Value Function Decomposition for Iterative Design of Reinforcement Learning Agents

James MacGlashan[†]
james.macglashan@sony.com

Evan Archer[†*]
evan.archer@sony.com

Alisa Devlic[†*]
alisa.devlic@sony.com

Takuma Seno[†*]
takuma.seno@sony.com

Craig Sherstan[†*]
craig.sherstan@sony.com

Peter R. Wurman[†]
peter.wurman@sony.com

Peter Stone[†‡]
pstone@cs.utexas.edu

Abstract

Designing reinforcement learning (RL) agents is typically a difficult process that requires numerous design iterations. Learning can fail for a multitude of reasons, and standard RL methods provide too few tools to provide insight into the exact cause. In this paper, we show how to integrate *value decomposition* into a broad class of actor-critic algorithms and use it to assist in the iterative agent-design process. Value decomposition separates a reward function into distinct components and learns value estimates for each. These value estimates provide insight into an agent's learning and decision-making process and enable new training methods to mitigate common problems. As a demonstration, we introduce SAC-D, a variant of soft actor-critic (SAC) adapted for value decomposition. SAC-D maintains similar performance to SAC, while learning a larger set of value predictions. We also introduce decomposition-based tools that exploit this information, including a new reward *influence* metric, which measures each reward component's effect on agent decision-making. Using these tools, we provide several demonstrations of decomposition's use in identifying and addressing problems in the design of both environments and agents. Value decomposition is broadly applicable and easy to incorporate into existing algorithms and workflows, making it a powerful tool in an RL practitioner's toolbox.

1 Introduction

Deep reinforcement-learning (RL) approaches have achieved successes in a range of application areas such as gaming ([5, 32, 36, 38]), robotics ([22]), and the natural sciences ([25, 35]). Despite these successes, applying RL techniques to complex control problems remains a daunting undertaking, where initial attempts often result in underwhelming performance. Unfortunately, there are many reasons why an agent may fail to learn a good policy, making it difficult to diagnose which reason(s) caused a particular agent to fail. For example: an agent may fail because the state features were insufficient to make accurate predictions, different task objectives defining the reward function may be imbalanced, the agent may fail to sufficiently explore the state-action space, values may not accurately propagate to more distant states, the neural network may not have sufficient capacity to

[†]Sony AI
[*]Equal contribution
[‡]The University of Texas at Austin

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

approximate the policy or value function(s), or, there may be subtle differences between training and evaluation environments. Without a way to diagnose the causes of poor performance or to recognize when a problem has been remedied, practitioners typically engage in a long trial-and-error design process until an agent reaches a desired level of performance. Frustrations with this trial-and-error process have been expressed in other work [16].

We describe how *value decomposition*, a simple, broadly-applicable technique, can address these application challenges. In RL, the agent receives a reward that is often a sum of many reward components, each designed to encode some aspect of the desired agent behavior. From this *composite reward*, it learns a single *composite value function*. Using value decomposition, an agent learns a *component value function* for each reward component. To perform policy optimization, the composite value function is recovered by taking a weighted sum of the component value functions. While prior work has proposed value decomposition methods for discrete-action Q-learning [19, 29, 31], we show how value decomposition can be incorporated into a broad class of actor-critic (AC) methods. In addition, we introduce SAC-D, a version of soft actor-critic (SAC) [13, 14] with value decomposition, and explore its use in multi-dimensional continuous-action environments. We also introduce the *influence* metric, which measures how much an agent's decisions are affected by each reward component.

While earlier work focuses on its use in reward design [16, 19], value decomposition can facilitate diagnosis of a wide range of issues and enable new training methodologies. To demonstrate its utility, in Sec. 6 we show how to use it to: (1) diagnose insufficient state features; (2) diagnose value prediction errors and exploit the decomposed structure to inject background-knowledge; and (3) identify reward components that are inhibiting exploration and mitigate the effect by gradually incorporating component predictions into policy optimization.

Value decomposition's additional diagnostic and training capabilities come at the cost of a more-challenging prediction problem: instead of learning a single value function, many must be learned. To investigate if this difficulty negatively impacts agent performance, we compare the average performance of SAC-D to SAC on benchmark environments. We find that a naive implementation of SAC-D underperforms SAC and then show how to improve SAC-D so that it matches and sometimes exceeds SAC's performance. These improvements may also be applied to value decomposition for other AC algorithms.

While variations of value decomposition has been explored extensively in past work (see Sec. 7), in this paper, we make the following contributions. (1) We show how to integrate value decomposition into a broad class of actor-critic algorithms. (2) We analyze the performance of different implementations of value decomposition for SAC on a range of benchmark continuous-action environments. (3) We introduce the *influence* metric: a novel value decomposition metric for measuring how much each reward component affects decision-making. (4) We provide a set of illustrative examples of how value decomposition and influence can be used to diagnose various kinds of learning challenges. (5) We describe new training methods that exploit the value decomposition structure and can be used to mitigate different learning challenges.

2 Background

2.1 MDPs and Q-functions

In RL, an agent's interaction with the environment is modeled as a Markov Decision Process (MDP): (S, A, P, R, γ), where S is a set of states, A is a set of actions, $P : S \times A \times S \to \mathbb{R}$ is a state transition probability function $P(s, a, s') = \Pr(S_{t+1} = s' | S_t = s, A_t = a)$, $R : S \times A \to \mathbb{R}$ is a reward function $R(s, a) = E[R_{t+1} | S_t = s, A_t = a]$, and $\gamma \in [0, 1]$ discounts future rewards.[4] The goal of an agent is to learn a policy $\pi(a|s)$ that maps states to an action probability distribution that maximizes the sum of future rewards. The agent is trained to maximize the discounted return $E\left[\sum_t^\infty \gamma^t R(s_t, a_t)\right]$. The Q-function maps state-action pairs to the expected cumulative discounted reward when starting in state s, taking action a, and then following policy π thereafter:

$$Q^\pi(s, a) \triangleq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | \pi, s_0 = s, a_0 = a\right]. \tag{1}$$

[4]For continuing tasks, γ must be < 1, and we only consider algorithms for $\gamma < 1$.

2.2 Soft actor-critic

Soft actor-critic (SAC) [13, 14] is an off-policy actor-critic algorithm parameterized with five neural networks: a stochastic policy network π with parameters ϕ, and two pairs of Q-functions and target Q-functions with parameters (θ_1, θ_2) and ($\bar{\theta}_1$, $\bar{\theta}_2$), respectively. As with other actor-critic algorithms, SAC has two main steps: policy evaluation (in which it estimates the Q-function for policy π), and policy improvement (in which it optimize the policy to maximize its Q-function estimates). Unlike other actor-critic algorithms, SAC optimizes a maximum entropy formulation of the MDP, in which rewards are augmented with policy entropy bonuses that prevent premature policy collapse. To perform policy evaluation and improvement, SAC minimizes the following loss functions simultaneously:

$$L_{Q_i} = \mathbb{E}\left[\frac{1}{2}\left(Q(s,a;\theta_i) - y\right)^2\right] \text{ for } i \in \{1,2\}\,, \tag{2a}$$

$$L_\pi = \mathbb{E}\left[\alpha \log \pi(u|s;\phi) - \min_{j\in\{1,2\}} Q(s,u;\theta_j)\right], \tag{2b}$$

where (s,a,r,s') transitions are drawn from an experience replay buffer, $y := r + \gamma\left(\min_{j\in\{1,2\}} Q(s',a';\bar{\theta}_j) - \alpha \log \pi(a'|s';\phi)\right)$, $a' \sim \pi(\cdot|s';\phi)$, $u \sim \pi(\cdot|s;\phi)$, and α is an (optionally learned) entropy regularization parameter. The $\min$ of Q-function pairs addresses overestimation bias in value function estimation [11, 15]. The parameters $\bar{\theta}_1$ and $\bar{\theta}_2$ are updated toward θ_1 and θ_2 via an exponentially moving average each step.

2.3 Environments

Throughout the paper, we italicize descriptive names for the components of the continuous-action Lunar Lander (LL), Bipedal Walker (BW) and Bipedal Walker Hardcore (BWH) environments. In LL, an agent must land a spacecraft in the center of a landing zone using as little fuel as possible. The reward components include: a reward for successful *landing*; penalties for crashing (*crash*) and engine usage (*main*, *side*); shaping rewards used to encourage the agent to stay upright (*angle*), move towards the center of the landing pad (*position*) with low velocity (*velocity*) and land with both legs (*right leg*, *left leg*). In BW, an agent learns to make a 2-legged robot walk. The reward components include: a reward for *forward* progress, a penalty for falling (*failure*), a cost for actions (*control*), and a shaping reward to discourage head movement (*head*). BWH is identical to BW, but adds additional obstacles for the agent to navigate.

3 Value decomposition for actor-critic methods

Most RL algorithms estimate the value function and use it to improve its policy. Unfortunately, value functions and policies provide little insight into the agent's decision-making. However, reward functions are often *composite* functions of multiple *component* state-action signals. By learning a value function estimate for the current policy for each component, practitioners gain insight into what the agent expects to happen and how these reward components interact. Naturally, policy improvement still requires the composite value function. In Sec. 3.1 we show how the composite Q-function can be recovered from the component Q-functions. From this property, a range of actor-critic algorithms can be adapted to use value decomposition by following the below template.[5]

1. Alter Q-function networks to have m outputs instead of 1, where m is the number of reward components.
2. Use the base algorithm's Q-function update for each of the m components, replacing the composite reward term with the respective component reward term.
3. Apply the base algorithm's policy improvement step by first recovering the composite Q-function.

For example, this template can be applied to algorithms that use TD(0) [33], or ones that use Retrace [27]. It works with algorithms that improve the policy by differentiating through the Q-function [11, 14, 23], and ones that fit it toward non-parametric target action distributions [1].

[5] Composite state value functions can similarly be recovered; however, Q-functions allow for deeper introspection, so we focus on that setting in this work.

Algorithm 1 SAC-D and SAC-D-CAGrad Update

Require: Experience replay buffer B; twin Q-function parameters θ_1, θ_2 (with $\Theta = \theta_1 \cup \theta_2$) and target parameters $\bar{\theta_1}, \bar{\theta_2}$; policy parameters ϕ; discount factor γ; entropy parameter α; reward weights $w \in \mathbb{R}^{m+1}$; learning rates λ_q, λ_π; target network step size η; Boolean use_cagrad for SAC-D-CAGRAD or SAC-D.

1: Sample transition (minibatch) $(s, a, r, s') \sim B$ ▷ $r \in \mathbb{R}^m$ is a vector of m reward components
2: Sample policy actions $a' \sim \pi(\cdot|s'; \phi)$ and $u \sim \pi(\cdot|s; \phi)$
3: $r_{m+1} \leftarrow \gamma\alpha \log \pi(a'|s'; \phi)$ ▷ Extend reward vector to include entropy reward
4: $j \leftarrow \underset{j \in \{1,2\}}{\arg\min} \sum_i^{m+1} w_i Q_i(s', a'; \bar{\theta}_j)$ ▷ Find target network by minimum composite Q-value
5: $y_i \leftarrow r_i + \gamma Q_i(s', a'; \bar{\theta}_j)$
6: $LQ_i \leftarrow \sum_{j=1}^{2} \frac{1}{2} \left(Q_i(s, a; \theta_j) - y_i\right)^2$
7: $L\pi \leftarrow \alpha \log \pi(u|s; \phi) - \min_{j \in \{1,2\}} \sum_i^{m+1} w_i Q_i(s, u; \theta_j)$
8: **if** `use_cagrad` **then**
9: $\quad \Theta \leftarrow \Theta - \lambda_q \text{CAGRAD}(\mathbf{J}_{LQ}, \Theta)$
10: **else**
11: $\quad \Theta \leftarrow \Theta - \lambda_q \nabla_\Theta \frac{1}{m+1} \sum_i^{m+1} LQ_i$
12: **end if**
13: $\phi \leftarrow \phi - \lambda_\pi \nabla_\phi L\pi$
14: Update target networks $\bar{\Theta} \leftarrow (1 - \eta)\bar{\Theta} + \eta\Theta$

Although this template is conceptually simple, learning component Q-functions poses a more difficult prediction problem: multiple predictions must be learned instead of one composite prediction. Ideally, this increased difficulty would not negatively impact agent performance. In Sec. 3.2 we introduce SAC-D, an adaptation of SAC to use value decomposition, and describe additions we made to the above template to maintain performance with conventional SAC. Although these additions are contextualised to SAC, they are general and can be used when adapting other actor-critic algorithms.

3.1 Recovering the composite Q-function

We assume the environment's reward function is a linear combination of m components: $R(s, a) \triangleq \sum_i^m w_i R_i(s, a)$, where $w_i \in \mathbb{R}$ is a scalar *component weight* for the ith component and $R_i(s, a) \to \mathbb{R}$ is the reward function of the ith component for state-action pair (s, a). Applying the linearity of expectation, we find the Q-function inherits the linear structure from the reward[6]:

$$Q^\pi(s, a) = \sum_i^m w_i Q_i^\pi(s, a), \tag{3}$$

where we define the i'th *component Q-function* as $Q_i^\pi(s, a) \triangleq E[\sum_t \gamma^t R_i(s_t, a_t)|\pi, s_0 = s, a_0 = a]$. Unless otherwise specified, we assume $w_i = 1$ for all i. Because the component weights are factored out of the component Q-functions, they may be varied without changing the component prediction target, allowing the policy to be evaluated for any weight combination. Although the assumption of linearity may seem restrictive, note that each reward component may be a non-linear function of state variables, allowing for very expressive environment rewards. Furthermore, many environments, including all the environments we investigate in this paper, are naturally structured as a sum of (non-linear) reward components.

3.2 SAC with value decomposition

Here we introduce SAC-D (Alg. 1), an adaptation of SAC to use value decomposition. Adapting SAC only requires one additional consideration from our template: the entropy bonus reward term SAC adds is treated as an $m + 1$'th reward component: $R_{m+1}(s') \triangleq \gamma\alpha \log \pi(a'|s'; \phi)$ (line 3). However, this approach, which we refer to as *SAC-D-Naive*, underperforms SAC in many settings. We found two additional modifications essential to match the performance of SAC. The first concerns how we apply the twin-network minimum of Eq. 2 in the context of value decomposition. The second is to use **C**onflict-**A**verse **Grad**ient descent (CAGrad) [24] to address optimization problems that arise when training multi-headed neural networks. We refer to SAC with value decomposition and the

[6]See Theorem A.1 for the proof. This linear decomposition property of value functions has been explored elsewhere [3, 9], but in different contexts and with different motivations. See Sec. 7 for more information.

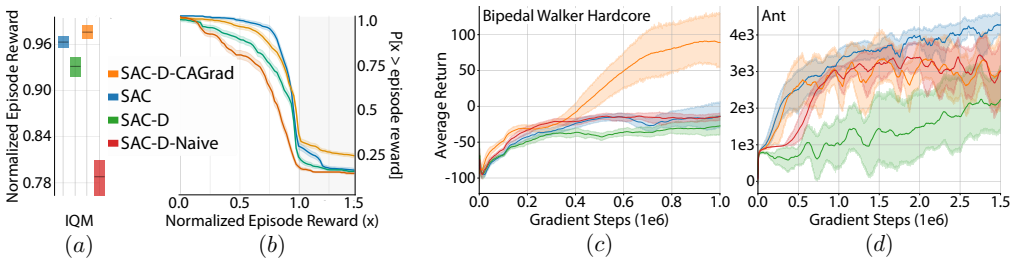


Figure 1: **Benchmark results (a)-(b)** Comparison of algorithms over 8 environments. For each algorithm, scores are collected from the 10 final checkpoints (10 episodes $\times$ 10 final policies), and normalized on individual environments. The 95% CIs are estimated using stratified bootstrap sampling. **(a)** The interquartile mean (IQM), which gives the mean over the central 50% of values. **(b)** For each normalized score x, % of runs achieving a score of at least x. Grey region, $x > 1.0$, corresponds to runs outperforming the mean SAC baseline score. **(c)** Training curves on continuous-control benchmarks for BWH. Solid lines represent mean episode return over 10 trials. Shaded areas are confidence intervals. Lines are uniformly smoothed for clarity. **(d)** Same as **(c)**, but for Ant.

twin-network correction as *SAC-D*, and the variant with twin-network correction and CAGrad as *SAC-D-CAGrad*.

Twin-network minimums in value decomposition: In SAC, the Q-value target is the minimum of two Q-function networks (Eq. 2). Using the same Q-value update rule for each component, as described in our template, suggests using a minimum for each component target: $q_i := \min_{j\in\{1,2\}} Q_i(s, a; \bar{\theta}_j)$. However, this is not a good choice in practice. The purpose of the twin-network minimum is to mitigate overestimation bias from the feedback loop of the policy optimizing the Q-function. Because the policy optimizes the composite Q-function, a better approach is to use all the predictions from the network with the minimum composite Q-function (Alg. 1, lines 5-6)). This approach reduces underestimation bias and improves performance compared to an element-wise minimum (see Sec. 4).

Mediating the difficulty of multi-objective optimization: Even though the scalar values of the composite Q-function are identical to those used in SAC, simultaneous optimization of all Q_i^π components may introduce training problems common in multi-objective optimization: conflicting gradients, high curvature and large differences in gradient magnitudes [39].[7] The CAGrad method, designed for the multi-task RL setting, addresses these issues by replacing the gradient of a multi-task objective with a weighted sum of per-task loss gradients. This updated gradient step maximizes the improvement of the worst-performing task on each optimization step, and still converges to a minimum of the unmodified loss. We incorporate CAGrad into SAC-D by treating each component as a "task", and update the gradient vector accordingly (Alg. 1, line 9).

4 Robustness experiments

We benchmark SAC-D, SAC-D-CAGrad and SAC-D-Naive against SAC on a selection of continuous-action Gym [7] environments. For each environment, we exposed existing additive reward components *without* altering the behavior of the environments or their composite rewards. That is, these environments already implemented their reward functions as a linear combination of separate reward components and we simply exposed that information to the algorithm (for details, see App. B). As outlined in App. C, we used hyperparameters previously published for use with SAC [14] for all experiments. We tied SAC-D's hyperparameters to SAC's because our goal is for value decomposition to be a drop-in addition without significant loss in agent-performance. However, it is possible better performance could be reached with tuning.

Figure 1(a) shows the performance of each algorithm aggregated across all environments and all experimental runs. Figure 1(b) shows the same information, but highlights the distribution of scores across experimental runs. In aggregate, SAC-D-CAGrad slightly outperforms SAC, although it has a broader range of performance scores. SAC-D-Naive significantly underperforms SAC.

[7] Multi-headed prediction can also improve representation learning, as in work on auxiliary tasks [18, 26]

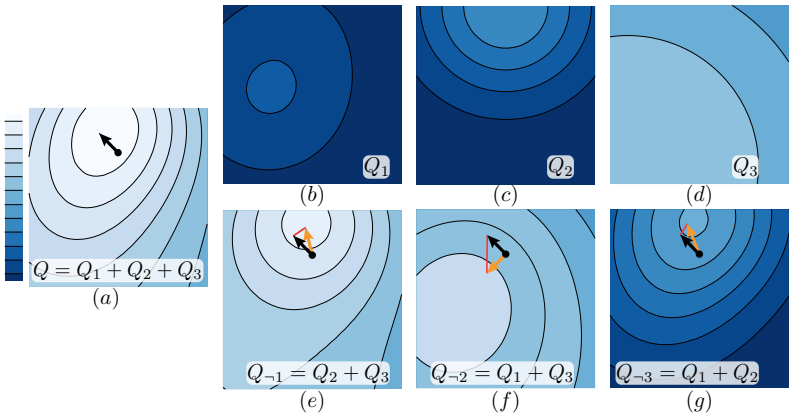


Figure 2: **Influence calculation for a three-component environment with a 2D action space.** In this illustrative example, component 2 has high influence, while components 1 and 3 have low influence, but for different reasons. Each figure shows a contour-map view of the Q-function over the 2D action space for the same state. **(a)** The composite Q-function the policy optimizes. The black point is a (near-optimal) policy selection, and the black arrow is the policy gradient. **(b)-(d)** Each component Q-function. **(e)-(g)** Component-ablated composite Q-function surfaces $Q_{\neg i}$ used to compute influence for each component. Influence is defined by the length of the red line, which is the difference of the policy gradients computed for the composite Q-function Q (black arrow) and for $Q_{\neg i}$ (orange arrow). **(e)** Component 1 has low influence because it is out-weighed by component 2. **(f)** Conversely, component 2 has high influence; without it, the policy would move toward the peak of component 1. **(g)** While component 3 contributes a large value, it has low influence because the value is nearly uniform; much of its value is received regardless of the action selected.

We provide training curves for the 8 environments investigated in App. D, but highlight the atypical training curves for BWH (Fig. 1(c)) and Ant (Fig. 1(d)). In the case of BWH, SAC-D-CAGrad significantly outperforms SAC. It was not the goal of this work for SAC-D to improve on SAC. Rather, we sought to provide more insights into the learning process. As such, we make no strong claims about when SAC-D can be expected to outperform SAC. Nevertheless, this result does suggest that SAC-D may sometimes benefit from auxiliary task learning. We leave this question as a subject for future investigation.

In Ant, SAC-D (without CAGrad) underperforms all other methods. We found that infrequent environment termination causes large Q-function errors and leads to catastrophic gradient conflicts. Further analysis of termination issues and CAGrad behavior is described in Appendices E and F respectively.

5 Reward component influence

It can be difficult to understand how an agent's predictions interact to affect decision-making. We now introduce the reward *influence* metric, which indicates how much each component contributes to an agent's decision. Intuitively, low influence means that removing a component would not alter decision-making; high influence means that removing it would significantly alter decision-making.

For multi-dimensional continuous actions, we define the *optimal influence* of component i in state s by how much the optimal policy π^* in state s differs from the optimal policy when component i is removed: $\mathcal{I}_i^*(s) \triangleq ||\pi^*(s) - \pi^*_{\neg i}(s)||_2$, where $\pi^*_{\neg i} \triangleq \arg\max_\pi E\left[\sum_t^\infty \gamma^t \sum_{j \neq i} w_j R_j(s_t, a_t)|\pi\right]$.[8]

In practice, we apply two approximations to $\mathcal{I}_i^*(s)$ for computational efficiency. First, rather than compare the difference of optimal policies, we compare the difference between one step of policy improvement: $\mathcal{I}_i^\pi(s) \triangleq ||\arg\max_a Q^\pi(s,a) - \arg\max_a Q^\pi_{\neg i}(s,a)||_2$, where $Q^\pi_{\neg i}(s,a) \triangleq \sum_{j \neq i} w_j Q^\pi_j(s,a)$. Second, since $\arg\max$ is computationally demanding and sensitive to statistical noise, we replace the $\arg\max_a Q(s,a)$ policy improvement operator with a policy gradient-step

[8]Discrete-action spaces could use a probability distance measure, but we focus on continuous-action spaces.

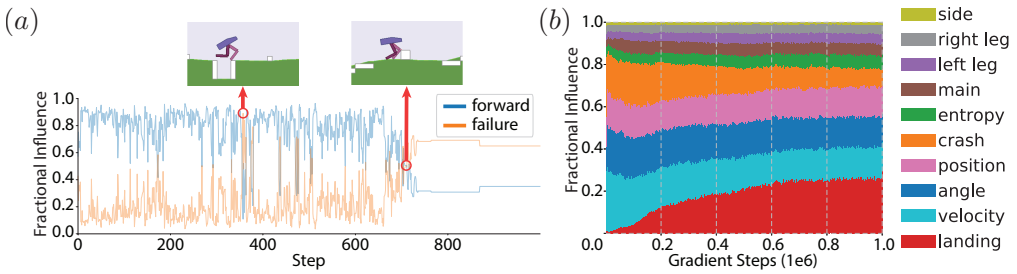


Figure 3: **Fractional influence (a)** Fractional influence for 1 evaluation trajectory of BWH. For this agent and trajectory, the *forward* component dominates the *failure* component in decision-making, except at two points (red arrows). At the first point, the agent was at risk of falling into a pit. After the agent stabilizes, the *forward* component returns to being dominant. After the second point, the agent becomes unable to move forward. *Failure*'s dominance indicates that if the agent cannot move forward, it prefers standing to falling. If it is possible to surmount this obstacle, more exploration may be required. **(b)** Mean stacked fractional influence across 10 trials of Lunar Lander. We observe: (1) the sparse landing reward doesn't meaningfully contribute to decision-making in the earliest part of training; (2) crashing influence starts high and then decreases, suggesting the agent initially learns to avoid crashing; (3) the position, angle, and velocity reward components – each a shaping reward – maintain large influence, suggesting possible overreliance on shaping rewards; (4) numerous reward components never have much influence; they might be unnecessary or require more attention.

operator (typical in RL algorithms like SAC): $\bar{a} + \lambda \nabla_{\bar{a}} Q(s, \bar{a})$, where $\bar{a}$ is a deterministic policy action selection (such as the mode) and $\lambda \in (0, 1)$ is a step size.

When taking the difference of the gradient-step operator applied to the Q^π and $Q^\pi_{\neg i}$ surfaces, the $\bar{a}$ terms cancel, and λ can be factored out of the norm. The result is the *influence* metric (Fig. 2):

$$I_i^\pi(s;\theta) \triangleq \lambda ||\nabla_{\bar{a}} Q^\pi(s, \bar{a}; \theta) - \nabla_{\bar{a}} Q^\pi_{\neg i}(s, \bar{a}; \theta)||_2. \tag{4}$$

The raw magnitudes of component influence can be informative by themselves; for example, a sharper Q-function surface leads to larger influence values. However, to compare influence values across components, we typically compute the *fractional influence* by normalizing the (always non-negative) influence: $\hat{I}_i^\pi(s;\theta) \triangleq \frac{I_i^\pi(s;\theta)}{\sum_j^m I_j^\pi(s;\theta)}$.

We use several techniques to visualize the fractional influence. For trajectories, plotting influences over timesteps may help identify and explain key decision points (Fig. 3(a)). When studying an agent's behavior across training, we maintain summary statistics of fractional influence. We visualize mean fractional influence across all components as a stack plot, sorted so that the component with the maximum influence at the end of training is at the bottom. Figure 3(b) shows such a diagram for the Lunar Lander environment; we provide figures for all environments in App. G.

6 Value decomposition: strategies for agent design

Agent design is often a brute-force process of trial and error: when an agent doesn't perform as expected, we choose some aspect of the agent's design to vary, and then we train again. Although this approach can succeed, it can be expensive in both time and computation.

In this section, we illustrate a different approach, showing how a decomposed reward helps break from trial-and-error by encouraging the designer to consider an agent's point of view. In three examples, each using either the LL or BWH environments (Sec. 2.3), we draw on value decomposition tools to: (1) identify learning problems by comparing components' empirical returns to their predictions; (2) constrain component value estimation; (3) identify adverse reward interactions with the influence metric; (4) dynamically re-weight reward components.

These examples are not just meant to demonstrate these specific techniques, nor to demonstrate significant performance improvements on these well-tested benchmark environments (in fact, SAC-D eventually produces good policies for both environments without additional tuning). Rather, the goal is to showcase a general approach to agent design for novel applications in which iteration and

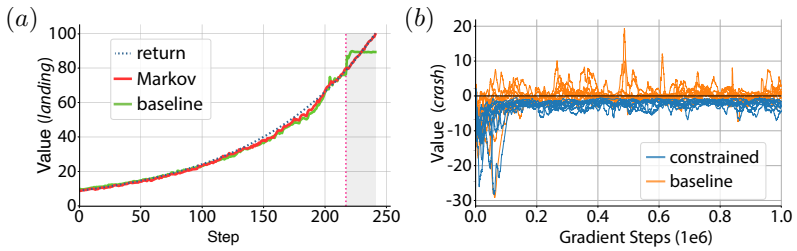


Figure 4: **Remedies for incorrect value predictions.** **(a)** For the *landing* component empirical return (dashed blue), the value prediction (green) for a value function trained with baseline features matches the return well except for a final plateau in the shaded region. Training a value function with the added V_0^{trace} feature (red) improves predictions in this region. The dashed pink line indicates the point after which we compute error metrics in App. H. **(b)** Value predictions for the *crash* component with sign constraints (blue lines; using both target and prediction constraints) and without (orange lines). Each line represents an independent trial. Unconstrained estimates can be positive; constrained value estimates keep predictions negative.

failure is costly. Combined with statistical tools that allow us to reason with small-sample statistics, value decomposition provides a vocabulary to describe an agent's behavior and interpretable tools for targeted interventions. Even these small examples are more intuitive – and less computationally demanding – than they would be with a trial-and-error approach.

For all examples, training parameters are identical to the robustness results of Sec. 4.

Diagnosing and improving insufficient features: We analyze the behavior of an agent trained on Lunar Lander by comparing each component's empirical return to the agent's value predictions, Q_i. The agent is trained to land successfully, and generally the component value predictions match their empirical returns well. Curiously, all component predictions are flat near the end of the episode. For most components, these flat predictions are a good match for their returns, but not for *landing*. Investigating the landing dynamics, we found the simulator waits many steps after touchdown before producing a landing reward. During this period, the observations are constant, suggesting the features are inadequate to represent *landing*'s return. To make the observations Markov, we introduce a new feature to the agent's observations that indicates the duration since the agent's velocity (horizontal, vertical and angular) went to zero: $V_0^{\text{trace}}(t) = V_0^{\text{steps}}(t)/c$, where $V_0^{\text{steps}}(t)$ is the number of time steps since all the velocities dropped below a threshold and c is a fixed normalizing constant. With this feature, post-landing predictions show a marked improvement (see Figure 4(a) and Appendices C, H).

Diagnosing and mitigating value errors using domain knowledge: Lunar Lander's design makes clear that certain reward components are always non-positive (*crash, main, side*) while others are always non-negative (*landing*). However, we observe that the agent's decomposed value predictions do not always match these bounds. In particular, value predictions of *crash* have a tendency to oscillate about 0 after the agent learns to land. Value decomposition allows us to explicitly enforce a sign constraint on *crash* (Figure 4(b); see App. I for details). In this particular example, constraints do not alter policy learning performance, but the resulting predictions are easier to interpret, and the same technique may improve performance in more complex environments.

Mitigating an adverse reward with component weight scheduling: Under the BWH reward function, a random policy is far more likely to experience an unsuccessful outcome (falling over) than a successful one (walking forward). This bias can inhibit agent exploration early in training, causing an agent's policy to fall into a local minimum (the agent stands still). Here, we diagnose this dynamic using component predictions and influence metrics, and remedy it by varying a single component weight during training.

We find that the *failure* component's fractional influence dominates the *forward* component's fractional influence early in training, and that this relationship reverses as agent performance improves (Fig. 5(a)). The *forward* component's near-zero value predictions (Fig. 5(b)) early in learning indicate that in many episodes, the agent neither moves nor expects to move (Fig. 5(c)). Early dominance by the easy-to-observe *failure* suggests that it is inhibiting exploration.

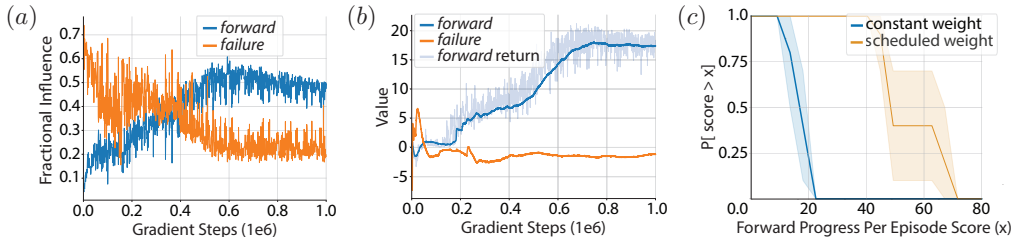


Figure 5: **Remedy for an adverse reward (a)** The initially low fractional influence of *forward* (blue curve) indicates it may be initially inhibited by failure penalties with higher influence. **(b)** The *forward* value predictions and returns (blue curves) are near zero early in training, indicating the agent does not expect to move forward. **(c)** An agent trained with its failure weight scheduled from a value close to zero up to one (orange curve) improves early forward progress compared to an agent with a constant failure weight (blue curve). Forward progress is measured as the average forward progress return from all evaluation episodes of the first 200k gradient steps. Shaded areas represent 95% bootstrap confidence intervals from 10 trials.

To mitigate this problem, we vary the *failure* component weight, w_{failure}, from 0.01 to 1 over the course of learning according to the schedule described in App. J. This schedule significantly increases the agents' *forward* progress (Fig. 5(c)), and accelerates learning.

7 Related work

Our work builds upon earlier studies of value decomposition in the explainable reinforcement learning (XRL) literature: DrQ [19], DrSARSA [29], and HRA [31].[9] Like our approach, these methods learn separate value function estimates for each term of a linear reward function. DrQ and HRA are off-policy Q-learning-like methods, while DrSARSA is an on-policy method. HRA does not converge to a globally optimal policy, as each value function is only *locally*-optimal for the reward component it measures. The RDX and MSX metrics proposed by DrQ could be adopted in our setting, but the influence metric is easier to use with continuous-actions, and aggregate with summary statistics. Our approach improves upon these prior contributions by: (1) working in continuous-action and discrete-action environments; (2) allowing for and demonstrating dynamic re-weighting of reward components during training; and (3) being applicable to a family of actor-critic methods. While these approaches and ours explore using value decomposition for explainability, these works focus on describing why an agent took certain actions to users, whereas we focus on how to use value decomposition to diagnose and remedy learning challenges.

The Horde architecture [34] and UVFA [30], methods for multi-goal learning, also employ multiple value functions (one for each goal). UVFAs use a parameterized continuous space of goals, while Horde makes multiple discrete value function predictions. The value functions in our work are conditioned on a policy that optimizes the global reward, whereas in Horde and UVFAs the value functions are conditioned on independent policies that greedily optimize local goals (similar to HRA). Additionally, in our approach, the composite value function can be recovered from the components. Other value decomposition work includes Empathic Q-learning [21] and Orchestrated Value Mapping [10]. The primary difference between our work these other approaches regards the motivation and application of value decomposition. These works focus on how value decomposition can improve sample efficiency, generalization, or other aspects of the core learning problem, rather than how to diagnose and remedy problems.

Mathematically, value decomposition bears resemblance to work on successor features, which has focused primarily on state representation and transfer learning [3, 4, 6, 12]. Methods for multi-objective RL [16, 28] learn sets of policies, each with a distinct linear weight over multiple reward objectives. Our work also recovers the value function for different linear reward preferences, but uses this capability to diagnose behavior and learning problems rather than to learn multiple policies.

[9] For a broad overview of XRL, we direct the reader to Heuillet et al. [17].

8 Concluding remarks

We have argued that the iterative design of reinforcement learning agents can be improved through the use of *value decomposition*, in which we keep individual reward components separate and learn value estimates of each. We provided a simple prescription for deriving value decomposition algorithms from actor-critic methods, and applied it to SAC to derive the SAC-D algorithm. Combined with the CAGrad method, SAC-D meets or exceeds SAC's performance in all environments we tested. We introduced the *influence* metric, and demonstrated its use in measuring each reward component's effect on an agent's decisions. Finally, we provided several examples of how value decomposition can diagnose and remedy agent learning problems.

Although value decomposition is a simple and broadly applicable tool, we note the following limitations. (1) Our method requires a composite reward function of multiple components. (2) We only study linear reward decomposition. (3) Component predictions only tell you the agent expectations under the single learned policy; changing the weights doesn't tell you what to expect after re-optimizing the policy for them. (4) Our approach benefits most with methods that learn Q-functions. Methods that optimize the policy with empirical returns have a weaker link between agent expectations (component Q-functions) and policy decisions. The influence metric also requires a Q-function model.

Our method presents the same societal benefits and risks as other RL methods. However, we believe this technology has a net beneficial impact, because making agent decisions more introspectable enables developers to catch problematic behavior before deploying the technology. One particular concern, however, is that value estimates represent an agent's beliefs, not ground truth; this should be kept in mind when such predictions are used in real-world decision-making, as they may reinforce biases or lead to incorrect conclusions.

References

[1] A. Abdolmaleki, J. T. Springenberg, J. Degrave, S. Bohez, Y. Tassa, D. Belov, N. Heess, and M. Riedmiller. Relative entropy regularized policy iteration. *arXiv*, 1812.02256, 2018.

[2] R. Agarwal, M. Schwarzer, P. S. Castro, A. Courville, and M. G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[3] A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. van Hasselt, and D. Silver. Successor Features for transfer in reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[4] A. Barreto, D. Borsa, S. Hou, G. Comanici, E. Aygün, P. Hamel, D. Toyama, J. Hunt, S. Mourad, D. Silver, and D. Precup. The Option Keyboard: Combining skills in reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[5] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with large scale deep reinforcement learning. *arXiv*, 1912.06680, 2019.

[6] D. Borsa, A. Barreto, J. Quan, D. J. Mankowitz, R. Munos, H. V. Hasselt, D. Silver, and T. Schaul. Universal Successor Features approximators. *arXiv*, 1812.07626, 2019.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *arXiv*, 1606.01540, 2016.

[8] A. Cassirer, G. Barth-Maron, E. Brevdo, S. Ramos, T. Boyd, T. Sottiaux, and M. Kroiss. Reverb: A framework for experience replay. *arXiv*, 2102.04736, 2021.

[9] P. Dayan. Improving generalization for temporal difference learning: The Successor Representation. *Neural Computation*, 5(4):613–624, 1993.

[10] M. Fatemi and A. Tavakoli. Orchestrated value mapping for reinforcement learning. *arXiv preprint arXiv:2203.07171*, 2022.

[11] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning (ICML)*, 2018.

[12] C. Grimm and S. Singh. Learning independently-obtainable reward functions. *arXiv*, 1901.08649, 2019.

[13] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning (ICML)*, 2018.

[14] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications. *arXiv*, 1812.05905, 2018.

[15] H. Hasselt. Double Q-learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2010.

[16] C. F. Hayes, R. Rădulescu, E. Bargiacchi, J. Källström, M. Macfarlane, M. Reymond, T. Verstraeten, L. M. Zintgraf, R. Dazeley, F. Heintz, E. Howley, A. A. Irissappane, P. Mannion, A. Nowé, G. Ramos, M. Restelli, P. Vamplew, and D. M. Roijers. A practical guide to multi-objective reinforcement learning and planning. *Autonomous Agents and Multi-Agent Systems*, 36(1):26, Apr. 2022.

[17] A. Heuillet, F. Couthouis, and N. Díaz-Rodríguez. Explainability in deep reinforcement learning. *Knowledge-Based Systems*, 214:106685, 2021.

[18] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv*, 1611.05397, 2016.

[19] Z. Juozapaitis, A. Koul, A. Fern, M. Erwig, and F. Doshi-Velez. Explainable reinforcement learning via reward decomposition. In *IJCAI/ECAI Workshop on Explainable Artificial Intelligence*, 2019.

[20] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. *arXiv*, 1412.6980, 2014.

[21] R. Laroche, M. Fatemi, J. Romoff, and H. van Seijen. Multi-advisor reinforcement learning. *arXiv preprint arXiv:1704.00756*, 2017.

[22] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[23] 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 *International Conference on Learning Representations (ICLR)*, 2016.

[24] B. Liu, X. Liu, X. Jin, P. Stone, and Q. Liu. Conflict-averse gradient descent for multi-task learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[25] K. Mills, P. Ronagh, and I. Tamblyn. Finding the ground state of spin hamiltonians with reinforcement learning. *Nature Machine Intelligence*, 2:509–517, 2020.

[26] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, D. Kumaran, and R. Hadsell. Learning to navigate in complex environments. *arXiv*, 1611.03673, 2016.

[27] R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare. Safe and efficient off-policy reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

[28] D. M. Roijers, P. Vamplew, S. Whiteson, and R. Dazeley. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, 48:67–113, 2013.

[29] S. J. Russell and A. Zimdars. Q-decomposition for reinforcement learning agents. In *International Conference on Machine Learning (ICML)*, 2003.

[30] T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal value function approximators. In *International Conference on Machine Learning ICML*, 2015.

[31] H. V. Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes, and J. Tsang. Hybrid reward architecture for reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[32] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, 2017.

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

[34] R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, and D. Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2011.

[35] K. Tunyasuvunakool, J. Adler, Z. Wu, T. Green, M. Zielinski, A. Žídek, A. Bridgland, A. Cowie, C. Meyer, A. Laydon, et al. Highly accurate protein structure prediction for the human proteome. *Nature*, 596:590–596, 2021.

[36] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575:350–354, 2019.

[37] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[38] P. R. Wurman, S. Barrett, K. Kawamoto, J. MacGlashan, K. Subramanian, T. J. Walsh, R. Capobianco, A. Devlic, F. Eckert, F. Fuchs, L. Gilpin, P. Khandelwal, V. Kompella, H. Lin, P. MacAlpine, D. Oller, T. Seno, C. Sherstan, M. D. Thomure, H. Aghabozorgi, L. Barrett, R. Douglas, D. Whitehead, P. Dürr, P. Stone, M. Spranger, and H. Kitano. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602:223–228, 2022.

[39] T. Yu, S. Kumar, A. Gupta, S. Levine, K. Hausman, and C. Finn. Gradient surgery for multi-task learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] We outline three limitations of our approach in the Concluding Remarks.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] We devote a paragraph of Sec. 8 to these issues.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] We assume linearity of the reward function and other standard MDP requirements.
   (b) Did you include complete proofs of all theoretical results? [Yes] Our paper is largely empirical; however, we provide a proof for the Q-function identity used in the introduction of the composite Q-function.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] At present, we are unable to release our source code or data.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We report error bars in all experimental results. For results within individual environments, we report standard deviation error bars. To report aggregate results across all environments, we report bootstrap-resampled error bars computed using the `rliable` framework [2].
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We cite the creators of Gym and the `rliable` toolkit.
   (b) Did you mention the license of the assets? [N/A] We have no new assets that require licenses.
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] All empirical results are based on simultation using standard benchmark environments.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

A Value decomposition of linear reward functions

The following proof shows that composite Q-function can be recovered from the component Q-functions of a linear reward function.

Theorem A.1. *Let the reward function $R(s, a)$ of an MDP be a linear combination of a finite number of m components:*

$$R(s, a) \triangleq \sum_{i}^{m} w_i R_i(s, a),$$

where $w_i \in \mathbb{R}$ is a weight for the ith component and $R_i(s, a) \to \mathbb{R}$ defines the ith reward component for state-action pair s and a.

Let Q_i^π be the component Q-function for the ith reward component under policy π:

$$Q_i^\pi(s, a) \triangleq E\left[\sum_{t=0}^{\infty} \gamma^t R_i(s_t, a_t) | s_0 = s, a_0 = a\right].$$

Then the composite Q-function

$$\begin{aligned} Q^\pi(s, a) \triangleq & E_\pi\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | s_0 = s, a_0 = a\right] \\ = & \sum_i w_i Q_i^\pi(s, a). \end{aligned}$$

Proof. In the following proof, we use the annotation "By def." to mean substitution of a definition; "Distrib." to mean by the distributive property; "C.A." to mean by the commutativity and associativity property; and "Lin. E." to mean by the linearity of expectation property. For notational simplicity, the expectation $E_\pi[...]$ implies the conditional expectation $E_\pi[...|\, s_0 = s, a_0 = a]$.

$$\begin{aligned} Q^\pi(s, a) &\triangleq E_\pi\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right] \\ &= E_\pi\left[\sum_{t=0}^{\infty} \gamma^t \sum_{i}^{k} w_i R_i(s_t, a_t)\right] && \text{By def.} \\ &= E_\pi\left[\sum_{t=0}^{\infty} \sum_{i}^{k} \gamma^t w_i R_i(s_t, a_t)\right] && \text{Distrib.} \\ &= E_\pi\left[\sum_{i}^{k} \sum_{t=0}^{\infty} \gamma^t w_i R_i(s_t, a_t)\right] && \text{C.A.} \\ &= E_\pi\left[\sum_{i}^{k} w_i \sum_{t=0}^{\infty} \gamma^t R_i(s_t, a_t)\right] && \text{Distrib.} \\ &= \sum_{i}^{k} E_\pi\left[w_i \sum_{t=0}^{\infty} \gamma^t R_i(s_t, a_t)\right] && \text{Lin. E.} \\ &= \sum_{i}^{k} w_i E_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_i(s_t, a_t)\right] && \text{Lin. E.} \\ &= \sum_{i}^{k} w_i Q_i^\pi(s, a) && \text{By def.} \end{aligned}$$

□

B Environment reward components

The decomposed reward components used in the various environments are listed in this section. All of these are internally implemented in Gym [7], but only the aggregated scalar rewards are exposed by default. To this end, we implemented a modified version to expose these decomposed reward functions. For Ant-v3 and Humanoid-v3, there is an issue with contact calculation due to the compatibility with the latest open-sourced version of MuJoCo simulator (`https://github.com/openai/gym/issues/1541`). Therefore, we removed the contact cost component in our experiments.

LunarLander-v2.

- `main`: cost for using the main engine.
- `side`: cost for using the side engines.
- `crash`: penalty for crashing.
- `landing`: reward for landing.
- `left_leg`: shaping reward for the left leg contacting the ground.
- `right_leg`: shaping reward for the right leg contacting the ground.
- `angle`: shaping penalty for being oriented away from the vertical.
- `position`: shaping penalty to encourage the lander to come down and to the center of the landing pad.
- `velocity`: shaping penalty to drive the lander to prefer low speed.

Hopper-v3.

- `forward`: forward progress along the movement axis.
- `control_cost`: cost of actuator actions.
- `healthy`: constant alive bonus.

BipedalWalker-v3.

- `forward`: forward progress.
- `head`: steadiness of the head.
- `control_cost`: cost of actuator actions.
- `failure`: penalty at failure.

BipedalWalkerHardcore-v3.

- `forward`: forward progress.
- `head`: steadiness of the head.
- `control_cost`: cost of actuator actions.
- `failure`: penalty at failure.

HalfCheetah-v3.

- `forward`: forward progress along the movement axis.
- `healthy`: constant alive bonus.

Walker2d-v3.

- `forward`: forward progress along the movement axis.
- `control_cost`: cost of actuator actions.
- `healthy`: constant alive bonus.

Ant-v3.

- `forward`: forward progress along the movement axis.
- `control_cost`: cost of actuator actions.
- `healthy`: constant alive bonus.

Humanoid-v3.

- `forward`: forward progress along the movement axis.
- `control_cost`: cost of actuator actions.
- `healthy`: constant alive bonus.

C Experiment details

All algorithms are trained in a distributed manner with separated processes of a rollout worker that collects experiences and a trainer that updates the neural network parameters asynchronously. We used Reverb [8] to implement the experience replay buffer.

The policy network outputs the vector mean (μ) and (diagonal) log standard deviation ($\log(\sigma)$) of a squashed Gaussian distribution whose output is constrained to $(-1, +1)$. During data collection actions are sampled by first sampling the random (unsquashed) vector $Z_t \sim N(\mu(S_t), \sigma(S_t))$, and then applying $\tanh$ to that to get the action: $A_t \leftarrow \tanh(Z_t)$. The action was scaled to the appropriate domain before being executed in the environment for environments with different action domains than $(-1, +1)$. The trained policies were evaluated for 10 episodes every 1000 gradient steps. During an evaluation episode, actions were drawn deterministically from the policy by using the output mean $Z_t \leftarrow \mu(S_t)$, instead of sampling.

In SAC-D and SAC-D-Naive training, the gradients of the hidden layers are divided by the size of reward dimension. For SAC-D-CAGrad training, we implemented the CAGrad part based on the publicly available code[10] released by the authors. We found that the scipy's `minimize` function used for the inner optimization is sensitive to the gradient scale. Therefore the gradients are normalized by an averaged norm during the inner optimization. Table 1 shows the hyperparameters used in the experiments.

Table 1: Hyperparameters used in robustness experiments

Algorithm	Hyperparameter	Value
SAC	Actor learning rate	3e-4
	Critic learning rate	3e-4
	Discount factor	0.99
	Mini-batch size	256
	Entropy target	$-\|A\|$
	Target update rate	5e-3
	Hidden layers	[256, 256]
	Activation	ReLU
	Replay buffer size	1000000
	Optimizer	Adam [20]
	Steps until timeout	1000
SAC-D-CAGrad	c [24]	0.5

C.1 Computation

Our computational resources were a custom cluster of machines deployed to allow distributed training in which data collection and training are performed on different machines, the *rollout worker* and *trainer* respectively. The rollout worker used a single CPU with 2 GB of RAM (AWS c5.2xlarge:

[10] https://github.com/Cranial-XIX/CAGrad

Intel Xeon, 3.9 GHz) while the trainer used 8 CPU cores with 8 GB of RAM (AWS p3.2xlarge: Intel Xeon E5-2686 v4 processors, 2.3 GHz).

Computing the mean training steps per second for Bipedal Walker Hardcore we see that SAC-D is moderately slower than SAC, but SAC-D-CAGrad is significantly slower (Table 2). Applying CAGrad slows down the rate of computation because on each step it solves an additional optimization step in order to compute the gradient. For the values reported in Table 2 we use the default minimization algorithm from SciPy [37] for this computation, but other approaches might be used to improve the computational efficiency.

Table 2: Training Steps/s for different algorithms on BipedalWalkerHardcore-v3

Variant	Avg Training Steps/s (std)
SAC	174 (13)
SAC-D	142 (8)
SAC-D-CAGrad	26 (2)

D Training curves for robustness experiment

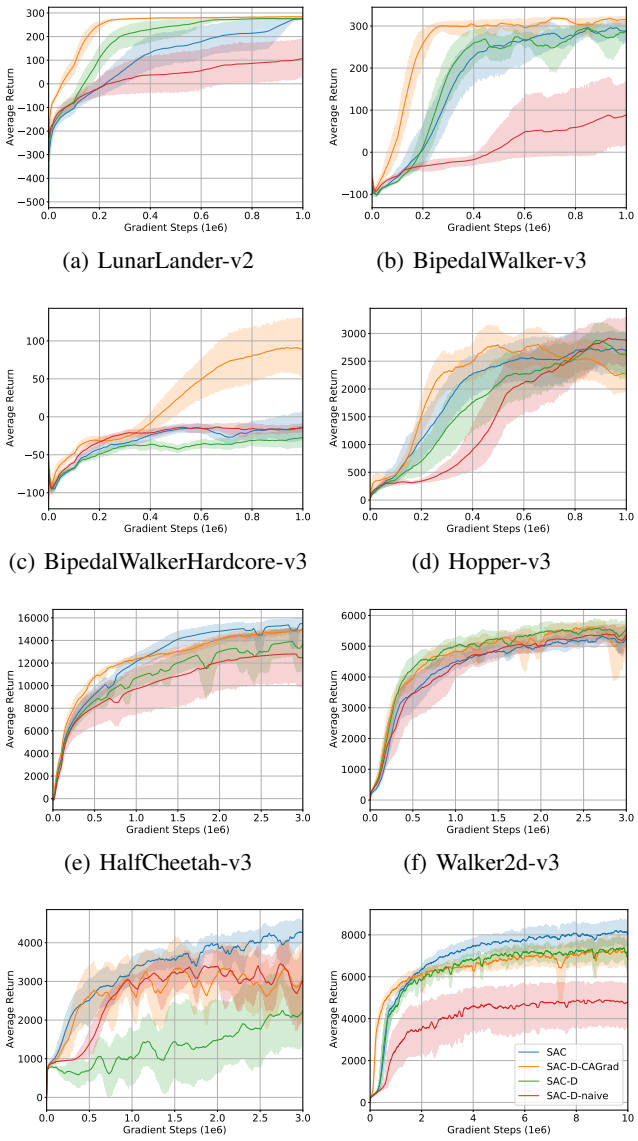
In Fig. 6 we show the training curves for all environments for SAC, SAC-D-Naive, SAC-D, and SAC-D-CAGrad. In most cases, SAC, SAC-D, and SAC-D-CAGrad resulted in final policies with comparable performance, with SAC-D-Naive sometimes under performing the others. Of these, the results on Ant-v3 and BipedalWalkerHardcore-v3 are the most atypical. On Ant-v3, SAC-D performed significantly worse than the others. See App. E for a larger investigation of this result. On BipedalWalkerHardcore-v3, we found that SAC-D-CAGrad significantly outperformed the other methods. Because we trained agents using an asynchronous training server, one possible reason for this result is that computational differences in the experiments resulted in different data distributions, rather than the improved performance being due to the differences in the algorithms. We trained SAC-D-CAGrad and SAC synchronously (with one gradient step per environment step) on BipedalWalkerHardcore-v3 to rule out this factor. While we found that asynchronous training tended to be more effective than synchronous training, SAC-D-CAGrad still outperformed SAC after 1 million gradient steps (see Fig. 7). These results suggest that value decomposition with CAGrad may in some cases benefit learning compared to baseline algorithms without value decomposition. We leave investigating this question further for future work.

E Performance Analysis of SAC-D on Ant-v3

In our results, SAC-D performed similarly or only slightly worse than SAC on most environments, with the exception of Ant-v3 in which it performed significantly worse than SAC. While using CAGrad significantly improved results of SAC-D on Ant-v3, we wanted to further analyze why results were so different on this environment.

From our investigation, we found that the abnormal behavior was due to poor component value predictions of rare environment terminations that led to large critic losses. Unlike other environments, in Ant, it is uncommon for even an untrained policy to terminate in failure. Consequently, terminating transitions were less well represented in the replay buffer for Ant than in other environments (Fig. 8(a).). Because terminating transitions constitute a large component value difference compared to non-terminating transitions (no more control cost, forward progress, nor health reward can be accrued), these rare instances in the replay buffer were associated with very large critic losses (and gradients) that destabilized learning. Fig. 8(c) shows that the average critic loss of SAC-D for Ant was much larger compared to other environments. It's possible SAC suffered less from this issue because the component rewards partially cancel out in the composite reward (e.g., negative control cost partially cancels positive forward progress when added together), resulting in a smaller loss.

To add further evidence that the difficulty was due to predicting the value of terminating transitions, we also trained SAC and SAC-D on a version on Ant that never terminates. In this case, we found that both SAC and SAC-D performed better than their respective terminating settings and that their



(a) LunarLander-v2 (b) BipedalWalker-v3

(c) BipedalWalkerHardcore-v3 (d) Hopper-v3

(e) HalfCheetah-v3 (f) Walker2d-v3

(g) Ant-v3 (h) Humanoid-v3

Figure 6: Training curves on continuous-control benchmarks. The solid lines represent the average episode returns over 10 trials. The shaded region represents the confidence interval. The lines are uniformly smoothed for visual clarity.

performance with each other was much more comparable (Fig. 8(b). We similarly found that the critic loss was much better behaved (Fig. 8(c)).

We believe the nature of CAGrad to re-weight component gradients to maximize the worst-component improvement stabilized results because if one component had an overly large error that conflicted with other components, CAGrad would decrease the weight of the gradient thereby keeping the entire process more stable. However, our results here suggest it might also be possible to use SAC-D without CAGrad when being careful about the definition of termination conditions, data representation, or other measures that make the value prediction task easier.

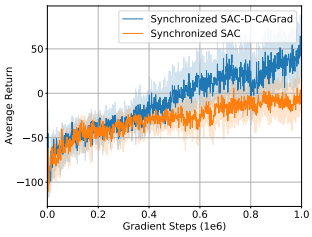
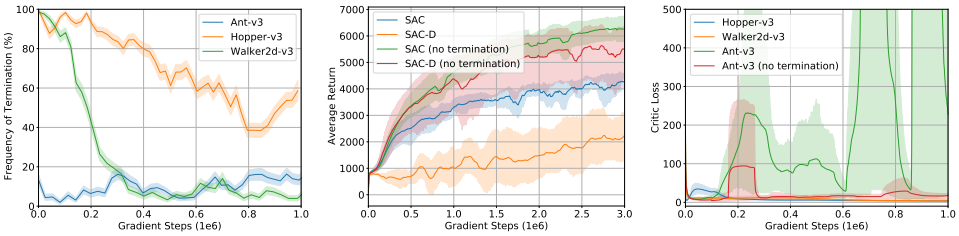


Figure 7: Average training curves of synchronized SAC and SAC-D-CAGrad on BipedalWalkerHardcore-v3 over ten trials. SAC-D-CAGrad still outperforms SAC in the synchronized setting after 1 million gradient steps, suggesting there may sometimes be an algorithmic benefit to using SAC with value decomposition.



(a) Frequency of termination (b) Performance without terminal (c) Critic loss

Figure 8: **(a)** Frequency of environmental terminations during training. The metrics data is collected by training SAC-D for 1M steps and averaged over 10 trials. The shaded region represents the confidence interval. The lines are uniformly smoothed for visual clarity. There are significantly fewer terminated episodes in Ant-v3 **(b)** Training curves on Ant-v3 with no termination conditions. The performance gap between SAC-D and SAC is significantly smaller without environment termination. **(c)** Critic loss of SAC-D training. The critic loss in Ant-v3 is enormously large. By removing the terminations, the critic loss is substantially suppressed.

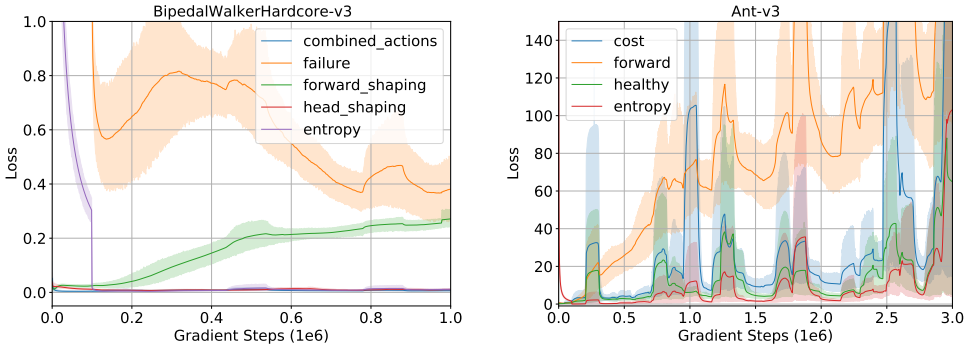
F Analysis of CAGrad behavior

In the context of SAC-D, every gradient step, CAGrad optimizes the weighting vector of component Q-function gradients:

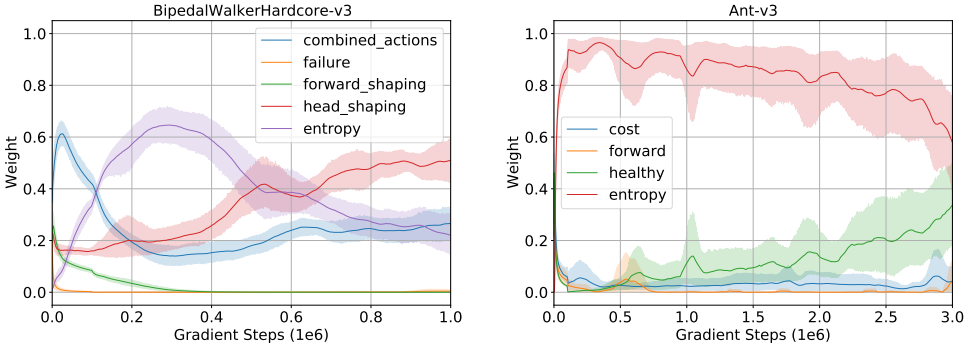
$$\min_w F(w) := g_w^\top g_0 + \sqrt{\phi}\|g_w\|, \text{where } g_w = \frac{1}{m+1}\sum_{i=1}^{m+1} w_i \nabla LQ_i(\theta) \tag{5}$$

where θ is a shared parameter, $g_0 = \frac{1}{m+1}\sum \nabla LQ_i(\theta)$, $\phi = c^2\|g_0\|^2$, c is a constant hyperparameter, and $\nabla LQ_i(\theta)$ is the gradient of the loss for the ith component Q-function with respect to Q-network parameters θ. The solution to this optimization problem, $w^* = \arg\min_w F(w)$, is used to derive the final gradient step direction: $\vec{\theta} \triangleq \sum_i^{m+1} w_i^* \nabla LQ_i(\theta)$. Equation 5 can be interpreted as a function that balances the gradient magnitudes by assigning the small w_i^* to the large gradients that conflict with other gradients.

The relationship of decomposed critic loss values and the result of the inner optimization is shown in Fig. 9. In both BipedalWalkerHardcore-v3 and Ant-v3, the component Q-function losses are inversely related to the gradient weight CAGrad associates with the component gradients: the higher the critic loss, the lower the weight CAGrad tended to assign the component. This result shows CAGrad prevents particularly large component errors from dominating the entire process, and helps explain why CAGrad effectively combated the loss of performance in Ant that was due to particularly large errors for rare terminating transitions. We expect CAGrad will be particularly useful in domains where different reward components induce returns with very different magnitudes.



(a) Decomposed critic loss

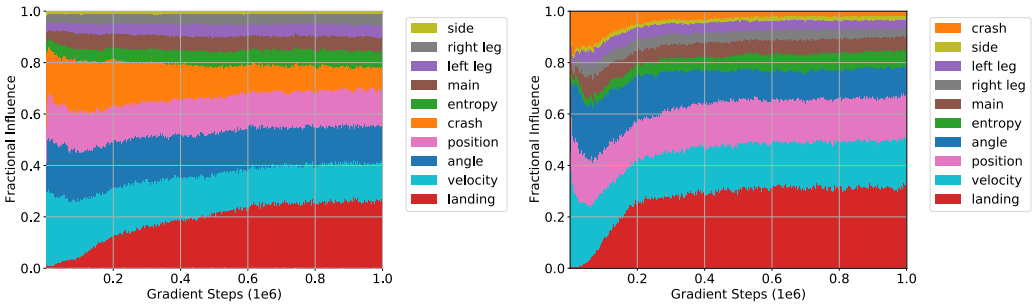


(b) Optimized gradient weights

Figure 9: **(a)** Decomposed critic loss during SAC-D-CAGrad training. The shaded regions represents the confidence interval. The left figure shows the decomposed critic loss of BipedalWalkerHardcore-v3, and the right figure shows the one of Ant-v3. **(b)** Weights of gradients optimized by CAGrad. Here we see that the component loss is inversely related to the weights CAGrad assigns to the component gradients.

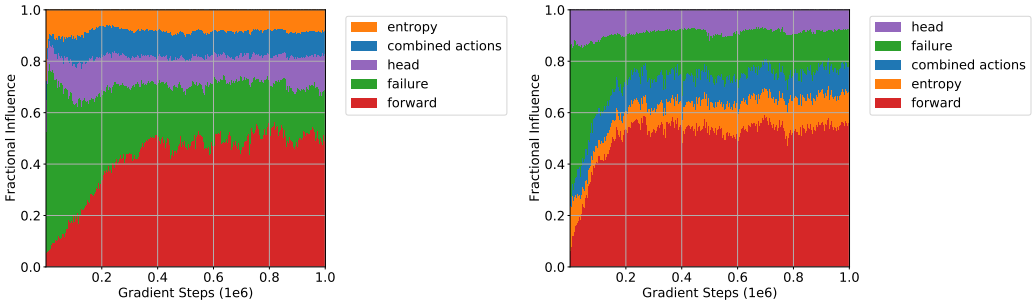
G Influence

Here we provide fractional influence plots for the various environments studied. For each plot the factors are sorted by the fractional influence at the very last gradient step with larger influence towards the bottom.



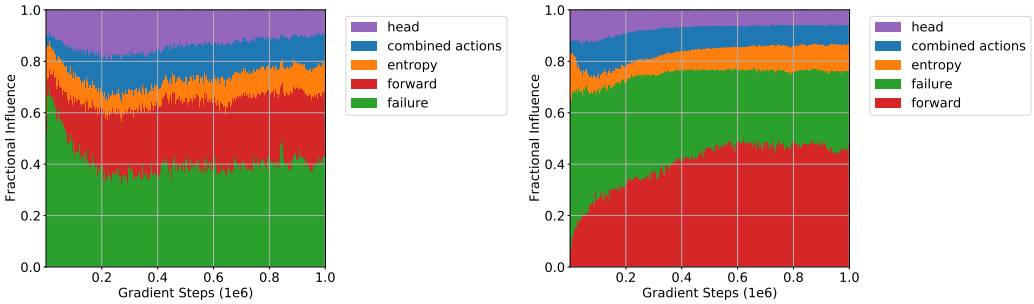
(a) SAC-D (b) SAC-D-CAGrad

Figure 10: Lunar Lander



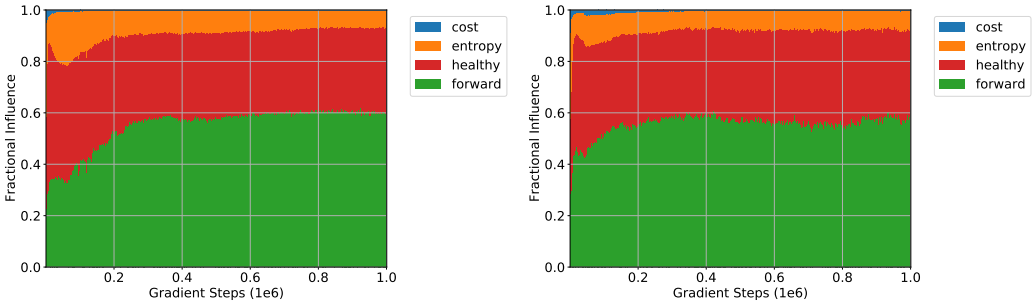
(a) SAC-D (b) SAC-D-CAGrad

Figure 11: Bipedal Walker



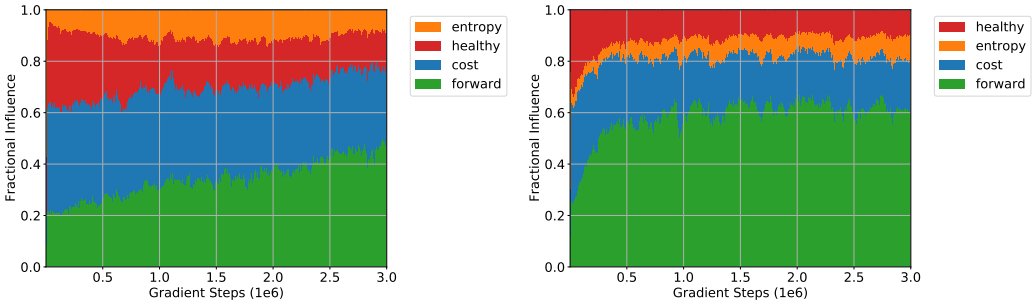
(a) SAC-D (b) SAC-D-CAGrad

Figure 12: Bipedal Walker Hardcore



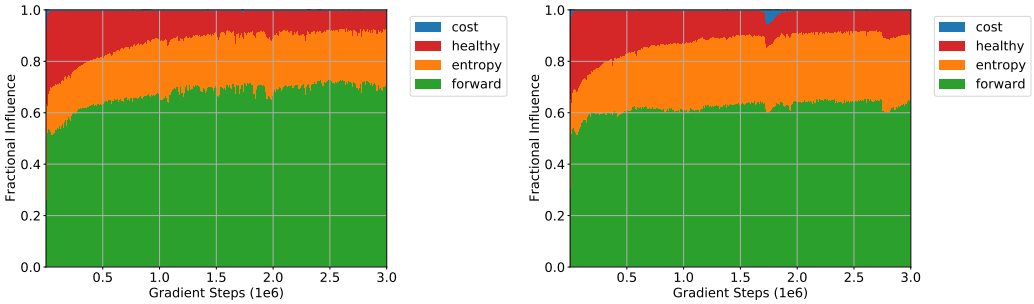
(a) SAC-D (b) SAC-D-CAGrad

Figure 13: Hopper



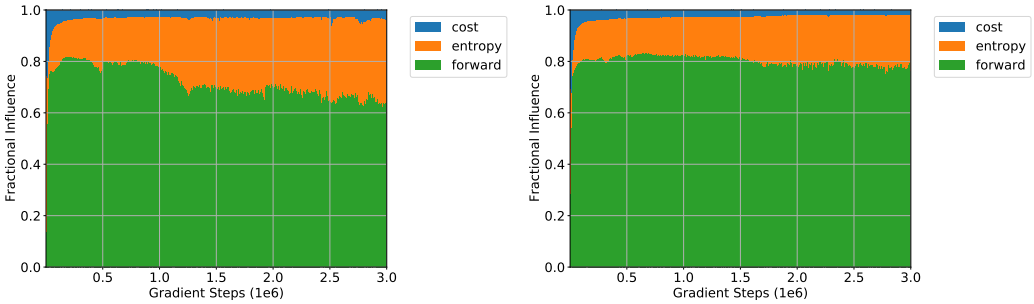
(a) SAC-D (b) SAC-D-CAGrad

Figure 14: Ant



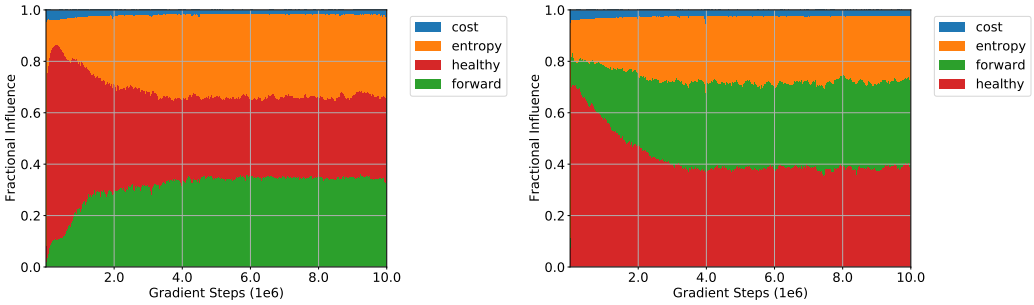
(a) SAC-D (b) SAC-D-CAGrad

Figure 15: Walker 2D



(a) SAC-D (b) SAC-D-CAGrad

Figure 16: Half Cheetah



(a) SAC-D

(b) SAC-D-CAGrad

Figure 17: Humanoid

H Markov features

In Sec. 6, we showed that the Lunar Lander observation features were insufficient to properly predict when landing would occur, because the simulator only signaled successful landing after the agent had been stable on the ground for a certain number of time steps. Here we provide the description of the additional *zero velocity trace feature* we added to the environment to make it Markov. We define the zero velocity trace feature as: $V_0^{\text{trace}}(t) = V_0^{\text{steps}}(t)/c$, where $V_0^{\text{steps}}(t)$ is the number of time steps since all the velocities dropped below a threshold and c is a fixed normalizing constant. In Fig. 4(a) we use a value $c = 40$. Velocity was considered to be zero when all velocity fields (horizontal, vertical and rotational) fell below 1e-3.

We compared the predictive accuracy with and without the $V_0^{\text{trace}}(t)$ feature. Using a policy trained with the baseline features we collected a dataset of 2000 trajectories (using the exploration policy). Next we trained two new value function models to predict the landing return (independent of all other factors), one for the baseline features and the other including the $V_0^{\text{trace}}(t)$ feature. For each time step, we computed the discounted Monte Carlo return ($\gamma = 0.99$). Note that we did not use bootstrapping in the return calculations. This could have caused inaccuracies in the targets as some trajectories timed out rather than terminating in either a crash or a landing. Each model was then trained for 100 epochs, with each epoch consisting of randomly subdividing time steps from the 2000 trajectories into batches of 256 time steps each and training on each subdivision. We used the mean squared loss and computed updates using the ADAM optimizer with a learning rate of 0.001. We then collected an additional 200 trajectories in which the agent successfully landed (again, using the exploration policy). For these trajectories we computed the value estimates and returns. We then computed the interquartile mean (IQM) values of correlation and RMSE for the last 25 time steps of each trajectory (indicated by the dashed pink line in Fig. 4(a)). From results in Table 3) we see a noticeable improvement in prediction accuracy. In particular, we see that the correlation is markedly improved, confirming that our predictions match the shape of the returns better in this time span.

Table 3: Landing Prediction Accuracy

Features	RMSE (std)	Correlation (std)
Baseline	5.61 (0.12)	0.521 (0.030)
Markov	**1.05 (0.28)**	**0.996 (0.001)**

I Constraint experiment details

The value function constraint experiment described in Sec. 6 and used to produce Fig. 4(b) followed the same training procedures as the robustness experiments described in App. C. However, the poor behavior, in which the unconstrained value estimates crossed zero, was most pronounced when we lowered the replay buffer size to 20k. The results shown Fig. 4(b) were trained under this condition.

Constraints can be applied in at least two ways. One is to constrain the component Q-value target by clipping. This prevents the update from pulling the predictions towards values that are invalid. Such a situation might result due to invalid bootstrapping estimates of the network. For example, the crash component in lunar lander can only be non-positive. Consequently, we constrain the target values as follows:

$$y_{crash} = \min\left(0, r_{crash} + \gamma Q_{crash}(s', a')\right).$$

In addition to constraining the bootstrap target, we can also penalized the component value function estimate for being positive with the following additional loss term:

$$L_{+;crash} = 0.5 \cdot \mathbb{1}_{\mathbb{R}^+}(Q_{crash}(s, a)) \cdot Q_{crash}(s, a).$$

Both these constraints have analogs for non-negative components. In the results shown in Fig. 4(b), we show only the effect on the *crash* component (which benefited the most), but we also applied constraints to the *side*, *main*, and *landing* components.

J Weight scheduling details

In Sec.6, we schedule the component weight of the Bipedal Walker Hardcore *failure* reward from zero to one to remedy its exploration inhibiting properties. Specifically, we used the schedule function:

$$w_{\text{failure}}(t) = \tanh\left(\beta \left|t - \tau^{\text{warmup}}\right|_{+}\right), \tag{6}$$

where t is the number of gradient steps taken; $|x|_{+} = \begin{cases} 0.01, & x < 0 \\ x, & x \geq 0 \end{cases}$; τ^{warmup} is the number of gradient steps required before the value begins increasing; and β controls the speed at which the weight returns to 1 after the warm-up period. We used $\tau^{\text{warmup}} = 100$ and $\beta = 0.0004$. With these values, the *failure* component weight came close to its original value of 1 after one million gradient steps, as shown in Fig. 18.

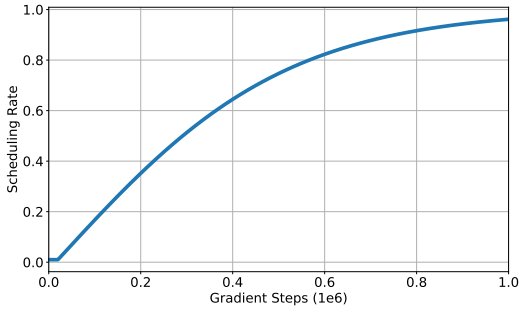
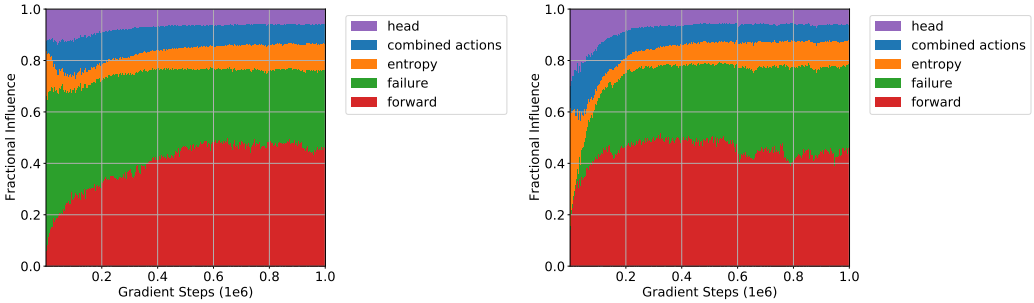


Figure 18: Scheduling rate function for reverse annealing

The results of these *failure* component weight scheduling experiments are shown in the score distribution plot of *forward* progress per episode score in Fig. 5(c) and compared to the baseline (constant weight scenario). Additionally, in Fig. 19 we see the effect that weight scheduling has on the influence of the *failure* and *forward* reward components. With *failure*'s initial influence reduced, *forward* is able to dominate the policy behavior from early on.



(a) Constant weights

(b) Reverse annealing

Figure 19: Figures show the impact of reverse annealing on learning with SAC-D-CAGrad in the Bipedal Walker Hardcore environment. (**a**) When a constant weighting of 1 is applied to all components *failure* dominates early learning. (**b**) Reverse annealing the weight on *failure* allows *forward* to drive policy behavior from early on.