{Tanh}}{\partial x} = \operatorname{sech}^2(x) \tag{32}$$

These derivatives provide insight into the issues that networks utilizing them may run into if excessively deep. `ReLU`'s issues are best known as it suffers from the *Dying ReLU Problem* where, because the gradient is zero for negative inputs, if any deep layer using `ReLU` has a negative input, the impacted neurons and all terms that must flow through this neuron to get gradient updates will not get updated for that sample. If, at any point in training, the weights adjust such that these neurons get negative inputs consistently, the result is a cascade of dying neurons. Similarly, `Sigmoid` suffers from having a maximum gradient of 0.25, which causes the vanishing gradient problem if enough layers use this activation in a deep network. `Tanh` has a maximum gradient of 1 but is quickly decaying for larger non-zero points (even more so than `Sigmoid`), which leads to a risk of limited expressiveness of the network and the persistence of vanishing gradient issues in deep nets. Despite these tradeoffs, `ReLU` remains a highly utilized activation function due to its gradient of 1 for positive values, preventing contributions to exploding/vanishing gradients, and it causes sparsity within the network through effective feature selection by zeroing out negative inputs which leads to good expressiveness abilities. As a result of these benefits, a substantial amount of research has been done on designing activation functions that maintain the strengths of `ReLU` but combat the weaknesses. To this extent, there are many modern alternatives to `ReLU` with comparable performance, such as `GELU`, `Mish`, `Swish`, and `Leaky ReLU`.

In order to design a framework that scales well for deep networks, it becomes necessary to keep gradients near 1 to prevent gradient decay/explosion. Activation function choice impacts this during training, but the initialization of the weights and biases contribute to early training stability based on the same principle. One cannot arbitrarily define the initial weights or biases, otherwise initial gradients can be unstable. A solution to this is to utilize orthogonal initialization of the weight matrices. The motivation for this form of initialization is due to the beneficial properties that arise from orthogonal matrices. Notably, these matrices are both norm preserving and dot-product preserving. As a consequence, signals passing through these layers will not be perturbed in magnitude or angle and will instead be rotated. This form of transformation is a neutral operation that is beneficial for initial training steps as it is inherently non-interactive in ways that could lead to exploding or decaying signals or gradients. While some of this neutrality is inevitably lost for layers where the input and output dimensions are different, it is a good design choice nevertheless. One caveat that complicates the use of this technique is nonlinearities and desired behavior. Activation functions such as `ReLU` effectively zero out half of the values, and so their gain is usually set to $\sqrt{2}$ instead of 1. Early training can be chaotic, so often the final layer in the Actor is set to have small gain of roughly 0.01 so that actions are not excessively aggressive early on. More generally, the method would require additional consideration of inputs and outputs to these initialized layers so that properties such as the second moment are preserved. Some well known baselines such as the `ReLU` case of $\sqrt{2}$ require assumptions on the structure of the pre-activation distribution, such as zero centered-ness, which needs to be considered during the design process.

Algorithm 1: TD3

Initialize *critic networks* Q_{θ_1}, Q_{θ_2}, *and actor network* π_ϕ *with random parameters* θ_1, θ_2, ϕ;
Initialize *target networks* $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$;
Initialize *replay buffer* $\mathcal{D}$;
while *not converged* **do**
  Observe the environment's current state s;
  Select action with exploration noise $a \leftarrow \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0,1)$;
  Clip to obtain $\tilde{a}$ within lower and upper bounds: $a_{Lower} \le \tilde{a} \le a_{Upper}$;
  Apply $\tilde{a}$ and observe reward r, new state s', and done signal d;
  Store transition tuple $(s, \tilde{a}, r, s', d)$ in $\mathcal{D}$;
  if *it is time to update* **then**
    Sample mini-batch of N transitions $\left(s_j, \tilde{a}_j, r_j, s'_j, d_j\right)_{j=1}^{N} \sim \mathcal{D}$;
    $a'_j \leftarrow \pi_{\phi'}\left(s'_j\right) + \epsilon_j$,
    $\epsilon_j \sim \text{clip}\left(\mathcal{N}\left(0, \texttt{policy\_noise}\right), -\texttt{noise\_clip}, \texttt{noise\_clip}\right)$;
    Clip to obtain $\tilde{a}'_j$ within lower and upper bounds: $a_{Lower} \le \tilde{a}'_j \le a_{Upper}$;
    $y_j \leftarrow r_j + \gamma\left(1 - d_j\right) \min\left(Q_{\theta'_1}\left(s'_j, \tilde{a}'_j\right), Q_{\theta'_2}\left(s'_j, \tilde{a}'_j\right)\right)$;
    Compute critic losses: $\mathcal{L}_1 = \text{MSE}\left(Q_{\theta_1}\left(s_j, \tilde{a}_j\right), y_j\right)$,
    $\mathcal{L}_2 = \text{MSE}\left(Q_{\theta_2}\left(s_j, \tilde{a}_j\right), y_j\right)$;
    $\mathcal{L}_{critic} \leftarrow \mathcal{L}_1 + \mathcal{L}_2$;
    Zero gradients for critics;
    Backward pass on $\mathcal{L}_{critic}$;
    Gradient descent step for θ_1 and θ_2;
    if *it is time to update the delayed networks* **then**
      Compute actor loss: $\mathcal{L}_{actor} \leftarrow -\text{MEAN}\left(Q_{\theta_1}\left(s_j, \pi_\phi\left(s_j\right)\right)\right)$;
      Zero gradients for actor;
      Backward pass on $\mathcal{L}_{actor}$;
      Gradient descent step for ϕ;
      Soft update targets: $\theta'_k \leftarrow \tau\theta_k + (1-\tau)\theta'_k$ for $k = 1, 2$;
      $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$;
  if s' *is in the terminal region* **then**
    Reset environment;
    Select a new initial state s;

3.3.2. Beyond weights and biases

While modifications of the activation function allow for design levers through their impact on the training of the weights and biases via gradients and the activation/expressiveness of these weights and biases through their non-linearity, these approaches can be enhanced further through additional structures in the network. Recall the issue of vanishing gradients. A common workaround is to use residual layers. Residual layers operate by adding the input to the post-activation output to yield the final output. Consider a residual block:

$$y = x + F(x, W) \tag{33}$$

where $F(x, W)$ is the layers with weights W that the input x passes through. The gradient term from this block is:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}\left(1 + \frac{\partial F}{\partial x}\right) \tag{34}$$

While the mechanism is simple, the result is that weights updated in layers shallower than the input will get an additional +1 term added to partial gradient of the deeper terms, effectively resetting the multiplier and counteracting the decay. A consequence of this is that the model may learn to take advantage of these residual pathways. In the extreme, such behavior results in a network that functions like an ensemble of shallower networks.

More generally, data that gets passed between layers of a neural network can suffer from a multitude of issues, especially in deep networks with more complex data handling. Exploding and vanishing gradients are arguably the simplest forms of bad signals during training, and their inference-time counterparts of exploding or vanishing internal signals stem from similar numerical problem. There are broader problems of internal covariate shift, where distributions of these internal signals move during training in such a way that makes the network struggle from an internal moving-target problem. To tackle these broader issues, normalization techniques are used. Notably, `BatchNorm` and `LayerNorm` serve as strong baseline normalization techniques that are useful depending on the use case. Mathematically, both methods normalize using the same equation shown below.

$$y = \gamma\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{35}$$

In this formulation, γ and β operate as the linear map and translation that start as 1 and 0, respectively. μ and σ^2 are the mean and variance terms that are calculated so that the inputs are first normalized to unit variance and zero mean before an affine transformation is applied. ϵ in this formula is a sufficiently small constant that exists to prevent division by zero. Specifically, `BatchNorm` is a normalization method that brings each feature into a distribution that has zero mean and unit variance by calculating the mean and variance of that feature across the entire batch. Thus, in order to be of use, the training method utilizing this method must train in mini-batches. `LayerNorm` performs the same normalization but solves the variance and mean across the feature set for each individual sample. This difference is best understood through a visualization of the relevant data for a given input as shown in Fig. 3.

Because `BatchNorm` solves the mean and variance estimate over the mini-batch, it can also internally store a running average of these terms. This is useful during inference where the batch size is 1 so that the mean and variance terms are meaningful. Notably, because this normalization may be undesirable regarding how it impact the layer's expressivity, the γ and β terms are learnable parameters that are trained alongside the weights and biases of the network. `BatchNorm` is best in cases where the train and test distributions are similar (so the running statistic is accurate) and when data is independent and identically distributed. The mini-batching induces noise in the sampled mean and variance calculations, which allows `BatchNorm` to function as a regularizer as well as a normalizer. `BatchNorm` is often suited for cases using images where the RGB channels each individually can be normalized across the batch and spatial dimensions which gives additional redundancy. As a result, `BatchNorm` is poorly suited for cases where batch-size needs to be small, training and inference distributions are different or nonstationary, or if data comes from coupled samples/features. `LayerNorm` can end up voiding information that is useful from magnitudes of the hidden states, and the normalization across the feature space will couple all of the features through the shared mean and variance.

Overall, `LayerNorm` and `BatchNorm` help stabilize hidden state dynamics, which can improve optimization; however, normalization also removes certain scale and shift information. As a result, the normalized output can become insensitive to transformations of the input group that scale and shift it uniformly. This invariance is not identical for `LayerNorm` and `BatchNorm`, since the normalized groups are different, but the underlying mechanism is the same.

To see this, we will revisit the normalization operation in Eq. (35). Suppose that the entire input group is scaled by a positive constant $c > 0$ and shifted uniformly by a constant α, so that $x' = cx + \alpha$ where α

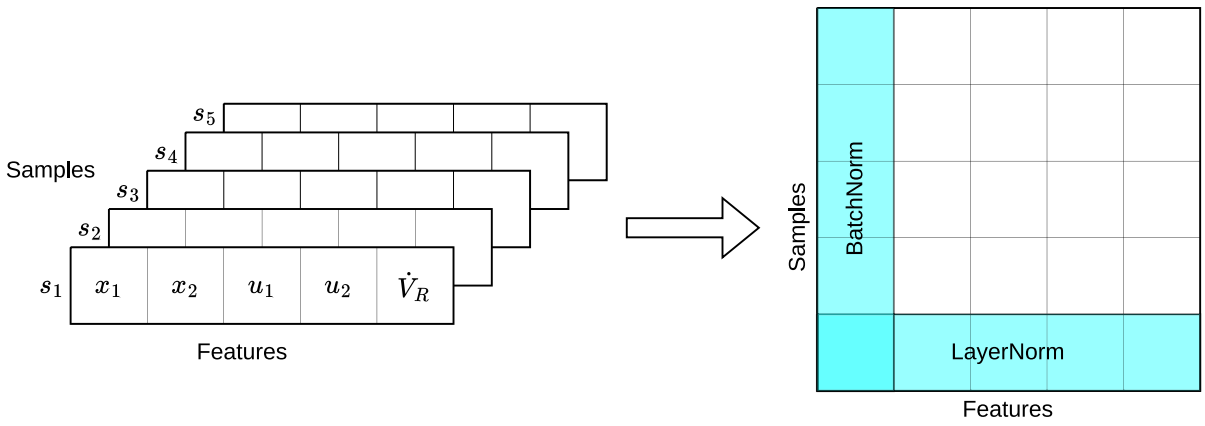

Fig. 3. Visualization of the different dimensions that BatchNorm and LayerNorm use for their normalization calculations.

is assumed to be a vector of sufficient shape to match the normalized input group with elements that are all equal to the same constant. In other words, α is a uniform shift. The corresponding mean satisfies:

$$\mu' = \frac{1}{d}\sum_{i=1}^{d}\left(cx_i + \alpha\right) = c\mu + \alpha \tag{36}$$

and the corresponding variance satisfies:

$$\sigma'^2 = \frac{1}{d}\sum_{i=1}^{d}\left(\left(cx_i\right) + \alpha - \left(c\mu + \alpha\right)\right)^2 = \frac{1}{d}\sum_{i=1}^{d}\left(c\left(x_i - \mu\right)\right)^2 = c^2\sigma^2 \tag{37}$$

With $\epsilon > 0$, this invariance is approximate whenever ϵ is small relative to σ^2:

$$y' = \gamma\frac{x' - \mu'}{\sqrt{\sigma'^2 + \epsilon}} + \beta = \gamma\frac{c\left(x - \mu\right)}{\sqrt{c^2\sigma^2 + \epsilon}} + \beta \approx \gamma\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta = y \tag{38}$$

However, ϵ is included only to prevent division by zero, and in practice it is chosen to be small enough that it is negligible relative to the normalized variance. Therefore, for the purpose of analyzing scale-invariance, we take $\epsilon = 0$. In this case, the normalized output is exactly invariant to positive uniform scaling and uniform shifting:

$$y' = \gamma\frac{x' - \mu'}{\sqrt{\sigma'^2}} + \beta = \gamma\frac{c\left(x - \mu\right)}{\sqrt{c^2\sigma^2}} + \beta = \gamma\frac{x - \mu}{\sqrt{\sigma^2}} + \beta = y \tag{39}$$

Thus, the general invariance condition for the normalization operation is:

$$x' = cx + \alpha, \qquad c > 0 \tag{40}$$

where α denotes a uniform shift over the normalized input group. In other words, normalization removes positive uniform scaling and uniform shifting over the entries used to compute the normalization statistics.

The parameter block that produces this invariant transformation depends on the normalization method. Consider the case where x is the output of some layer in a NN prior to the application of the activation function. We can formulate the per-sample output of the layer as:

$$x^{(n)} = Wh^{(n)} + b \tag{41}$$

where $h^{(n)}$ is the input to the layer for sample n. For `LayerNorm`, the normalized group is the feature vector $x^{(n)}$ for a single sample. For `BatchNorm`, each feature is normalized separately across the batch, so a batch-wise formulation must be considered.

For `BatchNorm`, the normalized group associated with feature j is:

$$x_j = \left[x_j^{(1)}, x_j^{(2)}, \ldots, x_j^{(B)}\right]^\top \tag{42}$$

where each element is given by:

$$x_j^{(n)} = w_j^\top h^{(n)} + b_j \tag{43}$$

Here, w_j is a column vector whose transpose $w_j^\top$ forms the jth row of W. Now, suppose that the incoming weight vector for feature j is scaled by $c > 0$, while the bias is allowed to become any value $\bar{b}_j$. This includes the cases where the bias is unchanged, scaled, shifted, or replaced entirely. Then:

$$x_j'^{(n)} = cw_j^\top h^{(n)} + \bar{b}_j \tag{44}$$

This can be rewritten relative to the original feature value as:

$$x_j'^{(n)} = c\left(w_j^\top h^{(n)} + b_j\right) + \left(\bar{b}_j - cb_j\right) \tag{45}$$

or, equivalently,

$$x_j'^{(n)} = cx_j^{(n)} + \alpha_j \tag{46}$$

where:

$$\alpha_j = \bar{b}_j - cb_j \tag{47}$$

Since α_j is the same for every sample n in the batch because the bias of a given feature does not vary across the batch dimension, the full batch-wise normalized group satisfies:

$$x_j' = cx_j + \alpha_j \tag{48}$$

where α_j is added uniformly across the batch. This is exactly the general invariant form of the normalization operation. Therefore, for `BatchNorm`, the scale-invariant parameter block for feature j is the jth row of W. Notably, the bias term is invariant broadly and can be removed entirely from the layer that feeds into `BatchNorm`. The fact that the bias gets canceled is seen in the general invariant form's proof in Eq. (36) where the mean of the normalized block perfectly preserves the uniform shift. By definition, if a feature is normalized over the batch, and the bias of a feature is constant for all batches, then the bias of a layer is itself a uniform shift that is canceled in the subtraction operation that occurs in the numerator of the normalization equation.

For `LayerNorm`, normalization is applied across the features of a single sample. For sample n, the normalized group is the full vector:

$$x^{(n)} = Wh^{(n)} + b \tag{49}$$

Unlike `BatchNorm`, a general bias vector does not automatically produce a uniform shift over the normalized group, because the normalized group is now the feature vector within one sample rather than one feature across many samples. To see the condition for exact invariance, suppose that the weights and biases are scaled separately:

$$W' = cW \qquad b' = db \tag{50}$$

where $c > 0$ and d is a scalar. Then the new input to `LayerNorm` is:

$$x'^{(n)} = W'h^{(n)} + b' = cWh^{(n)} + db \tag{51}$$

Rewriting this relative to the original input gives:

$$x'^{(n)} = c\left(Wh^{(n)} + b\right) + (d - c)\, b \tag{52}$$

or, equivalently,

$$x'^{(n)} = cx^{(n)} + (d - c)\, b \tag{53}$$

For this to have the general invariant form:

$$x'^{(n)} = cx^{(n)} + \alpha \tag{54}$$

where α is added uniformly to every feature in the normalized group, the mismatch caused by scaling the bias differently from the weights must satisfy:

$$(d - c)\, b_i = \alpha, \qquad i = 1, \ldots, d \tag{55}$$

for some scalar α. Equivalently, every entry of $(d - c)\, b$ must be the same constant.

Under this condition, the mean and variance over the features of sample n satisfy the general invariant form of the normalization operation. Therefore, for `LayerNorm`, exact invariance is achieved through scaling of the entire weight matrix as opposed to the row-by-row basis that `BatchNorm` satisfied. For the bias, `LayerNorm`'s invariance condition varies based on the structure of the bias vector itself. Generally, $d = c$ is a sufficient condition for invariance, but d can differ from c if b is a constant vector, i.e., every element is the same constant. Unlike `BatchNorm`, the bias term is not always canceled out, so reduction to a zero-bias input is not inherently the best choice.

The invariance described above has a direct consequence for the gradients of the parameter blocks θ that produce the invariant transformation. In these cases, the output, and thus the outputs impact on the loss function, are invariant for positive radial scaling of θ:

$$\mathcal{L}(c\theta) = \mathcal{L}(\theta), \qquad c > 0 \tag{56}$$

where the corresponding gradient is thus zero with respect to c:

$$\frac{d}{dc}\mathcal{L}(c\theta)\bigg|_{c=1} = \theta^{\top}\nabla_{\theta}\mathcal{L}(\theta) = 0 \tag{57}$$

In other words, the loss gradient is orthogonal to the radial scaling direction of the invariant parameter block; however, this does not mean that gradient descent preserves the magnitude of θ. Let $g = \nabla_{\theta}\mathcal{L}(\theta)$ and consider the gradient descent update:

$$\theta^{+} = \theta - \eta g \tag{58}$$

The invariant parameter blocks norm after the update is:

$$\left\|\theta^{+}\right\|^2 = \|\theta - \eta g\|^2 \tag{59}$$

which expands to:

$$\left\|\theta^{+}\right\|^2 = \|\theta\|^2 - 2\eta\theta^{\top}g + \eta^2\|g\|^2 \tag{60}$$

Due to the orthogonality from earlier, this becomes:

$$\left\|\theta^{+}\right\|^2 = \|\theta\|^2 + \eta^2\|g\|^2 \tag{61}$$

And so even though the scale direction has no gradient term, the orthogonal movement still results in an increase in the new vector's norm. This results in a problem; the layers weights do not impact the loss, and yet the loss increases the weights, so the network silently explodes until it suddenly fully breaks down at the limit of the numerical precision that is chosen. A workaround to this issue is weight decay, where every update to the weights involves a removal of a small percent of the original weights. With weight decay, the update becomes:

$$\theta^{+} = (1 - \eta\lambda)\,\theta - \eta g \tag{62}$$

where the new norm is:

$$\left\|\theta^{+}\right\|^2 = (1 - \eta\lambda)^2\|\theta\|^2 + \eta^2\|g\|^2 \tag{63}$$

In this formulation, the growth in the norm from the orthogonal direction addition is counteracted by the new percent removal term; however, the invariance changes the interpretation of weight decay on normalized layers. Since positive radial scaling of θ induces an invariant transformation of the normalized input group, the magnitude of θ no longer impacts the loss function. Instead, its impact during training is primarily seen in how it interacts with the orthogonal component. Because the updates are orthogonal to the scale, we can quantify them like a right triangle where for some update $\Delta\theta = -\eta g$ orthogonal to θ we get:

$$\tan(\phi) = \frac{\|\Delta\theta\|}{\|\theta\|} = \frac{\eta\|g\|}{\|\theta\|} \tag{64}$$

The tangent function is roughly linear for small values, which is expected for networks with small step sizes, thus the angle of this right triangle vector shift operation is roughly $\frac{\eta\|g\|}{\|\theta\|}$. For contrast, the relative growth of the norm is:

$$\frac{\left\|\theta^{+}\right\| - \|\theta\|}{\|\theta\|} \approx \frac{1}{2}\phi^2 \tag{65}$$

where a Taylor series expansion is used for the case where $\|\Delta\theta\| \ll \|\theta\|$. This tells us three things for small steps: the angle is roughly the magnitude of the step update, the angle is inversely proportional to the current norm, and the relative impact of the step to the norm is proportional to the squared angle. These properties lead to smaller relative change when $\|\theta\|$ is large and a larger relative change when $\|\theta\|$ is small. In other words, the norm functions as an effective learning-rate. Consequently, weight decay partially acts as an effective learning-rate controller for these scale-invariant parameter blocks rather than only as a conventional regularizer as is usually true. This is not necessarily desirable, as having variable learning rates within a network complicates the design process as learning rates are a critical hyperparameter. This motivates normalization-and-projection (NaP) methods, which explicitly control the magnitude of scale-invariant parameter blocks in order to maintain a more consistent effective learning rate (Lyle et al., 2024). In practice, these methods take many forms, but the core principle is that the normalization is handled by blocks such as `LayerNorm` and the projection is an extra step done during training that takes the existing weights and consistently projects them so that they satisfy certain criteria.

Remark 3. The discussion thus far is under the assumption that the normalization via either `BatchNorm` or `LayerNorm` is applied prior to the NN-layer's activation function. Equivalently, the discussion holds true for positively homogeneous activation functions such as `ReLU` due to the fact that $\texttt{ReLU}(cx) = c\texttt{ReLU}(x)\,\forall c > 0$. For cases where some activation function that does not satisfy positive homogeneity is used prior to the normalization operation, the above results do not generally hold true.

Remark 4. In the context of modern transformer architectures, normalization techniques are applied in a variety of methods beyond the pre-activation form demonstrated here due to the stabilization properties of the normalization process. Older frameworks use what is referred to as post-norm, where the layer and its residual are added and this is the input to a `LayerNorm`. More modern approaches utilize pre-norm with simpler normalization techniques such as `RMSNorm` which removes the mean-centering operation from Eq. (35), where the residual branch's input is normalized prior to entering the layer. There exist many variants of these which attempt to strike a balance between stability and expressiveness, but key focus is that the invariance property is a secondary consideration relative to the normalization operations ability to stabilize data flow.

3.3.3. Safe networks

By design, standard NNs do not have strict constraints on their outputs. As a result, there needs to be additional consideration before NNs can be used in real-world process control applications as operating constraints are a necessity for safe and stable operation. Thus, a subset of viable NN frameworks, referred to as Safe NNs, is considered. These networks are capable of enforcing the required constraints on their outputs in a strict manner, thereby allowing them to function as viable controllers regardless of the quality of the network itself. One such group of networks are networks using differentiable Quadratic Programming (QP)-layers. QP, shown in its standard form in Eq. (66), is a subset of nonlinear constrained optimization problems in which the objective is quadratic and the constraints are linear and have the following general form:

$$x^* = \arg\min_x \frac{1}{2} x^T P x + q^T x \tag{66a}$$

$$\text{s.t.} \quad Gx \leq h \tag{66b}$$

$$Ax = b \tag{66c}$$

In particular, these formulations operate with convex objectives due to the various beneficial mathematical properties of such functions. As such, the P matrix is an element of the set of symmetric positive-semidefinite matrices of size $n \times n$, whereas $q \in \mathbb{R}^n$, $G \in \mathbb{R}^{m \times n}$, $h \in \mathbb{R}^m$, $A \in \mathbb{R}^{p \times n}$, and $b \in \mathbb{R}^p$ represent the miscellaneous real-valued problem data upon which $x \in \mathbb{R}^n$ is optimized. This formulation for convex QP problems, defined by CVXPY, is used as the baseline for CVXPY layers; a differentiable layer form of the convex optimization problem that can be inserted into NNs (Agrawal et al., 2019). More generally, one can ensure constraints are satisfied by using these layers for constraint enforcement; however, these layers come with a significant computational cost increase due to the use of iterative methods. One of the main reasons for using NNs as controllers is the computational benefits that arise from the inherent benefit of data-based methods; the ability to model high-dimensional problems through low-dimensional approximations. Several works have demonstrated that NN-based controllers can achieve comparable performance to MPC-based approaches with significant computational complexity reduction. By using QP-layers, the distinguishing factor between these two approaches diminishes, and the result is a notable increase in computational complexity. Additionally, such designs may not be compatible with modern techniques for speeding up the computation of NNs, which would further broaden the computational cost gap. Some alternative approaches, such as RAYEN, avoid orthogonal projection logic and instead opt to use projection with respect to an interior point, which enables significant computational overhead improvements over QP-layers (Tordesillas et al., 2023). Regardless, there is a critical drawback to all such projection methods that had adverse impacts on the training of the model; gradient loss.

To visualize the problem, we explore the linear inequality case. Consider an arbitrary unconstrained output from the NN denoted $z \in \mathbb{R}^n$. To be considered safe, the constrained output $y = f(z)$ must satisfy some set of linear inequalities described by:

$$Cy \leq d \tag{67}$$

However, not all constraints are necessarily active at the projected output. Thus, it is beneficial to analyze the local behavior of the projection using only the active boundary constraints. Let $\mathcal{A}$ denote the locally fixed active set at y, and define $A = C_{\mathcal{A}} \in \mathbb{R}^{p \times n}$ and $b = d_{\mathcal{A}} \in \mathbb{R}^p$. The rows of A therefore collect the normals of the active constraint boundaries, which are locally represented by:

$$Ay = b \tag{68}$$

The remaining inactive constraints satisfy:

$$C_I y < d_I \tag{69}$$

To generalize the formulation, assume the projection from z to y occurs along known correction directions collected in $V \in \mathbb{R}^{n \times p}$ such that:

$$y = z + V\lambda \tag{70}$$

where λ is chosen so that $Ay = b$. Substitution gives:

$$A(z + V\lambda) = b \tag{71}$$

$$Az + AV\lambda = b \tag{72}$$

$$AV\lambda = b - Az \tag{73}$$

$$\lambda = (AV)^{-1}(b - Az) \tag{74}$$

assuming that AV is nonsingular. Thus, the projection and its corresponding Jacobian are:

$$y = z + V(AV)^{-1}(b - Az) \tag{75a}$$

$$J = \frac{\partial y}{\partial z} = I - V(AV)^{-1}A \tag{75b}$$

The above formulation has several notable properties. By multiplying by A, it is seen that:

$$AJ = A - AV(AV)^{-1}A = 0 \tag{76}$$

Therefore, for any perturbation δz, the corresponding constrained-output perturbation $\delta y = J\delta z$ satisfies:

$$A\delta y = 0 \tag{77}$$

In other words, the forward pass through this constraint enforcement layer allows the projected output to be locally perturbed only along directions that preserve the currently active constraint values. For active inequality constraints, this means that both outward perturbations that would increase violation and inward perturbations that would increase slack are removed; the surviving directions are those that move along the active boundary.

As for the backward pass, which is critical to the training of the neural network, the gradient that flows to the NN layers prior to the constraint enforcement layer is defined as:

$$g_z = J^\top g_y \tag{78}$$

where g_y is the gradient of the loss with respect to the constrained output. Using the expression for J, this becomes:

$$g_z = g_y - A^\top (AV)^{-\top} V^\top g_y \tag{79}$$

This gradient has a corresponding orthogonality property with respect to V:

$$V^\top g_z = V^\top g_y - V^\top A^\top (AV)^{-\top} V^\top g_y \tag{80}$$

$$= V^\top g_y - (AV)^\top (AV)^{-\top} V^\top g_y \tag{81}$$

$$= 0 \tag{82}$$

This property demonstrates that the backward gradient passed into the shallower layers of the NN is orthogonal to the directions in which the projection occurs. In other words, the gradient signal calculated from the loss with respect to the constrained output is modified along the span of the active constraint normals until the remaining signal is orthogonal to the projection directions.

If the NN were trainable in a manner that allowed the output to be directly modified by gradient descent, then the physical intuition of the constraint enforcer is that the projected control action can only be modified, to first order, in directions that do not change the currently active constraint values. This means that even if the gradient contains a component that would improve the active constraint satisfaction, i.e., by moving the action towards or into the safe set, that component is removed by the local projection Jacobian. Therein lies the primary limitation of strict constraint enforcers within neural networks: the

inability to retain information along the directions eliminated by the projection.

To demonstrate this, we will consider two examples. First, consider the case in which the projection is optimized through a QP-layer such that the minimal projection that satisfies the constraints is used. Specifically, we consider the Lyapunov constraint and actuator bound constraints from the LMPC formulation in Eq. (6). In this case, the constrained output is the solution to:

$$u^* = \arg\min_u \frac{1}{2}\left(u - u_{NN}\right)^\top \left(u - u_{NN}\right) \tag{83a}$$

$$\text{s.t.} \quad u_{\min} \le u \le u_{\max} \tag{83b}$$

$$\dot{V}(x,u) \le \dot{V}\left(x, u_R\right) \tag{83c}$$

where u_{NN} denotes the unconstrained NN output, u^* denotes the constrained control action, and u_R denotes the reference control action used in the Lyapunov constraint. For a fixed state x, the Lyapunov derivative can be written as:

$$\dot{V}(x,u) = L_f V(x) + L_g V(x)^\top u \tag{84}$$

where $L_f V(x)$ is independent of u and $L_g V(x)^\top$ is the row vector multiplying the control input. Therefore, the Lyapunov constraint can be rewritten as:

$$L_f V(x) + L_g V(x)^\top u \le L_f V(x) + L_g V(x)^\top u_R \tag{85}$$

$$L_g V(x)^\top u \le L_g V(x)^\top u_R \tag{86}$$

Thus, for a fixed state, the Lyapunov constraint is linear in u. Similarly, the actuator bounds can be written as two separate sets of linear inequality constraints:

$$u_{\min} \le u \quad \Rightarrow \quad -u \le -u_{\min} \tag{87}$$

$$u \le u_{\max} \tag{88}$$

The QP-layer projection is therefore governed by three linear inequality constraint families: the lower actuator bounds, the upper actuator bounds, and the Lyapunov constraint. Because the optimization problem is formulated with a quadratic objective and linear constraints, the Karush-Kuhn–Tucker (KKT) conditions can be used to explicitly characterize the optimal control action. The Lagrangian of the inequality-constrained problem can be written as:

$$\mathcal{L}\left(u, \mu_{\min}, \mu_{\max}, \mu_L\right) = \frac{1}{2}\left(u - u_{NN}\right)^\top \left(u - u_{NN}\right) \tag{89}$$

$$+ \mu_{\min}^\top \left(-u + u_{\min}\right) \tag{90}$$

$$+ \mu_{\max}^\top \left(u - u_{\max}\right) \tag{91}$$

$$+ \mu_L \left(L_g V(x)^\top u - L_g V(x)^\top u_R\right) \tag{92}$$

where $\mu_{\min}$, $\mu_{\max}$, and μ_L are nonnegative KKT multipliers. The stationarity condition is:

$$\partial_u \mathcal{L}\left(u^*, \mu_{\min}, \mu_{\max}, \mu_L\right) = u^* - u_{NN} \tag{93}$$

$$- \mu_{\min} + \mu_{\max} + \mu_L L_g V(x) \tag{94}$$

$$= 0 \tag{95}$$

Primal feasibility requires:

$$-u_i^* + u_{\min,i} \le 0, \qquad i = 1, \dots, m \tag{96}$$

$$u_i^* - u_{\max,i} \le 0, \qquad i = 1, \dots, m \tag{97}$$

$$L_g V(x)^\top u^* - L_g V(x)^\top u_R \le 0 \tag{98}$$

Dual feasibility requires:

$$\mu_{\min,i} \ge 0, \qquad i = 1, \dots, m \tag{99}$$

$$\mu_{\max,i} \ge 0, \qquad i = 1, \dots, m \tag{100}$$

$$\mu_L \ge 0 \tag{101}$$

Complementary slackness requires:

$$\mu_{\min,i}\left(-u_i^* + u_{\min,i}\right) = 0, \qquad i = 1, \dots, m \tag{102}$$

$$\mu_{\max,i}\left(u_i^* - u_{\max,i}\right) = 0, \qquad i = 1, \dots, m \tag{103}$$

$$\mu_L\left(L_g V(x)^\top u^* - L_g V(x)^\top u_R\right) = 0 \tag{104}$$

Complementary slackness demonstrates that each inequality constraint falls into one of two cases. If a constraint is inactive, then its multiplier is zero and it does not contribute to the stationarity condition. If a constraint has a nonzero multiplier, then the corresponding inequality constraint must hold with equality. Thus, the local form of the projection is derived from the stationarity condition after selecting the active constraints, i.e., the constraints whose multipliers are nonzero.

To analyze the local Jacobian of this projection, we consider the locally fixed active set after projection. Active upper-bound constraints correspond to rows with a 1 in the coordinate of the active actuator and 0 elsewhere, while active lower-bound constraints correspond to rows with a -1 in the coordinate of the active actuator and 0 elsewhere. That is, if the ith upper bound is active, the corresponding row is:

$$e_i^\top u = u_{\max,i} \tag{105}$$

whereas if the ith lower bound is active, the corresponding row is:

$$-e_i^\top u = -u_{\min,i} \tag{106}$$

where e_i denotes the ith standard basis vector for the Euclidean space. Let $A_{\max}$ and $A_{\min}$ collect the active upper-bound and lower-bound rows, respectively, and let $b_{\max}$ and $b_{\min}$ collect the corresponding upper-bound and lower-bound values, respectively. The Lyapunov constraint contributes an active row only when it is active at the projected output. In that case, define:

$$A_L = L_g V(x)^\top \tag{107}$$

and:

$$b_L = L_g V(x)^\top u_R \tag{108}$$

If the Lyapunov constraint is inactive, then this row is omitted from the active set. The full locally active constraint matrix can therefore be written as:

$$A_{\mathcal{A}} = \begin{bmatrix} A_{\min} \\ A_{\max} \\ A_L \end{bmatrix}, \qquad b_{\mathcal{A}} = \begin{bmatrix} b_{\min} \\ b_{\max} \\ b_L \end{bmatrix} \tag{109}$$

where any inactive constraint rows are omitted. The active constraints are therefore locally represented by:

$$A_{\mathcal{A}} u^* = b_{\mathcal{A}} \tag{110}$$

On this locally fixed active branch, the KKT stationarity condition reduces to:

$$u^* - u_{NN} + A_{\mathcal{A}}^\top \mu_{\mathcal{A}} = 0 \tag{111}$$

where $\mu_{\mathcal{A}}$ collects the active KKT multipliers. Solving for u^* gives:

$$u^* = u_{NN} - A_{\mathcal{A}}^\top \mu_{\mathcal{A}} \tag{112}$$

Substituting this expression into the active constraints gives:

$$A_{\mathcal{A}} u^* = b_{\mathcal{A}} \tag{113}$$

$$A_{\mathcal{A}}\left(u_{NN} - A_{\mathcal{A}}^\top \mu_{\mathcal{A}}\right) = b_{\mathcal{A}} \tag{114}$$

$$A_{\mathcal{A}} u_{NN} - A_{\mathcal{A}} A_{\mathcal{A}}^\top \mu_{\mathcal{A}} = b_{\mathcal{A}} \tag{115}$$

$$A_{\mathcal{A}} A_{\mathcal{A}}^\top \mu_{\mathcal{A}} = A_{\mathcal{A}} u_{NN} - b_{\mathcal{A}} \tag{116}$$

$$\mu_{\mathcal{A}} = \left(A_{\mathcal{A}} A_{\mathcal{A}}^\top\right)^{-1}\left(A_{\mathcal{A}} u_{NN} - b_{\mathcal{A}}\right) \tag{117}$$

assuming $A_{\mathcal{A}} A_{\mathcal{A}}^\top$ is nonsingular. Substitution into $u^* = u_{NN} - A_{\mathcal{A}}^\top \mu_{\mathcal{A}}$ gives:

$$u^* = u_{NN} - A_{\mathcal{A}}^\top \left(A_{\mathcal{A}} A_{\mathcal{A}}^\top\right)^{-1}\left(A_{\mathcal{A}} u_{NN} - b_{\mathcal{A}}\right) \tag{118}$$

Equivalently:

$$u^* = u_{NN} + A_{\mathcal{A}}^\top \left(A_{\mathcal{A}} A_{\mathcal{A}}^\top\right)^{-1} \left(b_{\mathcal{A}} - A_{\mathcal{A}} u_{NN}\right) \tag{119}$$

Therefore, the local Jacobian of the QP-layer projection with respect to the unconstrained NN output is:

$$J_{\mathcal{A}} = \frac{\partial u^*}{\partial u_{NN}} = I - A_{\mathcal{A}}^\top \left(A_{\mathcal{A}} A_{\mathcal{A}}^\top\right)^{-1} A_{\mathcal{A}} \tag{120}$$

Since $A_{\mathcal{A}} A_{\mathcal{A}}^\top$ is nonsingular, $A_{\mathcal{A}}$ has full row rank and:

$$A_{\mathcal{A}}^+ = A_{\mathcal{A}}^\top \left(A_{\mathcal{A}} A_{\mathcal{A}}^\top\right)^{-1} \tag{121}$$

Thus, the Jacobian can equivalently be written as:

$$J_{\mathcal{A}} = I - A_{\mathcal{A}}^+ A_{\mathcal{A}} \tag{122}$$

Although the original QP is formulated using inequality constraints, the local Jacobian is governed by the active rows at the projected output, as identified by the KKT conditions. These active rows are represented in the equality form $A_{\mathcal{A}} u^* = b_{\mathcal{A}}$ because active inequalities lie on their boundaries at the projected output. Therefore, the local Jacobian removes perturbation components normal to the active constraint boundaries and preserves only perturbations tangent to the active set. This can be verified directly by multiplying by $A_{\mathcal{A}}$:

$$A_{\mathcal{A}} J_{\mathcal{A}} = 0. \tag{123}$$

Therefore, for any perturbation δu_{NN}, the corresponding constrained-output perturbation $\delta u^* = J_{\mathcal{A}} \delta u_{NN}$ satisfies:

$$A_{\mathcal{A}} \delta u^* = 0. \tag{124}$$

In other words, the projected control action can only be locally perturbed along directions that preserve the currently active Lyapunov and actuator-bound constraint values. Furthermore, this projection is orthogonal which implies that the Jacobian is symmetric:

$$J_{\mathcal{A}}^\top = J_{\mathcal{A}} \tag{125}$$

Thus, the backward gradient passed to the unconstrained NN output is:

$$g_{u_{NN}} = J_{\mathcal{A}}^\top g_u^* = J_{\mathcal{A}} g_u^* \tag{126}$$

and it satisfies:

$$A_{\mathcal{A}} g_{u_{NN}} = 0 \tag{127}$$

Therefore, the backward gradient is also tangent to the currently active constraint boundary. For active actuator bounds, this means the gradient components corresponding to saturated actuator coordinates are zeroed out. For an active Lyapunov constraint, this means the backward gradient loses the component along the Lyapunov constraint normal $L_g V(x)$. As a result, once the QP-layer places the output on the active boundary, the NN does not receive gradient information through this local Jacobian in directions that would change the active Lyapunov or actuator-bound constraint values.

As a second example, consider a RAYEN-style projection applied to the same feasible set. Unlike the QP-layer projection, RAYEN does not compute the minimal Euclidean projection. Instead, it projects the unconstrained NN output along a ray from an interior point of the feasible set. Let u_0 denote this interior point and define:

$$r = u_{NN} - u_0 \tag{128}$$

The projected output is written as:

$$u^* = u_0 + \alpha r \tag{129}$$

where α is chosen so that u^* lies on the active boundary of the feasible set. For the locally active constraint row:

$$a_R^\top u^* = b_R \tag{130}$$

substitution gives:

$$a_R^\top (u_0 + \alpha r) = b_R \tag{131}$$

$$a_R^\top u_0 + \alpha a_R^\top r = b_R \tag{132}$$

$$\alpha = \frac{b_R - a_R^\top u_0}{a_R^\top r} \tag{133}$$

Thus:

$$u^* = u_0 + \frac{b_R - a_R^\top u_0}{a_R^\top r} r \tag{134}$$

where $a_R^\top$ is the active row selected from the same actuator-bound or Lyapunov constraint rows used in the QP-layer formulation.

Let:

$$\beta_R = b_R - a_R^\top u_0 \tag{135}$$

so that:

$$\alpha = \frac{\beta_R}{a_R^\top r} \tag{136}$$

Since $r = u_{NN} - u_0$, we have $dr = du_{NN}$. Differentiating $u^* = u_0 + \alpha r$ gives:

$$du^* = \alpha du_{NN} + r d\alpha \tag{137}$$

and differentiating $\alpha = \beta_R \left(a_R^\top r\right)^{-1}$ gives:

$$d\alpha = -\frac{\alpha}{a_R^\top r} a_R^\top du_{NN} \tag{138}$$

Therefore:

$$du^* = \alpha du_{NN} - r \frac{\alpha}{a_R^\top r} a_R^\top du_{NN} \tag{139}$$

$$= \alpha \left(I - \frac{r a_R^\top}{a_R^\top r}\right) du_{NN} \tag{140}$$

Thus, the local Jacobian of the RAYEN projection is:

$$J_R = \frac{\partial u^*}{\partial u_{NN}} = \alpha \left(I - \frac{r a_R^\top}{a_R^\top r}\right) \tag{141}$$

This has the same oblique projection structure as the general formulation, but the projection direction is the ray r rather than the active constraint normal a_R.

As in the QP-layer case, multiplying by the active row gives:

$$a_R^\top J_R = 0 \tag{142}$$

so the forward perturbation $\delta u^* = J_R \delta u_{NN}$ satisfies:

$$a_R^\top \delta u^* = 0 \tag{143}$$

Thus, the projected output can only be locally perturbed along directions that preserve the currently active constraint value.

However, unlike the QP-layer projection, J_R is generally not symmetric. The backward gradient is:

$$g_{u_{NN}} = J_R^\top g_{u^*} = \alpha \left(g_{u^*} - a_R \frac{r^\top g_{u^*}}{a_R^\top r}\right) \tag{144}$$

Multiplying by $r^\top$ gives:

$$r^\top g_{u_{NN}} = 0 \tag{145}$$

Therefore, the backward gradient is orthogonal to the ray direction r, rather than the active constraint normal a_R. This is the key distinction from the QP-layer case. For the QP-layer projection, the projection direction is the active constraint normal, making the projection orthogonal. For RAYEN, the projection direction is the ray from the interior point to the unconstrained NN output, making the projection oblique. As a result, RAYEN does not remove the gradient loss issue; it changes

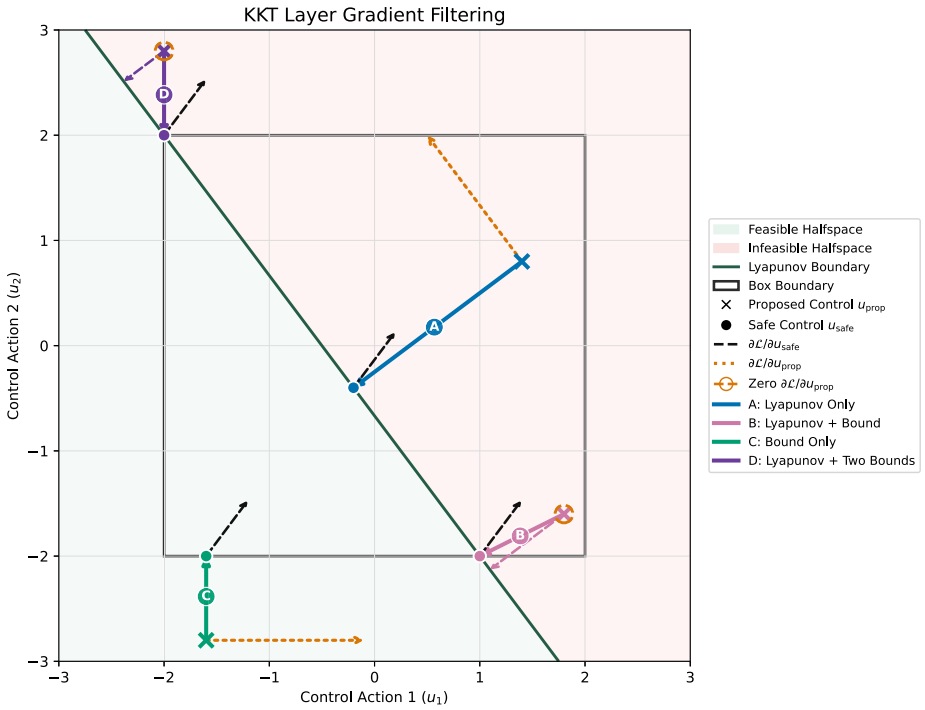

Fig. 4. Visualization of the impact on the gradient direction from KKT projection of a toy problem with box and Lyapunov constraints. The gradient in all four cases is assumed to be the same with respect to the projected loss in order for the impact on the gradient to be consistent and clear. Because the system is two dimensional, it is not possible to be orthogonal to 2 non-parallel vectors at once, hence why violation of the Lyapunov constraint and at least one bound results in no gradient.

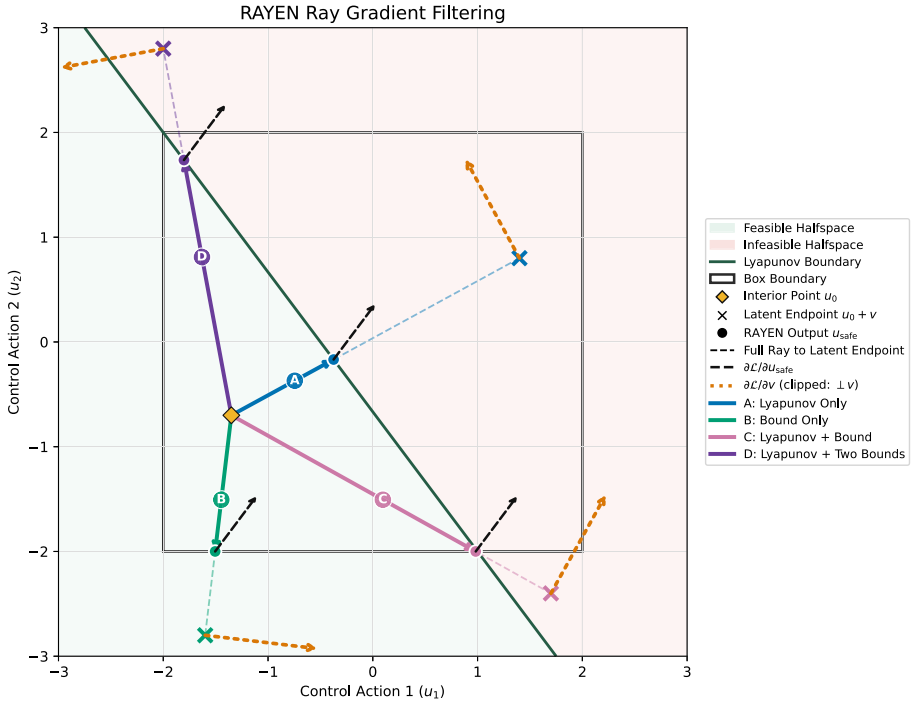

Fig. 5. Visualization of the impact on the gradient direction from RAYEN-based projection of a toy problem with box and Lyapunov constraints. The gradient in all four cases is assumed to be the same with respect to the projected loss in order for the impact on the gradient to be consistent and clear. Because the projection direction is based on the ray alone, violating the Lyapunov and bound constraints at once does not result in a zeroing of the gradient.

the direction in which gradient information is lost. This distinction is visualized in Figs. 4 and 5.

As is seen in these simpler examples, projection methods of various forms lead to a kind of destructive interference of the gradient signal in which the only remaining modality of learning for constraint-violating samples is to modify weights in ways that have no effect on the currently active constraint values. This occurs for bad signals that would otherwise lead to even worse violations, but it also occurs for beneficial signals that would allow for simultaneous cost function and constraint improvements. More broadly, because the forward pass in

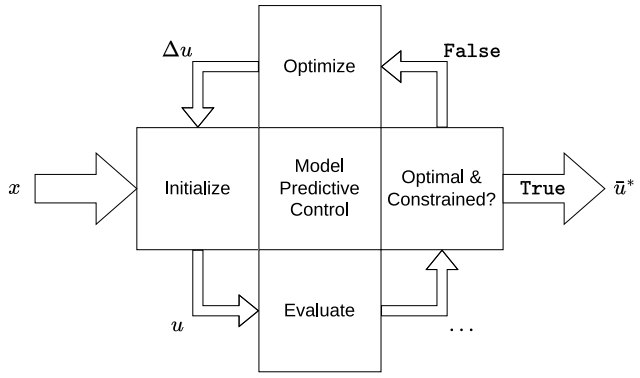

Fig. 6. Block diagram representation of a Model Predictive Control loop. Evaluation criteria are left as dots as the values solved vary based on the solver chosen. The optimization step simultaneously optimizes for the objective and constraint satisfaction.

constraint-violating regions applies a projection, the constrained output can become locally insensitive to changes in the unconstrained NN output along the projection directions. In other words, when constraints are active, hard constraint enforcement functions as a locally rank-deficient mapping from the unconstrained output space to the active feasible boundary. Therefore, multiple unconstrained outputs can map to the same constrained output, resulting in a loss of information in the backward pass gradient. Thus, safe NNs cannot directly learn how to improve the loss through the local estimate of the impact of improved constraint satisfaction, as this signal is lost. This is the remaining barrier that prevents safe NNs from being considered true replacements to MPC, and can be visualized through Figs. 6 and 7. Whereas MPC is an iterative loop that modifies its proposed control action trajectory based on constraint satisfaction and objective optimality simultaneously, NNs train through an iterative loop that modifies their weights and biases through these gradients that, depending on the design and positioning of the constraint enforcement, have varying degrees of information loss in constrained cases that all result in the inability to improve both metrics simultaneously.

3.3.4. Straight-Through Estimators (STEs)

Whereas QP-layers and equivalent methods for strict constraint enforcement propose frameworks that enable a broad category of problems to be modeled under a unified design, there exists a lesser known approach that is often used outside of the network for external enforcement, but has not seen significant research as an alternative method for strict internal constraint enforcement; gates. In many ways, gates are similar to projection methods, but operate in a simpler modular form. As an example, a constraint that selects between two functions based on an if-else statement can be formulated as a single expression using the Heaviside step function. The consequence of these hard gates is that such functions do not have well-defined derivatives.

A workaround to this would be to use a straight-through estimator (STE) where the forward pass and backward pass are defined with respect to different functions. This allows a strict forward pass using non-differentiable techniques while the backward pass must be some differentiable approximation. In the case of the Heaviside step function, some good approximates are shown in Fig. 8. Notably, a direct estimate of the Heaviside function can be done by modulating input weight of the `Sigmoid` function (σ). The consequence is that as this weight increases, so does the scale of the gradient. Because the unweighted sigmoid has a peak gradient of 0.25, it is seen that a weight of 4 will result in a peak gradient of 1. While one can improve the estimation error, the impact of this choice on gradients in deep networks is worth noting. The `ReLU (tanh)` approach is another valid forward pass method which can be used to guarantee enforcement of one of the conditions but not the other. In particular, if the input is reliably large enough (roughly larger than 3) then the enforcement is a close approximation of strict enforcement.

Alternatively, several surrogates can be utilized for the backward pass gradient calculations. A particularly viable option is `tanh (softplus)` which can also be scaled through `softplus`'s β term, which converges to `tanh (ReLU)` in the limit. This option is notable compared to other `ReLU` estimates such as `Mish` or `Swish` due to the non-negativity of the resulting gradient. These options highlight two approaches that can be used during training: a purely hard constraint enforcement gate whose gradient must be approximated via straight-through-estimation, or a soft constraint enforcement gate whose gradient can be approximated via straight-through-estimation if desired but is not required.

A core benefit of these STE-based gates comes from the gradient behavior. Unlike the projection or QP-layer approaches, the gradient in the case of the `softplus` and `Sigmoid` surrogate approaches but is never truly zero. In other words, there is signal that flows back to the weights of the NN that does not suffer from the orthogonality issues of strict enforcers. Even so, as an estimate, there is still an effective modeling error within the training loop where the gradient is a mix of terms from either the surrogate pass or the true forward pass.

To see the impact of the STE on the gradient terms, as was done with the KKT and RAYEN cases, we will consider the case of box and Lyapunov constraints. For brevity, we will denote the difference between the proposed and reference control action vectors as:

$$\delta u = u_{NN} - u_R \tag{146}$$

and denote the input to the gate as:

$$z = -l^\top \delta u \tag{147}$$

where l is shorthand for $L_g V(x)$. It is convenient to formulate the gate as an alpha-gate where α is defined, in the hard case, by:

$$\alpha = \text{Heaviside}(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases} \tag{148}$$

In this formulation, we take the origin to be zero for the Heaviside function to allow for marginal conservativeness of the constraint satisfaction. The resulting constraint enforcement through replacement of the unsafe proposal with the safe reference can be done through:

$$u^* = \alpha u_{NN} + (1-\alpha) u_R \tag{149}$$

whose derivative is:

$$\mathrm{d}u^* = \alpha \mathrm{d}u_{NN} + \delta u \mathrm{d}\alpha \tag{150}$$

Because of the STE, the evaluation of α follows Eq. (148) whereas the derivative uses the form of α as follows:

$$\alpha = \tanh(\texttt{softplus}(z)) \tag{151}$$

The `softplus` equation is defined as:

$$\texttt{softplus}(x) = \frac{1}{\beta} \ln\left|1 + e^{\beta x}\right| \tag{152}$$

but for the sake of this proof we will assume $\beta = 1$. The resulting derivatives can be broken down into components. The derivative of α becomes a function involving the `Sigmoid` function σ defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{153}$$

The derivative of α with respect to the proposed output is broken into two parts, the first in terms of z:

$$\alpha'(z) = \frac{\partial \alpha}{\partial z} = \text{sech}^2(\texttt{softplus}(z))\,\sigma(z) \tag{154}$$

and the second in terms of u_{NN}:

$$\frac{\partial z}{\partial u_{NN}} = -l^\top \tag{155}$$

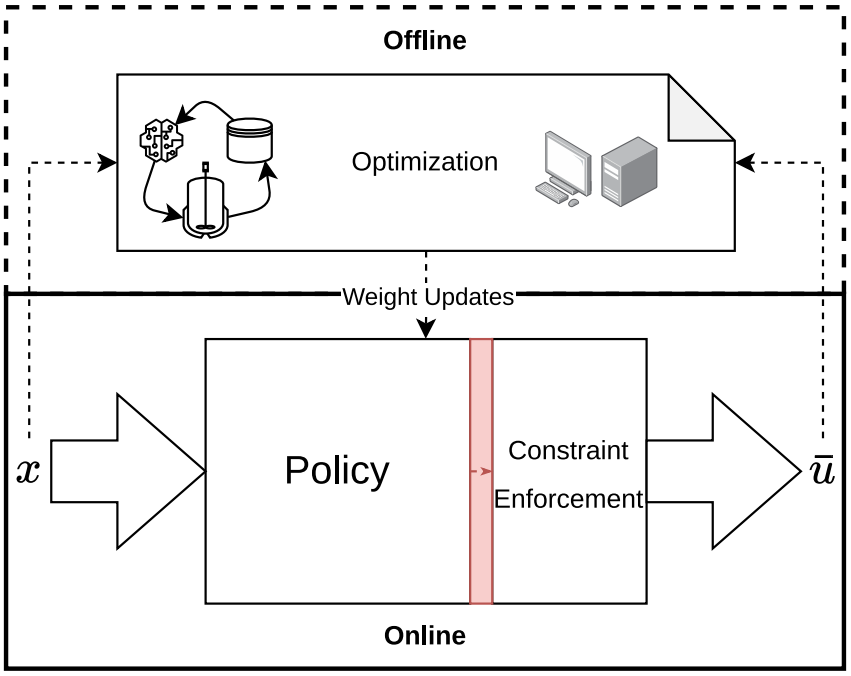

Fig. 7. Block diagram representation of a safe Neural Network controller. The red splice denotes the fact that constraint enforcement can either be implemented within the Policy neural network or externally as a standalone operation.

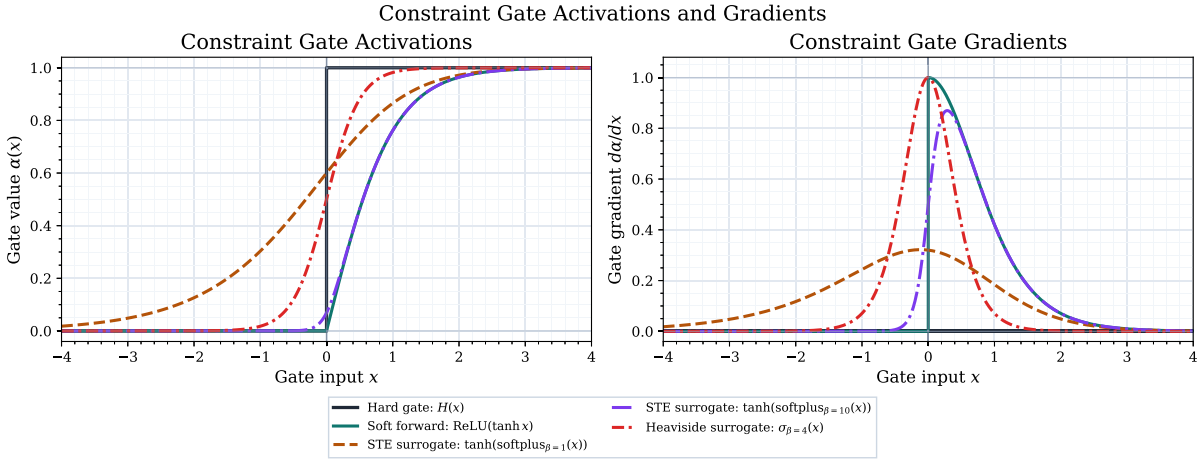

Fig. 8. Graphical comparison of several approximate replacement functions for the Heaviside step function.

These expressions are then substituted into the derivative of u^* to yield:

$$\mathrm{d}u^* = \alpha \mathrm{d}u_{NN} - \alpha'(z)(\delta u)\, l^\top \mathrm{d}u_{NN} \tag{156}$$

with the corresponding Jacobian of the safe output with respect to the proposed output:

$$J = \frac{\partial u^*}{\partial u_{NN}} = \text{Heaviside}(z)\, I - \alpha'(z)(\delta u)\, l^\top \tag{157}$$

This yields the following gradient to the shallower layers in the NN:

$$g_{u_{NN}} = J^\top g_{u^*} = \text{Heaviside}(z)\, g_{u^*} - \alpha'(z)\left(\delta u^\top g_{u^*}\right) l \tag{158}$$

Unlike the prior evaluations through multiplication and perturbation consideration, the existence of a hard and soft component in this expression causes the impact to collapse into sub-cases. We need to consider the 'depth' of feasibility, i.e., the proximity to the region near the feasibility boundary where the hard and soft approximations differ notably. If the system is feasible, the Heaviside function evaluates to 1, and if not, it evaluates to 0. If the feasibility is 'deep' in either direction, then the soft term is roughly equivalent to the Heaviside function, and so $\alpha' \approx 0$ collapses the gradient impact to solely consider the Heaviside term. In other words, strong satisfaction yields:

$$g_{u_{NN}} = J^\top g_{u^*} = \text{Heaviside}(z)\, g_{u^*} = g_{u^*} \tag{159}$$

which corresponds to no gradient loss, whereas strong violation yields

$$g_{u_{NN}} = J^\top g_{u^*} = \text{Heaviside}(z)\, g_{u^*} = 0 \tag{160}$$

which corresponds to complete gradient loss. In such cases, the impact on the gradient is equivalent to what clipping would achieve, a constraint enforcement that either preserves the entire gradient or removes the entire gradient. The benefit comes from the "shallow" cases where constraint satisfaction is near the boundary. In such cases, the soft term is non-zero, and so the gradient in the shallow violation case becomes:

$$g_{u_{NN}} = J^\top g_{u^*} = \text{Heaviside}(z)\, g_{u^*} - \alpha'(z)\left(\delta u^\top g_{u^*}\right) l = -\alpha'(z)\left(\delta u^\top g_{u^*}\right) l = -cl \tag{161}$$

The implication of this form is that the corresponding vector is some constant multiple of the vector l; however, the constraints are defined such that the constraint boundary is $l^\top$. This means that the remaining gradient is not parallel to the boundary as was the case for KKT or RAYEN. Instead, the remaining gradient is perpendicular to the boundary. The shallow satisfaction case is of similar form:

$$g_{u_{NN}} = J^\top g_{u^*} = \text{Heaviside}\,(z)\, g_{u^*} - \alpha'\,(z)\left(\delta u^\top g_{u^*}\right) l = g_{u^*} - cl \tag{162}$$

In this case, the expression retains the original gradient and instead perturbs it through this perpendicular term. The remaining question is whether or not these gradients are beneficial. The l component is a fixed vector that dictates the direction of the final gradient, which in both cases is either towards or away from the Lyapunov boundary depending on its sign. α' is a constant term whose sign is always positive, so it has no impact on direction and instead impacts the magnitude in a manner that is inversely proportional to how well it fits the Heaviside function. In other words, a near-perfect fitting to the Heaviside function would cause this term to be near-zero whereas bad fitting would allow some scale to persist. Thus, $\delta u^\top g_{u^*}$ is the constant that dictates whether the direction of the resulting gradient drives the gradient towards or away from the Lyapunov boundary. $\delta u^\top$ is a vector that always points from the reference action (which lies on the Lyapunov boundary) to the proposed action. g_{u^*} is the original gradient, which points in the direction that u^* should move to yield the largest first-order increase to the loss function. Because the loss function is often minimized through gradient descent, the actual direction that the control action is driven during updates is opposite to this. The product of these vectors is a constant $S = \delta u^\top g_{u^*}$. We can view the signage of S as an indicator for when $\delta u^\top$ and g_{u^*} point in the same direction. Conceptually, this means that when $S > 0$, moving from u_R to u_{NN} is estimated to increase the loss, and as such, gradient descent favors driving u_{NN} towards u_R. In such a case, the original gradient is perturbed by $-cl$, where $c > 0$, causing the update to u_{NN} to be driven towards the infeasible halfspace. The opposite is true when $S < 0$, as in such a case $c < 0$ yields a perturbation in the $+l$ direction, driving u_{NN} towards the feasible halfspace. As such, the impact on the gradient, and thus the impact on the driving force that modifies the update of u_{NN} per gradient descent step is an objective-weighted term that either tightens or worsens the constraint satisfaction based on the estimated impact on the objective. Of course, u_{NN} is not directly modified in NN training, but the logic remains true regarding how gradient descent attempts to modify the NN's raw output.

While this gradient behavior can be beneficial, it is not always so. Consider the four cases that can exist near the constraint boundary as depicted through Fig. 9. If the original gradient points towards the infeasible halfspace, then the perturbation will perturb this to be a stronger signal that aims more directly towards the Lyapunov boundary, thereby aggressively pushing the gradient descent step towards moving the control action away from the boundary to safer actions as is the case for point C. In this case, the gradient impact is favorable as it both improves on the objective while pushing for less risky actions. If the gradient instead points towards the feasible halfspace, then the perturbation will both amplify the magnitude of the vector and orient it so that the gradient descent step follows a shorter path towards the Lyapunov boundary as is true for case E. This case can be good or bad depending on how aggressive the steps are, as pushing too much towards the boundary may cause it to overshoot and result in fallback. As for the two cases when the proposed action is already unsafe, case D showcases how the gradient aiming towards the feasible halfspace would result in a gradient whose resulting step would directly push the action back towards the safe set along the shortest path to this set. This is a highly favorable case as it allows for a signal that ignores the objective signal, which is irrelevant information due to it attempting to push deeper into the infeasible halfspace, and instead entirely focuses on returning to the feasible halfspace where such signals would then be relevant. This leaves case B, where the gradient signal originally points towards the infeasible region, indicating that the objective is improved by moving towards the feasible region; however, the result is a gradient that aims perpendicular to the Lyapunov boundary, which causes the step to push the control actions directly towards the infeasible halfspace. This is the main problem that remains in the STE case. Whereas all the other cases can be favorable under sufficiently sized steps, this case is always unfavorable as it teaches the network to further destabilize an already unstable output. In all cases, the magnitude of this perturbation is scaled by both the value of the soft-estimate α' and the magnitude of the inner product between the original gradient and the reference difference vector. In other words, so long as the reference action is not very different from the proposed unsafe action, the resulting perturbations will not have a large impact, as is seen for all cases in Fig. 9.

Because the strength of this force is dependent on how well the soft estimation fits the Heaviside function, a poor estimation will yield a stronger force whereas a good estimation will yield a weaker one. This enables a method of controlling this constraint-informed driving force through the β term in the `softplus` function. Recall from Fig. 8 how the `tanh(softplus)` gradients behave when β is increased. As β approaches infinity, the driving force for negative inputs gets a weight of 0 whereas positive inputs keep some weight so long as they are near the origin. In other words, this formulation, despite being an imperfect estimation of the Heaviside step function in the limit relative to other smooth estimates like the `Sigmoid` function, allows for the undesired form of the force (case B) to decay to 0 in the limit while preserving most of the desired forms. The consequence is that the remaining signal for all violating cases, including case D, would collapse to the zeroing behavior of generic clipping methods.

Thus, the alpha-gate approach via STEs is potentially beneficial for constraint handling due to the addition of perturbations to the gradient that inform the model about the impact of opening or closing the constraint gate. Normally, such information would not exist due to Heaviside gates having no derivatives, but the STE solves this problem. The surrogate signal is often beneficial, but can also be detrimental. As such, the alpha-gate method is not without its flaws, and instead exists as another approach to consider when designing safe NNs.

3.3.5. Action handling in constrained RL

An often overlooked aspect of RL is what action to use for a given step in the training pipeline. Even in systems that are otherwise unconstrained, there are simple physical limits that must be respected by the proposed actions in the form of actuator limits. In the documentation of many training loops, such as the TD3 formulation from SpinningUp, clipping of the action to satisfy bounds is not universally done. SpinningUp's TD3 implementation, for example, does not clip the Actor's output that is fed into the Critic during the Actor loss calculation, but all other instances of action proposals (such as the post-noise exploration action and the post-noise future action for the Critic loss) clip the final action (Achiam, 2018).

The addition of constraints further complicates this, as there are now a total of 3 types of action forms; the raw policy output, the clipped policy output, and the fully constrained policy output. Regardless of framework, RL must provide the environment with the types of actions that are expected for real-world applications, and thus the environment must always receive the fully constrained action. Similarly, there is no reason to design the network with the intent of its outputs being infeasible to implement, and so exploration and thus the Critic loss terms often use the feasible clipped outputs. This still leaves room for variation in how the action is used in the Replay Buffer, Critic Loss, and Actor Loss calculations. Existing works provide a convenient pair of baseline templates referred to as `DPre+` and `DPro+` (Kasaura et al., 2023). In the case of `DPre+`, the action stored in the Replay Buffer and the future action solved for in the Critic Loss step are all the clipped actions, whereas the Actor loss step uses the unmodified policy output. `DPro+` is similar, but the action stored in the Replay Buffer is

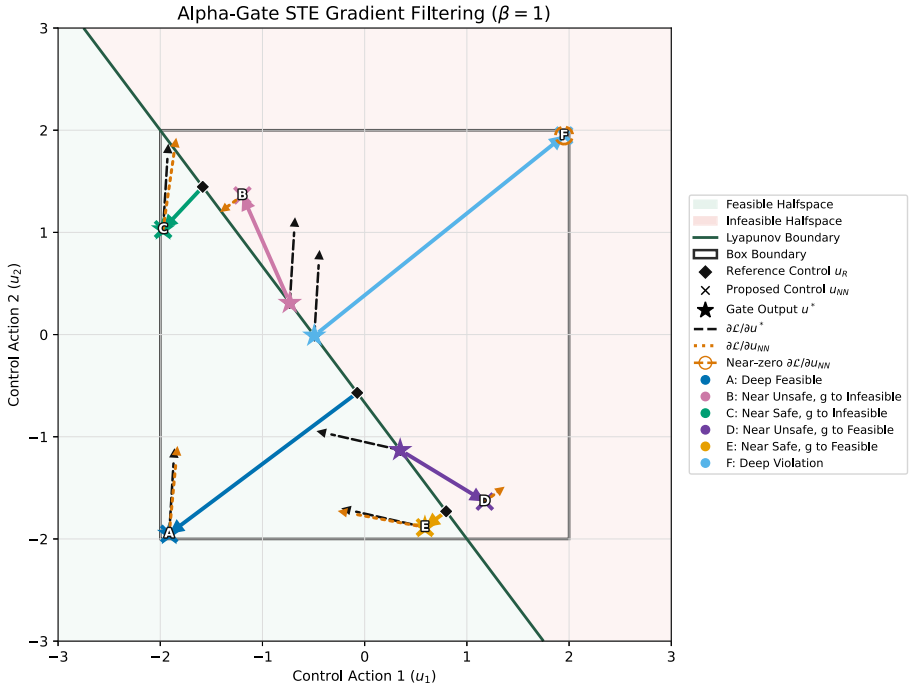

Fig. 9. Visualization of the impact on the gradient direction from constraint enforcement via reference controller fallback alpha-gating with STE on a toy problem with box and Lyapunov constraints. Two toy gradient signals are considered for the case where the signal aims towards and away from the infeasible halfspace. The soft estimate is assumed to use the `tanh(softplus)` form with $\beta = 1$. The box-constraint is assumed to always be satisfied due to the existence of `tanh` activation, which does not impact the resulting gradient directions. g is shorthand for $\frac{\partial \mathcal{L}}{\partial u}^*$.

Table 1
Action handling variants for RL training loops.

Mode	Environment action	Replay buffer action	Critic loss $Q(s,a)$	Critic loss $Q'(s',a')$	Actor loss $\rightarrow Q$	Penalty (+)
DPre+	Safe	Feasible	Feasible	Feasible	Policy	Policy
DPro+	Safe	Safe	Safe	Feasible	Policy	Policy
DProj+	Safe	Safe	Safe	Safe	Safe	Policy

Here, *Policy* denotes the actor proposal, *Feasible* denotes the actuator-clipped action that does not necessarily satisfy the Lyapunov safety constraint whereas *Safe* denotes the feasible action after projection/fallback such that it satisfies both the actuator bounds and the Lyapunov safety constraint. The + modifier adds a policy-action violation penalty to the actor loss.

the fully constrained action instead. We further consider an extension of these templates referred to as DProj+ where all instances of the action in the training loop are the fully constrained form. Additionally, this extension has an extra configuration referred to as DProj+∼ in which the baseline Actor loss of the training method uses a STE that estimates the backwards gradient of the full constrained action as if the unmodified policy output was used instead. In all of these templates, the + denotes an additional penalty term that is added to bias the network away from relying on the projection phenomenon. While the exact implementation of this penalty varies, with some implementations adding it to the reward function itself, it is sufficient to add it as an auxiliary loss term to the Actor loss that scales proportionally to the degree of constraint violation for the various constraints. These templates are summarized in Table 1. Each of these methods comes with their own set of theoretical drawbacks.

DPre+ yields a different formulation than the typical value function because it captures the augmented closed-loop system. In other words, the Critic estimates the value of submitting the unconstrained a into a system where the constraint enforcement is treated as a part of the environment dynamics. This poses a problem when exploration yields many unsafe actions for a given state that results in the same reward and future state after constraint enforcement because the Critic loses the ability to distinguish between what is a good or bad action to provide the system. Since the Actor is trained to maximize the Critic, the Actor may learn to output unsafe actions that are only valuable because they are rescued by the constraint enforcer. This is undesirable because the goal of the training loop is to use the Critic to yield a good Actor. Thus, one can finish training with good loss terms only to find that the Actor's policy is poor without the constraint enforcement in practice.

DPro+ aims to avoid this problem by formulating the Critic using the fully constrained action, and thus does not model the augmented closed-loop system. In other words, this formulation does not assign the value of the constrained outcome directly to the unconstrained action that produced it and thus avoids masking the value of many distinct raw actions with the same enforced outcome. As a result, the Actor loss can continue to use the raw policy output but still learn to treat safe and unsafe actions differently; however, this is under the assumption that the Critic learns to quantify the value of unsafe actions. This is the root of the new issue that arises from the Actor training loop. DPro+ can be interpreted as learning the value of safe current actions followed by a feasible continuation policy. This makes the Critic meaningful on the safe/feasible action region covered by the training process, but not on the entire action region because only safe or feasible actions ever get involved in the Critic loss formulation. The Actor training loop issue arises when the Actor is evaluated at actions outside this region, especially infeasible raw actions. This out-of-distribution data will cause the Critic to extrapolate from data that it was trained off of, which risks overestimation and reintroducing the lazy Actor problem of DPre+.

Ideally, there should not be such a distribution issue, and this can be fixed by making sure that the bootstrapped Critic term and the Actor loss action term are safe actions, which motivated the design of DProj+. Under further analysis, there are still issues that can arise

from this design. Notably, unlike DPro+ or DPre+, the process of ensuring that these action terms are safe involves applying the constraint enforcement within both the Critic and Actor training loops instead of applying it during the exploration phase outside of the training loops. This is a significant problem for designs that have complex constraint enforcement procedures because of the added computation cost. A major benefit of NNs is that training in simulations can be massively accelerated due to the simplicity of the networks along with various methods of optimizing with respect to this structure, and so these added constraint enforcement steps compound over each update to drastically slow down training iteration speed. The second problem is with regards to gradients. As discussed previously, there is no free lunch with constraint enforcement in a gradient-based algorithm. The Actor training loop passes gradients backwards through the entire Critic and then the entire Actor; thus, even external constraint enforcement is functionally treated as an internal layer for the Actor due to its impact on the gradients used to modify the Actor weights. The same issue is present if the feasible action is used instead of safe if the feasibility is naively enforced through hard clipping mechanisms; however, this can be safely worked around by using tanh activation in the output layer due to the fact that tanh is a smooth reparameterization that does not map several points to the same clipped bound and instead squeezes them into the open interval of $(-1, 1)$. Similarly, the Critic training loop involves two Critic forward passes; once with the current state action pair, and once with the future state and action pair. The current pair is from the Replay Buffer, so no gradients pass into the Actor, but the future action is based on the Actor's output, and so it suffers from the same internalized constraint enforcer problem. In the case of algorithms like TD3, the target Critics are involved in this future action portion, and so the gradients are detached because the goal is to not update the target networks during the Critic update. Thus, DProj+ improves consistency between the Actor objective and the Critic's training distribution, but it does so at the cost of projection-induced gradient degradation in the Actor update.

While DProj+ may seem to be the least prone to the distribution issues of the 3, the gradient and computational impacts are non-negligible. Of DPro+ and DPre+, it is difficult to quantify which of the two sets of issues will cause the least problems in a given application. As such, it is often left as a matter of design preference. Both DPre+ and DPro+ are well suited for problems with simple or easy-to-satisfy constraints. If the risk of the Actor producing out-of-distribution actions is minimal, DPro+ will be less likely to cause problems. Similarly, if the risk of the Actor becoming over-reliant on the quality of safe fallback actions is minimal, then DPre+ will be less likely to cause problems. These cases tend to overlap, and thus it is more beneficial to view DPre+ as the low-effort lightweight implementation and DPro+ as the implementation that has more directly interpretable results at the expense of more design effort and a notable computational impact during exploration.

4. Guaranteeing stability in gated neural network-based control

The following section details the closed-loop form of the proposed RL-based control architecture alongside a formal derivation of its stability properties.

4.1. Closed-loop implementation

We consider two closed-loop implementations of safe control systems involving neural networks denoted as systems B and C in Fig. 10. In both formulations, the closed-loop system is designed to operate with respect to two sources of control signals: the fallback controller and the safe RL-controller. The fallback controller is as defined in Section 2.3 where the relevant form of the constraint to be enforced is based on the formulation of the constraint as would be used in LMPC such as Eq. (6e) or Eq. (7). As noted previously, the safe RL-controller can take two forms, depicted in systems B and C, respectively. In system B, the safe RL-controller is safe through external enforcement where the proposed control action from the NN is compared to the fallback controllers action in the form of the stability constraint. If violated, the fallback is used, and if safe, the proposed control action is used. By contrast, system C's safe RL-controller is safe through internal enforcement. To help clarify which type is being referred to, system C's design will be referred to as the Constraint Enforcing Neural Network Controller (CENNC). Safe NNCs can be considered the broader definition that includes hybrid designs while CENNCs are a particular case that is a strict constraint enforcer and NNC in one NN. To achieve this, the design involves 2 augmentations to the underlying NNC: the internalized fallback controller section and the internalized constraint enforcer section. In order to check the constraint satisfaction and fallback if needed, additional information from the fallback controller is needed. We can consider the two LMPC constraint forms provided from Section 2.4. For Eq. (6e), recall that the class of systems in this work are defined by Eq. (1). Further note that for a given forward pass of the NN, the state parameter is held constant. Thus, when we consider the reformulation of the constraint as a difference in $\dot{V}$, the equation simplifies to exclude lone parameters based on x alone.

$$\dot{V}\left(\tilde{x}\left(t_k\right), u\left(t_k\right)\right) - \dot{V}\left(\tilde{x}\left(t_k\right), \boldsymbol{\Phi}\left(\tilde{x}\left(t_k\right)\right)\right) \leq 0 \tag{163}$$

$$\left(L_f V\left(\tilde{x}\left(t_k\right)\right) + L_g V\left(\tilde{x}\left(t_k\right)\right)^{\top} u\left(t_k\right)\right)$$

$$-\left(L_f V\left(\tilde{x}\left(t_k\right)\right) + L_g V\left(\tilde{x}\left(t_k\right)\right)^{\top} \boldsymbol{\Phi}\left(\tilde{x}\left(t_k\right)\right)\right) \leq 0 \tag{164}$$

$$L_g V\left(\tilde{x}\left(t_k\right)\right)^{\top}\left(u\left(t_k\right) - \boldsymbol{\Phi}\left(\tilde{x}\left(t_k\right)\right)\right) \leq 0 \tag{165}$$

Here, $L_f V$ and $L_g V$ are the Lie derivatives of the Lyapunov function. With this simplified formulation, we can see that besides u, which would be the proposed action from the NN, all that is needed is $L_g V$ and reference control action, both of which are calculated at the current state. For Eq. (7):

$$\dot{V}\left(\tilde{x}\left(t_k\right), u\left(t_k\right)\right) \leq -\alpha V\left(\tilde{x}\left(t_k\right)\right) \tag{166}$$

$$\dot{V}\left(\tilde{x}\left(t_k\right), u\left(t_k\right)\right) + \alpha V\left(\tilde{x}\left(t_k\right)\right) \leq 0 \tag{167}$$

$$L_f V\left(\tilde{x}\left(t_k\right)\right) + L_g V\left(\tilde{x}\left(t_k\right)\right)^{\top} u\left(t_k\right) + \alpha V\left(\tilde{x}\left(t_k\right)\right) \leq 0 \tag{168}$$

Here, although the equation does not simplify much, the end result is still an inequality that is purely composed of functions of the state. Since the state is already an input to the NN, and none of these terms involve learnable parameters, the Lie derivatives corresponding nonlinear function and the fallback controllers functions can safely be internally implemented. In particular, much like how LMPC requires a model of the system dynamics to calculate the evolution of the state based on the current proposed actions, the CENNC formulation requires information derived from a first-principles model to calculate the corresponding Lie derivatives. Alternatively, these terms can be calculated externally and passed as auxiliary inputs alongside the state. Both forms of this, and any mix of the two, can be used as the internalized fallback controller section depicted in red in Fig. 10.

To handle the internal constraint enforcement, any hard constraint enforcement method from Section 3.3.3 or 3.3.4 would be sufficient. For the form of the constraint considered in this work, the STE-Gate based approach from Section 3.3.4 is best suited as it is both simple to design and has potentially favorable gradient impacts. We once again consider the two forms of the constraint from the LMPC formulation. For both cases the only remaining step is to re-frame the constraint as an if-else condition:

$$\text{if } z > 0 \quad \text{then} \quad u = u_{NN}\left(t_k\right) \quad \text{else} \quad u = \boldsymbol{\Phi}\left(\tilde{x}\left(t_k\right)\right)$$

where z denotes the two Lyapunov cases and thus is either $-L_g V\left(\tilde{x}\left(t_k\right)\right)^{\top}\left(u_{NN}\left(t_k\right) - \boldsymbol{\Phi}\left(\tilde{x}\left(t_k\right)\right)\right)$ or $-\left[L_f V\left(\tilde{x}\left(t_k\right)\right) + L_g V\left(\tilde{x}\left(t_k\right)\right)^{\top} u\left(t_k\right) + \alpha V\left(\tilde{x}\left(t_k\right)\right)\right]$ for the LMPC constraint forms. Recall that ReLU and the Heaviside step function both yield zero for negative inputs.

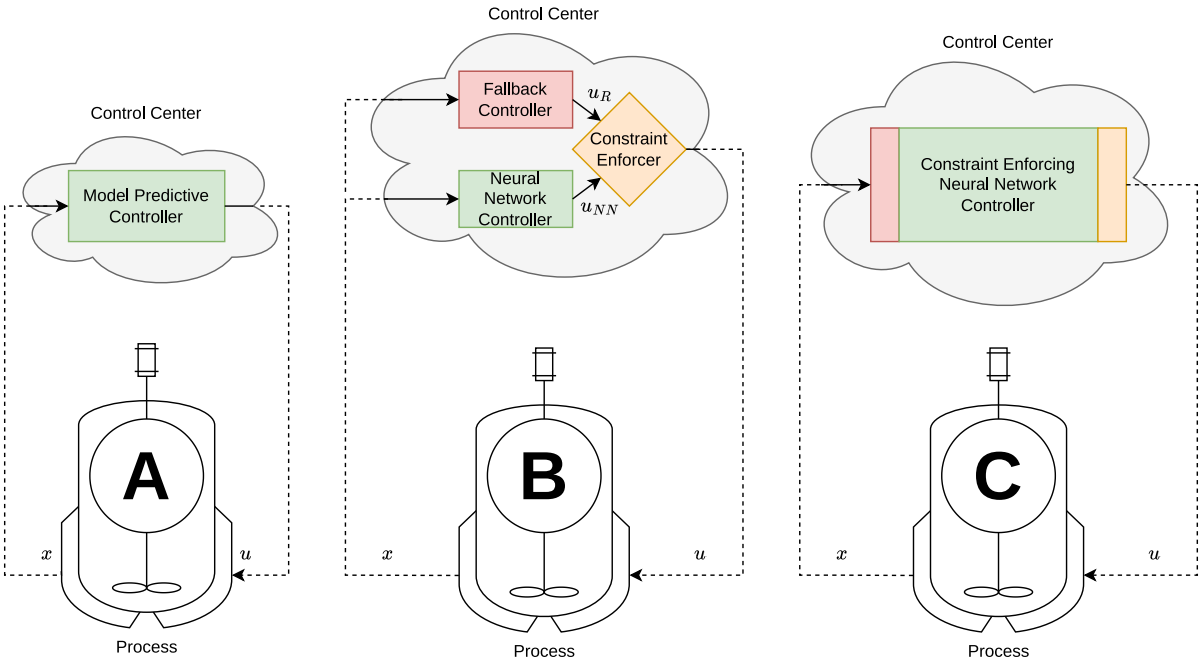

Fig. 10. Generalized safe process control block diagrams for Model Predictive Control (A), Reinforcement Learning-based control (B), and Constraint Enforcing Reinforcement Learning-based control (C). The fallback control for the Model Predictive Controller block is assumed to be integrated as a part of the MPC itself. Constraint Enforcing Neural Network Controller utilizes the fallback control as input and integrates the constraint enforcer internally as depicted by the shaded sections of the block. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Thus, we can consider the negative of the expression and collapse the if-else formulation into a Heaviside step function formulation. In particular, it is convenient to formulate these gates in an alpha-gate manner where we define α as the switch parameter defined by:

$$\alpha = \text{Heaviside}\,(z) \tag{169}$$

where the Heaviside operation can either treat the zero case as one for strict adherence to the constraint or zero for a more conservative approach. The resulting alpha-gate is thus:

$$u = \alpha u_{NN}\left(t_k\right) + (1-\alpha)\,\boldsymbol{\Phi}\left(\tilde{x}\left(t_k\right)\right) \tag{170}$$

or alternatively

$$u = \boldsymbol{\Phi}\left(\tilde{x}\left(t_k\right)\right) + \alpha\left(u_{NN}\left(t_k\right) - \boldsymbol{\Phi}\left(\tilde{x}\left(t_k\right)\right)\right) \tag{171}$$

Finally, the STE must be designed, where

```
alpha_hard = torch.heaviside(z, torch.zeros_like(z))
alpha_soft = torch.tanh(F.softplus(z, beta=beta))
alpha = alpha_soft + (alpha_hard - alpha_soft).detach()

u_current = alpha * u_nn + (1.0 - alpha) * u_current
```

In this code, the definition of α involves a detachment operation which is the code form of the STE where the forward pass uses all 3 terms and thus uses the hard form whereas the backwards pass uses only the non-detached terms, and is thus the soft form. Overall, this alpha-gate approach functions as a strict internal enforcer of the Lyapunov constraint. There remains the issue of the actuator bounds; however, the fact that the constraint violation results in a replacement of the action with a reference that is guaranteed to satisfy the constraints allows for simpler handling. Instead, the proposed control action needs to be derived from a layer in the NN that uses `tanh` activation. This ensures that if the proposed action satisfies the constraint, it already inherently satisfies the bounds as well. The alpha-gate approach or any other similar strict internal enforcement approach are represented in Fig. 10 as system C.

Remark 5. Prior works using the form of system B from Fig. 10 utilized a short-horizon form of the LMPC that was feasible for real-time application to the process in contrast to the long-horizon form which took too long to solve and exceeded the sampling time of the system. The CENNC design in system C is still compatible with such a framework, but because the LMPC operates based solely on the state as input, including the LMPC internally is functionally equivalent to solving the LMPC externally and injecting its output into the NN wherever needed.

Remark 6. While the CENNC design can operate individually with strict constraint satisfaction, it can be beneficial to use the external constraint enforcer approach of system B from Fig. 10 as a performance condition instead of a stability condition. One can see that if the fallback controller is involved in the forward pass of the CENNC that using a high-quality fallback controller can slow down training. Thus, the CENNC can be constructed with respect to a fallback controller that continue to satisfy the constraints but does so in a lightweight manner at the expense of control quality. The resulting CENNC will continue to satisfy the constraints, but will suffer from a worse baseline performance in cases where internal fallback occurs, in which case the external constraint enforcer can be used to switch the control action to a better performing reference controller.

Remark 7. The novelty of the framework depicted in Fig. 10 as system C is not necessarily the strict internal constraint enforcement, as that has been explored in several works that involve differentiable QP-layers. Instead, the novelty is mainly in the composition for Lyapunov-based stability guarantees through the alpha-gate approach. This approach is built in a manner that is both simple to design and tunable as needed. In doing so, the framework provides a method that achieves strict internal constraint satisfaction while also being lightweight, easy to work with, interpretable, and allows for constraint informed gradient signals to exist in a manner that can be tuned to strengthen or weaken specific near-boundary cases as needed. Additionally, the alpha-gate approach to the depicted framework, due to its reliance on

the Heaviside step function for strictness, is capable of strictly enforcing any scalar inequality or equality constraint, which makes it applicable to generalized continuous optimization problems whereas alternative approaches such as QP-layers are limited to linear constraints.

4.2. Closed-loop stability

We now explore the closed-loop systems stability properties under the proposed controllers. In the continuous-time case, Section 2.3 demonstrates how exponential stability of the origin is assured by the reference controller; however, the closed-loop design cannot be built under the assumption of continuous-time control. In real-world systems, signals need time to transmit, measurements need time to complete, and computation times for even the lightest of controllers are non-zero. As such, the process evolves as a continuous-time system, but must be controlled through discrete-time control via the sample-and-hold control implementation. In this implementation, actions from the controllers are applied in fixed intervals and the actions are held constant for the entire interval $t \in [t_k, t_k + \Delta)$. This has consequences on the stability guarantees which leads to the system instead converging to a small region around the origin. In summary, it can be seen that under the assumptions from Section 2.3 (and by further assuming that the sample-and-hold implementation is done with sufficiently small sampling time) that a known stabilizing reference controller, sufficiently smooth Lyapunov function that satisfies Eq. (4), and state feedback are sufficient for solving and applying the necessary control actions that guarantee boundedness inside the stability region and convergence to a small region around the origin for the closed-loop system state.

4.2.1. Closed-loop stability under LMPC

Theorem 1. *Consider an LMPC as is defined in Eq. (6) whose stability constraint is an element in the set of valid stability constraints:*

$$S(x) \in \left\{\dot{V}(x, \Phi(x)), -\alpha V(x)\right\} \tag{172}$$

where the systems dynamics are enforced through Eq. (6b) and modeled by Eq. (1). Further consider the first control input vector that is extracted from the full LMPC's control trajectory, $u(t)$. If the stabilizability assumption, as defined in Section 2.3, is applied to the given system, then, for any $x(t_0) \in \Omega_\rho$, there must exist a pair of positive constants $\epsilon_w, \alpha > 0$ such that the closed-loop state is driven to a small region around the origin (denoted $\Omega_{\rho_{\min}}$) under the sample-and-hold implementation of $u(t)$ if the conditions below hold for any sampling instant t_k:

$$L_x' M_F \Delta - \alpha \rho_s \leq -\epsilon_w \tag{173a}$$

$$\rho_{\min} := \max\left\{V\left(x\left(t_k + \Delta\right) | V\left(x\left(t_k\right)\right) \leq \rho_s\right)\right\} \tag{173b}$$

$$\rho_s < V\left(x\left(t_k\right)\right) \tag{173c}$$

$$\rho_s < \rho_{\min} < \rho \tag{173d}$$

Proof. Starting with the definition of the time-derivative for the Lyapunov function taken for any time within the current sampling-interval $t \in [t_k, t_k + \Delta)$:

$$\dot{V}\left(x(t), u\left(t_k\right)\right) = \frac{\partial V(x(t))}{\partial x} F\left(x(t), u\left(t_k\right)\right) \tag{174}$$

we manipulate the expression algebraically through addition and subtraction of a constant. Namely, we add and subtract the time-derivative of the Lyapunov function taken at the first time-instant in the current sampling interval:

$$\dot{V}\left(x(t), u\left(t_k\right)\right) = \frac{\partial V(x(t))}{\partial x} F\left(x(t), u\left(t_k\right)\right) \tag{175}$$

$$- \frac{\partial V\left(x\left(t_k\right)\right)}{\partial x} F\left(x\left(t_k\right), u\left(t_k\right)\right) \tag{176}$$

$$+ \frac{\partial V\left(x\left(t_k\right)\right)}{\partial x} F\left(x\left(t_k\right), u\left(t_k\right)\right) \tag{177}$$

This expression can be simplified through substitution of expressions derived from the stabilizability assumption. Namely, Eq. (5c) combines the first two terms into one:

$$\dot{V}\left(x(t), u\left(t_k\right)\right) \leq L_x' \left|x(t) - x\left(t_k\right)\right| + \frac{\partial V\left(x\left(t_k\right)\right)}{\partial x} F\left(x\left(t_k\right), u\left(t_k\right)\right) \tag{178}$$

This expression further simplifies via the integral triangle inequality:

$$\left|x(t) - x\left(t_k\right)\right| \leq \int_{t_k}^{t} \left|F\left(x(\tau), u\left(t_k\right)\right)\right| \mathrm{d}\tau \leq M_F \Delta \tag{179}$$

which yields:

$$\dot{V}\left(x(t), u\left(t_k\right)\right) \leq L_x' M_F \Delta + \frac{\partial V\left(x\left(t_k\right)\right)}{\partial x} F\left(x\left(t_k\right), u\left(t_k\right)\right) \tag{180}$$

The remaining derivative term simplifies depending on the form of the stability constraint used in the LMPC formulation; however, both cases presented collapse to the same form. In the case where $S(x) = \dot{V}(x, \Phi(x))$, Eq. (4b) simplifies the term to $-c_3 |x|^2$. This term further simplifies through additional algebraic manipulation of Eq. (4a):

$$V(x) \leq c_2 |x|^2 \tag{181}$$

$$-c_2 |x|^2 \leq -V(x) \tag{182}$$

$$-c_3 |x|^2 \leq -\frac{c_3}{c_2} V(x) = -\alpha V(x) \tag{183}$$

In the case where $S(x) = -\alpha V(x)$, it is seen that both expressions take the same form. Finally, note how $V(x)$ is a term that scales proportionally to $|x|$. It can be seen that, as $|x| \to 0$, the impact of this term becomes 0, and thus the upper-bound of the time derivative across any arbitrary time for one sampling-period is not negative, hence exponential stability is no longer valid. Instead, consider two smaller level sets of ρ defined in Eq. (173d) where ρ_s and $\rho_{\min}$ are related through Eq. (173b) such that $\rho_{\min}$ is the level set that captures all single sampling-period deviations from states starting in ρ_s. Thus, we consider ρ_s to be the level-set in which the upper-bound of the time-derivative of the Lyapunov function is non-negative. In the context of this proof, for any sampling instant t_k, we consider points that lie outside of this region (i.e., when Eq. (173c) holds true). Thus, the final upper-bound is defined as:

$$\dot{V}\left(x(t), u\left(t_k\right)\right) \leq L_x' M_F \Delta - \alpha \rho_s \tag{184}$$

In order for this upper-bound to be negative, the sampling-period must be sufficiently small, which leads to Eq. (173a). Thus, with sufficiently small Δ, any closed-loop state $x(t_k) \in \Omega_\rho \backslash \Omega_{\rho_s}$ will decay over time towards Ω_{ρ_s} but ultimately converge to $\Omega_{\rho_{\min}}$. □

Remark 8. When $S(x) = \dot{V}(x, \Phi(x))$, the effective $\alpha = \frac{c_3}{c_2}$; however, when $S(x) = -\alpha V(x)$, α can be any positive constant. Thus, in the latter case, $\tilde{\alpha} = \alpha \frac{c_2}{c_3}$ can be interpreted as either an effective decay rate or as a relative strength weight that allows the user to loosen or tighten the stability strength requirements relative to the reference controller. Both extremes have caveats. If α is dialed too low, then the resulting ρ_s and $\rho_{\min}$ may no longer satisfy Eq. (173b) for any point within Ω_ρ because the necessary size of these regions will exceed the size encompassed by ρ. On the other extreme, if α is dialed too high, then the required control action may exceed the actuator bounds. Additionally, since the Lyapunov function properties are satisfied under the bounded reference controller, there is no guarantee that the Lyapunov function will abide by the necessary properties under these unbounded actions (assuming they are feasible to apply in the first place).

Remark 9. The proof as presented is done with respect to any point in time within a single sampling interval. As such, the same proof would apply for every sampling interval along the horizon that LMPC works with; however, we use the receding horizon formulation of LMPC in

Eq. (6). In such a formulation, the stability constraint is not applied beyond the initial time-instant. In other words, the stability properties only hold for the first sampling interval, not the entire horizon. This is practically fine because only the first control action from the resulting control trajectory is used before resolving the LMPC problem, but it is worth noting that all sampling intervals beyond the first are inherently suboptimal for the constrained optimal control problem due to their lack of stability constraint consideration. As a result, this relaxing of constraints makes longer horizon LMPC solutions deviate from the truly optimal trajectory.

4.2.2. Closed-loop stability under CENNC

Because NNs do not inherently satisfy constraints, the only way to guarantee that such unpredictable outputs consistently satisfy the stability constraints would be to in some form modify or replace them. In our prior works, this was done through an external constraint enforcer which operated as a hierarchical safety filter that directly replaces proposals that fail the constraint with the reference control action and selects with priority from NN first and MPC second (if used) (Khodaverdian et al., 2025b, 2026, 2025a). The stability proof of such a control system follows a case-by-case analysis to demonstrate how all cases collapse to some controller that satisfies the constraint. Because our proposed framework allows for the use of a soft constraint enforcer in the alpha-gate, such cases would fall under the category of unpredictable outputs, and thus we defer to our prior proofs for their stability guarantees under external constraint enforcement, which is now necessary. For the remaining strict case, such as the Heaviside formulation, external enforcement is no longer necessary.

Theorem 2. *Assume the existence of a reference controller ($\Phi(x)$) and Lyapunov function ($V(x)$) as described in Section 2.3. Consider any neural network controller whose outputs are elements of the admissible input set U as defined in Eq. (2). For such a controller, consider the implementation of an additional layer in the form of a final strict alpha-gate:*

$$u^* = \alpha u_{NN} + (1-\alpha)\, u_R \tag{185}$$

Here, u_{NN} is the prior proposed output, u^* is the new output, u_R is the reference control action, and α is a gating term defined by $\alpha = Heaviside(z)$ where $z = -\left(\dot{V}\left(x, u_{NN}\right) - S(x)\right)$ and $S(x)$ is an element in the set of valid stability constraints:

$$S(x) \in \left\{\dot{V}\left(x, u_R\right), -\beta V(x)\right\} \tag{186}$$

For this resulting CENNC, if the information needed to solve z is made available to the alpha-gate, then there must exist a pair of positive constants $\epsilon_w, \beta > 0$ such that the closed-loop state is driven to a small region around the origin (denoted $\Omega_{\rho_{\min}}$) under the sample-and-hold implementation of $u^*(t)$ if the conditions in Eqs. (173a) to (173d) hold for any sampling instant t_k.

Proof. Because the arbitrary NN must yield outputs that are elements of U, we start by considering an arbitrary neural network whose output layer is activated by `tanh`. Through an affine transformation, this output (y) can be mapped to any action within the actuator bounds, thereby ensuring that box constraints are pre-satisfied and thus the output is an element of U.

$$u_{NN} = u_{\min} + \frac{u_{\max} - u_{\min}}{2}(y+1) \tag{187}$$

We denote the difference between this proposed action and u_R as δu:

$$\delta u = u_{NN} - u_R \tag{188}$$

This formulation is used in the case where $S(x) = \dot{V}\left(x, u_R\right)$ such that:

$$z = -\left(\dot{V}\left(x, u_{NN}\right) - \dot{V}\left(x, u_R\right)\right) \tag{189}$$

$$z = -\left(\left(L_f V(x) + L_g V(x)^\top u_{NN}\right) - \left(L_f V(x) + L_g V(x)^\top u_R\right)\right) \tag{190}$$

$$z = -L_g V(x)^\top \left(u_{NN} - u_R\right) \tag{191}$$

This formulation would thus require, at minimum, $L_g V(x)$ and u_R to be provided to the alpha-gate in order for z to be solved. If instead $S = -\beta V(x)$, the resulting z is solved by:

$$z = -\left(\dot{V}\left(x, u_{NN}\right) + \beta V(x)\right) \tag{192}$$

$$z = -\left(L_f V(x) + L_g V(x)^\top u_{NN} + \beta V(x)\right) \tag{193}$$

which requires u_R, $V(x)$, and either $L_f V(x)$ and $L_g V(x)$ or a means to calculate $\dot{V}\left(x, u_{NN}\right)$ explicitly. With this information, z can be solved internally.

In the case where $z > 0$, $\alpha = 1$, which indicates that the stability constraint is satisfied by the proposed action. This is because the form of z is a reordered form of the stability constraint.

$$z > 0 \tag{194}$$

$$-\left(\dot{V}\left(x, u_{NN}\right) - S(x)\right) > 0 \tag{195}$$

$$\dot{V}\left(x, u_{NN}\right) - S(x) < 0 \tag{196}$$

$$\dot{V}\left(x, u_{NN}\right) < S(x) \tag{197}$$

Because this case corresponds to the satisfaction of the stability constraint, we want $u^* = u_{NN}$, which turns out to be true:

$$u^* = \alpha u_{NN} + (1-\alpha)\, u_R \tag{198}$$

$$u^* = 1 u_{NN} + (1-1)\, u_R \tag{199}$$

$$u^* = u_{NN} \tag{200}$$

Thus, the resulting action is one that satisfies the stability constraint, and therefore the stability proof follows as in Eq. (173). If instead $z \leq 0$, then $\alpha = 0$, which indicates that the stability constraint is violated by the proposed action. Once again, this is seen by reordering z:

$$z \leq 0 \tag{201}$$

$$-\left(\dot{V}\left(x, u_{NN}\right) - S(x)\right) \leq 0 \tag{202}$$

$$\dot{V}\left(x, u_{NN}\right) - S(x) \geq 0 \tag{203}$$

$$\dot{V}\left(x, u_{NN}\right) \geq S(x) \tag{204}$$

Because this case corresponds to the violation of the stability constraint, we want $u^* = u_R$, which turns out to be true:

$$u^* = \alpha u_{NN} + (1-\alpha)\, u_R \tag{205}$$

$$u^* = 0 u_{NN} + (1-0)\, u_R \tag{206}$$

$$u^* = u_R \tag{207}$$

Thus, the resulting action is one that satisfies the stability constraint, and therefore the stability proof follows as in Eq. (173). □

Remark 10. In Eq. (7), α is used to denote the constant for this form of the Lyapunov-based stability constraint. In Theorem 2, this constant is denoted by β for clarity as α is used to refer to the Heaviside value from the alpha-gate.

Remark 11. In the case where $z = 0$, the choice of which controller is needed becomes irrelevant as both are equally stabilizing. By defining the Heaviside function to use 0 at 0, we make the internal constraint enforcement negligibly more conservative than the equivalent LMPC's stability constraint formulation. A truly equivalent representation would define Heaviside as 1 at 0.

Remark 12. The reference controller used in Theorem 2 can take the form of an LMPC, assuming it is solvable within the time-frame, because Theorem 1 demonstrates that the LMPC is at least as stabilizing as the reference controller it is defined with respect to. As such, the CENNC can use either controller as its reference, where the practical

consequence will be in terms of the scale of the δu component that is present in the gradient as shown in Eq. (158).

Remark 13. If the soft form of the alpha-gate is used, the resulting α is no longer strictly 1 or 0, and thus u^* becomes a weighted sum of the two control actions. Since both actions satisfy the box constraint, the resulting action will too; however, the satisfaction of the Lyapunov constraint is not guaranteed. Thus, this form of the internal constraint enforcement would still require external constraint enforcement.

5. Application to a chemical process example

This section explores the implementation of the CENNC framework to a benchmark chemical process example. The goal is to demonstrate the viability of the framework in safety-critical processes. Additionally, a set of additional controllers (LMPC, an example IL-based controller, a non-CENNC RL-based controller, and a QP-layer based CENNC) are designed as a point of comparison for various aspects of the framework.

5.1. Process description

The benchmark chemical process example is a singular continuous stirred tank reactor (CSTR) example. The CSTR is assumed to be perfectly mixed while operating non-isothermally due to the presence of a singular exothermic irreversible reaction. This reaction is an arbitrary liquid-phase reaction ($A \rightarrow B$) with second-order dynamics. The reactor is controlled through direct manipulation of the feed concentration (C_{A0}) and through direct heat addition or removal at a rate $\dot{Q}$. This system is modeled through the following dynamics:

$$\frac{\mathrm{d}C_A}{\mathrm{d}t} = \frac{F}{V_L}\left(C_{A0} - C_A\right) - kC_A^2 \tag{208}$$

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{F}{V_L}\left(T_0 - T\right) - \frac{\Delta H}{\rho_L C_p} kC_A^2 + \frac{\dot{Q}}{\rho_L C_p V_L} \tag{209}$$

where $k = k_0 \exp\left[-\frac{E}{RT}\right]$ and k_0 denotes the isothermal rate-constant. All other terms in Eq. (208) that have a 0 in their subscript denote feed values for the respective term. C_A, T, ρ_L, C_p, ΔH, E, and V_L denote the concentration of species A, temperature, density, specific heat, heat of reaction, activation energy and liquid volume, all for the bulk fluid phase, respectively.

5.2. Control problem

The state variables are chosen to be C_A and T whereas the control variables are chosen to be C_{A0} and $\dot{Q}$, yielding:

$$x^\top = \left[C_A, T\right] \tag{210}$$

$$u^\top = \left[C_{A0}, \dot{Q}\right] \tag{211}$$

which define the absolute state and control vectors, respectively. Because the Lyapunov function is defined with respect to the origin, we redefine the state and control variables in deviation variable form without loss of generality.

$$x^\top = \left[C_A - C_{As}, T - T_s\right] \tag{212}$$

$$u^\top = \left[C_{A0} - C_{A0_s}, \dot{Q} - \dot{Q}_s\right] \tag{213}$$

Here, the values of the steady state parameters are derived from an open-loop stable steady state and can be found in Table 2 along with other constants used in the CSTR model. Additionally, the control input is bounded such that:

$$-3.5 \leq C_{A0} - C_{A0s} \leq 3.5\,\mathrm{kmol\,m^{-3}} \tag{214}$$

$$-5 \times 10^5 \leq \dot{Q} - \dot{Q}_s \leq 5 \times 10^5\,\mathrm{kJ\,h^{-1}} \tag{215}$$

Table 2
Parameter values of the CSTR model.

Var.	Value	Var.	Value
C_{As}	$1.954\,\mathrm{kmol\,m^{-3}}$	C_{A0s}	$4.0\,\mathrm{kmol\,m^{-3}}$
C_p	$0.231\,\mathrm{kJ\,kg^{-1}\,K^{-1}}$	ΔH	$-11{,}500\,\mathrm{kJ\,kmol^{-1}}$
E	$5.0 \times 10^4\,\mathrm{kJ\,kmol^{-1}}$	F	$5\,\mathrm{m^3\,h^{-1}}$
k_0	$8.46 \times 10^6\,\mathrm{m^3\,kmol^{-1}\,h^{-1}}$	$\dot{Q}_s$	$0.0\,\mathrm{kJ\,h^{-1}}$
R	$8.314\,\mathrm{kJ\,kmol^{-1}\,K^{-1}}$	ρ_L	$1.0 \times 10^3\,\mathrm{kg\,m^{-3}}$
T_0	300.0 K	T_{0s}	300.0 K
T_s	401.9 K	V_L	$1\,\mathrm{m^3}$

For this system, a reference controller that satisfies the stabilizability assumption as defined in Section 2.3 is found to be two Proportional (P) controllers with a weight vector of $[2.0, 5000]^\top$. The corresponding Lyapunov function is found to be of the quadratic form $V = x^\top P x$ with $P = \begin{bmatrix} 1{,}060 & 22 \\ 22 & 0.52 \end{bmatrix}$. The design goal for this process example is to create a controller that will drive the closed-loop system from any initial state in Ω_ρ where $\rho = 120$ to a small region around the origin. For this system, we can conservatively estimate this small region to be contained within the region where $V \leq 1$. For all controllers, the sampling time of the system is taken to be $\Delta = 7.2\,\mathrm{s}$.

5.3. LMPC design

The LMPC design follows the form shown in Eq. (6) with a horizon length of $N = 60$ corresponding to a 7.2 min trajectory over which the states are simulated and the actions are optimized. The cost function used for LMPC in this system is of the form:

$$L(x, u) = x'^\top Q_x x' + u^\top Q_u u \tag{216}$$

where $Q_x = \mathrm{diag}\,[1{,}000 \quad 1]$, $Q_u = \mathrm{diag}\left[10 \quad 1 \times 10^{-8}\right]$, and x' is a shorthand denotation for the state after the application of u to the current x. This form of the cost function is designed due to the need for a discrete approximation of the continuous-time framework shown in Eq. (6). In such a framework, where the integral is replaced with a sum series, the inclusion of the initial state is irrelevant to the control objective as the measured initial state cannot be changed. As such, the summation is shifted to be with respect to the future sampling instance for the state so that it captures the impact of the current control action on the state. This cost function is utilized in the design of all RL-based controllers and all LMPCs that aim to satisfy the design objective.

Because of the long horizon length, the LMPC formulation is implemented using direct multiple shooting instead of single shooting. The motivation behind this choice is that the sensitivity of the trajectory to earlier actions leads to difficulties in single shooting approaches for long horizon tasks that multiple shooting suffers less from Betts (1998). Multiple shooting additionally controls the state variables during optimization but uses equality constraints based on the dynamics function to ground them to the numerical integration logic seen in single shooting methods. In particular, this was implemented using the CasADi Python package via the Opti stack (Andersson et al., 2019). Numerical integration was done via the RK4 method with an integration step size of $h_c = 0.36\,\mathrm{s}$. The nonlinear solver used in LMPC is IPOPT with default settings and warm-starting.

Remark 14. The design of the reference controller can be any such controller that satisfies conditions discussed in Section 2.3 and thus is guaranteed to satisfy the stability constraint. As discussed in Section 4.2, the reference controller can itself be an LMPC whose fallback, referred to as the fail-safe controller, satisfies the stability constraints. While this is a viable option for non-CENNCs, it is not needed for CENNCs. We avoid using a lightweight LMPC, one with a shorter horizon length, because the need to solve such a problem repeatedly during training would impact the training time heavily. Instead, we use

the reference P controller, which has the added impact of strengthening the impact of the STE on the gradient as the δu term is typically larger as a consequence of the reference controller being suboptimal.

5.4. Imitation learning-based control design

For the sake of comparison, we design an IL-based control baseline design to demonstrate the relative impact of RL and the CENNC framework. In prior works, such a design has demonstrated notable improvements to computation time with good fitting on data that it was trained on Khodaverdian et al. (2025b, 2026), and so it functions as a viable framework for comparison. This IL-based controller is constructed as a Feedforward Neural Network (FNN) whose input is x and whose output is the `tanh` scaled form of u which is rescaled afterwards. Specifically, the network is a feedforward neural network with 3 hidden layers of width 32, 32, and 64, respectively, for a total of 4 layers that convert the dimensions $2 \to 32 \to 32 \to 64 \to 2$. The network uses `Mish` activation with pre-activation `LayerNorm`. The weights and biases of the network use `float64` precision.

Data generation for the IL task uses the LMPC formulation above over a total of 20,000 trajectories. The LMPC is run in `float64` precision with fixed seeding using seed 42. Two datasets are generated, one where the initial state is randomly sampled between the level sets of $V = 1$ and $V = 50$ and the other generated with random initial states within $V = 1$ and $V = 120$. In both cases, the closed-loop system utilized LMPC for a maximum of 2,500 steps with early trajectory termination in the case where the state enters the region $V \leq 1$. The data generation yielded two datasets of 442,867 and 549,261 state–action pairs, respectively, containing 52 and 37 instances of controller fallback across 51 and 36 trajectories, respectively. It can be seen from Fig. 11 that the trajectory-based sampling process yields data that concentrates near the terminal region and is sparse near the boundary of the maximal level set defined for the dataset. By viewing the same type of plot in the action space as shown in Fig. 12, it can be seen that the actuator bounds on C_{A0} function as an inhibitor to the optimality of the resulting actions for the given objective. As such, it is expected that NN training will have a concentration of actions that saturate these bounds in an attempt to achieve the increased cost optimality. It is seen by Figs. 13 and 14 that neither the state nor action space contain sharp changes in the optimal action or average state, respectively, which is beneficial for NN training as jaggedness is difficult to accurately model through differentiable means. This behavior is consistent for the other action and state space plots for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$, and it is also consistent for the same plots for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

Training was done over 5,000 epochs with an initial learning rate of 1×10^{-4} using an 80/10/10 train/val/test split where data is randomly split from a concatenated 1D vector of all the recorded state–action pairs. The `CosineAnnealingWarmRestarts` scheduler is used to decay the learning rate after every training step to a minimum of 1×10^{-8} before resetting after 2,500 epochs. Mini-batching was used with a batch size of 1,000 where the dataset is first purged of any state–action pairs that involve fallback control and then subsequently shuffled randomly. An epoch is concluded after the full dataset is processed in batches of 1,000 with no data repetition. The loss function is the Huber loss between the FNNs output and the LMPC initial control action for a given state. The optimizer used for training is a hybrid approach utilizing `Muon` for weights and `AdamW` for biases. Due to the scale difference between C_A and T, T is divided by 20 prior to being fed to the NN as this brings both states to roughly the same magnitude without excessive standardization. Additionally, the loss function is calculated with respect to the scaled control action. In other words, the raw NN output is within $(-1, 1)$ due to the `tanh` activation in the output layer, and the corresponding datasets actions are scaled to the same range via a division by the actuator bounds which works cleanly due to their uniform design. As seen in Table 3, the dataset generation

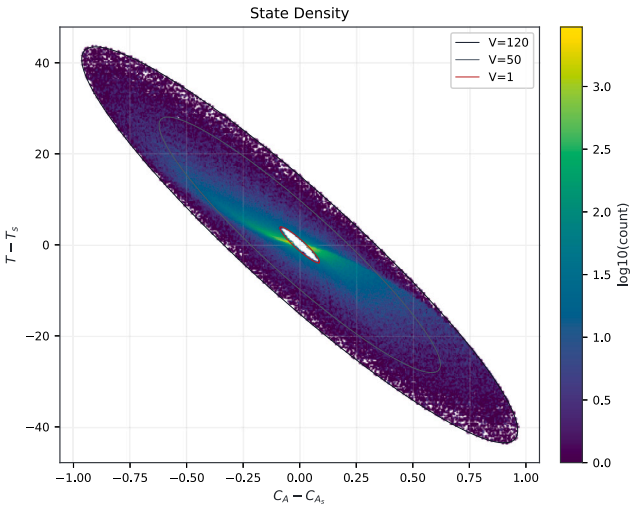

Fig. 11. 2D Histogram of the number of samples for a given pair of states for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$.

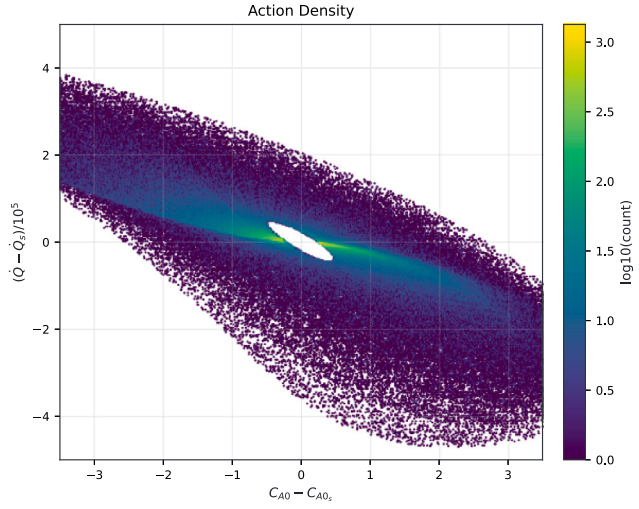

Fig. 12. 2D Histogram of the number of samples for a given pair of control actions for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$.

Table 3
Compute time for LMPC state–action pair dataset generation and IL training. Dataset generation CPU-hours are computed from the sum of per-trajectory solve times. Wall-clock time is the true generation time after parallelizing across the CPU cores of one ml.c7i.48xlarge instance using Amazon SageMaker Studio Code Editor. IL training was performed on a single RTX A2000.

Stage	Domain	Wall-clock time	Compute-hours
MPC dataset generation	$\Omega_{50} \setminus \Omega_1$	3h 17m 05s	582.1 CPU-h
MPC dataset generation	$\Omega_{120} \setminus \Omega_1$	3h 55m 44s	704.8 CPU-h
IL training	$\Omega_{50} \setminus \Omega_1$	3h 36m 47s	3.61 GPU-h
IL training	$\Omega_{120} \setminus \Omega_1$	4h 26m 54s	4.45 GPU-h

poses the largest computational burden in the training pipeline in the absence of a sufficiently powerful CPU. Even with a 192 core compute optimized system, the dataset generation took roughly 4 h to complete which is roughly equivalent to the time it took to train the model.

The trained model contains two checkpoints, the final checkpoint (for recovery or further training if desired) and the best performing checkpoint based on the evaluation set. The best checkpoints in both cases occurred prior to the learning-rate reset at epoch 2,500 instead

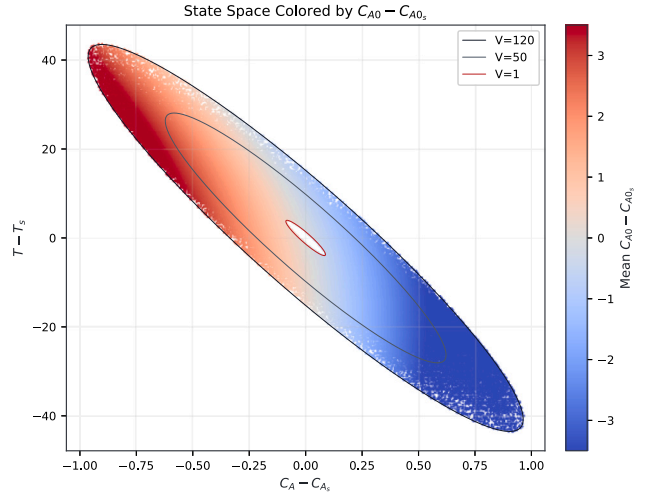

Fig. 13. 2D state space plot color-coded by the value of u_1 for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

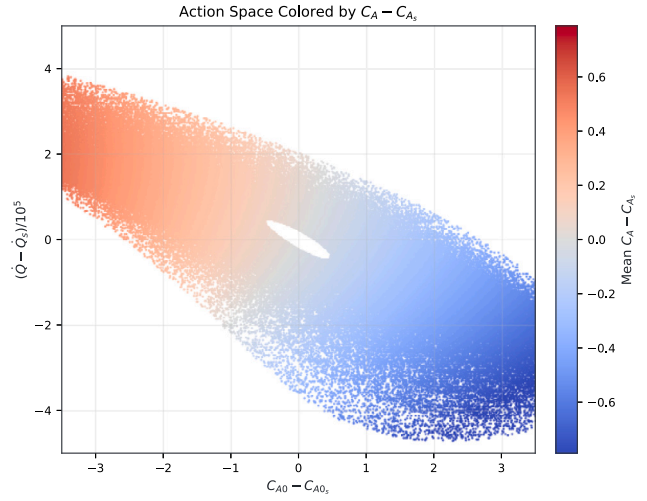

Fig. 14. 2D action space plot color-coded by the value of x_1 for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

occurring at epochs 2,285 and 2,318 for the $\Omega_{50} \setminus \Omega_1$ dataset and $\Omega_{120} \setminus \Omega_1$ dataset, respectively. These checkpoints yielded a validation loss of 5.7809×10^{-8} and 1.6930×10^{-7}, respectively, with a test loss of 5.9858×10^{-8} and 1.7314×10^{-7}, respectively, indicating good fit and potentially good generalization. The loss curve for both datasets exhibits a rapid convergence to a low loss value as shown in Figs. 15 and 16. Note that in these figures, the test loss is defined with respect to the final checkpoint, which performs marginally worse than the best case. Even in the case of the final checkpoint, the test, validation and train losses are all within the same order of magnitude, indicating that overfitting is likely not an issue. To further ensure that stable training occurred various additional metrics were monitored. Due to the existence of `LayerNorm` but lack of Normalize-and-Project techniques, there are possible concerns regarding trainable parameter scales. To monitor this, the gradient of all trainable parameters concatenated into a single vector is plotted per-epoch as shown in Figs. 17 and 18 which demonstrates that the gradient also exhibits rapid collapse to a small

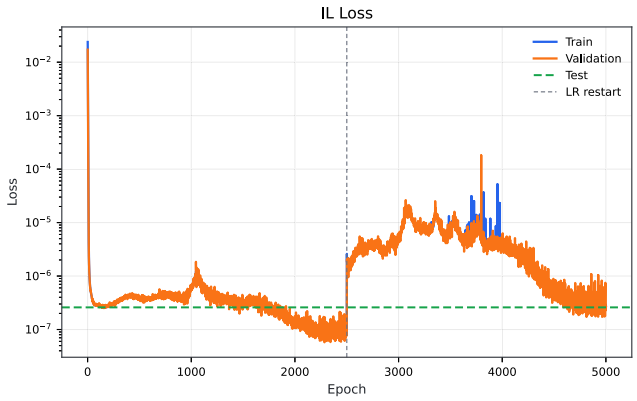

Fig. 15. Loss curve of the IL-based network trained from data generated by the LMPC for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

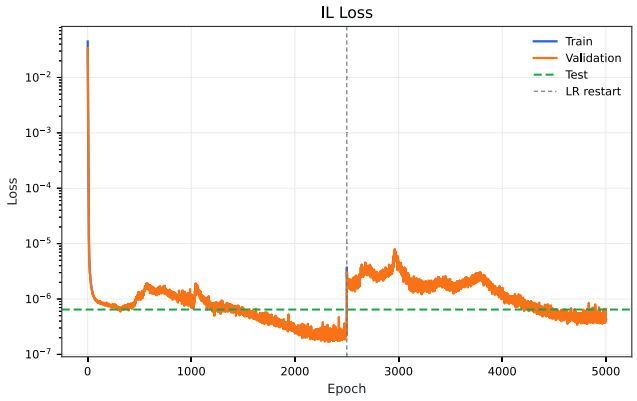

Fig. 16. Loss curve of the IL-based network trained from data generated by the LMPC for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$.

value. Although the gradient seemingly grows in magnitude as training progresses despite the decaying learning rate, it is seen by the plot of various metrics for the weights and bias terms shown in Figs. 19 and 20 that the networks weights and biases do not exhibit growth over the training run. In fact, the trends indicate that some parameters grow despite an overall decay in scale, indicating a trend towards sparsity patterns. This is verified through computing the Hoyer sparsity at the best performing epoch as 0.7977 and 0.8246 for the weights of the $\Omega_{50} \setminus \Omega_1$ dataset and $\Omega_{120} \setminus \Omega_1$ dataset, respectively, and 0.5094 and 0.6154 for the biases, respectively. Hoyer sparsity is a metric of sparsity based on the ratio of the L_1 and L_2 norms that ranges from 0 for vectors whose elements are all equivalent to 1 for a vector containing a singular non-zero element (Hoyer, 2004). This indicates that the weight matrices in particular exhibit this pattern of increasing sparsity that can be of use in further compressing the size of the network to improve computation time or decrease hardware requirements.

The final model is tested on the total database to observe its fit via a plot of the generated actions vs. the LMPC reference actions as shown in Figs. 21 and 22. The ideal case is for most points to lie near the $y = x$ diagonal, as this would indicate a closer fit. As observed, the model fits very well to its training data and exhibits few outlier points. During closed-loop simulations on the trained model, the IL-based controller will utilize the external constraint enforcer framework as shown in system B of Fig. 10.

To better understand how this model is expected to deviate from LMPC, we further consider plots that look at similar 2D histograms using bins based on the NN-controllers values. In particular, the dataset can be filtered into these new bins to allow for computation of a percent

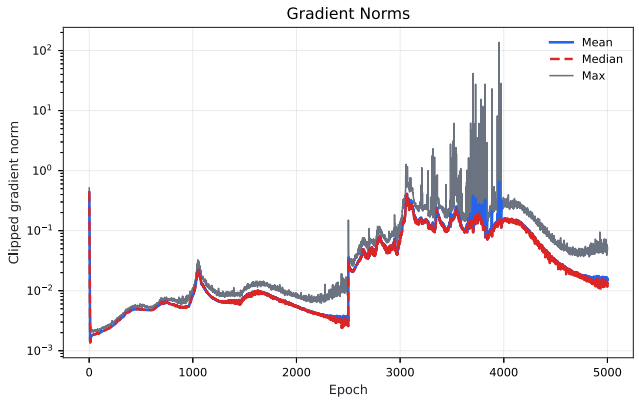

Fig. 17. Average gradients of the concatenated trainable parameter vector over the training run for the IL-based network trained from data generated by the LMPC for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

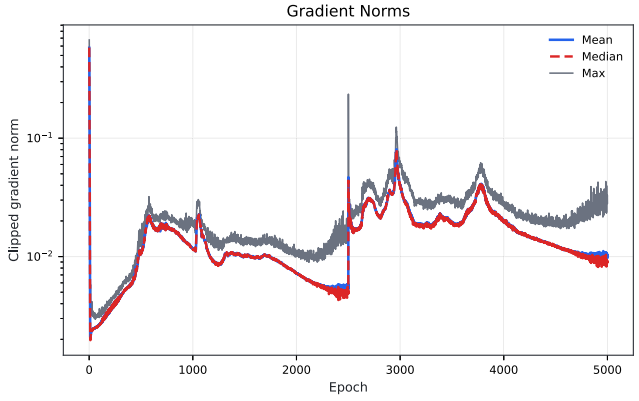

Fig. 18. Average gradients of the concatenated trainable parameter vector over the training run for the IL-based network trained from data generated by the LMPC for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$.

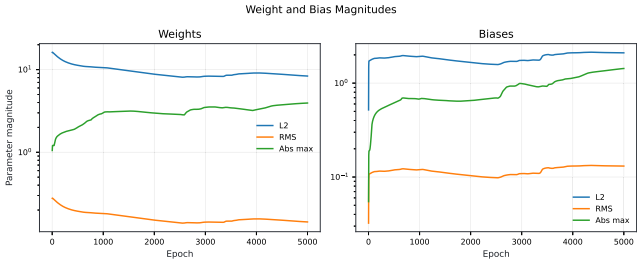

Fig. 19. Various normalized metrics of the concatenated weights and biases vectors over the training run for the IL-based network trained from data generated by the LMPC for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$. *L2* refers to the Euclidean norm, *RMS* refers to the root mean squared of the vector, and *Abs max* refers to the maximum magnitude element of the vector.

error that defines how metrics from the NN-controller vary from the LMPC in the state and action spaces. Figs. 23 and 24 demonstrate that for the majority of the action space, the LMPC and NN controller exhibit similar densities of states. Notably, the regions near the boundary of the state space elliptical level set for the datasets (which correspond to the boundaries of the action space where samples exist) demonstrate the most variance. This is expected, as the trajectory based sampling sparsely samples these regions and densely samples points near the origin. The validity of this intuition can be observed by contrasting the regions of variance with the densely samples regions from Fig. 12 to

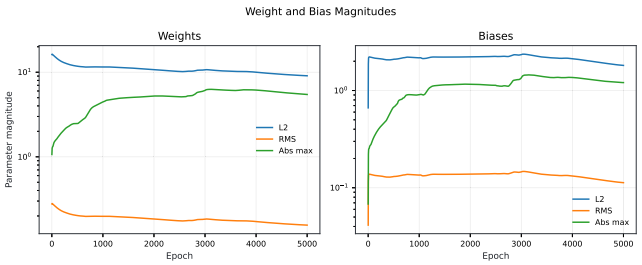

Fig. 20. Various normalized metrics of the concatenated weights and biases vectors over the training run for the IL-based network trained from data generated by the LMPC for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$. *L2* refers to the Euclidean norm, *RMS* refers to the root mean squared of the vector, and *Abs max* refers to the maximum magnitude element of the vector.

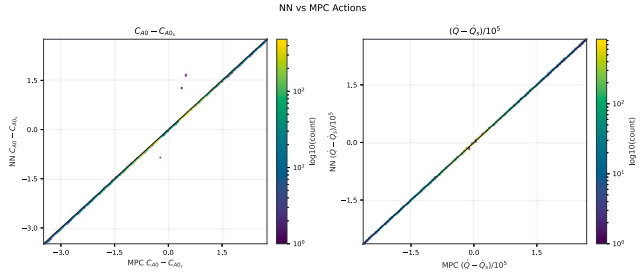

Fig. 21. 2D Histogram of the number of samples for a given pair of NN generated control actions and stored MPC control actions in the overall dataset for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

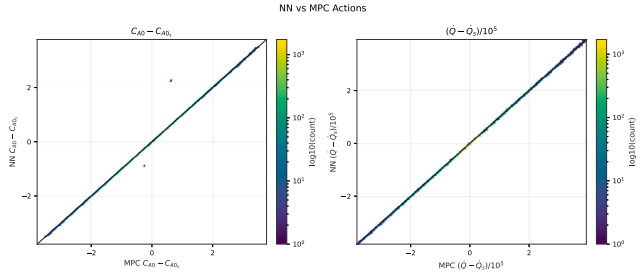

Fig. 22. 2D Histogram of the number of samples for a given pair of NN generated control actions and stored MPC control actions in the overall dataset for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$.

observe how the densely sampled regions lack this variance. Further, it is seen that the previously mentioned expected issue of actuator bound handling is observably true; the NN-controller is biased towards producing u_1 values at the actuator boundary. In terms of the accuracy of the network, loss curves show that overall the behavior is good; however, these curves lack information on which regions, if any, the network struggles on. Figs. 25 to 28 demonstrate that the NN-control has a tendency of poorly estimating the optimal action when either action is near 0. Figs. 25 and 26 in particular are worth considering as the variance is not uniformly under or over estimating the importance of the concentration term, but instead seems to form distinct regions in the action space that either underestimate or overestimate the state measurement. Additionally, the issue of boundary estimation is observed in these figures. Unlike Figs. 23 and 24, the manifestation of this problem is in the form of striped regions of under or overestimation for the respective state measurement. In particular, the concentration term is seen to suffer the worst estimation of importance near the actuator bounds of u_1 whereas the estimation issues for temperatures importance exist across the entire dataset with less density near the actuator

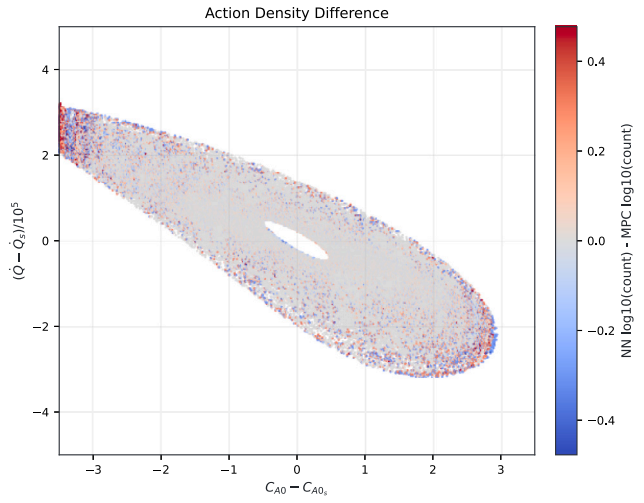

Fig. 23. 2D Histogram of the difference in the number of samples for a given pair of control actions in the overall dataset for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

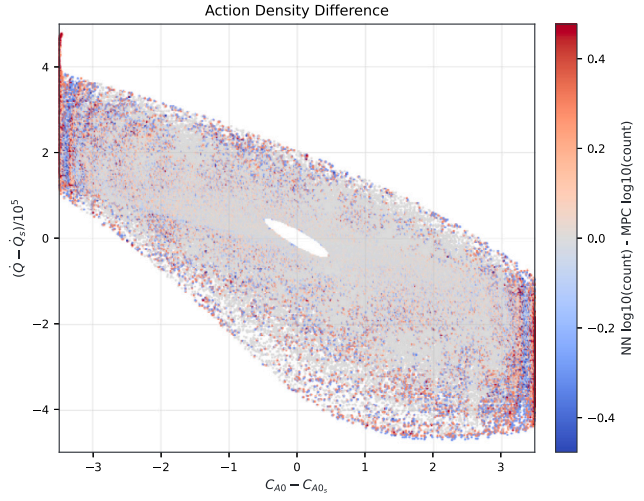

Fig. 24. 2D Histogram of the difference in the number of samples for a given pair of control actions in the overall dataset for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$.

bounds. Both concerns are potential artifacts from the scaling choices, as T will still have a higher magnitude range than C_A due to the range of the state space (which means T has a more sensitive gradient impact) and the `tanh` scale being compared to a linear scaling of the dataset actions (which means actions near the boundary are in the saturated gradient region of `tanh` and are thus harder to train). Figs. 29 to 32 depict the consequence of these problems; there is notable estimation error concentrated in these regions which only scales in magnitude as the datasets coverage expands. Overall, the results demonstrate fairly minimal variation on the order of 1 % error across any given section of the state or action space. Thus, the model can be considered sufficiently well trained and ready for use.

Remark 15. The motivation behind the two datasets is to observe the generalizability of the framework to points outside of its training set. It further allows us to see how the broadening of the dataset in a manner that sacrifices density near the origin for density far from the origin impact the model's performance.

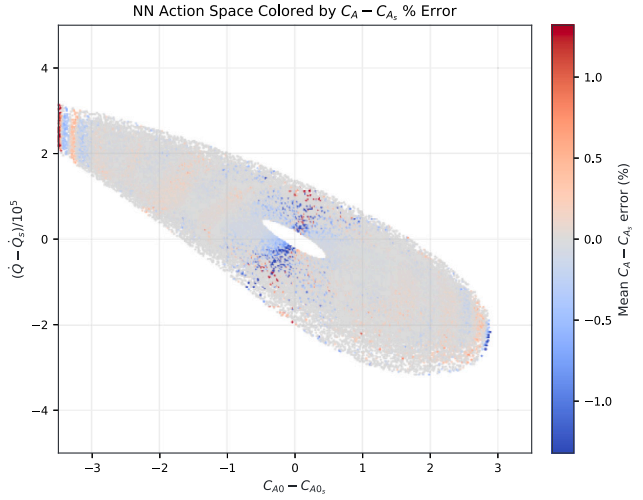

Fig. 25. 2D action space plot color-coded by the difference in x_1 between the IL-controller and the stored MPC data for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

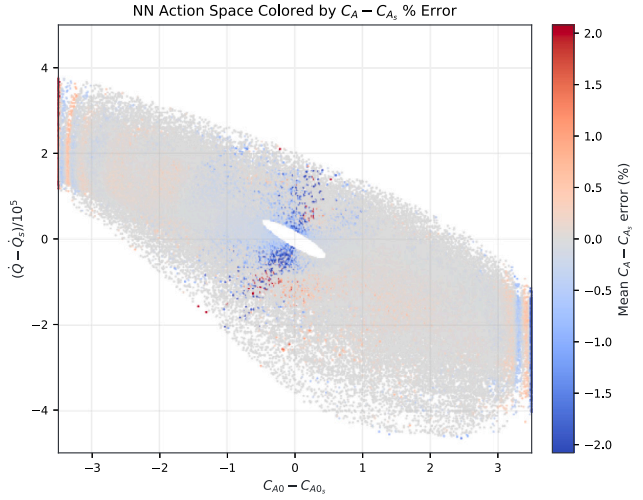

Fig. 26. 2D action space plot color-coded by the difference in x_1 between the IL-controller and the stored MPC data for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Remark 16. The LMPC is used during data generation without regards as to whether or not such an LMPC is feasible to implement as a real-time controller. This is appropriate, and often beneficial, in the case of IL because the LMPC data operates as a target for training but is not required after training is complete. In other words, the LMPC can be infeasible for practical use, but can instead operate as a high-quality reference for IL.

Remark 17. `Huber` loss was used instead of the Mean Squared Error (MSE) loss because of concerns that outlier points may lead to overly aggressive gradients. We applied this logic to all such instances where MSE may be used so long as the range of values that can be expected notably exceeds 1, at which point the squared operation becomes a risk.

Remark 18. `Muon` is utilized in this work due to its close parity to `AdamW` in terms of performance, but notably reduced computational

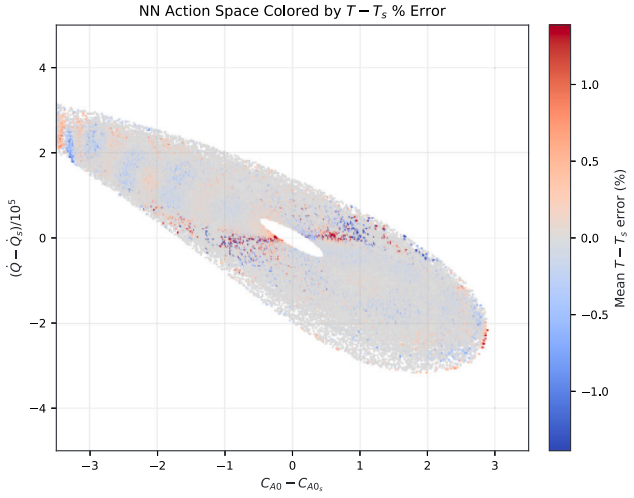

Fig. 27. 2D action space plot color-coded by the difference in x_2 between the IL-controller and the stored MPC data for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

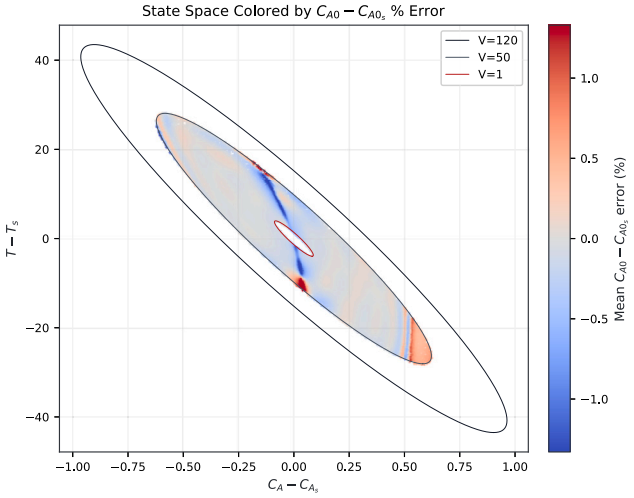

Fig. 29. 2D state space plot color-coded by the difference in u_1 between the IL-controller and the stored MPC data for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Fig. 28. 2D action space plot color-coded by the difference in x_2 between the IL-controller and the stored MPC data for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

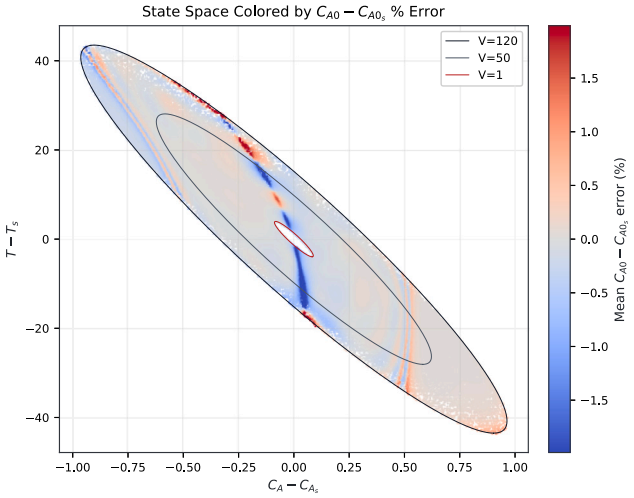

Fig. 30. 2D state space plot color-coded by the difference in u_1 between the IL-controller and the stored MPC data for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

burden. Due to how the optimizer is implemented in PyTorch, `Muon` does not work for the bias vectors, and thus `AdamW` is used.

Remark 19. While the Normalize-and-Project approach was previously mentioned as a suitable modification to the training loop under `LayerNorm`, it was found that the weights and biases did not begin to diverge and thus the approach was deemed unnecessary. This held true for all networks that were designed in this work.

Remark 20. Hyperparameters for this network were not tuned with the exception of the initial learning rate, which was hand tuned in order-of-magnitude increments. It was observed during prior works involving similar chemical reactor systems that the other hyperparameters, such as dropout, had an insignificant impact during training.

5.5. Reinforcement learning-based control design

In this work, we consider 3 different RL-based controllers, of which 2 are CENNC type designs. For all 3, the training code was kept the same. Namely, we implement a modified form of Algorithm 1 using the same CSTR system as the environment. Because the reward function needs to be designed in a manner that mimics the objective of the LMPC, the same future state-based stage cost function is used in the design of the reward function used in all RL-based controllers:

$$r = -\left(s'^{\top} Q_x s' + a^{\top} Q_u a\right) \tag{217}$$

Note that because the cost function is minimized in LMPC but the reward function is maximized in RL, the stage cost's sign is flipped. Additionally, the reward is divided by 1,000 to limit the gradient scale during training. This scale was chosen based on rough observations during training runs that were done prior to finalizing the framework. Regarding the constraint enforcement, the environment is designed

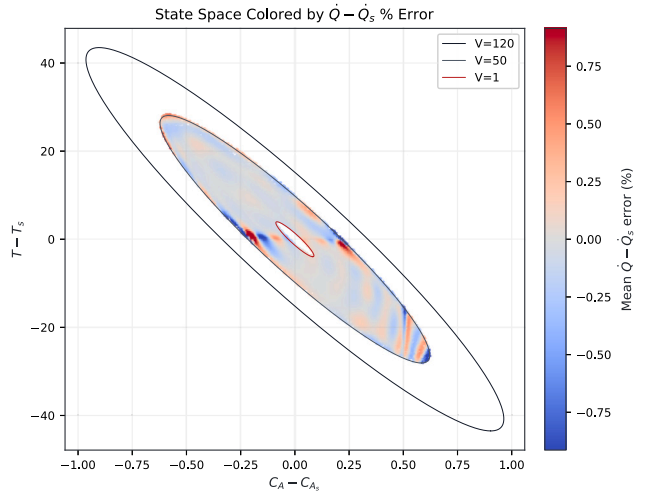

Fig. 31. 2D state space plot color-coded by the difference in u_2 between the IL-controller and the stored MPC data for the system whose initial states satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

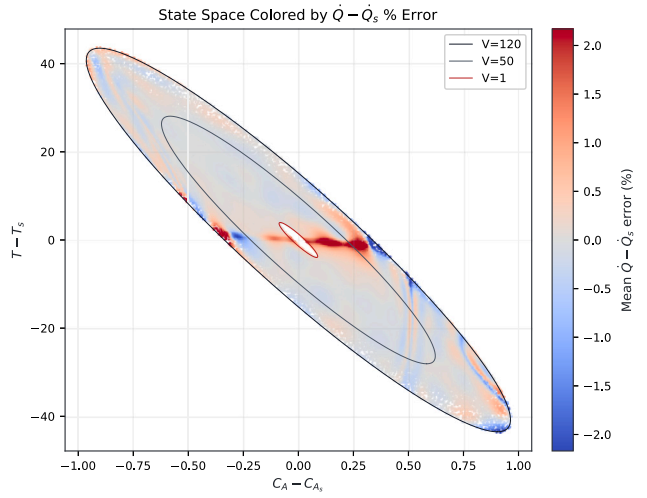

Fig. 32. 2D state space plot color-coded by the difference in u_2 between the IL-controller and the stored MPC data for the system whose initial states satisfy $x_0 \in \Omega_{120} \backslash \Omega_1$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

with respect to the mode of action handling as described in Table 1. As such, `DPre` modal designs will implement the constraint enforcement as effectively being merged with the environment's dynamics whereas `DPro` and `DProj` modal designs will implement the constraint enforcement as a part of the Agent, but not necessarily as a part of the Policy/Actor. Note that the Agent can be considered the full control block whereas the Actor typically is reserved to representing the Policy network. In this context, the `DProj` modality can be considered the case where the Actor that the Critic evaluates is the Agent.

Regarding the modifications done to Algorithm 1, the most notable is the initial random exploration phase being replaced with a noisy controlled exploration phase where the reference P controller generates the action upon which exploration noise is applied (in the same manner as the normal exploration phase in standard TD3). This modification is critical for the design as chosen because random exploration is excessively prone to divergence in safety critical systems. In the context of the chemical process example, thermal runaway is easily achieved in a system with exothermic reactions when the control actions lack any consideration of stability. Because these initial trajectories still have a heightened risk of divergence and because the replay buffer requires sufficient samples before training begins to prevent excessively sampling the same data, training does not begin until 10,000 steps are completed. After these 10,000 steps, the exploration phase continues to use the noisy reference controller for another 10,000. In other words, training starts, but the Actor is not used for exploration until 10,000 additional environment steps are completed.

Table 4
Design parameters for the TD3-based RL controllers.

Variable	Value	Variable	Value
Buffer size	1×10^6	Batch size	1,000
Training episodes	20,000	Max episode steps	5,000
Divergence penalty	1×10^6	Discount factor	1.0
Actor initial learning rate	1×10^{-4}	Critic initial learning rate	1×10^{-4}
Actor minimum learning rate	1×10^{-8}	Critic minimum learning rate	1×10^{-8}
Start steps	20,000	Warmup steps	40,000
Update after	10,000	Update every	1
Actor delay	2	Polyak weight	0.005
Exploration noise start	0.6	Exploration noise final	0.01
Policy noise start	0.4	Policy noise final	0.01
Noise clip	0.5	Reward scale	1,000
Gradient-norm clip	1.0	Penalty weights	1.0
Softplus weight start	1.0	Softplus weight final	10.0
Environment precision	`float64`	Network precision	`float64`

For exploration, trajectories start with initial states within $\Omega_{50} \setminus \Omega_1$ in order to observe how the networks generalize to the unsampled regions. Trajectories have a maximum length of 5,000 steps and are terminated early if the state enters Ω_1 or if the state diverges. We define divergence as any state whose value in Python is either `NaN` or `Inf`. In the case of divergence, the replay buffer stores a duplicate of the prior non-diverged states and replaces the reward with a flat penalty of 1×10^6. Because TD3 is designed for use in finite horizon tasks, the design of the loss term for the Critic include a termination flag as denoted by d in Eq. (26). This term is 1 in the case of divergence and convergence to Ω_1, but is 0 for the case of reaching the maximum step count. The reasoning behind this was that the optimization problem has an arbitrary horizon length, and we did not want to bias the system towards underestimating long horizon runs. The exploration noise used decays linearly after a trajectory concludes from an initial value of 0.6 to a minimum of 0.01 over the course of the training run. The motivation for this decaying exploration noise was to allow for exploration in later trajectories to be localized near the noise-free path in a manner that allows for a fine-tuning approach to the objective.

The training process is terminated after 20,000 episodes are completed instead of a step-based termination criteria or an early-termination criteria. Much like the exploration noise, the policy noise term also decays linearly after an episode is completed, from 0.4 to a minimum value of 0.01. Unlike exploration noise, the policy noise term exists to prevent exploitation, hence we aggressively smooth the Critic during early stages of training and weaken this smoothening effect as training stabilizes. To additionally assist in stabilizing the Critic, the Critic loss equation is modified to use the `Huber` loss function instead of MSE. In order to make the value function match the form of the infinite horizon LMPC cost for the system with early termination, we use $\gamma = 1$. Similar to the IL runs, the RL runs utilize `CosineAnnealingWarmRestarts` with learning rate decaying after every trajectory is complete and resetting after every 4,000 episodes. The remaining configuration details are shown in Table 4.

The Actor loss function is also modified to better suit the control focused framework by including up to 2 additional penalty terms; one based on the degree of constraint violation of the raw NN output and

one based on the degree of internal constraint enforcement (if applicable). In this system, there are only two constraints to consider; the actuator bounds and the Lyapunov constraint. As such, the constraint violation penalty is defined as follows:

$$\mathcal{L}_+ = \lambda_+ \text{MEAN}\left[\left\|\begin{bmatrix}\texttt{ReLU}\left(u_\pi\left(x_i\right)-\mathbf{1}\right)\\ \texttt{ReLU}\left(-\mathbf{1}-u_\pi\left(x_i\right)\right)\end{bmatrix}\right\|_2 + \texttt{ReLU}\left(\frac{L_g V\left(x_i\right)^\top\left(\hat{u}_\pi\left(x_i\right)-u_R\left(x_i\right)\right)}{\|L_g V\left(x_i\right)\|_2}\right)\right] \tag{218}$$

Here, $u_\pi\left(x_i\right)$ is the `tanh` normalized form of the action prior to the internal constraint enforcement layer, $\hat{u}_\pi\left(x_i\right)$ is the form of $u_\pi\left(x_i\right)$ after being multiplied by the actuator scaling so that it is bounded by the actuator bounds instead of $[-1,1]$. For the sake of simplicity, $\lambda_+ = 1$. This is also true for all other penalty weights. This penalty is composed of two components, the first of which is a formulation of the box penalty which abuses the desired output scaling of the network to penalize frameworks that yield actions that violate the actuator bounds. If `tanh` activation is used in the output layer, this term will always be 0, and the same is true for any network that is truly capable of strictly guaranteeing actuator bounds are met within the scaled form. The second term is the projection of the safe v.s. raw action difference vector along the unit normal of the Lyapunov boundary for the given x. This is equivalent to the signed distance to the boundary for a given action, which allows for `ReLU` to filter safe actions out, as negative w.r.t. the boundary is safe. Together, these terms provide a way to penalize the network for producing constraint violating actions based on the extent to which the constraint is violated.

The remaining additional Actor loss term, referred to as the Alpha penalty, penalizes the network for using internal constraint enforcement in an effort to prevent the network from over-relying on it. The form of this penalty is based on a calculated alpha value:

$$\mathcal{L}_\alpha = \lambda_\alpha\left(1-\bar{\alpha}\right) \tag{219}$$

The two CENNC frameworks that are evaluated are the Alpha-Gate approach and the QP-layer approach. Specifically, the QP-layer approach is designed to mimic the KKT optimal solution for the system with its respective gradient defined by Eq. (122) being applied via a STE. In the Alpha-Gate formulation, $\bar{\alpha}$ is simply the average of all of the α terms whereas for the KKT case $\bar{\alpha}$ is an average of the effective α terms based on the constraint satisfaction of the internal intermediary actions. This Alpha penalty term is disabled until 40,000 environment steps are completed as a means of allowing the network to *warm up*. The delayed penalty allows for the network to somewhat stabilize before facing the full multiobjective optimization problem.

Weights and biases were initialized through orthogonal initialization with various gains and biases based on the layer design. Hidden linear layers were initialized with $\sqrt{2}$ gain and no bias whereas the final layer that yields the action uses a gain of 0.01 to limit initial action scale. `LayerNorm` blocks were initialized with the default PyTorch initialization.

5.5.1. External constraint enforcer baseline

The only non-CENNC network that is trained using RL is a baseline controller that does not utilize the Alpha penalty and thus has no inherent consideration of the constraints beyond what is induced by the Action handling modality that is chosen. Specifically, the network is a feedforward neural network with 3 hidden layers of width 32, 32, and 64, respectively, for a total of 4 layers that convert the dimensions $2 \to 32 \to 32 \to 64 \to 2$. The input layer and each hidden layer have pre-activation `LayerNorm` with `Mish` activation with the exception of the output layer which has no `LayerNorm` and uses `tanh` activation.

As described above, the FNN-baseline does no internal constraint enforcement and has no Alpha penalty, meaning it has no consideration of the constraints. This makes the impact of the `+` form of the Action handling modality especially impactful as it injects signals to the network based on the constraints that otherwise would have no alternative.

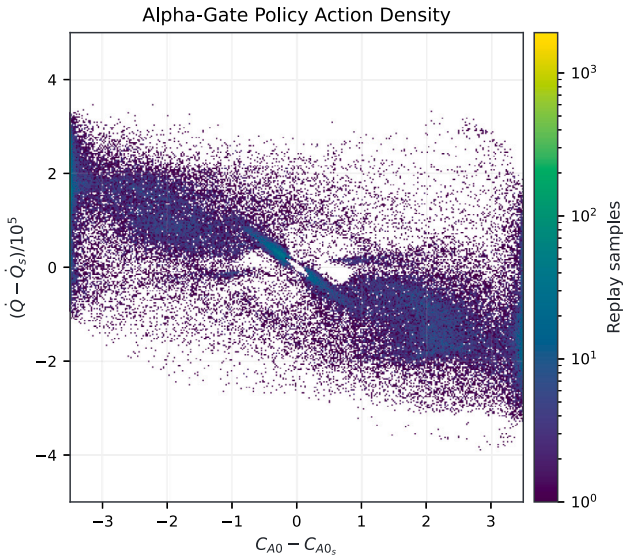

Fig. 33. 2D Histogram of the number of samples for a given pair of control actions generated by the `DPro+` variant of the Alpha-Gate framework CENNC whose initial states during training satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

5.5.2. CENNC design

Both of the CENNC designs are built on the same model with the sole exception of their constraint handling section. These networks are designed to be self-refining through the use of LSTMCells, which are the building blocks of Long Short-Term Memory (LSTM) NN designs. The individual cells were used because time-series data is not utilized in these designs. Instead, the internal constraint enforcement is used to generate candidate actions which are fed into subsequent LSTMCells, effectively generating a time series of progressively refined action proposals that would ideally utilize information from prior iterations to generate better quality actions. As designed, the networks use 1 LSTMCell with a hidden width of 32 unrolled for 5 steps. This is notably different than using 5 unique cells in the sense that all 5 steps have the same weights and biases in our case. Both networks utilize a repeated design aspect referred to as the *Actor head* that consists of the $32 \to 64 \to 2$ section from the non-CENNC FNN design.

Unlike the non-CENNC FNN design, these networks have additional inputs that serve as both auxiliary inputs and as necessary inputs for the constraint enforcement block. Notably, there are an additional 2 inputs; the reference control action and its corresponding $\dot{V}$ value. The control actions are scaled through division of their bounds and the $\dot{V}$ input is divided by 100 to keep the input scale roughly around 1. Although $\dot{V}$ can be used directly for the constraint, it is primarily included as an auxiliary input with the added benefit of enabling a soft constraint enforcement network design where $\dot{V}$ is predicted by the network for a given u instead of directly calculating it through the lie derivative formulation. While this form of the network would no longer strictly satisfy the constraints, it enables the framework to be applied to a system where the dynamics are unknown which can yield improved closed-loop performance relative to a network trained on a first-principles model that may poorly approximate the dynamics.

For the Alpha-Gate CENNC design, the constraint was formulated as the simplified form of Eq. (6e) as defined in Eq. (147). The Alpha-Gate formulation follows from the discussion in that section where the

softplus-based STE uses a form of β that linearly increases after each trajectory from an initial value of 1 to a final value of 10 at the end of the training. Because the bounds are pre-satisfied by the tanh activation in the Actor head, and there are no further constraints, there is no need for additional gates. The Alpha penalty for this formulation is simply the average of the 5 α terms that arise during the forward pass. The average was used instead of the final α in order to motivate the network to utilize all refinement steps instead of over-relying on the final fallback step.

For the KKT CENNC design, the safe action is of known form using Eq. (119) where the STE Jacobian is of the form Eq. (122). The example that derives these equations can be used as reference for the justification of the design. Unlike the Alpha-Gate, the KKT solution cannot ignore the issue of bound satisfaction because the minimal projection with respect to the Lyapunov constraint is not guaranteed to satisfy the bounds whereas the reference control is designed to inherently satisfy these bounds. Because of this, the equivalent QP-problem is formulated to consider the box constraints and the Lyapunov constraint simultaneously. The Alpha penalty used for this framework is the average of the 5 effective α values that would have been used if the Alpha-Gate formulation was used. In other words, the proposed refinement to the action is checked and yields either 0 or 1 based on if both constraints are satisfied before the KKT solution is applied.

Remark 21. Although these frameworks may utilize the + penalty from the Action handling modality, the penalty is effectively inactive because both frameworks strictly satisfy the constraints and thus will always yield a penalty of 0. The remaining issue is the problem of machine-precision, which may yield negligibly small penalty terms, hence the use of float64 in all networks for this paper. Because the LMPC uses float64 to accurately handle the nonlinearities during numerical optimization, all the constraints applied to it would be of the same numerical precision. This can lead to misleading instances of fallback if the NN designs are defined with respect to a different numerical precision despite being strictly safe for this precision. While using a higher numerical precision is beneficial, it is also detrimental to the scalability of the network, and thus it is recommended that the constraints be defined in a conservative manner using a small bias to mitigate accidental closed-loop fallback.

5.5.3. Critic design

Thus far, the RL-based designs have described the formulation of the CENNC, which only applies to the Actor. In all formulations, the Critic is defined to be the exact same. The Critic's loss function is the generic TD3 loss function but with Huber loss used instead. The NN design for the Critic is a residual feedforward neural network whose inputs consist of the normalized form of the state and action vectors and whose output is Q with respect to the scaled form of the reward function. The networks input layer has pre-activation LayerNorm and is fed into 3 residual blocks. The residual blocks utilize LayerNorm prior to the input of the linear layer (pre-LayerNorm) as well as an activation-free output layer. In other words, each residual block has two layers, the last of which has no nonlinear activation function. After the 3 residual blocks, there is a final LayerNorm followed by the activation function followed by the linear output layer. Overall, the network is 8 layers deep. With the exception of the input and output layers, these layers are all of width 16, and all instances of the activation function use GELU.

5.5.4. Training behavior

Like the IL run, the behavior during training allows for insights into how the models are expected to behave. Because RL has an exploration integrated into the training loop, the analysis of model performance can be partially done prior to any closed-loop runs through sufficient logging of performance. As such, several metrics were recorded during the training run across 4 groups of RL runs; the RL Baseline non-CENNC model, the Alpha-Gate CENNC model, the KKT CENNC model, and the representative well-performing configuration comparison. The last group is a group that takes the DPro+ variants of each framework and compares them across architectures to see how the architecture itself impacts performance. DPro+ was chosen due to its relatively better performance compared to DPre+ and its relatively simpler design compared to DProj variants. These 4 groups are plotted as sub-figures in one figure to aid in direct comparisons.

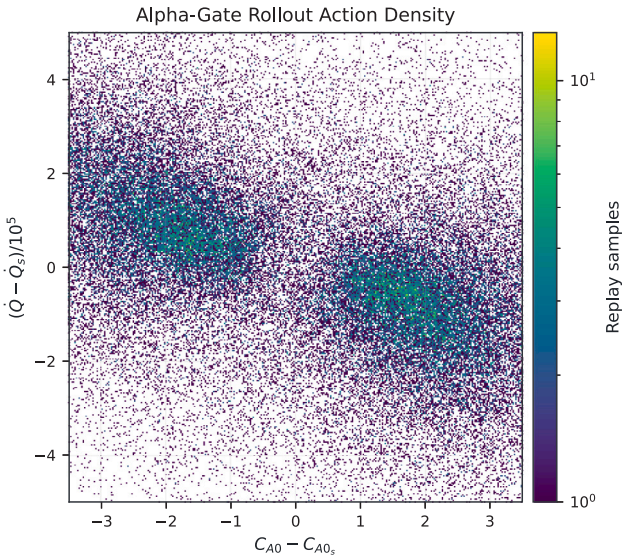

Fig. 34. 2D Histogram of the number of samples for a given pair of control actions generated by the DPro+ variant of the Alpha-Gate framework CENNC with added exploration noise and clipping whose initial states during training satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

Before looking at the information that is available during the training run, it is worth looking at the effective distributions of the replay buffers experienced in the runs. For this, consider Figs. 33 to 36. Figs. 33 to 35 are action density plots that demonstrate the distribution of actions during the training run as captured from the final replay buffer after training is complete. It can be seen that, although the raw network actions eventually converge to similar action distributions seen in the IL run from Fig. 12, without any noise the distribution does not sample the action space well. For RL, it is important that the sampling is both sufficiently diverse and sufficiently dense around the optimal regions. Without noise, these actions only satisfy the later, which can hinder model generalization to unseen behaviors. After adding noise, we get Fig. 34, but noisy action risk violating the stability constraints, and thus the safe actions are the only applied actions to the environment which yields Figs. 35 and 36. These plots demonstrate that the desired distribution is achieved for the action space and that the corresponding state space distribution also follows the desired distribution. Notably, the distributions are more densely aligned to the diagonals than the LMPC distributions, which is likely a consequence of the noise terms that exist during training. Regardless, the distributions indicate that the data is well sampled.

To start, the Actor and Critic loss function plots are presented in semi-log form in Figs. 37 and 38, respectively. Immediately after training starts, the Critic loss collapses as the limited data is trained on. Even with the rapid collapse, the loss remains fairly high overall because the actor in the bootstrapped term is effectively noise. As more data is introduced, the Critic can exhibit slower further improvement, flat lining, or a worsening in loss in a seemingly random manner. This is partially because of the actor updates which initially will pass noisy actions that are out-of-distribution relative to the reference controller actions to the Critic. As a result, the Critic value will be inaccurate, which risks leading the Actors' proposed actions to degrade in quality,

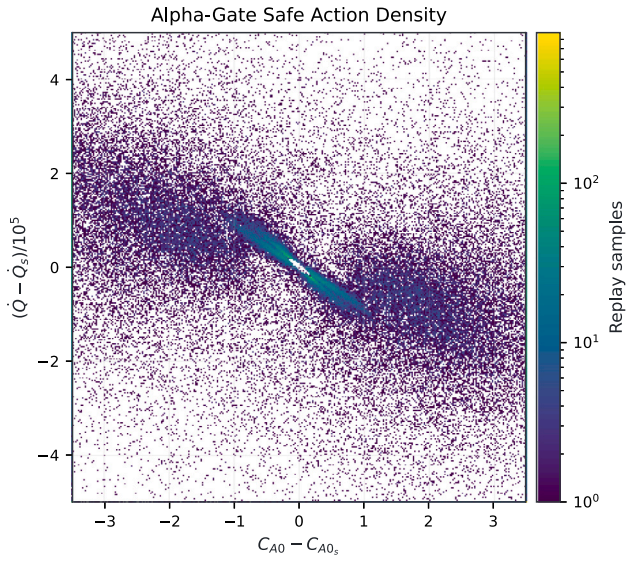

Fig. 35. 2D Histogram of the number of samples for a given pair of control actions generated by the DPro+ variant of the Alpha-Gate framework CENNC with added exploration noise, clipping, and reference controller fallback (if needed) whose initial states during training satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

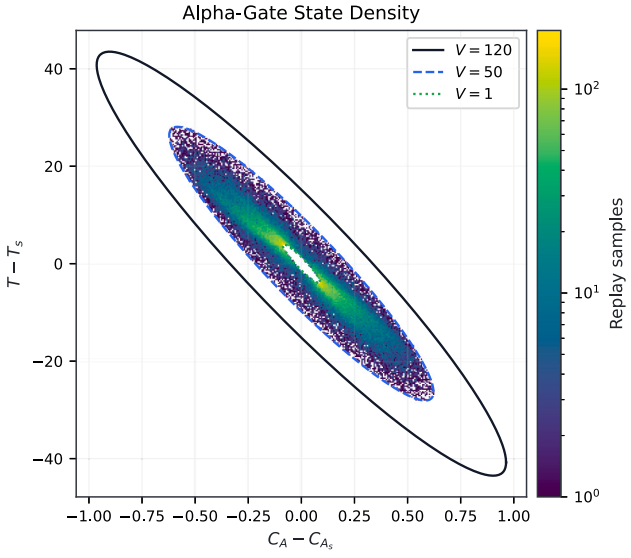

Fig. 36. 2D Histogram of the number of samples for a given pair of states explored by the DPro+ variant of the Alpha-Gate framework CENNC whose initial states during training satisfy $x_0 \in \Omega_{50} \backslash \Omega_1$.

but it can also improve the quality if the random weights are lucky and yield actions close to the reference. In either case, the Actor becomes less noisy and more predictable, but does so in a heavily delayed manner due to the update delay, Polyak averaging, and the gradual overriding of the low-magnitude output layer initializations. As a result, the Actors actions are not only out-of-distribution with respect to the reference controller, but are out-of-distribution with respect to itself from earlier steps, hence the gradual climb in Actor loss. Once the Actor begins influencing the exploration at the 20,000 step mark, the issues of early distribution mismatches begin to fade. By the 40,000 step mark, where the Alpha penalty loss is activated, all frameworks have exhibited gradual loss decay that remains stable for the remainder of the training run.

Regarding the gradient plots seen in Figs. 39 and 40, the frameworks begin to demonstrate unique behavior. For context, the Actor and Critic networks are both gradient clipped so that the gradients that truly impact the weights are no larger than a norm of 1. As such, all the gradient plots demonstrate the gradients prior to clipping. The RL Baseline framework exhibits an initial phase of high magnitude gradients that quickly stabilizes even before the Actor is utilized for exploration. The DPre+ variants gradients have effectively converged by the 40,000 step mark, which indicates that the Actor in this framework may be stuck oscillating between candidates unlike the DPro+ case which exhibits continually decaying gradients, indicating that it has reached a local minima and is refining the weights. The KKT frameworks all exhibit an extended period of heightened gradient scales indicating an increased difficulty in tuning the Actor. Much like the RL Baseline case, the DPre+ variant exhibits convergence to higher gradient norms later in training. The same behavior is seen in the Alpha-Gate framework, but the heightened gradient scales are short lived. Unlike the KKT and RL Baseline case, the gating mechanism is seen to occasionally result in very small gradients early in training which resulted in the zoomed-out scaling. When comparing the DPro+ variant of the 3 frameworks, it is seen that the RL baseline yields the smallest gradients, and thus the most stable training, while also reaching these small values in the quickest manner. Both KKT and Alpha-Gate eventually reach a similarly small gradient norm, but do so via a gradual decay instead of the aggressive decay seen in the RL Baseline.

Unlike the Actor, the Critic gradient trends are mostly similar. All frameworks saw an initial spike followed by an effectively flat gradient norm curve for the DPre+ variant whereas the other variants saw continual decay. Interestingly, the learning rate resets do not visibly impact these curves until later in training, indicating that the loss landscape for the training may be smooth for most of the training run.

Constraint violation is only a concern for the RL Baseline case, as the CENNC framework strictly satisfies the constraints regardless of if the KKT or Alpha-Gate framework is used. Regardless, the + penalty case is plotted for all frameworks to demonstrate the previously mentioned numerical precision impact. Fig. 41 demonstrates how the apparent violation of constraints reaches a stable minimum in both the DPre+ and DPro+ variants of the RL Baseline. Unlike the KKT or Alpha-Gate cases, because the RL Baseline has no inherent constraint enforcement, there is always a risk that the network produces a constraint violating action. Because of this, the fact that the + penalty is the only driving force for constraint satisfaction can be overpowered by the primary loss term. This conflict is a key issue in multi-objective frameworks which highlights the benefits of the CENNC framework.

For the CENNC framework, the Alpha penalty functions as the main method of reducing reliance on internal fallback, and so we monitor this penalty and its impact on α. Fig. 42 demonstrates that both the KKT and Alpha-Gate frameworks exhibit a high initial penalty that quickly stabilizes before entering a steady decay trend. As is true in many of the prior plots, DPre+ operates as an outlier from the other variants and has periods where the Actor relapses back to relying on internal fallback. In terms of the overall magnitude of the penalty, the Alpha-Gate approach relies less on internal fallback than the KKT approach. The RL Baseline had no Alpha penalty and thus the data is not plotted. Fig. 43 demonstrates the impact of this penalty and the main loss term on the actual average α. In both frameworks, it can be seen that the DPre+ case has a quicker initial convergence to a high α but a lower peak value after the Actor stabilizes during warm-up relative to the other variants. Both the KKT and Alpha-Gate behave similarly in the 10,000-20,000 region before exhibiting a second spike in the 20,000-40,000 region. While Alpha-Gate exhibits a delayed second spike, it stabilizes nearly at 1 without the influence of the Alpha penalty whereas KKT stabilizes near 0.9. Once the Alpha penalty is active, all networks quickly converge near 1 and remain there for the rest of the run with the exception of occasional deviations that are quickly

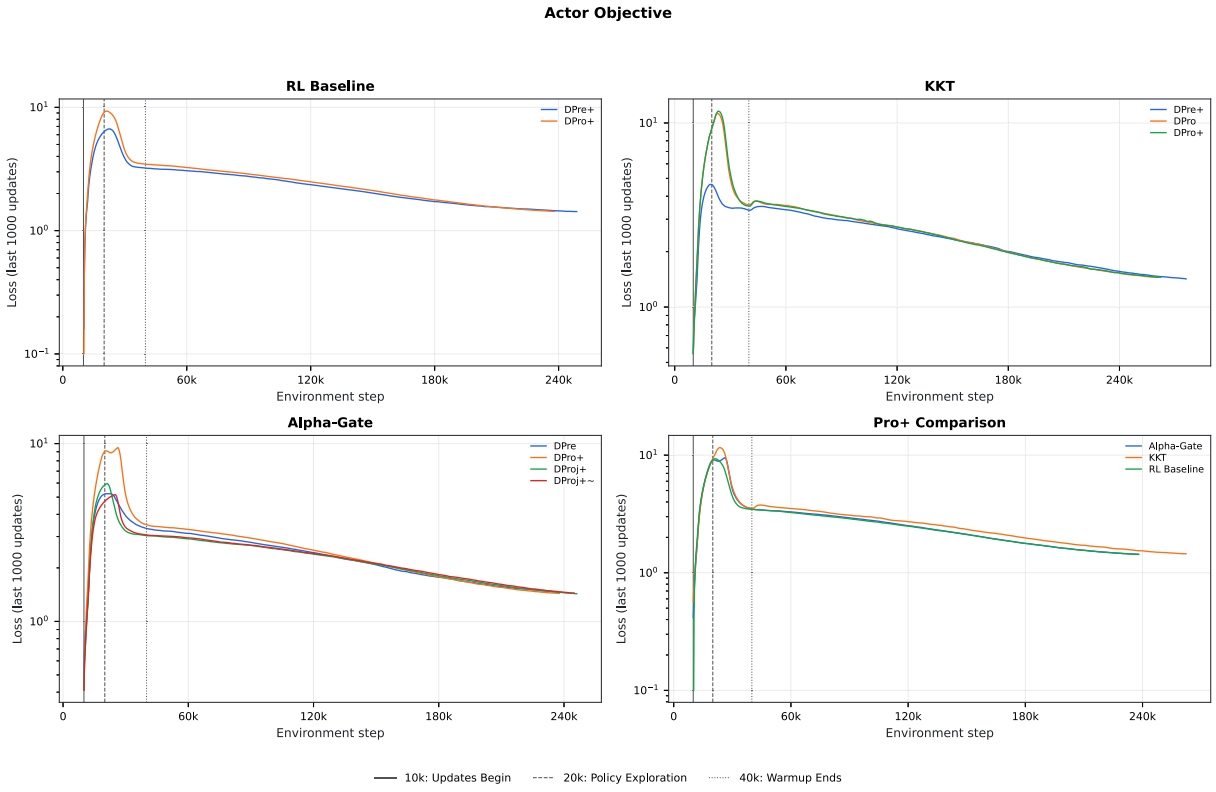

Fig. 37. Semi-Log plot of the overall loss term for the Actor network at each environment step during the training run. Objective refers to this overall loss term to account for the fact that some loss terms are not universally present across architectures or variants.

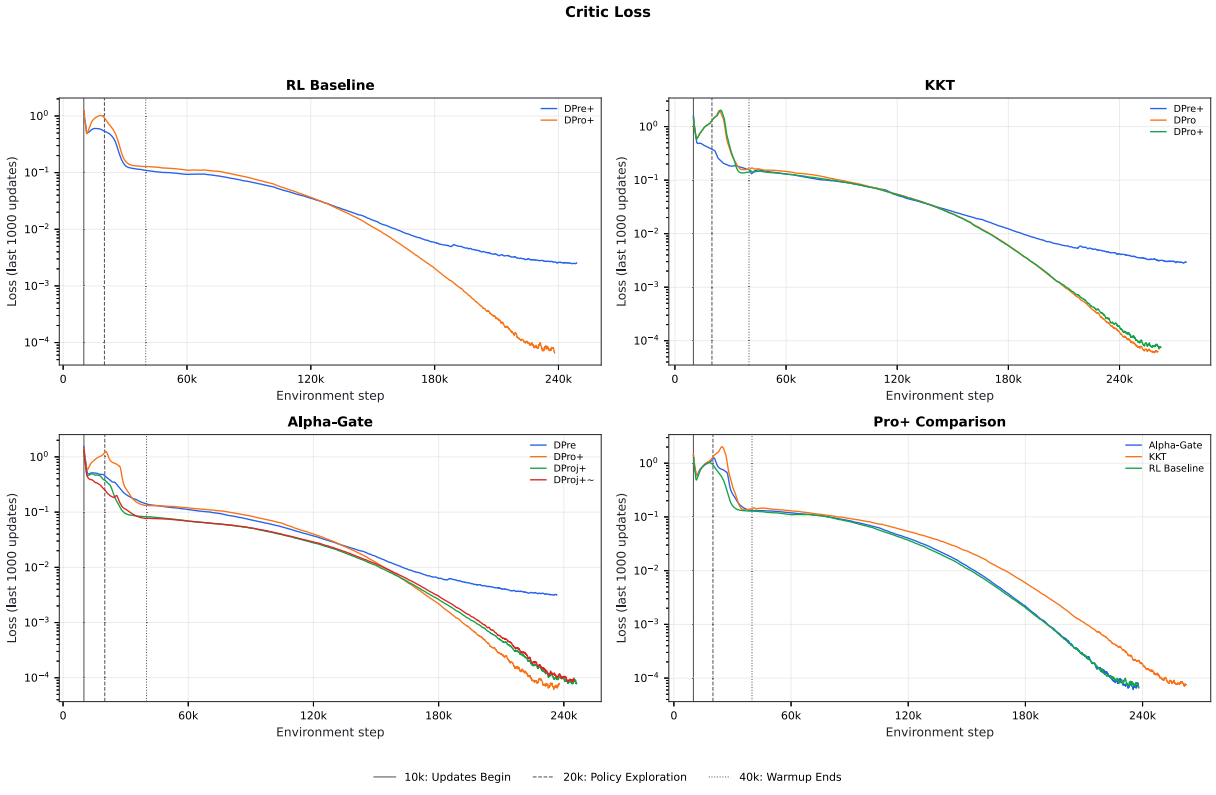

Fig. 38. Semi-Log plot of the temporal difference loss term for the Critic network at each environment step during the training run.

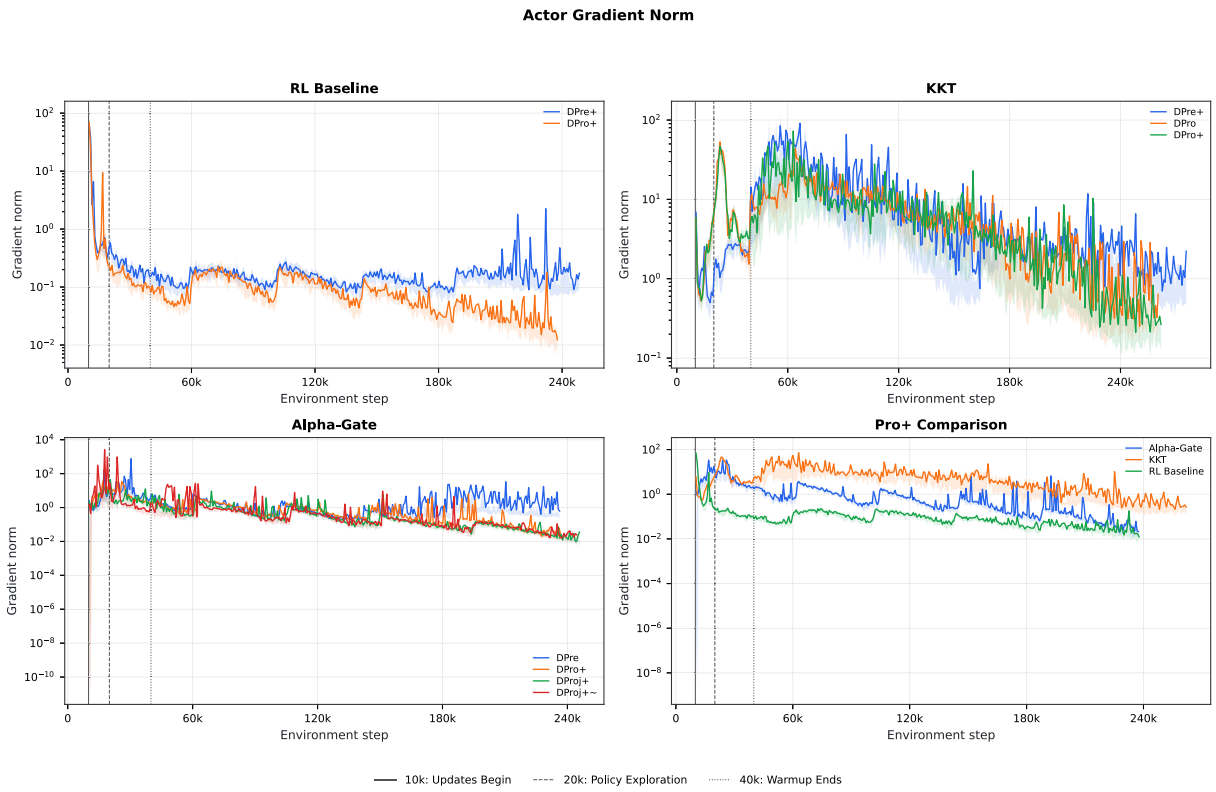

Fig. 39. Semi-Log plot of the gradient norm for the Actor network at each environment step during the training run as reported by PyTorch's `torch.nn.utils.clip_grad_norm_` function. The norm presented is the value prior to gradient clipping. To improve readability, the x-axis is binned into 350 equal sized bins to allow for the banded form of the plots to be shown. The banded form includes the median curve along with a semi-transparent banded region that spans the 25–75th percentiles.

corrected. Overall, both CENNC frameworks indicate that reliance on internal fallback is minimal.

Finally, we can roughly observe the expected performance of these models through their estimation error and the corresponding exploration phase cost trajectories. Although these involve the noise terms, the noise is decayed to a minimal degree by the end of the training run, which would make the exploration phase equivalent to a slightly noisy closed-loop run. In all runs, a metric referred to as the average step estimation error is calculated. This term takes the value of Q at a given point along a trajectory and compares it to the true remaining cost for that trajectory. The resulting step estimation error is then averaged for all steps along a trajectory. In other words, this metric roughly models how well the Critic is at predicting the true trajectory cost under the Actor. Fig. 44 demonstrates that all frameworks exhibit an initial gradual decay, followed by a rapid decrease in error when the Actor is allowed to influence exploration, followed by gradual decay that lasts for the remainder of the training. Notably, all variants experience gradual acceleration of this decay as the noise terms fade and the networks converge to accurate estimates. `DPre+` is once again the outlier, as it stops improving near the end of the training. In fact, in the Alpha-Gate case, it begins to worsen near the end of the training run. By the end of training, all frameworks roughly converge to the same degree of accuracy in this metric, although some frameworks take more exploration steps before finishing all the required training trajectories. Fig. 45 demonstrates that all models experience a rapid convergence to an overly-aggressive policy that finishes the trajectories in a few steps. KKT in particular converges to the least aggressive of the 3 frameworks, which leads to it taking more environment steps over the course of the training. All frameworks begin to polish their policies shortly after the halfway mark to be less aggressive, as seen by the gradual increase in step count. While this may seem unintuitive, it is important to note that the cost optimal control is not necessarily the most aggressively stabilizing controller. Fig. 46 demonstrates this fact by showing how all frameworks and their variants follow the same rough trend towards a lower episode cost. In fact, all networks that were trained end with a similar average episode cost, but the metric is not a reliable indicator of true performance because of the significance of the initial state on the optimal trajectory cost.

Training times varied between frameworks and variants. Because RL requires exploration and training in a sequential loop, there was no parallelization utilized, which slowed training down. Table 5 summarizes these results for the available configurations. In terms of training time, KKT stands out as the slowest framework to train at roughly 22 h despite already being simplified compared to true QP-layer designs. During the design stage of the frameworks, early trial runs with a true QP-layer approach indicated training times spanning several weeks. The Alpha-Gate approach (with runs spanning between 9 h and 14 h) is roughly twice as slow as the RL Baseline approach (with runs spanning between 5 h and 8 h) and roughly twice as fast as the KKT framework, but it is important to keep in mind that the NNs themselves are not structured equivalently between the RL Baseline and the CENNC frameworks. Of the variants, the `DPre` variants took the longest despite being the simplest to calculate as a result of the poorer training behavior that these variants exhibited. In particular, the `DPre` variants were upwards of 45 % slower than the same frameworks `DPro+` variants. `DProj` variants were seemingly the fastest despite being the most computationally expensive variant, but the difference in training time is not as significant as the `DPre` variant.

Remark 22. It can be seen in the loss curves from Figs. 37 and 38 that all of the runs continue to exhibit gradual improvements but are terminated prior to stabilizing at a minimum value. While this can partially be explained by the influence of the decaying noise

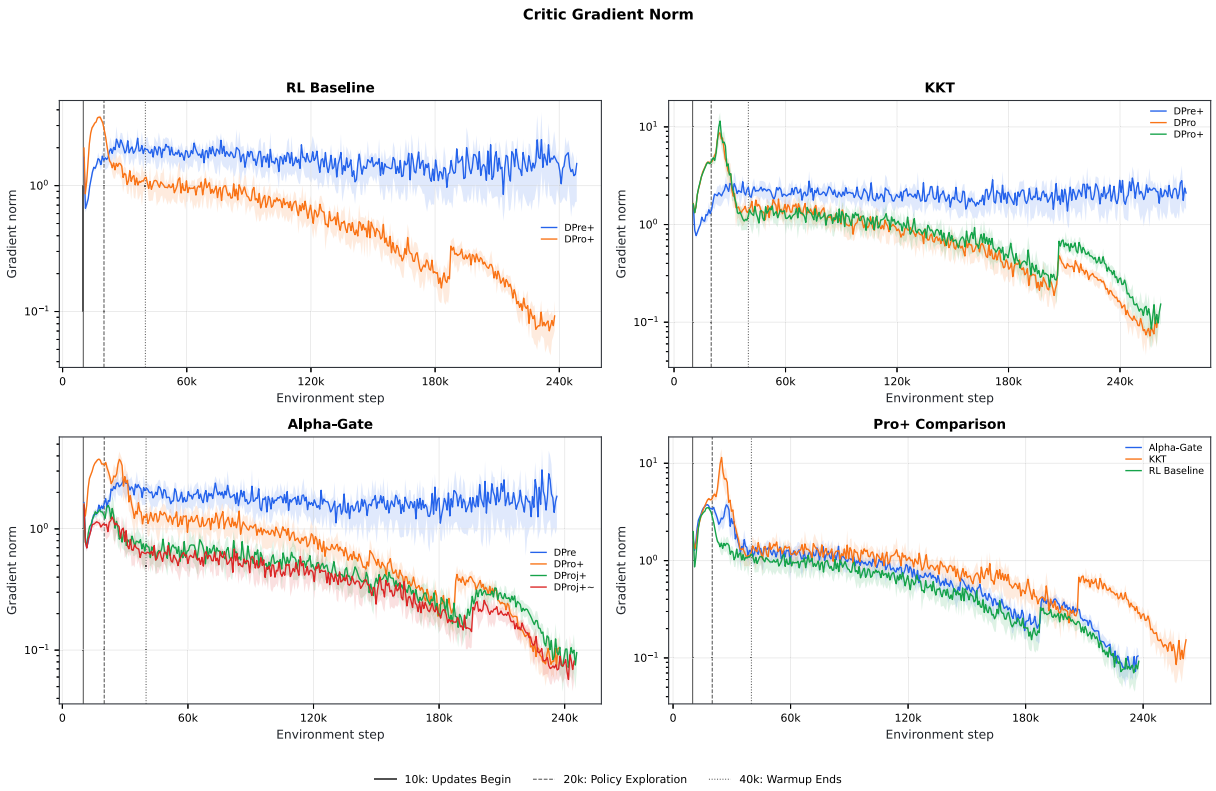

Fig. 40. Semi-Log plot of the gradient norm for the Critic network at each environment step during the training run as reported by PyTorch's `torch.nn.utils.clip_grad_norm_` function. The norm presented is the value prior to gradient clipping. To improve readability, the x-axis is binned into 350 equal sized bins to allow for the banded form of the plots to be shown. The banded form includes the median curve along with a semi-transparent banded region that spans the 25–75th percentiles.

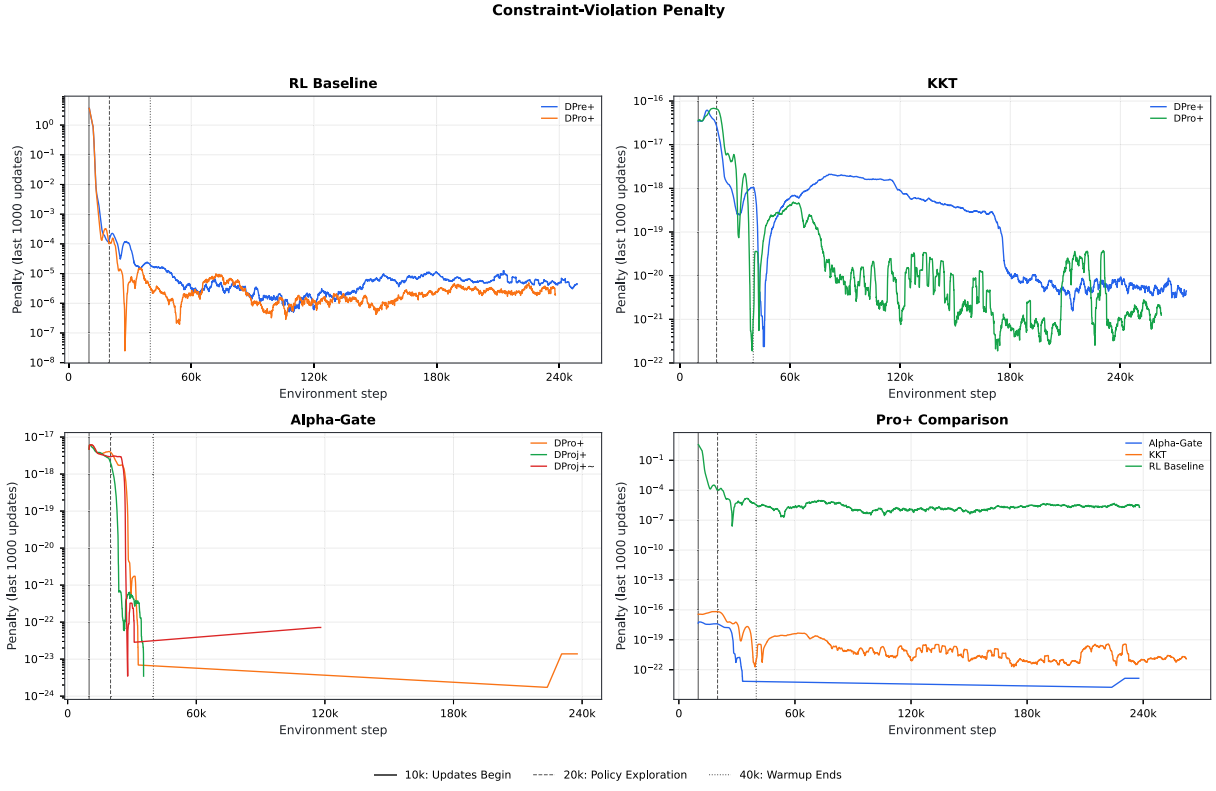

Fig. 41. Semi-Log plot of the constraint violation loss term ($\mathcal{L}_+$) for the Actor network at each environment step during the training run. Because 0 cannot be represented in a Semi-Log plot, these points are omitted, which results in the occasional large linear connection that linearly interpolates these values instead. To improve readability, the metric is averaged over the last 1,000 updates.

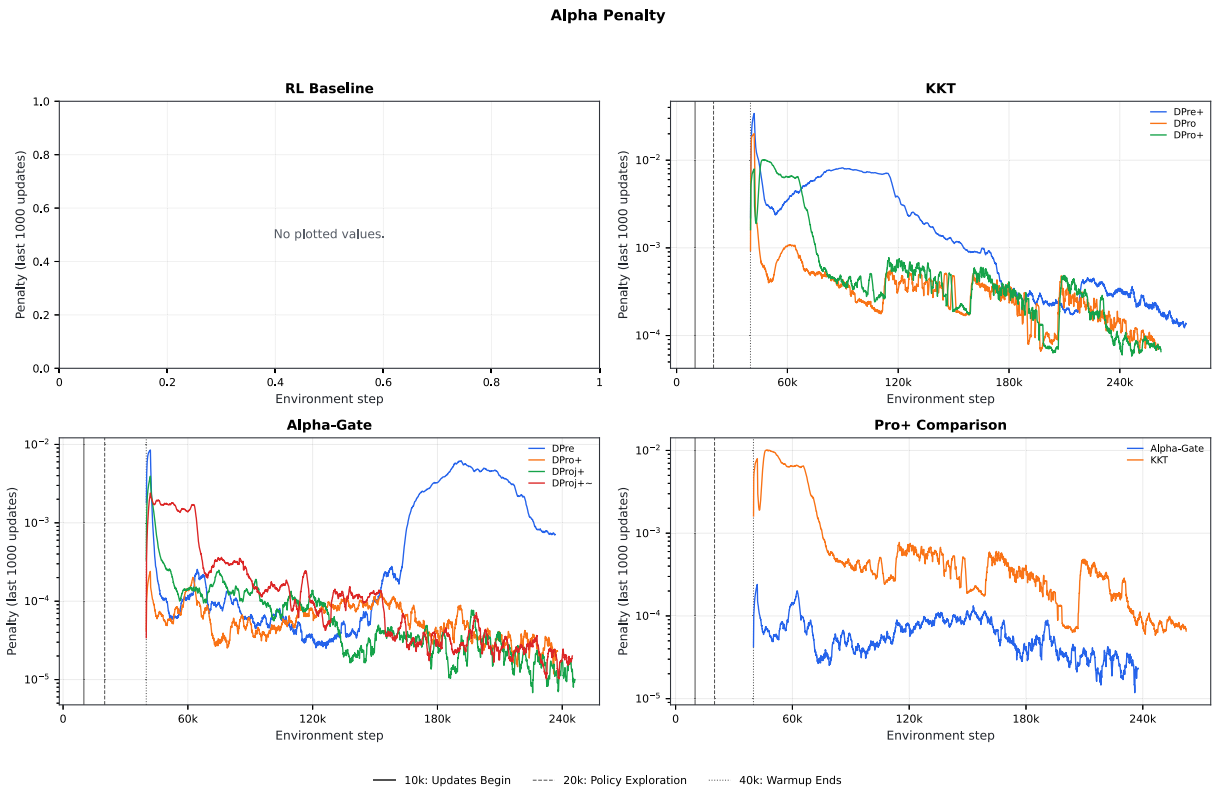

Fig. 42. Semi-Log plot of the Alpha penalty loss term ($\mathcal{L}_\alpha$) for the Actor network at each environment step during the training run. Because the RL Baseline framework does not use internal fallback mechanisms due to its design as a non-CENNC framework, its plot is left blank. To improve readability, the metric is averaged over the last 1,000 updates.

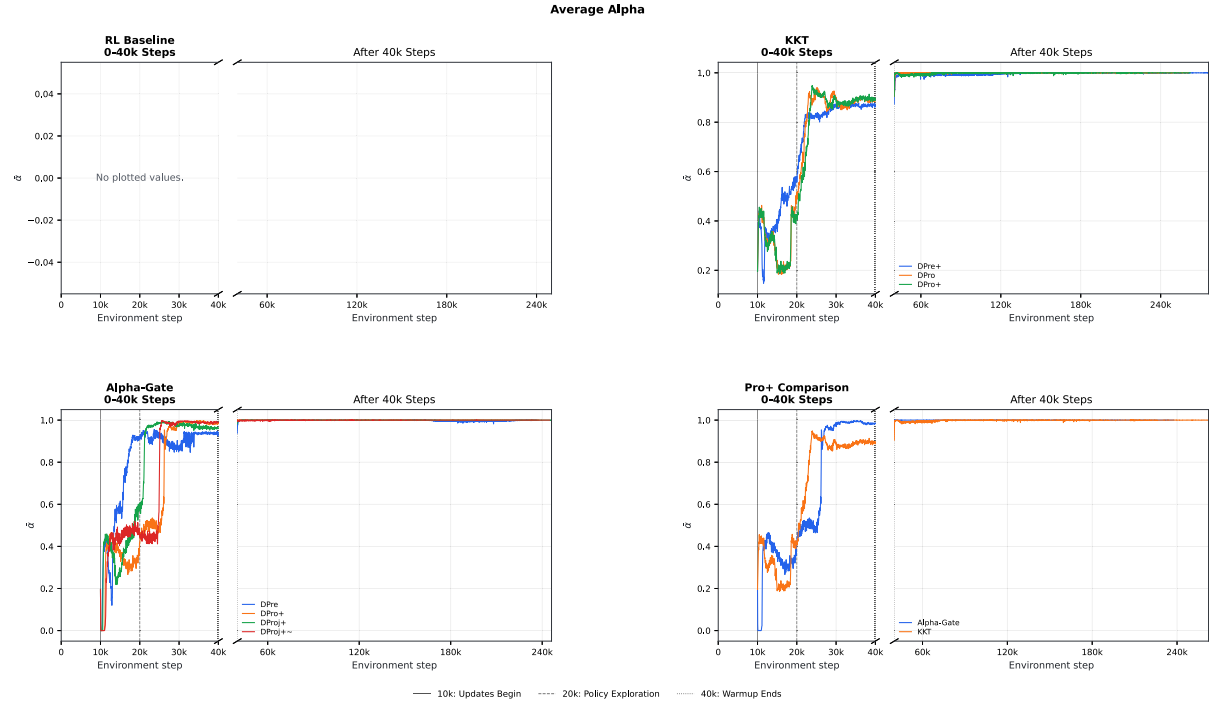

Fig. 43. Plot of the average α term ($\bar{\alpha}$) for the Actor network at each environment step during the training run. For the KKT case, this term is calculated manually based on its internally proposed actions. Because the RL Baseline framework does not use internal fallback mechanisms due to its design as a non-CENNC framework, its plot is left blank.

design, the networks can potentially be improved further. The purpose of the various designs is not to demonstrate a perfectly optimal implementation. Instead, these designs exist to demonstrate, broadly, how the architectures behave under the same training conditions. The

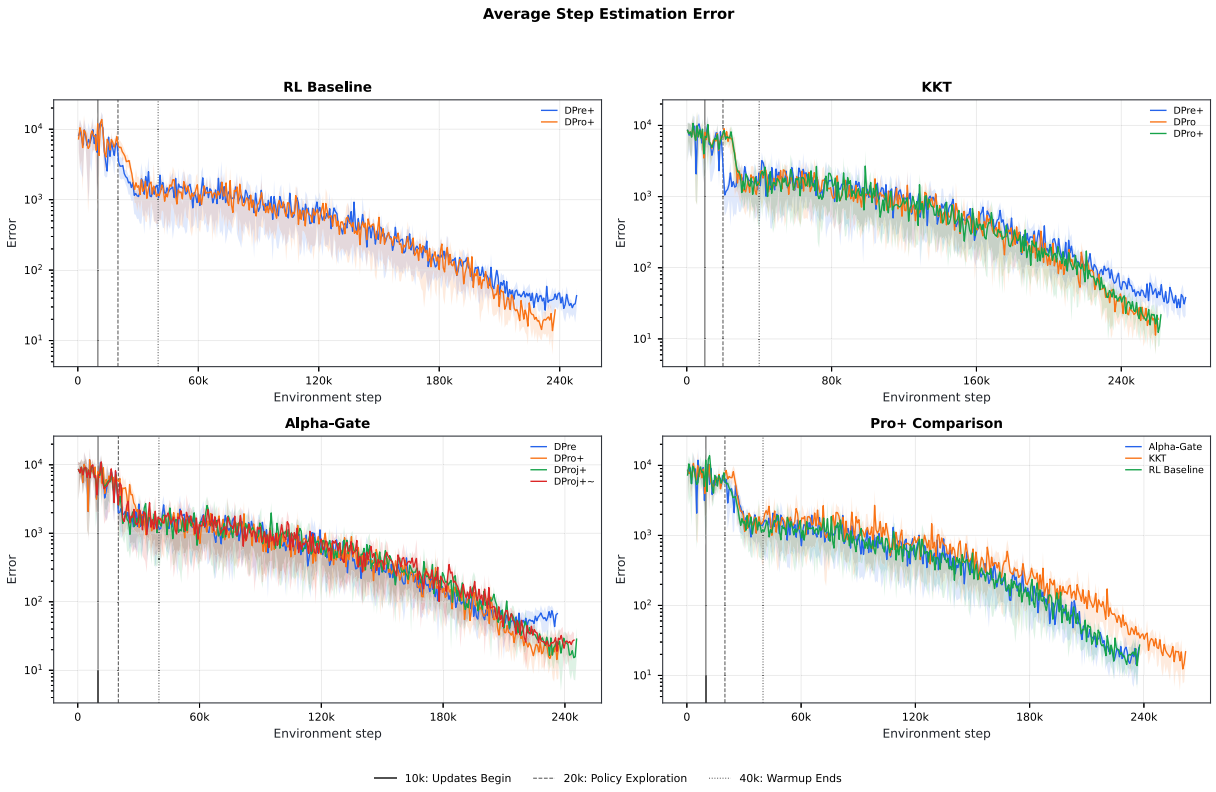

Fig. 44. Semi-Log plot of the average estimation error of the Critic network at each environment step during the training run. Step estimation error is calculated as the mean absolute error between the remaining cumulative trajectory cost and Q for the current state (which would ideally be 0 due to γ being 1). The average of this metric is calculated over the entire current trajectory. To improve readability, the x-axis is binned into 350 equal sized bins to allow for the banded form of the plots to be shown. The banded form includes the median curve along with a semi-transparent banded region that spans the 25–75th percentiles.

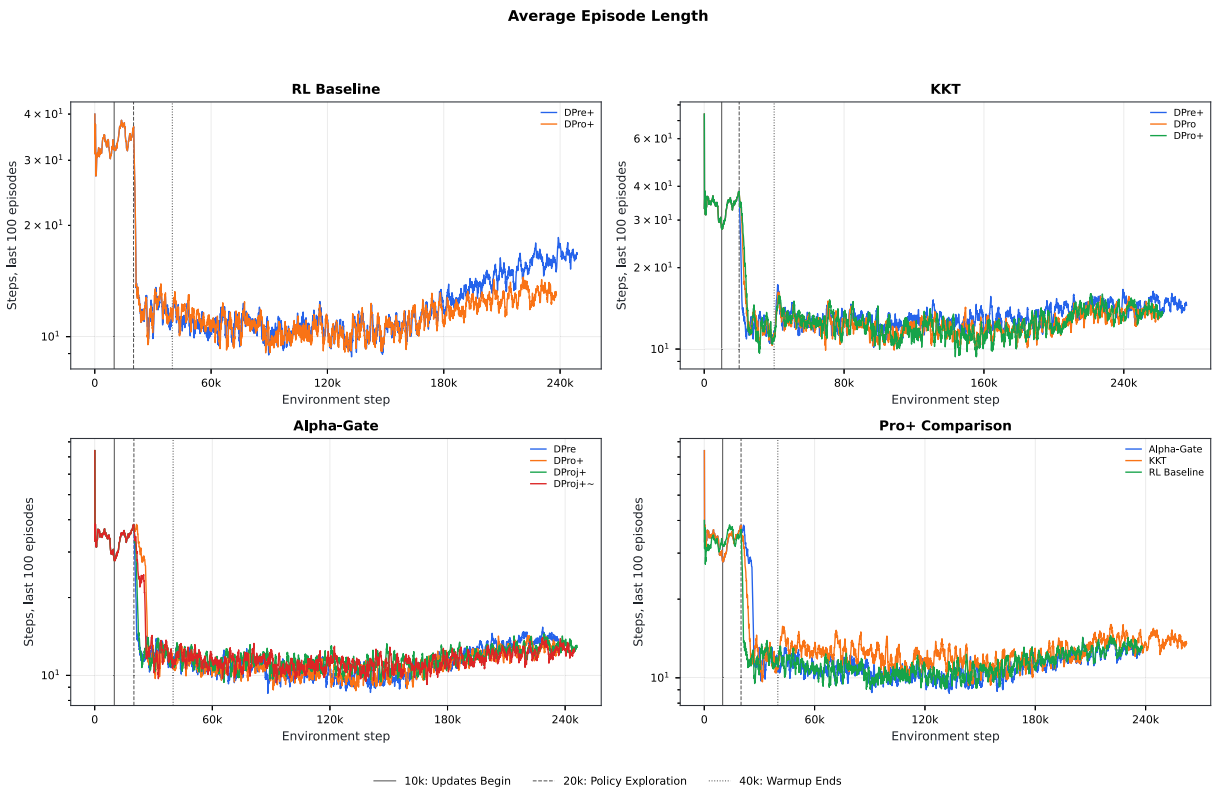

Fig. 45. Semi-Log plot of the average episode length of the exploration phase at each environment step during the training run. To improve readability, the metric is further averaged over the last 100 episodes.

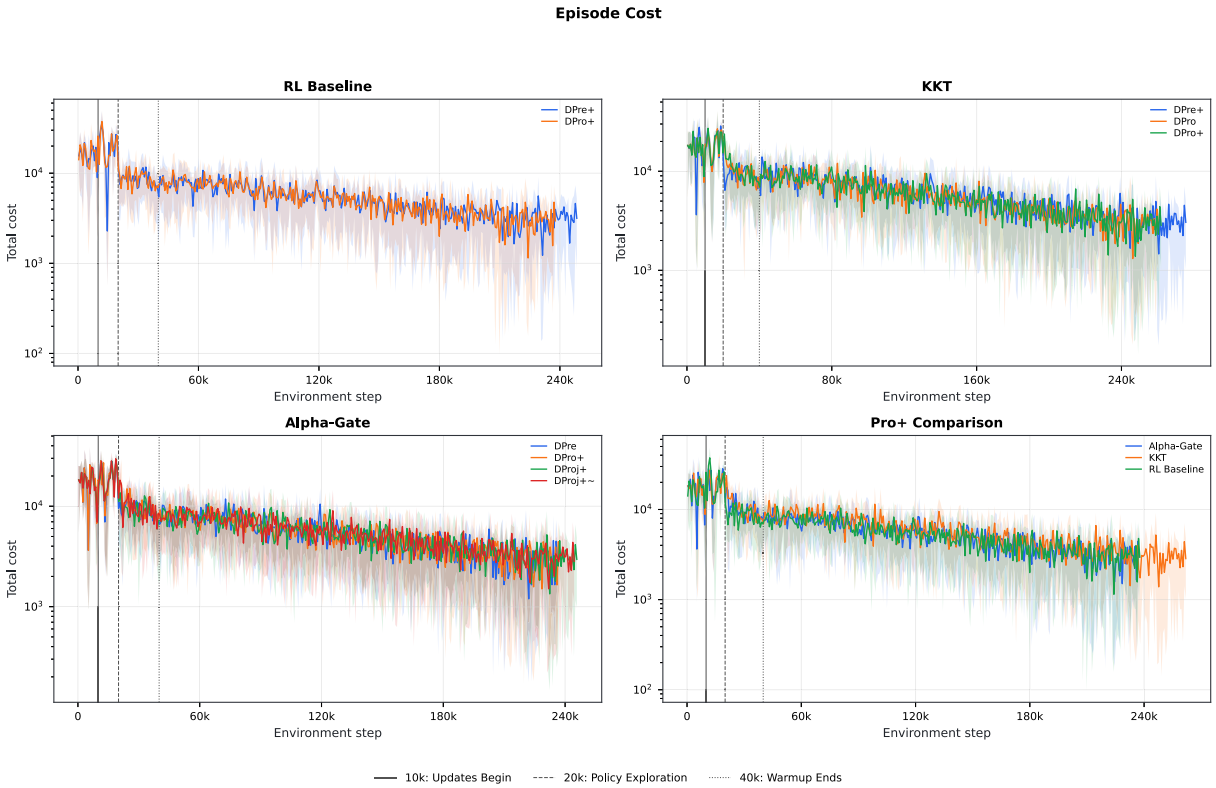

Fig. 46. Semi-Log plot of the total cost of the exploration phase episode at each environment step during the training run. To improve readability, the x-axis is binned into 350 equal sized bins to allow for the banded form of the plots to be shown. The banded form includes the median curve along with a semi-transparent banded region that spans the 25–75th percentiles.

Table 5
Compute time for TD3-based RL training runs. GPU-hours assume one RTX A2000 per training run.

Controller	Wall-clock time	Compute-hours
RL Baseline DPre+	7h 31m 21s	7.52 GPU-h
RL Baseline DPro+	5h 09m 51s	5.16 GPU-h
KKT DPre+	22h 05m 29s	22.09 GPU-h
KKT DPro	21h 09m 37s	21.16 GPU-h
KKT DPro+	21h 20m 34s	21.34 GPU-h
Alpha-Gate DPre	13h 42m 20s	13.71 GPU-h
Alpha-Gate DPro+	11h 27m 55s	11.47 GPU-h
Alpha-Gate DProj+	10h 32m 31s	10.54 GPU-h
Alpha-Gate DProj+ ~	9h 43m 31s	9.73 GPU-h

performance of the networks is secondary to the pros and cons of the frameworks themselves.

Remark 23. Although the gradient analysis may indicate that the constraint informed variants of the RL runs are at a heightened risk of unstable training, the similar gradient norms seen in the Critic indicates that the Actors discrepancies can be partially explained through the different NN architectures used between the non-CENNC and CENNC cases. Thus, the CENNC framework itself cannot be ruled as the sole cause of this behavior.

Remark 24. The + penalty is always active for variants that utilize it. This is not treated the same as the Alpha penalty, which is only active after the warm-up phase ends at step 40,000 for applicable frameworks.

Remark 25. Despite the fact that Alpha-Gate is not tested on the `DPre+` configuration, the `DPre` variant is referred to as `DPre+` in the above discussion for the sake of simplicity. In practice, the + has no impact on the model and the variation between the + and non-+ variants for a given framework is caused by the non-deterministic aspects of Python machine learning packages.

5.6. Closed-loop simulation results

To evaluate these frameworks and their variants, the closed-loop system was simulated using the same 120,000 uniformly sampled initial states for all configurations. Each trajectory is then simulated for a fixed 200 steps, which allows for the controllers to be evaluated in regions outside their training distributions at both extremes. The sole exception to this evaluation process was the LMPC controller, which was only evaluated using the first 1,215 initial states due to the significantly higher computational time that was required. For consistency, the cost metrics are reported both as the raw cumulative value for the given trajectories as well as a relative metric with respect to the LMPC cost. The IL runs are done twice, once without any fallback, to observe the impact of the lack of fallback on the controllers performance. The cost metrics from this evaluation run are summarized in Table 6 along with timing metrics.

In terms of cost, the reference controller is observed to be the worst performing controller by a large margin. This is fine, as the reference controller is not designed with cost optimality in mind and serves solely for ensuring stability of the system. The IL controllers marginally outperform the LMPC in this evaluation run, but it is important to keep in mind that the LMPC is under-sampled relative to the NN and reference evaluations and so the metrics are approximate. Fig. 47 demonstrates that if we only consider the samples that the LMPC is evaluated on, that all controllers are effectively equal or worse on average. The IL controllers struggle to maintain their fit for points far from the origin, and the RL controllers struggle to generalize their performance for points near the origin.

Table 6
Closed-loop evaluation summary for all evaluated controllers.

Controller	Case	Mean cost				Mean wall time (s)[b]	Mean wall time to $V \leq 1$ (s)[a]	Mean wall time per step (s)[a]
		Full trajectory		Early stopping				
		Value	% vs. LMPC	Value	% vs. LMPC			
Reference	P	1.083×10^4	41.6%	1.077×10^4	41.5%	3.188	1.106	0.01594
	LMPC	7,652	0%	7,611	0%	805.4	109.1	4.027
IL Baseline	Ω_{50} No Safety	7,614	−0.49%	7,572	−0.51%	0.5316	0.07286	0.002658
	Ω_{50} Safety	7,614	−0.49%	7,572	−0.51%	0.5311	0.07278	0.002656
	Ω_{120} No Safety	7,549	−1.34%	7,507	−1.36%	0.5311	0.07229	0.002656
	Ω_{120} Safety	7,595	−0.74%	7,553	−0.76%	0.5304	0.07231	0.002652
RL Baseline	DPre+	7,800	1.93%	7,655	0.57%	3.506	0.3891	0.01753
	DPro+	7,763	1.46%	7,462	−1.96%	3.597	0.3274	0.01799
KKT	DPre+	8,167	6.74%	7,622	0.15%	2.21	0.2157	0.01105
	DPro	9,247	20.9%	7,485	−1.67%	2.202	0.2126	0.01101
	DPro+	7,938	3.74%	7,511	−1.31%	2.209	0.2194	0.01104
Alpha-Gate	DPre	8,054	5.25%	7,577	−0.44%	0.6917	0.06631	0.003459
	DPro+	7,708	0.74%	7,466	−1.91%	4.107	0.3687	0.02053
	DProj+	7,743	1.20%	7,472	−1.83%	3.997	0.3611	0.01999
	DProj+ ∼	7,768	1.52%	7,487	−1.63%	0.6907	0.06044	0.003454

[a] Per-step timing and wall time to reach $V \leq 1$ are estimated from per-trajectory wall time divided by the number of executed steps; per-step timers were not stored.
[b] Evaluation wall-clock times were measured on a local workstation using CPU-parallel trajectory evaluation. 12 CPU workers on an Intel Core i7-10700K CPU @ 3.80 GHz (8 physical cores, 16 hardware threads) were used during evaluation in all cases.

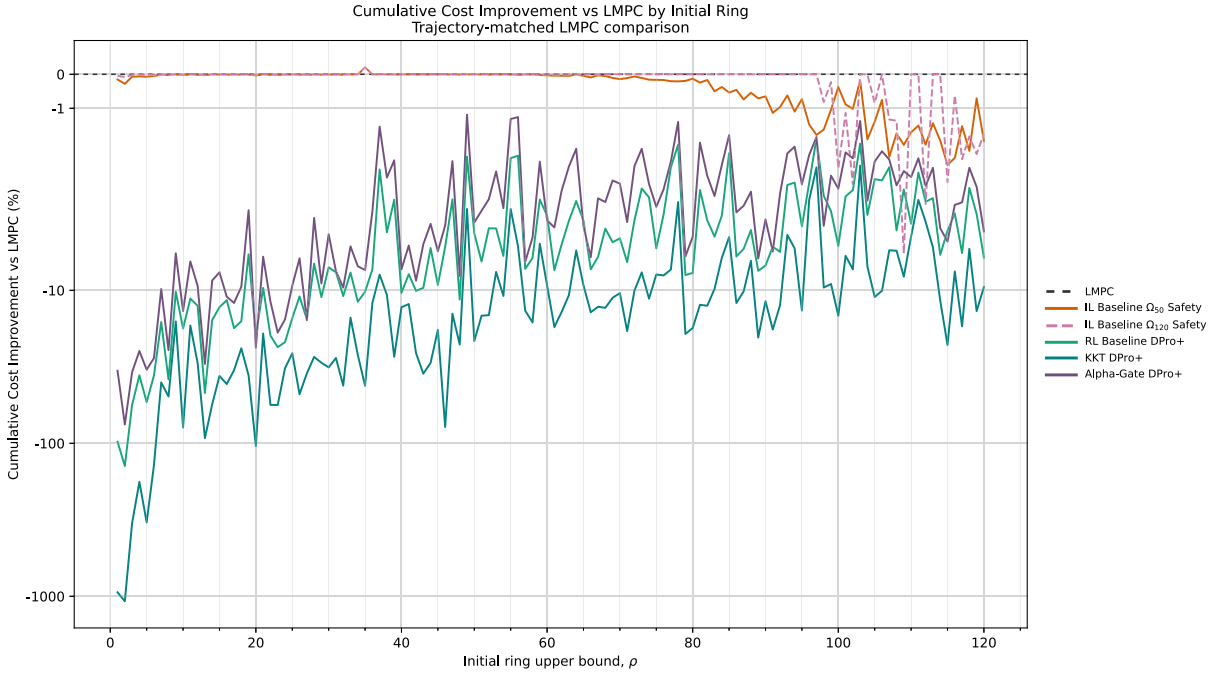

Fig. 47. Average trajectory cost difference relative to the LMPC for the various evaluated controllers. Costs are compared only for the 1,215 trajectories that LMPC is evaluated on. These trajectories are grouped into rings of width 1 whose average trajectory cost is used as the reference value for the resulting percent differences.

Additionally, the existence of fallback did not noticeably impact the IL controllers cost compared to the form of the controller with no stability guarantees. Of the RL controllers, the Baseline and Alpha-Gate are of comparable performance with the Alpha-Gate seemingly outperforming the RL Baseline in non-`DPre` configurations. KKT performed the worst of the frameworks, and also demonstrated a significant vulnerability to noise. As has been noted previously, + and non-+ variants are functionally equivalent barring numerical precision leaks, and plots of these leaks such as Fig. 41 indicate that the gradient impact is negligible. Despite this, the `DPro` and `DPro+` variants of the KKT framework have significantly different performance. In fact, the training metrics as a whole would otherwise indicate that the two controllers are expected to perform identically. This is actually true for the region that was sampled, but falls apart when generalization is considered. Consider the mean trajectory cost term. If it is split into the pre-terminal region and post-terminal region cumulative costs, it is seen that the `DPro` configuration has a pre-terminal cost of 7,511 and a post-terminal cost of 1,736.5 as opposed to the respective `DPro+` configuration's values of 7,535.1 and 402.9. Because the training was done in a limited ring of the stability region, but evaluation was done in the full stability region, the closed-loop performance mismatch that is seen between these two comes entirely from these untrained regions. In other words, the generalization of the KKT approach is unstable.

In terms of timing, because the networks used in this example are fairly small, the evaluation phase is a CPU-limited task. During the evaluation phase, the Python code used was gradually improved to

Table 7
Seed-421 effective fallback summary for evaluated non-reference controllers over 120,000 closed-loop trajectories of at most 200 steps initialized in Ω_{120}.

Controller	Case	Fallback[a]					
		$1 < V \leq 50$		$V > 50$		$V \leq 1$	
		Trajectories with fallback	Fallback count (region %)[b]	Trajectories with fallback	Fallback count (region %)[b]	Trajectories with fallback	Fallback count (region %)[b]
IL Baseline	Ω_{50} Safety	0	0 (0.00%)	0	0 (0.00%)	120,000	12,972,775 (62.63%)
	Ω_{120} Safety	0	0 (0.00%)	2,705	10,762 (3.10%)	688	2,831 (0.01%)
RL Baseline	DPre+	2,810	5,611 (0.24%)	833	1,599 (0.56%)	120,000	11,273,184 (52.83%)
	DPro+	0	0 (0.00%)	0	0 (0.00%)	119,801	7,135,202 (32.71%)
KKT	DPre+	0	0 (0.00%)	2,771	4,027 (1.18%)	120,000	17,651,119 (81.59%)
	DPro	0	0 (0.00%)	9,870	24,245 (5.71%)	120,000	9,791,073 (46.60%)
	DPro+	0	0 (0.00%)	17,165	56,041 (12.33%)	120,000	18,535,915 (86.16%)
Alpha-Gate	DPre	4,528	4,809 (0.24%)	442	442 (0.16%)	120,000	15,178,312 (69.94%)
	DPro+	0	0 (0.00%)	0	0 (0.00%)	60,177	8,710,409 (39.87%)
	DProj+	0	0 (0.00%)	0	0 (0.00%)	120,000	8,721,428 (39.95%)
	DProj+ ~	0	0 (0.00%)	0	0 (0.00%)	120,000	18,382,379 (83.94%)

[a] Fallback denotes effective fallback: non-CENNC frameworks exhibit no internal fallback, and CENNC frameworks exhibit no external fallback, thus external fallback is counted for non-CENNC frameworks and internal fallback is counted for CENNC frameworks (as indicated by the final value of α).
[b] Region percentages are fallback event counts divided by the cumulative number of valid steps taken in the indicated Lyapunov region, not by all evaluated state-steps.

optimize for this, hence why some runs take around 0.01 s whereas others are closer to 0.003 s. More generally, the NN and P runs exhibited roughly the same per-step wall time which positions all of the methods as significantly more computationally efficient than LMPC, which was roughly 2 to 3 orders of magnitude slower per step. Additionally, it is seen that when we look at the per-trajectory timings, that the NN methods outperform the P controller due to the stabilizing controllers generally slower convergence behavior. As a result, the per-trajectory improvement in computation time compared to the LMPC is closer to 3 to 4 orders of magnitude less time for the NN methods and 1 to 2 orders of magnitude less time for the NN methods compared to the reference controller. In all cases, the same number of workers was used for the same hardware, and so all runs may exhibit improved or worsened performance depending on the configuration used during deployment. It is more conclusive to compare the relative performance of these controllers. Thus, all runs with the exception of the LMPC framework are observed to be of comparable computational performance.

Fallback metrics are another important data point to consider, and these results are summarized in Table 7. The results are split into 3 regions, the region in which the controllers were trained on (with the exception of the full range IL run), and the remaining 2 unsampled regions during training. As seen, the IL runs have good generalization, but the extent of this depends on the sampling method. The IL run that was sampled on the smaller region generalized well to the terminal region but poorly for the outer region whereas the full range IL run generalized well to the terminal region but suffered regressions in the generalization to the outer region. As for the RL runs, the `DPre` variants are poorly behaved in the RL Baseline and Alpha-Gate run, but seemingly well behaved in the KKT run. As noted before, the KKT run exhibits a noisy generalization behavior, and thus it consistently struggles to generalize to the unseen regions. In all cases, the KKT generalized better to the outer region than the terminal, and the `DPro` variant is seen to require less internal fallback in both regions than the `DPro+` variant despite its worse closed-loop cost performance. The Alpha-Gate and RL Baseline both generalize well to the outer region, and both struggle to do so for the terminal region. Unlike KKT, the terminal region generalization is roughly comparable between the Alpha-Gate and RL Baseline frameworks. The `DProj+~` variant generalizes notably worse to the terminal region than the non-STE form, which itself is seen to perform worse than the `DPro+` variant. Thus, the `DPro+` functions as a solid baseline for comparisons between the frameworks. It is important to note that although the CENNC frameworks seemingly have high rates of fallback, these fallback metrics are an effective metric that include both internal and external fallback. In reality, the purpose of including the CENNC frameworks in this table is to demonstrate how heavily these controllers rely on internal fallback for the purposes of generalization, and it is seen that it is comparable to the reliance of external fallback in the RL Baseline case but better than the reliance in the IL case for sampling in a similar region. Thus, the CENNC framework enables models to generalize constraint handling in a manner that is comparable to the action handling modality but with the added benefit of not requiring external fallback.

To visualize the closed-loop performance, we take a representative sample from the evaluation dataset. In particular, this sample is taken from the outermost ring where the Lyapunov function is between 119 and 120, and the point is chosen to be in the region of the state-space where the IL networks struggled to fit, indicating difficult regional dynamics. Fig. 48 is a multi-figure plot that demonstrates this representative initial point and the corresponding closed-loop trajectories for the `DPro+` variant of each controller. The reference controller can be seen to have a far slower convergence to the origin than all other controllers which results in a larger cumulative cost. The controllers exhibit very similar trajectories with the exception of the KKT controller that not only begins with an action that is noticeably different from the other controllers but also exhibits occasional sharp action spikes that results in delayed convergence and a higher cumulative cost. The RL Baseline controller also exhibits similar spikes, but to a significantly lesser degree and in a manner that does not lead to the same oscillatory behavior as the KKT run. Of the RL controllers, the Alpha-Gate framework performed the best for this sample, but performed worse than the IL baseline. Additionally, the Alpha-Gate approach converged to a notably larger offset from the origin than the IL controllers. Fig. 49 demonstrates the same trajectory, but under early termination when entering the terminal region. In this scenario, the Alpha-Gate and RL Baseline controllers both outperform the LMPC and IL controllers. Because the RL runs are trained with respect to a reward function that terminates upon entering the terminal region, this result replicates the behavior seen in the early stopping section of Table 6; the RL runs are optimized for early stopping and will outperform LMPC in this context. This result is by design, since the LMPC is inherently a finite horizon optimization problem and thus we formulated the infinite horizon problem into a variable length finite horizon optimization problem with respect to the terminal region. Thus, the cost performance is not the priority and deviations within the single-digit percentage in cost is not of concern. Instead, the importance is that constraint handling is successfully internalized in the Alpha-Gate

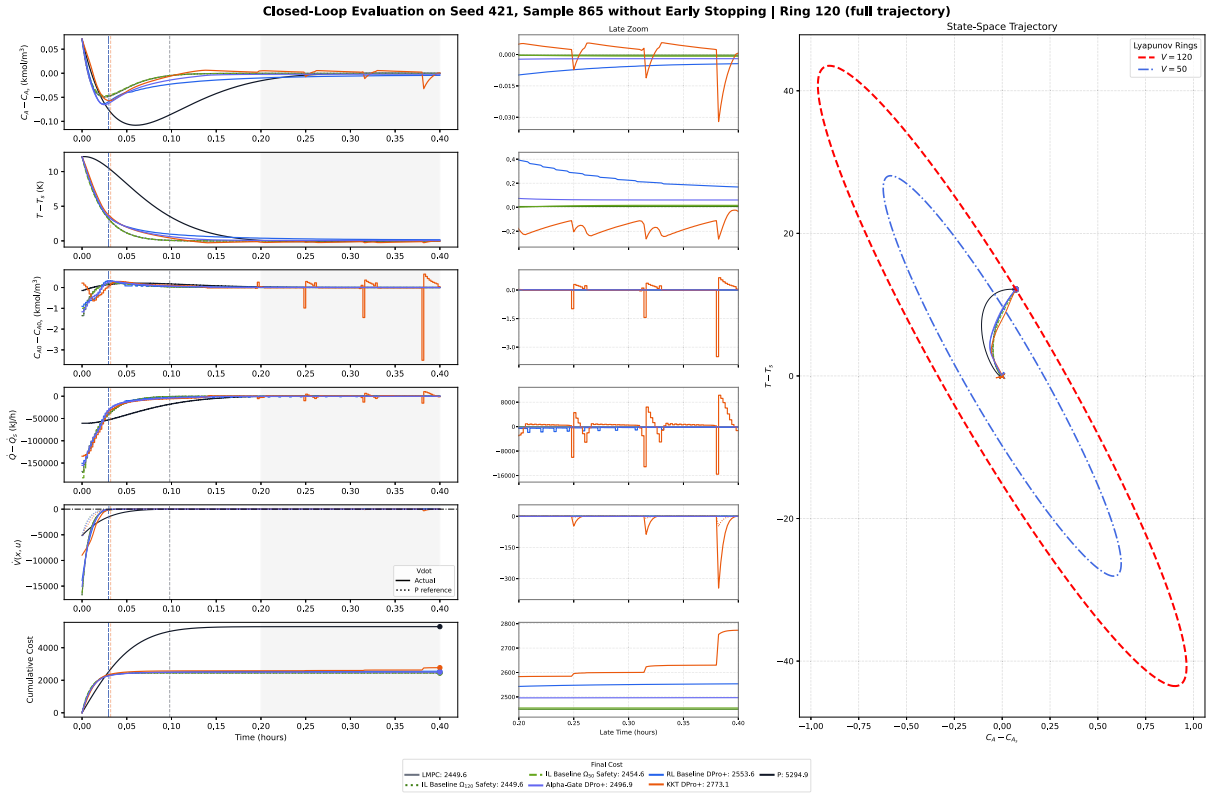

Fig. 48. Closed-loop trajectories of the states, actions, cumulative cost, and Lyapunov function's time derivative for a representative sample on the outermost Lyapunov ring. The closed-loop simulation is done over 200 steps without early termination. For clarity, the individual metric plots are split into an overall time axis and a zoomed in late time axis indicated by the shaded regions in the overall time axis plot. The resulting 2D state-space trajectory is provided in the rightmost figure where all controllers start from the same initial state indicated by the circle node and terminate after finishing all required steps where the final state is indicated by a cross that is colored to match the corresponding legend entry. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

approach with minimal consequences relative to the KKT approach, which is seen to have limitations in generalizing to unseen data.

Remark 26. As depicted in this work, the alpha-gate CENNC approach explicitly derives constraint information needed for proper constraint enforcement through the first-principles model of the process. In doing so, one may interpret this as a "model-based" implementation of the framework, where the environment dynamics are known through a mathematical model that is used to calculate terms such as $\dot{V}$ explicitly much like how LMPC would handle constraints. More generally, the framework can be designed with a "model-free" implementation where the relevant constraint metrics such as $\dot{V}$ are instead passed into the network as inputs much like how the reference action is done for the presented example.

6. Conclusion

In this work, we explore through first-principles how neural networks can be designed to be both a scalable design to larger networks without gradient collapse and designed to inherently satisfy and learn from constraints. We refer to controllers that satisfy the latter as Constraint Enforcing Neural Network Controllers (CENNC). This CENNC framework exists currently through differentiable Quadratic Programming (QP) layers, but it is demonstrated that this implementation leads to both less stable training dynamics as well as less stable generalizability of the final network to unseen data. We demonstrate that a simpler architecture utilizing the Heaviside step function and Straight-Through Estimators (STEs) is capable of stable training and stable generalizability. The resulting network is referred to as the Alpha-Gate CENNC, and this network is evaluated against a KKT optimal equivalent network to the QP-layer design for a chemical process example. It is further evaluated against a generic RL Baseline controller and an IL Baseline controller with various action handling modalities for the replay buffer regarding how constraint enforcement is applied during training and exploration. From the results it is seen that the Alpha-Gate framework exhibits similar performance to the RL Baseline with external enforcement. In particular, the rate of internal constraint enforcement, the resulting trajectory costs, computation time, and training dynamics are all similar. Thus, the Alpha-Gate framework is demonstrated to be a simple and efficient internalization of constraint handling that exhibits stable training, removes the need for external constraint enforcement, and allows the network to learn to respect constraints.

The findings of this work are limited to process systems in which the stabilizing reference control law and corresponding Lyapunov function are known explicitly. Additionally, the impact of modeling errors, disturbances, and data sparsity are not explored. As such, future work should involve demonstrating the impact of these metrics as well as means to counteract their impact. It is also worth exploring data driven approaches to applying the framework for model-free scenarios including deriving the reference control law, Lyapunov function, and process dynamics. Scalability of the framework to high dimensional processes is also not explored and serves as a point of future research.

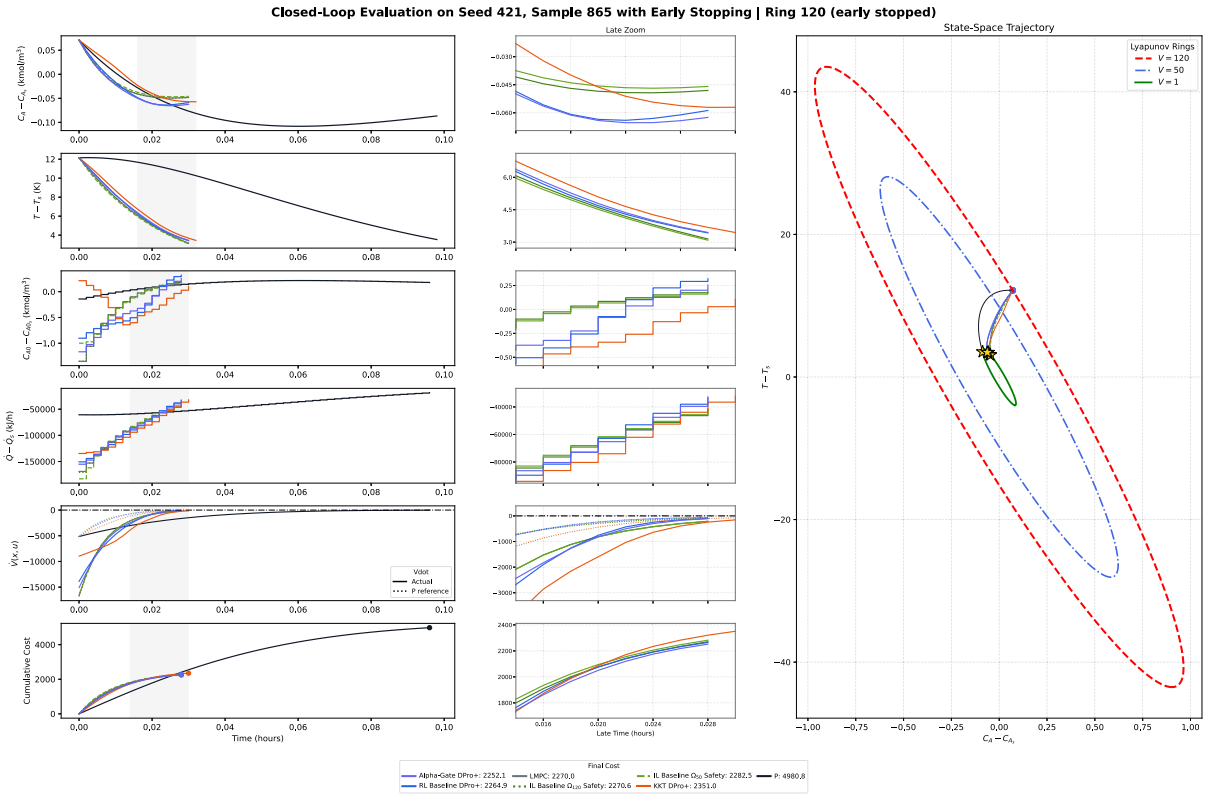

Fig. 49. Closed-loop trajectories of the states, actions, cumulative cost, and Lyapunov function's time derivative for a representative sample on the outermost Lyapunov ring. The closed-loop simulation is done over 200 steps with early termination upon entering the terminal region bounded by the level set $V = 1$. For clarity, the individual metric plots are split into an overall time axis and a zoomed in late time axis indicated by the shaded regions in the overall time axis plot. The resulting 2D state-space trajectory is provided in the rightmost figure where all controllers start from the same initial state indicated by the circle node and terminate after finishing all required steps where the final state is indicated by a cross that is colored to match the corresponding legend entry. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

CRediT authorship contribution statement

Arthur Khodaverdian: Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Xiaodong Cui:** Writing – original draft, Methodology, Conceptualization. **Panagiotis D. Christofides:** Writing – review & editing, Supervision, Conceptualization.

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.

Acknowledgments

Financial support from the National Science Foundation, CBET-2140506, is gratefully acknowledged. A.K. also acknowledges financial support from the 2026 Amazon AI Ph.D. Fellowship Program.

References

Ablett, T., 2025. Addressing Distribution Shift in Robotic Imitation Learning (Ph.D. thesis). University of Toronto, Toronto, Ontario, Canada.

Achiam, J., 2018. Spinning up in deep reinforcement learning. https://spinningup.openai.com/en/latest/.

Adabag, E., Atal, M., Gerard, W., Plancher, B., 2024. MPCGPU: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the GPU. In: Proceedings of International Conference on Robotics and Automation. Yokohama, Japan, pp. 9787–9794.

Agrawal, A., Amos, B., Barratt, S., Boyd, S., Diamond, S., Kolter, Z., 2019. Differentiable convex optimization layers. In: Advances in Neural Information Processing Systems.

Ahn, K., Mhammedi, Z., Mania, H., Hong, Z.-W., Jadbabaie, A., 2022. Model predictive control via on-policy imitation learning. arXiv preprint arXiv:2210.09206.

Alora, J.I., Pabon, L.A., Köhler, J., Cenedese, M., Schmerling, E., Zeilinger, M.N., Haller, G., Pavone, M., 2023. Robust nonlinear reduced-order model predictive control. In: Proceedings of 62nd Conference on Decision and Control. Marina Bay Sands, Singapore, pp. 4798–4805.

Alsmeier, H., Savchenko, A., Findeisen, R., 2024. Neural horizon model predictive control - Increasing computational efficiency with neural networks. In: Proceedings of the American Control Conference. Toronto, Canada, pp. 1646–1651.

Andersson, J.A.E., Gillis, J., Horn, G., Rawlings, J.B., Diehl, M., 2019. CasADi – A software framework for nonlinear optimization and optimal control. Math. Program. Comput. 11 (1), 1–36.

Betts, J.T., 1998. Survey of numerical methods for trajectory optimization. J. Guid. Control Dyn. 21 (2), 193–207.

Bonzanini, A.D., Paulson, J.A., Graves, D.B., Mesbah, A., 2020. Toward safe dose delivery in plasma medicine using projected neural network-based fast approximate NMPC. IFAC-PapersOnLine 53, 5279–5285.

Fujimoto, S., van Hoof, H., Meger, D., 2018. Addressing function approximation error in actor-critic methods. In: Proceedings of the 35th International Conference on Machine Learning. In: Proceedings of Machine Learning Research, vol. 80, PMLR, pp. 1587–1596.

Gill, P.E., Murray, W., Picken, S.M., Wright, M.H., 1979. The design and structure of a Fortran program library for optimization. ACM Trans. Math. Softw. 5 (3), 259–283.

Gonzalez, C., Asadi, H., Kooijman, L., Lim, C.P., 2024. Neural networks for fast optimisation in model predictive control: A review. arXiv preprint arXiv:2309.02668.

Gordon, D.C., Winkler, A., Bedei, J., Schaber, P., Pischinger, S., Andert, J., Koch, C.R., 2024. Introducing a deep neural network-based model predictive control framework for rapid controller implementation. In: Proceedings of American Control Conference. Toronto, Canada, pp. 5232–5237.

de Haan, P., Jayaraman, D., Levine, S., 2019. Causal confusion in imitation learning. arXiv:1905.11979.

Han, T., Bao, Y., Mehta, B., Guo, G., Vishwakarma, A., Kang, E., Jung, S., Scalise, R., Zhou, J., Xu, B., Boots, B., 2026. Model predictive adversarial imitation learning for planning from observation. arXiv:2507.21533.

Hornik, K., Stinchcombe, M., White, H., 1989. Multilayer feedforward networks are universal approximators. Neural Netw. 2 (5), 359–366.

Hoyer, P.O., 2004. Non-negative matrix factorization with sparseness constraints. arXiv: cs/0408058.

Kasaura, K., Miura, S., Kozuno, T., Yonetani, R., Hoshino, K., Hosoe, Y., 2023. Benchmarking actor-critic deep reinforcement learning algorithms for robotics control with action constraints. IEEE Robot. Autom. Lett. 8 (8), 4449–4456.

Khodaverdian, A., Cui, X., Christofides, P.D., 2025a. Utilizing reinforcement learning in feedback control of nonlinear processes with stability guarantees. Digit. Chem. Eng. 17, 100277.

Khodaverdian, A., Gohil, D., Christofides, P.D., 2025b. Neural network implementation of model predictive control with stability guarantees. Digit. Chem. Eng. 16, 100262.

Khodaverdian, A., Gohil, D., Christofides, P.D., 2026. Uniting neural network-based control and model predictive control: Application to a large-scale nonlinear process. Comput. Chem. Eng. 204, 109396.

Kraft, D., 1988. A Software Package for Sequential Quadratic Programming. In: Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht, Wiss. Berichtswesen d. DFVLR.

Lee, S.-W., Kang, X., Kuo, Y.-L., 2025. Diff-DAgger: Uncertainty estimation with diffusion policy for robotic manipulation. arXiv:2410.14868.

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D., 2016. Continuous control with deep reinforcement learning. In: International Conference on Learning Representations.

Lucia, S., Karg, B., 2018. A deep learning-based approach to robust nonlinear model predictive control. IFAC-PapersOnLine 51, 511–516.

Lyle, C., Zheng, Z., Khetarpal, K., Martens, J., van Hasselt, H., Pascanu, R., Dabney, W., 2024. Normalization and effective learning rates in reinforcement learning. In: Advances in Neural Information Processing Systems.

Macmurray, J., Himmelblau, D., 1995. Modeling and control of a packed distillation column using artificial neural networks. Comput. Chem. Eng. 19, 1077–1088.

Meng, D., Chu, H., Tian, M., Gao, B., Chen, H., 2024. Real-time high-precision nonlinear tracking control of autonomous vehicles using fast iterative model predictive control. IEEE Trans. Intell. Veh. 9, 3644–3657.

Mhaskar, P., El-Farra, N.H., Christofides, P.D., 2006. Stabilization of nonlinear systems with state and control constraints using Lyapunov-based predictive control. Syst. Control Lett. 55, 650–659.

Pardalos, P.M., Vavasis, S.A., 1991. Quadratic programming with one negative eigenvalue is NP-hard. J. Global Optim. 1, 15–22.

Patel, R., Bhartiya, S., Gudi, R.D., 2025. Neural network-based model predictive control framework incorporating first-principles knowledge for process systems. Ind. Eng. Chem. Res. 64 (18), 9287–9302.

Peng, Y., Yan, H., Rao, K., Yang, P., Lv, Y., 2024. Distributed model predictive control for unmanned aerial vehicles and vehicle platoon systems: a review. Intell. Robot. 4, 293–317.

Qin, S.J., Badgwell, T.A., 2003. A survey of industrial model predictive control technology. Control Eng. Pract. 11, 733–764.

Ren, Y.M., Alhajeri, M.S., Luo, J., Chen, S., Abdullah, F., Wu, Z., Christofides, P.D., 2022. A tutorial review of neural network modeling approaches for model predictive control. Comput. Chem. Eng. 165, 107956.

Ross, S., Bagnell, D., 2010. Efficient reductions for imitation learning. In: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics. Chia Laguna Resort, Sardinia, Italy, pp. 661–668.

Ross, S., Gordon, G.J., Bagnell, J.A., 2011. A reduction of imitation learning and structured prediction to no-regret online learning. arXiv:1011.0686.

Ruan, K., Zhang, J., Di, X., Bareinboim, E., 2023. Causal imitation learning via inverse reinforcement learning. In: The Eleventh International Conference on Learning Representations.

Sutton, R., Barto, A., 2020. Reinforcement Learning: An Introduction. [Online]. Available: http://www.incompleteideas.net/book/RLbook2020.pdf.

Tomasetto, M., Braghin, F., Manzoni, A., 2025. Latent feedback control of distributed systems in multiple scenarios through deep learning-based reduced order models. Comput. Methods Appl. Mech. Engrg. 442, 118030.

Tordesillas, J., How, J.P., Hutter, M., 2023. RAYEN: Imposition of hard convex constraints on neural networks. arXiv preprint arXiv:2307.08336.

Wang, Y., Wu, Z., 2024a. Control Lyapunov-barrier function-based safe reinforcement learning for nonlinear optimal control. AIChE J. 70 (3), e18306.

Wang, Y., Wu, Z., 2024b. Physics-informed reinforcement learning for optimal control of nonlinear systems. AIChE J. 70 (10), e18542.

Wolfe, C.R., 2026. Continual Learning with RL for LLMs. Deep (Learning) Focus.

Wu, Z., Tran, A., Rincon, D., Christofides, P.D., 2019. Machine learning-based predictive control of nonlinear processes. Part I: Theory. AIChE J. 65, e16729.

Yaren, T., Kizir, S., 2025. Real-time nonlinear model predictive control of a robotic arm using spatial operator algebra theory. J. Field Robot. 42, 2337–2354.

Zarrouki, B., Nunes, J., Betz, J., 2024. R^2NMPC: A real-time reduced robustified nonlinear model predictive control with ellipsoidal uncertainty sets for autonomous vehicle motion control. IFAC-PapersOnLine 58, 309–316.

Zhou, W., Li, W., 2024. Rethinking inverse reinforcement learning: from data alignment to task alignment. arXiv:2410.23680.

Zolna, K., Reed, S., Novikov, A., Colmenarejo, S.G., Budden, D., Cabi, S., Denil, M., de Freitas, N., Wang, Z., 2020. Task-relevant adversarial imitation learning. arXiv: 1910.01077.