Modification-Considering Value Learning for Reward Hacking Mitigation in RL

Anonymous Authors[1]

Abstract

Reinforcement learning (RL) agents can exploit unintended strategies to achieve high rewards without fulfilling the desired objectives, a phenomenon known as reward hacking. In this work, we examine reward hacking through the lens of General Utility RL, which generalizes RL by considering utility functions over entire trajectories rather than state-based rewards. From this perspective, many instances of reward hacking can be seen as inconsistencies between current and updated utility functions, where the behavior optimized for an updated utility function is poorly evaluated by the current one. Our main contribution is Modification-Considering Value Learning (MCVL), a novel algorithm designed to avoid this inconsistency during learning. Starting with a coarse, yet aligned initial utility function, the MCVL agent iteratively refines this function while considering the potential consequences of updates. We implement MCVL agents based on DDQN and TD3 and demonstrate their effectiveness in preventing reward hacking in diverse environments, including those from AI Safety Gridworlds and the MuJoCo gym.

1. Introduction

Reinforcement learning (RL) agents have solved a wide range of tasks by learning to maximize cumulative rewards (Mnih et al., 2015; Levine et al., 2016). However, this reward-maximization paradigm has a significant flaw: agents may exploit poorly defined or incomplete reward functions, leading to a behavior known as *reward hacking* (Skalse et al., 2022), where the agent maximizes the reward signal but fails to meet the designer's true objectives.

For instance, an RL agent tasked with stacking blocks may end up flipping them when the reward is based on the height of the bottom face of a block (Popov et al., 2017). As RL systems scale to more complex, safety-critical applications such as autonomous driving (Kiran et al., 2021) and medical diagnostics (Ghesu et al., 2017), ensuring reliable and safe agent behavior becomes increasingly important. Pan et al. (2022) showed that reward hacking becomes more common as models grow in complexity. Similar problems have been observed in large language models trained with RL (Denison et al., 2024; OpenAI, 2024).

Addressing reward hacking in practice has proven challenging. Skalse et al. (2022) demonstrated that mitigating it requires either restricting the agent's policy space or carefully managing the optimization process. The former has been approached by regularizing a policy to remain close to a known safe policy (Laidlaw et al., 2023), though this type of regularization may compromise optimality. Regarding the latter, prior work has suggested that certain reward hacking behaviors could be mitigated using *current utility optimization* (Orseau & Ring, 2011; Hibbard, 2012; Everitt et al., 2016; 2021). In this paradigm, agents optimize a utility function at each step and evaluate changes to their policy or utility function based on the utility function of the current step. However, these approaches remain largely conceptual. With the exception of Dewey (2011), they assume the utility function is task-specific and predefined, whereas Dewey (2011) proposed that it could be inferred from past interactions but did not provide further details. Additionally, these approaches lack a rigorous formalization of utility optimization and do not propose concrete implementable algorithms.

In this paper, we address this research gap by framing reward hacking within the General Utility RL (GU-RL) formalism (Zahavy et al., 2021; Geist et al., 2022), a generalization of classical RL. Intuitively, GU-RL considers utility functions that evaluate entire trajectories rather than individual transitions. We propose learning such utility functions from past observed rewards. To enable this, we introduce trajectory value functions and generalize value-based RL to general utilities. Moreover, we introduce the concept of *inconsistent updates*, where utility updates during training produce policies that perform poorly under the pre-update utility function. Our key insight is that many reward hacking scenarios in classical RL can be understood as inconsistent updates to the learned utility function, including manipulat-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

ing reward signals (Everitt et al., 2021) or tampering with sensors (Ring & Orseau, 2011).

Based on this insight, we introduce *Modification-Considering Value Learning* (MCVL). In MCVL, the agent updates its utility function based on observed rewards, similar to value-based RL. Additionally, it predicts the long-term consequences of potential updates and rejects those found to be inconsistent. Avoiding inconsistent utility updates is an optimal behavior in our formulation. We provide an algorithm for learning utility functions, estimating future policies, and comparing them using the current utility function. Furthermore, we introduce a learning setup where the initial utility function is trained in a *Safe* sandbox environment before transitioning to the *Full* version. We evaluate our approach in diverse environments, including benchmarks adapted from the AI Safety Gridworlds (Leike et al., 2017) and MuJoCo gym (Towers et al., 2024). To our knowledge, this work is the first to demonstrate successful learning of non-reward hacking behaviors in these environments. Our results offer insights into key factors influencing MCVL performance, providing a foundation for future research on mitigating reward hacking in RL.

2. Background

We consider the usual Reinforcement Learning (RL) setup, where an agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards (Sutton & Barto, 2018). This interaction is modeled as a Markov Decision Process (MDP) (Puterman, 2014) defined by the tuple $(S, A, P, R, \rho, \gamma)$, where S is the set of states, A is the set of actions, $P : S \times A \times S \to \mathbb{R}$ is the transition kernel, $R : S \times A \to \mathbb{R}$ is the reward function, ρ is the initial state distribution, and γ is the discount factor. The objective in a standard RL is to learn a policy $\pi : S \to A$ that maximizes the expected return, defined as the cumulative discounted reward $\mathbb{E}^{\pi}_{\rho}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]$. The expected return from taking action a in state s and subsequently following policy π is called state-action value function and denoted as $Q^{\pi}(s, a)$.

General-Utility RL (GU-RL) In this work, we focus on an agent that optimizes a learned utility function. This problem naturally falls within the framework of General-Utility Reinforcement Learning (GU-RL) (Zhang et al., 2020; Zahavy et al., 2021; Geist et al., 2022), a generalization of standard RL. Instead of assigning rewards merely to individual transitions, GU-RL uses a utility function F that assignes value to the distribution of state-action pairs induced by a policy. This broader framework encompasses tasks such as risk-sensitive RL (Mihatsch & Neuneier, 2002), apprenticeship learning (Abbeel & Ng, 2004), and pure exploration (Hazan et al., 2019).

Formally, the utility function F maps a state-action occupancy measure to a real value. An occupancy measure describes the distribution over state-action pairs encountered under a given policy. For a given policy π and an initial state distribution ρ, the occupancy measure λ^{π}_{ρ} is defined as

$$\lambda^{\pi}_{\rho}(s, a) \overset{\text{def}}{=} \sum_{t=0}^{+\infty} \gamma^t \mathbb{P}^{\pi}_{\rho}(s_t = s, a_t = a),$$

where $\mathbb{P}^{\pi}_{\rho}(s_t = s, a_t = a)$ is the probability of observing the state-action pair (s, a) at time step t under policy π starting from ρ. The utility function $F(\lambda^{\pi}_{\rho})$ assigns a scalar value to the occupancy measure induced by the policy π. The agent's objective is to find π that maximizes $F(\lambda^{\pi}_{\rho})$.

A trajectory $\tau = (s_0, a_0, \ldots, s_h, a_h)$ induces the occupancy measure $\lambda(\tau)$, defined as

$$\lambda(\tau) \overset{\text{def}}{=} \sum_{t=0}^{h} \gamma^t \delta_{s_t, a_t},$$

where $\delta_{s,a}$ is an indicator function that is 1 only for the state-action pair (s, a) (Barakat et al., 2023).

Standard RL is a special case of GU-RL, where the utility function F_{RL} is linear with respect to the occupancy measure, and maximizing it corresponds to maximizing the expected cumulative return:

$$F_{RL}(\lambda^{\pi}_{\rho}) = \langle R, \lambda^{\pi}_{\rho} \rangle = \mathbb{E}^{\pi}_{\rho}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right].$$

3. Method

We aim to address reward hacking in RL by introducing *Modification-Considering Value Learning* (MCVL). The MCVL agent continuously updates its utility function based on observed rewards while avoiding inconsistent utility modifications that could lead to suboptimal behavior under the current utility function. To achieve this, we generalize value-based RL to GU-RL setting by introducing trajectory value functions and value learning (VL) algorithm. Then we modify VL such that utility function is only updated if the policy induced by the updated utility function is not worse than the policy induced by the current utility function. The policies are compared by the values of the trajectories they produce according to the current utility function.

Trajectory Value Function We introduce *trajectory value functions* to compute the values of the trajectories. A trajectory value function $U^{\pi}(\tau)$ evaluates the utility of an occupancy measure induced by starting with a trajectory $\tau = (s_0, a_0, \ldots, s_h, a_h)$ and following a policy π after the end of this trajectory:

$$U^{\pi}(\tau) \overset{\text{def}}{=} F\left(\lambda(\tau) + \gamma^h \lambda^{\pi}_{S_{h+1}}\right), \quad (1)$$

Algorithm 1 Value-Learning (VL)

```
Input: Replay buffer $D$, policy $\pi_0$, initial utility $U_{VL_0}$
1: for time step $t = 0$, while not converged do
2:     $a_t \leftarrow \pi_t(s_t)$ {Select action}
3:     $s_{t+1}, r_t \leftarrow$ act$(a_t)$ {Perform action}
4:     $\pi_{t+1} \leftarrow$ ImprovePolicy$(\pi_t, U^{\pi_t}_{VL_t})$
       {Update utility:}
5:     $D \leftarrow D \cup \{T_{t-1}\}$
6:     $U^{\pi_{t+1}}_{VL_{t+1}} \leftarrow$ UpdateUtility$(U^{\pi_{t+1}}_{VL_t}, U^{\pi_{t+1}}_{RL}, \mathcal{T}_D)$

7:     $T_t \leftarrow (s_t, a_t, s_{t+1}, r_t)$
8: end for
```

Algorithm 2 Modification-Considering VL (MCVL)

```
Input: Replay buffer $D$, policy $\pi_0$, initial utility $U_{VL_0}$
1: for time step $t = 0$, while not converged do
2:     $(a_t, modify) \leftarrow \pi_t(T_{t-1})$ {Select action}
3:     $s_{t+1}, r_t \leftarrow$ act$(a_t)$ {Perform action}
4:     $\pi_{t+1} \leftarrow$ ImprovePolicy$(\pi_t, U^{\pi_t}_{VL_t})$
5:     if $modify$ then
6:         $D \leftarrow D \cup \{T_{t-1}\}$
7:         $U^{\pi_{t+1}}_{VL_{t+1}} \leftarrow$ UpdateUtility$(U^{\pi_{t+1}}_{VL_t}, U^{\pi_{t+1}}_{RL}, \mathcal{T}_D)$
8:     end if
9:     $T_t \leftarrow (s_t, a_t, s_{t+1}, r_t)$
10: end for
```

where S_{h+1} is the distribution of the states following the τ, and $\lambda^{\pi}_{S_{h+1}}$ represents the occupancy measure induced by following π from S_{h+1}. In the standard RL setting, this simplifies to the following:

$$U^{\pi}_{RL}(\tau) = \sum_{t=0}^{h-1} \gamma^t R(s_t, a_t) + \gamma^h Q^{\pi}(s_h, a_h). \tag{2}$$

Every trajectory value function has a corresponding utility function $F(\lambda^{\pi}_{\rho}) = \mathbb{E}_{\tau \sim \mathcal{T}^{\pi}_{\rho}} U^{\pi}(\tau)$, where $\mathcal{T}^{\pi}_{\rho}$ denotes a distribution of trajectories started from state distribution ρ and continued by following a policy π. Thus, it is also referred to as *utility* for brevity.

Value Learning (VL) The *value-learning* agent optimizes a utility U_{VL}, which is learned from observed transitions (Dewey, 2011). Algorithm 1 provides a description of a value learning agent that learns the utility from rewards. In our framework, the policy at each step is updated towards maximizing the current utility U_{VL_t}, while the utility is updated towards RL-based utility U_{RL} using trajectories $\mathcal{T}_D$ formed from the set of previously observed transitions D:

$$\mathcal{T}_D = \{(s_0, a_0, \ldots, s_h, a_h) \mid \forall t \in \{0, \ldots, h-1\} \\ \exists\, r \in \mathbb{R} : (s_t, a_t, s_{t+1}, r) \in D\}.$$

Value-based RL algorithms such as DDQN (van Hasselt et al., 2016) and TD3 (Fujimoto et al., 2018) can be seen as special cases of the value-learning, where $U^{\pi_t}_{VL_t}$ only considers the state-action value of the first state and action in a trajectory: $U^{\pi_t}_{VL_t}(s_0, a_0, \ldots, s_h, a_h) = Q^{\pi_t}(s_0, a_0)$.

Modification-Considering VL (MCVL) The distinction between VL agents and standard RL agents becomes apparent when the agent is *modification-considering*, meaning it is aware of its learning process and evaluates the consequences of modifying its utility function. For the agents optimizing U_{RL}, it is always optimal to learn from all transitions, as they provide information about the utility being optimized. However, for VL agents optimizing U_{VL_t} at time step t, it may be optimal to avoid learning from certain transitions. Specifically, the agent may predict its future behavior after updating its utility to $U_{VL_{t+1}}$ and compare it to the predicted behavior under its current utility U_{VL_t}. If the updated behavior has lower utility according to U_{VL_t}, it is optimal to avoid such an update since the agent is currently optimizing U_{VL_t}.

To make this decision-making process explicit, we introduce an additional boolean action that determines whether to modify the utility function after an interaction with the environment. The modified action space is $A^m = A \times \{0, 1\}$, where each action $a^m_i = (a_i, modify_i)$ includes a decision to modify or to keep the current utility. The policy is adjusted to take the full transition as input, rather than just the environment state because the decision whether to modify the utility may also depend on the observed reward and next state. After each interaction, the agent explicitly decides whether to update its utility function based on the new experience. The MCVL algorithm is presented in Algorithm 2, with the modifications highlighted in red. We refer to the transitions where the optimal choice is *modify* $=$ *false* as *utility-inconsistent*, and to the process of selecting *modify* as *utility inconsistency detection*.

Implementation We implement an MCVL agent for discrete action spaces using DDQN and for continuous action spaces using TD3. These implementations are referred to as MC-DDQN and MC-TD3, respectively. Here, we focus on describing MC-DDQN; the implementation of MC-TD3, which is highly similar, is detailed in Appendix B. In MC-DDQN, $U_{VL}(\tau; \theta, \psi)$ is parameterized as

$$\sum_{t=0}^{h-1} \gamma^t \dot{R}(s_t, a_t; \psi) + \gamma^h \dot{Q}(s_h, a_h; \theta), \tag{3}$$

where $\dot{R}(s, a; \psi)$ is a learned reward model, and $\dot{Q}(s, a; \theta)$ is the state-action value function. The policy $\pi(T)$ outputs an environment action a and a boolean *modify*, which

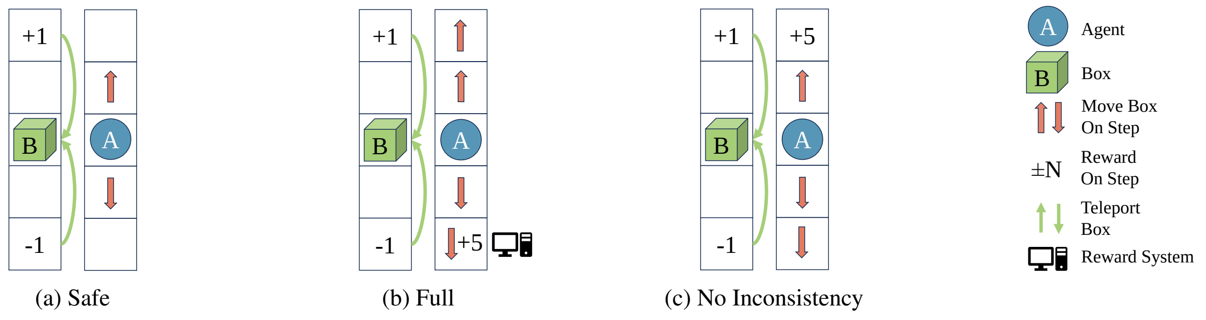


Figure 1. (a) *Safe* version of *Box Moving* environment: the optimal policy is to repeatedly press the up arrow by moving up and down. (b) In *Full* version, maximum returns are achieved by pressing the down arrows, receiving +5 observed reward for each press of the bottom-most arrow, but this also moves the box down, which is inconsistent with utility learned in the *Safe* version. There is also a policy that moves the box up twice as fast by alternating between up arrows. (c) In *No Inconsistency* version, collecting +5 reward does not conflict with moving the box up, so the agent trained in *Safe* should not encounter utility inconsistency in this version of the environment.

indicates whether to update the utility function. The environment action a is chosen as $\arg\max_a \dot{Q}(s, a; \theta)$, while decision $modify$ is determined by comparing expected future utilities. Specifically, the agent compares the expected utility of future policies: a modified π_m, assuming T was added to the replay buffer D, and unmodified π_u, assuming it was not. It then computes

$$modify = \mathop{\mathbb{E}}_{\tau \in \mathcal{T}_\rho^{\pi_m}} [U_{VL_t}(\tau)] \geq \mathop{\mathbb{E}}_{\tau \in \mathcal{T}_\rho^{\pi_u}} [U_{VL_t}(\tau)], \quad (4)$$

where the expectations are computed by averaging over k trajectories of length h. The future policies π^m and π^u are computed by applying l DDQN updates to the current action-value function $\dot{Q}(s, a; \theta)$ using replay buffers $D \cup \{T\}$ and D, respectively. To speed up learning from the replay buffer $D \cup \{T\}$, we include transition T in each sampled mini-batch. The reward model parameters ψ are updated using L_2 loss on batches sampled from the replay buffer D, the action-value function parameters θ are updated through DDQN updates on the same batches. The full implementation of MC-DDQN is presented in Appendix A.

Initial Utility Function An MCVL agent described in Algorithm 2 requires some initial utility function as input. In this work, we propose to learn this initial utility function in a *Safe* sandbox version of the environment, where unintended behaviors cannot be discovered by the exploratory policy. Examples of *Safe* environments include simulations or closely monitored lab settings where the experiment can be stopped and restarted without consequences if undesired behaviors are detected. To differentiate from the *Safe* version, we refer to the broader environment as the *Full* environment. This *Full* environment may include the *Safe* one, for example, if the agent's operational scope is expanded beyond a restricted lab setting. Alternatively, the *Safe* and *Full* environments may be distinct, such as when transitioning from simulation to real-world deployment. For the proposed approach to perform effectively, however, the *Safe* and *Full* environments must be sufficiently similar to allow for successful generalization of the learned utility function. We do not require a separate *Safe* environment if the reward hacking behavior is sufficiently hard to discover, as demonstrated in our Reacher experiment.

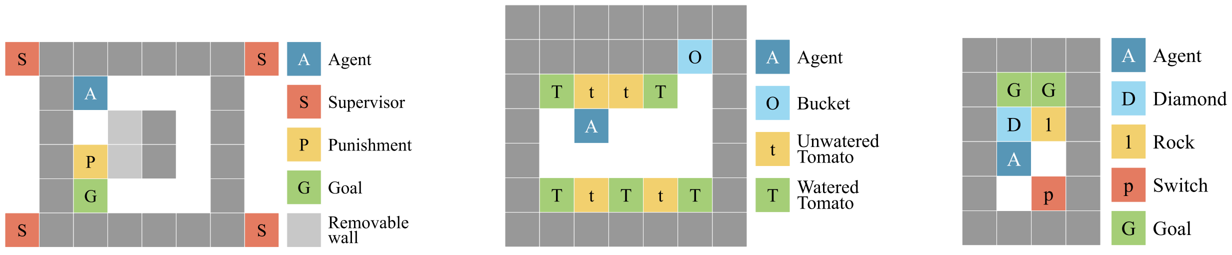
4. Experiments

To empirically validate our approach, we introduce environments that can be switched between *Safe* and *Full* variants. Following Leike et al. (2017), each environment includes a performance metric in addition to the observed reward. This metric tracks how well the agent follows the intended behavior. A high observed reward combined with a low performance metric indicates reward hacking. In the *Safe* versions of the environments, the performance metric is identical to the reward.

4.1. Environments

To illustrate a scenario where utility inconsistency might arise, we introduce the *Box Moving* environment, shown in Figure 1. Additionally, we adopt several established environments to evaluate our method's performance on known challenges. These include the *Absent Supervisor* and *Tomato Watering* environments from AI Safety Gridworlds (Leike et al., 2017), as well as the *Rocks and Diamonds* environment from Everitt et al. (2021), all depicted in Figure 2. To test our algorithm in continuous action spaces, we introduce the possibility of reward hacking into the Reacher environment from Gymnasium (Towers et al., 2024).

Box Moving Environment The environment consists of two parts: the left part represents an external world with a box that can be moved up and down, while the right part is a room where the agent can move. When the box reaches the top-most or bottom-most cell, the agent receives a reward of +1 or -1, respectively, and the box teleports back to the



(a) Absent Supervisor (b) Tomato Watering (c) Rocks and Diamonds

Figure 2. Gridworld environments. Each environment has *Safe* and *Full* versions. In *Safe* version, the *Absent Supervisor* guarantees the presence of supervisors, *Tomato Watering* does not have the bucket, and *Rocks and Diamonds* does not have the switch. We provide additional customization options for testing different scenarios.

central cell. The agent can move up and down, and stepping on arrows on the floor moves the box in the corresponding direction. Staying on the arrows does not influence the box.

In the *Safe* version, the top and bottom of the room are empty, and the optimal strategy is to repeatedly press the up arrow. This setting allows the agent to learn an initial utility function that favors moving the box upward. In the *Full* version, the agent has more interaction opportunities. The top cell is replaced by an up arrow, allowing the agent to move the box up twice as fast by alternating between two up arrows. This setup shows that the *Full* environment can offer more efficient ways to achieve outcomes that were learned to be good in the *Safe* version. The bottom empty cell is replaced by a down arrow. However, by stepping into this cell, the agent can trick the reward system to give itself a +5 reward. While this results in higher observed rewards, it also moves the box downward, which was learned to be undesirable in the *Safe* version.

The *No Inconsistency* version modifies the top cell to provide +5 reward instead. The agent can collect this reward without disrupting upward box movement, demonstrating that not all changes in reward structure result in utility inconsistency.

Absent Supervisor This environment tests whether the agent will alter its behavior based on the presence of a supervisor. The goal is to reach a target cell for +50 reward, but a punishment cell along the shortest path gives -30 reward if stepped on while a supervisor is present. Each movement costs -1 reward, encouraging the agent to take the shortest path. In the *Safe* version, the supervisor is always present, leading the initial utility function to learn that going through the punishment cell is undesirable. In the *Full* version, the supervisor is present only half the time. The performance metric penalizes the agent for stepping on the punishment cell, even in the supervisor's absence. We expect that when the agent steps on the punishment cell without supervision, it will experience utility inconsistency, as learning from this transition leads to the policy that visits the punishment cell, which contradicts the low utility assigned to this behavior previously.

Tomato Watering This environment models tampering with observations. The agent is rewarded based on the number of watered tomatoes, which dry out randomly. After stepping on the bucket, all empty cells appear to the agent as watered tomatoes, representing a delusion. In the *Safe* version, the bucket is removed from the environment. The performance metric tracks the true number of watered tomatoes. In the *Full* version, we expect utility inconsistency when the agent tries to delude itself using the bucket if it compares policies by sampling trajectories from the true environment transition model where stepping on the bucket does not affect the tomatoes. In this case, its new behavior, standing on the bucket, should have lower utility than watering the tomatoes. However, if a delusional transition model is used, a room full of watered tomatoes should still appear to have high utility. Without access to the labels we assigned to cells, the agent cannot distinguish between deluding itself by putting a bucket on its head and the real change, i.e. efficient watering of all tomatoes at once.

Rocks and Diamonds This environment allows the agent to change its reward function by visiting a switch. Initially, diamonds in the goal area provide +1 reward at every step, and rocks in the goal area yield -1. Activating the switch alters the observed reward for rocks to +10, while the performance metric remains the same. In the *Safe* version, the switch is removed. Both diamonds and rocks can be pushed. We expect utility inconsistency when the agent encounters a situation where the switch is active and there is a rock in a goal area since updating from this transition would lead to a policy that puts rocks in the goal area and this action had low utility in the past.

Reacher This environment is based on the Reacher-v5 environment from the MuJoCo subset of the Gymnasium benchmark (Towers et al., 2024). It involves a robotic arm tasked with reaching a randomly placed target, starting from

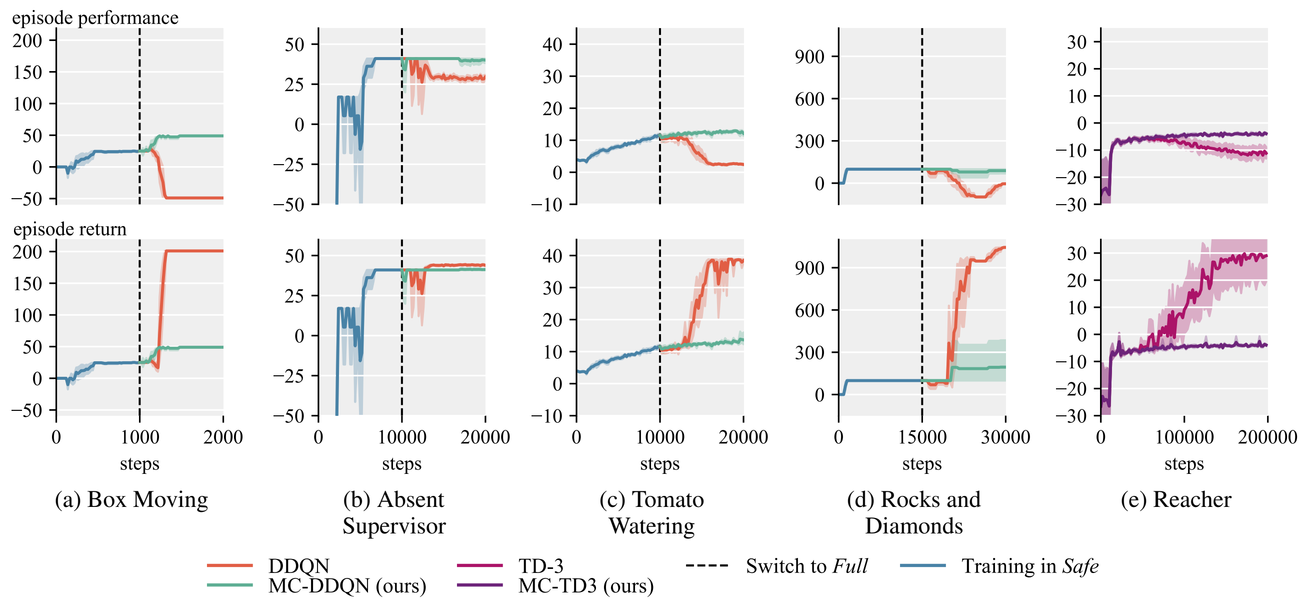


Figure 3. Episode performance (top) and returns (bottom) of MC-DDQN and MC-TD3 in comparison to DDQN and TD3. Performance tracks the intended behavior, while returns are discounted sum of the observed reward. After switching to *Full* version, the returns of baselines grow while performance drops, indicating that they engage in reward hacking. The performance of our algorithms does not drop and improves in environments with better policies available in *Full* version. Bold lines represent the mean over 10 seeds, and shaded regions indicate a bootstrapped 95% confidence interval.

randomized joint rotations and velocities. We modified the environment by adding an invisible button that provides a one-time reward of +50 when held for 15 consecutive steps, simulating reward tampering. The performance metric does not include this reward. Since reward hacking is difficult to discover in this environment, we train the initial utility function from random policy rollouts without using the *Safe* environment. Further details are provided in Appendix D.

4.2. Experimental Setup

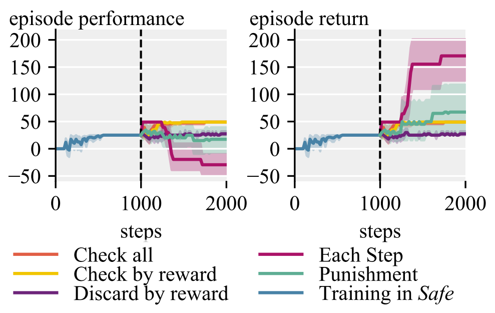
Unless specified otherwise, we train the initial utility function in the *Safe* versions of environments for a fixed number of steps. We use ϵ-greedy exploration (Watkins, 1989) and linearly decay ϵ. We compare our MC-DDQN approach with the standard DDQN, both initialized with weights and the replay buffer obtained in the *Safe* version and trained with the same hyperparameters. In the Reacher environment, we compare our MC-TD3 to TD3. The only difference of MC-DDQN and MC-TD3 compared to the baselines is the utility inconsistency detection. To accelerate training, we check for utility inconsistency only when observed rewards deviate from predicted rewards by more than a fixed threshold $\delta = 0.05$. Section 4.4 confirms that using the threshold does not change the empirical performance, while ignoring all such transitions prevents learning the optimal non-hacking policy. The complete hyperparameters are provided in Appendix G.

4.3. Results

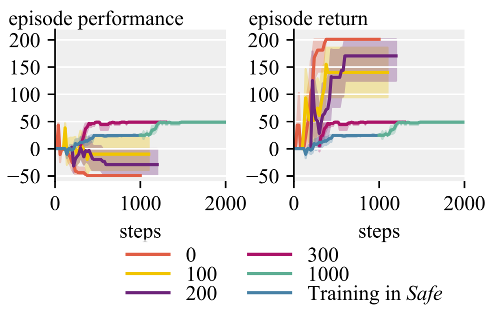
Our findings show that incorporating utility inconsistency detection can prevent reward hacking and learned behavior aligns with the intended tasks. Our algorithm continues to improve the performance in the *Full* version after learning the initial utility in the *Safe* version of each environment. DDQN and TD3 baselines learn unintended behaviors leading to a drop in the performance metric. The key results are shown in Figure 3.

Our approach relies on the generalization of the initial utility function from *Safe* to *Full* version of the environment. For the results in Figure 3b, we set the number of supervisors to one to minimize the distribution shift. We examine performance under greater distribution shift in Appendix C. Forecasting modified future policies from a single transition was particularly challenging and required careful hyperparameter tuning. In one out of 10 runs in the *Rocks and Diamonds* environment, utility inconsistency went undetected due to incorrect policy forecasting. Further qualitative analysis of failure modes is presented in Appendix E.

In the *Tomato Watering* experiment, we provided MC-DDQN with a non-delusional transition model for policy comparisons. This model did not include rewards, and the agent still encountered delusional transitions in the environment. This scenario simulates a situation where the agent can change its observations while retaining an accurate world model, akin to a human using a VR headset. In



(a) Alternative training schemes



(b) Initial training steps

Figure 4. Additional experiments in Box Moving environment. (a) Comparison of the different training schemes: *Check all* corresponds to checking all transitions for utility inconsistency; *Check by reward* checks only transitions for which predicted reward differs from the observed by at least δ; *Discard by reward* discards all transitions where predicted reward sufficiently differs from the observed; *Each step* evaluates policies before and after each gradient step without forecasting the future policies; *Punishment* replaces utility-inconsistent transitions' rewards with a punishment reward. (b) Effect of different amounts of initial utility function training in *Safe* environment.

this setting our algorithm correctly identifies the inconsistent transitions. However, as expected, when the delusional model was used to roll out trajectories for comparisons, no utility inconsistencies were detected and the behavior of MC-DDQN was identical to DDQN.

4.4. Ablations and sensitivity analysis

We tested several alternative schemes for utility inconsistency detection and mitigation. As shown in Figure 4a, checking all transitions for utility inconsistency yields similar results to checking only those where the predicted reward significantly differs from the observed reward. However, discarding all such transitions prevents the algorithm from learning an optimal non-hacking policy. Comparing policies before and after each gradient step without forecasting future policies also fails to prevent reward hacking. Surprisingly, replacing the reward of inconsistent transitions with large negative values is less effective at preventing reward hacking than removing them from the replay buffer. Having such transitions in the replay buffer prevents the algorithm from forecasting the correct future policy when checking for inconsistency, and over time the replay buffer gets populated with both transitions with positive and negative rewards, destabilizing training.

Figure 4b illustrates the performance with varying amounts of initial utility function training in the *Safe* version. Remarkably, one run avoided reward hacking after just 100 steps of such training. After 300 steps, all seeds converged to the optimal non-hacking policy, even though most had not discovered the optimal policy within the *Safe* version by that point. This result suggests that future systems might avoid reward hacking with only moderate training in a *Safe* environment. Additionally, this experiment shows that without any training in *Safe* environment (0 steps) our algorithm behaves identical to the baseline. Additional experiments are reported in Appendix C.

5. Limitations

While our method effectively mitigates reward hacking in several environments, it comes with computational costs, which are detailed in Appendix F. Checking for utility inconsistency requires forecasting two future policies by training the corresponding action-value functions until convergence. In the worst case, where each transition is checked for potential utility inconsistency, this process can lead to a runtime slowdown proportional to the number of iterations used to update the action-value functions. Checking only transitions where the predicted reward deviates from the observed reward by more than δ can significantly reduce the computational burden. However, this approach introduces an additional hyperparameter. Balancing computational efficiency with effectiveness is a key area for future research. Promising avenues include leveraging Meta-RL (Schmidhuber, 1987) to accelerate policy forecasting. A particularly promising direction is in-context RL (Laskin et al., 2022) which can learn new behaviors in-context during inference, quickly and without costly training (Bauer et al., 2023).

Another limitation is that our approach addresses only a subset of reward hacking scenarios. Specifically, it depends on the reward model and value function generalizing correctly to novel trajectories. This approach may not address reward hacking issues caused by incorrect reward shaping, like in the CoastRunners problem (OpenAI, 2023). In this case, if the agent already learned about a small positive reward (e.g., knocking over a target), the agent's current utility function may assign a high utility to behaviors that exploit this reward, even if they fail to achieve the final goal (completing the loop). Alternative methods, such as potential-based reward shaping (Ng et al., 1999), may be more appropriate to address such issues.

Finally, our current implementation assumes access to rollouts from the true environment transition model, while only the reward model is learned. Extending our approach to work with learned latent transition models represents a promising direction for future research. Furthermore, using a learned world model to predict utility-inconsistent transitions before they occur could further enhance the applicability and efficiency of the method. Improvements to computational efficiency and the integration of learned transition models would also enable testing our method in more complex environments, which is an important direction for future work.

6. Related Work

The problem of agents learning unintended behaviors by exploiting misspecified training signals has been extensively discussed in the literature as *reward hacking* (Skalse et al., 2022), *reward gaming* (Leike et al., 2018), or *specification gaming* (Krakovna et al., 2020). Krakovna et al. (2020) provide a comprehensive overview of these behaviors across RL and other domains. The theoretical foundations for understanding reward hacking are explored by Skalse et al. (2022).

Laidlaw et al. (2023) propose addressing reward hacking by regularizing the divergence between the occupancy measures of the learned policy and a known safe policy. Unlike their approach, which may overly restrict the agent's ability to learn effective policies, our method does not require the final policy to remain close to any predefined policy. Eisenstein et al. (2024) investigate whether ensembles of reward models trained from human feedback can mitigate reward hacking, showing that while ensembles reduce the problem, they do not completely eliminate it. To avoid additional computational overhead, we do not use ensembles in this work, but they could complement our method by improving the robustness of the learned utility function.

A specific form of reward hacking, where an agent manipulates the mechanism by which it receives rewards, is known as *wireheading* (Amodei et al., 2016; Taylor et al., 2016; Everitt & Hutter, 2016; Majha et al., 2019) or *reward tampering* (Kumar et al., 2020; Everitt et al., 2021). Related phenomena, where an agent manipulates its sensory inputs to deceive the reward system, are discussed as *delusion-boxing* (Ring & Orseau, 2011), *measurement tampering* (Roger et al., 2023), and *reward-input tampering* (Everitt et al., 2021). Several studies have hypothesized that current utility optimization could mitigate reward or sensor tampering (Yudkowsky, 2011; Hibbard, 2012; Yampolskiy, 2014). One of the earliest discussions of this issue is in by Schmidhuber (2003), who developed the concept of *Gödel-machine* agents, capable of modifying their own source code, including the utility function. They suggested that such modifications should only occur if the new values are provably better according to the old ones. However, none of these works addressed learning the utility function or described the optimization process in full detail.

Everitt & Hutter (2016) considered a setting where the agent learns a posterior given a prior over manually specified utility functions, proposing an agent that is not incentivized to tamper with its reward signal by selecting actions that do not alter its beliefs about the posterior. More recently, Everitt et al. (2021) formalized conditions under which an agent optimizing its current reward function would lack the incentive to tamper with the reward signal. Our work suggests an implementation of value learning in standard RL environments, where the utility function is learned from the past rewards. Additionally, our method is applicable to other instances of reward hacking beyond reward tampering. Moreover, it aims to prevent reward hacking, rather than simply removing the incentive for it.

7. Conclusion

In this work, we introduced *Modification-Considering Value Learning*, an algorithm that allows an agent to optimize its current utility function, learned from observed transitions, while considering the future consequences of utility updates. Using the General Utility RL framework, we formalized the concept of current utility optimization. Our implementations, MC-DDQN and MC-TD3, demonstrated the ability to avoid reward hacking in several previously unsolved environments. Furthermore, we experimentally showed that our algorithm can improve the policy performance while remaining aligned with the initial objectives.

To the best of our knowledge, this is the first implementation of an agent that optimizes its utility function while considering the potential consequences of modifying it. We believe that studying such agents is an important direction for future research in AI safety, especially as AI systems become more general and aware of their environments and training processes (Berglund et al., 2023; Denison et al., 2024; Greenblatt et al., 2024). One of the key contributions of this work is providing tools to model such agents using contemporary RL algorithms.

Our empirical results also identify best practices for modeling these agents, including the importance of forecasting future policies and excluding utility-inconsistent transitions from the training process. Additionally, we introduced a set of modified environments designed for evaluating reward hacking, where agents first learn what to value in *Safe* environments before continuing their training in *Full* environments. We believe this evaluation protocol offers a valuable framework for studying reward hacking and scaling solutions to real-world applications.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Abbeel, P. and Ng, A. Y. Apprenticeship learning via inverse reinforcement learning. In *ICML*, 2004. 2

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016. 8

Barakat, A., Fatkhullin, I., and He, N. Reinforcement learning with general utilities: Simpler variance reduction and large state-action space. In *ICML*, 2023. 2

Bauer, J., Baumli, K., Behbahani, F., Bhoopchand, A., Bradley-Schmieg, N., Chang, M., Clay, N., Collister, A., Dasagi, V., Gonzalez, L., Gregor, K., Hughes, E., Kashem, S., Loks-Thompson, M., Openshaw, H., Parker-Holder, J., Pathak, S., Perez-Nieves, N., Rakicevic, N., Rocktäschel, T., Schroecker, Y., Singh, S., Sygnowski, J., Tuyls, K., York, S., Zacherl, A., and Zhang, L. M. Human-timescale adaptation in an open-ended task space. In *ICML*, 2023. 7

Berglund, L., Stickland, A. C., Balesni, M., Kaufmann, M., Tong, M., Korbak, T., Kokotajlo, D., and Evans, O. Taken out of context: On measuring situational awareness in llms. *arXiv preprint arXiv:2309.00667*, 2023. 8

Denison, C., MacDiarmid, M., Barez, F., Duvenaud, D., Kravec, S., Marks, S., Schiefer, N., Soklaski, R., Tamkin, A., Kaplan, J., et al. Sycophancy to subterfuge: Investigating reward-tampering in large language models. *arXiv preprint arXiv:2406.10162*, 2024. 1, 8

Dewey, D. Learning what to value. In *International conference on artificial general intelligence*. Springer, 2011. 1, 3

Eisenstein, J., Nagpal, C., Agarwal, A., Beirami, A., D'Amour, A. N., Dvijotham, K. D., Fisch, A., Heller, K. A., Pfohl, S. R., Ramachandran, D., Shaw, P., and Berant, J. Helping or herding? reward model ensembles mitigate but do not eliminate reward hacking. In *First Conference on Language Modeling*, 2024. 8

Everitt, T. and Hutter, M. Avoiding wireheading with value reinforcement learning. In *Artificial General Intelligence*, 2016. 8

Everitt, T., Filan, D., Daswani, M., and Hutter, M. Self-modification of policy and utility function in rational agents. In *Artificial General Intelligence*, 2016. 1

Everitt, T., Hutter, M., Kumar, R., and Krakovna, V. Reward tampering problems and solutions in reinforcement learning: A causal influence diagram perspective. *Synthese*, 198(Suppl 27), 2021. 1, 2, 4, 8

Fujimoto, S., Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. In *ICML*, 2018. 3, 12

Geist, M., Perolat, J., Laurière, M., Elie, R., Perrin, S., Bachem, O., Munos, R., and Pietquin, O. Concave utility reinforcement learning: the mean-field game viewpoint. In *AAMAS*, 2022. 1, 2

Ghesu, F.-C., Georgescu, B., Zheng, Y., Grbic, S., Maier, A., Hornegger, J., and Comaniciu, D. Multi-scale deep reinforcement learning for real-time 3d-landmark detection in ct scans. *IEEE transactions on pattern analysis and machine intelligence*, 41(1):176–189, 2017. 1

Greenblatt, R., Denison, C., Wright, B., Roger, F., MacDiarmid, M., Marks, S., Treutlein, J., Belonax, T., Chen, J., Duvenaud, D., et al. Alignment faking in large language models. *arXiv preprint arXiv:2412.14093*, 2024. 8

Hazan, E., Kakade, S., Singh, K., and Van Soest, A. Provably efficient maximum entropy exploration. In *ICML*, 2019. 2

Hibbard, B. Model-based utility functions. *Journal of Artificial General Intelligence*, 3(1):1–24, 2012. 1, 8

Huang, S., Dossa, R. F. J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., and Araújo, J. G. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. 12, 16

Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Al Sallab, A. A., Yogamani, S., and Pérez, P. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23 (6):4909–4926, 2021. 1

Krakovna, V., Uesato, J., Mikulik, V., Rahtz, M., Everitt, T., Kumar, R., Kenton, Z., Leike, J., and Legg, S. Specification gaming: the flip side of AI ingenuity. *DeepMind Blog*, 3, 2020. 8

Kumar, R., Uesato, J., Ngo, R., Everitt, T., Krakovna, V., and Legg, S. REALab: An embedded perspective on tampering. *arXiv preprint arXiv:2011.08820*, 2020. 8

Laidlaw, C., Singhal, S., and Dragan, A. Preventing reward hacking with occupancy measure regularization. In *ICML Workshop on New Frontiers in Learning, Control, and Dynamical Systems*, 2023. 1, 8

Laskin, M., Wang, L., Oh, J., Parisotto, E., Spencer, S., Steigerwald, R., Strouse, D., Hansen, S. S., Filos, A., Brooks, E., Gazeau, M., Sahni, H., Singh, S., and Mnih, V. In-context reinforcement learning with algorithm distillation. In *NeurIPS Foundation Models for Decision Making Workshop*, 2022. 7

Leike, J., Martic, M., Krakovna, V., Ortega, P. A., Everitt, T., Lefrancq, A., Orseau, L., and Legg, S. AI safety gridworlds. *arXiv preprint arXiv:1711.09883*, 2017. 2, 4

Leike, J., Krueger, D., Everitt, T., Martic, M., Maini, V., and Legg, S. Scalable agent alignment via reward modeling: a research direction. *arXiv preprint arXiv:1811.07871*, 2018. 8

Levine, S., Finn, C., Darrell, T., and Abbeel, P. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016. 1

Majha, A., Sarkar, S., and Zagami, D. Categorizing wireheading in partially embedded agents. *arXiv preprint arXiv:1906.09136*, 2019. 8

Mihatsch, O. and Neuneier, R. Risk-sensitive reinforcement learning. *Machine learning*, 49:267–290, 2002. 2

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533, 2015. 1

Ng, A. Y., Harada, D., and Russell, S. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pp. 278–287, 1999. 7

OpenAI. Faulty reward functions. `https://openai.com/research/faulty-reward-functions`, 2023. Accessed: 2024-04-10. 7

OpenAI. Openai o1 system card. `https://openai.com/index/openai-o1-system-card/`, September 2024. Accessed: 2024-09-26. 1

Orseau, L. and Ring, M. Self-modification and mortality in artificial agents. In *Artificial General Intelligence*, 2011. 1

Pan, A., Bhatia, K., and Steinhardt, J. The effects of reward misspecification: Mapping and mitigating misaligned models. In *ICLR*, 2022. 1

Popov, I., Heess, N., Lillicrap, T., Hafner, R., Barth-Maron, G., Vecerik, M., Lampe, T., Tassa, Y., Erez, T., and Riedmiller, M. Data-efficient deep reinforcement learning for dexterous manipulation. *arXiv preprint arXiv:1704.03073*, 2017. 1

Puterman, M. L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014. 2

Ring, M. and Orseau, L. Delusion, survival, and intelligent agents. In *Artificial General Intelligence: 4th International Conference, AGI 2011, Mountain View, CA, USA, August 3-6, 2011. Proceedings 4*. Springer, 2011. 2, 8

Roger, F., Greenblatt, R., Nadeau, M., Shlegeris, B., and Thomas, N. Measurement tampering detection benchmark. *arXiv preprint arXiv:2308.15605*, 2023. 8

Schmidhuber, J. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987. 7

Schmidhuber, J. Gödel machines: self-referential universal problem solvers making provably optimal self-improvements. *arXiv preprint cs/0309048*, 2003. 8

Skalse, J., Howe, N., Krasheninnikov, D., and Krueger, D. Defining and characterizing reward gaming. *NeurIPS*, 2022. 1, 8

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018. 2

Taylor, J., Yudkowsky, E., LaVictoire, P., and Critch, A. Alignment for advanced machine learning systems. *Ethics of Artificial Intelligence*, 2016. 8

Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024. 2, 4, 5

van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *AAAI*. AAAI Press, 2016. 3

Watkins, C. J. C. H. *Learning from delayed rewards*. PhD thesis, King's College, 1989. 6

Yampolskiy, R. V. Utility function security in artificially intelligent agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 26(3):373–389, 2014. 8

Yudkowsky, E. Complex value systems in friendly ai. In *Artificial General Intelligence: 4th International Conference, AGI 2011, Mountain View, CA, USA, August 3-6, 2011. Proceedings 4*, pp. 388–393. Springer, 2011. 8

Zahavy, T., O'Donoghue, B., Desjardins, G., and Singh, S. Reward is enough for convex mdps. *NeurIPS*, 2021. 1, 2

Zhang, J., Koppel, A., Bedi, A. S., Szepesvari, C., and Wang, M. Variational policy gradient method for reinforcement learning with general utilities. *NeurIPS*, 2020. 2

A. Implementation Details of MC-DDQN

Algorithm 3 Policy Forecasting

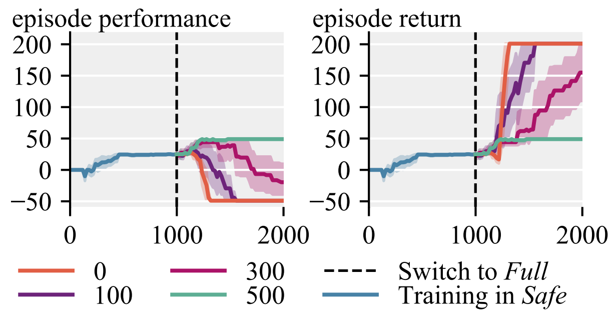
```
Input: Set of transitions $T$, replay buffer $D$, current Q-network parameters $\theta$, training steps $l$
Output: Forecasted policy $\pi_f$
 1: $\theta_f \leftarrow$ COPY($\theta$) {Copy current Q-network parameters}
 2: for training step $t = 1$ to $l$ do
 3:     Sample random mini-batch $B$ of transitions from $D$
 4:     $\theta_f \leftarrow$ TRAINDDQN($\theta_f, B \cup T$)
 5: end for
return $\pi_f(s) = \arg\max_a \dot{Q}(s, a; \theta_f)$ {Return forecasted policy}
```

Algorithm 4 Utility Estimation

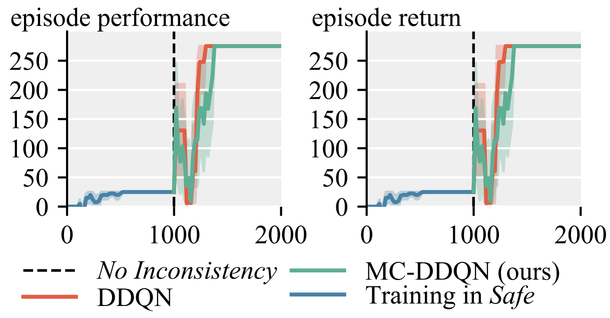
```
Input: Policy $\pi$, environment transition model $P$, utility parameters $\theta$ and $\psi$, initial states $\rho$, rollout steps $h$, number of rollouts $k$
Output: Estimated utility of the policy $\pi$
 1: for rollout $r = 1$ to $k$ do
 2:     $u_r \leftarrow 0$ {Initialize utility for this rollout}
 3:     $s_0 \sim \rho$ {Sample an initial state}
 4:     $a_0 \leftarrow \pi(s_0)$ {Get action from policy}
 5:     for step $t = 0$ to $h - 1$ do
 6:         $u_r \leftarrow \dot{R}(s_t, a_t; \psi) + \gamma u_r$ {Accumulate predicted reward}
 7:         $s_{t+1} \sim P(s_t, a_t)$ {Sample next state from transition model}
 8:         $a_{t+1} \leftarrow \pi(s_{t+1})$ {Get action from policy}
 9:     end for
10:     $u_r \leftarrow u_r + \gamma^h \dot{Q}(s_h, a_h; \theta)$ {Add final Q-value}
11: end for
return $\frac{1}{k}\sum_{r=1}^{k} u_r$ {Return average utility over rollouts}
```

Algorithm 5 Modification-Considering Double Deep Q-learning (MC-DDQN)

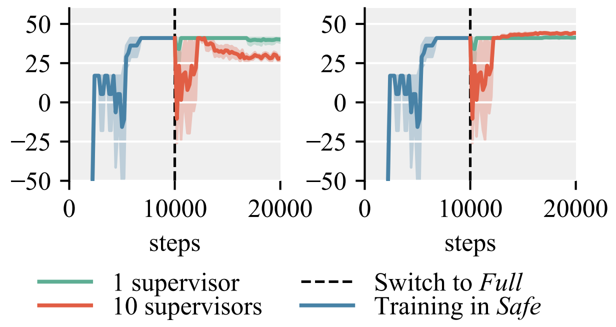
```
Input: Initial utility parameters $\theta$ and $\psi$, replay buffer $D$, environment transition model $P$, initial states $\rho$, rollout horizon $h$, number of rollouts $k$, forecasting trainig steps $l$, number of time steps $n$.
Output: Trained Q-network and reward model
 1: for time step $t = 1$ to $n$ do
 2:     $a_t \leftarrow \epsilon$-GREEDY($\arg\max_a \dot{Q}(s_t, a; \theta)$)
 3:     $\pi_m \leftarrow$ POLICYFORECASTING($\{T_{t-1}\}, D, \theta, l$) {Forecast a policy for modified utility}
 4:     $\pi_u \leftarrow$ POLICYFORECASTING($\{\}, D, \theta, l$) {Forecast a policy for an unmodified utility}
 5:     $F_m \leftarrow$ UTILITYESTIMATION($\pi_m, P, \theta, \psi, \rho, h, k$) {Utility of modified policy}
 6:     $F_u \leftarrow$ UTILITYESTIMATION($\pi_u, P, \theta, \psi, \rho, h, k$) {Utility of unmodified policy}
 7:     $modify \leftarrow (F_m \geq F_u)$ {Check that modified policy isn't worse according to current utility}
 8:     if $modify$ then
 9:         Store transition $T_{t-1}$ in $D$
10:         Sample random mini-batch $B$ of transitions from $D$
11:         $\theta \leftarrow$ TRAINDDQN($\theta, B$) {Update Q-network}
12:         $\psi \leftarrow$ TRAIN($\psi, B$) {Update reward model using $L_2$ loss}
13:     end if
14:     Execute action $a_t$, observe reward $r_t$, and transition to state $s_{t+1}$
15:     $T_t \leftarrow (s_t, a_t, s_{t+1}, r_t)$
16: end for
```



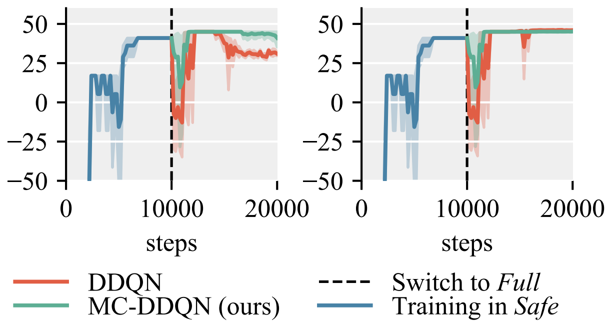
(a) Inconsistency check training steps



(b) Box Moving *No Inconsistency*



(c) Number of supervisors



(d) Removing walls in *Absent Supervisor*

Figure 5. (a) Sensitivity to inconsistency check training step in Box Moving environment. (b) Results in the *No Inconsistency* version of the Box Moving environment. (c) Varying the number of supervisors in Absent Supervisor environment. (d) Results in a variant of Absent Supervisor where a shorter path becomes available in the *Full* version of the environment.

B. Implementation Details of MC-TD3

Our implementation is based on the implementation provided by Huang et al. (2022). The overall structure of the algorithm is consistent with MC-DDQN, described in Appendix A, with key differences outlined below. TD3 is an actor-critic algorithm, meaning that the parameters θ define both a policy (actor) and a Q-function (critic). In Algorithm 3 and Algorithm 5, calls to TRAINDDQN are replaced with TRAINTD3, which updates the actor and critic parameters θ as specified by Fujimoto et al. (2018). Additionally, in Algorithm 3, the returned policy $\pi_f(s)$ corresponds to the actor rather than $\arg\max_a \dot{Q}(s, a; \theta_f)$ and in Algorithm 5 the action executed in the environment is also selected by the actor.

C. Additional Experiments

In Figure 5a, we investigated the necessary number of inconsistency check training steps l to effectively avoid undesired behavior in the Box Moving environment. We observed that with an insufficient number of training steps, certain undesired transitions are not recognized as utility inconsistent, yet our algorithm still slows down the learning of reward hacking behavior.

In Figure 5b, we examine the behavior of MC-DDQN in the *No Inconsistency* version of the *Box Moving* environment. In this version, the agent receives a +5 reward on the top cell, allowing it to move the box upward while collecting this reward. As anticipated, in this scenario, our agent does not detect utility inconsistency for any transitions and successfully learns the optimal policy.

We also conducted experiments in the *Absent Supervisor* environment, varying the number of supervisors. In Figure 5c, it can be observed that increasing the number of supervisors from 1 to 10 leads to unstable utility inconsistency detection, despite the change being purely visual. Qualitative analysis revealed that our neural networks struggled to adapt to this distribution shift, resulting in predicted rewards deviating significantly from the ground truth.

Furthermore, we explored the impact of removing two walls from the *Absent Supervisor* environment after training in the *Safe* version. Without these two walls, a shorter path to the goal is available that bypasses the Punishment cell, although going through the *Punishment* cell remains faster. In Figure 5d, it is evident that while our algorithm can learn a better policy that avoids the *Punishment* cell, the inconsistency detection becomes unreliable. This decline in reliability is attributed to the increased distribution shift between the *Safe* and *Full* versions of the environment.

D. Details of the Experiment in the Reacher Environment

The rewards in the original Reacher-v5 environment are calculated as the sum of the negative distance to the target and the negative joint actuation strength. This reward structure encourages the robotic arm to reach the target while minimizing large, energy-intensive actions. The target's position is randomized at the start of each episode, and random noise is added to the joint rotations and velocities. Observations include the angles and angular velocities of each joint, the target's coordinates, and the difference between the target's coordinates and the coordinates of the arm's end. Actions consist of torques applied to the joints, and each episode is truncated after 50 steps.

We modified the environment by introducing a +50 reward when the arm's end remains within a small, fixed region for 15 consecutive steps. This region remains unchanged across episodes, simulating a scenario where the robot can tamper with its reward function, but such behavior is difficult to discover. In our setup, a reward-tampering policy is highly unlikely to emerge through random actions and is typically discovered only when the target happens to be inside the reward-tampering region.

In accordance with standard practice, each training run begins with exploration using random policy. For this experiment, we do not need a separate *Safe* environment; instead, the initial utility function is trained using transitions collected during random exploration. This demonstrates that our algorithm can function effectively even when a *Safe* environment is unavailable, provided that the initial utility function is learned from transitions that do not include reward hacking.

E. Qualitative Observations

During our preliminary experiments, we encountered several instances where our algorithm failed to detect utility inconsistencies, leading to reward hacking behaviors. Here, we describe these occurrences and how they can be addressed.

Utility Inconsistency Check Rollout Steps When using smaller inconsistency check rollout steps h, we noticed that during the evaluation of future trajectories, the non-hacking policy sometimes needed to traverse several states with low rewards to reach a high reward region. In such cases, the reward hacking policy, which remained stationary, had a higher estimated utility. Increasing h resolved this issue.

Forecasting Future Policy for Current Utility Initially, we forecasted only one future policy by training with the checked transition added to each mini-batch, and compared the resulting policy to the current one. We didn't train a second policy without adding anything to mini-batches. However, in some cases, this led to situations where the copy learned better non-hacking behaviors than the current policy because it was trained for longer, and thus performed better in evaluations on average even if in some cases it expressed the reward-hacking behavior. The solution was to forecast two policies, one with the checked transition added to each mini-batch and one without.

Sensitivity to Stochasticity Evaluations in stochastic environments were found to be noisy. To mitigate this noise, we compared the two policies starting from the same set of states and using the same random seeds of the environment transition model. We also ensured consistency of the updates by keeping the random seeds the same while sampling mini-batches.

Handling Inconsistent Transitions We observed that if a utility-inconsistent transition was removed from the replay buffer and another utility-inconsistent transition occurred in the same episode, the algorithm sometimes failed to detect the inconsistency the second time because there was no set of transitions in the replay buffer connecting this second utility-inconsistent transition to the starting state. To resolve this issue, we reset the environment every time the agent detected utility inconsistency. In practical applications, it would be reasonable to assume that after detecting potential reward hacking, the agent would be returned to a safe state instead of continuing exploration.

Irreversible Changes In the *Rocks and Diamonds* environment, when comparing policies starting from the current state after the rock was pushed into the goal area, the comparison results were always the same, as it was impossible to move the rock out of the goal area. We addressed this by evaluating not only from the current state but also from the initial state of the environment. This underscores the importance of future research into detecting utility inconsistency before it occurs or avoiding irreversible changes in general.

Small Inconsistent Rewards In the *Rocks and Diamonds* environment, the agent sometimes failed to detect the utility inconsistency if the switch changed the reward of the stone to +1. Qualitative analysis revealed that during inconsistency checks, our agent didn't always learn the optimal policy of pushing both the diamond and the rock into the goal area since the reward model didn't learn to add together those rewards. With just a rock in a goal area, and switch on, the reward was not higher than just pushing the diamond in, so the agent didn't learn to do that either. Thus, both the policy learned with inconsistent transition and the policy learned without it behaved identically and the inconsistency was not detected. After updating from such a transition, the agent's current utility no longer assigned negative utility to trajectories pushing the rock when the lever was pressed. We sidestepped this issue by changing the reward for the rock to +10. This issue would also be resolved if the reward model generalized better to add the rewards from different sources.

F. Computational Requirements

All experiments were conducted on workstations equipped with Intel® Core™i9-13900K processors and NVIDIA® GeForce RTX™4090 GPUs. The experiments in the *Absent Supervisor*, *Tomato Watering*, and Reacher environments each required 2 GPU-days, running 10 seeds in parallel. In the *Rocks and Diamonds* environment, experiments took 3 GPU-days, while in the *Box Moving* environment, they required 2 hours each. In total, all the experiments described in this paper took approximately 12 GPU-days, including around 1 GPU-day for training the baselines.

G. Hyperparameters

All hyperparameters are listed in Table 3. Our algorithm introduces several additional hyperparameters beyond those typically used by standard RL algorithms:

Reward Model Architecture and Learning Rate Hyperparameters specify the architecture and learning rate of the reward model $\dot{R}$. Since learning a reward model is a supervised learning task, these hyperparameters can be tuned on a dataset of transitions collected by any policy. The reward model architecture may be chosen to match the Q-function $\dot{Q}$.

Inconsistency check training steps l This parameter describes the number of updates to the Q-function needed to predict the future policy based on a new transition. As shown in Figure 5a, this value must be sufficiently large to update the learned values and corresponding policy. It can be selected by artificially adding a transition that alters the optimal policy and observing the number of training steps required to learn the new policy.

Inconsistency check rollout steps h This parameter controls the length of the trajectories used to compare two predicted policies. The trajectory length must be adequate to reveal behavioral differences between the policies. In this paper, we used a fixed, sufficiently large number. In episodic tasks, a safe choice is the maximum episode length; in continuing tasks, a truncation horizon typically used in training may be suitable. Computational costs can be reduced by choosing a smaller value based on domain knowledge.

Number of inconsistency check rollouts k This parameter specifies the number of trajectories obtained by rolling out each predicted policy for comparison. The required number depends on the stochasticity of the environment and policies. If both the policy and environment are deterministic, k can be set to 1. Otherwise, k can be selected using domain knowledge or replaced by employing a statistical significance test.

Predicted reward difference threshold This threshold defines the minimum difference between the predicted and observed rewards for a transition to trigger an inconsistency check. As discussed in Section 4.4, this parameter does not impact the algorithm's performance and can be set to 0. However, it can be adjusted based on domain knowledge to speed up training by minimizing unnecessary checks. The key requirement is that any reward hacking behavior must increase the reward by more than this threshold relative to the reward predicted by the reward model.

Table 1. Hyperparameters used for the experiments.

Hyperparameter Name	**Value**
$\dot{Q}$ and $\dot{R}$ hidden layers	2
$\dot{Q}$ and $\dot{R}$ hidden layer size	128
$\dot{Q}$ and $\dot{R}$ activation function	ReLu
$\dot{Q}$ and $\dot{R}$ optimizer	Adam
$\dot{Q}$ learning rate	0.0001
$\dot{R}$ learning rate	0.01
$\dot{Q}$ loss	SmoothL1
$\dot{R}$ loss	L_2
Batch Size	32
Discount factor γ	0.95
Training steps on *Safe*	10000
Training steps on *Full*	10000
Replay buffer size	10000
Exploration steps	1000
Exploration ϵ_{start}	1.0
Exploration ϵ_{end}	0.05
Target network EMA coefficient	0.005
Inconsistency check training steps l	5000
Inconsistency check rollout steps h	30
Number of inconsistency check rollouts k	20
Predicted reward difference threshold δ	0.05
Add transitions from transition model	False

G.1. Environment-specific Parameters

Table 2. Environment-specific hyperparameters overrides.

Hyperparameter Name	**Value**
Box Moving	
Training steps on *Safe*	1000
Training steps on *Full*	1000
Replay buffer size	1000
Exploration steps	100
Inconsistency check training steps l	500
Absent Supervisor	
Number of supervisors	1
Remove walls	False
Tomato Watering	
Number of inconsistency check rollouts k	100
Rocks and Diamonds	
Training steps on *Safe*	15000
Training steps on *Full*	15000
Inconsistency check training steps l	7500
Add transitions from transition model	True

The training steps in the Box Moving environment were reduced to speed up the training process. *Tomato Watering* has many stochastic transitions because each tomato has a chance of drying out at each step. To increase the robustness of evaluations, we increased the number of inconsistency check rollouts k. *Rocks and Diamonds* required more steps to converge to the optimal policy. Additionally, using the transition model to collect fresh data while checking for utility inconsistency in *Rocks and Diamonds* makes inconsistency detection much more reliable. Each environment's rewards were scaled to be in the range [-1, 1].

G.2. Hyperparameters of MC-TD3

Table 3. Hyperparameters used for the MC-TD3 experiment.

Hyperparameter Name	**Value**
Actor, critic, and reward model hidden layers	2
Actor, critic, and reward model hidden layer size	256
Actor, critic, and reward model activation function	ReLu
Actor, critic, and reward model optimizer	Adam
Actor and critic learning rate	0.0003
$\dot{R}$ learning rate	0.003
Batch Size	256
Discount factor γ	0.99
Training steps	200000
Replay buffer size	200000
Exploration steps	30000
Target networks EMA coefficient	0.005
Policy noise	0.01
Exploration noise	0.1
Policy update frequency	2
Inconsistency check training steps l	10000
Inconsistency check rollout steps h	50
Number of inconsistency check rollouts k	100
Predicted reward difference threshold	0.05

We didn't perform extensive hyperparameter tuning, most hyperparameters are inherited from the implementation provided by Huang et al. (2022).