Q-functionals for Value-Based Continuous Control

Samuel Lobel[*1], **Sreehari Rammohan**[*1], **Bowen He**[*1], **Shangqun Yu**[2], **George Konidaris**[1],

[1] Brown University
[2] University of Massachusetts, Amherst
bowen_he@brown.edu, samuel_lobel@brown.edu, sreehari_rammohan@brown.edu,
shangqunyu@umass.edu, gdk@cs.brown.edu

Abstract

We present Q-functionals, an alternative architecture for continuous control deep reinforcement learning. Instead of returning a single value for a state-action pair, our network transforms a state into a function that can be rapidly evaluated in parallel for many actions, allowing us to efficiently choose high-value actions through sampling. This contrasts with the typical architecture of off-policy continuous control, where a policy network is trained for the sole purpose of selecting actions from the Q-function. We represent our action-dependent Q-function as a weighted sum of basis functions (Fourier, Polynomial, etc) over the action space, where the weights are state-dependent and output by the Q-functional network. Fast sampling makes practical a variety of techniques that require Monte-Carlo integration over Q-functions, and enables action-selection strategies besides simple value-maximization. We characterize our framework, describe various implementations of Q-functionals, and demonstrate strong performance on a suite of continuous control tasks.

Introduction

The ultimate product of a successful reinforcement learning system is a *policy* that performs well in interacting with the environment, as measured by expected cumulative discounted reward. A dominant paradigm in reinforcement learning is to derive policies from a *Q-function* (Watkins and Dayan 1992) that estimates the expected cumulative reward for taking a given action. When the number of actions available is finite, a strong policy can be derived by enumerating all action-values for a given state and choosing the one with the highest value. However, enumeration is impossible when there are large or infinite possible actions, for example when actions are drawn from a continuous vector space. This is the natural description of, for example, robotic locomotion and control; significant effort has therefore gone into alternative approaches for action-selection in these domains (Gu et al. 2016; Asadi et al. 2021; Lillicrap et al. 2015). A common framework for so-called *continuous control* problems is to train a separate *policy network* that selects actions according to some criteria of the Q-values (Fujimoto, van Hoof, and Meger 2018; Haarnoja et al. 2018a).

The gradient update for a policy network comes completely from the Q-function, and as such it essentially summarizes information about the Q-function for a given state so as to efficiently produce high-value actions. For example, standard practice is to train a policy network to estimate the *maximum*-valued action for any given state(Lillicrap et al. 2015). The quantities policy networks estimate are difficult to calculate on a per-state basis, so policy networks can be thought of as *amortizing* the expensive computation of finding desirable actions: the training procedure is expensive, but it allows actions to be identified with a single pass of a neural network. There are two potential pitfalls of this framework. First, since the policy network is not reinitialized after every Q-function update, at any given timestep most of the policy training has been done on *old* versions of the Q-function. The policy network is therefore maximizing a *stale* estimate of action-value. Second, policies are generally either deterministic or sampled from a tight distribution around the single highest valuable action. Therefore, it is unlikely for the policy to sample two high-value actions if they are too far apart from each other (Tessler, Tennenholtz, and Mannor 2019). This may limit exploration as well as the robustness of the value function. Perhaps most importantly, we show later that even a well-trained policy network is less effective than previously thought at its assigned task of maximizing Q-values.

We take a different perspective on this problem: instead of training a network to amortize expensive action-value computations, we propose a network architecture that ensures these evaluations are cheap to do in parallel. This opens up a new avenue for action-selection: instead of using a policy network, we can simply evaluate many actions in parallel and choose from them one with high value. Since the policy is directly derived from Q-values at each timestep it always reflects the most current action-value estimates. In addition, random sampling can easily generate high-value actions from throughout the action-space.

We introduce a class of Q-functions we call *Q-functionals* that allow for efficient sampling. A *functional* is a function that returns another function; as such, a Q-functional transforms a state into a function over only the action space, such that Q-values for many actions can be evaluated in parallel and with little overhead. Specifically, we compute Q-values as the dot product between state-dependent coefficients and

[*]These authors contributed equally.

an expressive basis representation of an action; thus, evaluating multiple actions for a state reduces to a single matrix multiplication. We describe a variety of implementations of this architecture, investigate its speed and effectiveness at sampling and action-maximization, and demonstrate competitive performance on a suite of continuous control reinforcement learning tasks.

Background

This work examines sequential decision making problems represented as Markov Decision Processes (MDPs) denoted by $\langle \mathcal{S}, \mathcal{A}, T, R, P_0, \gamma \rangle$, where $\mathcal{S}$, $\mathcal{A}$ and γ are the state space, action space, and the discount factor, respectively (Sutton and Barto 2018). The transition and reward functions are given by $T(s, a)$ and $R(s, a)$ respectively, and $P_0(s)$ is the initial state distribution. We seek to learn an action-selection strategy, or *policy* $\pi : \mathcal{S} \to \mathcal{A}$, resulting in high cumulative discounted reward.

For a given policy π, the state-action value function, or Q-function, is defined as:

$$Q^\pi(s_t, a_t) = \mathbb{E}_\pi[R(s_t, a_t) + \gamma V^\pi(s_{t+1})],$$

where $V^\pi(s)$ is the expected value of following policy π from state s:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s, a)].$$

We also investigate τ-Entropy-Regularized RL (ER-RL), in which case the value of a state is regularized by the entropy of the current policy (Schulman, Chen, and Abbeel 2017; Haarnoja et al. 2018b):

$$V^\pi_{\text{ENT}}(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s, a) - \tau \log(\pi(a|s))],$$

where τ controls the degree of regularization.

When we are interested in maximizing cumulative reward, we can derive a policy from a Q-function by choosing the maximum-valued action at every state:

$$\pi^Q(s) = \arg\max_a Q(s, a).$$

A common method for iteravely improving a Q function parameterized by θ is through bootstrapping (Sutton 1988):

$$\begin{aligned} \theta &\leftarrow \theta + \alpha\delta\Delta_\theta\hat{Q}(s, a; \theta) \\ \text{where} \quad \delta &= r + \gamma V^\pi(s') - \hat{Q}(s, a; \theta), \end{aligned} \tag{1}$$

where α is the step-size and (s, a, r, s') tuples are drawn from experience. The fixed point of this update equation is Q^*, the Q-function that describes the optimal policy π^*:

$$Q^*(s_t, a_t) = \mathbb{E}_{\pi^*}[R(s_t, a_t) + \gamma V^\pi(s_{t+1})].$$

Central to this process is deriving a policy from a Q-function through maximization, or related techniques. When $\mathcal{A}$ is a set of discrete actions, this maximization can be done easily by comparing the Q-value for each action (Mnih et al. 2015). Alternatively, this work focuses on the problem of continuous control, where actions are drawn from a continuous vector space.

Related Work

Continuous-control is concerned with domains that have *continuous* action spaces; thus, instead of choosing from a finite set of actions, an agent must output a *vector* of continuous entries at every timestep. This difference necessitates characteristically different agent architectures from discrete-action agents such as DQN.

Policy-gradient (PG) methods are a dominant paradigm of continuous control due to their expressive policies and end-to-end training scheme (Sutton et al. 1999). Modern PG methods represent policies as deep neural networks, that are trained to output high-valued actions using a gradient signal produced by the Q-function. Deterministic PG methods output a single action, while stochastic PG methods generally parameterize a simple distribution (such as Gaussian) that can be sampled during action-selection (Haarnoja et al. 2018a). The output of a "perfectly" trained policy network would maximize the value (as calculated by the Q-function) at any given state. The chief observation motivating our work is that throughout training, this is far from the case.

Standard PG methods require a single evaluation of the policy network to produce an action, and a single evaluation of the Q-network to compute the value of that action. A fruitful line of research involves incorporating sampling methods, which compute the value of multiple actions, for either improved action-selection or better value-estimation. Prior work utilizes the cross-entropy method (CEM), a zero-order sample-based optimization method, to choose actions that have higher value than the original action output by the policy network (Simmons-Edler et al. 2019). The Expected Policy Gradient (EPG) framework reduces variance in the value computation through Monte Carlo integration over the policy's outputs (Ciosek and Whiteson 2018). Evaluating actions from a distribution around the policy's output has been found to reduce overfitting of a policy network to a Q-function (Fujimoto, van Hoof, and Meger 2018). All of these methods, however, are limited to evaluating small numbers of action-values in practice, because every computed value requires an additional evaluation of the Q-function's neural network.

Our method does away with the policy network altogether, putting it in the class of *value-function only continuous control*. We list other examples from this class for completeness. "Continuous Action Q-Learning" (Millán, Posenato, and Dedieu 2002) uses mixed-integer programming to solve for maximal valued actions (though in practice runtime considerations stop this at approximately maximal) in piecewise-linear value functions (such as neural networks with rectified linear activation functions). "Normalized Advantage Functions" analytically select maximum-valued actions by restricting their action-value function to quadratic polynomials (Gu et al. 2016). Perhaps the most similar method to ours is RBF-DQN (Asadi et al. 2021), which computes value as the weighted sum over learned centroids, and describes a method to select approximately maximum-action values. The primary difference in our approach compared with past value-function only methods is the focus on sampling, which allows for a variety of action-selection strategies besides simple maximization, as well as use for

variance-reducing Monte Carlo techniques.

Q-Functionals

Our method is based on a simple premise: many of continuous control's subproblems described above (action selection/maximization, value-estimation, variance reduction) can be improved if we can rapidly compute the value of many actions.

Q-functionals are a way of expressing Q-functions so that many action-values can be evaluated in parallel for a given state. Consider a traditional deep Q-function, represented by a neural network:

$$Q(s, a) : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathcal{R}.$$

A standard design for such a Q-function in continuous control is to concatenate states and actions, and then pass the concatenated vectors through a neural network that outputs their estimated values. For this standard architecture, evaluating two action-values for the same state takes twice as many operations as evaluating one. A Q-functional breaks the computation of action-values into two parts: first, a state is transformed into parameters that define a function over the action space. Then, this function is evaluated for the action(s) in question:

$$Q_{\text{FUNC}}(s, a) : \mathcal{S} \rightarrow (\mathcal{A} \rightarrow \mathcal{R}). \tag{2}$$

We design these functions such that while the per-state computation may be expensive, the per-action computation is relatively cheap. One simple way to represent these functions is by learning state-dependent coefficients of basis functions over the action space. As a brief example, a one-dimensional polynomial basis function represents a scalar action a by the vector $[1, a, a^2, \ldots]$. An appropriate choice of basis function also adds an *inductive bias* to learning action-values: though we may expect values to have complex dependence on state (especially when the state is high-dimensional such as in image representations), in general we expect values to be relatively smooth with respect to the action dimensions. In Figure 1, we visualize the difference between the two architectures as differing places where the actions enter the computation.

Q-Functional Implementations

We describe three implementations of Q-functionals using different basis functions. First, a Q-functional calculated using the Fourier basis is shown below in equation 3. The

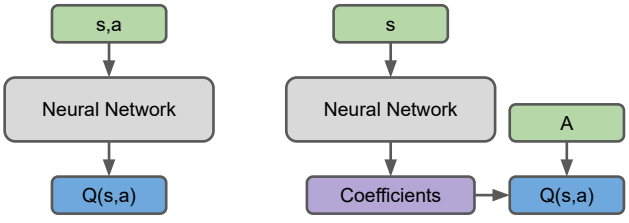

Figure 1: Network architecture of traditional Q-function (left) and Q-functional (right)

Fourier basis representation of *states* had great success when used for Q-learning in small state spaces (Konidaris, Osentoski, and Thomas 2011); we rely on the same inductive bias over the moderately-sized *action* spaces typical of continuous control. L represents the difference between the max and min action space value allowed, $x_i(s), y_i(s)$ are the state-dependent coefficients output by the Q-functional network. $\mathbf{C}$ is a matrix of frequencies for the Fourier basis. For a rank R Q-functional using the Fourier basis and operating in an environment with action dimension d, the full set of frequencies is represented by the Cartesian product $\{0, 1, 2, ..., R\}^d$. In practice however, we find that limiting the set of frequencies to those where the sum of all elements in the frequency is less than or equal to the rank provides sufficient generalization. Formally, $\mathbf{C} = \left\{C_i = \{0, 1, 2, ..., R\}^d | \sum_{j=0}^{d} C_{ij} \leq R\right\}$

$$Q_{\text{FUNC}}(s, a) = \sum_{C_i \in \mathbf{C}} x_i(s) \sin(\frac{\pi}{L} a \cdot C_i) + y_i(s) \cos(\frac{\pi}{L} a \cdot C_i). \tag{3}$$

Crucially, the learned coefficients $x_i(s)$ and $y_i(s)$ are computed by the Q-functional neural network using only the state as input. Each action-evaluation for that state then reuses these same coefficients. Thus, Q-functionals can compute many action-values for the same state with only a single neural network evaluation. As demonstrated later, this allows for significantly faster sampling of Q-values.

We can similarly define a second basis over polynomials: we represent our action-value function such that the total polynomial-order is less than some R:

$$Q_{\text{FUNC}}(s, a) = \sum_{C_i \in \mathbf{C}} x_i(s) \prod_{j=0}^{d} a_j^{C_{ij}}. \tag{4}$$

A third implementation of Q-functionals is the Legendre basis. To encourage feature-independence, we can orthogonalize the polynomial basis over $\mathcal{A}$ by extending the Legendre polynomials to multiple dimensions (see the Appendix for details on the Legendre basis). Let $\mathcal{L}_i(x)$ be the i^{th}-order Legendre polynomial. Then, each frequency vector defines an orthogonal polynomial:

$$\hat{\mathcal{L}}_{C_i}(a) = \prod_{j=0}^{d} \mathcal{L}_{C_{ij}}(a_j),$$

and our Q-function can be calculated from learned coefficients:

$$Q_{\text{FUNC}}(s, a) = \sum_{C_i \in \mathbf{C}} x_i(s) \hat{\mathcal{L}}_{C_i}(a). \tag{5}$$

Previous work demonstrates that action-values can be analytically maximized for quadratic polynomials over actions ($R = 2$) (Gu et al. 2016). Without a policy network, however, Q-functions defined as such are restricted to convex functions over actions which can be overly limiting. Our sampling framework can derive a strong policy from more complex, higher-order polynomial Q-functions. In addition,

Algorithm 1 Q-functional action-evaluation / selection

Set rank, BASIS, numRandom
Initialize model(rank, BASIS, θ)

Function `EvaluateManyActions(s, A)`**:**
  coefficients = model.getCoefficients(s)
  representations = BASIS.getRepresentation(A, rank)
  actionValues = matmul(A, coefficients)

Function `getBestActionAndValue(s, A)`**:**
  A = drawRandomActions(1, numRandom)
  actionValues = EvaluateManyActions(s, A)
  return A[argmax(actionValues)], max(actionValues)

as described in the next section, sampling allows us to calculate state-values with various methods that work better than simple maximization.

We note a simple interpretation of each method listed above: the Q-function is computed as the *dot product* between an action-representation and a vector of learned coefficients. For example, the Q-value of rank-1 polynomial functional is simply an affine linear function over the action space. Higher-order functions allow trading off more expressive function classes with evaluation speed.

Extracting Policies and State-Values from Q-Functionals

In standard continuous-control RL, the policy network is used for two distinct purposes: calculating state-value functions (bootstrapping) and action-selection (policy-evaluation). Here we describe how a variety of strategies for both can be implemented using sampling.

The most common strategy for both is *action maximization* (Lillicrap et al. 2015): aiming to find the highest-value action for every state, and using this for both bootstrapping and action-selection:

$$\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a) \quad ; \quad V(s) = Q(s, \pi(s)).$$

Instead of training a policy network, we can directly estimate the maximum through sampling k actions:

$$\pi(s) = \arg\max_{a_i \in \mathcal{A}^k} Q(s,a_i) \quad ; \quad V(s) = \max_{a_i \in \mathcal{A}^k} Q(s,a_i). \tag{6}$$

This sampling scheme is directly analogous to how actions are selected in discrete-action domains. These functions are detailed in Algorithm 1, and can be used as drop-in replacements for backpropogation-trained policy networks in the learning process.

For ER-RL, the optimal policy is proportional to the Boltzmann distribution over Q-values, as this optimally trades off between policy entropy and expected return (Schulman, Chen, and Abbeel 2017). Thus, given a trained Q-function, the optimal policy and associated value are (Haarnoja et al. 2017):

$$\pi(s) = \Pr(a|s) \leftarrow \frac{\exp\{Q(s,a)/\tau\}}{\int_{\mathcal{A}} \exp\{Q(s,a)/\tau\}}$$
$$V(s) = \int_{\mathcal{A}} \pi(s) Q(s,a) - \log(\Pr(a|s))$$
$$= \tau \log\Big(\int_{\mathcal{A}} \exp\{Q(s,a)/\tau\}\Big).$$

Due to limitations imposed by common representations of stochastic policies (often constrained to state-dependent normal distributions over actions), this relation between policy and value cannot be achieved by standard policy-parameterizations (Haarnoja et al. 2017). By contrast, we can consistently estimate these qualities through Monte Carlo sampling:

$$\pi(s) = \Pr(a|s) = \frac{\exp\{Q(s,a)/\tau\}}{\frac{1}{k}\sum_{a_i \in \mathcal{A}^k} \exp\{Q(s,a)/\tau\}}$$
$$V(s) = \tau \log\Big(\frac{1}{k} \sum_{a_i \in \mathcal{A}^k} \exp\{Q(s,a)/\tau\}\Big). \tag{7}$$

In our experiments, we derive a robust policy $\pi(s)$ and value function $V(s)$ through a "top-n of k" approach: we evaluate k random actions, and represent our policy and value by sampling from the best n actions during interaction with the environment and averaging these n values during bootstrapping updates for stability.

$$\pi(s) \sim \text{top-n}\Big\{Q(s,a_1), ...Q(s,a_k)\Big\}$$
$$V(s) = \text{mean}\Big[\text{top-n}\Big\{Q(s,a_1), ...Q(s,a_k)\Big\}\Big]. \tag{8}$$

This is a similar strategy to "target policy smoothing" (Fujimoto, van Hoof, and Meger 2018), which computes $V(s)$

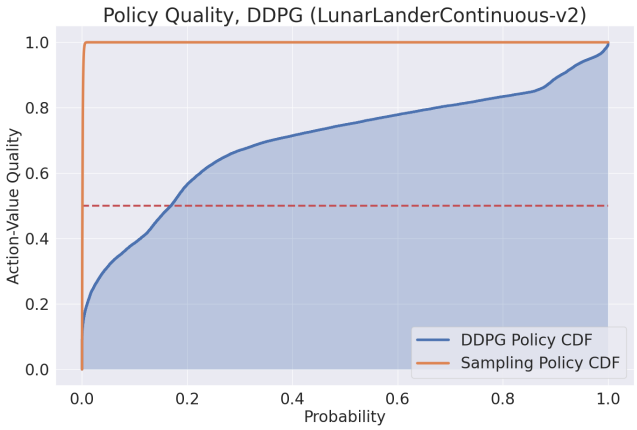

Figure 2: Quality of actions selected by trained policy network, and random sampling, on the LunarLanderContinuous task.

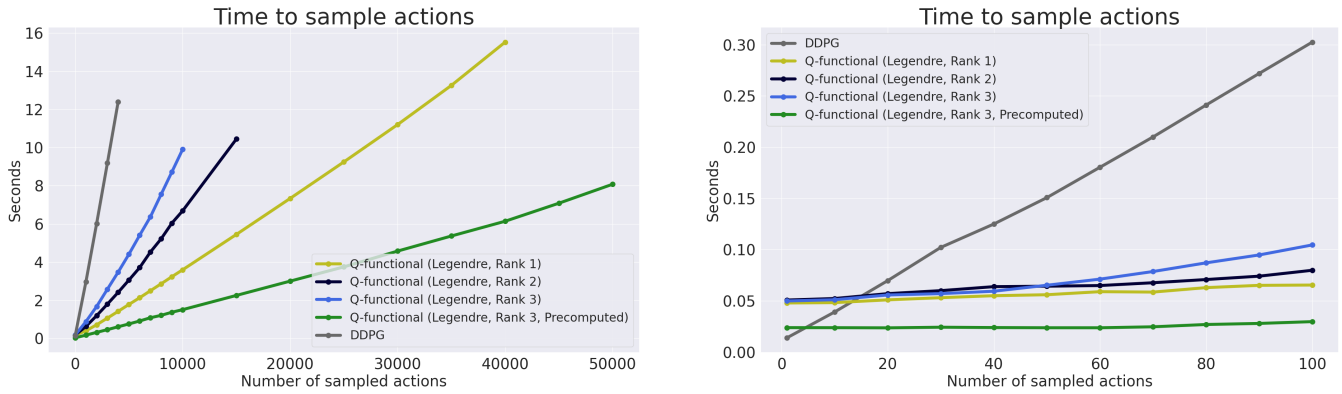

Figure 3: Speed comparison of 100 sets of action-evaluations of the same states (batch size 1024), for varying number of action-samples. Left: samples up to 10,000. Right: zoomed in sampling up to 100.

as a single-sample mean of the Q-values surrounding the policy-networks output. We find that averaging over *many* values, instead of simply choosing one, leads to decreased noisiness of $V(s)$ for the Bellman target in the update, and superior performance at maximizing reward.

Comparisons

The core claim of this paper is that by structuring our Q-functions as functionals we gain access to superior action-selection strategies that standard architectures cannot efficiently implement: that policy networks do not effectively maximize the Q-function they are trained on, and that Q-functionals allow for significantly larger sampling with the same computational budget. All experiments are done on tasks from the OpenAI Gym continuous control suite (Brockman et al. 2016). Reproducing code can be found at the linked repository[1].

Evaluating Policy Networks

We begin by investigating the implicit assumption that policies trained through gradient descent are effective at maximizing the output of the Q-function they are trained on. We can quantify this concept by comparing the Q-value of the policy network's output with the distribution of Q-values of randomly sampled actions. A well-trained policy network should consistently output actions with higher value than the large majority of random actions. We train three instances of a DDPG agent on the "LunarLanderContinuous-v2" task, and for each one collect a dataset of 10,000 states from on-policy rollouts. For each state, we compute the Q-value of the policy's output using the agent's Q-function. We then compare this to the value of 10,000 randomly sampled actions, and determine the fraction of these for which the policy's output has lower value (Figure 2). The horizontal red line indicates the median action's value at each state. This shows that in 15% of states, on average the policy network selects an action with lower value than one chosen at random. Furthermore, the policy's output is only in the top decile of action-values in 5% of states. This enormous gap between a policy network's implied function and its empirical behavior suggests that alternative action-selection strategies can prove useful.

By contrast, the orange line represents the Q-functional policy from Equation 6, that selects the highest-valued action from a pool of 1,000 random samples. The output of this policy is better than a single random action 99.9% of the time. We include similar figures for a variety of environments in the Appendix.

Sampling Q-functions Versus Q-functionals

As demonstrated in the previous section, the standard Q-function architecture can naïvely evaluate many actions by simply passing the same state to its input many times. However, Figure 3 demonstrates the speedup of using the Q-functional architecture for multiple evaluations. For a single batch of 1024 states, we evaluate an increasing number of actions on the Hopper task (action dimension of 3) for 100 iterations. We find that a rank 3 Legendre Q-functional evaluates actions roughly 3.5 times faster on a single Nvidia 2080-ti GPU than a neural network that takes in both states and action as inputs. In addition, the lightweight action-evaluations of Q-functionals allows for larger sampling size: the standard architecture runs out of GPU memory with 4,000 samples, while the Fourier Q-functional can evaluate up to 10,000 actions in a single pass. In the low-sample regime (up to 50 actions per state), Q-functionals evaluate in near constant time, while the standard architecture shows the expected linear dependence.

These advantages are substantially improved when introducing a simple optimization: instead of sampling directly from the action space, we pre-compute the basis representations for a large amount of actions (one million in our experiments) and sample from this distribution instead. This optimization is especially helpful for the Legendre basis, which requires a more expensive computation than Fourier or Polynomial. By precomputing the representations, the speed of action-evaluation is decoupled from the complexity of the basis. Using this procedure, we can evaluate actions us-

[1]Code available at https://github.com/samlobel/q_functionals

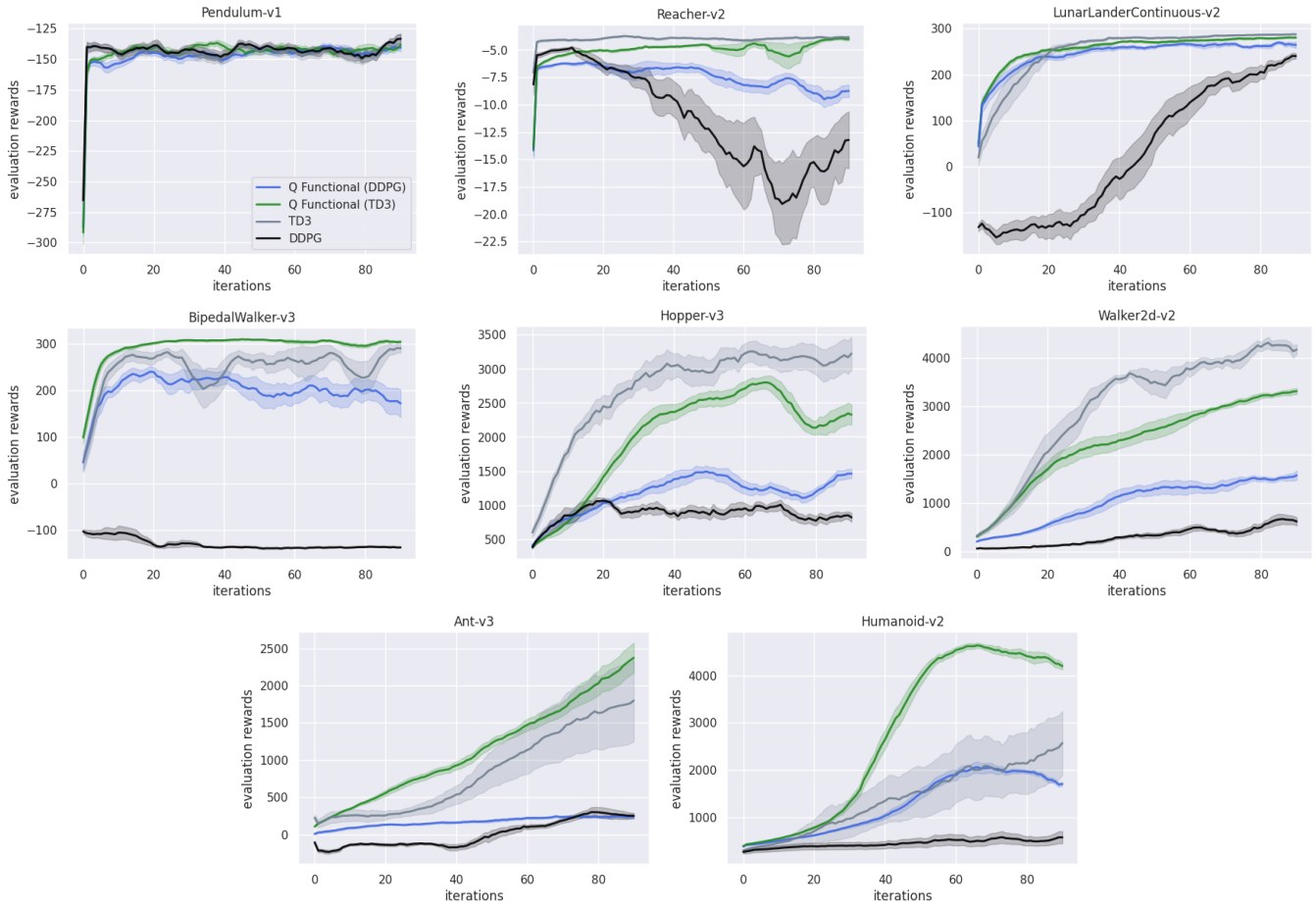

Figure 4: Performance on 8 tasks from the OpenAI continuous control suite. The shaded region represents the standard error over 8 runs. Colors correspond to the same methods across domains. One iteration corresponds to 10,000 environment steps.

ing a rank-3 Legendre Q-functional more than 20 times as quickly, and with less than one tenth the memory-footprint, as a standard Q-function. This procedure has minimal to no effect on reinforcement learning behavior, as confirmed in the Appendix, and we use precomputed representations through the remainder of the experiments.

Experimental Results

In this section, we favorably compare our method to existing off-policy policy-gradient methods, on both the standard and the entropy-regularized RL objective.

Benchmarking

We compare Q-functionals to two popular and strong off-policy policy-gradient baselines:

Deep Deterministic Policy Gradient (DDPG) is a deterministic policy gradient method which relies on a deterministic policy for action-selection and bootstrapping (Lillicrap et al. 2015). The policy network is trained to maximize the Q-function using batch gradient descent, and the value-function is calculated from the Q-function using the policy network's output.

Twin Delayed DDPG (TD3) is an extension of DDPG that addresses issues of overestimation bias and overfitting present in the original algorithm (Fujimoto, van Hoof, and Meger 2018). They do this with the addition of two components. To address overestimation bias, the Q-function is parameterized by two networks, and a state's value is computed as the minimum of both, evaluated on the policy's output. To address overfitting, TD3 introduces *target policy smoothing*, which adds Gaussian noise to the policy's output for use in bootstrapping.

We construct two versions of our architecture for fair comparison, analogous to the two policy gradient methods above. The first computes Q-values using a single network, and chooses the maximum-valued action (Equation 6) to compute values for bootstrapping, analogously to DDPG. Like TD3, the second version computes Q-values using the minimum of two networks to address overestimation bias. To address overfitting, in this version we calculate value using the "top-n of k" strategy detailed in Equation 8.

In Figure 4, we compare these four methods on the OpenAI Gym continuous control suite (Brockman et al. 2016; Todorov, Erez, and Tassa 2012). Tasks have action-

dimension ranging from 1 (Pendulum) to 17 (Humanoid). For all benchmark experiments, we use the Legendre basis with rank 3, and use 1,000 samples for action-selection both in bootstrapping and interaction. Details on environments and architectural choices can be found in the Appendix. We find that Q-functionals compare favorably to their policy-gradient analogues across a variety of tasks. Somewhat surprisingly, Q-functionals perform best on Ant and Humanoid, the two tasks with the highest action-dimension. This is contrary to popular wisdom that sampling methods are unsuitable for large vector spaces, and perhaps points to policy gradient methods sharing the same flaw.

Maximizing Regularized Objectives

Sampling-based value estimation allows for additional flexibility in representing policies over the action space. Instead of being restricted to policies that are either deterministic (Lillicrap et al. 2015), or parameterize a simple state-dependent distribution (Haarnoja et al. 2018a), Q-functionals can sample actions from arbitrary functions over their Q-values. We demonstrate the utility of this approach to ER-RL, under which the Boltzmann distribution over Q-values $Q(s, a)$ maximizes a state's value $V(s)$ (Schulman, Chen, and Abbeel 2017).

We compare a rank 3 Q-functional using the Fourier basis to Soft Actor Critic (SAC), a popular continuous control algorithm for solving ER-RL problems (Haarnoja et al. 2018a), on a simple bandit task (Slivkins et al. 2019). The reward for a given action is the sum of two Gaussian functions centered around -0.5 and 0.5. Figure 5 shows the learned policies of both algorithms. The optimal policy under ER-RL, shown in yellow, gives equal weight to each peak of the bimodal distribution. Because the SAC agent represents its policy as a Gaussian distribution squashed to the valid action space, it cannot represent the bimodal nature of the optimal policy. By contrast, using Equation 7 to derive a policy allows the Q-functional agent to fit the general shape of the optimal policy.

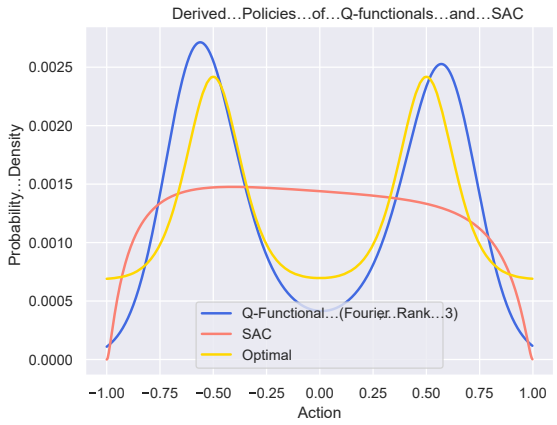

Figure 5: Policies for SAC and Q-functional on a bandit task

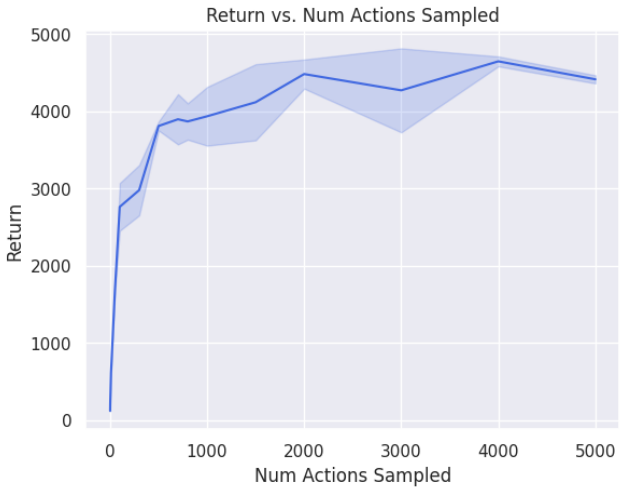

Figure 6: Performance versus samples used to select max-value action on the Humanoid-v2 environment. Shaded region represents the standard error over 2 runs.

Performance with Increased Sampling

As the number of samples used to select actions increases, so does the average Q-value of the returned action. In Figure 6, we investigate whether increased sampling also leads to better performance on the Humanoid task. We train Q-functional agents using the configurations from our benchmark experiments, and evaluate their performance as we vary the number of samples used in action-selection. On this task, we see drastic improvement as sampling increases to 1,000, and then continued diminishing returns as it continues to increase.

Conclusion

The translation between a Q-function and a policy is a central research question in continuous-control reinforcement learning. In this work, we highlight the substantial room for improvement in this question over the currently accepted method of policy-gradients. We introduce Q-functionals as a general framework for Q-functions that can evaluate many actions in parallel. Q-functionals allow for far greater flexibility in how one derives a policy from a Q-function, and achieve impressive results even on tasks with extremely high action dimensions. In addition, under the Q-functional framework, a variety of sample-based techniques become much more practical. We view the flexibility of this framework as well as its strong empirical performance as an attractive alternative to policy-gradient methods in continuous control.

Acknowledgements

We thank Cameron Allen, Saket Tiwari, Rafael Rodriguez-Sanchez, and other members of BigAI for their valuable input. This research was supported in part by NSF GRF #2040433, NSF grants #1717569 #1955361 and NSF CAREER award #1844960. This research was conducted us-

ing computational resources and services at the Center for Computation and Visualization, Brown University. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NSF.

References

Asadi, K.; Parikh, N.; Parr, R. E.; Konidaris, G. D.; and Littman, M. L. 2021. Deep radial-basis value functions for continuous control. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*, 6696–6704.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Ciosek, K.; and Whiteson, S. 2018. Expected policy gradients. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Fujimoto, S.; van Hoof, H.; and Meger, D. 2018. Addressing Function Approximation Error in Actor-Critic Methods. *arXiv e-prints*, arXiv:1802.09477.

Gu, S.; Lillicrap, T.; Sutskever, I.; and Levine, S. 2016. Continuous Deep Q-Learning with Model-based Acceleration. *arXiv e-prints*, arXiv:1603.00748.

Gu, S.; Lillicrap, T.; Sutskever, I.; and Levine, S. 2016. Continuous deep q-learning with model-based acceleration. In *International conference on machine learning*, 2829–2838. PMLR.

Haarnoja, T.; Tang, H.; Abbeel, P.; and Levine, S. 2017. Reinforcement learning with deep energy-based policies. In *International Conference on Machine Learning*, 1352–1361. PMLR.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018a. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018b. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, 1861–1870. PMLR.

Konidaris, G.; Osentoski, S.; and Thomas, P. 2011. Value function approximation in reinforcement learning using the Fourier basis. In *Twenty-fifth AAAI conference on artificial intelligence*.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv e-prints*, arXiv:1509.02971.

Millán, J. D. R.; Posenato, D.; and Dedieu, E. 2002. Continuous-action Q-learning. *Machine Learning*, 49(2): 247–265.

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

Schulman, J.; Chen, X.; and Abbeel, P. 2017. Equivalence between policy gradients and soft q-learning. *arXiv preprint arXiv:1704.06440*.

Simmons-Edler, R.; Eisner, B.; Mitchell, E.; Seung, S.; and Lee, D. 2019. Q-learning for continuous actions with cross-entropy guided policies. *arXiv preprint arXiv:1903.10605*.

Slivkins, A.; et al. 2019. Introduction to multi-armed bandits. *Foundations and Trends® in Machine Learning*, 12(1-2): 1–286.

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1): 9–44.

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

Sutton, R. S.; McAllester, D.; Singh, S.; and Mansour, Y. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In Solla, S.; Leen, T.; and Müller, K., eds., *Advances in Neural Information Processing Systems*, volume 12. MIT Press.

Tessler, C.; Tennenholtz, G.; and Mannor, S. 2019. Distributional policy optimization: An alternative approach for continuous control. *Advances in Neural Information Processing Systems*, 32.

Todorov, E.; Erez, T.; and Tassa, Y. 2012. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, 5026–5033. IEEE.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning*, 8(3-4): 279–292.