**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

# Benchmarking Actor-Critic Deep Reinforcement Learning Algorithms for Robotics Control with Action Constraints

Kazumi Kasaura*[1], Shuwa Miura*[2], Tadashi Kozuno[1], Ryo Yonetani[1], Kenta Hoshino[3], Yohei Hosoe[4]

***Abstract*—This study presents a benchmark for evaluating action-constrained reinforcement learning (RL) algorithms. In action-constrained RL, each action taken by the learning system must comply with certain constraints. These constraints are crucial for ensuring the feasibility and safety of actions in real-world systems. We evaluate existing algorithms and their novel variants across multiple robotics control environments, encompassing multiple action constraint types. Our evaluation provides the first in-depth perspective of the field, revealing surprising insights, including the effectiveness of a straightforward baseline approach. The benchmark problems and associated code utilized in our experiments are made available online at github.com/omron-sinicx/action-constrained-RL-benchmark for further research and development.**

***Index Terms*—reinforcement learning, action constraints, safety**

## I. Introduction

ACTION-CONSTRAINED reinforcement learning (RL) imposes explicit constraints on actions taken by the learning system. These constraints can come from a variety of sources, such as physical limitations of robots (*e.g.*, torque or power limits) [1] and safety considerations. In manufacturing applications, for example, action constraints can prevent robot arms from hitting obstacles [2] or moving beyond designated boundaries [3]. Other examples include collision avoidance for autonomous vehicles [4] and maintenance for energy-efficient building operations [5]. For real systems, it is important that each action should satisfy constraints throughout the training process, not just at the end, to guarantee its feasibility [2].

Many works have explored how to introduce action constraints to existing deep RL algorithms [2], [5]–[8]. Deep RL algorithms, which use neural network policies, have achieved success in various continuous control tasks [9], [10]. However, without explicit constraints, agents can try to execute infeasible actions in the environment. Existing work has addressed this problem by projecting the outputs of policies onto feasible actions, ensuring that constraints are satisfied before actions are executed. Examples of algorithms that adopt this approach encompass differentiable optimization layers [2], [4]–[7], [11], Neural Frank-Wolfe Policy Optimization (NFWPO) [8], and $\alpha$-projection [12]. Despite the growing interest in action-constrained RL, a comprehensive comparison of these algorithms has not been conducted.

In this paper, we present the first benchmark study for the existing action-constrained RL algorithms. Our primary contribution is the evaluation of action-constrained RL algorithms in terms of learning performance and computational time requirements, laying the foundation for future research in this domain. To this end, our evaluation uses various simulated, and thus easy-to-reproduce, robotics control tasks from MuJoCo [13] and PyBullet-Gym [14] on OpenAI gym [15] and evaluate existing work with several different types of action constraints. Our evaluation centers on off-policy actor-critic deep RL algorithms, such as Deep Deterministic Policy Gradients (DDPG) [9] and Soft Actor-Critic (SAC) [10], due to their superior sample efficiency. Actor-critic algorithms [16] involve training both an actor and a critic. The actor adjusts policy parameters using the policy gradient, while the critic learns the Q-function.

In addition to evaluating existing action-constrained RL algorithms, we also introduce several variants of these algorithms in our evaluation. One of these variants is a simple method that treats action constraints as part of the state transition of the environment, serving as a baseline for comparison with other approaches to action constraints. We also propose an alternative action mapping method called radial squashing, which can be seen as a natural variant of $\alpha$-projection. In our evaluation, we use a variant of DDPG called Twin Delayed DDPG (TD3) [17] and SAC as the base deep RL algorithms. DDPG variants are common choices for this type of problem [7], [8], while the use of action constraints with SAC has not been previously explored and requires nontrivial modifications to the original algorithm. We describe how to incorporate action constraints into SAC.

Our experimental results suggest that a simple approach that trains the critic with pre-projected actions is empirically shown to achieve performance comparable to that of specialized algorithms. Moreover, the results also show that alternative mapping techniques ($\alpha$-projection and radial squashing) can

Manuscript received: January, 23, 2023; Revised April, 11, 2023; Accepted May, 15, 2023.

This paper was recommended for publication by Aleksandra Faust upon evaluation of the Associate Editor and Reviewers' comments.

[1] KK, TK, and RY are with OMRON SINIC X Corporation, Hongo, Bunkyo-ku, Tokyo, Japan. {kazumi.kasaura, tadashi.kozuno, ryo.yonetani}@sinicx.com. [2] SM is with Manning College of Information and Computer Sciences University of Massachusetts Amherst, Amherst, Massachusetts. This work was done while he was a research intern at OMRON SINIC X Corporation. smiura@cs.umass.edu. [3] KH is with Graduate School of Informatics, Kyoto University, Sakyo-ku, Kyoto, Japan. hoshino@i.kyoto-u.ac.jp. [4] YH is with Graduate School of Engineering, Kyoto University, Nishikyo-ku, Kyoto, Japan. hosoe@kuee.kyoto-u.ac.jp. *Equal contribution.

Digital Object Identifier (DOI): see top of this page.

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

achieve competitive performance compared to the existing algorithms, while requiring less computation time. On the other hand, the results show that the use of differentiable optimization layers, a common method for action-constrained RL, does not outperform the simple baseline and can take a long time to run.

## II. Background

### A. Action-Constrained Reinforcement Learning

We consider an RL problem, where an agent interacts with its environment modeled by a Markov decision process (MDP). An MDP is represented by a tuple $M = \langle \mathcal{S}, \mathcal{A}, T, d_0, d_R, \gamma \rangle$, where $\mathcal{S}$ and $\mathcal{A}$ in this work are continuous state and action spaces, respectively. $T$ is a conditional density function describing the dynamics of the environment ($S_{t+1} \sim T(S_t, A_t)$), and $d_0$ is an initial state distribution. $d_R$ describes how rewards are generated ($R_t \sim d_R(S_t, A_t, S_{t+1})$), and $\gamma$ is a discount factor. Given the current state, an agent determines what action to take based on its *policy* parameterized by $\theta$, which is either deterministic (represented by $\mu_\theta$; mapping a state to an action) or stochastic ($\pi_\theta$; mapping the state to the probability distribution on $\mathcal{A}$). The goal of RL is to find a policy that maximizes the expected discounted return $J(\pi_\theta) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R_t | d_0, \pi_\theta]$. The *action value function* or *Q-function* for policy $\pi$ is defined as $Q^\pi(s,a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R_t | S_0 = s, A_0 = a, \pi]$.

In this paper, we consider action-constrained RL problems, where for each state $s \in S$, there is a feasible set of actions $\mathcal{A}_s \subseteq \mathcal{A}$. As in most previous work, we assume $\mathcal{A}_s$ to be known apriori and characterized by linear [2], [7] or convex constraints [5], [8]. Note that in action-constrained RL, actions are required to satisfy constraints throughout training.[1] Action constraints can arise from a variety of sources such as physical limitations of robots [1]. Such constraints can also be used to guarantee the safety of the system. For example, several previous studies have used Control Barrier Functions (CBF) [19] as action constraints [4], [20].

*Box Constraints:* Existing deep RL algorithms such as DDPG and SAC, by default, are limited to handle a specific type of action constraints called *box constraints*, which have the form $a_i \in [-a_i^{\max}, a_i^{\max}]$ for each $i$th dimension of the action space. Input limits like box constraints are almost ubiquitous in continuous control tasks as used in all control tasks from MuJoCo [13] in OpenAI gym [15]. Existing implementations of deep RL such as [21] commonly handle such box constraints using a hyperbolic tangent function ($\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$) as the final activation layer of policies, which we refer to as *squashing*. As the outputs of $\tanh$ range from $-1$ to 1, we can enforce the box constraints by scaling the outputs by $a_i^{\max}$. While squashing is simple and effective, it cannot handle more complex constraints such as upper-bounds on the weighted sum of $a_i$.

[1]The exception can be found in [18], where constraints are not enforced during training.

### B. Policy Gradient Algorithms

Policy gradient algorithms are widely used for continuous control tasks. These algorithms adjust policy parameters $\theta$ using the gradient estimate of the objective function $J(\pi_\theta)$. For deterministic policies $\mu_\theta$, the deterministic policy gradient [22] is given by:

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s \sim \rho^{\mu_\theta}} [\nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s)], \quad (1)$$

where $\rho^{\mu_\theta}$ is a discounted state distribution under $\mu_\theta$.

Deep Deterministic Policy Gradient (DDPG) [9] is a model-free actor-critic algorithm that combines the deterministic policy gradients with function approximation using neural networks. DDPG uses off-policy data (replay buffer) to train the critic ($Q_w$ where $w$ represents the weights) and uses the critic to train the actor ($\mu_\theta$). For each policy update step, a minibatch of transitions ($\{s_i, a_i, r_i, s_i'\}_{i=1}^N$) is sampled from the replay buffer. Then DDPG updates its critic in the following direction:

$$\nabla_w \tfrac{1}{N} \textstyle\sum_{i=1}^N (Q_w(s_i, a_i) - y(s_i', r_i))^2 \quad (2)$$

where $y(r_i, s_i) = r_i + \gamma Q_{w'}(s_i', \mu_{\theta'}(s_i'))$. The target networks, represented by $Q_{w'}$ and $\mu_{\theta'}$, have distinct parameters $w'$ and $\theta'$ and serve to stabilize the training process. DDPG next updates its actor using the following estimate for the policy gradient:

$$\nabla_\theta J(\mu_\theta) \approx \tfrac{1}{N} \textstyle\sum_{i=1}^N \nabla_a Q_w(s_i, a)|_{a=\mu_\theta(s_i)} \nabla_\theta \mu_\theta(s_i) \quad (3)$$

As described above, DDPG handles box constraints via squashing.

Twin Delayed DDPG (TD3) [17] is a refined version of DDPG that addresses the approximation errors often encountered in DDPG. TD3 introduces several improvements over the original DDPG algorithm. One notable enhancement is the *clipped double-Q* trick, which employs the minimum value of two Q-function outputs to decrease overestimation bias, thereby enhancing the stability and performance of the algorithm.

Soft Actor Critic (SAC) [10] is a model-free actor-critic algorithm that optimizes stochastic policies with the following entropy-regularized objective:

$$\mathbb{E}[\textstyle\sum_{t=0}^{\infty} \gamma^t R_t + \alpha H(\pi(\cdot|S_t)) | d_0, \pi] \quad (4)$$

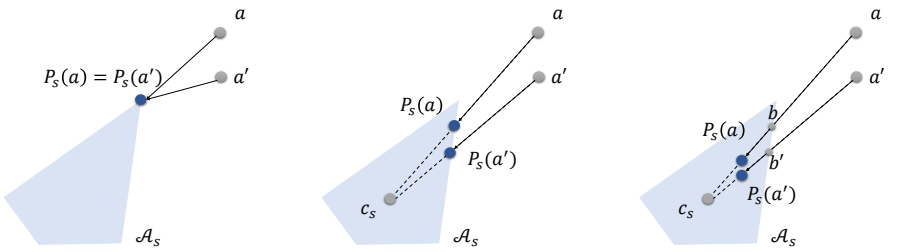
where $H$ is the entropy and $\alpha > 0$ is an entropy coefficient. The entropy regularization term in the SAC algorithm encourages exploration during training by promoting a more stochastic policy. The critic in SAC learns the Q-function incorporating the entropy term (the soft Q-function), which is trained using the clipped double-Q trick as in TD3. The actor is then updated using:

$$\nabla_\theta \tfrac{1}{N} \textstyle\sum_{i=1}^N \min_{j=1,2} Q_{w_j}(s_i, \tilde{a}_\theta(s_i)) - \alpha \pi_\theta(\tilde{a}_\theta(s_i)|s_i) \quad (5)$$

where $\tilde{a}_\theta(s_i)$ is sampled from a squashed Gaussian policy $\pi_\theta(\cdot|s_i)$ using the *reparametrization trick* [10].

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**




(a) Closest point projection (b) $\alpha$-projection (c) Radial squashing

Fig. 1. Illustrations of different mappings used in the paper. $a$ and $a'$ are actions before mappings. $P_s(a)$ and $P_s(a')$ are the corresponding actions after mappings. $\mathcal{A}_s$ is the set of feasible actions. $c_s$ is the Chebyshev center of $\mathcal{A}_s$.

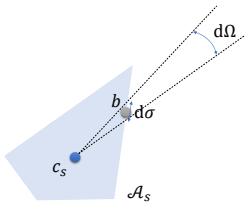



Fig. 2. Illustration of the volume element $\mathrm{d}\sigma$ of $\partial\mathcal{A}_s$ and the differential solid angle $\mathrm{d}\Omega$ corresponding to it

## III. Algorithms for Action-Constrained RL

In this section, we overview algorithms for action-constrained RL. To handle non-trivial constraints, most methods such as differentiable optimization layers [2], [4]–[7], NFWPO [8], and $\alpha$-projection [12], use a mapping to feasible action sets ($P : \mathcal{S} \times \mathcal{A} \to \mathcal{A}_s$). [2] This is because the outputs of policies before the mapping, $\mu_\theta(s)$, do not necessarily satisfy action constraints. Then, rather than the original outputs, the corresponding feasible actions after the mapping, $P(s, \mu_\theta(s))$ or also denoted as $P_s(\mu_\theta(s))$, are executed in the environment.

In what follows, we first present existing algorithms that use the closest projection to feasible actions (Sec. III-A). Then we describe other algorithms that use different mappings in Sec. III-B. Unless specified otherwise, our discussion in this section assumes the use of DDPG.

### A. Algorithms Based on the Closest Point Projection

In this section, we describe a family of algorithms based on the *closest point projection*, *i.e.*, projecting the outputs of policies to the closest feasible actions typically in terms of the Euclidean norm. The closest point projection $P_s$ is defined by the following optimization problem:

$$P_s(\mu_\theta(s)) = \underset{a \in \mathcal{A}_s}{\arg\min} ||a - \mu_\theta(s)||_2. \tag{6}$$

The problem is quadratic programming for linear constraints and convex programming for convex constraints. The following algorithms differ in how the actor and critic are updated in the presence of the projection.

*1) Training Critic with Projected Actions:* A baseline algorithm in some previous work uses the closest point projection and trains the critic with the projected actions [2], [4], [7], [8]. Namely, it trains the critic $Q_w$ in Eq. (2) with $a_i = P_s(\mu_\theta(s_i))$. The algorithm, however, keeps the actor's update rule unchanged. In the case of DDPG, the actor is updated in the direction of $\nabla_\theta Q_w(s_i, \mu_\theta(s_i))$ (Eq. (3)). Intuitively, the actor is updated obliviously to the projection.

As previous work [7] points out, however, updating the actor in $\nabla_\theta Q_w(s_i, \mu_\theta(s_i))$ is not theoretically principled. This is because Q-values for pre-projected actions ($Q^\mu(s, \mu_\theta(s_i))$ are ill-defined when $\mu_\theta(s_i)$ are infeasible. The approach also has a practical issue. Since the critic is only trained with projected actions ($P_s(\mu_\theta(s_i))$), the critic is unlikely to learn action values for pre-projected actions ($\mu_\theta(s_i)$).

The problem is that, since a parameterized policy $\mu_\theta$ is now combined with the projection $P$, policy gradients should be computed for $\nabla_\theta J(P_s \circ \mu_\theta)$ instead of for $\nabla_\theta J(\mu_\theta)$, as:

$$\mathbb{E}_{s \sim \rho^{P \circ \mu}}[\nabla_a Q^{P \circ \mu}(s, a)|_{a = P_s(\mu_\theta(s))} \nabla_\theta P_s \circ \mu_\theta(s)] \tag{7}$$

$$= \mathbb{E}_{s \sim \rho^{P \circ \mu}}[\nabla_a Q^{P \circ \mu}(s, a)|_{a = P_s(\mu_\theta(s))} \frac{dP_s}{d\mu_\theta(s)} \nabla_\theta \mu_\theta(s)] \tag{8}$$

where $\frac{dP_s}{d\mu_\theta(s)}$ is the Jacobian of the projection $P_s$ with respect to suggested actions.

*2) Differentiable Optimization Layer:* To compute estimates for the policy gradient with projections in Eq. (8), most previous work has used *differentiable optimization layers* as the final layer of the policy network [2], [5]–[7]. OptLayer [2] combined a differentiable quadratic programming (QP) layer [23] with deep RL to handle linear constraints. PROF [5] used differentiable convex optimization layers [24] to handle a class of convex programming called disciplined convex programming. Differentiable optimization layers enable the calculation of projection gradients with respect to suggested actions $\frac{dP_s}{d\mu_\theta(s)}$, which changes the actor's update to:

$$\frac{1}{N} \sum_{i=1}^{N} \nabla_a Q_w(s_i, a)|_{a = P_s(\mu_\theta(s_i))} \frac{dP_s}{d\mu_\theta(s)} \nabla_\theta \mu_\theta(s_i). \tag{9}$$

Note that the actor is now updated in a way that incorporates the effect of the projection, *i.e.*, $\frac{dP_s}{d\mu_\theta(s)}$.

Existing methods using differentiable optimization layers share some common challenges. Unlike the original actor update in Eq. (3), the new update rule in Eq. (9) additionally involves projections ($P_s$) and the Jacobian ($\frac{dP_s}{d\mu_\theta(s)}$), which incurs non-negligible computational overhead to processing each state in a mini-batch. More importantly, closest point projection with differentiable optimization layers can suffer from the *zero-gradient issue*. That is, since many actions initially violating constraints can be projected to the same action, the gradients with respect to $\theta$ tend to vanish when $\mu_\theta$ proposes an action violating constraints. For example, in Fig. 1(a), both $a = \mu_\theta(s)$ and $a' = \mu_\theta(s)'$ are projected to the same action. Since the gradient of the projection is zero at $s$, the gradient of the policy $\frac{dP_s}{d\mu_\theta(s)} \nabla_\theta \mu_\theta(s)$ also vanishes at $s$.

[2]Note that mappings discussed here are on actions. This contrasts with projected gradient algorithms, where the parameters ($\theta$) are projected to satisfy the constraints. With neural network policies, projecting $\theta$ so that $\mu_\theta$ satisfies constraints in every continuous state, in general, is not trivial.

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

As a result, the actor ends up completely wasting the sample. To alleviate the zero-gradient issue, some methods introduce penalty terms for constraint violations [2], [5].

*3) Neural Frank-Wolfe Policy Optimization (NFWPO) :* To overcome the zero-gradient issue, NFWPO [8] proposes to decouple the projection from actor updates. This is in contrast to previous methods using differentiable optimization layers, where $\frac{dP_s}{d\mu_\theta(s)}$ is a part of the actor update in Eq. (9). For training the critic $Q_w$, NFWPO is identical to DDPG, and uses Eq.(2) with projected actions. Regarding the actor training, NFWPO initially determines a reference action $a_s \in \mathcal{A}_s$ for each state $s \in S$ in a mini-batch via the Frank-Wolfe algorithm [25]:

$$a_s = P_s(\mu_\theta(s) + \alpha(c_s - P_s(\mu_\theta(s)))) \tag{10}$$

Here, $\alpha$ represents the Frank-Wolfe learning rate, and

$$c_s = \arg\max_{c \in \mathcal{A}_s} \langle c, \nabla_a Q_w(s,a)|_{a=P_s(\mu_\theta(s))} \rangle \tag{11}$$

Subsequently, the policy parameters $\theta$ are updated to minimize the distance between $\mu_\theta(s)$ and the reference action $a_s$. By doing so, NFWPO avoids computing $\frac{dP_s}{d\mu_\theta(s)}$ and policy gradients, thereby overcoming the zero-gradient issue. Without any action constraints, NFWPO is equivalent to DDPG [8].

*4) Training Critic with Pre-Projected Actions:* In addition to the existing methods above, we consider another simple, yet surprisingly effective baseline that trains the actor and the critic with *pre-projected*, possibly infeasible actions $\mu_\theta(s)$, while the projection of the actions to feasible ones is done as a part of state transitions of the environment. In this method, the critic learns $\tilde{Q}^\mu = Q^\mu \circ P$ instead of $Q^\mu$, as it is defined even for infeasible actions. In other words, the critic now learns action-values for suggested actions, given that suggested actions might be projected to feasible actions. Then, the actor is updated using Eq. (3). This approach is computationally more efficient than other techniques using differentiable optimization or NFWPO, since the projection is performed only during rollouts. Moreover, this approach can be easily combined with other RL algorithms, including SAC.

Note that this baseline has several known limitations. As actions are projected to feasible ones as part of state transitions, agents must learn the effects of projection through experienced transitions and rewards. Moreover, this approach can suffer from the zero-gradient issue and may require additional penalty terms to alleviate the problem.

### B. Other Mapping Techniques

In this section, we present several other differentiable mapping techniques. Similar to using differentiable optimization layers (Sec III-A2), the algorithms in this section employ differentiable mappings to feasible actions as the final layers of policies. However, the techniques presented here do not necessarily map actions to the closest feasible ones. The key insight is that since RL algorithms can learn to adjust their suggestions to achieve better performance, there is no inherent need for using the closest point projection.

*1) $\alpha$-Projection Layer:* $\alpha$-projection [12] can be used as an alternative to the closet point projection. $\alpha$-projection assigns an interior point $c_s \in \mathrm{Int}(\mathcal{A}_s)$ for each state $s \in \mathcal{S}$. As Fig. 1(b) shows, given a suggested action $a \in \mathcal{A}$, $\alpha$-projection moves $a$ toward $c_s$ until it reaches $\mathcal{A}_s$. Formally, it picks an action on the ray from $c_s$ to $a$ that is closest to $a$ (each $a \in \mathcal{A}_s$ is mapped to itself). Since points on different rays are mapped to different points, the gradient does not vanish in the example.

As originally studied in [12], for linear-inequality constraints, a closed-form expression exists for $\alpha$-projection and can be implemented as an additional neural network layer. The actor is then updated using Eq. (9), where $P$ is replaced with $\alpha$-projection. It is also possible to extend the $\alpha$-projection to problems with an elliptical constraint, which has the form $(a - c(s))^\mathsf{T} Q(s)(a - c(s)) - b(s)$ where $Q(s)$ is a positive-semi-definite matrix.

Note that $\alpha$-projection assumes the existence of an interior point $c_s \in \mathrm{Int}(\mathcal{A}_s)$ such that, for any point $x \in \mathcal{A}_s$, the segment between $c_s$ and $x$ is contained in $\mathcal{A}_s$. For linear constraints, we use the Chebychev center of $\mathcal{A}_s$ as $c_s$ as in [12]. During training of the actor, $\alpha$-projection must be performed for each state in a minibatch. Computing $c_s$ for each state can be time-consuming, so we store the previously computed Chebyshev centers in the replay buffer and reuse them during training. For problems with an elliptical constraint, we use $c(s)$ as $c_s$.

*2) Radial Squashing Layer:* We also propose an alternative mapping to feasible actions called *radial squashing*, which can also be implemented as a neural network layer. Instead of clipping the ray from $c_s$ as in $\alpha$-projection, radial squashing shrinks actions into the feasible set of actions around its center $c_s$, as illustrated in Fig. 1(c). Let $a$ be the given point and $b$ be the intersection of the boundary of $\mathcal{A}_s$ and the ray from $c_s$ to $a$. The radial squashing layer maps:

$$a \mapsto c_s + \tanh\left(\|a - c_s\| / \|b - c_s\|\right)(b - c_s). \tag{12}$$

Note that, this mapping is differentiable at $c_s$ (the Jacobian becomes the identity matrix there) and gradients never vanish for any direction.

### C. Incorporating Action Constraints to SAC

Although SAC has been a popular algorithm for general RL problems and can be easily combined with the approach described in III-A4, combining it with mapping techniques for action-constrained RL is not trivial. This is mainly due to the entropy term in Eq. (5) that requires the calculation of the probability density of a policy after actions are mapped to feasible ones. In general, for a random variable $X$ with a probability density function $p$, the probability density function $q$ of $f(X)$ after a differentiable mapping $f$ is given by:

$$q(f(X)) = |\det J_f(X)|^{-1} p(X), \tag{13}$$

when $\det J_f(X) \neq 0$, where $J_f(X)$ is the Jacobian of $f$.

*1) Radial Squashing with SAC:* Combining radial squashing with SAC is straightforward since the determinant of Jacobian never vanishes. To evaluate the change in probability

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

density by the radial squashing, it is enough to calculate the Jacobian of Eq. (12) as follows:

$$(a - c_s)\left(\operatorname{grad}\frac{\tanh L}{L}\right)^{\mathsf{T}} + \frac{\tanh L}{L} I, \tag{14}$$

where $I$ is the identity matrix and $L := \|a - c_s\|/\|b - c_s\|$ is a function of $a$ that depends on the constraint.

*2) $\alpha$-projection with SAC:* On the other hand, combining the $\alpha$-projection layer with SAC is non-trivial because the probability density after projection may become infinity (*i.e.*, the determinant of Jacobian may be zero.) To overcome this difficulty, we consider two probability density functions for the policy after projection: a $d$-dimensional density function on the interior of $\mathcal{A}_s$ and a $(d-1)$-dimensional density function on the boundary of $\mathcal{A}_s$, where $d$ is the dimension of $\mathcal{A}_s$. The entropy of the distribution is defined by the sum of the entropies for these distributions on the interior and boundary. Mathematically, we define the measure of $\mathcal{A}_s$ as the sum[3] of the standard measure of the interior of $\mathcal{A}_s$ and that of the boundary of $\mathcal{A}_s$ as a Riemannian submanifold of the Euclidean space [26].

While the probability density for the interior of $\mathcal{A}_s$ remains unchanged by the projection, we need to integrate the probability density before projection for the boundary of $\mathcal{A}_s$. Let $p$ be the probability density function before projection, and let $q$ be the desired probability density function on the boundary after projection. We use the spherical coordinate system with origin $c_s$. Let $b \in \partial\mathcal{A}_s$ be the point on the boundary, and let $\mathrm{d}\sigma$ be the volume element of $\partial\mathcal{A}_s$ at $b$. We must integrate $p$ on the solid cone projected to $\mathrm{d}\sigma$:

$$q\,\mathrm{d}\sigma = \int_{r_0}^{\infty} p r^{d-1}\,\mathrm{d}r\,\mathrm{d}\Omega\,, \tag{15}$$

where $r_0 := \|b - c_s\|$ and $\mathrm{d}\Omega$ is the differential solid angle corresponding to $\mathrm{d}\sigma$ (see Fig. 2). This can be calculated by $\mathrm{d}\Omega = r_0^{-1}\cos\theta\,\mathrm{d}\sigma$, where $\theta$ is the angle between the vector $b - c_s$ and the normal vector of $\partial\mathcal{A}_s$ at $b$. Since $p$ is a Gaussian distribution for SAC, it is enough to calculate an integration of form $\int r^{d-1}\exp\bigl(-(Ar+B)^2\bigr)\,\mathrm{d}r$ after completing the square. We can get its explicit expression using $\exp\bigl(-(Ar+B)^2\bigr)$ and $\operatorname{erf}(Ar+B)$, where erf is the error function, by applying partial integration repeatedly.

*3) Remark on Differential Optimization Layer:* Combining differentiable optimization layers with SAC, on the other hand, may not be feasible. This is because closest point projection can map a region with more than one dimension to one point, making it difficult to determine the change in probability distribution.

### D. ConstraintNet

Another possible approach for handling action-constraints is to modify the output layer of the actor network based on the type of constraints. ConstraintNet [27] uses a convex combination of $n$ given vertices as the output layer when the feasible set of actions is a convex polytope and the vertices of the polytope are known. However, this approach has some limitations: the number of vertices in a polytope can be exponential in the dimension of the action, and it is not straightforward to extend this approach to state-dependent constraints. Therefore, we did not include this approach in our evaluations.

[3]The weight of the entropy (or the measure) for the boundary in this sum is arbitrary and not scale-invariant. It can be considered as a hyperparameter of this algorithm. In this paper, we simply take one. Note that the action space is scaled in the implementation of these algorithms.

## IV. Experiments

In this section, we conduct an empirical evaluation of action-constrained RL algorithms on various simulated control tasks from PyBullet-Gym [14] and MuJoCo [13] in OpenAI Gym [15]. We additionally assess the running time of each algorithm in Sec. IV-D.

### A. Algorithms

We compare the following 13 algorithms in total.

- TD3 family:
  - **DPro**: TD3 with critic trained using projected actions (Sec. III-A1)
  - **DPro+**: **DPro** with the penalty term (see below)
  - **DPre**: TD3 with pre-projected actions (Sec. III-A4)
  - **DPre+**: **DPre** with penalty term
  - **DOpt**: TD3 with optimization layer (Sec. III-A2)
  - **DOpt+**: **DOpt** with penalty term
  - **NFW**: NFWPO (Sec. III-A3) with TD3 techniques (clipped double Q learning, target policy smoothing and delayed policy update)
  - **DAlpha**: TD3 with $\alpha$-projection (Sec. III-B1)
  - **DRad**: TD3 with radial squashing (Sec. III-B2)
- SAC family:
  - **SPre**: SAC with pre-projected actions (Sec. III-A4)
  - **SPre+**: **SPre** with penalty term
  - **SAlpha**: SAC with $\alpha$-projection (Sec. III-C2)
  - **SRad**: SAC with radial squashing (Sec. III-C1)

**DPro+**, **DPre+**, **DOpt+**, and **SPre+** introduce the penalty term for constraint violations discussed in Sec. III-A2 or Sec. III-A4. Specifically, we added $\|\max\{Ax - b, \mathbf{0}\}\|$ for linear constraints $Ax \le b$, where $A$ is normalized so that the norm of each row is 1, as in [2]. For elliptical constraint $(x-c)^{\mathsf{T}}Q(x-c) \le b$, $\max\{\sqrt{(x-c)^{\mathsf{T}}Q(x-c)} - \sqrt{b}, 0\}$ was added where $Q$ is normalized so that $\operatorname{tr} Q$ equals to the dimension. In case there exist both constraints, we take the sum of two penalties.

### B. Implementation Details

We adapted the implementations of TD3 and SAC in Stable Baselines 3 (SB3) [21], a popular reinforcement learning library using PyTorch [28]. We use tuned hyperparameters in RL Baselines Zoo [29] for each base algorithm (TD3 or SAC) and each environment. For **NFW**, the Frank-Wolfe learning rate is tuned to $0.05$ in Reacher and $0.01$ in other environments. Table I shows the used hyperparameters. For train frequency, '1 epi.' means that the model is updated every episode. Similarly, '1 step' means that the model is updated every step. For gradient steps, '-1' means to do as many

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

| Hyperparameters | Reacher | | Hopper | | Others | |
|---|---|---|---|---|---|---|
| | TD3 | SAC | TD3 | SAC | TD3 | SAC |
| Discount factor | 0.98 | | 0.99 | | 0.99 | |
| Net. arch. 1st | 400 | 400 | 400 | 256 | 400 | 256 |
| Net. arch. 2nd | 300 | 300 | 300 | 256 | 300 | 256 |
| Batch size | 100 | 256 | 256 | 256 | 100 | 256 |
| Learning rate | 1e-3 | 7.3e-4 | 3e-4 | 3e-4 | 1e-3 | 3e-4 |
| Buffer size | 2e5 | 3e5 | 1e6 | 1e6 | 1e6 | 1e6 |
| Target Update Ratio | 0.005 | 0.02 | 0.005 | 0.005 | 0.005 | 0.005 |
| Action noise | 0.1 | - | 0.1 | - | 0.1 | - |
| FW learning rate | 0.05 | - | 0.01 | - | 0.01 | - |
| Learning starts | 1e5 | 1e5 | 1e5 | 1e5 | 1e5 | 1e5 |
| Use SDE | - | True | - | False | - | False |
| Train freq. | 1 epi. | 8 steps | 1 step | 1 step | 1 epi. | 1 step |
| Gradient steps | -1 | 8 | 1 | 1 | -1 | 1 |

TABLE I
HYPERPARAMETERS USED IN THE EXPERIMENTS.

| Environment | Name | Constraint |
|---|---|---|
| Reacher | R+N | No additional constraint |
| | R+L2 | $a_1^2 + a_2^2 \leq 0.05$ |
| | R+O03 | $\sum_{i=1}^{2} \lvert w_i a_i \rvert \leq 0.3$ |
| | R+O10 | $\sum_{i=1}^{2} \lvert w_i a_i \rvert \leq 1.0$ |
| | R+O30 | $\sum_{i=1}^{2} \lvert w_i a_i \rvert \leq 3.0$ |
| | R+M | $\sum_{i=1}^{2} \max\{w_i a_i, 0\} \leq 1.0$ |
| | R+T | $a_1^2 + 2a_1(a_1 + a_2)\cos\theta_2 + (a_1 + a_2)^2 \leq 0.05$ |
| HalfCheetah | HC+O | $\sum_{i=1}^{6} \lvert w_i a_i \rvert \leq 20$ |
| | HC+MA | $w_1 a_1 \sin(\theta_1 + \theta_2 + \theta_3) + w_4 a_4 \sin(\theta_4 + \theta_5 + \theta_6) \leq 5$ |
| Hopper | H+M | $\sum_{i=1}^{3} \max\{w_i a_i, 0\} \leq 10$ |
| | H+O+S | $\sum_{i=1}^{3} \lvert w_i a_i \rvert \leq 10$, $\sum_{i=1}^{3} a_i^2 \sin^2\theta_i \leq 0.1$ |
| Walker2d | W+M | $\sum_{i=1}^{6} \max\{w_i a_i, 0\} \leq 10$ |
| | W+O+S | $\sum_{i=1}^{6} \lvert w_i a_i \rvert \leq 10$, $\sum_{i=1}^{6} a_i^2 \sin^2\theta_i \leq 0.1$ |

TABLE II
EXPERIMENT ENVIRONMENTS AND CONSTRAINTS

gradient steps as steps done in the environment during the rollout.

We also used gurobi [30] for solving linear or quadratic programming and cvxpylayers [24] for differentiable optimization layers. To address potential failures of the optimization solver for projections from distant points, we applied the squashing technique to each coordinate of the neural network outputs before projection, as done in [8], for methods utilizing pre-projected actions, NFWPO and the optimization layer.

### C. Evaluations with Various Environments and Constraints

In this experiment, we compared algorithms on the Reacher in PyBullet-Gym and Hopper, Walker2d, and HalfCheetah in MuJoCo[4] with various constraints. Each action is represented by a vector $(a_1, \ldots, a_d)$ corresponding to torques given to $d$ joints, where $d = 2$ for Reacher, $d = 3$ for Hopper and $d = 6$ for Walker2d and HalfCheetah. Let $\theta_1, \ldots, \theta_d$ and $w_1, \ldots, w_d$ be the angle and the angular velocity of joints respectively.

In addition to the original box constraint $-1 \leq a_i \leq 1$ for each $i$, we consider constraints shown in Table II. Note that the constraint for R+L2 and HC+O are the same as those in [8]. For R+N, R+L2, R+O03, R+O10, R+O30, R+T, HC+O, H+O+S and W+O+S, the center ($c_s$) of the feasible actions is at the origin, but that is not necessarily for other cases.

For each combination of algorithms and constraints, we ran the algorithm with 10 different seeds. The number of timesteps is $3 \times 10^5$ in Reacher and $10^6$ in other three environments as in [29]. In each run, we evaluated five episodes and took their mean in every $5,000$ timesteps. After each run, we evaluated 50 episodes again for the best parameters and consider their mean as the final result.

Table III shows the average values and standard errors of rewards over the 10 seeds, where the top three algorithms for each constraint are emphasized with bold text. Due to the limitation of cvxpylayer, **DOpt** and **DOpt+** cannot be applied to some constraints.[5] So we excluded these algorithms from some results and display "unavailable" in the table. Furthermore, since **DOpt** and **DOpt+** are too computationally expensive to run with the tuned batch size for MuJoCo environments, we excluded them and additionally ran all algorithms including them with batch size 16 for constraints for which **DOpt** and **DOpt+** are available. The bottom three rows of the table shows the results of additional experiments. Fig. 3 and Fig. 4 present the learning curves averaged over the 10 seeds for R+L2 and HC+O, respectively.

### D. Measurements of Training Runtime

Finally, we measured the runtime of each algorithm for the first 1000 gradient steps of the training procedure in the HalfCheetah environment, with batch sizes $1, 16$ or $100$. Fig. 5 summarizes the runtimes in seconds averaged over 10 trials with different seeds.

### E. Findings and Discussion

*1) Training the critic using pre-projected actions is a good baseline especially with penalty terms:* We found that **DPro**, *i.e.*, training the critic using projected actions, demonstrated limited performance compared to **DPre** with pre-projected actions in HalfCheetah and Walker environments, despite that it was used as a baseline in some previous work [7], [8]. This could be due to its fundamental defect discussed in Section III-A1 and indicates that *pre-projected actions should be used for training the critic*. **DPre+**, on the other hand, achieved performance comparable to other TD3 variants. We contend that both **SPre** and **SPre+**, which are SAC algorithms utilizing pre-projected actions, serve as viable initial selections among the evaluated algorithms. As demonstrated in Table III, these methods consistently rank within the top three across most conditions, while also offering the advantages of relatively low implementation efforts and computational runtime as shown in Fig. 5

*2) The use of optimization layers and NFWPO comes with significant runtime overheads:* **DOpt** requires significant computation time, as clearly shown in Fig. 5. Nevertheless, it does not demonstrate significantly better performance than **DPre+**, **DAlpha** or **DRad**.

[4]The descriptions of the tasks are available at www.gymlibrary.dev.

[5]Constraints should adhere to the disciplined parameterized programming rules [24].

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

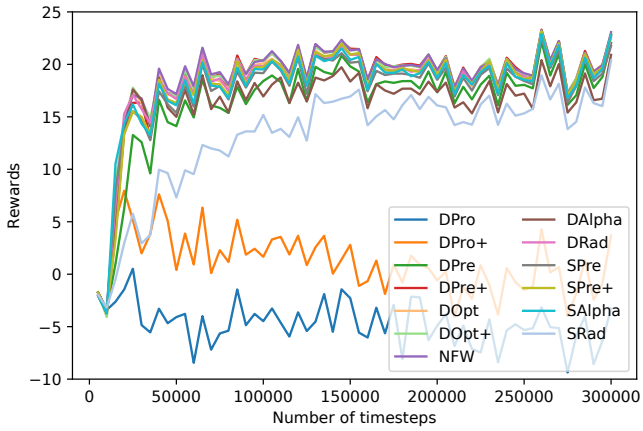



Fig. 3. Learning curves for R+L2

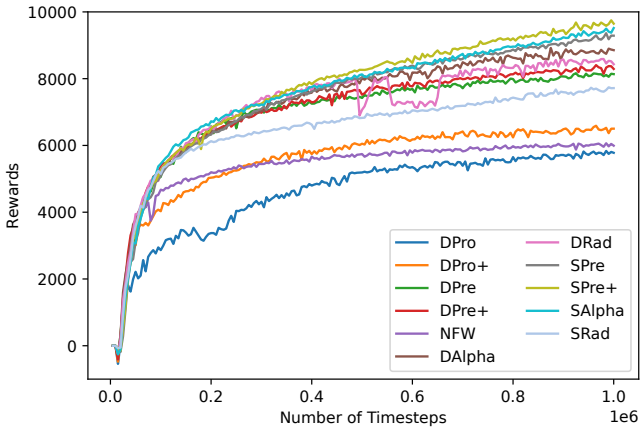



Fig. 4. Learning curves for HC+O with batch size $\geq 100$

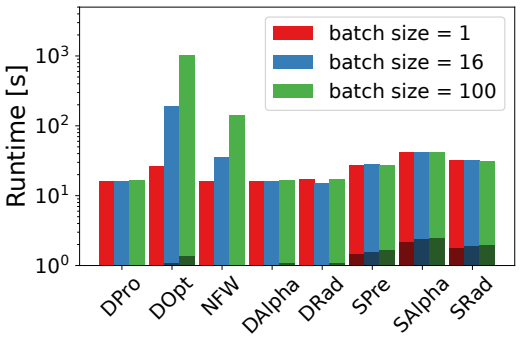



Fig. 5. Training runtime for HalfCheetah. The light areas of the bars show the time on a CPU and the dark areas show the time on a GPU.

While **NFW** achieved the best learning performance among the TD3 family in the Reacher environment, its performance is not significantly better in other environments. On the other hand, it also requires a considerably large runtime as the batch size becomes larger.

*3) Mapping techniques can be less expensive alternatives to optimization layers:* **SAlpha** and **SRad** demonstrated good performance for a small batch size at the cost of 140-150% runtime overheads compared to **SPre+**. Also, **DAlpha** and **DRad** performed on par (if not better) with **DOpt** and **DOpt+**, while requiring much less runtime. Nonetheless, under certain conditions (R+L2 and R+T), SRad exhibited subpar performance. Investigating the reasons behind this underperformance will be the subject of future research.

## V. Conclusion

In this paper, we compared variants of TD3 and SAC on a variety of continuous control tasks in the presence of action constraints. Our evaluation includes new variants of the existing action-constrained RL algorithms. Our benchmark evaluation has led to the following main findings. 1) Training the critic with pre-projected actions is a good baseline, especially with penalty terms. 2) The use of optimization layers and NFWPO comes with significant runtime overheads. 3) Mapping techniques are useful alternatives to optimization layers. A complete implementation of our benchmark evaluation is available online at github.com/omron-sinicx/action-constrained-RL-benchmark.

## References

[1] Y. Fujita and S. Maeda, "Clipped Action Policy Gradient," in *International Conference on Machine Learning*, 2018, pp. 1597–1606.

[2] T.-H. Pham, G. De Magistris, and R. Tachibana, "OptLayer - Practical Constrained Optimization for Deep Reinforcement Learning in the Real World," in *2018 IEEE International Conference on Robotics and Automation*. IEEE Press, 2018, pp. 6236–6243.

[3] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE International Conference on Robotics and Automation*. IEEE Press, 2017, pp. 3389–3396.

[4] R. Cheng, G. Orosz, R. M. Murray, and J. W. Burdick, "End-to-End Safe Reinforcement Learning through Barrier Functions for Safety-Critical Continuous Control Tasks," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, 2019, pp. 3387–3395.

[5] B. Chen, P. L. Donti, K. Baker, J. Z. Kolter, and M. Bergés, "Enforcing Policy Feasibility Constraints through Differentiable Projection for Energy Optimization," in *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*. ACM, 2021, pp. 199–210.

[6] G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru, and Y. Tassa, "Safe Exploration in Continuous Action Spaces," 2018, arXiv:1801.08757.

[7] A. Bhatia, P. Varakantham, and A. Kumar, "Resource Constrained Deep Reinforcement Learning," in *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*, 2019, pp. 610–620.

[8] J.-L. Lin, W.-T. Hung, S. Yang, P.-C. Hsieh, and X. Liu, "Escaping from Zero Gradient: Revisiting Action-Constrained Reinforcement Learning via Frank-Wolfe Policy Optimization," in *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, 2021.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *Proceedings of the Fourth International Conference on Learning Representations 2016*, 2016.

[10] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," in *Proceedings of the Thirty-Fifth International Conference on Machine Learning*. PMLR, 2018, pp. 1861–1870.

[11] Y. Chow, O. Nachum, A. Faust, E. Duenez-Guzman, and M. Ghavamzadeh, "Lyapunov-based Safe Policy Optimization for Continuous Control," 2019, arXiv:1901.10031.

[12] S. Sanket, A. Sinha, P. Varakantham, P. Andrew, and M. Tambe, "Solving Online Threat Screening Games using Constrained Action Space Reinforcement Learning," 2020, pp. 2226–2235.

[13] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**

| | TD3 Family | | | | | | | | | SAC Family | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Conditions | DPro | DPro+ | DPre | DPre+ | DOpt | DOpt+ | NFW | DAlpha | DRad | SPre | SPre+ | SAlpha | SRad |
| R+N | 17.57 ±0.27 | Same as DPro due to no constraints | | | | | **18.47** ±0.22 | 17.40 ±0.38 | 17.34 ±0.24 | **18.61** ±0.23 | Same SPre | 15.01 ±2.00 | **18.47** ±0.26 |
| R+L2 | -4.65 ±0.59 | 9.90 ±2.69 | 17.81 ±0.33 | **19.70** ±0.30 | 19.12 ±0.30 | 19.15 ±0.51 | **19.70** ±0.33 | 18.46 ±0.86 | 18.76 ±0.37 | 18.84 ±0.27 | **19.45** ±0.29 | 18.97 ±0.31 | 15.69 ±0.46 |
| R+O03 | 17.32 ±0.37 | 18.74 ±0.32 | 18.17 ±0.30 | 18.19 ±0.33 | 17.72 ±0.42 | 18.55 ±0.26 | **18.86** ±0.38 | 18.20 ±0.29 | 17.88 ±0.32 | **18.78** ±0.35 | 15.82 ±0.43 | 18.58 ±0.27 | **19.11** ±0.30 |
| R+O10 | 17.44 ±0.53 | 17.78 ±0.34 | 17.32 ±0.31 | 17.76 ±0.34 | 18.01 ±0.18 | 17.80 ±0.42 | **18.70** ±0.30 | 17.52 ±0.50 | 17.98 ±0.23 | 18.30 ±0.35 | **18.37** ±0.32 | 18.29 ±0.32 | **18.82** ±0.33 |
| R+O30 | 17.59 ±0.32 | 17.14 ±0.44 | 17.32 ±0.36 | 17.15 ±0.36 | 16.48 ±0.60 | 16.42 ±0.44 | **18.44** ±0.21 | 17.07 ±0.32 | 16.40 ±0.50 | **18.51** ±0.25 | **18.41** ±0.26 | 16.90 ±0.79 | 17.51 ±0.88 |
| R+M | 17.90 ±0.29 | 17.70 ±0.33 | 17.58 ±0.28 | 17.82 ±0.35 | 17.54 ±0.34 | 17.73 ±0.29 | **18.53** ±0.31 | 17.89 ±0.31 | 17.48 ±0.43 | 18.29 ±0.39 | **18.57** ±0.31 | 16.18 ±1.86 | **18.80** ±0.25 |
| R+T | 0.35 ±2.34 | **18.51** ±0.26 | 15.95 ±1.29 | **18.53** ±0.34 | Unavailable | | 17.97 ±0.35 | 18.34 ±0.31 | 17.79 ±0.30 | 18.19 ±0.41 | **18.49** ±0.30 | 14.94 ±2.72 | -2.26 ±0.44 |
| HC+O | 6062 ±405 | 6759 ±352 | 8255 ±474 | 8475 ±426 | - | - | 6213 ±216 | 9165 ±212 | 8736 ±315 | **9448** ±154 | **9973** ±223 | **9646** ±265 | 7904 ±323 |
| HC+MA | 9668 ±282 | 9853 ±367 | 9780 ±403 | 10003 ±394 | Unavailable | | 7166 ±321 | 9898 ±134 | 9515 ±170 | **10079** ±443 | **10318** ±328 | **10235** ±202 | 8369 ±324 |
| H+M | 3151 ±103 | 3258 ±55 | **3354** ±10 | 3295 ±35 | - | - | **3309** ±20 | 3265 ±19 | **3305** ±16 | 3253 ±60 | 3278 ±23 | 3180 ±54 | 3190 ±58 |
| H+O+S | 987 ±78 | 2245 ±233 | 3179 ±30 | **3261** ±34 | Unavailable | | 1785 ±316 | **3260** ±26 | 3251 ±26 | 3257 ±19 | 2992 ±300 | 2867 ±290 | **3300** ±21 |
| W+M | 1479 ±280 | **4694** ±169 | 4460 ±84 | **4898** ±155 | - | - | 4454 ±51 | 4071 ±190 | 4341 ±146 | 4446 ±131 | **4595** ±147 | 4268 ±158 | 4319 ±107 |
| W+O+S | 856 ±70 | 931 ±113 | 3959 ±77 | **4102** ±100 | Unavailable | | 3408 ±300 | 3377 ±37 | 3467 ±65 | 3828 ±87 | **4127** ±79 | **4105** ±76 | 3961 ±96 |
| (batch size = 16) | | | | | | | | | | | | | |
| HC+O | 4108 ±399 | 5303 ±115 | 5826 ±366 | 6787 ±378 | 7053 ±239 | 7306 ±362 | 5984 ±319 | 7245 ±191 | 7387 ±204 | **8059** ±259 | 7071 ±762 | **7748** ±245 | **7459** ±169 |
| H+M | 3166 ±70 | 3260 ±28 | 3193 ±61 | 3247 ±62 | 3154 ±122 | **3290** ±33 | 3041 ±52 | **3262** ±31 | 3221 ±49 | **3303** ±19 | 3211 ±58 | 3249 ±34 | 3243 ±52 |
| W+M | 2138 ±251 | 3021 ±176 | 3409 ±267 | **3767** ±246 | 2837 ±431 | 3292 ±506 | 3644 ±128 | 3483 ±117 | **4013** ±137 | 3483 ±182 | 3600 ±135 | **3672** ±106 | 3431 ±182 |

TABLE III
AVERAGE REWARDS.


[14] B. Ellenberger, "PyBullet gymperium," https://github.com/benelot/pybullet-gym, 2018–2019.

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[16] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

[17] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," Oct. 2018, arXiv:1802.09477 [cs, stat].

[18] J. Li, D. Fridovich-Keil, S. Sojoudi, and C. J. Tomlin, "Augmented Lagrangian Method for Instantaneously Constrained Reinforcement Learning Problems," in *Proceeding of the Sixtieth IEEE Conference on Decision and Control*. IEEE, 2021, pp. 2982–2989.

[19] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada, "Control Barrier Functions: Theory and Applications," in *Proceeding of the Eighteenth European Control Conference*, 2019, pp. 3420–3431.

[20] M. Pereira, Z. Wang, I. Exarchos, and E. Theodorou, "Safe Optimal Control Using Stochastic Barrier Functions and Deep Forward-Backward SDEs," in *Proceedings of the 2020 Conference on Robot Learning*. PMLR, 2021, pp. 1783–1801.

[21] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[22] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the Thirty-First International Conference on International Conference on Machine Learning*. JMLR, 2014, pp. 387–395.

[23] B. Amos and J. Z. Kolter, "OptNet: Differentiable optimization as a layer in neural networks," in *Proceedings of the Thirty-Fourth International Conference on Machine Learning*. JMLR, 2017, pp. 136–145.

[24] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter, "Differentiable convex optimization layers," *Advances in neural information processing systems*, vol. 32, 2019.

[25] M. Frank and P. Wolfe, "An algorithm for quadratic programming," *Naval Research Logistics Quarterly*, vol. 3, no. 1-2, 1956.

[26] J. M. Lee and J. M. Lee, *Smooth manifolds*. Springer, 2012.

[27] M. Brosowsky, F. Keck, O. Dünkel, and M. Zöllner, "Sample-Specific Output Constraints for Neural Networks," in *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*, 2021, pp. 6812–6821.

[28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[29] A. Raffin, "RL baselines3 zoo," https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

[30] B. Bixby, "The gurobi optimizer," *Transp. Re-search Part B*, vol. 41, no. 2, pp. 159–178, 2007.


**IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.**