A Experiment Details

A.1 2D Plots

In Figures 1 and 2, we plot the estimated variational objective, $\mathcal{J}$, as a function of two dimensions of the policy mean, $\boldsymbol{\mu}$. To create these plots, we first perform policy optimization (direct amortization in Figure 1 and iterative amortization in Figure 2), estimating the policy mean and variance. This is performed using on-policy trajectories from evaluation episodes (for a direct agent in Figure 1 and an iterative agent in Figure 2). While holding all other dimensions of the policy constant, we then estimate the variational objective while varying two dimensions of the mean (1 & 3 in Figure 1 and 2 & 6 in Figure 2). Iterative amortization is additionally performed while preventing any updates to the constant dimensions. Even in this restricted setting, iterative amortization is capable of optimizing the policy. Additional 2D plots comparing direct vs. iterative amortization on other environments are shown in Figure B.3, where we see similar trends.

A.2 Value Bias Estimation

We estimate the bias in the Q-value estimator using a similar procedure as Fujimoto et al. (2018), comparing the estimate of the Q-networks ($\widehat{Q}_\pi$) with a Monte Carlo estimate of the future objective in the actual environment, Q_π, using a set of state-action pairs. To enable comparison across setups, we collect 100 state-action pairs using a uniform random policy, then evaluate the estimator's bias, $\mathbb{E}_{\mathbf{s},\mathbf{a}}\left[\widehat{Q}_\pi - Q_\pi\right]$, throughout training. To obtain the Monte Carlo estimate of Q_π, we use 100 action samples, which are propagated through all future time steps. The result is discounted using the same discounting factor as used during training, $\gamma = 0.99$, as well as the same Lagrange multiplier, α. Figure 3 shows the mean and $\pm$ standard deviation across the 100 state-action pairs.

A.3 Amortization Gap Estimation

Calculating the amortization gap in the RL setting is challenging, as properly evaluating the variational objective, $\mathcal{J}$, involves unrolling the environment. During training, the objective is estimated using a set of Q-networks and/or a learned model. However, finding the optimal policy distribution, $\widehat{\pi}$, under these learned value estimates may not accurately reflect the amortization gap, as the value estimator likely contains positive bias (Figure 3). Because the value estimator is typically locally accurate near the current policy, we estimate the amortization gap by performing gradient ascent on $\mathcal{J}$ w.r.t. the policy distribution parameters, $\boldsymbol{\lambda}$, initializing from the amortized estimate (from π_ϕ). This is a form *semi-amortized* variational inference (Hjelm et al., 2016; Krishnan et al., 2018; Kim et al., 2018). We use the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 5×10^{-3} for 100 gradient steps, which we found consistently converged. This results in the estimated optimized $\widehat{\pi}$. We estimate the gap using 100 on-policy states, calculating $\mathcal{J}(\theta, \widehat{\pi}) - \mathcal{J}(\theta, \pi)$, i.e. the improvement in the objective after gradient-based optimization. Figure 6 shows the resulting mean and $\pm$ standard deviation. We also run iterative amortized policy optimization for an additional 5 iterations during this evaluation, empirically yielding an additional decrease in the estimated amortization gap.

A.4 Hyperparameters

Our setup follows that of soft actor-critic (SAC) (Haarnoja et al., 2018a;b), using a uniform action prior, i.e. entropy regularization, and two Q-networks (Fujimoto et al., 2018). Off-policy training is performed using a replay buffer (Lin, 1992; Mnih et al., 2013). Training hyperparameters are given in Table 6.

Temperature Following Haarnoja et al. (2018b), we adjust the temperature, α, to maintain a specified entropy constraint, $\epsilon_\alpha = |\mathcal{A}|$, where $|\mathcal{A}|$ is the size of the action space, i.e. the dimensionality.

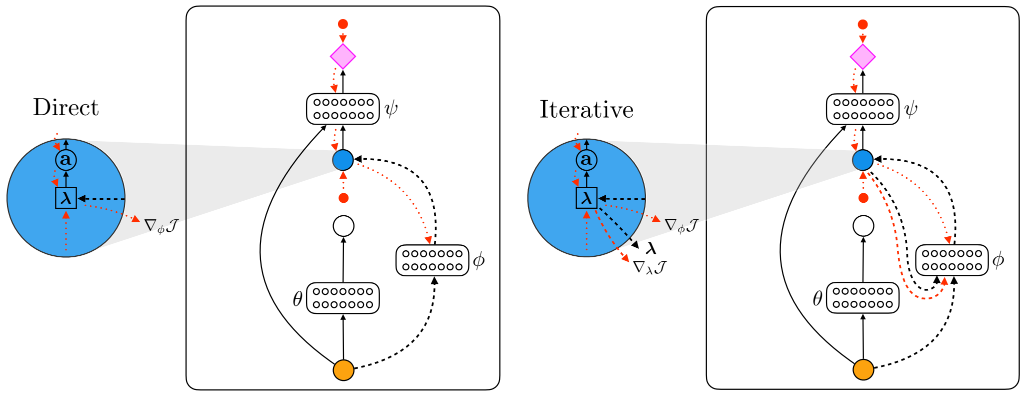
Policy We use the same network architecture (number of layers, units/layer, non-linearity) for both direct and iterative amortized policy optimizers (Table 2). Each policy network results in Gaussian distribution parameters, and we apply a `tanh` transform to ensure $\mathbf{a} \in [-1, 1]$ (Haarnoja et al., 2018a). In the case of a Gaussian, the distribution parameters are $\boldsymbol{\lambda} = [\boldsymbol{\mu}, \boldsymbol{\sigma}]$. The inputs and

Table 1: **Policy Inputs & Outputs**.

	Inputs	Outputs
Direct	$\mathbf{s}$	$\boldsymbol{\lambda}$
Iterative	$\mathbf{s}$, $\boldsymbol{\lambda}$, $\nabla_{\boldsymbol{\lambda}}\mathcal{J}$	$\boldsymbol{\delta}$, $\boldsymbol{\omega}$

Table 2: **Policy Networks**.

Hyperparameter	Value
Number of Layers	2
Number of Units / Layer	256
Non-linearity	`ReLU`


Direct
a
λ
$\nabla_\phi \mathcal{J}$
ψ
θ
ϕ
Iterative
a
λ
$\nabla_\phi \mathcal{J}$
$\boldsymbol{\lambda}$
$\nabla_{\boldsymbol{\lambda}} \mathcal{J}$
ψ
θ
ϕ

(a) Direct Amortization (b) Iterative Amortization

Figure A.1: **Amortized Optimizers**. Diagrams of **(a)** direct and **(b)** iterative amortized policy optimization. As in Figure 1, larger circles represent probability distributions, and smaller red circles represent terms in the objective. Red dotted arrows represent gradients. In addition to the state, $\mathbf{s}_t$, iterative amortization uses the current policy distribution estimate, $\boldsymbol{\lambda}$, and the policy optimization gradient, $\nabla_{\boldsymbol{\lambda}}\mathcal{J}$, to iteratively optimize $\mathcal{J}$. Like direct amortization, the optimizer network parameters, ϕ, are updated using $\nabla_\phi \mathcal{J}$. This generally requires some form of stochastic gradient estimation to differentiate through $\mathbf{a}_t \sim \pi(\mathbf{a}_t|\mathbf{s}_t, \mathcal{O}; \boldsymbol{\lambda})$.

outputs of each optimizer form are given in Table 1. Again, $\boldsymbol{\delta}$ and $\boldsymbol{\omega}$ are respectively the update and gate of the iterative amortized optimizer (Eq. 11), each of which are defined for both $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. Following Marino et al. (2018b), we apply layer normalization (Ba et al., 2016) individually to each of the inputs to iterative amortized optimizers. We initialize iterative amortization with $\boldsymbol{\mu} = \mathbf{0}$ and $\boldsymbol{\sigma} = \mathbf{1}$, however, these could be initialized from a learned action prior (Marino et al., 2018a).

Table 3: Q**-value Network Architecture A**.

Hyperparameter	Value
Number of Layers	2
Number of Units / Layer	256
Non-linearity	`ReLU`
Layer Normalization	False
Connectivity	Sequential

Table 4: Q**-value Network Architecture B**.

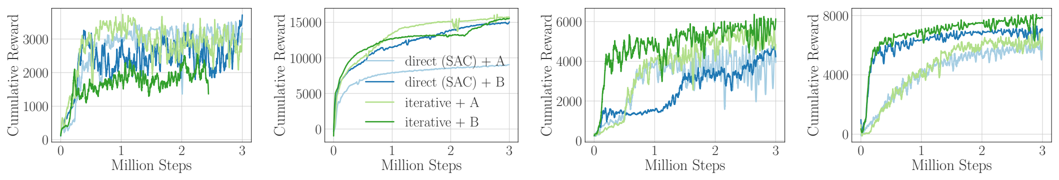
Hyperparameter	Value
Number of Layers	3
Number of Units / Layer	512
Non-linearity	`ELU`
Layer Normalization	True
Connectivity	Highway

Q-value We investigated two Q-value network architectures. Architecture A (Table 3) is the same as that from Haarnoja et al. (2018a). Architecture B (Table 4) is a wider, deeper network with highway connectivity (Srivastava et al., 2015), layer normalization (Ba et al., 2016), and ELU non-linearities (Clevert et al., 2015). We initially compared each Q-value network architecture using each policy optimizer on each environment, as shown in Figure A.2. The results in Figure 5 were obtained using the better performing architecture in each case, given in Table 5. As in Fujimoto et al. (2018), we use an ensemble of 2 separate Q-networks in each experiment.

Value Pessimism (β) As discussed in Section 3.2.2, the increased flexibility of iterative amortization allows it to potentially exploit inaccurate value estimates. We increased the pessimism hyperpa-

Table 5: Q**-value Network Architecture by Environment**.

	`Hopper-v2`	`HalfCheetah-v2`	`Walker2d-v2`	`Ant-v2`
Direct	A	B	A	B
Iterative	A	A	B	B



(a) `Hopper-v2` (b) `HalfCheetah-v2` (c) `Walker2d-v2` (d) `Ant-v2`

Figure A.2: **Value Architecture Comparison**. Plots show performance for 1 seed for each value architecture (A or B) for each policy optimization technique (direct or iterative). Note: results for iterative + B on `Hopper-v2` were obtained with an overly pessimistic value estimate ($\beta = 2.5$ rather than $\beta = 1.5$) and are consequently worse.

rameter, β, to further penalize variance in the value estimate. Experiments with direct amortization use the default $\beta = 1$ in all environments, as we did not find that increasing β helped in this setup. For iterative amortization, we use $\beta = 1.5$ on `Hopper-v2` and $\beta = 2.5$ on all other environments. This is only applied during training; while collecting data in the environment, we use $\beta = 1$ to not overly penalize exploration.

Table 6: **Training Hyperparameters**.

Hyperparameter	Value
Discount Factor (γ)	0.99
Q-network Update Rate (τ)	$5 \cdot 10^{-3}$
Network Optimizer	Adam
Learning Rate	$3 \cdot 10^{-4}$
Batch Size	256
Initial Random Steps	$5 \cdot 10^{3}$
Replay Buffer Size	10^{6}

A.5 Model-Based Value Estimation

For model-based experiments, we use a single, deterministic model together with the ensemble of 2 Q-value networks (discussed above).

Model We use separate networks to estimate the state transition dynamics, $p_{\text{env}}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$, and reward function, $r(\mathbf{s}_t, \mathbf{a}_t)$. The network architecture is given in Table 7. Each network outputs the mean of a Gaussian distribution; the standard deviation is a separate, learnable parameter. The reward network directly outputs the mean estimate, whereas the state transition network outputs a residual estimate, $\Delta_{\mathbf{s}_t}$, yielding an updated mean estimate through:

$$\boldsymbol{\mu}_{\mathbf{s}_{t+1}} = \mathbf{s}_t + \Delta_{\mathbf{s}_t}.$$

Model Training The state transition and reward networks are both trained using maximum log-likelihood training, using data examples from the replay buffer. Training is performed at the same frequency as policy and Q-network training, using the same batch size (256) and network optimizer. However, we perform 10^3 updates at the beginning of training, using the initial random steps, in order to start with a reasonable model estimate.

Table 7: **Model Network Architectures**.

Hyperparameter	Value
Number of Layers	2
Number of Units / Layer	256
Non-linearity	`Leaky ReLU`
Layer Normalization	True

Table 8: **Model-Based Hyperparameters**.

Hyperparameter	Value
Rollout Horizon, h	2
Retrace λ	0.9
Pre-train Model Updates	10^3
Model-Based Value Targets	True

Value Estimation To estimate Q-values, we combine short model rollouts with the model-free estimates from the Q-networks. Specifically, we unroll the model and policy, obtaining state, reward, and policy estimates at current and future time steps. We then apply the Q-value networks to these future state-action estimates. Future rewards and value estimates are combined using the Retrace estimator (Munos et al., 2016). Denoting the estimate from the Q-network as $\widehat{Q}_\psi(\mathbf{s}, \mathbf{a})$ and the reward estimate as $\widehat{r}(\mathbf{s}, \mathbf{a})$, we calculate the Q-value estimate at the current time step as

$$\widehat{Q}_\pi(\mathbf{s}_t, \mathbf{a}_t) = \widehat{Q}_\psi(\mathbf{s}_t, \mathbf{a}_t) + \mathbb{E}\left[\sum_{t'=t}^{t+h} \gamma^{t'-t}\lambda^{t'-t}\left(\widehat{r}(\mathbf{s}_{t'}, \mathbf{a}_{t'}) + \gamma \widehat{V}_\psi(\mathbf{s}_{t'+1}) - \widehat{Q}_\psi(\mathbf{s}_{t'}, \mathbf{a}_{t'})\right)\right], \quad (1)$$

where λ is an exponential weighting factor, h is the rollout horizon, and the expectation is evaluated under the model and policy. In the variational RL setting, the state-value, $V_\pi(\mathbf{s})$, is

$$V_\pi(\mathbf{s}) = \mathbb{E}_\pi\left[Q_\pi(\mathbf{s}, \mathbf{a}) - \alpha \log \frac{\pi(\mathbf{a}|\mathbf{s}, \mathcal{O})}{p_\theta(\mathbf{a}|\mathbf{s})}\right]. \quad (2)$$

In Eq. 1, we approximate V_π using the Q-network to approximate Q_π in Eq. 2, yielding $\widehat{V}_\psi(\mathbf{s})$. Finally, to ensure consistency between the model and the Q-value networks, we use the model-based estimate from Eq. 1 to provide target values for the Q-networks, as in Janner et al. (2019).

Future Policy Estimates Evaluating the expectation in Eq. 1 requires estimates of π at future time steps. This is straightforward with direct amortization, which employs a feedforward policy, however, with iterative amortization, this entails recursively applying an iterative optimization procedure. Alternatively, we could use the prior, $p_\theta(\mathbf{a}|\mathbf{s})$, at future time steps, but this does not apply in the max-entropy setting, where the prior is uniform. For computational efficiency, we instead learn a separate direct (amortized) policy for model-based rollouts. That is, with iterative amortization, we create a separate direct network using the same hyperparameters from Table 2. This network distills iterative amortization into a direct amortized optimizer, through the KL divergence, $D_{\text{KL}}(\pi_{\text{it.}}||\pi_{\text{dir.}})$. Rollout policy networks are common in model-based RL (Silver et al., 2016; Piché et al., 2019).

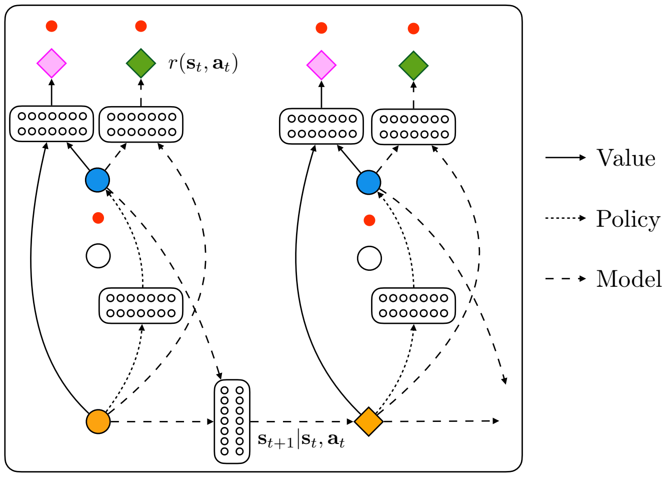
B Additional Results

B.1 Improvement per Step

In Figure B.1, we plot the average improvement in the variational objective per step throughout training, with each curve showing a different random seed. That is, each plot shows the average change in the variational objective after running 5 iterations of iterative amortized policy optimization. With the exception of `HalfCheetah-v2`, the improvement remains relatively constant throughout training and consistent across seeds.

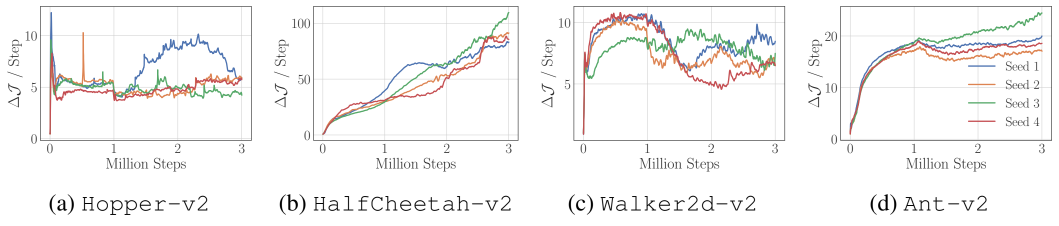
B.2 Comparison with Iterative Optimizers

Iterative amortized policy optimization obtains the *accuracy* benefits of iterative optimization while retaining the *efficiency* benefits of amortization. In Section 4, we compared the accuracy of iterative and direct amortization, seeing that iterative amortization yields reduced amortization gaps (Figure 6) and improved performance (Figure 5). In this section, we compare iterative amortization with two popular iterative optimizers: Adam (Kingma & Ba, 2014), a gradient-based optimizer, and cross-entropy method (CEM) (Rubinstein & Kroese, 2013), a gradient-free optimizer.



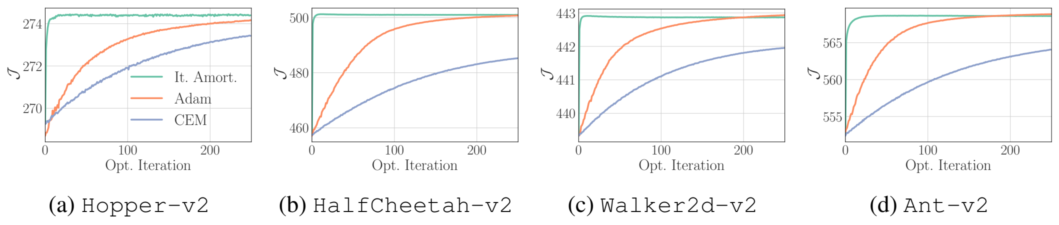
$r(\mathbf{s}_t, \mathbf{a}_t)$
$\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t$
Value
Policy
Model

Figure A.3: **Model-Based Value Estimation**. Diagram of model-based value estimation (shown with direct amortization). For clarity, the diagram is shown without the policy prior network, $p_\theta(\mathbf{a}_t|\mathbf{s}_t)$. The model consists of a deterministic reward estimate, $r(\mathbf{s}_t, \mathbf{a}_t)$, (green diamond) and a state estimate, $\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t$, (orange diamond). The model is unrolled over a horizon, H, and the Q-value is estimated using the Retrace estimator (Munos et al., 2016), given in Eq. 1.



(a) `Hopper-v2` (b) `HalfCheetah-v2` (c) `Walker2d-v2` (d) `Ant-v2`

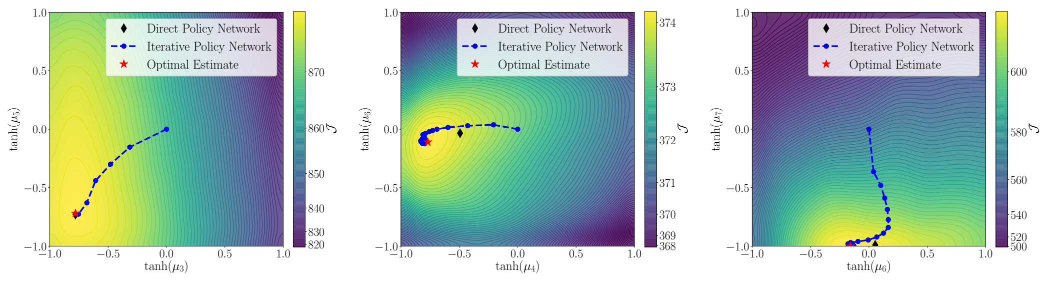
Figure B.1: **Per-Step Improvement**. Each plot shows the per-step improvement in the estimated variational RL objective, $\mathcal{J}$, throughout training resulting from iterative amortized policy optimization. Each curve denotes a different random seed.



(a) `Hopper-v2` (b) `HalfCheetah-v2` (c) `Walker2d-v2` (d) `Ant-v2`

Figure B.2: **Comparison with Iterative Optimizers**. Average estimated objective over policy optimization iterations, comparing with Adam (Kingma & Ba, 2014) and CEM (Rubinstein & Kroese, 2013). These iterative optimizers require over an order of magnitude more iterations to reach comparable performance with iterative amortization, making them impractical in many applications.

To compare the accuracy and efficiency of the optimizers, we collect 100 states for each seed in each environment from the model-free experiments in Section 4.2.2. For each optimizer, we optimize the variational objective, $\mathcal{J}$, starting from the same initialization. Tuning the step size, we found that 0.01 yielded the steepest improvement without diverging for both Adam and CEM. Gradients are evaluated with 10 action samples. For CEM, we sample 100 actions and fit a Gaussian mean and variance to the top 10 samples. This is comparable with QT-Opt (Kalashnikov et al., 2018), which draws 64 samples and retains the top 6 samples.



(a) `HalfCheetah-v2` (b) `Walker2d-v2` (c) `Ant-v2`

Figure B.3: **2D Optimization Plots**. Each plot shows the optimization objective over two dimensions of the policy mean, $\boldsymbol{\mu}$. This optimization surface contains the value function trained using a direct amortized policy. The black diamond, denoting the estimate of this direct policy, is generally near-optimal, but does not match the optimal estimate (red star). Iterative amortized optimizers are capable of generalizing to these surfaces in each case, reaching optimal policy estimates.

The results, averaged across states and random seeds, are shown in Figure B.2. CEM (gradient-free) is less efficient than Adam (gradient-based), which is unsurprising, especially considering that Adam effectively approximates higher-order curvature through momentum terms. However, Adam and CEM both require over *an order of magnitude* more iterations to reach comparable performance with iterative amortization. While iterative amortized policy optimization does not always obtain asymptotically optimal estimates, we note that these networks were trained with only 5 iterations, yet continue to improve and remain stable far beyond this limit. Finally, comparing wall clock time for each optimizer, iterative amortization is only roughly $1.25\times$ slower than CEM and $1.15\times$ slower than Adam, making iterative amortization still substantially more efficient.

B.3 Additional 2D Optimization Plots

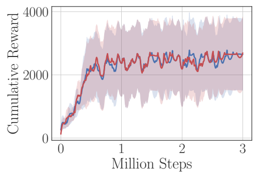
In Figure 1, we provided an example of the suboptimal optimization resulting from direct amortization on the `Hopper-v2` environment. We also demonstrated that iterative amortization is capable of automatically generalizing to this optimization surface, outperforming the direct optimizer. To show that this is a general phenomenon, in Figure B.3, we present examples of corresponding 2D plots for each of the other environments considered in this paper. As before, we see that direct amortization is near-optimal, but, with the exception of `HalfCheetah-v2`, does not match the optimal estimate. In contrast, iterative amortization is able to find the optimal estimate, again, generalizing the unseen optimization surfaces.

B.4 Additional Optimization & the Amortization Gap

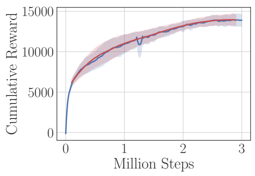
In Section 4, we compared the performance of direct and iterative amortization, as well as their estimated amortization gaps. In this section, we provide additional results analyzing the relationship between policy optimization and the performance in the actual environment. As we have emphasized previously (see Section 3.2.2), this relationship is complex, as optimizing an inaccurate Q-value estimate does not improve task performance.

The amortization gap quantifies the suboptimality in the objective, $\mathcal{J}$, of the policy estimate. As described in Section A.3, we estimate the optimized policy by performing additional gradient-based optimization on the policy distribution parameters (mean and variance). However, when we deploy this optimized policy for evaluation in the actual environment, as shown for direct amortization in Figure B.4, we do not observe a noticeable difference in performance. Thus, while amortization may find suboptimal policy estimates, we observe that the actual difference in the objective is either too small or inaccurate to affect performance at test time.

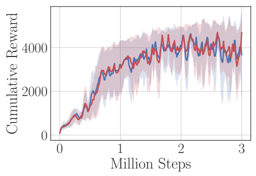
Likewise, in Section 4.2.2, we observed that using additional amortized iterations during evaluation further decreased the amortization gap for iterative amortization. However, when we deploy this



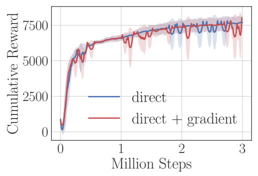
(a) `Hopper-v2`



(b) `HalfCheetah-v2`

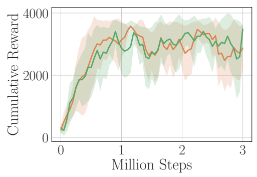


(c) `Walker2d-v2`

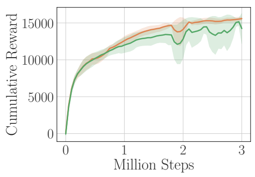


(d) `Ant-v2`

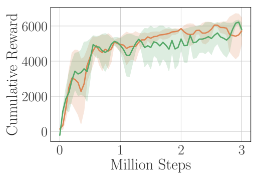
Figure B.4: **Test-Time Gradient-Based Optimization**. Each plot compares the performance of direct amortization vs. direct amortization with 50 additional gradient-based policy optimization iterations. Note that this additional optimization is only performed at test time.



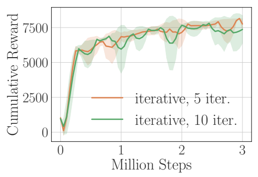
(a) `Hopper-v2`



(b) `HalfCheetah-v2`

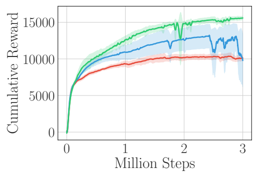


(c) `Walker2d-v2`

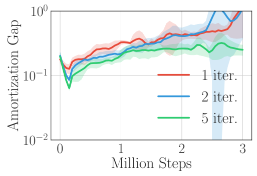


(d) `Ant-v2`

Figure B.5: **Additional Amortized Test-Time Iterations**. Each plot compares the performance of iterative amortization (trained with 5 iterations) vs. the same agent with an additional 5 iterations at evaluation. Performance remains similar or slightly worse.



(a)



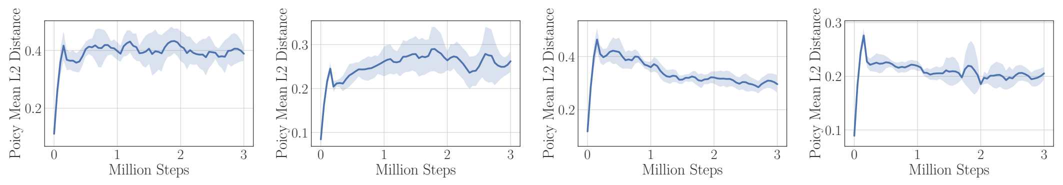
(b)

Figure B.6: **Iterations During Training**. **(a)** Performance and **(b)** estimated amortization gap for varying numbers of policy optimization iterations per step during training on `HalfCheetah-v2`. Increasing the iterations improves performance and decreases the estimated amortization gap.

more fully optimized policy in the environment, as shown in Figure B.5, we do not generally observe a corresponding performance improvement. In fact, on `HalfCheetah-v2` and `Walker2d-v2`, we observe a slight *decrease* in performance. This further highlights the fact that additional policy optimization may exploit inaccurate Q-value estimates.

However, importantly, in Figures B.4 and B.5, the additional policy optimization is only performed for evaluation. That is, the data collected with the more fully optimized policy is not used for training and therefore cannot be used to correct the inaccurate value estimates. Thus, while more accurate policy optimization, as quantified by the amortization gap, may not substantially affect evaluation performance, it does play a significant role in improving training.

This was shown in Section 4.2.4, where we observed that training with additional iterative amortized policy optimization iterations, i.e., a more flexible policy optimizer, generally results in improved performance. By using a more accurate (or exploitative) policy for data collection, the agent is able to better evaluate its Q-value estimates, which accrues over the course of training. This trend is shown for `HalfCheetah-v2` in Figure B.6, where we observed the largest difference in performance across numbers of iterations. We generally observe that increasing the number of iterations during training improves performance and decreases the amortization gap. Interestingly, when performance dips for the agents trained with 2 iterations, there is a corresponding increase in the amortization gap.



(a) Hopper-v2 (b) HalfCheetah-v2 (c) Walker2d-v2 (d) Ant-v2

Figure B.7: **Distance Between Policy Means**. Each plot shows the L2 distance between the estimated policy means from two separate policy optimization runs at a given state. Results are averaged over 100 on-policy states at each point in training and over experiment seeds.

B.5 Multiple Policy Estimates

As discussed in Section 3.2.1, iterative amortization has the added benefit of potentially obtaining multiple policy distribution estimates, due to stochasticity in the optimization procedure (as well as initialization). In contrast, unless latent variables or normalizing flows are incorporated into the policy, direct amortization is limited to a single policy estimate. To estimate the degree to which iterative amortization obtains multiple policy estimates during training, we perform two separate runs of policy optimization per state and evaluate the L2 distance between the means of these policy estimates (after applying the `tanh`). Note that in MuJoCo action spaces, which are bounded to $[-1, 1]$, the maximum distance is $2\sqrt{|\mathcal{A}|}$, where $|\mathcal{A}|$ is the size of the action space. We average the policy mean distance over 100 states and all experiment seeds, with the results shown in Figure B.7. In all environments, we see that the average distance initially increases during training, remaining relatively constant for `Hopper-v2` and `HalfCheetah-v2` and decreasing slightly for `Walker2d-v2` and `Ant-v2`. Note that the distance for direct amortization would be exactly 0 throughout. This indicates that iterative amortization does indeed obtain multiple policy estimates, maintaining some portion of multi-estimate policies throughout training.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pp. 1587–1596, 2018.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pp. 1856–1865, 2018a.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018b.

Devon Hjelm, Ruslan R Salakhutdinov, Kyunghyun Cho, Nebojsa Jojic, Vince Calhoun, and Junyoung Chung. Iterative refinement of the approximate posterior for directed belief networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 4691–4699, 2016.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems*, pp. 12519–12530, 2019.

Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*, 2018.

Yoon Kim, Sam Wiseman, Andrew C Miller, David Sontag, and Alexander M Rush. Semi-amortized variational autoencoders. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

Durk P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Rahul G Krishnan, Dawen Liang, and Matthew Hoffman. On the challenges of learning with inference networks on sparse, high-dimensional data. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 143–151, 2018.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

Joseph Marino, Milan Cvitkovic, and Yisong Yue. A general method for amortizing variational filtering. In *Advances in Neural Information Processing Systems*, 2018a.

Joseph Marino, Yisong Yue, and Stephan Mandt. Iterative amortized inference. In *International Conference on Machine Learning*, pp. 3403–3412, 2018b.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, 2013.

Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 1054–1062, 2016.

Alexandre Piché, Valentin Thomas, Cyril Ibrahim, Yoshua Bengio, and Chris Pal. Probabilistic planning with sequential monte carlo methods. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=ByetGn0cYX`.

Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems (NIPS)*, pp. 2377–2385, 2015.