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	$-\lvert A\rvert$
	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

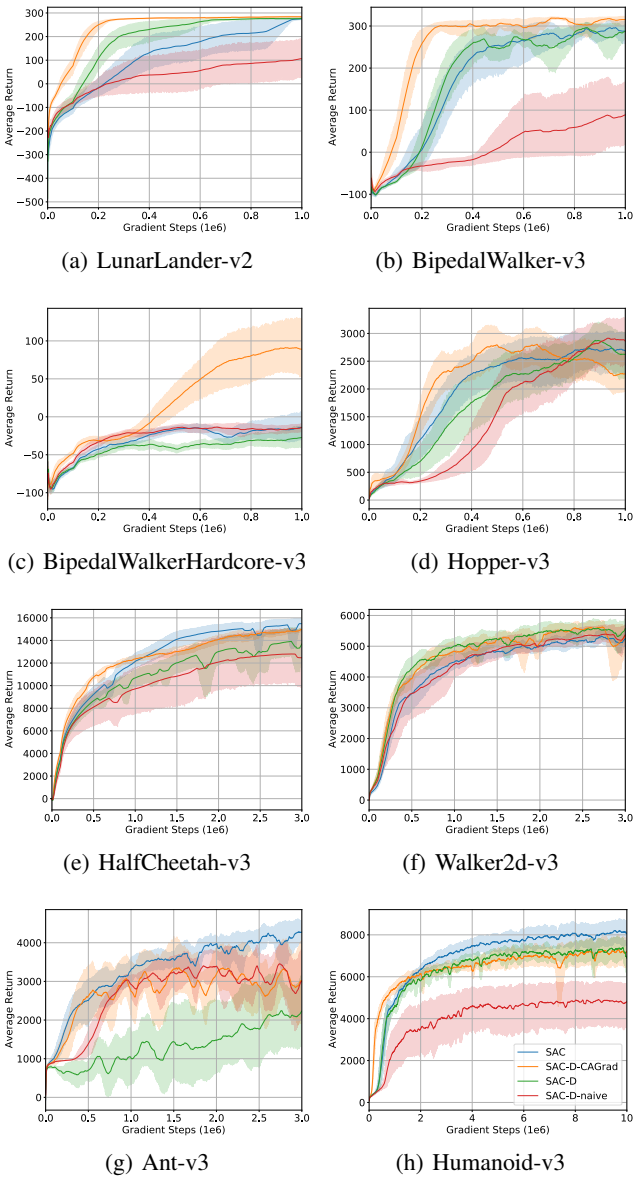
In Fig. 6 we show the training curves for all environments for SAC, SAC-D-Naive, SAC-D, and SAC-D-CAGrad. In most cases, SAC, SAC-D, and SAC-D-CAGrad resulted in final policies with comparable performance, with SAC-D-Naive sometimes under performing the others. Of these, the results on Ant-v3 and BipedalWalkerHardcore-v3 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.

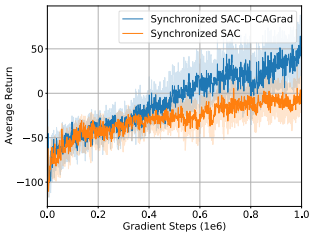
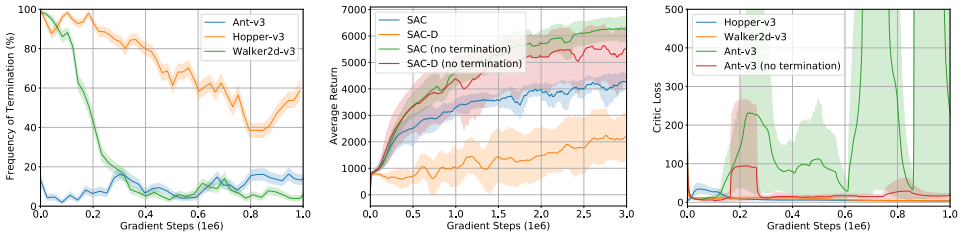


Figure 7: Average training curves of synchronized SAC and SAC-D-CAGrad on BipedalWalkerHardcore-v3 over ten trials. SAC-D-CAGrad still outperforms SAC in the synchronized setting after 1 million gradient steps, suggesting there may sometimes be an algorithmic benefit to using SAC with value decomposition.



(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.

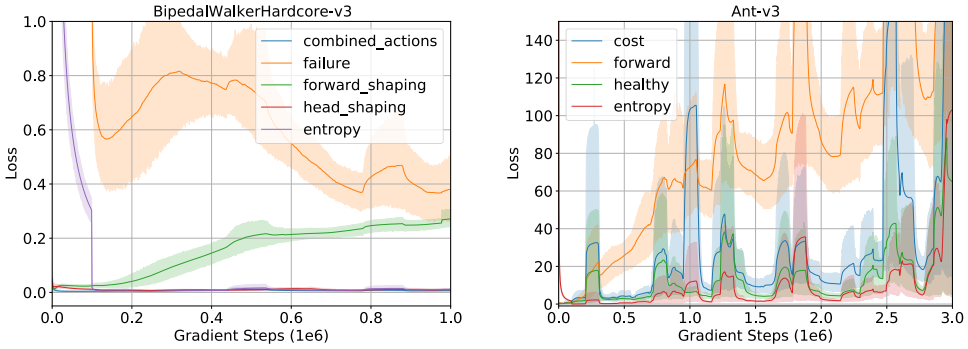
F Analysis of CAGrad behavior

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

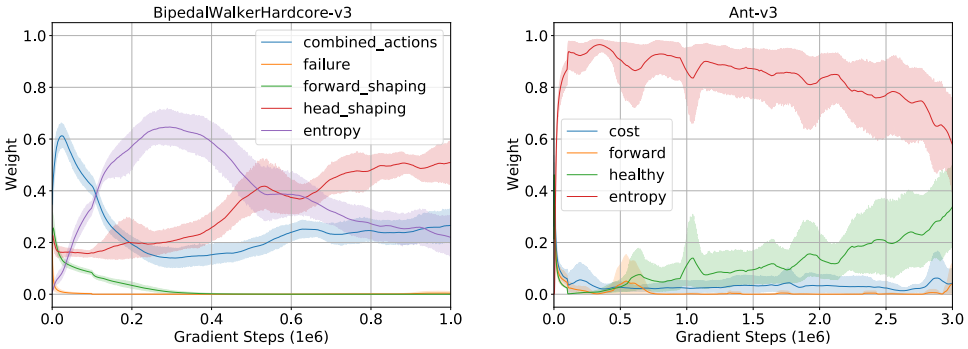
$$\min_w F(w) := g_w^\top g_0 + \sqrt{\phi}\|g_w\|, \text{where } g_w = \frac{1}{m+1}\sum_{i=1}^{m+1} w_i \nabla LQ_i(\theta) \tag{5}$$

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

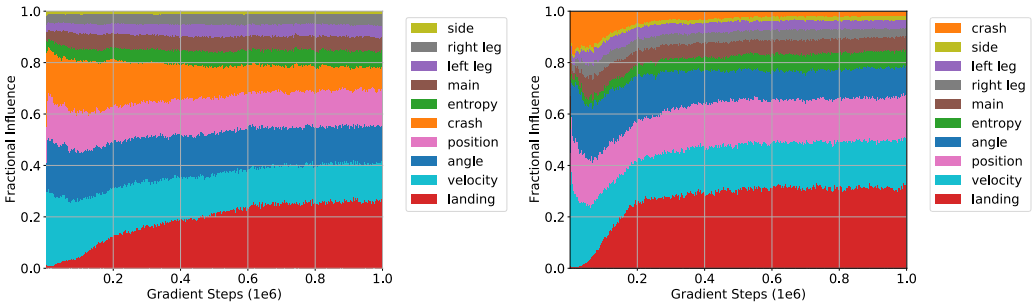


(b) Optimized gradient weights

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.

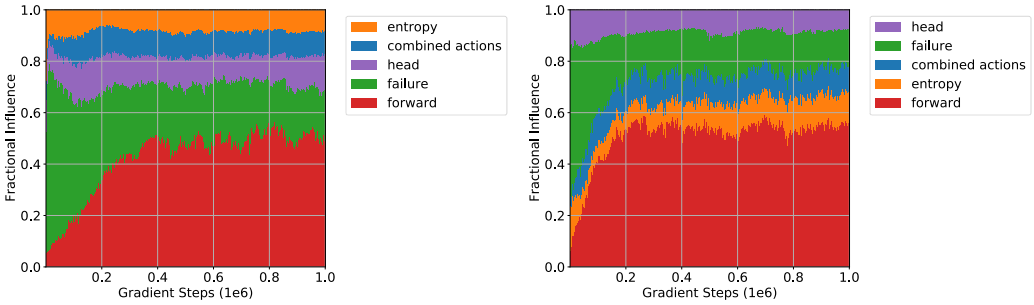
G Influence

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.



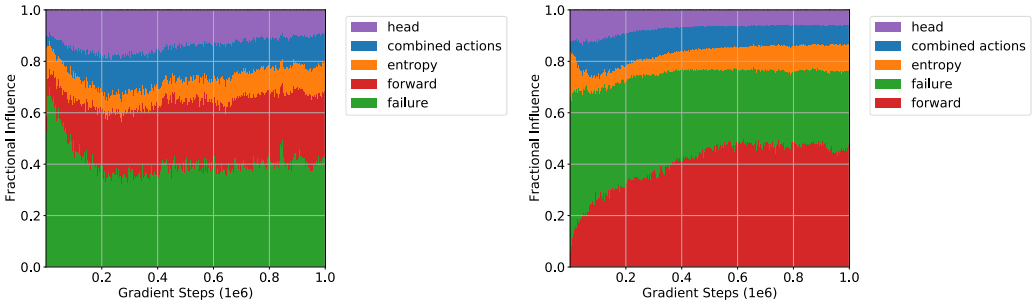
(a) SAC-D (b) SAC-D-CAGrad

Figure 10: Lunar Lander



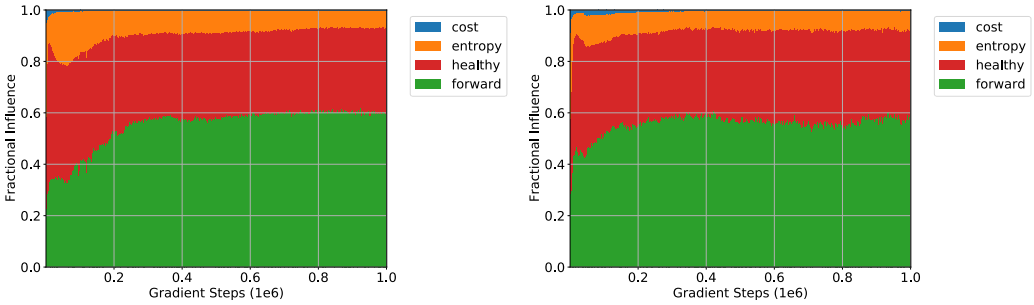
(a) SAC-D (b) SAC-D-CAGrad

Figure 11: Bipedal Walker



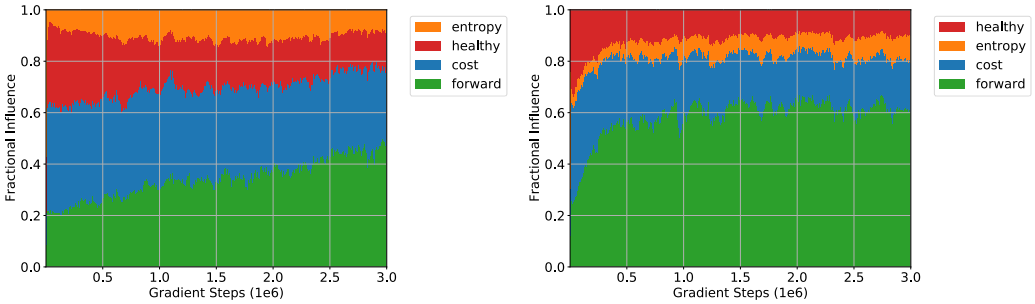
(a) SAC-D (b) SAC-D-CAGrad

Figure 12: Bipedal Walker Hardcore



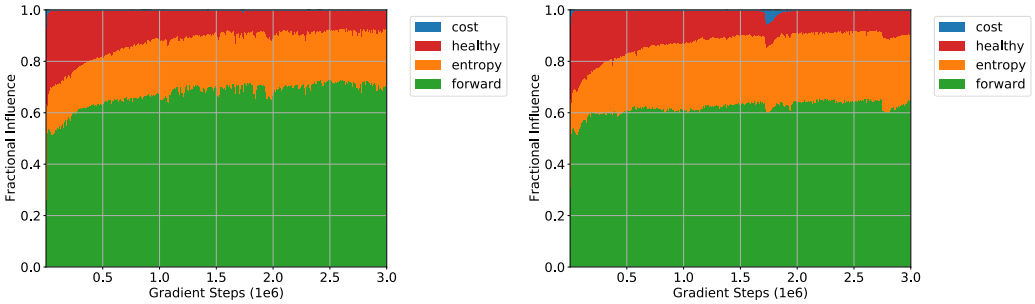
(a) SAC-D (b) SAC-D-CAGrad

Figure 13: Hopper



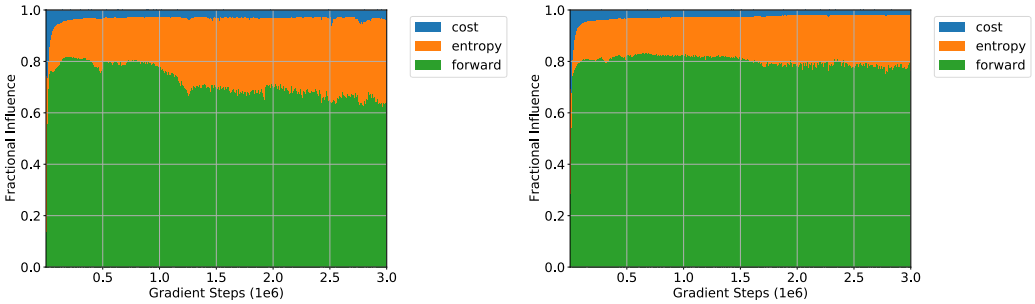
(a) SAC-D (b) SAC-D-CAGrad

Figure 14: Ant



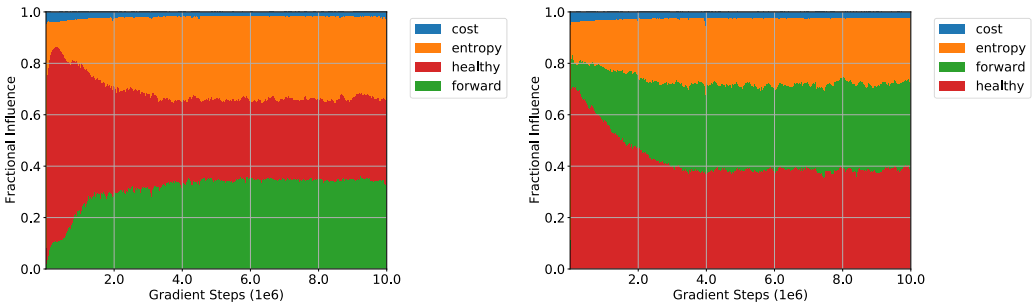
(a) SAC-D (b) SAC-D-CAGrad

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.

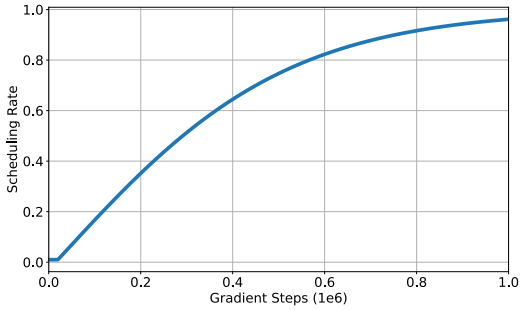
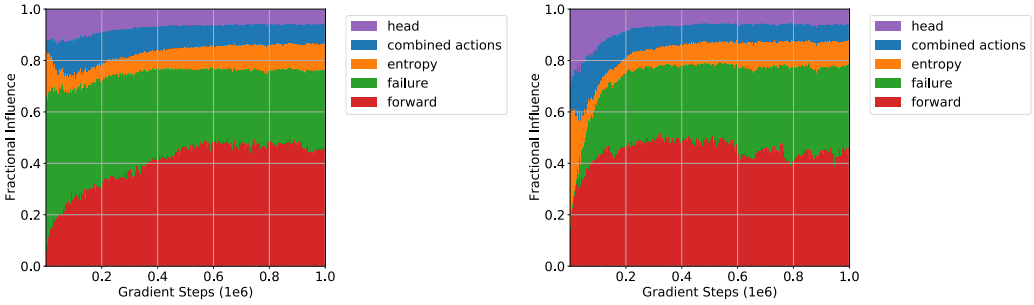


Figure 18: Scheduling rate function for reverse annealing

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.