Improving Sample Efficiency in Off-policy and Offline Deep Reinforcement Learning

by

Yanqiu Wu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
September, 2022

Keith W. Ross

Acknowledgements

First, I would like to thank my advisor Professor Keith Ross for guiding and supporting me throughout my Ph.D. program. I deeply enjoyed working with Keith over all these years. His creativity, his passion and rigor to education and research has encouraged me when I was assailed by self-doubt and shaped me into the researcher I am now. Without Keith's support and encouragement, I would not be able to achieve what I have accomplished today and find the right path of my research work.

I also want to express special thanks to Professor Kyunghyun Cho and Professor Gus Xia for providing valuable feedback for my Ph.D. depth qualification exam presentation and thesis proposal presentation. I highly appreciate their advice for the entire process. I am also grateful for all the other members of my dissertation committee, Professor He He and Professor Azza Abouzied, for sparing their time and sharing their valuable feedback on my thesis.

I have had the great fortune to work with many wonderful people during my Ph.D. program and research internships. I will try my best to name a few: Yiming Zhang, Che (Watcher) Wang, Quan Vuong, Xinyue Chen, Zijian Zhou, Zheng Wang, Ren Yi, Qing Deng, Qingyang Li, Tony Qin, Ruofan Wu and LeiLei Shi.

As a member of NYU community, I am thankful to the staff members who have helped me in many ways. Thank you Santiago Pizzini, Eric Mao, Xiaoyun (Vivien) Du, Fangqi (Maggie) Mao, Lin Hong and many others for all your hard work to keep the smooth operations of the graduate program and being so supportive and thoughtful during my research time at NYU Shanghai.

Thank you Zhiguo Qi for maintaining the NYU Shanghai HPC center and helping me out of many last minute crises and technical difficulties on the computing clusters.

Thank you Xuanyang for always being supportive and bringing laughter and joy in my life when I stress out.

Finally, I want to thank my parents for their unconditional support and love. Thank you for always taking my education as priority. This journey would not be possible without their support and sacrifices. I am forever grateful to my parents for their encouragement.

Abstract

Reinforcement Learning (RL) is an area of Machine Learning, where agents are trained through trial and error to make a sequence of decisions in some given environment to achieve a goal. Traditional reinforcement learning methodology suffers from the curse of dimensionality. Fortunately, with the help of deep learning, Deep Reinforcement Learning (DRL) can overcome the issue and can often find high performing policies for applications with large state and action spaces. Over the past few years, DRL has achieved major breakthroughs in complex tasks, such as outperforming human players in video games [Mnih et al. 2013; Vinyals et al. 2019], defeating the human world champion in Go [Silver et al. 2016, 2018] and autonomous robotics control [Lillicrap et al. 2019; Haarnoja et al. 2018a].

Despite the recent breakthroughs, sample efficiency remains an important issue in deep reinforcement learning. In some complex tasks, where data collection is very expensive and agents require relatively few interactions with the environment for training, sample efficiency is of central concern for making DRL practical for applications. This thesis addresses the sample efficiency problem in the context of off-policy and offline Deep Reinforcement Learning. We develop training algorithms which not only lead to high asymptotic performing policies, but are also highly sample efficient in both on-line and offline settings. We demonstrate the performance of our methods in simulated robotic locomotion environments.

In the first part of this thesis, we develop a streamlined off-policy algorithm that utilizes an output normalization scheme and non-uniform sampling. We identify the squashing exploration problem and show how maximum entropy DRL [Haarnoja et al. 2018a,b] helps to resolve it. Based on our observation, we develop an alternative output normalization scheme for maximum entropy algorithms. We show that this normalization scheme can then be combined with non-uniform sampling, resulting in high performing policies. Next, we develop a simple off-policy algorithm that takes advantage of a high update-to-data (UTD) ratio and Q-ensembles. Our algorithm demonstrates superior sample efficiency at the early-stage training and also achieve high asymptotic performance at the late-stage training. We employ Q-ensembles and keep several lowest values for updating to address the overestimation bias. Finally, we consider offline deep reinforcement learning. We introduce the novel notion of "upper envelope of the data" and then develop an Imitation-Learning based algorithm based on the notion. Our algorithm is computationally much faster and achieves state-of-the art performance.

Contents

List of Figures

List of Tables

1 Introduction

Building fully autonomous high-performing sequential decision-making agents through trial and error is one of the major difficulties in artificial intelligence [Russell and Norvig 2010; Arulkumaran et al. 2017]. Such intelligent agents have applications in many areas including robotics, natural languages, autonomous driving, etc. Reinforcement Learning (RL) provides a mathematical framework for this learning from experience criteria. It builds on the theory of Markov Decision Processes (MDPs) [Sutton and Barto 2018]. Despite the moderate success on low-dimensional state and action spaces problems, traditional tabular reinforcement learning fails to scale to problems with high-dimensional state and actions spaces due to the curse of dimensionality [Bellman 1957].

With the help of deep learning [Goodfellow et al. 2016b], a modern sub-field of Deep Reinforcement Learning (DRL) has risen. Parameterized policies and value functions are approximated by deep neural networks, which allows the reinforcement learning algorithms to overcome the curse of dimensionality, generalizing to large continuous state and action spaces.

Over the past few years, deep reinforcement learning has achieved major breakthroughs in complex tasks. Agents have learned to play Atari games, outperforming human players [Mnih et al. 2013; Vinyals et al. 2019; Mnih et al. 2016; Bellemare et al. 2016; Wang et al. 2016]. AlphaGo defeats the human world champions in Go [Silver et al. 2018]. In continuous and high-dimensional robotic locomotion simulations, DRL agents have made great strides in various tasks. However, sample efficiency remains an important bottleneck for the feasibility of real world applications of DRL. Interacting with the real world is often fragile, slow or costly, which makes DRL training in the real world expensive both in terms of money and time [Dulac-Arnold et al. 2019]. Because progress in sample efficiency will translate to future real world applications, research on improving sample efficiency in DRL is an important topic. In this thesis, we attempt to address the sample-efficiency issue in the context of off-policy and offline DRL.

1.1 Sample Efficiency

As proposed by Dulac-Arnold et al. [2019] and Hernandez and Brown [2020], to measure algorithmic progress, we measure progress in sample efficiency by comparing the number of samples needed for policy to reach a certain performance level over time and define the sample efficiency of an algorithm for a given task and performance level as:

$$J^{\mathit{eff.}} = \min |\mathcal{D}_i| \quad \text{s.t.} \quad R(\text{Train}(\mathcal{D}_i)) > R_{\min} \tag{1.1}$$

where $|\mathcal{D}_i|$ is the number of samples needed to train to the given performance level on the task and where $R_{\min}$ is the desired performance level. In practice, an alternative approach is to compare the performance of different algorithms for a fixed amount of samples.

1.2 Why Off-policy

Off-policy deep reinforcement learning is opposed to on-policy methods. In off-policy algorithms, the policy that the agent is attempting to evaluate or improve, which we refer to as target policy, is different from the behavioral policy that is used to collect data. This means, off-policy algorithm agents can learn from data generated by a different given policy (such as past experience), while this is impossible for on-policy algorithms. By definition, with the ability to reuse past experience, one of the major advantages of off-policy algorithms is that they are in general more sample-efficient than on-policy methods [Haarnoja et al. 2018b; Chen et al. 2021], as shown in recent works. Therefore, this thesis focuses on improving sample efficiency and performance in off-policy algorithms instead of on-policy algorithms.

1.3 Why Offline

Similar to off-policy algorithms, in the offline setting (also called batch RL), agents utilize previously collected data; however, no additional online data collection is allowed [Ernst et al. 2005a; Fu et al. 2020]. In other words, the offline agent no longer has the ability to interact with the environment using the behaviour policy. Thus, offline RL is also commonly referred as "fully off-policy" RL [Levine et al. 2020]. This problem formulation closely resembles the standard supervised learning problem statement. Progress in sample efficiency under this formulation helps to turn fixed datasets into more powerful decision making policies, which is necessary for training in wide range of safety-critical systems where a partially trained policy cannot be deployed online to collect data. Moreover, a batch RL algorithm can also be deployed as part of a growing-batch algorithm [Lange et al. 2012], where the batch algorithm seeks a high-performing exploitation policy using the data in an experience replay buffer, combines this policy with exploration to add fresh data to the buffer, and then repeats the whole process. Hence, offline DRL is also within the scope of this thesis.

1.4 Thesis Outline

The broader impact of this work is to bring RL closer to more practical applications with positive societal impacts. In the real world, higher sample efficiency usually translates to time saved, lower costs and lower resource consumption. In this thesis, we develop training algorithms which not only lead to high asymptotically performing policies, but are also highly sample efficient in both off-policy and offline settings.

Chapter 2 gives necessary background knowledge for this thesis, which includes the basic definition of Markov Decision Processes (MDPs), a brief introduction to the MuJoCo benchmark environments, basic knowledge of the Q-learning training scheme which is of fundamental importance for off-policy DRL, and an introduction to one of the biggest challenges in offline DRL, extrapolation error.

In Chapter 3, we develop a steamlined off-policy algorithm that improves sample efficiency and achieves state-of-the art performance for the MuJoCo benchmark. We first identify the squashing exploration problem. We then discuss how maximum entropy reinforcement learning helps to address this issue. Based on our observation, we develop an alternative and simpler output normalization scheme that can match the sample efficiency and asymptotic performance of maximum entropy DRL algorithms. We further improve sample efficiency and performance for the MuJoCo benchmark by combining our output normalization with a non-uniform sampling scheme.

Chapter 4 develops a simple model-free off-policy algorithm that takes advantages of a high update-to-data (UTD) ratio and Q-ensembles which demonstrates superior sample efficiency in early-stage training and also achieve high asymptotic performance in late state training. To address the overestimation bias while using a high UTD ratio during training, we employ Q-ensembles and keep several lowest values for updating. Our theoretical and experimental results show that the estimation bias can be controlled using the framework.

In Chapter 5, we consider offline deep reinforcement learning, where extrapolation error is one of the major challenges. We first introduce the novel notion of "upper envelope of the data". We then develop our Imitation-Learning based algorithm based on the notion. Our algorithm is computationally much faster and achieves state-of-the art performance.

Conclusions and future research directions are discussed in Chapter 6.

2 | Preliminaries

In this chapter, we will present the necessary preliminaries and notations employed in this thesis. We divide the chapter into four sections. The first section presents the basic definitions of Markov Decision Processes. Section 2.2 gives a brief introduction on the MuJoCo benchmark environments. Section 2.3 introduces Q-learning, which is of fundamental importance in off-policy deep reinforcement learning algorithms. Finally, in the last section 2.4, we present the most crucial problem in offline reinforcement learning, namely, extrapolation error.

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) are a mathematically idealized form of the reinforcement learning problem for which precise theoretical analysis can be made. Consider a Markov Decision Process [Sutton and Barto 2018] $(\mathcal{S}, \mathcal{A}, r, p, \rho, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ are continuous multi-dimensional state and action spaces. The transition probability is denoted by $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ where $p(s'|s, a)$ is the probability of transitioning into the next state s' by taking action a in state s. The bounded reward function $r(s, a)$ maps state action pairs into the reward received by the agent by taking action a while in state s. ρ is the initial state distribution and γ is the discount factor.

Let $\pi = \pi(a|s), s \in \mathcal{S}, a \in \mathcal{A}$ denote the policy. The policy maps from the state space $\mathcal{S}$ to a probability distribution over the action space $\mathcal{A}$. We further denote $\tau = (s_0, a_0, s_1, a_1, \dots)$

as a sample trajectory following policy π, i.e. $a_t \sim \pi(\cdot|s_t)$, and $s_{t+1} \sim p(\cdot|s_t, a_t)$. Based on the environment and a policy π, we can describe the data collection phase of a reinforcement learning agent as follows: at time t, the agent is currently in state s_t, takes action a according to the policy π, receives reward $r(s, a)$ and then transit to the next state s_{t+1} at time $t+1$ according to $p(\cdot|s_t, a_t)$.

The expected discounted return for policy π beginning in state s is given by:

$$V_\pi(s) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)|s_0 = s] \tag{2.1}$$

The action value function, which is the expected discounted return starting in state s and initially taking action a but then following policy π, is defined as:

$$Q_\pi(s, a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)|s_0 = s, a_0 = a] \tag{2.2}$$

We denote π^* for optimal policies. The corresponding state value function, called the optimal state value function $V^*(s)$, is defined as:

$$V^*(s) \doteq \max_\pi V_\pi(s) \tag{2.3}$$

The corresponding optimal action value function, denoted Q^*, is given as:

$$Q^*(s, a) \doteq \max_\pi Q_\pi(s, a) \tag{2.4}$$

Standard MDP and RL problem formulations seek to maximize the state value function $V_\pi(s)$ over all policies π. For finite state and action spaces, and under suitable conditions for continuous state and action spaces, there exists an optimal policy that is deterministic [Puterman 2014; Bertsekas and Tsitsiklis 1996]. In RL with an unknown environment, exploration is required to learn a suitable policy.

In Deep Reinforcement Learning with continuous action spaces, typically the policy π is modeled by a parameterized policy network which takes as input a state s and outputs a value $\mu(s;\theta)$, where θ represents the current parameters of the policy network [Schulman et al. 2015, 2017; Vuong et al. 2018; Lillicrap et al. 2019; Fujimoto et al. 2018b]. During training, typically additive random noise is added for exploration, so that the actual action taken when in state s takes the form $a = \mu(s;\theta) + \epsilon$ where ϵ is a K-dimensional (K represents the action dimension) Gaussian random vector with each component having zero mean and variance σ. During testing, ϵ is set to zero.

2.2 MuJoCo Benchmark

For most experiments in this thesis, we will be using the MuJoCo physical simulator implemented using OpenAI gym [Todorov et al. 2012; Brockman et al. 2016]. MuJoCo stands for Multi-Joint dynamics with Contact. It is a physics engine for faciliatating research and development in robotics, biomechanics, graphics and animation, and other areas where fast and accurate simulation is needed [Brockman et al. 2016]. The detailed description of each MuJoCo environment is shown in table 2.1. We only consider the five most challenging tasks Ant, HalfCheetah, Hopper, Humanoid and Walker2d in this thesis.

In figure 2.1, we show renderings of all the MuJoCo Environments. All of these environments are stochastic in terms of their initial state, with a Gaussian noise added to a fixed initial state in order to add stochasticity [Brockman et al. 2016].

2.3 Q-learning

In this section, we introduce the Q-learning training scheme which is fundamental for many of the training algorithms developed in this thesis.

Table 2.1: The MuJoCo environments. Information is taken from the Gym official website. The dimensions of the state and action space are listed in the last two columns.

Environment	Goal	State Space	Action Space
Ant	Make a 3D four-legged robot walk	27	8
HalfCheetah	Make a 2D cheetah robot run	17	6
Hopper	Make a 2D robot hop	11	3
Humanoid	Make a 3D two-legged robot walk	376	17
HumanoidStandup	Make a 3D two-legged robot standup	376	17
InvertedDoublePendulum	Balance a pole on a pole on a cart	11	1
InvertedPendulum	Balance a pole on a cart	4	1
Reacher	Make a 2D robot reach to a randomly located target	11	2
Swimmer	Make a 2D robot swim	8	2
Walker2d	Make a 2D robot walk	17	6

The objective of a reinforcement learning agent is to find the best action given its current state. To achieve this, if the optimal action value function $Q^*(s, a)$ is known, then in any given state s, the optimal action a^* can be found by solving

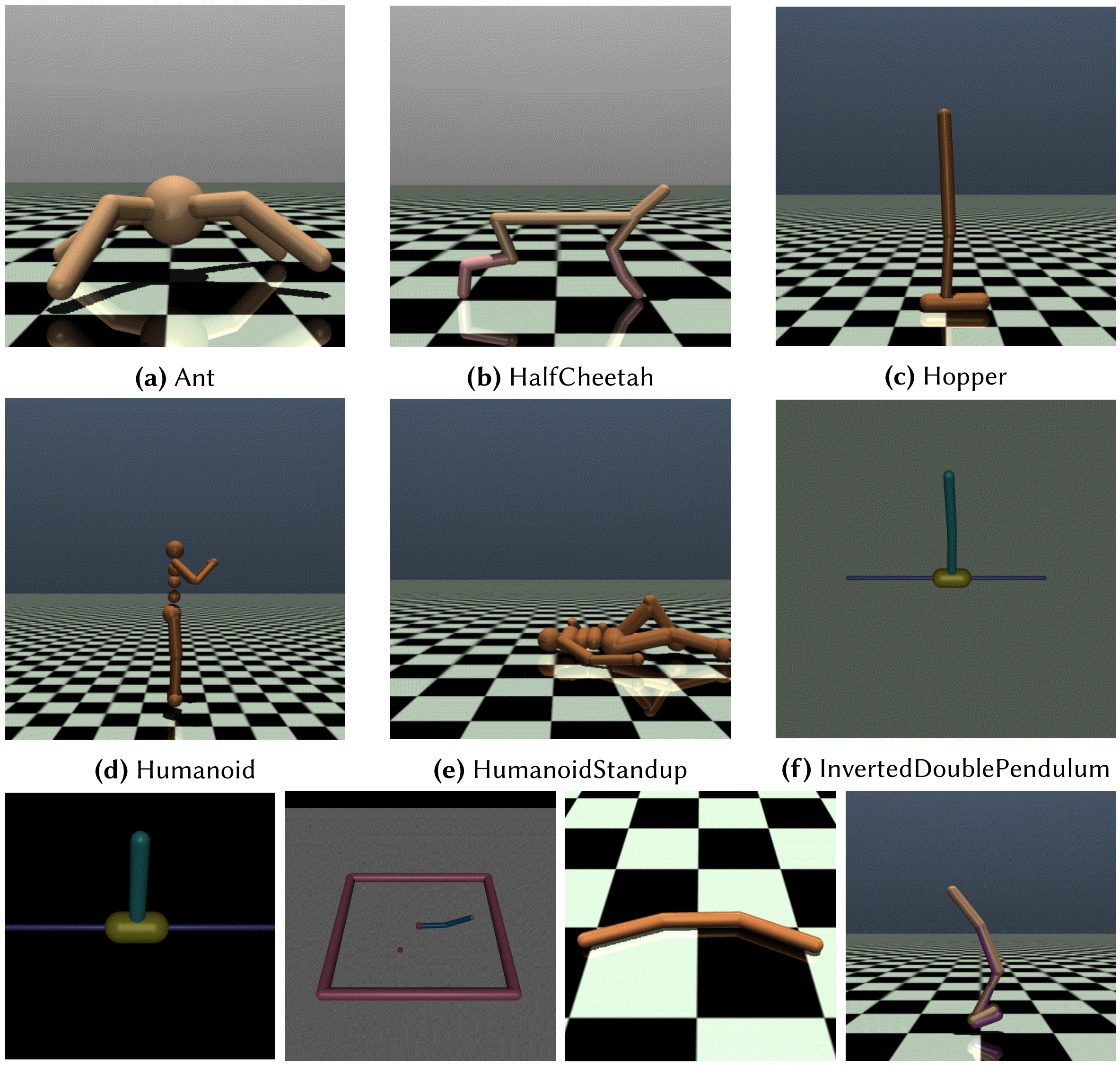
$$a^* = \arg\max_a Q^*(s, a) \tag{2.5}$$

Although the optimal action value function $Q^*(s, a)$ is generally unknown, based on Sutton [1988] and Watkins [1989], the action value function can be learned using temporal difference (TD) learning. Temporal Difference is an update rule based on the Bellman equation [Bellman 1957], which provides a fundamental relationship between the value of a state action pair (s_t, a_t) and the value of the subsequent state action pair (s_{t+1}, a_{t+1}). It is given as:

$$Q^*(s_t, a_t) = \mathbb{E}_{s_{t+1}\sim p}[r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})] \tag{2.6}$$

where $s_{t+1} \sim p$ represents that the next state, s_{t+1}, is sampled by the environment from a distribution $p(\cdot|s_t, a_t)$.

In a large state space, Q-learning algorithms aim to learn an approximator for $Q^*(s,a)$. Basically, we estimate $Q^*(s,a)$ values using a differentiable function approximator $Q_\phi(s,a)$ with parameters ϕ. Starting from the Bellman equation, the mean-squared Bellman error (MSBE) was proposed [Mnih et al. 2015; Lillicrap et al. 2019] for learning the parameters ϕ. It can measure



(a) Ant **(b)** HalfCheetah **(c)** Hopper

(d) Humanoid **(e)** HumanoidStandup **(f)** InvertedDoublePendulum

(g) InvertedPendulum **(h)** Reacher **(i)** Swimmer **(j)** Walker2d

Figure 2.1: The MuJoCo Environments

how closely Q_ϕ comes to satisfying the Bellman equation for a set $\mathcal{D}$ of transitions:

$$L(\phi, \mathcal{D}) = \underset{(s_t,a_t,r,s_{t+1},d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s_t, a_t) - \left(r + \gamma(1-d) \max_{a_{t+1}} Q_\phi(s_{t+1}, a_{t+1}) \right) \right)^2 \right] \tag{2.7}$$

where d indicates whether s_{t+1} is a terminal state or not. That is, when s_{t+1} is a terminal state, the agent gets no additional rewards after the current state s_t. When minimizing this MSBE loss function, to make the training more stable, Q-learning algorithms such as Deep Q-learning (DQN) [Mnih et al. 2015] and Deep Deterministic Policy Gradient (DDPG) [Lillicrap et al. 2019], introduced a target action value network $Q_{\phi'}$ to maintain a fixed target over multiple updates:

$$y = r + \gamma(1-d) \max_{a_{t+1}} Q_{\phi'}(s_{t+1}, a_{t+1}) \tag{2.8}$$

Actions a_{t+1} that maximize the $Q_{\phi'}$ are selected from a target policy network $\mu(s;\theta')$ where the policy network θ is simply learned by:

$$\theta \leftarrow \arg\max_\theta \mathbb{E}_{s\sim\mathcal{D},a\sim\mu_\theta}[Q_\phi(s,a)] \tag{2.9}$$

The parameters of a target network are either updated periodically to match the parameters of the current network, or updated by polyak averaging with some weight ρ: $\phi' \leftarrow \rho\phi + (1-\rho)\phi'$.

2.3.1 Overestimation Bias

For the MSBE update rule introduced in the previous section, the action value function is updated with a greedy target $y = r + \gamma(1-d)\max_{a_{t+1}} Q_{\phi'}(s_{t+1}, a_{t+1})$. However, in discrete action spaces, if the target is susceptible to error ϵ, then the maximum over the value along with its error will generally be greater than the true maximum. In other words, the overestimation bias occurs since the target $\max_{a_{t+1}} Q_{\phi'}(s_{t+1}, a_{t+1})$ is used in the Q-learning update. More concretely, Q_ϕ is an approximation, it is possible that the state action value approximation is higher than the true value for one or more of the actions for the given state. Then taking the maximum over these estimated values is likely to be skewed towards an overestimate [Fujimoto et al. 2018b; Lan et al. 2020a].

For example, even unbiased Q-function estimates $Q_\phi(s_{t+1}, a_{t+1})$ for all actions a_{t+1} vary due to stochasticity: $Q_\phi(s_{t+1}, a_{t+1}) = Q^*(s_{t+1}, a_{t+1}) + \epsilon_{a_{t+1}}$ which for some actions, $\epsilon_{a_{t+1}}$ is positive. Hence, we have $\mathbb{E}[\max_{a_{t+1}} Q_\phi(s_{t+1}, a_{t+1})] \geq \max_{a_{t+1}} \mathbb{E}[Q_\phi(s_{t+1}, a_{t+1})] = \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$ [Lan et al. 2020a]. As a result, even initially zero-mean error can cause value updates to result in a consistent overestimation bias, which will then be propagated through the Bellman equation. It is also proved in Fujimoto et al. [2018b] that in actor-critic methods, the value estimate in deterministic policy gradients will also be an overestimation under some basic assumptions. This overestimation issue is problematic as the errors induced by function approximation are unavoidable.

To address the overestimation problem, Fujimoto et al. [2018b] proposed Clipped Double Q-learning, which is a clipped variant of Double Q-learning [Hasselt 2010; Hasselt et al. 2016]. Clipped Double Q-learning has two action value functions Q_{ϕ_1}, Q_{ϕ_2} and correspondingly two target action value functions $Q_{\phi_1'}, Q_{\phi_2'}$. The minimum between the two estimates is taken to calculate the target update:

$$y = r + \gamma(1-d) \min_{i=1,2} Q_{\phi_i'}(s_{t+1}, a_{t+1}) \qquad a_{t+1} = \mu(s_{t+1}; \theta) \tag{2.10}$$

Both action value functions use the same target to update. This update rule may induce an underestimation bias, but underestimation is better than overestimation bias, as the value of underestimated actions will not be explicitly propagated through the Bellman equation.

Another technique proposed by Fujimoto et al. [2018b] which is helpful with overestimation bias is called Target Policy Smoothing Regularization. When using a deterministic policy network, if the action value function incorrectly estimates a sharp peak for some actions, the policy will quickly exploit that error peak resulting in brittle and incorrect behaviors. In practice, target policy smoothing regularization adds a small amount of random noise to the target policy and averages over mini-batches to approximate the Q-function target:

$$\begin{aligned} y &= r + \gamma(1-d) \min_{i=1,2} Q_{\phi'_i}(s_{t+1}, a_{t+1}) \\ a_{t+1} &= \mu(s_{t+1};\theta) + \epsilon, \epsilon \sim \text{clip}(\mathcal{N}(0,\sigma), -c, c) \end{aligned} \tag{2.11}$$

The added noise is clipped to keep the target close to the original action. In summary, target policy smoothing smooths out the action value function over similar actions to avoid exploitation of incorrect peak for some actions.

2.4 Extrapolation Error

In offline reinforcement learning, the agent has no access to the environment and is unable to collect data during training. The goal is to train agents to learn from a fixed batch of data, which has already been generated by some other policy interacting with the environment. Standard off-policy deep reinforcement learning algorithms, such as DQN and DDPG [Mnih et al. 2015; Lillicrap et al. 2019], are incapable of learning in this fixed batch setting due to errors introduced by extrapolation.

Extrapolation error [Fujimoto et al. 2018a; Kumar et al. 2019b] is an error in off-policy value learning which is introduced by the mismatch between the dataset and true state-action visitation data of the policies used during training. The action value estimate $Q_\phi(s_t, a_t)$ is affected by extrapolation error during the value update where the target policy $\mu(s_{t+1};\theta)$ selects an unfamiliar action at the next state s_{t+1} in the boostrap value estimate, such that (s_{t+1}, a_{t+1}) is unlikely contained in the dataset.

Extrapolation error occurs due to three related factors. The first related factor is that we have absent data. That is, if any state action pair (s, a) is unavailable or not seen in the dataset, then error is introduced as some function of the amount of similar data and approximation error [Fujimoto et al. 2018a]. In other words, when estimating the value of $Q_\phi(s_{t+1}, a_{t+1})$ in equation 2.7, where a_{t+1} is selected by the target policy, the approximation may be arbitrarily bad without sufficient data near (s_{t+1}, a_{t+1}).

Another cause of the extrapolation error is model bias. When performing off-policy Q-learning with a batch $\mathcal{D}$, the Bellman operator is approximated by sampling transitions tuples (s_t, a_t, r, s_{t+1}) from $\mathcal{D}$ to estimate the expectation over s_{t+1}. However, for a stochastic MDP, without infinite state-action access, this can lead to a biased approximation of the transition dynamics:

$$Q(s_t, a_t) \approx \mathbb{E}_{s_{t+1} \sim \mathcal{D}}[r + \gamma Q(s_{t+1}, \mu(s_{t+1}; \theta))] \tag{2.12}$$

The expectation is with respect to transitions data in the batch $\mathcal{D}$ instead of the true MDP.

Training mismatch is another factor leading to extrapolation error. Suppose we have sufficient data, and transitions are sampled uniformly from the dataset. The loss 2.7 is weighted with respect to the likelihood of data in the batch, which is now given as

$$\approx \frac{1}{|\mathcal{D}|} \sum_{(s_t, a_t, r, d, s_{t+1}) \sim \mathcal{D}} \|r + \gamma(1-d) Q'_\phi(s_{t+1}, \mu(s_{t+1}; \theta)) - Q_\phi(s_t, a_t)\|^2 \tag{2.13}$$

If the distribution of data in the batch $\mathcal{D}$ mismatches the distribution of the current policy $\mu(;\theta)$, the value function may do a poor job estimating state action values for actions selected by the current policy.

In summary, training agents with fixed offline data can result in large amounts of extrapolation error which state of the art off-policy deep reinforcement learning algorithms fail to address. In the following sections, we will discuss two common ways, policy regularization and critic penalty, which aim to address extrapolation error by using appropriate regularizers to constrain the learned policy to stay close to the batch data.

2.4.1 Policy Regularization

Policy regularization can be imposed either for critic or policy learning. The basic idea is to restrict the policy from choosing unfamiliar actions.

Batch-Constrained deep Q-learning (BCQ) proposed by Fujimoto et al. [2018a] restricts the policy to select familiar actions in the offline dataset by adopting a conditional Variational AutoEncoder (VAE) [Kingma and Welling 2014; Sohn et al. 2015]. The VAE, denoted as G_ω, models the distribution by transforming an underlying latent space, which takes state and action pairs as input, and is trained to reconstruct the input actions. When optimizing the approximate Q-function over actions, instead of optimizing over all actions, BCQ optimizes over a subset of actions generated by the VAE. In other words, n candidate actions are sampled from the VAE and the action with the largest critic-approximated value is selected among the candidates for update. A perturbation model, which employs an additional neural network ξ_ψ that outputs an adjustment to an action, is further introduced in BCQ:

$$\theta \leftarrow \arg \max_{a_i + \xi_\psi(s,a)} Q_\phi(s, a_i + \xi_\psi(s,a)) \quad \{a_i \sim G_\omega(s)\}_{i=1}^{n} \tag{2.14}$$

One limitation of BCQ is that a large number of sampled candidate actions is required for competitive performance [Ghasemipour et al. 2020].

An alternative approach is to use divergence penalty. Instead of applying hard constraints on action selection, Wu et al. [2019] proposed to use KL-divergence to regularize the standard policy learning in equation 2.9 to stay close to the batch distribution:

$$\theta \leftarrow \arg\max_\theta \mathbb{E}_{s\sim\mathcal{D}}\Big[\mathbb{E}_{a\sim\mu_\theta}[Q_\phi(s,a)] - \alpha D_{KL}(\mu(s;\theta)\|\mu(s;\theta_B))\Big] \tag{2.15}$$

where θ_B represents the behavioral policy which is used to generate the offline dataset $\mathcal{D}$. However, a common failure for this kind of policy regularization methods is that the Q-function receives no learning signal for actions not observed in the offline buffer. Thus, when Q-function is queried on out-of-distribution actions, critic extrapolation may still dominate policy regularization.

2.4.2 Critic Penalty

Critic penalty methods attempt to incorporate some divergence regularization into the action value functions. The basic idea is to punish the state action value estimates for out-of-distribution data. Specifically, Q-values are pushed down for actions sampled from the training policy while minimizing standard TD-error.

Based on the idea, Kumar et al. [2020] created Conversative Q-learning (CQL), which extends the standard critic loss in equation 2.7 with additional regularization. The regularization is to minimize the Q-values if actions a are sampled from a training policy, and maximize the Q-values if actions a are sampled from the offline dataset $\mathcal{D}$:

$$\min_{\phi} L(\phi, \mathcal{D}) + \lambda \mathbb{E}_{s\sim\mathcal{D}}\left[\log \sum_{a} \exp(Q_{\phi}(s,a)) - \mathbb{E}_{a\sim\mathcal{D}}[Q_{\phi}(s,a)]\right] \tag{2.16}$$

Unlike policy regularization algorithms, such critic penalty methods provide a learning signal to the critic Q-values on the entire action space. However, the log-sum-exp term that appears in the formulation is not tractable for continuous actions and must be computed via numerical integration; thus, there is a significant computational burden for actor critic training.

3 | Streamlined Off-policy Algorithms

3.1 Introduction

Recently a number of new off-policy deep reinforcement learning algorithms have been proposed for control tasks with continuous state and action spaces, including Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3) [Lillicrap et al. 2019; Fujimoto et al. 2018b]. TD3, which introduced clipped double-Q learning, delayed policy updates and target policy smoothing, has been shown to be significantly more sample efficient than popular on-policy methods for a wide range of MuJoCo benchmarks.

The field of Deep Reinforcement Learning (DRL) has also recently seen a surge in the popularity of maximum entropy RL algorithms. In particular, Soft Actor-Critic (SAC), which combines off-policy learning with maximum-entropy RL, not only has many attractive theoretical properties, but can also give superior performance on a wide-range of MuJoCo environments, including on the high-dimensional environment Humanoid for which both DDPG and TD3 perform poorly [Haarnoja et al. 2018a,b; Langlois et al. 2019]. SAC and TD3 have similar off-policy structures with clipped double-Q learning, but SAC also employs maximum entropy reinforcement learning.

Maximum entropy reinforcement learning takes a different approach than Equation (2.1) by optimizing policies to maximize both the expected return and the expected entropy of the policy [Ziebart et al. 2008; Ziebart 2010; Todorov 2008; Rawlik et al. 2013; Levine and Koltun 2013; Levine et al. 2016; Nachum et al. 2017; Haarnoja et al. 2017, 2018a,b].

In particular, the maximum entropy RL objective is:

$$V_\pi(s) = \sum_{t=0}^{\infty} \gamma^t \mathbb{E}_\pi[r(s_t, a_t) + \lambda H(\pi(\cdot|s_t))|s_0 = s] \tag{3.1}$$

where $H(\pi(\cdot|s))$ is the entropy of the policy when in state s, and the temperature parameter λ determines the relative importance of the entropy term against the reward. For maximum entropy DRL, when given state s the policy network will typically output a K-dimensional vector $\sigma(s;\theta)$ in addition to the vector $\mu(s;\theta)$. The action selected when in state s is then modeled as $\mu(s;\theta) + \epsilon$ where $\epsilon \sim N(0, \sigma(s;\theta))$.

Maximum entropy RL has been touted to have a number of conceptual and practical advantages for DRL [Haarnoja et al. 2018a,b]. For example, it has been argued that the policy is incentivized to explore more widely, while giving up on clearly unpromising avenues. It has also been argued that the policy can capture multiple modes of near-optimal behavior, that is, in problem settings where multiple actions seem equally attractive, the policy will commit equal probability mass to those actions.

In this chapter, we first seek to understand the primary contribution of the entropy term to the performance of maximum entropy algorithms. We demonstrate that when using the standard objective without entropy along with standard additive noise exploration, there is often insufficient exploration due to the bounded nature of the action spaces in the MuJoCo benchmark. Specifically, the outputs of the policy network are often way outside the bounds of the action space, so that they need to be squashed to fit within the action space. The squashing results in actions persistently taking on their maximal values, resulting in insufficient exploration. The entropy term in the SAC objective forces the outputs to have sensible values, so that even with squashing, exploration is maintained.

With this insight, we propose the Streamlined Off Policy (SOP) algorithm, which is a minimalistic off-policy algorithm that includes a simple but crucial output normalization. The normalization addresses the bounded nature of the action spaces, allowing satisfactory exploration throughout training. We also consider using inverting gradients (IG) [Hausknecht and Stone 2015] with the streamlined scheme, which we refer to as SOP_IG. Both approaches use the standard objective without the entropy term. Our results show that SOP and SOP_IG match the sample efficiency and robust performance of SAC, including on the challenging Ant and Humanoid environments.

Having just matched SAC performance without using entropy maximization, we then seek to further improve sample-efficiency and attain state-of-the-art performance by employing a non-uniform sampling method for selecting transitions from the replay buffer during training. Our method, called Emphasizing Recent Experience (ERE), samples more aggressively recent experience while not neglecting past experience. We show that when SOP, SOP IG, or SAC is combined with ERE, the resulting algorithm out-performs SAC and provides state of the art performance.

3.2 Squashing Exploration Problem

3.2.1 Bounded Action Spaces

Continuous environments typically have bounded action spaces, that is, along each action dimension k, there is a minimum possible action value $a_k^{\min}$ and a maximum possible action value $a_k^{\max}$. When selecting an action, the action needs to be selected within these bounds before the action can be taken. DRL algorithms often handle this by squashing the action so that it fits within the bounds. For example, if along any one dimension the value $\mu(s;\theta)+\epsilon$ exceeds $a^{\max}$, the action is set (clipped) to $a^{\max}$. Alternatively, a smooth form of squashing can be employed. For example, suppose $a_k^{\min} = -M$ and $a_k^{\max} = +M$ for some positive number M, then a smooth form of squashing could use $a = M \tanh(\mu(s;\theta)+\epsilon)$ in which tanh() is being applied to each component of the K-dimensional vector. DDPG [Hou et al. 2017] and TD3 [Fujimoto et al. 2018b] use clipping, and

SAC [Haarnoja et al. 2018a,b] uses smooth squashing with the tanh() function. For concreteness, henceforth we will assume that smooth squashing with the tanh() is employed.

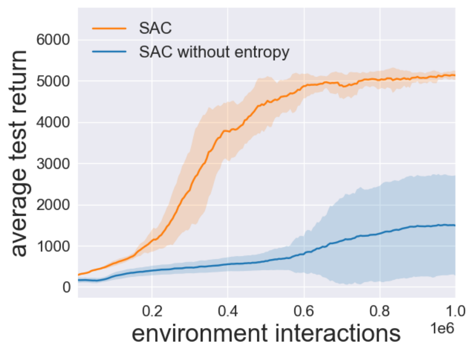
We now make a simple but crucial observation: squashing actions to fit into a bounded action space can have a disastrous effect on additive-noise exploration strategies. To see this, let the output of the policy network be $\mu(s) = (\mu_1(s), \ldots, \mu_K(s))$. Consider an action taken along one dimension k, and suppose $\mu_k(s) >> 1$ and $|\epsilon_k|$ is relatively small compared to $\mu_k(s)$. Then the action $a_k = M \tanh(\mu_k(s) + \epsilon_k)$ will be very close (essentially equal) to M. If the condition $\mu_k(s) >> 1$ persists over many consecutive states, then a_k will remain close to 1 for all these states, and consequently there will be essentially no exploration along the kth dimension. We will refer to this problem as the *squashing exploration problem*. We will argue that algorithms using the standard objective (Equation 2.1) with additive noise exploration can be greatly impaired by squashing exploration.

3.2.2 Contribution of the Entropy Maximization

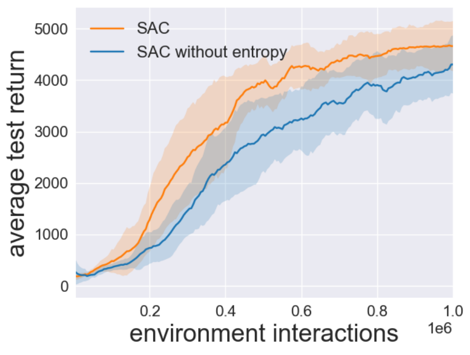
In this section, we argue that the principal contribution of the entropy term in the SAC objective is to resolve the squashing exploration problem, thereby maintaining sufficient exploration when facing bounded action spaces.

To argue this, we consider two DRL algorithms: SAC with adaptive temperature [Haarnoja et al. 2018b], and SAC with entropy removed altogether (temperature set to zero) but everything else the same. We refer to them as SAC and as SAC without entropy. For SAC without entropy, for exploration we use additive zero-mean Gaussian noise with σ fixed at 0.3. Both algorithms use tanh squashing. We compare these two algorithms on two MuJoCo environments: Humanoid-v2 and Walker2d-v2.

Figure 3.1 shows the performance of the two algorithms with 10 seeds. For Humanoid, SAC performs much better than SAC without entropy. However, for Walker, SAC without entropy performs nearly as well as SAC, implying maximum entropy RL is not as critical for this environment.



(a) Humanoid-v2

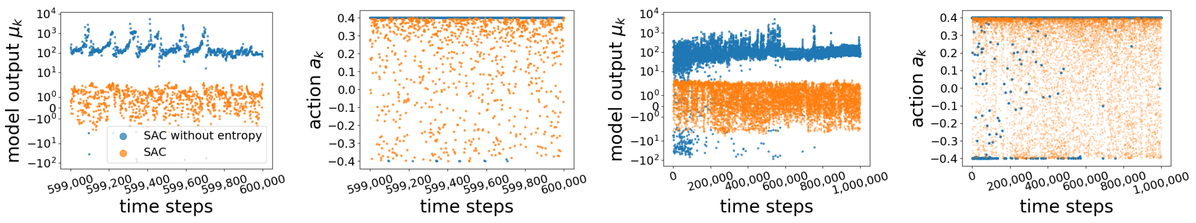


(b) Walker2d-v2

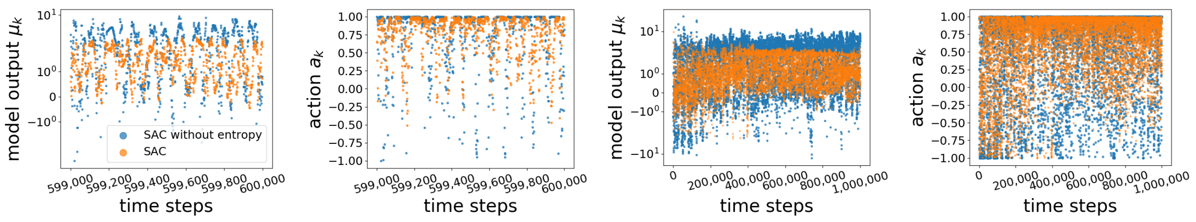
Figure 3.1: SAC performance with and without entropy maximization

To understand why entropy maximization is important for one environment but less so for another, we examine the actions selected when training these two algorithms. Humanoid and Walker have action dimensions $K = 17$ and $K = 6$, respectively. Here we show representative results for one dimension for both environments. The top and bottom rows of Figure 3.2 shows results for Humanoid and Walker, respectively. The first column shows the μ_k values for an interval of 1,000 consecutive time steps, namely, for time steps 599,000 to 600,000. The second column shows the actual action values passed to the environment for these time steps. The third and fourth columns show a concatenation of 10 such intervals of 1000 time steps, with each interval coming from a larger interval of 100,000 time steps.

The top and bottom rows of Figure 3.2 are strikingly different. For Humanoid using SAC with entropy, the $|\mu_k|$ values are small, mostly in the range [-1.5,1.5], and fluctuate significantly. This allows the action values to also fluctuate significantly, providing exploration in the action space. On the other hand, for SAC without entropy the $|\mu_k|$ values are typically huge, most of which are well outside the interval [-10,10]. This causes the actions a_k to be persistently clustered at either M or -M, leading to essentially no exploration along that dimension. For Walker, we see that for both algorithms, the μ_k values are sensible, mostly in the range [-1,1] and therefore the actions chosen by both algorithms exhibit exploration.



(a) Humanoid-v2



(b) Walker2d-v2

Figure 3.2: μ_k and a_k values from SAC and SAC without entropy maximization. See section 3.2 for a discussion.

In conclusion, the principal benefit of maximum entropy RL in SAC for the MuJoCo environments is that it resolves the squashing exploration problem. For some environments (such as Walker), the outputs of the policy network take on sensible values, so that sufficient exploration is maintained and overall good performance is achieved without the need for entropy maximization. For other environments (such as Humanoid), entropy maximization is needed to reduce the magnitudes of the outputs so that exploration is maintained and overall good performance is achieved.

3.3 Methods

3.3.1 Output Normalization

As we observed in the previous section, in some environments the policy network output values $|\mu_k|, k = 1, \ldots, K$, can become persistently huge, which leads to insufficient exploration due to the squashing. Based on the observation, we propose a simple solution of normalizing the outputs of the policy network when they collectively (across the action dimensions) become too large.

Algorithm 1 Streamlined Off-Policy

```
 1: Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$
 2: Throughout the output of the policy network $\mu_\theta(s)$ is normalized if $G > 1$. (See Section 4.1.)
 3: Set target parameters equal to main parameters $\phi_{\text{targ}_i} \leftarrow \phi_i$  for i = 1, 2
 4: repeat
 5:    Generate an episode using actions $a = M\tanh(\mu_\theta(s) + \epsilon)$ where $\epsilon \sim \mathcal{N}(0, \sigma_1)$.
 6:    for $j$ in range(however many updates) do
 7:       Randomly sample a batch of transitions, $B = \{(s, a, r, s)\}$ from $\mathcal{D}$
 8:       Compute targets for Q functions:
             $y_q(r, s') = r + \gamma \min_{i=1,2} Q_{\phi_{\text{targ}_i}}(s', M\tanh(\mu_\theta(s') + \delta)) \quad \delta \sim \mathcal{N}(0, \sigma_2)$
 9:       Update Q-functions by one step of gradient descent using
             $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(Q_{\phi_i}(s, a) - y_q(r, s')\right)^2$ for $i = 1, 2$
10:       Update policy by one step of gradient ascent using
             $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, M\tanh(\mu_\theta(s)))$
11:       Update target networks with
             $\phi_{\text{targ}_i} \leftarrow \rho\phi_{\text{targ}_i} + (1 - \rho)\phi_i$  for $i = 1, 2$
12:    end for
13: until Convergence
```

To this end, let $\mu = (\mu_1, \ldots, \mu_K)$ be the output of the original policy network, and let $G = \sum_k |\mu_k|/K$. The G is simply the average of the magnitudes of the components of μ. The normalization procedure is as follows. If $G > 1$, then we reset $\mu_k \leftarrow \mu_k/G$ for all $k = 1, \ldots, K$; otherwise, we leave μ unchanged. With this simple normalization, we are assured that the average of the normalized magnitudes is never greater than one.

Our Streamlined Off Policy (SOP) algorithm is described in Algorithm 1. The SOP algorithm is "streamlined" as it has no entropy terms, temperature adaptation, target policy parameters or delayed policy updates. It is essentially TD3 minus the delayed policy updates and the target policy parameters but with the addition of the normalization described above.

3.3.2 Inverting Gradients

We also consider using SOP but replacing the output normalization with the Inverting Gradients (IG) scheme [Hausknecht and Stone 2015]. In this scheme, when gradients suggest increasing the action magnitudes, gradients are down scaled if actions are within the boundaries, and inverted if otherwise. We remove the tanh function from SOP and use Inverting Gradients instead to bound the actions. More specifically, let p be the output of the last layer of the policy network. During exploration p will be the mean of a normal distribution that we sample actions from. Let $p_{\min}$ and $p_{\max}$ be the action boundaries. The IG approach can be summarized as follows [Hausknecht and Stone 2015]:

$$\nabla_p = \nabla_p \cdot \begin{cases} \frac{p_{\max}-p}{p_{\max}-p_{\min}} & \text{if } \nabla_p \text{ suggests increasing } p \\ \frac{p-p_{\min}}{p_{\max}-p_{\min}} & \text{otherwise} \end{cases} \tag{3.2}$$

Where ∇_p is the gradient of the policy loss w.r.t to p. During a policy network update, we first backpropagate the gradients from the outputs of the Q network to the output of the policy network for each data point in the batch, we then compute the ratio $(p_{\max} - p)/(p_{\max} - p_{\min})$ or $(p_{\max} - p)/(p_{\max} - p_{\min})$ for each p value (each action dimension), depending on the sign of the gradient. We then backpropagate from the output of the policy network to parameters of the policy network, and we modify the gradients in the policy network according to the ratios we computed. We refer to SOP with IG as SOP_IG.

We compares SAC (with temperature adaptation [Haarnoja et al. 2018a,b]) with SOP, SOP_IG, and TD3 plus the simple normalization (which we call TD3+) for five of the most challenging MuJoCo environments in section 3.4. Results show that SOP, the simplest of all the schemes, performs as well or better than all other schemes. In particular, SAC and SOP have similar sample efficiency and robustness across all environments

3.3.3 Non-uniform Sampling

Having matched SAC performance by SOP without using entropy maximization is not enough, we propose a novel non-uniform sampling method for selection transitions from the replay buffer during training to further improve sample efficiency and attain state-of-the-art performance.

Algorithm 2 SOP with Emphasizing Recent Experience

```
1:  Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$ of
    size $N$, initial $\eta_0$, recent and max performance improvement $I_{recent} = I_{max} = 0$.
2:  Set target parameters equal to main parameters $\phi_{\text{targ,i}} \leftarrow \phi_i$ for i = 1, 2
3:  **repeat**
4:      Generate an episode using actions $a = M\tanh(\mu_\theta(s) + \epsilon)$ where $\epsilon \sim \mathcal{N}(0, \sigma_1)$.
5:      update $I_{recent}, I_{max}$ with training episode returns, let $K$ = length of episode
6:      compute $\eta = \eta_0 \cdot \frac{I_{recent}}{I_{max}} + (1 - \frac{I_{recent}}{I_{max}})$
7:      **for** $j$ in range($K$) **do**
8:          Compute $c_k = N \cdot \eta^{k \frac{1000}{K}}$
9:          Sample a batch of transitions, $B = \{(s, a, r, s)\}$ from most recent $c_k$ data in $\mathcal{D}$
10:         Compute targets for Q functions:
                $y_q(r, s') = r + \gamma \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', M\tanh(\mu_\theta(s') + \delta)) \qquad \delta \sim \mathcal{N}(0, \sigma_2)$
11:         Update Q-functions by one step of gradient descent using
                $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(Q_{\phi,i}(s, a) - y_q(r, s')\right)^2$ for $i = 1, 2$
12:         Update policy by one step of gradient ascent using
                $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi,1}(s, M \tanh(\mu_\theta(s)))$
13:         Update target networks with
                $\phi_{\text{targ, i}} \leftarrow \rho \phi_{\text{targ, i}} + (1 - \rho)\phi_i$ for $i = 1, 2$
14:     **end for**
15: **until** Convergence
```

The basic idea is that during the parameter update phase, the first mini-batch is sampled from the entire buffer, then for each subsequent mini-batch we gradually reduce our range of sampling to sample more from recent data. Specifically, assume that in the current update phase we are to make 1000 mini-batch updates. Let N be the max size of the buffer. Then for the k^{th} update, we sample uniformly from the most recent c_k data points, where $c_k = N \cdot \eta^k$ and $\eta \in (0, 1]$ is a hyper-parameter that determines how much emphasis we put on recent data. $\eta = 1$ is uniform sampling. When $\eta < 1$, c_k decreases as we perform each update. η can be made to adapt to the learning speed of the agent so that we do not have to tune it for each environment. We call this non-uniform sampling scheme Emphasizing Recent Experience (ERE). The algorithmic details of such an adaptive scheme is given in algorithm 2.

The effect of such a sampling formulation is twofold. The first is recent data have a higher chance of being sampled. The second is that sampling is done in an ordered way: we first sample from all the data in the buffer, and gradually shrink the range of sampling to only sample from the most recent data. This scheme reduces the chance of over-writing parameter changes made by new data with parameter changes made by old data [French 1999; McClelland et al. 1995; McCloskey and Cohen 1989; Ratcliff 1990; Robins 1995]. This allows us to quickly obtain information from recent data, and better approximate the value functions near recently-visited states, while still maintaining an acceptable approximation near states visited in the more distant past.

Consider the case of uniform sampling, if we uniformly sample several times from a fixed buffer (uniform fixed), where the buffer is filled, and no new data is coming in, then the expected number of times a data point has been sampled is the same for all data points. Now consider a scenario where we have a buffer of size 1000 (FIFO queue), we collect one data at a time, and then perform one update with mini-batch size of one. If we start with an empty buffer and sample uniformly (uniform empty), as data fills the buffer, each data point gets less and less chance of being sampled. Specifically, start from timestep 0, over a period of 1000 updates, the expected number of times the tth data (the data point collected at tth timestep) has been sampled is: $\frac{1}{t} +$

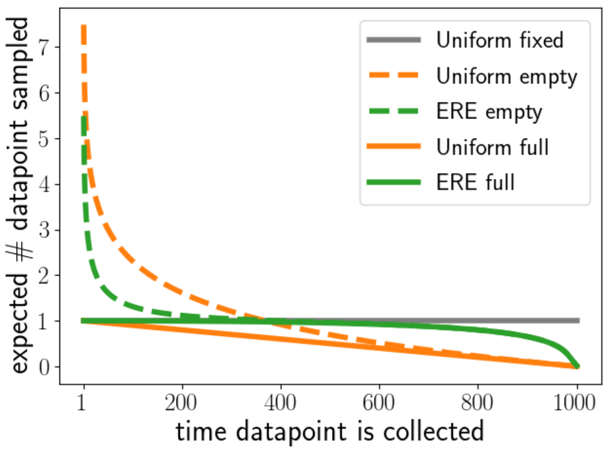


Figure 3.3: Comparison between uniform and ERE sampling

$\frac{1}{t+1} + \cdots + \frac{1}{1000}$. And if we start with a filled buffer and sample uniformly (uniform full), then the expected number of times the tth data has been sampled is $\sum_{t'=t}^{1000} \frac{1}{1000} = \frac{1000-t}{1000}$.

Figure 3.3 shows the expected number of times a data point has been sampled (at the end of 1000 updates) as a function of its position in the buffer. We see that when uniform sampling is used, older data are expected to get sampled much more than newer data, especially in the empty buffer case. This is undesirable because when the agent is improving and exploring new areas of the state space; new data points may contain more interesting information than the old ones, which have already been updated many times.

When we apply the ERE scheme, we effectively skew the curve towards assigning higher expected number of samples for the newer data, allowing the newer data to be frequently sampled soon after being collected, which can accelerate the learning process. In Figure 3.3 we can see that the curves for ERE (ERE empty and ERE full) are much closer to the horizontal line (Uniform fixed), compared to when uniform sampling is used. With ERE, at any point during training, we expect all data points currently in the buffer to have been sampled approximately the same number of times.

We also implement the proportional variant of Prioritized Experience Replay [Schaul et al. 2015] with SOP.

Since SOP has two Q-networks, we redefine the absolute TD error $|\delta|$ of a transition (s, a, r, s') to be the average absolute TD error in the Q network update:

$$|\delta| = \frac{1}{2} \sum_{l=1}^{2} |y_q(r, s') - Q_{\phi,l}(s, a)| \tag{3.3}$$

Within the sum, the first term $y_q(r, s') = r + \gamma \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tanh(\mu_\theta(s') + \delta)), \delta \sim \mathcal{N}(0, \sigma_2)$ is simply the target for the Q network, and the term $Q_{\theta,l}(s, a)$ is the current estimate of the l^{th} Q network. For the i^{th} data point, the definition of the priority value p_i is $p_i = |\delta_i| + \epsilon$. The probability of sampling a data point $P(i)$ is computed as:

$$P(i) = \frac{p_i^{\beta_1}}{\sum_j p_j^{\beta_1}} \tag{3.4}$$

where β_1 is a hyperparameter that controls how much the priority value affects the sampling probability, which is denoted by α in Schaul et al. [2015], but to avoid confusion with the α in SAC, we denote it as β_1. The importance sampling (IS) weight w_i for a data point is computed as:

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^{\beta_2} \tag{3.5}$$

where β_2 is denoted as β in Schaul et al. [2015].

Based on the SOP algorithm, we change the sampling method from uniform sampling to sampling using the probabilities $P(i)$, and for the Q updates we apply the IS weight w_i. This gives SOP with Prioritized Experience Replay (SOP+PER). We note that as compared with SOP+PER, ERE does not require a special data structure and has negligible extra cost, while PER uses a sum-tree structure with some additional computational cost.

The ERE scheme is also similar to an exponential sampling (EXP) scheme where we assign the probability of sampling according to the probability density function of an exponential distribution. Essentially, in such a sampling scheme, the more recent data points get exponentially more probability of being sampled compared to older data.

For the i^{th} most recent data point, the probability of sampling a data point $P(i)$ is computed as:

$$P(i) = \lambda e^{-\lambda x} \tag{3.6}$$

We apply this sampling scheme to SOP and refer to this variant as SOP+EXP.

3.4 Experiments

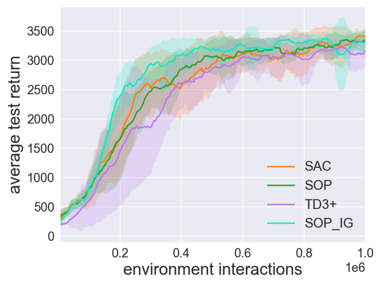
3.4.1 Results on MuJoCo

We use the same code baseline code comparing SAC (with temperature adaptation [Haarnoja et al. 2018a,b]) with SOP, SOP_IG, and TD3 plus the simple normalization (which we call TD3+) for five of the most challenging MuJoCo environments. We train each of the algorithms with 10 seeds. Each algorithm performs five evaluation rollouts every 5000 environment steps. The solid curves correspond to the mean, and the shaded region to the standard deviation of the returns over seeds. In figure 3.4, results show that SOP, the simplest of all the schemes, performs as well or better than all other schemes. In particular, SAC and SOP have similar sample efficiency and robustness across all environments.

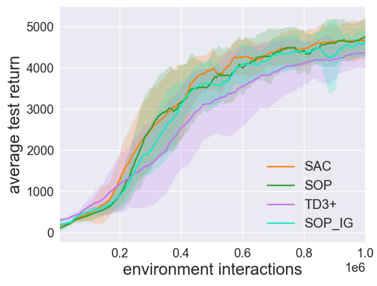
TD3+ has slightly weaker asymptotic performance for Walker and Humanoid. SOP_IG initially learns slowly for Humanoid with high variance across random seeds, but gives similar asymptotic performance. These experiments confirm that the sample efficiency and performance of SAC can be achieved without maximum entropy RL.

3.4.2 Experimental Results for ERE

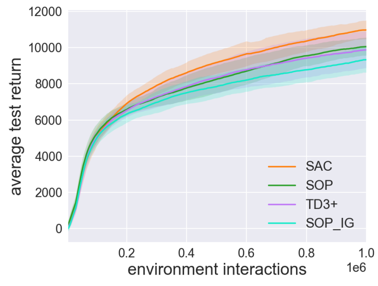
Figure 3.5 compares the performance of SAC (considered the baseline here), SAC+ERE, SOP+ERE, and SOP_IG+ERE. ERE gives a significant boost to all three algorithms, surpassing SAC and



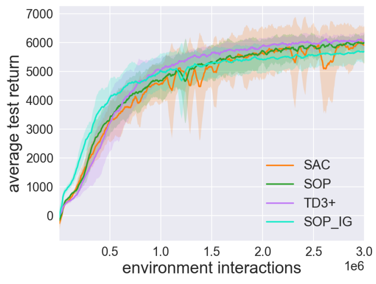
(a) Hopper-v2



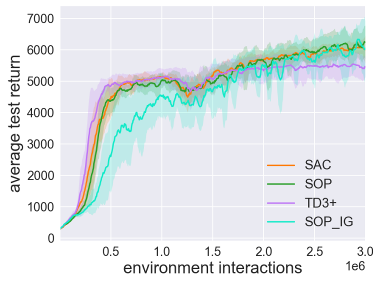
(b) Walker2d-v2



(c) HalfCheetah-v2



(d) Ant-v2



(e) Humanoid-v2

Figure 3.4: Streamlined Off-Policy (SOP) versus SAC, SOP_IG and TD3

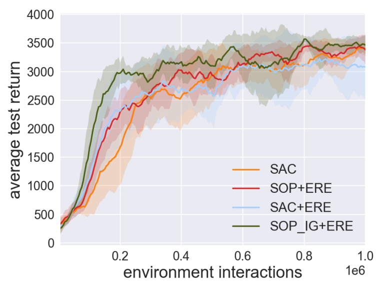
achieving a new SOTA. Among the three algorithms, SOP+ERE gives the best performance for Ant and Humanoid (the two most challenging environments) and performance roughly equivalent to SAC+ERE and SOP_IG+ERE for the other three environments.

In particular, for Ant and Humanoid, SOP+ERE improves performance by 21% and 24% over SAC at 1 million samples, respectively. For Humanoid, at 3 million samples, SOP+ERE improves performance by 15%. In conclusion, SOP+ERE is not only a simple algorithm, but also exceeds state-of-the-art performance.

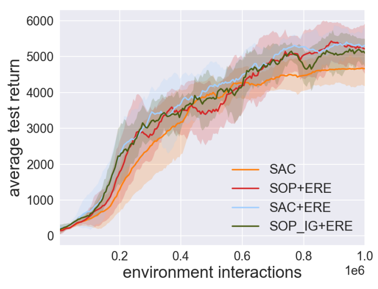
3.4.3 Ablation Study

In this section, we first separately examine the importance of (i) the normalization at the output of the policy network; (ii) the double Q networks; (iii) and randomization used in the line 9 of the SOP algorithm (that is, target policy smoothing [Fujimoto et al. 2018b]).

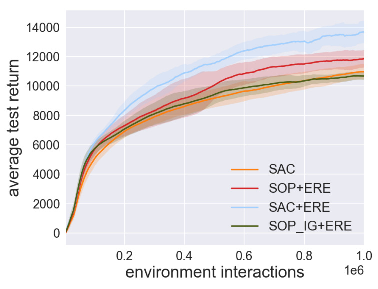
Figure 3.6 shows the results for the five environments considered in this chapter. In Figure



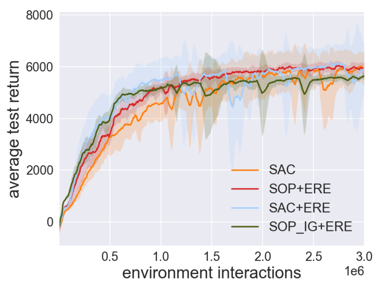
(a) Hopper-v2



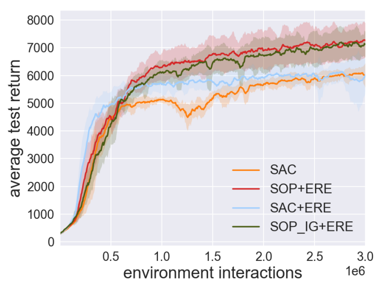
(b) Walker2d-v2



(c) HalfCheetah-v2



(d) Ant-v2



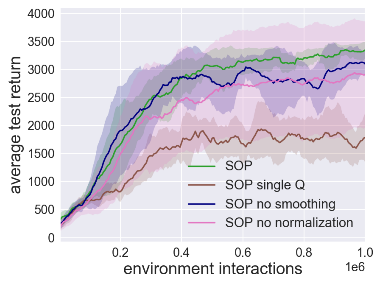
(e) Humanoid-v2

Figure 3.5: (a) to (e) show the performance of SAC baseline, SOP+ERE, SAC+ERE, and SOP_IG+ERE.

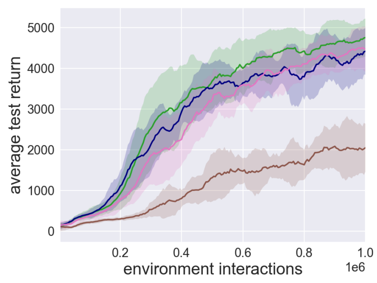
3.6, “no normalization” is SOP without the normalization of the outputs of the policy network; “single Q” is SOP with one Q-network instead of two; and “no smoothing” is SOP without the randomness in line 8 of the algorithm.

Figure 3.6 confirms that double Q-networks are critical for obtaining good performance [Van Hasselt et al. 2016; Fujimoto et al. 2018b; Haarnoja et al. 2018a]. Figure 3.6 also shows that output normalization is critical. Without output normalization, performance fluctuates wildly, and average performance can decrease dramatically, particularly for Humanoid and HalfCheetah. Target policy smoothing improves performance by a relatively small amount.

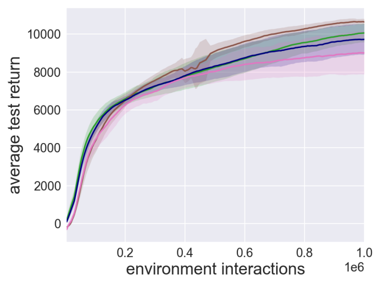
In addition, to better understand whether the simple normalization term in SOP achieves a similar effect compared to explicitly maximizing entropy, we plot the entropy values for SOP and SAC throughout training for all environments. We found that SOP and SAC have very similar entropy values across training, while removing the entropy term from SAC makes the entropy value much lower, as shown in figure 3.7. This indicates that the effect of the action normalization is very similar to maximizing entropy.



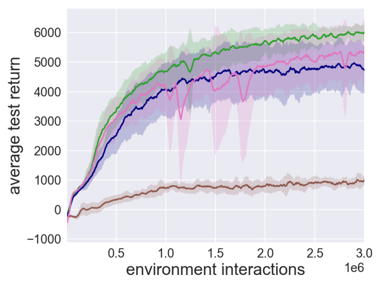
(a) Hopper-v2



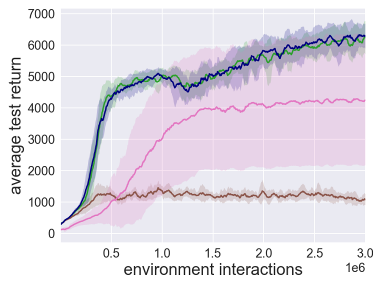
(b) Walker2d-v2



(c) HalfCheetah-v2

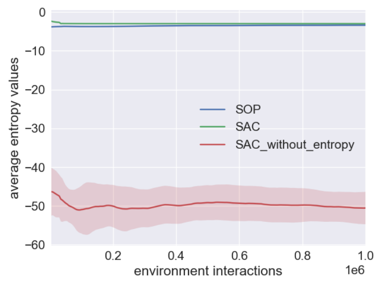


(d) Ant-v2

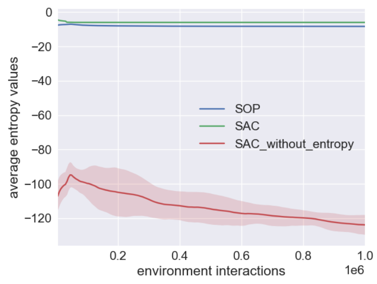


(e) Humanoid-v2

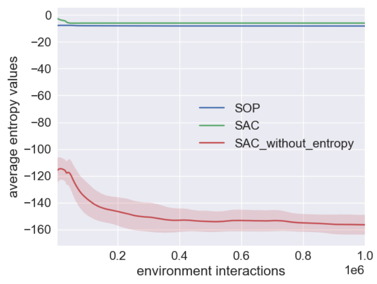
Figure 3.6: Ablation Study for SOP



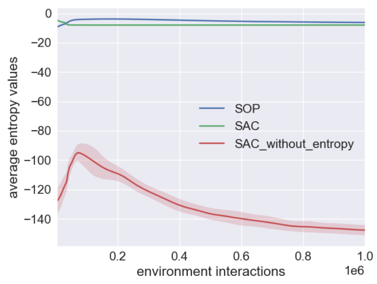
(a) Hopper-v2



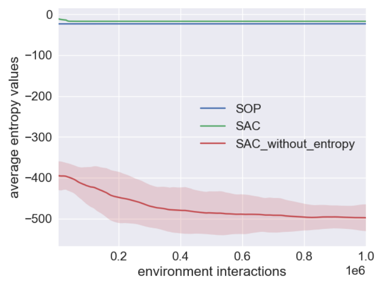
(b) Walker2d-v2



(c) HalfCheetah-v2



(d) Ant-v2

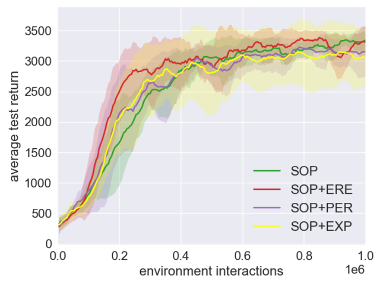


(e) Humanoid-v2

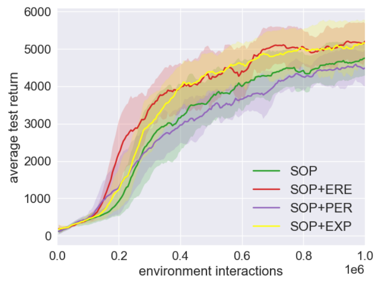
Figure 3.7: Entropy value comparison between SOP, SAC, and SAC without entropy maximization

Finally, we investigate the effect of Prioritized Experience Replay (PER) [Schaul et al. 2015] and Exponential Sampling (EXP) with SOP on the MuJoCo benchmark.

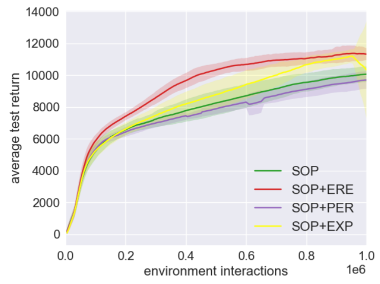
Figure 3.8 shows a performance comparison of SOP, SOP+ERE, SOP+EXP and SOP+PER. Results show that the exponential sampling scheme gives a boost to the performance of SOP, and especially in the Humanoid environment, although not as good as ERE. Surprisingly, SOP+PER does not give a significant performance boost to SOP (if any boost at all). We also found that it is difficult to find hyperparameter settings for SOP+PER that work well for all environments. Some of the other hyperparameter settings actually reduce performance. It is unclear why PER does not work so well for SOP. A similar result has been found in another recent paper [Fu et al. 2019], showing that PER can significantly reduce performance on TD3. Further research is needed to understand how PER can be successfully adapted to environments with continuous action spaces and dense reward structure.



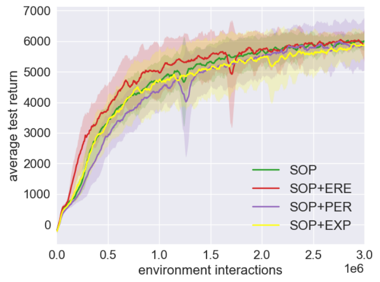
(a) Hopper-v2



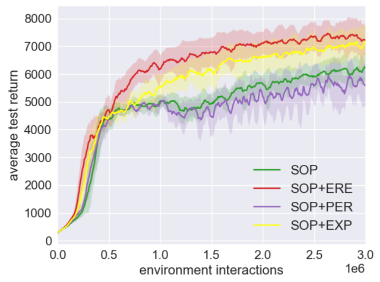
(b) Walker2d-v2



(c) Halfcheetah-v2



(d) Ant-v2



(e) Humanoid-v2

Figure 3.8: Streamlined Off-Policy (SOP), with ERE and PER sampling schemes

3.4.4 Implementation Details

All experiments are run on cpu nodes only. Each job runs on a single Intel(R) Xeon(R) CPU E5-2620 v3 with 2.40GHz. In table 3.1, we show hyperparameters used for SOP, SOP+ERE and SOP+PER. For adaptive SAC, we use our own PyTorch implementation for the comparisons. Our implementation uses the same hyperparameters as used in the original paper [Haarnoja et al. 2018b]. Our implementation of SOP variants and adaptive SAC share most of the code base. For TD3, our implementation uses the same hyperparamters as used in the authors' implementation, which is different from the ones in the original paper [Fujimoto et al. 2018b]. They claimed that the new set of hyperparamters can improve performance for TD3. We now discuss hyperparameter search for better clarity, fairness and reproducibility [Henderson et al. 2018; Duan et al. 2016; Islam et al. 2017].

For the η value in the ERE scheme, in our early experiments we tried the values (0.993, 0.994, 0.995, 0.996, 0.997, 0.998) on the Ant and found 0.995 to work well. This initial range of values was decided by computing the ERE sampling range for the oldest data. We found that for smaller values, the range would simply be too small. For the PER scheme, we did some informal preliminary search, then searched on Ant for β_1 in (0, 0.4, 0.6, 0.8), β_2 in (0, 0.4, 0.5, 0.6, 1), and learning rate in (1e-4, 2e-4, 3e-4, 5e-4, 8e-4, 1e-3), we decided to search these values because the original paper used $\beta_1 = 0.6$, $\beta_2 = 0.4$ and with reduced learning rate. For the exponential sampling scheme, we searched the λ value in (3e-7, 1e-6, 3e-6, 5e-6, 1e-5, 3e-5, 5e-5, 1e-4) in Ant, this search range was decided by plotting out the probabilities of sampling, and then pick a set of values that are not too extreme. For σ in SOP, in some of our early experiments with SAC, we accidentally found that $\sigma = 0.3$ gives good performance for SAC without entropy and with Gaussian noise. We searched values (0.27, 0.28, 0.29, 0.3). For σ values for TD3+, we searched values (0.1, 0.15, 0.2, 0.25, 0.3).

In the ERE scheme, the sampling range always starts with the entire buffer (1M data) and then gradually shrinks. This is true even when the buffer is not full. So even if there are not many data

Table 3.1: SOP Hyperparameters

Parameter	Value
Shared	
optimizer	Adam [Kingma and Ba 2014]
learning rate	$3 \cdot 10^{-4}$
discount (γ)	0.99
target smoothing coefficient (ρ)	0.005
target update interval	1
replay buffer size	10^6
number of hidden layers for all networks	2
number of hidden units per layer	256
mini-batch size	256
nonlinearity	ReLU
SAC adaptive	
entropy target	-dim($\mathcal{A}$) (e.g., 6 for HalfCheetah-v2)
SOP	
gaussian noise std $\sigma = \sigma_1 = \sigma_2$	0.29
TD3	
gaussian noise std for data collection σ	0.1 * action limit
guassian noise std for target policy smoothing $\tilde{\sigma}$	0.2
TD3+	
gaussian noise std for data collection σ	0.15
guassian noise std for target policy smoothing $\tilde{\sigma}$	0.2
ERE	
ERE initial η_0	0.995
PER	
PER β_1 (α in PER paper)	0.4
PER β_2 (β in PER paper)	0.4
EXP	
Exponential λ	$5e-06$

points in the buffer, we compute c_k based as if there are 1M data points in the buffer. One can also modify the design slightly to obtain a variant that uses the current amount of data points to compute c_k. In addition to the reported scheme, we also tried shrinking the sampling range linearly, but it gives less performance gain.

In our implementation we set the number of updates after an episode to be the same as the number of timesteps in that episode. Since environments do not always end at 1000 timesteps, we can give a more general formula for c_k. Let K be the number of mini-batch updates, let N be the max size of the replay buffer, then:

$$c_k = N \cdot \eta^{k \frac{1000}{K}} \tag{3.7}$$

With this formulation, the range of sampling shrinks in more or less the same way with varying number of mini-batch updates. We always do uniform sampling in the first update, and we always have $\eta^{K \frac{1000}{K}} = \eta^{1000}$ in the last update.

When η is small, c_k can also become small for some of the mini-batches. To prevent getting a mini-batch with too many repeating data points, we set the minimum value for c_k to 5000. We did not find this value to be too important and did not find the need to tune it. It also does not have any effect for any $\eta \geq 0.995$ since the sampling range cannot be lower than 6000.

In the adaptive scheme with buffer of size 1M, the recent performance improvement is computed as the difference of the current episode return compared to the episode return 500,000 timesteps earlier. Before we reach 500,000 timesteps, we simply use η_0. The exact way of computing performance improvement does not have a significant effect on performance as long as it is reasonable.

3.5 Related Work

In recent years, there has been significant progress in improving the sample efficiency of DRL for continuous robotic locomotion tasks with off-policy algorithms [Lillicrap et al. 2019; Fujimoto et al. 2018b; Haarnoja et al. 2018a,b]. There is also a significant body of research on maximum entropy RL methods [Ziebart et al. 2008; Ziebart 2010; Todorov 2008; Rawlik et al. 2013; Levine and Koltun 2013; Levine et al. 2016; Nachum et al. 2017; Haarnoja et al. 2017, 2018a,b]. Ahmed et al. [2019] very recently shed light on how entropy leads to a smoother optimization landscape. By taking clipping in the MuJoCo environments explicitly into account, Fujita and Maeda [2018] modified the policy gradient algorithm to reduce variance and provide superior performance among on-policy algorithms. Eisenach et al. [2018] extend the work of Fujita and Maeda [2018] for when an action may be direction. Hausknecht and Stone [2015] introduce Inverting Gradients, for which we provide expermintal results in this chapter for the MuJoCo environments. Chou et al. [2017] also explores DRL in the context of bounded action spaces. Dalal et al. [2018] consider safe exploration in the context of constrained action spaces.

Experience replay [Lin 1992] is a simple yet powerful method for enhancing the performance of an off-policy DRL algorithm. Experience replay stores past experience in a replay buffer and reuses this past data when making updates. It achieved great successes in Deep Q-Networks (DQN) [Mnih et al. 2013, 2015].

Uniform sampling is the most common way to sample from a replay buffer. One of the most well-known alternatives is prioritized experience replay (PER) [Schaul et al. 2015]. PER uses the absolute TD-error of a data point as the measure for priority, and data points with higher priority will have a higher chance of being sampled. This method has been tested on DQN [Mnih et al. 2015] and double DQN (DDQN) [Van Hasselt et al. 2016] with significant improvement and applied successfully in other algorithms [Wang et al. 2015; Schulze and Schulze 2018; Hessel et al. 2018; Hou et al. 2017] and can be implemented in a distributed manner [Horgan et al. 2018].

When new data points lead to large TD errors in the Q update, PER will also assign high sampling probability to newer data points. However, PER has a different effect compared to ERE. PER tries to fit well on both old and new data points. While for ERE, old data points are always considered less important than newer data points even if these old data points start to give a high TD error. A performance comparison of PER and ERE are given in the supplementary materials.

There are other methods proposed to make better use of the replay buffer. The ACER algorithm has an on-policy part and an off-policy part, with a hyper-parameter controlling the ratio of off-policy to on-policy updates [Wang et al. 2016]. The RACER algorithm [Novati and Koumoutsakos 2018] selectively removes data points from the buffer, based on the degree of "off-policyness," bringing improvement to DDPG [Lillicrap et al. 2019], NAF [Gu et al. 2016] and PPO [Schulman et al. 2017]. In De Bruin et al. [2015], replay buffers of different sizes were tested, showing large buffer with data diversity can lead to better performance. Finally, with Hindsight Experience Replay[Andrychowicz et al. 2017], priority can be given to trajectories with lower density estimation [Zhao and Tresp 2019] to tackle multi-goal, sparse reward environments.

3.6 Conclusion

In this chapter, we first showed that the primary role of maximum entropy RL for the MuJoCo benchmark is to maintain satisfactory exploration in the presence of bounded action spaces. With the insight, we then developed a new streamlined algorithm which does not employ entropy maximization but nevertheless matches the sampling efficiency and robust performance of SAC for the MuJoCo benchmarks. Finally, we combined our streamlined algorithm with a simple non-uniform sampling scheme to create a simple algorithm that further improves sample efficiency and achieves state-of-the art performance for the MuJoCo benchmark.

4 | Aggressive Q-learning with Ensembles

4.1 Introduction

In the last chapter, we have discussed a number of off-policy Deep RL algorithms for control tasks with continuous state and action spaces, including Deep Deterministic Policy Gradient (DDPG), Twin Delayed DDPG (TD3) and Soft Actor Critic (SAC) [Lillicrap et al. 2019; Fujimoto et al. 2018b; Haarnoja et al. 2018a,b]. Techniques such as reusing past experience from a replay buffer, clipped double-Q learning, maximum entropy and output normalization proposed by these off-policy Deep RL algorithms are proved to provide excellent sample efficiency and asymptotic performance in a wide-range of MuJoCo environments.

More recently, Kuznetsov et al. [2020] proposed Truncated Quantile Critics (TQC), a model-free algorithm which includes distributional representations of critics, truncation of critics prediction, and ensembling of multiple critics. Instead of the usual modeling of the Q-function of the expectation of return, TQC approximates the distribution of the return random variable conditioned on the state and action. By dropping several of the top-most "atoms" and varying the number of dropped atoms of the return distribution approximation, TQC can control the overestimation bias. TQC's asymptotic performance (that is after a long period of training) was shown to be better than that of SAC on the continuous control MuJoCo benchmark suite, including a 25%

improvement on the most challenging Humanoid environment. However, TQC is not sample efficient in that it generally requires a large number of samples to reach even moderate performance levels.

Chen et al. [2021] proposed Randomized Ensembled Double Q-learning(REDQ), a model-free algorithm which includes a high Update-To-Data (UTD) ratio, an ensemble of Q functions, and in-target minimization across a random subset of Q functions from the ensemble. Using a UTD ratio much larger than one, meaning that several gradient steps are taken for each environment interaction, improves sample efficiency, while the ensemble and in-target minimization allows the algorithm to maintain stable and near-uniform bias under the high UTD ratio. The algorithm was shown to attain much better performance than SAC at the early stage of training, and to match or improve the sample-efficiency of the state-of-the-art model-based algorithms for the MuJoCo benchmarks. However, although REDQ is highly sample efficient for early-stage training, its asymptotic performance is significantly below that of TQC.

Is it possible to design a simple, streamlined model-free algorithm which can achieve REDQ's high sample efficiency in early-stage training and also achieve TQC's high asymptotic performance in late state training? In this chapter, we achieve this goal with a new model-free algorithm, Aggressive Q-Learning with Ensembles (AQE). Like TQC and REDQ, AQE uses an ensemble of Q-functions, and like REDQ it uses a UTD ratio > 1. However AQE is very simple, requiring neither distributional representation of critics as in TQC nor target randomization and double-Q learning as in REDQ. AQE controls over estimation bias and the standard deviation of the bias by varying the number of ensemble members N and the number of ensembles $K \leq N$ that are kept when calculating the targets.

Through carefully designed experiments, we provide a detailed analysis of AQE. We perform extensive and comprehensive experiments for both MuJoCo and DeepMind Control Suite (DMC) environments. We first show that for the five most challenging MuJoCo benchmark, AQE provides state-of-the-art performance, surpassing the performance of SAC, REDQ, and TQC at *all* *stages of training*. When averaged across the five MuJoCo environments, AQE's early stage performance is 2.9 times better than SAC, 1.6 times better than TQC and 1.1 times better than REDQ. AQE's asymptotic performance is 26%, 22%, and 6% higher than SAC, REDQ, and TQC, respectively. Then we provide additional experimental results for the nine most challenging DeepMind Control Suite (DMC) environments, which TQC and REDQ did not consider. We show that AQE also provides state-of-the-art performance at both early stage and late-stage of training. When averaged over nine environments, AQE's early stage performance is 13.71 times better than SAC, 7.59 times better than TQC and 1.02 times better than REDQ. AQE's asymptotic performance is 37% better than SAC, 3% better than REDQ, and 8% better than TQC. We also perform an ablation study, and show that AQE is robust to choices of hyperparameters: AQE can work well with small ensembles consisting of 10-20 ensemble members, and performance does not vary significantly with small changes in the keep parameter K. We show that that AQE performs better than several variations, including using the median of all ensemble members and removing the most extreme minimum and maximum outlier in the targets. In order to improve computational time, we also consider different multi-head architectures for the ensemble of critics: consistent with the supervised convolutional network literature, we find that a two-head architecture not only reduces computational time but can actually improve performance for some environments. Additionally, we show that AQE continues to out-perform SAC and TQC even when these algorithms are made aggressive with a UTD $\gg 1$.

4.2 Related Work

Overestimation bias due to in target maximization in Q-learning can significantly slow learning [Thrun and Schwartz 1993]. For tabular Q-learning, Hasselt [2010] introduced Double Q-Learning, and showed that it removes the overestimation basis and in general leads to an underestimation bias. Van Hasselt et al. [2016] showed that adding Double Q-learning to deep-Q networks can have a similar effect, leading to a major performance boost for the Atari games benchmark. As stated in the Introduction, for continuous-action spaces, TD3 and SAC address the overestimation bias using clipped-double Q-learning, which brings significant performance improvements [Fujimoto et al. 2018b; Haarnoja et al. 2018a,b].

As mentioned in the Introduction, Kuznetsov et al. [2020] control the over-estimation bias by estimating the distribution of the return random variable, and then by dropping several of the top-most "atoms" from the estimated distribution. The distribution estimate is based on a methodology developed in Bellemare et al. [2017]; Dabney et al. [2018a,b], which employs an asymmetric Huber loss function to minimize the Wasserstein distance between the neural network output distribution and the target distribution. In this chapter, in order to counter over-estimation bias, we also drop the top-most estimates, although we do so solely with an ensemble of Q-function mean estimators rather than with an ensemble of the more complex distributional models employed in [Kuznetsov et al. 2020].

It is well-known that using ensembles can improve the performance of DRL algorithms [Faußer and Schwenker 2015; Nachum et al. 2017; Lee et al. 2021]. For Q-learning based methods, Anschel et al. [2017] use the average of multiple Q estimates to reduce variance. Lan et al. [2020b] introduced Maxmin Q-learning, which uses the minimum of all the Q-estimates rather than the average. Agarwal et al. [2020] use Random Ensemble Mixture (REM), which employs a random convex combination of multiple Q estimates.

Model-based methods often attain high sample efficiency by using a high UTD ratio. In particular, Model-Based Policy Optimization (MBPO) [Janner et al. 2019] uses a large UTD ratio of 20-40. Compared to Soft-Actor-Critic (SAC), which is model-free and uses a UTD of 1, MBPO achieves much higher sample efficiency in the OpenAI MuJoCo benchmark [Todorov et al. 2012; Brockman et al. 2016]. REDQ [Chen et al. 2021], a model-free algorithm, also successfully employs a high UTD ratio to achieve high sample efficiency.

4.3 Aggressive Q-learning with Ensemble

In this section, we introduce details of our proposed algorithm, Aggressive Q-learning with Ensembles (AQE), a simple model-free algorithm which provides state-of-the-art performance for both MuJoCo and DeepMind Control Suite (DMC) benchmark for both early and late stage of training. The pseudo-code is shown in Algorithm 3.

4.3.1 Architecture

As is the case with most off-policy continuous-control algorithms, AQE has a single actor (policy network) and multiple critics (Q-function networks), and employs Polyak averaging of the target parameters to enhance stability. Building on this algorithmic base, it also employs an update-to-data ratio $G > 1$, an ensemble of $N \geq 3$ Q-functions (rather than just two as in TD3 and SAC), and targets that average all the Q-functions excluding the Q-functions with the highest $N - K$ values. As discussed in section 4.3.2, we demonstrate theoretically in the tabular case of the algorithm that we can control over-estimation through adjusting K and N. More concretely, we can bring the bias term from above zero (i.e. overestimation) to under zero (i.e. underestimation) by decreasing K and/or increasing N. In contrast, REDQ employs two randomly chosen ensemble members when calculating the target, the bias does not depend on the number of ensemble models N [Chen et al. 2021]. Unlike REDQ, AQE can control the bias term through both the number of ensemble models used in the average calculation K and the total number of ensembles N, allowing for more flexibility. One other drawback for REDQ is that it ignores the estimates of all other ensemble estimates except for the minimal one in the randomly chosen set, which diminishes the power of the multiple ensemble sets. In contrast, AQE utilizes most of the ensemble models when calculating the target.The resulting algorithm is not only simple and streamlined, but also provides state-of-the art performance. For exploration, it uses entropy maximization as in SAC, although it could easily incorporate alternative exploration schemes.

Algorithm 3 Aggressive Q-Learning with Ensembles

```
1: Initial policy parameters $\theta$, $N$ Q-function parameters $\phi_i$, $i = 1,\ldots,N$, empty replay buffer $\mathcal{D}$.
   Set target parameters $\phi_{\text{targ},i} \leftarrow \phi_i$ for $i = 1, 2,\ldots, N$.
2: repeat
3:   Take one action $a_t \sim \pi_\theta(\cdot|s_t)$. Observe reward $r_t$, new state $s_{t+1}$.
4:   Add data to replay buffer: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r_t, s_{t+1})\}$
5:   for $G$ updates do
6:     Randomly sample a mini-batch $B = \{(s, a, r, s')\}$ from $\mathcal{D}$.
7:     for each $(s, a, r, s') \in B$ do
8:       Sample $\tilde{a}' \sim \pi_\theta(\cdot|s')$.
9:       Determine the $K$ indices from $i = 1, \ldots, N$ that minimize $Q_{\text{target},i}(s', \tilde{a}')$.
10:      Compute the Q target $y$:
           $$y(s,a) = r + \gamma\left(\frac{1}{K}\sum_{i\in K} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right)$$
11:    end for
12:    for $i = 1, \ldots, N$ do
13:      Update $\phi_i$ with gradient descent using
           $$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s')\in B} \left(Q_{\phi_i}(s,a) - y(s,a)\right)^2$$
14:      Update target networks with $\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i$
15:    end for
16:  end for
17:  Update policy parameters $\theta$ with gradient ascent using
       $$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} \left(\frac{1}{N}\sum_{i=1}^{N} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s)\right) \qquad \tilde{a}_\theta(s) \sim \pi_\theta(\cdot|s)$$
18: until Convergence
```

AQE has three key hyperparameters, G, N, and K. If we set $N = 2$, $K = 1$ and $G = 1$, AQE is simply the underlying off-policy algorithm such as SAC. When $N > 2$, $K = 1$ and $G = 1$, then AQE becomes similar to, but not equivalent to, Maxmin Q-learning [Lan et al. 2020b].

AQE uses an ensemble of Q networks (as does REDQ and TQC). Employing multiple networks, one for each Q-function output, can be expensive in terms of computation and memory. In order to reduce the computation and memory requirements, we also consider multi-head architectures for generating multiple Q-function outputs. Instead of each network providing a single Q-estimate output, we consider N separate Q networks each with h heads, providing a total of $h \cdot N$ estimates. The h heads from one network share all of the layers except the final fully-connected layer. In practice, we have found $h = 2$ heads works well for AQE, consistent with work in ensembles of convolutional neural networks for computer vision tasks [Lee et al. 2015]. When properly sharing low level weights, multi-headed networks may not only retain the performance of full ensembles, but can sometimes outperform them. We conduct ablation studies on the multi-head architecture in AQE in Section 4.4.3.

4.3.2 Theoretical Analysis

In this section, we characterize how changing the size of the ensemble N and the keep parameter K affects the estimation bias term in the AQE algorithm. We will restrict our analysis to the tabular version of AQE in algorithm 4.

Algorithm 4 Tabular AQE

```
1: **Initialize:** $Q^j(s,a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, $j = 1, \dots, N$.
2: **repeat**
3:     For some state $s \in \mathcal{S}$, choose $a \in \mathcal{A}$ based on $\left\{Q^j(s,a)\right\}_{j=1}^{N}$, observe $r$, $s'$.
4:     For each $a' \in \mathcal{A}$, let $E_{K,N}(s',a')$ be the ensemble members in $\{1, \dots, N\}$ with the $K$ lowest
       values of $Q^j(s',a')$, $j = 1, \dots, N$.
5:     Get target
           $$y = r + \gamma \max_{a' \in \mathcal{A}} \frac{1}{K} \sum_{j \in E_{K,N}(s',a')} Q^j(s',a')$$
6:     **for** $j = 1, \dots, N$ **do**
7:         Update each $Q^j(s,a)$
               $$Q^j(s,a) \leftarrow Q^j(s,a) + \alpha(y - Q^j(s,a))$$
8:     **end for**
9:     $s \leftarrow s'$
10: **until** end
```

Our analysis will follow similar lines of reasoning as Lan et al. [2020b] and Chen et al. [2021] which extends upon the theoretical framework introduced in Thrun and Schwartz [1993].

For each $a \in \mathcal{A}$, let $E_{K,N}(s,a)$ be the ensemble members in $\{1, \dots, N\}$ with the K lowest values of $Q^j(s,a)$, $j = 1, \dots, N$. In the tabular case, the target for the Q networks take the form:

$$r + \gamma \max_{a'} \left(\frac{1}{K} \sum_{j \in E_{K,N}(s',a')} Q^j(s',a') \right). \tag{4.1}$$

Define the *post-update estimation bias* as

$$\begin{aligned} Z_{K,N} &:= r + \gamma \max_{a'} \left(\frac{1}{K} \sum_{j \in E_{K,N}(s',a')} Q^j(s',a') \right) - \left(r + \gamma \max_{a'} Q^\pi(s',a') \right) \\ &= \gamma \left[\max_{a'} \left(\frac{1}{K} \sum_{j \in E_{K,N}(s',a')} Q^j(s',a') \right) - \max_{a'} Q^\pi(s',a') \right] \end{aligned} \tag{4.2}$$

Under this definition, if $\mathbb{E}[Z_{K,N}] > 0$, then the expected post-update estimation bias is positive and there is a tendency for the positive bias to accumulate during updates. Similarly, if $\mathbb{E}[Z_{K,N}] < 0$, then the expected post-update estimation bias is negative and there is a tendency for the negative bias to accumulate during updates. Ideally, we would like $\mathbb{E}[Z_{K,N}] \approx 0$

Also let

$$Q^j(s,a) = Q^\pi(s,a) + e^j(s,a) \tag{4.3}$$

where $e^j(s,a)$ is an independent and identically distributed error term across all j's and all a's for each fixed s. We further assume that $\mathbb{E}[e^j(s,a)] = 0$. Note that with this assumption

$$\mathbb{E}\left[\frac{1}{N} \sum_{j=1}^{N} Q^j(s,a) \right] - Q^\pi(s,a) = 0,$$

that is the pre-update estimation bias is zero. The following theorem shows how the expected estimation bias changes with N and K:

Theorem 4.1. *The following results hold for* $\mathbb{E}[Z_{K,N}]$*:*

1. $\mathbb{E}[Z_{N,N}] \geq 0$ *for all* $N \geq 1$.

2. $\mathbb{E}[Z_{K-1,N}] \leq \mathbb{E}[Z_{K,N}]$ *for all* $K \leq N$.

3. $\mathbb{E}[Z_{K,N+1}] \leq \mathbb{E}[Z_{K,N}]$.

4. *Suppose that* $e_{sa}^{j} \leq c$ *for some* $c > 0$ *for all* s, a *and* j. *Then there exists an* N *sufficiently large and* $K < N$ *such that* $\mathbb{E}[Z_{K,N}] < 0$.

Proof Sketch. Part 1 is a result of Jensen's Inequality, and Parts 2 and 3 can be shown by analyzing how the average of the K smallest ensembles changes when adding an extra ensemble model. Given the first three parts, we only need to show that $\mathbb{E}[Z_{1,N}] < 0$ to show that there exists a K for a sufficiently large N where the expected bias is negative. See below for full proof. □

We first present the following lemma:

Lemma 4.2 (Chen et al. 2021). *Let* $X_1, X_2, \ldots$ *be an infinite sequence of i.i.d. random variables with cdf* $F(x)$ *and let* $\tau = \inf x : F(x) > 0$. *Also let* $Y_N = \min\{X_1, X_2, \ldots, X_N\}$. *Then* $Y_1, Y_2, \ldots$ *converges to* τ *almost surely.*

Proof. See Appendix A.2 of Chen et al. [2021] □

Theorem 4.1. *The following results hold for* $\mathbb{E}[Z_{K,N}]$:

1. $\mathbb{E}[Z_{N,N}] \geq 0$ *for all* $N \geq 1$.

2. $\mathbb{E}[Z_{K-1,N}] \leq \mathbb{E}[Z_{K,N}]$ *for all* $K \leq N$.

3. $\mathbb{E}[Z_{K,N+1}] \leq \mathbb{E}[Z_{K,N}]$.

4. *Suppose that* $e_{sa}^{j} \leq c$ *for some* $c > 0$ *for all* s, a *and* j. *Then there exists an* N *sufficiently large and* $K < N$ *such that* $\mathbb{E}[Z_{K,N}] < 0$.

Proof. 1. By definition,

$$\begin{aligned}\mathbb{E}[Z_{N,N}] &= \gamma\, \mathbb{E}\left[\max_{a'}\left(\frac{1}{N}\sum_{j=1}^{N} Q^j(s',a')\right) - \max_{a'} Q^{\pi}(s',a')\right] \\ &\geq \gamma \left[\max_{a'} E\left[\left(\frac{1}{N}\sum_{j=1}^{N} Q^j(s',a')\right)\right] - \max_{a'} Q^{\pi}(s',a')\right] \\ &= \gamma \left[\max_{a'} Q^{\pi}(s',a') - \max_{a'} Q^{\pi}(s',a')\right] = 0\end{aligned} \tag{4.4}$$

2. Let

$$\bar{Q}_{K,N}(s,a) = \frac{1}{K}\sum_{j\in E_{K,N}} Q^j(s,a). \tag{4.5}$$

Since for any state s, $\max_a \bar{Q}_{K+1,N}(s,a) \geq \max_a \bar{Q}_{K,N}(s,a)$,

$$\begin{aligned}\mathbb{E}[Z_{K+1,N}] &= \gamma\, \mathbb{E}\left[\max_{a'} \bar{Q}_{K+1,N}(s',a') - \max_{a'} Q^{\pi}(s',a')\right] \\ &\geq \gamma\, \mathbb{E}\left[\max_{a'} \bar{Q}_{K,N}(s',a') - \max_{a'} Q^{\pi}(s',a')\right] \\ &= \mathbb{E}[Z_{K,N}]\end{aligned} \tag{4.6}$$

3. Comparing $\mathbb{E}[Z_{K,N}]$ and $\mathbb{E}[Z_{K,N+1}]$ is equivalent to comparing $\bar{Q}_{K,N}(s,a)$ and $\bar{Q}_{K,N+1}(s,a)$. Since $e^j(s,a)$ is i.i.d., by extension $Q^j(s,a)$ is also i.i.d. for $j = 1, 2, \cdots$. Suppose $Q^j(s,a)$ is drawn from some probability distribution F, then given $\bar{Q}_{K,N}(s,a)$, $\bar{Q}_{K,N+1}(s,a)$ can be calculated by generating an additional $Q^i(s,a)$ from F. The new sample $Q^i(s,a)$ affects the calculation of $\bar{Q}_{K,N+1}(s,a)$ under the following two cases:

   - If $Q^i(s,a) > \max_{j\in E_{K,N}} Q^j(s,a)$, then the lowest K values remain unchanged hence $\bar{Q}_{K,N}(s,a) = \bar{Q}_{K,N+1}(s,a)$.
   - Else if $Q^i(s,a) \leq \max_{j\in E_{K,N}} Q^j(s,a)$, then $\max_{j\in E_{K,N}} Q^j(s,a)$ would be removed from and $Q^i(s,a)$ would be added to the set of lowest K values, therefore $\bar{Q}_{K,N+1}(s,a) \leq \bar{Q}_{K,N}(s,a)$.

Combining the two cases $\bar{Q}_{K,N+1}(s,a) \leq \bar{Q}_{K,N}(s,a)$, therefore $\mathbb{E}[Z_{K,N+1}] \leq \mathbb{E}[Z_{K,N}]$

4. Since $\mathbb{E}[Z_{N,N}] \geq 0$, $\mathbb{E}[Z_{K,N}] \leq \mathbb{E}[Z_{K+1,N}]$ and $\mathbb{E}[Z_{K,N+1}] \leq \mathbb{E}[Z_{K,N}]$. It is suffice to show that $\mathbb{E}[Z_{1,N}] < 0$ for some N. The rest of the proof largely follows Theorem 1 of Chen et al. [2021].

   Let $\tau = \inf\{x : F_a(x) > 0\}$ where $F_a(x)$ is the cdf of $Q^j(s,a)$, $j = 1, 2, \ldots$. By Lemma 1, $\bar{Q}_{1,N}(s,a) = \min_{1 \leq j \leq N} Q^j(s,a)$ converges almost surely to to τ_a for each a. Since the action space is finite, it then follows that $\max_a \bar{Q}_{1,N}(s,a)$ converges almost surely to to $\tau = \max_a \tau_a$. Due to our assumption that $e^j(s,a) \leq c$ and that $Q^\pi(s,a)$ is finite, it then follows that $\max_a \bar{Q}_{1,N}(s,a)$ is also bounded above. By Part 3 of the theorem, $\bar{Q}_{1,N}(s,a)$ is monotonoically decreasing w.r.t. N. and since $\max_a \bar{Q}_{1,N}(s,a)$ is also bounded above and converges almost surely to τ, we have

$$\begin{aligned}\mathbb{E}[Z_{1,N}] &= \gamma\left(\mathbb{E}[\max_a \min_{1\leq j\leq N} Q^j(s,a)] - \max_a Q^\pi(s,a)\right) \\ &= \gamma\left(\mathbb{E}[\max_a Y_a^N] - \max_a Q^\pi(s,a)\right) \overset{N\to\infty}{\longrightarrow} \gamma\left(\max_a \tau_a - \max_a Q^\pi(s,a)\right) < 0\end{aligned} \tag{4.7}$$

   where the last equality follows from the assumption that the error $e^j(s,a)$ is non-trivial, and hence $\tau_a < \max_a Q^\pi(s,a)$ for all a. Therefore for a sufficiently large N, there exists a $1 \leq K \leq N$ such that $\mathbb{E}_{K,N} < 0$.

□

Theorem 4.1 shows that we can control the expected post-update bias $\mathbb{E}[Z_{K,N}]$ through adjusting K and N. More concretely, we can bring the bias term from above zero (i.e. over estimation) to under zero (i.e. under estimation) by decreasing K and/or increasing N.

We note also that similar to Chen et al. [2021], we make very few assumptions on the error term $e_{s,a}$. This is in contrary to Thrun and Schwartz [1993] and Lan et al. [2020b], both of whom assume that the error term is uniformly distributed.

In REDQ, a random subset of ensemble models of size M is chosen and for any fixed M, the bias does not depend on the number of ensemble models N [Chen et al. 2021]. We can thus see from Figure 4.7 in Appendix, with $M = 2$ fixed, increasing the size of ensemble N with the multi-head architecture does not necessarily help the training of REDQ. Unlike REDQ, AQE can control the bias term through both the number of ensemble models used in the average calculation K and the total number of ensembles N, allowing for more flexibility.

4.4 Experiments

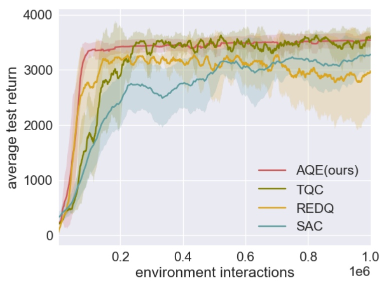
4.4.1 Results on MuJoCo

We provide experimental results in this section for AQE, TQC, REDQ and SAC for the five most challenging MuJoCo environments, namely, Hopper, Walker2d, HalfCheetah, Ant and Humanoid. To make a fair comparison, the TQC, REDQ and SAC results are reproduced using the authors' open source code, and use the same network sizes and hyperparameters reported in their papers. In particular, TQC employs 5 critic networks with 25 distributional samples for a total of 125 atoms. TQC drops 5 atoms per critic for Hopper, 0 atoms per critic for Half Cheetah, and 2 atoms per critic for Walker, Ant, and Humanoid. For REDQ, we also use the authors' suggested values of $N = 10$ and $M = 2$, where M is the number ensemble members used in the target calculation.

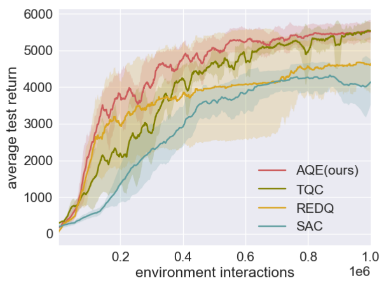
The REDQ paper uses $G = 20$ for the update-to-data ratio, and provides results for up to 300K environment interactions. Using such a high value for G is computationally infeasible in our experimental setting, since we use 3 million environment interactions for Ant and Humanoid in order to investigate asymptotic performance as well early-stage sample efficiency. In the experiments reported here, we use a value of $G = 5$ for both REDQ and AQE.

For AQE, we use 10 Q-networks each with 2 heads, producing 20 Q-values for each input. The AQE networks are the same size as those in the REDQ paper. AQE keeps 10 out of 20 values for

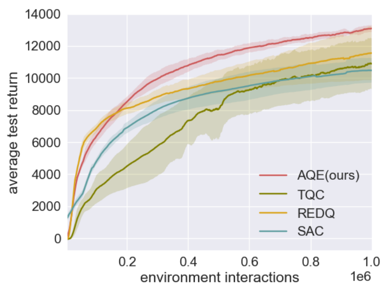
Hopper, all 20 values for half-Cheetah, and 16 out of 20 values for Walker, Ant and Humanoid.



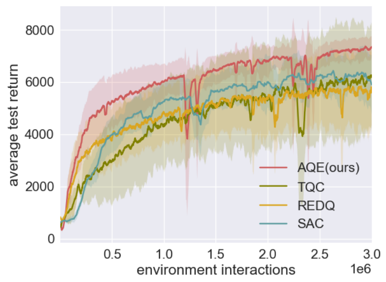
(a) Hopper-v2



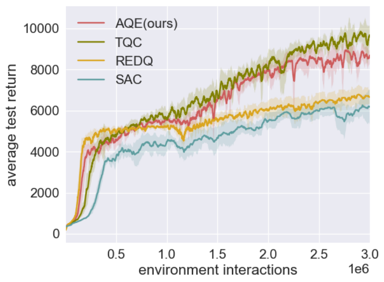
(b) Walker2d-v2



(c) HalfCheetah-v2



(d) Ant-v2



(e) Humanoid-v2

Figure 4.1: AQE versus TQC, REDQ and SAC. AQE is the only algorithm that beats SAC in all five environments during all stages of training, and it typically beats SAC by a wide margin.

Figure 4.1 shows the training curves for AQE, TQC, REDQ, and SAC. For each algorithm, we plot the average return of 5 independent trials as the solid curve, and plot the standard deviation across 5 seeds as the shaded region. For each environment, we train each algorithm for exactly the same number of environment interactions as done in the SAC paper. We use the same evaluation protocol as in the TQC paper. Specifically, after every epoch, we run ten test episodes with the current policy, record the undiscounted sum of all the rewards in each episode and take the average of the sums as the performance. A more detailed discussion on hyperparameters and implementation details is given in section 4.4.4.

We see from Figure 4.1 that AQE is the only algorithm that beats SAC in all five environments during all stages of training. Moreover, it typically beats SAC by a very wide margin. Table 4.1 shows that, when averaged across the five environments, AQE achieves SAC asymptotic performance approximately 3x faster than SAC and 2x faster than REDQ and TQC. As seen from Figure 4.1 and Table 4.2, in the early stages of training, AQE matches the excellent performance of REDQ in all five environments, and both algorithms are much more sample efficient than SAC and TQC. As seen from Figure 4.1 and Table 4.3, in late-stage training, AQE always matches or beats all other algorithms, except for Humanoid, where TQC is about 10% better. Table 4.3 shows that, when averaged across all five environments, AQE's asymptotic performance is 26%, 22%, and 6% higher than SAC, REDQ, and TQC, respectively.

Performance	SAC	TQC	REDQ	AQE	AQE/SAC	AQE/TQC	AQE/REDQ
Hopper at 3000	506K	184K	136K	77K	6.57	2.39	1.77
Walker2d at 4000	631K	371K	501K	277K	2.28	1.34	1.81
HalfCheetah at 10000	763K	737K	552K	304K	2.51	2.42	1.82
Ant at 5500	1445K	1759K	1749K	632K	2.29	2.78	2.77
Humanoid at 6000	2469K	1043K	1862K	1345K	1.84	0.78	1.38
Average	-	-	-	-	3.10	1.94	1.91

Table 4.1: Sample efficiency comparison of SAC, TQC, REDQ and AQE. The numbers show the amount of data collected when the specified performance level is reached (roughly corresponding to 90% of SAC's final performance). The last three columns show how many times AQE is more sample efficient than SAC, TQC and REDQ in reaching that performance level.

Amount of data	SAC	TQC	REDQ	AQE	AQE/SAC	AQE/TQC	AQE/REDQ
Hopper at 100K	1456	1807	2747	3345	2.30	1.85	1.22
Walker2d at 100K	501	1215	1810	2150	4.29	1.77	1.19
HalfCheetah at 100K	3055	4801	6876	6378	2.09	1.33	0.93
Ant at 250K	2107	2344	3279	4153	1.97	1.77	1.27
Humanoid at 250K	1094	3038	4535	3973	3.63	1.31	0.88
Average at early stage	-	-	-	-	2.86	1.61	1.10

Table 4.2: Early-stage performance comparison of SAC, TQC, REDQ and AQE. The numbers show the performance achieved when the specific amount of data is collected. On average, AQE performs 2.9 times better than SAC, 1.6 times better than TQC and 1.1 times better than REDQ.

Following the TQC paper, in Figure 4.1 we used different drop atoms for TQC for the different environments. To make the comparison fair, we also used different keep values K for AQE for the different environments. In Figure 4.2, we repeat the experiment on the five MuJoCo envi-

Amount of data	SAC	TQC	REDQ	AQE	AQE/SAC	AQE/TQC	AQE/REDQ
Hopper at 1M	3282	3612	2954	3541	1.08	0.98	1.20
Walker2d at 1M	4134	5532	4637	5517	1.33	1.00	1.19
HalfCheetah at 1M	10475	10887	11562	13093	1.25	1.20	1.13
Ant at 3M	5903	6186	5785	7345	1.24	1.19	1.27
Humanoid at 3M	6177	9593	6649	8680	1.41	0.91	1.31
Average at late stage	-	-	-	-	1.26	1.06	1.22

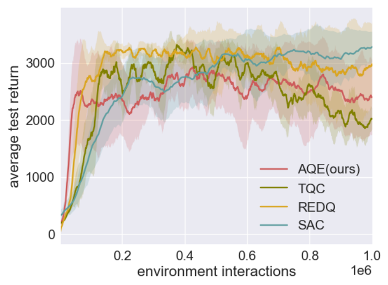
Table 4.3: Late-stage performance comparison of SAC, TQC, REDQ and AQE. The numbers show the performance achieved when the specific amount of data is collected. The last three columns show the ratio of AQE performance compared to SAC, TQC and REDQ performance. On average, during late-stage training, AQE performs 1.26 times better than SAC, 1.06 times better than TQC and 1.22 times better than REDQ.

ronments, but now use the same hyperparameter values across environments for TQC (drop two atoms per network) and AQE ($K = 16$). These choices of fixed hyper-parameters appear to give the best overall performance for the two algorithms. We report detailed early-stage and late-stage performance comparisons of all algorithms in Table 4.4 and Table 4.5.

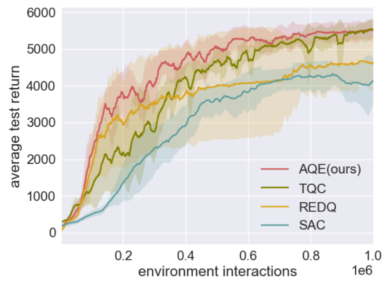
We can see from the results that with fixed hyperparamters, the conclusions for AQE remain largely unchanged, except for Hopper, where REDQ becomes the strongest algorithm. Table 4.4 shows that when averaging performance across environments, AQE still matches the high sample efficiency of REDQ during the early stages of training. Furthermore, Table 4.5 shows that, on average, AQE's asymptotic performance is still 16%, 11% and 9% higher than SAC, REDQ and TQC, respectively.

Amount of data	SAC	TQC	REDQ	AQE	AQE/SAC	AQE/TQC	AQE/REDQ
Hopper at 100K	1456	1719	2747	2294	1.58	1.33	0.84
Walker2d at 100K	501	1215	1810	2150	4.29	1.77	1.19
HalfCheetah at 100K	3055	3594	6876	6325	2.10	1.76	0.92
Ant at 250K	2107	2344	3279	4153	1.97	1.77	1.27
Humanoid at 250K	1094	3038	4535	3973	3.63	1.31	0.88
Average at early stage	-	-	-	-	2.71	1.59	1.02

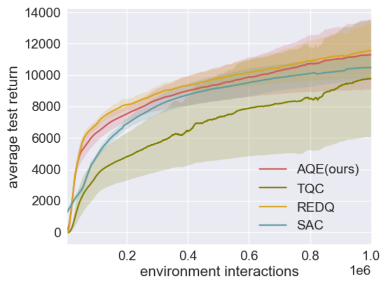
Table 4.4: Early-stage performance comparison of SAC, TQC, REDQ and AQE when AQE and TQC using the same hyperparameters across the environments. On average, AQE performs 2.71 times better than SAC, 1.59 times better than TQC and 1.02 times better than REDQ.



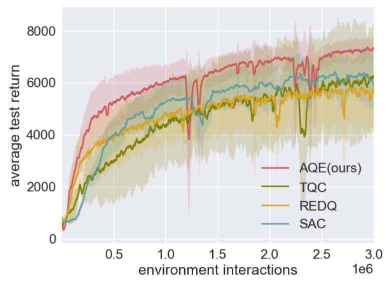
(a) Hopper-v2



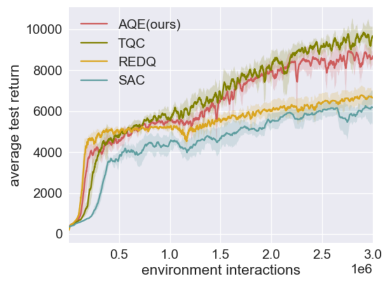
(b) Walker2d-v2



(c) HalfCheetah-v2



(d) Ant-v2



(e) Humanoid-v2

Figure 4.2: Performance for AQE and TQC using same hyper-parameters across the five environments. AQE uses $K = 16$ and TQC uses atoms = 2 per critic.

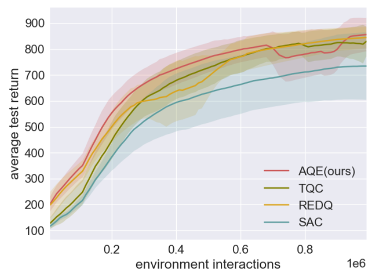
4.4.2 Results on DeepMind Control Suite

In this section, we provide detailed experimental results for AQE, TQC, REDQ and SAC for the nine most challenging DeepMind Control Suite (DMC) environments. The TQC and REDQ papers do not consider DMC benchmark, so we employ the same hyper-parameters for TQC and REDQ as for the MuJoCo environments. For TQC, we employ 5 critic networks with 25 distributional samples and drop 2 atoms per critic across environments. For REDQ, we keep using $N = 10$ and $M = 2$. To make the comparison fair, we also use the same hyperparameter values across environments for AQE. We present the learning curves in Figure 4.3. Similar to MuJoCo benchmark, for each algorithm, we plot the average return of 5 independent trials as the solid curve, and plot the standard deviation across 5 seeds as the shaded region. We run all algorithms to 1 million environment interactions except for the most challenging environment, Humanoid-run, where we run up to 4.5 million environment interactions. We use the same evaluation protocol

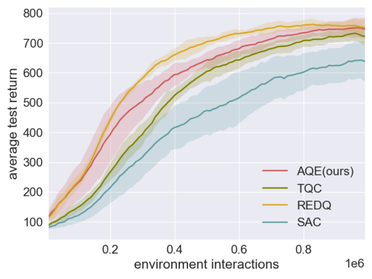
Amount of data	SAC	TQC	REDQ	AQE	AQE/SAC	AQE/TQC	AQE/REDQ
Hopper at 1M	3282	2024	2954	2404	0.73	1.19	0.81
Walker2d at 1M	4134	5532	4637	5517	1.33	1.00	1.19
HalfCheetah at 1M	10475	9792	11562	11293	1.08	1.15	0.98
Ant at 3M	5903	6186	5785	7345	1.24	1.19	1.27
Humanoid at 3M	6177	9593	6649	8680	1.41	0.91	1.31
Average at late stage	-	-	-	-	1.16	1.09	1.11

Table 4.5: Late-stage performance comparison of SAC, TQC, REDQ and AQE when AQE and TQC using the same hyperparameters across the environments. On average, AQE performs 16% better than SAC, 9% better than TQC and 11% times better than REDQ.

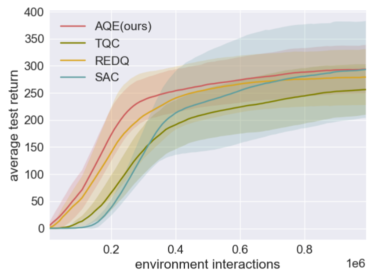
as for the MuJoCo environments.



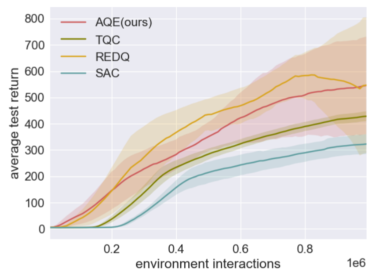
(a) Cheetah run



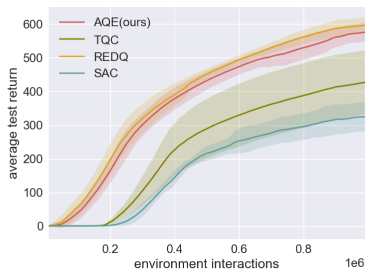
(b) Fish swim



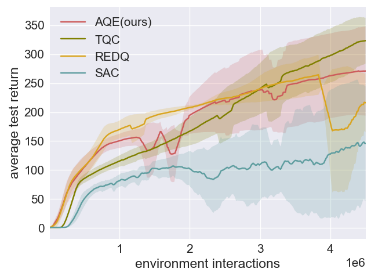
(c) Hopper hop



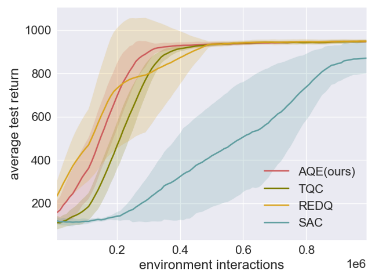
(d) Humanoid stand



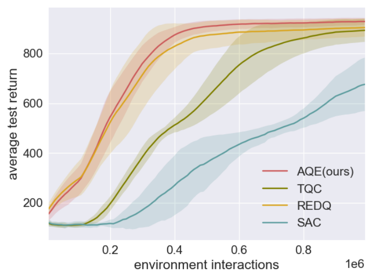
(e) Humanoid walk



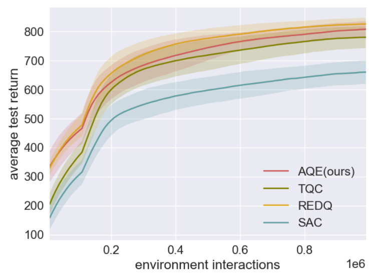
(f) Humanoid run



(g) Quadruped walk



(h) Quadruped run



(i) Walker run

Figure 4.3: AQE versus TQC, REDQ and SAC in DeepMind Control Suite benchmark. AQE and TQC use same hyperparameters across the nine environments.

Figure 4.3 shows that in DMC environments with fixed hyper-parameters, AQE continues to outperform TQC except for the Humanoid-run environment, where TQC performs better than AQE in the final stage of training. AQE and REDQ have comparable results in some of the DMC environments during traning, however, AQE usually outperforms REDQ in the more challenging environments, such as Hopper-hop, Humanoid-run, and Quadruped-run. We report detailed early-stage and late-stage performance comparisons of all algorithms in Table 4.6 and Table 4.7.

Amount of data	SAC	TQC	REDQ	AQE	AQE/SAC	AQE/TQC	AQE/REDQ
Cheetah-run at 100K	205	235	317	339	1.65	1.44	1.07
Fish-swim at 100K	121	149	234	230	1.90	1.54	0.98
Hopper-hop at 100K	2	11	50	64	32.0	5.81	1.28
Quadruped-walk at 100K	116	172	452	341	2.94	1.98	0.75
Quadruped-run at 100K	114	111	294	284	2.49	2.56	0.97
Walker-run at 100K	305	372	468	457	1.50	1.23	0.98
Humanoid-stand at 100K	5	5	37	52	10.4	10.4	1.41
Humanoid-walk at 100K	1	1	57	40	40.0	40.0	0.70
Humanoid-run at 250K	2	18	59	61	30.5	3.39	1.03
Average at early stage	-	-	-	-	13.71	7.59	1.02

Table 4.6: Early-stage performance comparison of SAC, TQC, REDQ and AQE when AQE and TQC using the same hyperparameters across the DMC environments. On average, AQE performs 13.71 times better than SAC, 7.59 times better than TQC and 1.02 times better than REDQ.

Amount of data	SAC	TQC	REDQ	AQE	AQE/SAC	AQE/TQC	AQE/REDQ
Cheetah-run at 1M	734	829	844	856	1.17	1.03	1.01
Fish-swim at 1M	639	722	753	747	1.17	1.03	0.99
Hopper-hop at 1M	293	256	279	294	1.00	1.15	1.05
Quadruped-walk at 1M	871	948	949	948	1.09	1.00	1.00
Quadruped-run at 1M	676	893	904	928	1.37	1.04	1.03
Walker-run at 1M	660	780	826	808	1.22	1.04	0.98
Humanoid-stand at 1M	323	429	547	546	1.69	1.27	1.00
Humanoid-walk at 1M	325	427	596	576	1.77	1.35	0.97
Humanoid-run at 4.5M	146	324	216	271	1.86	0.84	1.25
Average at late stage	-	-	-	-	1.37	1.08	1.03

Table 4.7: Late-stage performance comparison of SAC, TQC, REDQ and AQE when AQE and TQC using the same hyperparameters across the DMC environments. On average, AQE performs 1.37 times better than SAC, 1.08 times better than TQC and 1.03 times better than REDQ.

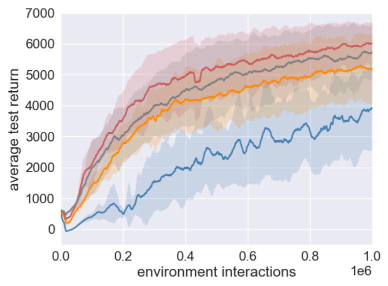
In summary, in the early stage of training (100K data), AQE performs 13.71x better than SAC, 7.59x better than TQC and matches the excellent performance of REDQ in nine environments. In the late-stage training (1M data), AQE always matches or outperforms all other algorithms, except for Humanoid-run, where TQC performs the best. On average, AQE performs 37% better than SAC, 8% better than TQC, and 3% better than REDQ. Moreover, as shown in Table 4.8, using the same hyper-parameters and averaged across nine DMC environments, AQE achieves the asymptotic performance of SAC approximately 3x faster than SAC, 1.57x faster than TQC, and 1.05x faster than REDQ.

Performance	SAC	TQC	REDQ	AQE	AQE/SAC	AQE/TQC	AQE/REDQ
Cheetah-run at 700	746K	440K	506K	350K	2.13	1.26	1.45
Fish-swim at 600	794K	494K	317K	417K	1.90	1.18	0.76
Hopper-hop at 250	580K	856K	451K	371K	1.56	2.31	1.22
Quadruped-walk at 800	844K	301K	302K	236K	3.58	1.28	1.28
Quadruped-run at 650	942K	521K	267K	248K	3.80	2.10	1.08
Walker-run at 600	516K	201K	156K	174K	2.97	1.16	0.90
Humanoid-stand at 250	626K	429K	279K	342K	1.83	1.25	0.82
Humanoid-walk at 300	820K	523K	279K	300K	2.73	1.74	0.93
Humanoid-run at 120	3940K	1100K	602K	603K	6.53	1.82	1.00
Average	-	-	-	-	3.00	1.57	1.05

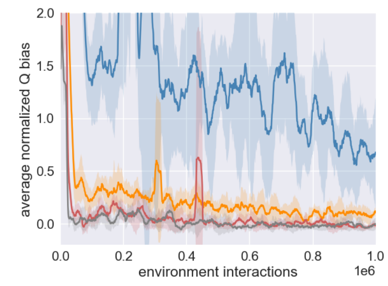
Table 4.8: Sample efficiency comparison of SAC, TQC, REDQ and AQE. The numbers show the amount of data collected when the specified performance level is reached (roughly corresponding to 90% of SAC's final performance). The last three columns show how many times AQE is more sample efficient than SAC, TQC and REDQ in reaching that performance level.

4.4.3 Ablation Study

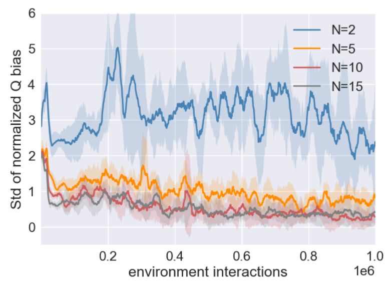
In this section, we use ablations to provide further insight into AQE. As in the REDQ paper, we consider not only performance but normalized bias and standard deviation of normalized bias as defined by the REDQ authors. We focus on the Ant environment, and run the experiments up to 1M time steps. We first look at how the ensemble size N affects AQE. The first row in Figure 4.4 shows AQE with N equal to 2, 5, 10 and 15, with two heads for each Q network, and the percentage of kept Q-values unchanged. As the ensemble size N increases, we generally obtain a more stable average bias, a lower std of bias, and stronger performance. When trained with high UTD value, a relatively small ensemble size, for example, $N = 5$, can greatly reduce bias accumulation, resulting in much stronger performance. This experimental finding is consistent with the results in Theorem 4.1 in Section 4.3.2.



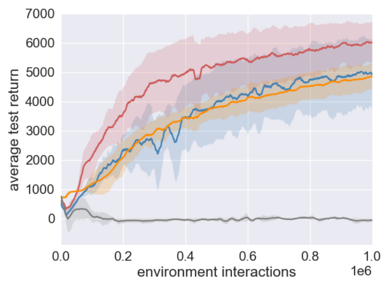
(a) Ensemble size: Perf



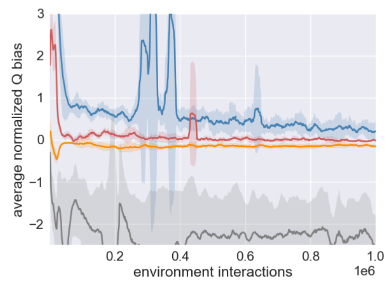
(b) Ensemble size: Bias



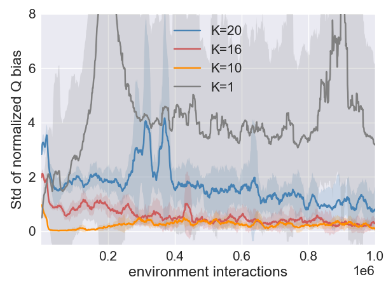
(c) Ensemble size:Std



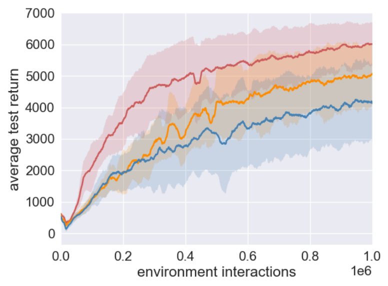
(d) Keep value: Perf



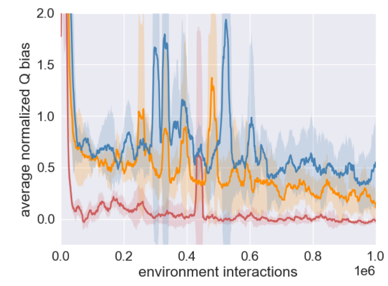
(e) Keep value: Bias



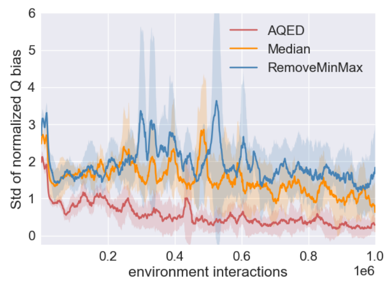
(f) Keep value: Std



(g) Variations: Perf



(h) Variations: Bias



(i) Variations: Std

Figure 4.4: AQE ablation results for Ant. The top row shows the effect of the ensemble size N. The second row shows the effect of keep number parameter K. The third row compares AQE to some variants.

The second row in Figure 4.4 shows the how the keep parameter can affect the algorithm's performance: under the same high UTD value, as K decreases, the average normalized Q bias goes from over-estimation to under-estimation. Consistent with the theoretical result in Theorem 4.1, by decreasing K we lower the average bias. When K becomes too small, the Q estimate becomes too conservative and starts to have negative bias, which makes learning difficult. We see that $K = 16$ has an average bias closest to 0 and also a consistently small std of bias. These results are similar for the other four environments, as shown in Figure 4.5.

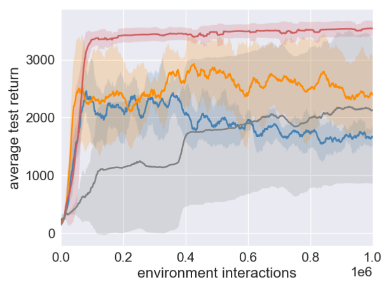
The third row in Figure 4.4 shows results for variants of the target computation methods. The Median curve uses the median value of all the Q estimates in the ensemble to compute the Q target. The RemoveMinMax curve drops the minimum and maximum values of all the Q estimates in the ensemble to compute the Q target. We see that these two variants give larger positive Q bias values.

We also considered different combinations of ensemble size N and the number of multi-heads h while keeping the total number of Q-function estimates $N \cdot h$ fixed. We performed these experiments for all five environments, and the results are shown in Figure 4.6. In terms of performance, we found the two best combinations to be $N = 20$, $h = 1$ and $N = 10$, $h = 2$, with the former being about 50% slower than latter in terms of computation time.

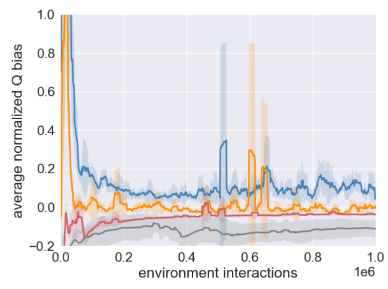
Figure 4.4 only compares the different size of the ensemble N and the number of heads h for Ant. Figure 4.6 shows the results for all five environments. We can see that the combination of $N = 10, h = 2$ and $N = 20, h = 1$ have comparable performance. However, $N = 10$ and $h = 2$ is faster in terms of computation time.

We also consider endowing REDQ with the same multi-head ensemble architecture as AQE. Figure 4.7 examines the performance of REDQ when it is endowed with the same multi-head architecture as AQE. We see that the performance of REDQ does not substantially improve.

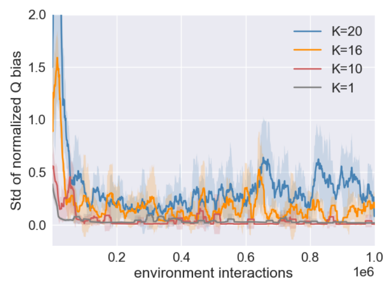
In addition, we include a UTD ratio of $G = 5$ in both SAC and TQC, and compare these aggressive versions with AQE, also with $G = 5$. Figure 4.8 presents the performance of AQE, SAC-5 and TQC-5 for all the environments. SAC-5 and TQC-5 uses UTD ratio G = 5 for SAC and TQC, respectively. We can see that AQE continues to outperform both algorithms except for Humanoid, where TQC performs somewhat better than AQE in the final stage training. SAC becomes more sample efficient with $G = 5$; however, AQE still beats SAC-5 by a large margin.



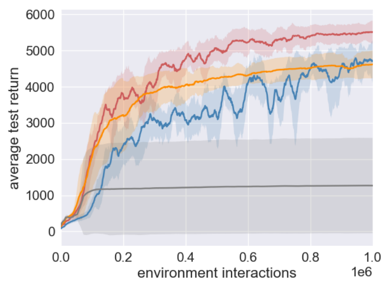
(a) Performance, Hopper



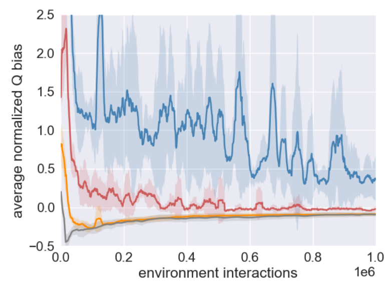
(b) Average normalized bias



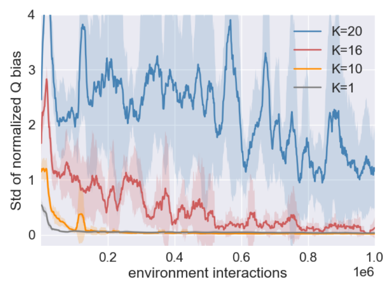
(c) Std of normalized bias



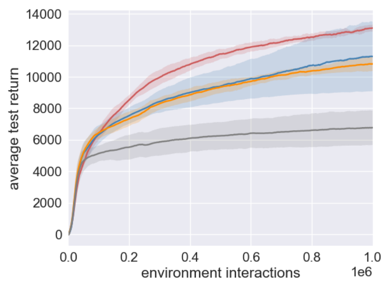
(d) Performance, Walker



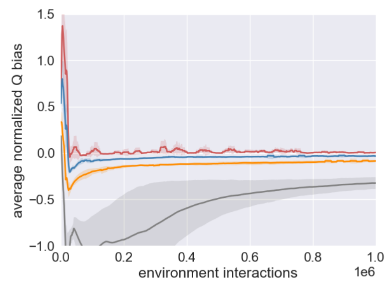
(e) Average normalized bias



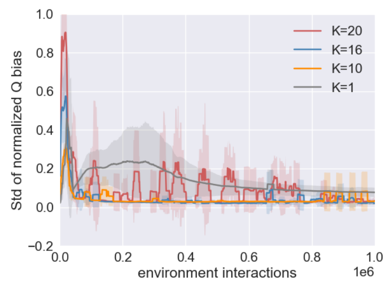
(f) Std of normalized bias



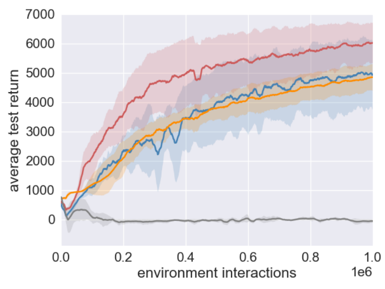
(g) Performance, HalfCheetah



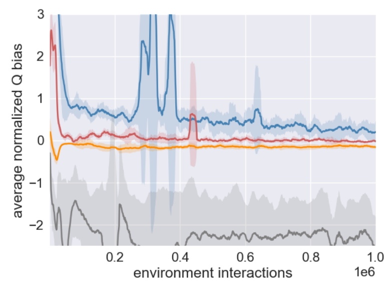
(h) Average normalized bias



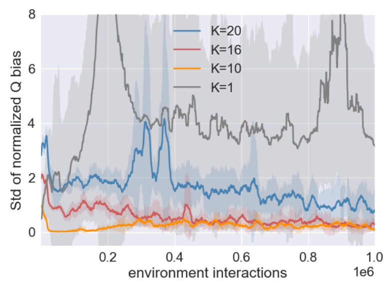
(i) Std of normalized bias



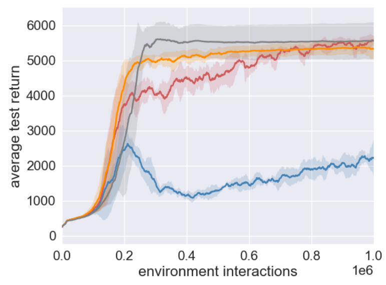
(j) Performance, Ant

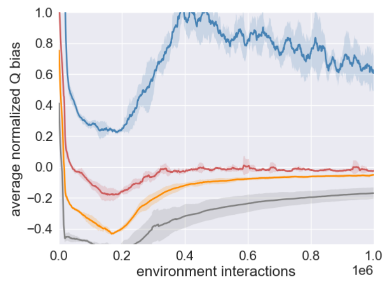


(k) Average normalized bias



(l) Std of normalized bias


(m) Performance, Humanoid


(n) Average normalized bias

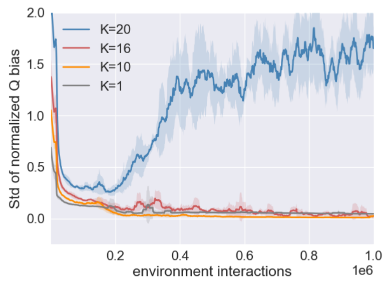
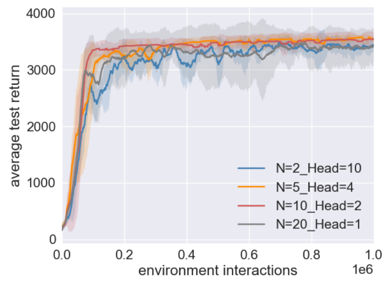
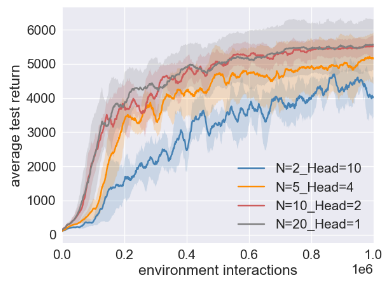
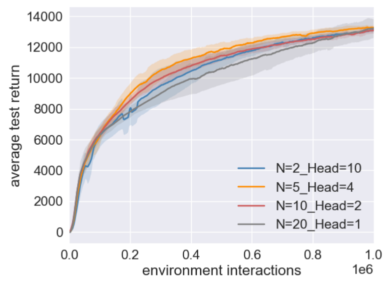
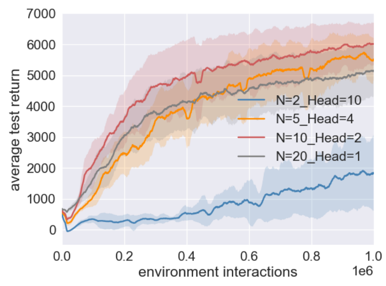

(o) Std of normalized bias

Figure 4.5: Performance, average and std of normalized Q bias for AQE with different values of K.


(a) Hopper-v2


(b) Walker2d-v2


(c) HalfCheetah-v2


(d) Ant-v2

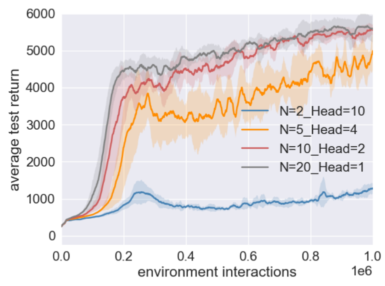
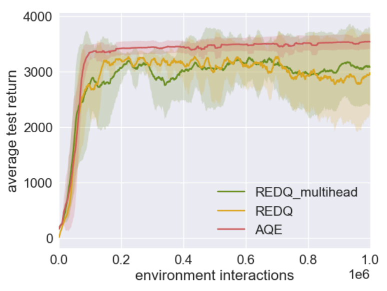

(e) Humanoid-v2

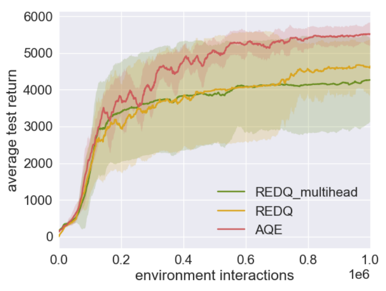
Figure 4.6: Performance for AQE with different combinations of number of Q networks and number of heads.

4.4.4 Implementation Details

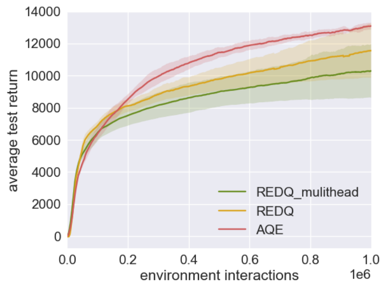
Table 4.9 gives a list of hyperparameters used in the experiments. Most of AQE's hyper-parameters are made the same as in the REDQ paper to ensure fairness and consistency in comparisons, except that AQE has 2-head critic networks. As compared with AQE and REDQ, TQC uses a larger



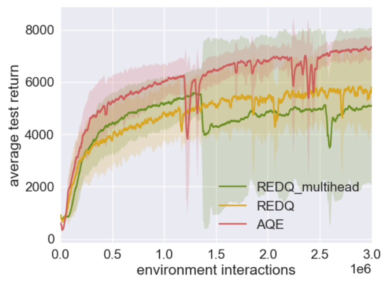
(a) Hopper-v2



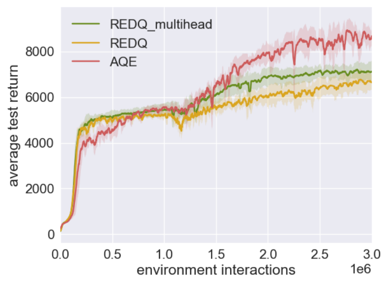
(b) Walker2d-v2



(c) HalfCheetah-v2



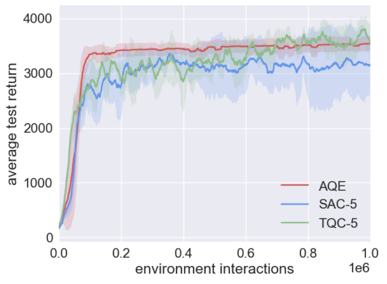
(d) Ant-v2



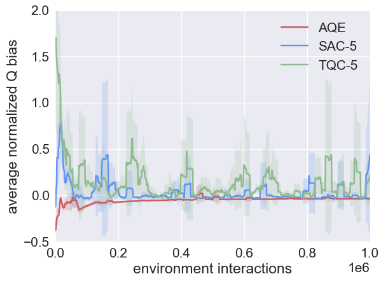
(e) Humanoid-v2

Figure 4.7: Performance of REDQ with N=10 and heads = 2 as compared with REDQ and AQE.

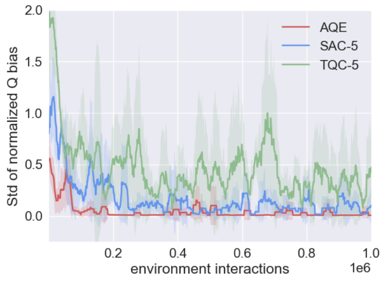
critic network with 3 layers of 512 units per layer. In table 4.10, we report the dropped atoms d for TQC and the number of Q values we keep in the ensemble to calculate the target in AQE.



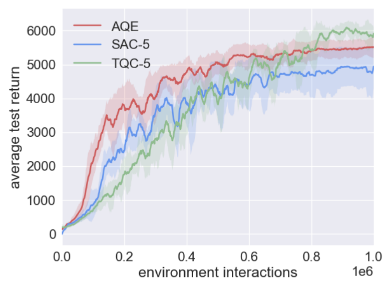
(a) Performance, Hopper



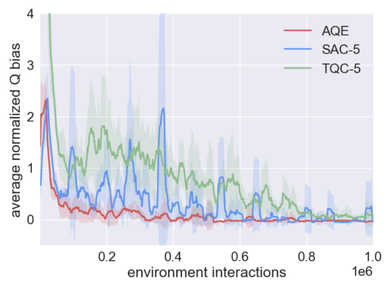
(b) Average normalized bias



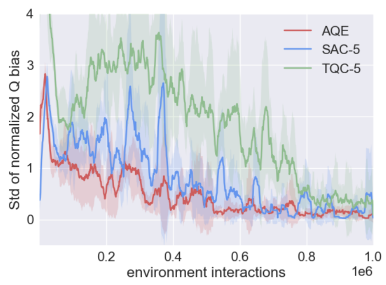
(c) Std of normalized bias



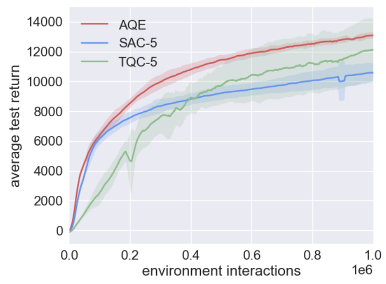
(d) Performance, Walker



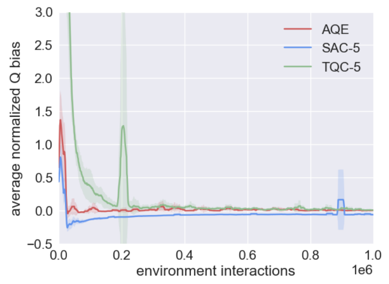
(e) Average normalized bias



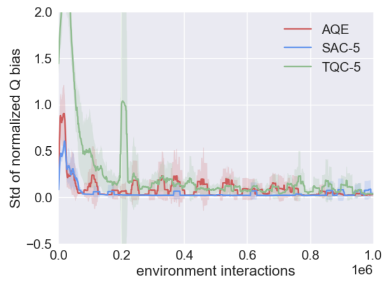
(f) Std of normalized bias



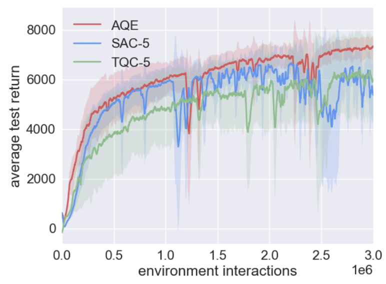
(g) Performance, HalfCheetah



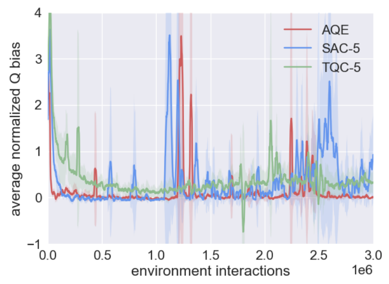
(h) Average normalized bias



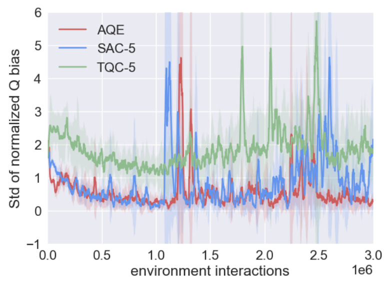
(i) Std of normalized bias



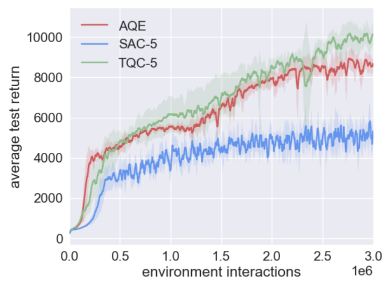
(j) Performance, Ant



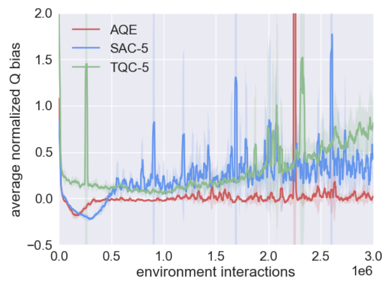
(k) Average normalized bias



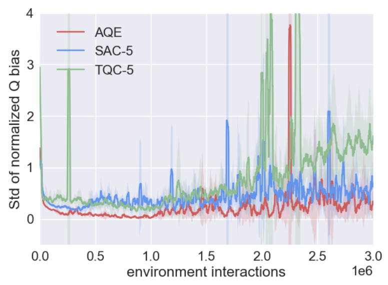
(l) Std of normalized bias



(m) Performance, Humanoid



(n) Average normalized bias



(o) Std of normalized bias

Figure 4.8: Performance, average and std of normalized Q bias for AQE, SAC-5 and TQC-5. All of the algorithms in this experiment use UTD = 5.

Hyperparameters	AQE	SAC	REDQ	TQC
optimizer	Adam[Kingma and Ba 2014]			
learning rate	$3 \cdot 10^{-4}$			
discount(γ)	0.99			
target smoothing coefficient(ρ)	0.005			
replay buffer size	$1 \cdot 10^{6}$			
number of critics N	10	2	10	5
number of hidden layers in critic networks	2	2	2	3
size of hidden layers in critic networks	256	256	256	512
number of heads in critic networks h	2	1	1	25
number of hidden layers in policy network	2			
size of hidden layers in policy network	256			
mini-batch size	256			
nonlinearity	ReLU			
UTD ratio G	5	1	5	1

Table 4.9: Hyperparameter values.

Environment	Dropped atoms per critic	Kept Q values out of $N \cdot h$ values
Hopper	5	10
HalfCheetah	0	20
Walker	2	16
Ant	2	16
Humanoid	2	16

Table 4.10: Environment-dependent hyper-parameters for TQC and AQE.

4.5 Conclusion

We develop a simple model-free algorithm can do surprisingly well, providing state-of-art performance at all stages of training. There is no need for a model, distributional representation of the return, or in-target randomization to achieve high sample efficiency and asymptotic performance.

With extensive experiments and ablations, we show that AQE is both performant and robust. In both OpenAI Gym and DMControl, AQE is able to achieve superior performance in all stages of training with the same hyperparameters, and it can be further improved with per-task finetuning. Our ablations show that AQE is robust to small changes in the hyperparameters. Our theoretical results complement the experimental results, showing that the estimation bias can be controlled by either varying the ensemble size N or the keep parameter K.

AQE along with prior works show that a high update-to-data ratio combined with refined bias control can lead to very significant performance gain. An interesting future work direction is to investigate whether there are other critical factors (in addition to bias control) that can allow us to further benefit from a high update-to-data ratio, and achieve even better sample efficiency with simple model-free methods.

5 | Best-Action Imitation Learning

5.1 Introduction

Offline reinforcement learning, also known as batch reinforcement learning, is the problem of sample-efficient learning from a given dataset without additional interactions with the environment. Batch reinforcement learning in both the tabular and function approximator settings has long been studied [Lange et al. 2012; Strehl et al. 2010] and continues to be a highly active area of research [Swaminathan and Joachims 2015; Jiang and Li 2015; Thomas and Brunskill 2016; Farajtabar et al. 2018; Irpan et al. 2019; Jaques et al. 2019]. It enables reusing the data collected by a policy to possibly improve the policy without further interactions with the environment, it has the potential to leverage existing large datasets to obtain much better sample efficiency. A batch reinforcement learning algorithm can also be deployed as part of a growing-batch algorithm, where the batch algorithm seeks a high-performing exploitation policy using the data in an experience replay buffer [Lin 1992], combines this policy with exploration to add fresh data to the buffer, and then repeats the whole process [Lange et al. 2012; Ernst et al. 2005b].

Standard off-policy deep reinforcement learning algorithms we have discussed in the previous chapters fail to learn well in the offline setting even entirely diverge due to extrapolation error. Many recent offline DRL algorithms address extrapolation errors by policy regularization and critic penalty, as introduced in section 2.4.1 and section 2.4.2. In addition to these two categories, Imitation Learning (IL)-based algorithms, such as Monotonic Advantage Re-Weighted Imitation

Learning (MARWIL) [Wang et al. 2018] and Advantage Weighted Regression (AWR) [Peng et al. 2019], can also obtain high-performing policies from batch data. IL-based algorithms tackle extrapolation error by avoiding Q-learning to query out-of-distribution actions. More concretely, MARWIL uses exponentially weighted imitation learning, with the weights being determined by estimates of the advantage function. AWR is conceptually very similar to MARWIL. It is primarily designed for online learning, but can also be employed in batch RL.

We propose a novel IL-based algorithm called Best-Action Imitation Learning (BAIL), which strives for both simplicity and performance in this chapter. BAIL, being an IL-based algorithm, shares some similarities with other IL-based algorithms MARWIL and AWR [Wang et al. 2018; Peng et al. 2019]. We summarize BAIL to three steps. In the first step, BAIL learns a state value function V by training a neural network to obtain the “upper envelope of the data”. In the second step, it selects from the dataset the state-action pairs whose Monte Carlo returns are close to the upper envelope. In the last step, it simply trains a policy network with vanilla imitation learning using the selected actions. The method thus combines a novel approach for V-learning with IL.

5.2 BAIL

In this section, we present the detailed explanation of BAIL, which not only provides state-of-the-art performance on simulated robotic locomotion tasks, but is also fast and algorithmically simple. The motivation behind BAIL is as follows. For a given deterministic MDP, let $V^*(s)$ be the optimal value function. For a particular state-action pair (s, a), let $G(s, a)$ denote a return using some policy when beginning state s and choosing action a. Any action a^* that satisfies $G(s, a^*) = V^*(s)$ is an optimal action for state s. Thus, ideally we would like to construct an algorithm which finds actions that satisfy $G(s, a^*) = V^*(s)$ for each state s.

In batch reinforcement learning, since we are only given limited data, we can only hope to obtain an approximation of $V^*(s)$. In BAIL, we first try to make the best possible estimate of $V^*(s)$

using only the limited information in the batch dataset. Call this estimate $V(s)$. We then select state-action pairs from the dataset whose associated returns $G(s,a)$ are close to $V(s)$. Finally, we train a policy with IL using the selected state-action pairs. Thus, BAIL combines both V-learning and IL. To obtain the estimate $V(s)$ of the value function, we introduce the "upper envelope of the data".

5.2.1 Upper Envelope of the Data

We first define a λ-regularized upper envelope, and then provide an algorithm for finding it. To the best of our knowledge, the notion of the upper envelope of a dataset is novel.

Recall that we have a batch of data $\mathcal{B} = \{(s_i, a_i, r_i, s_i'),\ i = 1, ..., m\}$. Although we do not assume we know what algorithm was used to generate the batch, we make the natural assumption that the data in the batch was generated in an episodic fashion, and that the data in the batch is ordered accordingly. For each data point $i \in \{1, \ldots, m\}$, we calculate the Monte Carlo return G_i as the sum of the discounted rewards from state s_i to the end of the episode as $G_i = \sum_{t=i}^{T} \gamma^{t-i} r_t$ where T denotes the time at which the episode ends for the episode that contains the ith data point.

Having defined the return for each data point in the batch, we now seek an upper-envelope of the data $\mathcal{G} := \{(s_i, G_i),\ i = 1, ..., m\}$. Let $V_\phi(s)$ denote a neural networks parameterized by $\phi = (w, b)$ that takes as input a state s and outputs a real number, where w and b denote the weights and bias, respectively. For a fixed $\lambda \geq 0$, we say that $V_{\phi^\lambda}(s)$ is a λ-regularized upper envelope for $\mathcal{G}$ if ϕ^λ is an optimal solution for the following constrained optimization problem:

$$\min_{\phi} \sum_{i=1}^{m} [V_\phi(s_i) - G_i]^2 + \lambda \|w\|^2 \qquad s.t. \qquad V_\phi(s_i) \geq G_i, \qquad i = 1, 2, \ldots, m \tag{5.1}$$

Note that a λ-regularized upper envelope always lies above all the returns. The optimization problem strives to bring the envelope as close to the data as possible while maintaining regularization to prevent overfitting. The solution ϕ^λ to the constrained optimization problem may not be unique. Nevertheless, we have the following theorem to characterize the limiting behavior of λ-regularized upper envelopes.

Theorem 5.1. *Suppose $V_\phi(s)$ is a multi-layer fully connected neural network with ReLu activation units, and there is a bias term at the output layer. For each $\lambda \geq 0$, let $V_{\phi^\lambda}(s)$ be a λ-regularized upper envelope for $\mathcal{G}$, that is, $\phi^\lambda = (w^\lambda, b^\lambda)$ is an optimal solution of the above constrained optimization problem. Then, we have*

(1) $\lim_{\lambda \to \infty} V_{\phi^\lambda}(s) = \max_{1 \leqslant i \leqslant m} \{G_i\}$ for all $s \in \mathcal{S}$.

(2) When $\lambda = 0$, if there are sufficient number of activation units and layers, then $V_{\phi^0}(s)$ will interpolate the data in $\mathcal{G}$, i.e., $V_{\phi^0}(s_i) = G_i$ for all $i = 1, \dots, m$.

Proof. Part (1): For any $\lambda \geq 0$ and $\phi = (w, b)$ define

$$J^\lambda(\phi) = \sum_{i=1}^{m} [V_\phi(s_i) - G_i]^2 + \lambda \|w\|^2 \tag{5.2}$$

Note that for any ϕ of the form $\phi = (0, b)$, we have $V_{(0,b)}(s) = b$ for all s and $J^\lambda(0, b) = \sum_{i=1}^{m} (b - G_i)^2$ for all $\lambda \geq 0$. Also define

$$G^* := \max_{1 \leqslant i \leqslant m} \{G_i\}$$

and $\hat{\phi} = (0, G^*)$. Note that $\hat{\phi}$ is feasible for the constrained optimization problem. It therefore follows that for any $\lambda \geq 0$:

$$J^\lambda(\phi^\lambda) \leq J^\lambda(\hat{\phi}) = \sum_{i=1}^{m} (G^* - G_i)^2 := H^* \tag{5.3}$$

We first show that $\lim_{\lambda \to \infty} w^\lambda = 0$. To proceed with a proof by contradiction, assume that this is not true. There then exists an $\epsilon > 0$ such that for any $\lambda \geq 0$ there exists some $\lambda' \geqslant \lambda$ such that

$\|w^{\lambda'}\|^2 > \epsilon$. Choosing $\lambda = H^*/\epsilon$, we have for some λ':

$$J^{\lambda'}(\phi^{\lambda'}) \geqslant \lambda' \|w^{\lambda'}\|^2 > \lambda \cdot \epsilon = H^* \tag{5.4}$$

But this contradicts (5.3), establishing $\lim_{\lambda \to \infty} w^\lambda = 0$.

Next, we show $\lim_{\lambda \to \infty} b^\lambda = G^*$. To prove this, we will show $\bar{b} := \limsup_{\lambda \to \infty} b^\lambda = G^*$ and also $\tilde{b} := \liminf_{\lambda \to \infty} b^\lambda = G^*$. First, consider a subsequence $\{b^{\lambda_n}\}$ such that $\lim_{n \to \infty} b^{\lambda_n} = \tilde{b}$. Due to the continuity of $\phi \longrightarrow V_\phi(s)$ and $\lim_{\lambda \to \infty} w^\lambda = 0$, we have

$$\lim_{n \to \infty} V_{\phi^{\lambda^n}}(s) = V_{(0,\tilde{b})}(s) = \tilde{b} \qquad \forall s \tag{5.5}$$

Moreover, since ϕ^{λ^n} has to satisfy the constraints, we also have

$$V_{\phi^{\lambda^n}}(s_j) \geqslant G_j = G^* \tag{5.6}$$

where $j = \arg\max_i G_i$. Therefore, combining (5.5) and (5.6) yields

$$\liminf_{\lambda \to \infty} b^\lambda \geqslant G^* \tag{5.7}$$

Similarly, consider another subsequence $\{b^{\lambda_k}\}$ such that $\lim_{k \to \infty} b^{\lambda_k} = \bar{b}$. Again, we have

$$\lim_{k \to \infty} V_{\phi^{\lambda_k}}(s) = \bar{b} \qquad \forall s \tag{5.8}$$

We have from (5.3) that

$$J^{\lambda_k}(\phi^{\lambda_k}) \leq H^* \tag{5.9}$$

Letting $k \longrightarrow \infty$ gives

$$\sum_{i=1}^{m} (\bar{b} - G_i)^2 \leqslant \sum_{i=1}^{m} (G^* - G_i)^2 \tag{5.10}$$

which implies that

$$\bar{b} = \limsup_{\lambda\to\infty} b^{\lambda} \leqslant G^* \tag{5.11}$$

Therefore, combining (5.7) and (5.11) together, we have finally shown that $\lim_{\lambda\to\infty} b^{\lambda} = G^*$. As we have previously shown $\lim_{\lambda\to\infty} w^{\lambda} = 0$, it follows that

$$\lim_{\lambda\to\infty} V_{\phi^{\lambda}}(s) = \max_{1\leqslant i\leqslant m}\{G_i\}, \qquad \forall s \tag{5.12}$$

Part (2): For the case of $\lambda = 0$, notice that we have finitely many inputs s_i to feed into the neural network. Therefore, this is a typical problem regarding the *finite-sample expressivity* of neural networks, and the proof directly follows from the work done in [Zhang et al. 2017]. $\square$

From the theorem 5.1, we see that when λ is very small, the upper envelope aims to interpolate the data, and when λ is large, the upper envelope approaches a constant going through the data point with the highest return. Just as in classical regression, there is a sweet-spot for λ, the one that provides the best generalization.

We solve the constrained optimization problem (5.1) with a penalty-function approach. Specifically, to obtain an approximate upper envelope of the data $\mathcal{G}$, we solve an unconstrained optimization problem with a penalty loss function (with λ fixed):

$$L^K(\phi) = \sum_{i=1}^{m}(V_\phi(s_i) - G_i)^2\{\mathbb{1}_{(V_\phi(s_i)\geqslant G_i)} + K\cdot\mathbb{1}_{(V_\phi(s_i)<G_i)}\} + \lambda\|w\|^2 \tag{5.13}$$

where $K >> 1$ is the penalty coefficient and $\mathbb{1}_{(\cdot)}$ is the indicator function. For a finite K value, the penalty loss function will produce an approximate upper envelope $V(s_i)$, since $V(s_i)$ may be slightly less than G_i for some data points. In practice, we find $K = 1000$ works well for all environments tested. For $K \to \infty$, we have the following theoretical justification of the approximation:

Theorem 5.2. *Let ϕ^K be a solution that minimizes $L^K(\phi)$ with penalty constant K. Let ϕ^* be a limit point of $\{\phi^K\}$. Then $V_{\phi^*}(s)$ is an exact λ-regularized upper envelope, i.e., ϕ^* is an optimal solution* *for the constrained optimization problem (5.1).*

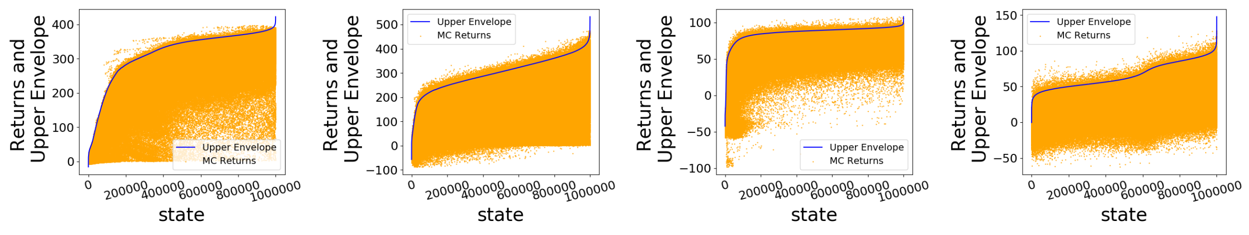
Proof. Let $J^{\lambda}(\phi) = \sum_{i=1}^{m} [V_{\phi}(s_i) - G_i]^2 + \lambda \|w\|^2$ be the loss function that defines the λ-regularized upper envelope. We notice that the penalty loss function $L^K(\phi)$ takes the form:

$$L^K(\phi) = J^{\lambda}(\phi) + (K-1) \sum_{i=1}^{m} (\max\{0, G_i - V_{\phi}(s_i)\})^2 \tag{5.14}$$

The theorem then directly follows from the standard convergence theorem for penalty functions [Luenberger and Ye 2008]. □

In practice, instead of L_2 regularization, we employ a mechanism similar to early-stopping regularization. We split the data into a training set and validation set. During training, after every epoch, we check whether the validation error for the penalty loss function (5.13) decreases, and stop updating the network parameters when the validation loss increases repeatedly. We describe the details of the early-stopping scheme in the section 5.3.4.

Figure 5.1 provides some examples of upper envelopes obtained with training sets consisting of 1 million data points. Each figure shows the upper envelope and the returns for one environment. To aid visualization, the states are ordered in terms of their upper envelope $V(s_i)$ values.



(a) Hopper **(b)** Walker2d **(c)** HalfCheetah **(d)** Ant

Figure 5.1: Upper Envelopes trained on batches from different MuJoCo environments.

Upper envelopes visualization for the rest of the environments is shown in supplementary material in section A.1.

5.2.2 Action Selection

For action selection, BAIL employs the upper envelope to select the best (s, a) pairs from the batch data $\mathcal{B}$. Let $V(s)$ denote the upper envelope obtained from minimizing the penalty loss function (5.13) for a fixed value of K. We consider two approaches for selecting the best actions. In the first approach, which we call BAIL-ratio, for a fixed $x > 0$, we choose all (s_i, a_i) pairs from the batch data set $\mathcal{B}$ such that

$$G_i > xV(s_i) \tag{5.15}$$

We set x such that $p\%$ of the data points are selected, where p is a hyper-parameter. In this chapter we use $p = 25\%$ for all environments and batches. In the second approach, which we call BAIL-difference, for a fixed $x > 0$, we choose all (s_i, a_i) pairs from the batch data set $\mathcal{B}$ such that

$$G_i \geq V(s_i) - x \tag{5.16}$$

In our experiments, BAIL-ratio and BAIL-difference have similar performance, with BAIL-ratio sometimes a little better. We henceforth only consider BAIL-ratio, and simply refer to it as BAIL.

In summary, BAIL employs two neural networks. The first network is used to approximate the optimal value function based on the data in the batch $\mathcal{B}$. The second network is the policy, which is trained with imitation learning. We refer to the algorithm just described as BAIL. We also consider a variation, which we call Progressive BAIL, in which we train the upper envelope parameters ϕ and the policy network parameters θ in parallel rather than sequentially. Progressive BAIL doesn't change how we obtain the upper envelope, since the upper envelope does not depend on the policy parameters in either BAIL or Progressive BAIL. It does, however, affect the training of the policy parameters. We provide detailed pseudo-code for both BAIL and Progressive BAIL in algorithm 5 and 6, which include the early stopping scheme. Note BAIL has two for loops in series, whereas Progressive BAIL has only one for loop.

Our experimental results in section 5.3.2 show that BAIL and Progressive BAIL both perform well with about the same performance over all batches. But BAIL might be a better choice since it's much faster.

Algorithm 5 BAIL

```
Initialize upper envelope parameters $\phi, \phi'$, policy parameters $\theta$. Obtain batch data $\mathcal{B}$. Randomly split data into training set $\mathcal{B}_t$ and validation set $\mathcal{B}_v$ for the upper envelope.
Compute return $G_i$ for each data point $i$ in $\mathcal{B}$.
Obtain upper envelope by minimizing the loss $L^K(\phi)$:
for $j = 1, \dots, J$ do
    Sample a mini-batch $B$ from $\mathcal{B}$.
    Update $\phi$ using the gradient: $\nabla_\phi \sum_{i \in B} (V_\phi(s_i) - G_i)^2 \{\mathbb{1}_{(V_\phi(s_i) > G_i)} + K\mathbb{1}_{(V_\phi(s_i) < G_i)}\} + \lambda \|\phi\|^2$
    if time to do validation for the upper envelope then
        Compute validation loss on $B_v$
        Update $\phi$ and $\phi'$ according to the validation loss
    end if
end for
Select data point $i$ if $G_i > xV_\phi(s_i)$, where $x$ is such that $p\%$ of data in $\mathcal{B}$ are selected. Let $\mathcal{U}$ be the set of selected data points.
for $l = 1, \dots, L$ do
    Sample a mini-batch $U$ of data from $\mathcal{U}$.
    Update $\theta$ using the gradient: $\nabla_\theta \sum_{i \in U} (\pi_\theta(s_i) - a_i)^2$
end for
```

Algorithm 6 Progressive BAIL

```
Initialize upper envelope parameters $\phi, \phi'$, policy parameters $\theta$.
Obtain batch data $\mathcal{B}$. Randomly split data into training set $\mathcal{B}_t$ and validation set $\mathcal{B}_v$ for the upper envelope.
Compute return $G_i$ for each data point $i$ in $\mathcal{B}$.
for $l = 1, \dots, L$ do
    Sample a mini-batch of data $B$ from the batch $\mathcal{B}_t$.
    Update $\phi$ using the gradient: $\nabla_\phi \sum_{i \in B_t} (V_\phi(s_i) - G_i)^2 \{\mathbb{1}_{(V_\phi(s_i) > G_i)} + K\mathbb{1}_{(V_\phi(s_i) < G_i)}\} + \lambda \|\phi\|^2$
    if time to validate then
        Compute validation loss on $B_v$
        Update $\phi$ and $\phi'$ according to validation loss
    end if
    Select data point $i$ if $G_i > xV_\phi(s_i)$, where $x$ is such that $p\%$ of data in $B$ are selected. Let $U$ be the set of selected data points.
    Update $\theta$ using the gradient: $\nabla_\theta \sum_{i \in U} (\pi_\theta(s_i) - a_i)^2$
end for
```

5.2.3 Augmented and Oracle Returns

Both BCQ and BEAR papers use the MuJoCo robotic locomotive benchmarks to gauge the performance of their algorithms [Fujimoto et al. 2018a] [Kumar et al. 2019a]. We will compare the performance of BAIL with BCQ, BEAR, MARWIL and BC using the same MuJoCo environments.

The MuJoCo environments are naturally infinite-horizon non-episodic continuing-task environments [Sutton and Barto 2018]. During training, however, researchers typically create artificial episodes of maximum length 1000 time steps; after 1000 time steps, a random initial state is chosen and a new episode begins. This means that to apply BAIL, we need to approximate infinite-horizon discounted returns using the finite-length episodes in the data set. For data points appearing near the beginning of the episode, the finite-horizon return will closely approximate the (idealized) infinite-horizon return due to discounting; but for a data point near the end of an episode, the finite horizon return can be inaccurate and should be augmented. To calculate the augmentation for the ith data point, we use the following heuristic. Let $\mathcal{E} \subset \mathcal{B}$ denote the episode of data that contains the ith data point, and let s' be the last state in episode $\mathcal{E}$. We then set s_j to be the state in the first $\max\{1000 - i, 200\}$ data points of the episode $\mathcal{E}$ that is closest (in Euclidean norm) to the "terminal state" s'. We then set

$$G_i = \sum_{t=i}^{T} \gamma^{t-i} r_t + \gamma^{T-i+1} \sum_{t=j}^{T} \gamma^{t-j} r_t \tag{5.17}$$

Note that G_i in (5.17) will have at least 800 terms, so there is no need for additional terms due to the discounting. Importantly, the rewards in the two sums in (5.17) are generated by the same policy. The first sum uses the actual rewards accrued until to the end of the episode; the second sum approximates what the actual rewards would have been if the episode was allowed to continue past 1000 time steps.

To validate this heuristic, we did an ablation study in section 5.3.3 comparing the performance of BAIL with the augmentation heuristic and with oracle for Hopper-v2 for seven diverse batches. The results are shown in figure 5.3.

5.3 Experiments

In this section, we provide a comprehensive comparison of five algorithms: BAIL, BCQ, BEAR, MARWIL and BC using 62 diverse batches (many of which are similar to those used in the BCQ and BEAR papers). The batch description is provided in section 5.3.1. We use authors' code and recommended hyper-parameters when available, and we strive to make the comparisons as fair as possible.

5.3.1 Data Batch Description

For generating datasets, we use the same procedures proposed in the BCQ and BEAR papers, and compare the algorithms using those data sets. The BCQ [Fujimoto et al. 2018a] and BEAR [Kumar et al. 2019a] papers generate data batches for five MuJoCo environments: HalfCheetah-v2, Hopper-v2, Walker2d-v2, Ant-v2, and Humanoid-v2. Both algorithm use batch datasets of one million samples, but generate the batches using different approaches. One important observation we make, which was not brought to light in previous offline DRL papers, is that batches generated with different seeds but with otherwise exactly the same algorithm can give drastically different results for offline DRL. Because of this, for every experimental scenario considered in this chapter, we generate two batches, each generated with a different random seed.

More specifically, We generate batches *while* training DDPG [Lillicrap et al. 2019] from scratch with exploration noise of $\sigma = 0.5$ for HalfCheetah-v2, Hopper-v2, and Walker2d-v2, as exactly done in the BCQ paper. We also generate batches with $\sigma = 0.1$ to study the robustness of tested algorithms with lower noise level. We also generate training batches for all five environments by training policies with adaptive Soft Actor Critic (SAC) [Haarnoja et al. 2018b]. This gives six

DDPG and five SAC scenarios. For each one, we generate two batches with different random seeds, giving a total of 22 "training batches" composed of non-expert data. These batches are the most important ones to measure the performance of a batch method, since they contain sub-optimal data obtained from the training process well before optimal performance is achieved (and in many cases using sub-optimal algorithms for training), which is difficult for vanilla behavioral cloning to use.

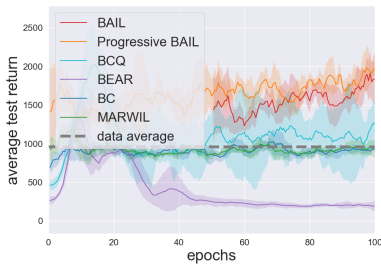
In addition to training batches, we also study execution batches. We do a similar procedure as in the BEAR paper: first train SAC [Haarnoja et al. 2018b] for a certain number of environment interactions, then *fixes the trained policy* and generates one million "execution data points". The BEAR paper generates batches with "mediocre data" where training is up to a mediocre performance, and with "optimal data" where training is up to near optimal performance. When generating the batches with the trained policy, the BEAR paper continues to include exploration noise, using the trained $\sigma(s)$ values in the policy network. Since after training, a test policy is typically deployed without exploration noise, we also consider noise-free data generation. The BEAR paper considers the same five MuJoCo environments considered here. This gives rise to 20 scenarios. For each one we generate two batches with random seeds, giving a total of 40 "execution batches".

5.3.2 Main Results

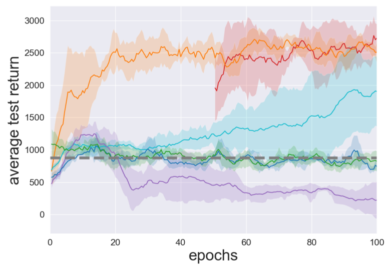
We now carefully compare the five algorithms. For a fair comparison, we keep all hyper-parameters fixed for all experiments, instead of fine-tuning for each one. For BCQ we use the authors' code with their default hyper-parameters. For BEAR we use the authors' code with their version "0" with "use ensemble variance" set to False and employ the recommended hyper-parameters. Because the MARWIL code is not publicly available, we write our own code, and use neural networks the same size as in BAIL. In the section 5.3.4, we provide more details on implementations, and explain how the comparisons are carefully and fairly done.

For each algorithm, we train for 100 epochs (with each epoch consisting of one million data points). For each algorithm, after every 0.5 epochs, we run ten test episodes with the current policy to evaluate performance. We then repeat the procedure for five seeds to obtain the mean and confidence intervals shown in the learning curves. Due to page length limitations, we cannot present the learning curves for all 62 datasets. Here we focus on the 6 DDPG training data batches with $\sigma = 0.5$ (corresponding to the datasets in the BCQ paper), and present the learning curves for the other batches in the supplementary material A.1. However, we present summary results for all datasets in this section.

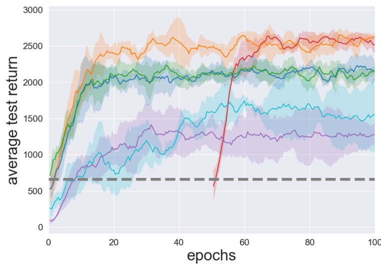
Figure 5.2 shows the learning curves for the 6 DDPG data sets over 100 epochs. As is commonly done, we present smoothed average performance and standard deviations. Note that for BAIL, all curves start at 50 epochs. This provides a fair comparison, since for BAIL we first use 50 epochs of data to train the upper envelopes and then use imitation learning to train the policy network. The horizontal grey dashed line indicates the average return of episodes contained in the batch.



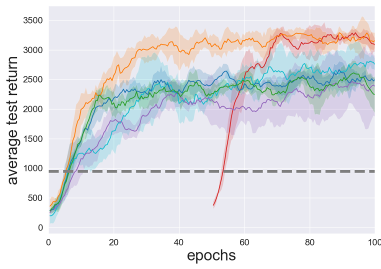
(a) Hopper, batch 1



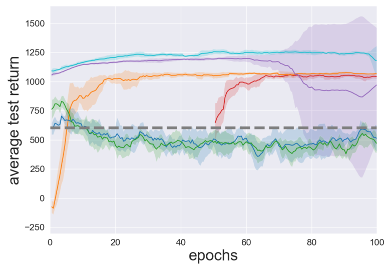
(b) Hopper, batch 2



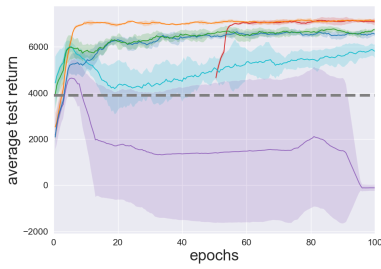
(c) Walker2d, batch 1



(d) Walker2d, batch 2



(e) HalfCheetah, batch 1



(f) HalfCheetah, batch 2

Figure 5.2: Learning curves using DDPG training batches with $\sigma = 0.5$.

Table 5.1: Performance of five Batch DRL algorithms for 22 different training datasets.

Environment	BAIL	BCQ	BEAR	BC	MARWIL
$\sigma = 0.1$ Hopper B1	**2173 ± 291**	1219 ± 114	505 ± 285	626 ± 112	827 ± 220
$\sigma = 0.1$ Hopper B2	**2078 ± 180**	1178 ± 87	985 ± 3	579 ± 141	620 ± 336
$\sigma = 0.1$ Walker B1	**1125 ± 113**	576 ± 309	610 ± 212	514 ± 17	436 ± 24
$\sigma = 0.1$ Walker B2	**3141 ± 300**	2338 ± 388	2707 ± 425	1741 ± 239	1810 ± 200
$\sigma = 0.1$ HC B1	**5746 ± 29**	**5883 ± 43**	0 ± 0	**5546 ± 29**	**5573 ± 35**
$\sigma = 0.1$ HC B2	**7212 ± 43**	**7562 ± 31**	0 ± 0	6765 ± 108	**6828 ± 111**
$\sigma = 0.5$ Hopper B1	**2054 ± 158**	1145 ± 300	203 ± 42	919 ± 52	946 ± 103
$\sigma = 0.5$ Hopper B2	**2623 ± 282**	1823 ± 555	241 ± 239	694 ± 64	818 ± 112
$\sigma = 0.5$ Walker B1	**2522 ± 51**	1552 ± 455	1248 ± 181	2178 ± 178	2111 ± 52
$\sigma = 0.5$ Walker B2	**3115 ± 133**	2785 ± 123	2302 ± 630	2483 ± 94	2364 ± 228
$\sigma = 0.5$ HC B1	1055 ± 9	**1222 ± 38**	924 ± 579	570 ± 35	512 ± 43
$\sigma = 0.5$ HC B2	**7173 ± 120**	5807 ± 249	-114 ± 140	**6545 ± 171**	**6668 ± 93**
SAC Hopper B1	**3296 ± 105**	2681 ± 438	1000 ± 110	2853 ± 318	2897 ± 227
SAC Hopper B2	1831 ± 915	**2134 ± 917**	1139 ± 317	**2240 ± 367**	**2063 ± 168**
SAC Walker B1	**2455 ± 211**	**2408 ± 84**	-3 ± 5	1674 ± 277	1484 ± 140
SAC Walker B2	**4767 ± 130**	3794 ± 398	325 ± 75	2599 ± 145	2651 ± 268
SAC HC B1	**10143 ± 77**	8607 ± 473	7392 ± 257	8874 ± 221	9105 ± 90
SAC HC B2	**10772 ± 59**	**10106 ± 134**	7217 ± 273	9523 ± 164	9488 ± 136
SAC Ant B1	**4284 ± 64**	**4042 ± 113**	3452 ± 128	**3986 ± 112**	**4033 ± 130**
SAC Ant B2	**4946 ± 148**	**4640 ± 76**	3712 ± 236	**4618 ± 111**	**4589 ± 130**
SAC Humanoid B1	**3852 ± 430**	1411 ± 250	0 ± 0	543 ± 378	589 ± 121
SAC Humanoid B2	**3565 ± 153**	1221 ± 207	0 ± 0	1216 ± 826	1033 ± 257

Table 5.1 presents our main results, comparing the five algorithms for the 22 training batches. In batch DRL, since there is no interaction with the environment, one cannot use the policy parameters that provided the highest test returns to assess performance. So in Table 5.1 , for each algorithm, we assume that the practitioner would use the policy obtained after 100 epochs of training. Since there can be significant variation in performance from one policy to the next during training, we calculate the average performance across epochs 95.5 to 100 (i.e., averaged over the last ten tested policies). We do this for each of the five training seeds. We then report the average and standard deviation of these values across the five seeds. For each batch, all the algorithms that are within 10% of the highest average value are considered winners and are indicated in bold.

From Table 5.1 we observe that for the training batches, BAIL is the clear winner. BAIL wins for 20 of the 22 batches, and BCQ is in second place winning for only 8 of the 22 batches. These results show that BAIL is very robust for a wide variety of training datasets, including non-expert datasets, datasets generated as described in the BCQ paper, and for datasets with the more challenging Ant and Humanoid environments. To evaluate the average performance improvement of BAIL over BCQ, for each batch we take the ratio of the BAIL performance to the BCQ performance and then average over the 22 bathes. We also do the same for BC. With this metric, BAIL performs 42% better than BCQ, and 101% better than BC. BAIL is also more stable across seeds: The normalized standard deviations (standard deviation divided by average performance) of BAIL, averaged over the 22 batches, is about half that of BCQ. Because BAIL performs so well for training batches, BAIL can potentially be successfully used for growing batch DRL.

We also note that BEAR occasionally performs very poorly. This is likely because we are using one set of recommended BEAR hyper-parameters for all environments, whereas the BEAR paper reports results using different hyper-parameters for different environments. We also note that for the MuJoCo environments, MARWIL performs similarly to BC.

For the execution batches, the results are given in table 5.2. When BAIL uses the same hyper-parameters as for training batches (though fine-tuning will yield better results, we strive for a fair comparison), BC, MARWIL, BAIL, and BCQ have similar overall performance, with BC being the most robust and the overall winner. Comparing BAIL and BCQ, BAIL has slightly stronger average performance score, and BCQ has a few more wins. It is no surprise that BC is the strongest here, since the execution batches are generated with a single fixed policy and are easy for BC to learn well. These results imply that the focus of future research on batch DRL should be on training batches, or other diverse datasets, since vanilla BC already works very well for fixed-policy datasets.

Intuitively, BAIL can perform better than BCQ and BEAR because these policy-constraint methods rely on carefully tuned constraints to prevent the use of out-of-distribution actions. A loose constraint can cause extrapolation error to accumulate quickly, and a tight constraint will prevent the policy from choosing some of the good actions. BAIL, however, identifies and imitates the highest-performing actions in the dataset, thus avoiding the need to carefully tune such a constraint.

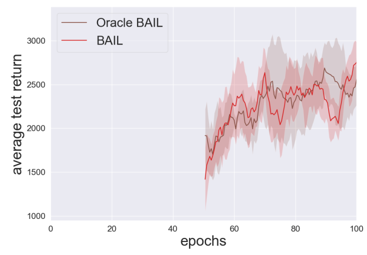
In our experiments we run all algorithms each for 100 epochs for five seeds for each batch. For training with one seed, it takes $1 \sim 2$ hours for BAIL (including the time for upper envelope training and imitation learning), $12 \sim 24$ hours for Progressive BAIL, $36 \sim 72$ hours for BCQ and $60 \sim 100$ hours for BEAR on a CPU node. Thus, roughly speaking, training BAIL is roughly 35 times faster than BCQ and 50 times faster than BEAR.

5.3.3 Ablation Study

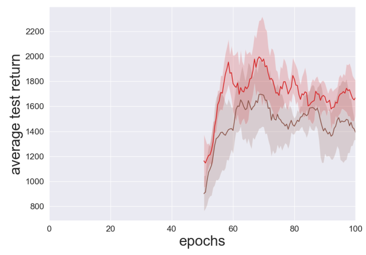
To validate our heuristic for the augmented returns in section 5.2.3, we compute oracle returns by letting episodes run up to 2000 time steps. In this manner, every return is calculated with at least 1000 actual rewards, and is therefore essentially exact due to discounting. Figure 5.3 compares the performance of BAIL using our augmentation heuristic and BAIL using the oracle for Hopper-v2 for seven diverse batches.

The results show that our augmentation heuristic typically achieves oracle-level performance. We conclude that our augmentation heuristic is a satisfactory method for addressing continual environments such as MuJoCo, which is also confirmed with its good performance shown in Table 5.2 .

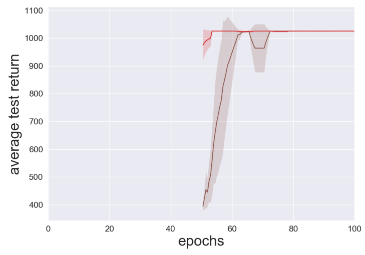
BAIL uses an upper envelope to select the “best” data points for training a policy network with imitation learning. It is natural to ask how BAIL would perform when using the more naive approach of selecting the best actions by simply selecting the same percentage of data points with the highest G_i values, and also by constructing the value function with regression rather than with an upper envelope. These schemes do not do as well as BAIL by a wide margin.



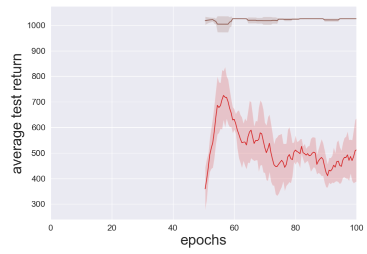
(a) DDPG training batch with $\sigma = 0.5$



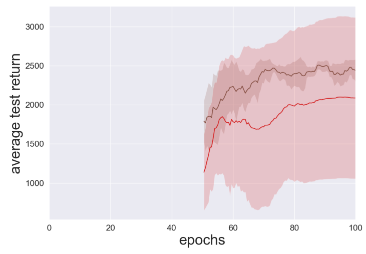
(b) DDPG training batch with $\sigma = 0.1$



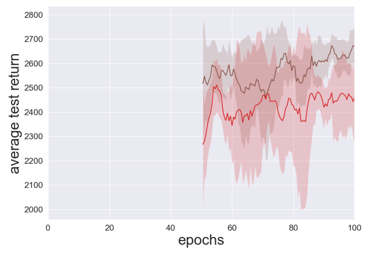
(c) SAC mediocre execution batch with $\sigma = 0$



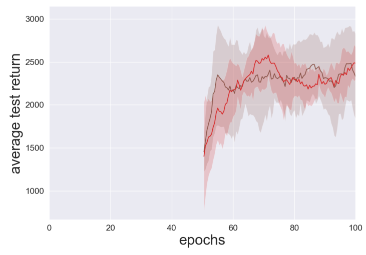
(d) SAC mediocre execution batch with $\sigma = \sigma(s)$



(e) SAC optimal execution batch with $\sigma = 0$



(f) SAC optimal execution batch with $\sigma = \sigma(s)$

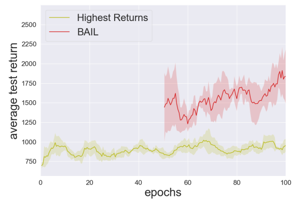


(g) SAC training batch

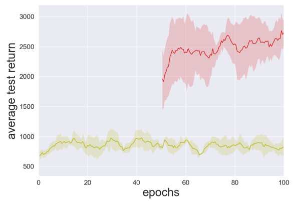
Figure 5.3: Augmented Returns versus Oracle Performance. All learning curves are for the Hopper-v2 environment. The x-axis ranges from 50 to 100 epochs since this comparison involves only BAIL. The results show that the augmentation heuristic typically achieves oracle-level performance.

Figure 5.4 compares BAIL with the algorithm that simply chooses the state-action pairs with the highest returns (without using an upper envelope). The learning curves show that the upper envelope is a critical component of BAIL.

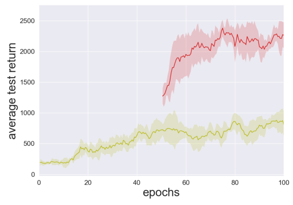
Figure 5.5 compares BAIL with the more naive scheme of using standard regression in place of an upper envelope. The learning curves show that the upper envelope is a critical component of BAIL.



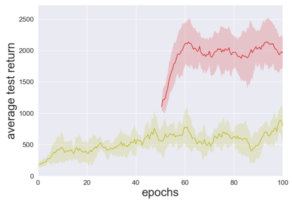
(a) Hopper $\sigma = 0.5$ 1st



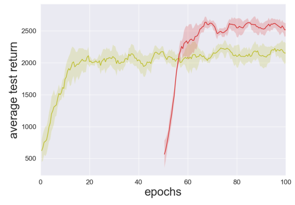
(b) Hopper $\sigma = 0.5$ 2nd



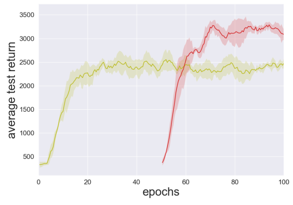
(c) Hopper $\sigma = 0.1$ 1st



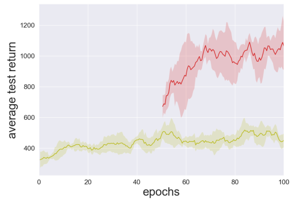
(d) Hopper $\sigma = 0.1$ 2nd



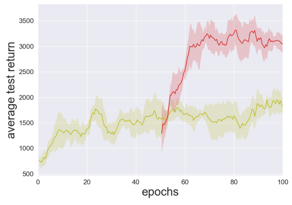
(e) Walker2d $\sigma = 0.5$ 1st



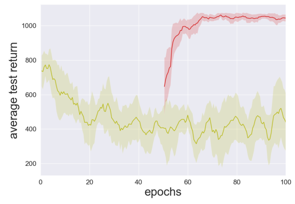
(f) Walker2d $\sigma = 0.5$ 2nd



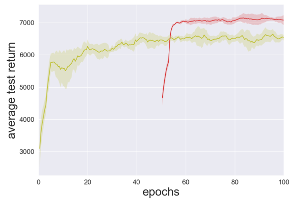
(g) Walker2d $\sigma = 0.1$ 1st



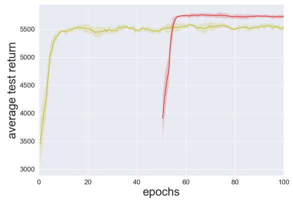
(h) Walker2d $\sigma = 0.1$ 2nd



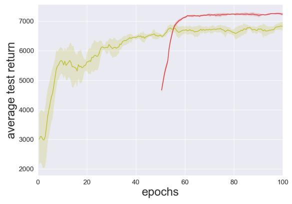
(i) HalfCheetah $\sigma = 0.5$ 1st



(j) HalfCheetah $\sigma = 0.5$ 2nd



(k) HalfCheetah $\sigma = 0.1$ 1st

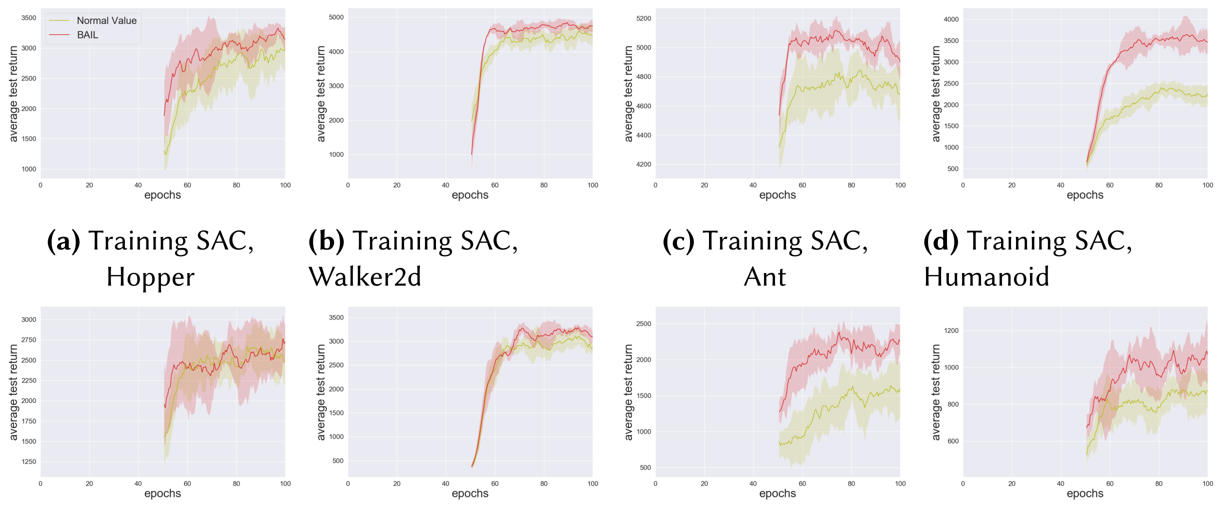


(l) HalfCheetah $\sigma = 0.1$ 2nd

Figure 5.4: Ablation study for data selection. The figure compares BAIL with the algorithm that simply chooses the state-action pairs with the highest returns (without using an upper envelope). The learning curves show that the upper envelope is critical components of BAIL.

5.3.4 Implementation Details

BAIL includes a regularization scheme to prevent over-fitting when generating the upper envelope. We refer to it as an "early stopping scheme" because the key idea is to return to the parameter values which gave the lowest validation error (see Section 7.8 of Goodfellow et al. [2016a]). In our implementation, we initialize two upper envelope networks with parameters ϕ and ϕ', where ϕ is trained using the penalty loss, and ϕ' records the parameters with the lowest validation error encountered so far. The procedure is done as follows: After every epoch, we calculate the validation loss L_ϕ as the penalty loss over all the data in the validation set $\mathcal{B}_v$. We compare this validation loss L_ϕ to $L_{\phi'}$, which is the minimum validation loss encountered so far



(a) Training SAC, Hopper **(b)** Training SAC, Walker2d **(c)** Training SAC, Ant **(d)** Training SAC, Humanoid

(e) Training DDPG $\sigma = 0.5$, Hopper **(f)** Training DDPG $\sigma = 0.5$, Walker2d **(g)** Training DDPG $\sigma = 0.1$, Hopper **(h)** Training DDPG $\sigma = 0.1$, Walker2d

Figure 5.5: Ablation study using standard regression instead of an upper envelope. The figure compares BAIL with the more naive scheme of using standard regression in place of an upper envelope. The learning curves show that the upper envelope is a critical component of BAIL.

(throughout the history of training). If $L_\phi < L_{\phi'}$, we set $\phi' \leftarrow \phi$. If $L_\phi > L_{\phi'}$, we count the number of consecutive times this occurs. The training parameters ϕ are returned to ϕ' once there are C consecutive times with $L_\phi > L_{\phi'}$. We use $C = 4$ in practice.

BAIL and Progressive BAIL use the same hyper-parameters except for the selection percentage p. Details are provided in Table 5.3. A common feature among all the batch DRL algorithms is that they have a policy neural network. BCQ and BEAR both have an architecture consisting of 400×300 hidden units with ReLU activation units. We use exactly the same network architecture for the policy network for BAIL and Progressive BAIL. For the IL-based algorithms, we also use this same policy network architecture. All algorithms considered in our experiments use the same learning rate of $1 \cdot 10^{-3}$ for the policy network, which is also the default in BCQ and BEAR.

When designing RL algorithms, it is desirable that they generalize over new, unforeseen environments and tasks. Therefore, consistent with common practice for online reinforcement learning [Schulman et al. 2015, 2017; Vuong et al. 2018; Lillicrap et al. 2019; Fujimoto et al. 2018b; Wang et al. 2020], when evaluating any given algorithm, we use the same hyper-parameters for all environments and all batches. The BCQ paper [Fujimoto et al. 2018a] also uses the same hyper-parameters for all experiments.

Alternatively, one could optimize the hyper-parameters for each environment separately. Not only is this not standard practice, but to make a fair comparison across all algorithms, this would require, for *each* of the five algorithms, performing a separate hyper-parameter search for *each* of the five environments.

In our experiments, we went the extra mile to make a fair comparison to other batch RL algorithms. We are therefore confident about properly using the authors' BCQ and BEAR code, and fairly reproducing MARWIL for the MuJoCo benchmark.

For BCQ, we use the authors' code and recommended hyper-parameters. In the BCQ paper, the "final buffer" batches are where the BCQ algorithm shines the most; therefore, included in our training batches are batches for which we used exactly the same "final buffer" experimental set-up. In our terminology, this corresponds to DDPG training batches with sigma = 0.5. Looking at the BCQ final-buffer results in Table 5.1, we see that they are consistent with the results in Figure 2a in the BCQ paper.

To ensure that we are running the BEAR code properly, we obtained a dataset directly from the BEAR authors and ran the BEAR algorithm with a specific set of hyper-parameters among their recommendations. Specifically, we used their version "0" with "use ensemble variance" set to False and employ Laplacian kernel. The dataset provided by the authors was for Hopper-v2 with mediocre performance. The performance we obtained is shown in Figure 5.6, which fully matches the Hopper-v2 case in Figure 3 in [Kumar et al. 2019a]. Also, we observed that for some of our batches, we obtained very similar results to what is shown in the BEAR paper.

The authors of MARWIL do not provide an open-source implementation of their algorithm. Furthermore, experiments in [Wang et al. 2018] are carried out on environments like HFO and TORCS which are considerably different from MuJoCo. We replicate all implementation details discussed in MARWIL, except that we use the same network architectures used for BCQ, BEAR

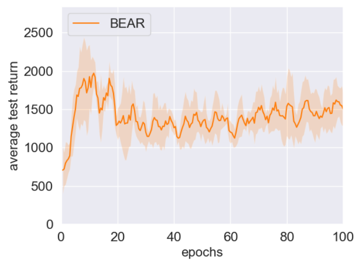


Figure 5.6: Our results when we apply BEAR to the authors' dataset. This figure matches Figure 3 in Kumar et al. [2019a].

and BAIL to ensure a fair comparison. We use the same augmentation heuristic for the returns as we use in BAIL. We use the recommended hyper-parameters given by the MARWIL authors.

To evaluate the performance of the current policy during training, we run ten episodes of test runs with the current policy and record the average of the returns. This is done with the same frequency for each algorithm considered in our experiments.

For a test episode, we sometimes encounter an error signal from the MuJoCo environment, and thus are not able to continue the episode. In these cases, we assign a zero value to the return for the terminated episode. In Tables 5.1 and 5.2, there are a few entries with zero mean and zero standard deviation. These zeros are due to repeatedly encountering this error signal for the test runs using different seeds, with each test run getting a zero value for the return. This happens for BEAR in several batches, which is likely because we are not using different hyper-parameters for each environment.

All experiments are run on Intel Xeon Gold 6248 CPU nodes, each job runs on a single CPU with base frequency of 2.50GHZ.

5.4 Conclusion

In conclusion, our experimental results show that (*i*) for the training data batches, BAIL is the clear winner, winning for 20 of 22 batches with a performance improvement of 42% over BCQ and 101% over BC; (*ii*) for the execution batches, vanilla BC does well with not much room for improvement, although BAIL and BCQ are almost as good and occasionally beat BC by a small amount; (*iii*) BAIL is computationally much faster than the Q-learning-based algorithms BCQ and BEAR.

The results in this chapter show that it is possible to achieve state-of-the art performance with a simple, computationally fast IL-based algorithm. BAIL is based on the notion of the "upper envelope of the data", which appears to be novel and may find applications in other machine-learning domains. One potential future research direction is to combine batch methods such as BAIL with exploration techniques to build robust online algorithms for better sample efficiency. Another potential direction is to develop methods that are more robust across different batches and hyperparameters and study what makes them robust. Such robustness can greatly improve computation time, and might be safer to work with when deployed to real-world systems.

Table 5.2: Performance of Five Batch DRL Algorithms for 40 different execution datasets.

Environment	Bail	BCQ	BEAR	BC	MARWIL
M $\sigma = 0$ Hopper B1	**1026 ± 0**	901 ± 132	4 ± 1	**1026 ± 0**	**1026 ± 0**
M $\sigma = 0$ Hopper B2	696 ± 233	805 ± 312	19 ± 23	**977 ± 0**	**977 ± 1**
M $\sigma = 0$ Walker B1	437 ± 20	**525 ± 45**	380 ± 194	444 ± 16	439 ± 17
M $\sigma = 0$ Walker B2	**500 ± 12**	**554 ± 29**	**546 ± 28**	489 ± 15	**504 ± 4**
M $\sigma = 0$ HC B1	**4057 ± 69**	**4255 ± 150**	**4470 ± 96**	**4032 ± 72**	**4073 ± 55**
M $\sigma = 0$ HC B2	**4013 ± 12**	**4438 ± 25**	**4395 ± 31**	**3998 ± 4**	**3999 ± 6**
M $\sigma = 0$ Ant B1	753 ± 9	**996 ± 52**	734 ± 43	730 ± 7	732 ± 11
M $\sigma = 0$ Ant B2	738 ± 4	**994 ± 12**	**988 ± 30**	708 ± 11	725 ± 7
M $\sigma = 0$ Humanoid B1	**4313 ± 139**	3108 ± 510	0 ± 0	**4507 ± 481**	**4521 ± 156**
M $\sigma = 0$ Humanoid B2	**4053 ± 252**	2906 ± 226	0 ± 0	**3994 ± 530**	**3940 ± 165**
M $\sigma = \sigma(s)$ Hopper B1	375 ± 52	881 ± 155	0 ± 0	**1026 ± 0**	**1026 ± 0**
M $\sigma = \sigma(s)$ Hopper B2	254 ± 102	**961 ± 25**	3 ± 7	**977 ± 0**	**977 ± 0**
M $\sigma = \sigma(s)$ Walker B1	384 ± 21	399 ± 21	**507 ± 7**	369 ± 10	359 ± 15
M $\sigma = \sigma(s)$ Walker B2	**512 ± 24**	**517 ± 19**	**515 ± 30**	**527 ± 12**	**532 ± 5**
M $\sigma = \sigma(s)$ HC B1	4744 ± 19	**5500 ± 12**	**5443 ± 21**	4415 ± 25	4439 ± 59
M $\sigma = \sigma(s)$ HC B2	4123 ± 19	**4712 ± 40**	**4824 ± 51**	3928 ± 18	3936 ± 18
M $\sigma = \sigma(s)$ Ant B1	790 ± 9	**1068 ± 12**	**1161 ± 32**	775 ± 7	774 ± 15
M $\sigma = \sigma(s)$ Ant B2	781 ± 6	**1089 ± 29**	**1150 ± 18**	768 ± 5	761 ± 6
M $\sigma = \sigma(s)$ Humanoid B1	1375 ± 387	489 ± 87	0 ± 0	**1947 ± 901**	**1963 ± 264**
M $\sigma = \sigma(s)$ Humanoid B2	1309 ± 372	816 ± 177	0 ± 0	**3021 ± 1042**	**2976 ± 241**
O $\sigma = 0$ Hopper B1	**2602 ± 5**	1976 ± 383	1904 ± 321	**2594 ± 8**	**2603 ± 4**
O $\sigma = 0$ Hopper B2	**3046 ± 34**	**3014 ± 47**	2202 ± 410	**3071 ± 10**	**3050 ± 22**
O $\sigma = 0$ Walker B1	**2735 ± 26**	2409 ± 235	877 ± 1077	**2646 ± 133**	**2691 ± 121**
O $\sigma = 0$ Walker B2	**3019 ± 6**	**3019 ± 45**	0 ± 0	**3014 ± 5**	**3013 ± 5**
O $\sigma = 0$ HC B1	**11265 ± 243**	10405 ± 275	1755 ± 1142	**11674 ± 90**	**11661 ± 49**
O $\sigma = 0$ HC B2	**11360 ± 265**	**10792 ± 209**	1139 ± 960	**11797 ± 29**	**11691 ± 96**
O $\sigma = 0$ Ant B1	**4901 ± 65**	**4646 ± 179**	1756 ± 2151	**4881 ± 74**	**4933 ± 74**
O $\sigma = 0$ Ant B2	**4975 ± 108**	**4734 ± 100**	0 ± 0	**5041 ± 29**	**4974 ± 52**
O $\sigma = 0$ Humanoid B1	4872 ± 895	4884 ± 641	0 ± 0	**5462 ± 124**	**5503 ± 1**
O $\sigma = 0$ Humanoid B2	**5320 ± 125**	**5362 ± 54**	0 ± 0	**5413 ± 64**	**5413 ± 29**
O $\sigma = \sigma(s)$ Hopper B1	2359 ± 153	**2650 ± 99**	1962 ± 300	1952 ± 85	2012 ± 101
O $\sigma = \sigma(s)$ Hopper B2	**2035 ± 217**	1678 ± 113	1461 ± 75	**2063 ± 95**	**2092 ± 100**
O $\sigma = \sigma(s)$ Walker B1	2834 ± 120	**3386 ± 196**	**3278 ± 128**	2024 ± 131	1987 ± 114
O $\sigma = \sigma(s)$ Walker B2	**3200 ± 16**	**3375 ± 12**	2100 ± 1715	**3091 ± 15**	**3090 ± 10**
O $\sigma = \sigma(s)$ HC B1	10258 ± 1255	**10928 ± 215**	694 ± 651	**11659 ± 75**	**11663 ± 44**
O $\sigma = \sigma(s)$ HC B2	**10882 ± 634**	**11755 ± 97**	1470 ± 1211	**11871 ± 57**	**11819 ± 78**
O $\sigma = \sigma(s)$ Ant B1	**4981 ± 91**	**4878 ± 117**	3462 ± 1740	**5000 ± 79**	**4992 ± 86**
O $\sigma = \sigma(s)$ Ant B2	**5067 ± 83**	**5054 ± 157**	0 ± 0	**5079 ± 55**	**5124 ± 47**
O $\sigma = \sigma(s)$ Humanoid B1	2129 ± 381	1715 ± 637	0 ± 0	**3514 ± 1195**	**3180 ± 503**
O $\sigma = \sigma(s)$ Humanoid B2	4328 ± 569	1970 ± 512	0 ± 0	**4875 ± 885**	**4772 ± 272**

Table 5.3: BAIL hyper-parameters

Parameter	Value
discount rate γ	0.99
horizon T	1000
training set size	$0.8 \cdot \lvert\mathcal{B}\rvert$
validation set size	$0.2 \cdot \lvert\mathcal{B}\rvert$
optimizer	Adam [Kingma and Ba 2014]
percentage $p\%$	30% for BAIL
	25% for Progressive BAIL
upper envelope network	
structure	128×128 hidden units, ReLU activation
learning rate	$3 \cdot 10^{-3}$
penalty loss coefficient K	1000
policy network	
structure	400×300 hidden units, ReLU activation
learning rate	$1 \cdot 10^{-3}$

6 | Conclusion and Future Directions

6.1 Conclusions

In this thesis, we have discussed various methods for improving sample efficiency in off-policy and offline deep reinforcement learning. Advances in theories and techniques, such as clipped double-Q learning, target policy smoothing, maximum entropy and non-uniform sampling [Fujimoto et al. 2018b; Haarnoja et al. 2018a; Hou et al. 2017], have allowed more effective training for off-policy algorithms. The introduction of extrapolation error in offline RL along with different regularizers to address the issue have shown some success in a wide-range of offline datasets [Fujimoto et al. 2018a; Fu et al. 2020; Kumar et al. 2020]. Driven by these research advances, we develop our own off-policy and offline algorithms aiming to further improve sample efficiency and asymptotic performance in the simulated robotic locomotion environments.

In Chapters 3 and 4, we focus on improving sample efficiency in off-policy DRL. Specifically, in Chapter 3, we first identify squashing exploration problem and show that the primary role of maximum entropy RL is to maintain satisfactory exploration in the presence of squashing exploration problem. We then introduce a simpler output normalization scheme for maximum entropy DRL algorithms. When combined with a novel non-uniform sampling scheme, our algorithm achieves higher sample efficiency and asymptotic performance in the MuJoCo benchmark environments.

In Chapter 4, to develop an algorithm which further improves sample efficiency and performance, we adopt a high update-to-data (UTD) ratio and address the overestimation bias issue using Q-ensembles during training. With extensive experiments and ablations, we show that our framework is robust and gives us an algorithm providing state-of-art performance at all stages of training. Our theoretical results also complement the experimental results, showing that the Q estimation bias can be controlled by either varying the ensemble size N or the keep parameter K.

In Chapter 5, we discussed the sample efficiency topic in the offline DRL setting. We introduce a novel notion of "upper envelope of the data" and develop our Imitation-Learning-based algorithm based on the notion. Our approach is computationally much faster than the Q-learning-based algorithms and achieves state-of-art performance for a wide range of offline datasets.

6.2 Future directions

We have shown that using a high update-to-data (UTD) ratio G combined with refined bias control can greatly improve sample efficiency in off-policy DRL [Chen et al. 2021; Wu et al. 2021]. When using a very high update-to-data (UTD) ratio in off-policy algorithms, during each training iteration, the agent is updating from a static replay buffer $G (G \gg 1)$ times before collecting new transitions, which resembles the offline training process. In our algorithm, AQE, we address the overestimation bias by adopting Q-ensembles and keeping several lowest values. One future direction is to combine regularization techniques from offline RL with a very high update-to-data ratio ($G \gg 1$) to develop novel off-policy algorithms. Another interesting future work direction is to investigate whether there are other critical factors (in addition to Q-bias control) that can allow us to further benefit from a high update-to-data ratio, and achieve even better sample efficiency with simple model-free methods.

We have discussed extrapolation error, introduced by the mismatch between the dataset and true state-action visitation of the current policy. Existing research work mainly focuses on training a policy to avoid out-of-distribution actions by applying critic penalty, policy regularization or using imitation-learning-based algorithms [Fujimoto et al. 2018a; Kumar et al. 2020; Chen et al. 2019]. A potential research direction is to study the effect of out-of-distribution states during policy evaluation. In particular, during policy evaluation, agent may visit out-of-distribution states which leads to huge performance drop. Keep taking in-distribution actions in the presence of out-of-distribution states may be problematic. One direction is to teach agents to avoid taking out-of-distribution actions as well as avoid visiting out-of-distribution states.

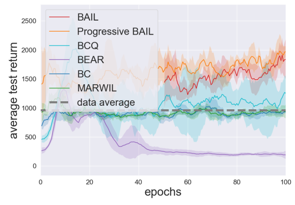
We recall from Chapter 5 that we have compared 5 offline RL algorithms in a wide range of MuJoCo offline datasets. Some algorithms perform really well for some of the offline datasets but fail for others. This is also commonly shown in other offline RL papers [Kostrikov et al. 2021a; Fu et al. 2020; Kostrikov et al. 2021b]. A potential research direction is to identify the reasons and patterns behind the variance in performance among the datasets.

A | Appendix

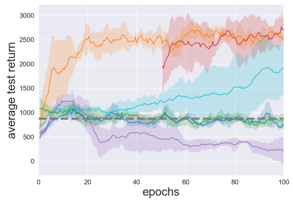
A.1 Supplementary Material for Chapter 5

A.1.1 Supplementary Figures

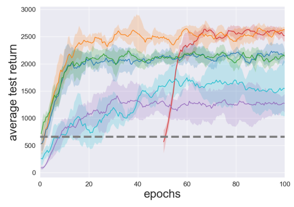
We present the learning curves for all 62 batches described in section 5.3.1 in this section.



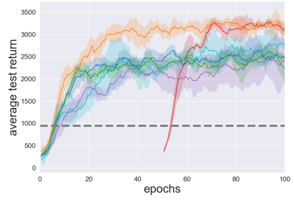
(a) Hopper, batch 1



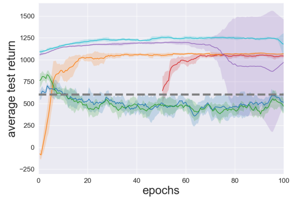
(b) Hopper, batch 2



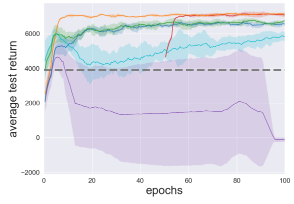
(c) Walker2d, batch 1



(d) Walker2d, batch 2

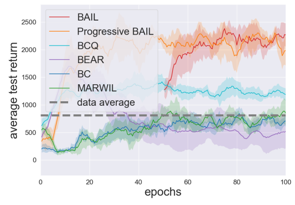


(e) HalfCheetah, batch 1

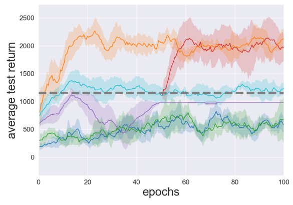


(f) HalfCheetah, batch 2

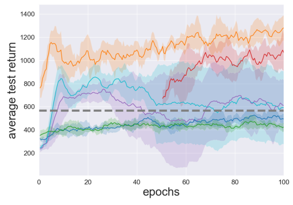
Figure A.1: Performance of batch DRL algorithms on DDPG training batches with $\sigma = 0.5$. The policy networks for all algorithms are trained for 100 epochs except BAIL, which is trained for 50 epochs after training the upper envelope for 50 epochs.



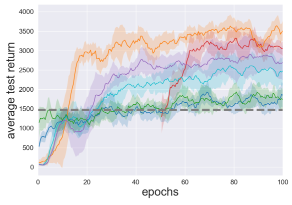
(a) Hopper, batch 1



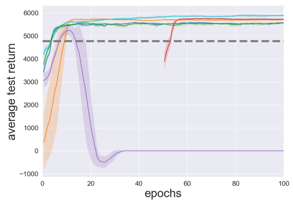
(b) Hopper, batch 2



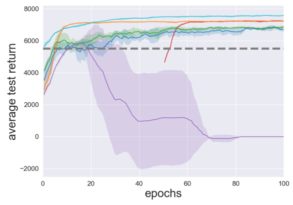
(c) Walker2d, batch 1



(d) Walker2d, batch 2

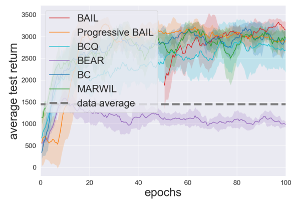


(e) HalfCheetah, batch 1

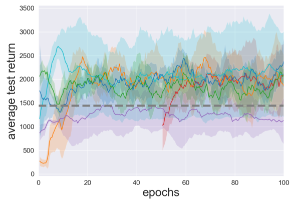


(f) HalfCheetah, batch 2

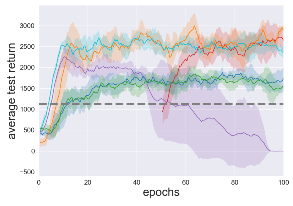
Figure A.2: Performance of batch DRL algorithms on DDPG training batches with $\sigma = 0.1$. The policy networks for all algorithms are trained for 100 epochs except BAIL, which is trained for 50 epochs after training the upper envelope for 50 epochs.



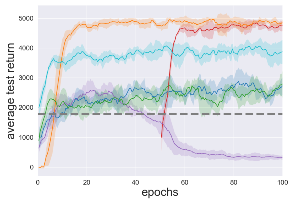
(a) Hopper, batch 1



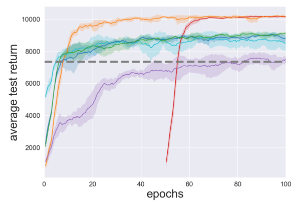
(b) Hopper, batch 2



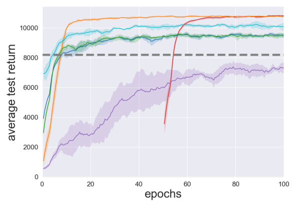
(c) Walker2d, batch 1



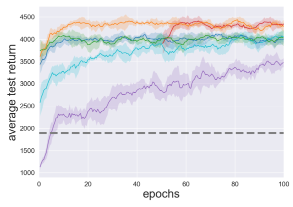
(d) Walker2d, batch 2



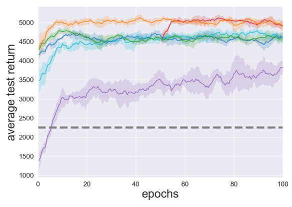
(e) HalfCheetah, batch 1



(f) HalfCheetah, batch 2

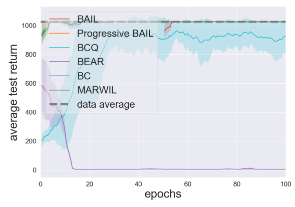


(g) Ant, batch 1

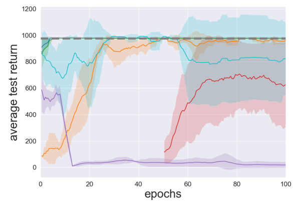


(h) Ant, batch 2

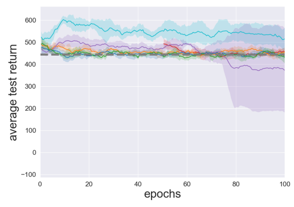
Figure A.3: Performance of batch DRL algorithms on SAC training batches. The policy networks for all algorithms are trained for 100 epochs except BAIL, which is trained for 50 epochs after training the upper envelope for 50 epochs.



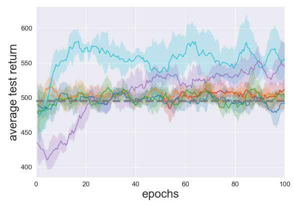
(a) Hopper, batch 1



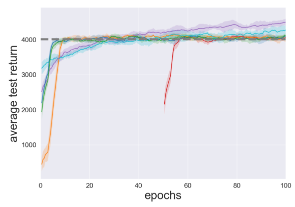
(b) Hopper, batch 2



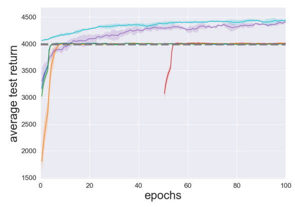
(c) Walker2d, batch 1



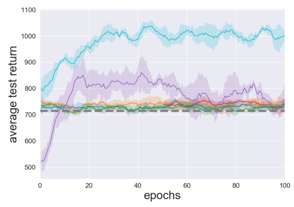
(d) Walker2d, batch 2



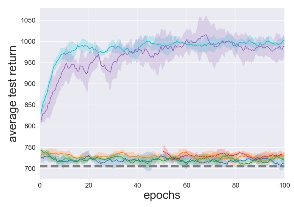
(e) HalfCheetah, batch 1



(f) HalfCheetah, batch 2

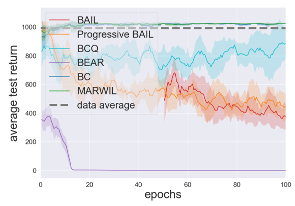


(g) Ant, batch 1

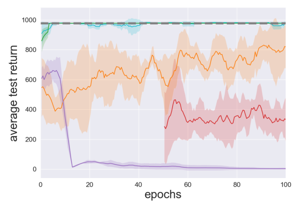


(h) Ant, batch 2

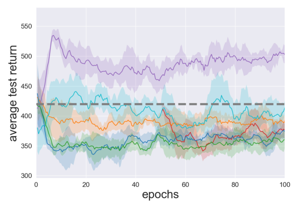
Figure A.4: Performance of batch DRL algorithms on SAC mediocre execution batches with $\sigma = 0$. The policy networks for all algorithms are trained for 100 epochs except BAIL, which is trained for 50 epochs after training the upper envelope for 50 epochs.



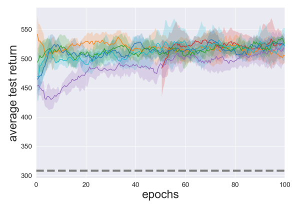
(a) Hopper, batch 1



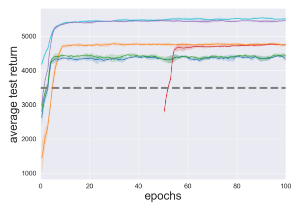
(b) Hopper, batch 2



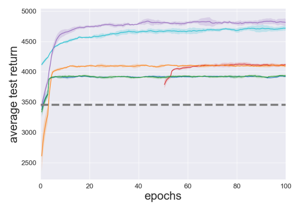
(c) Walker2d, batch 1



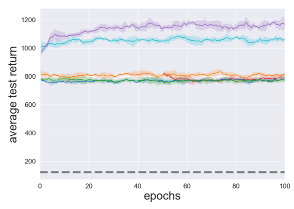
(d) Walker2d, batch 2



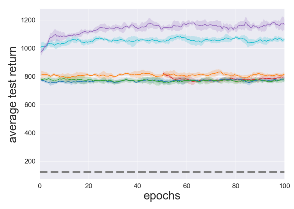
(e) HalfCheetah, batch 1



(f) HalfCheetah, batch 2

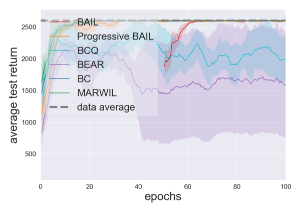


(g) Ant, batch 1

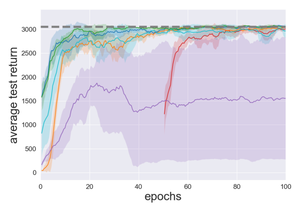


(h) Ant, batch 2

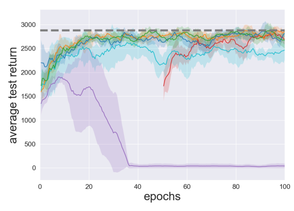
Figure A.5: Performance of batch DRL algorithms on SAC mediocre execution batches with $\sigma = \sigma(s)$. The policy networks for all algorithms are trained for 100 epochs except BAIL, which is trained for 50 epochs after training the upper envelope for 50 epochs.



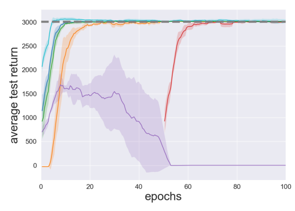
(a) Hopper, batch 1



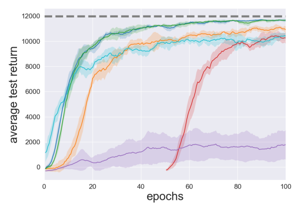
(b) Hopper, batch 2



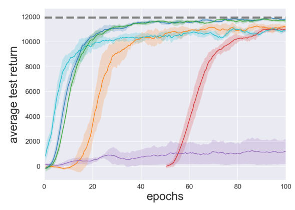
(c) Walker2d, batch 1



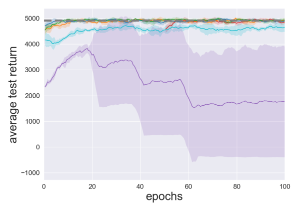
(d) Walker2d, batch 2



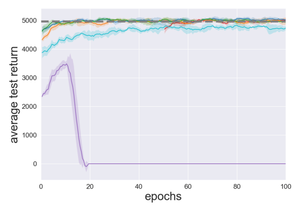
(e) HalfCheetah, batch 1



(f) HalfCheetah, batch 2

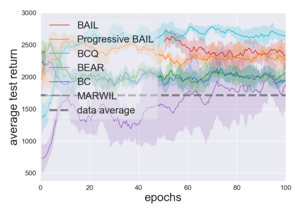


(g) Ant, batch 1

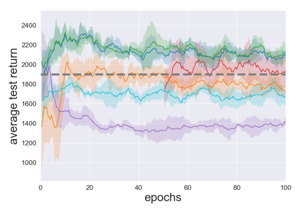


(h) Ant, batch 2

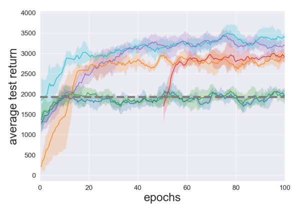
Figure A.6: Performance of batch DRL algorithms on SAC optimal execution batches with $\sigma = 0$. The policy networks for all algorithms are trained for 100 epochs except BAIL, which is trained for 50 epochs after training the upper envelope for 50 epochs.



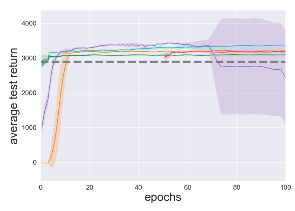
(a) Hopper, batch 1



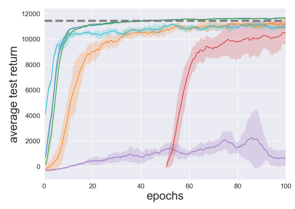
(b) Hopper, batch 2



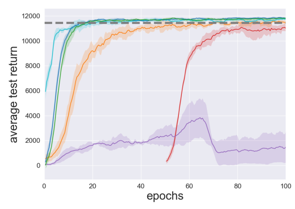
(c) Walker2d, batch 1



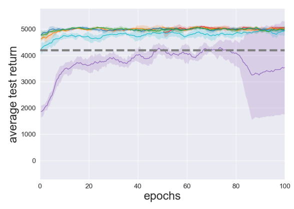
(d) Walker2d, batch 2



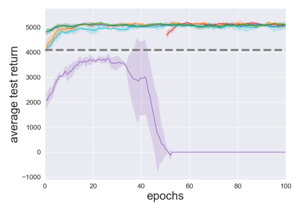
(e) HalfCheetah, batch 1



(f) HalfCheetah, batch 2

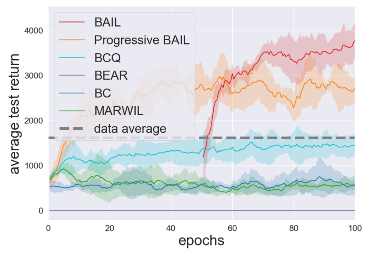


(g) Ant, batch 1

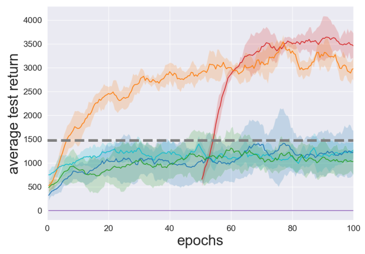


(h) Ant, batch 2

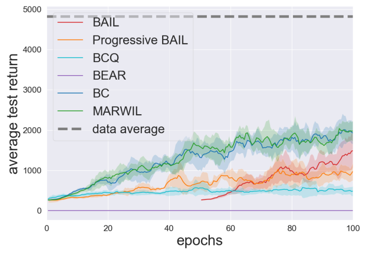
Figure A.7: Performance of batch DRL algorithms on SAC optimal execution batches with $\sigma = \sigma(s)$. The policy networks for all algorithms are trained for 100 epochs except BAIL, which is trained for 50 epochs after training the upper envelope for 50 epochs.



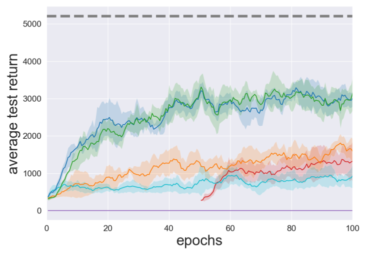
(a) training data, batch 1



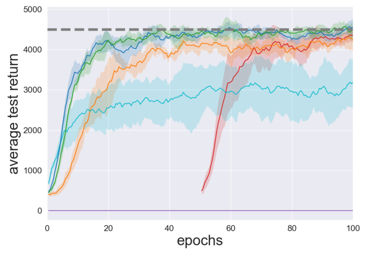
(b) training data, batch 2



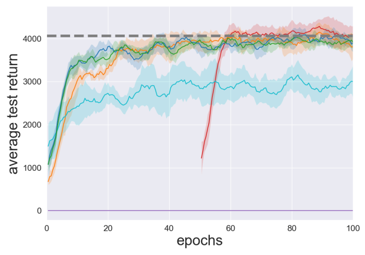
(c) mediocre $\sigma = \sigma(s)$, batch 1



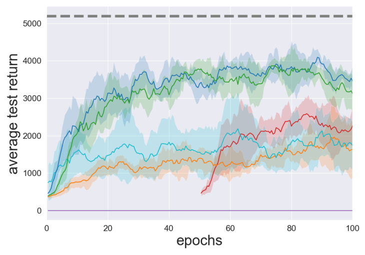
(d) mediocre $\sigma = \sigma(s)$, batch 2



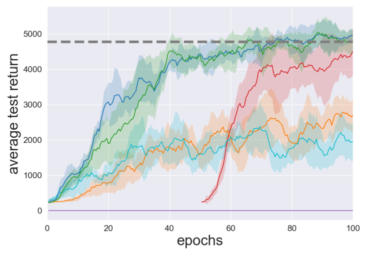
(e) mediocre $\sigma = 0$, batch 1



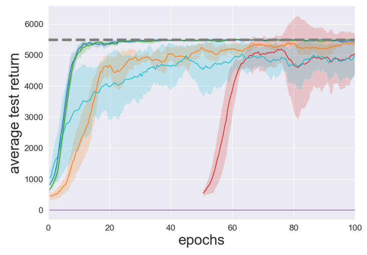
(f) mediocre $\sigma = 0$, batch 2



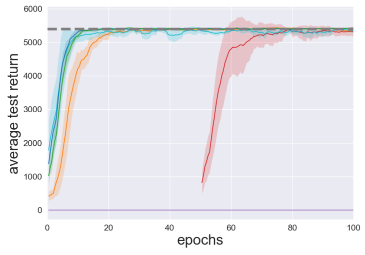
(g) optimal $\sigma = \sigma(s)$, batch 1



(h) optimal $\sigma = \sigma(s)$, batch 2

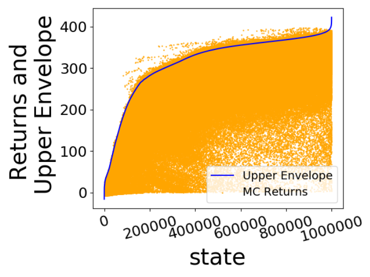


(i) optimal $\sigma = 0$, batch 1

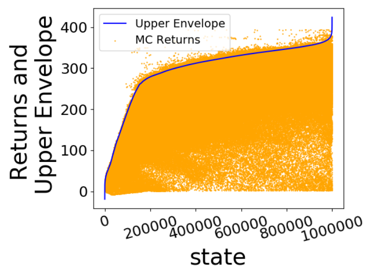


(j) optimal $\sigma = 0$, batch 2

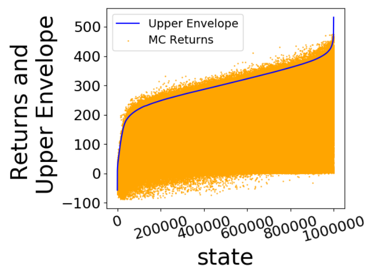
Figure A.8: Performance of batch DRL algorithms with the Humanoid-v2 environment. All batches are obtained with SAC.



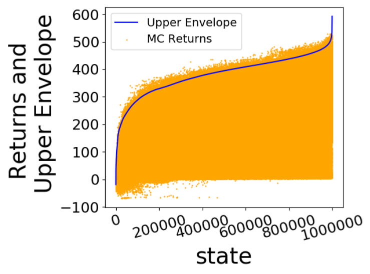
(a) Hopper $\sigma = 0.5$ 1st



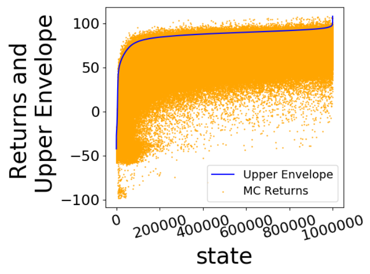
(b) Hopper $\sigma = 0.5$ 2nd



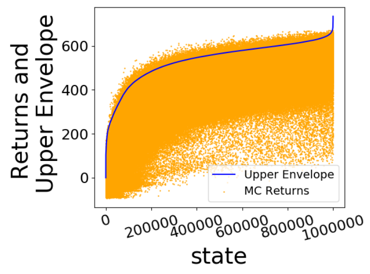
(c) Walker2d $\sigma = 0.5$ 1st



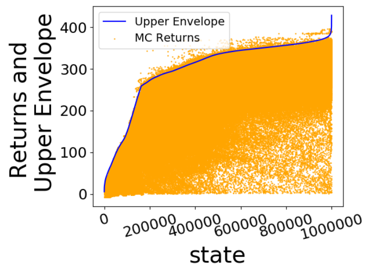
(d) Walker2d $\sigma = 0.5$ 2nd



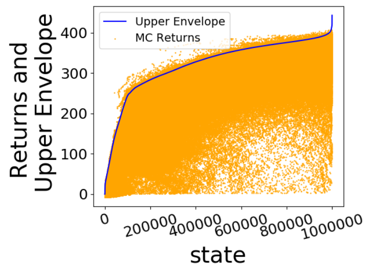
(e) HalfCheetah $\sigma = 0.5$ 1st



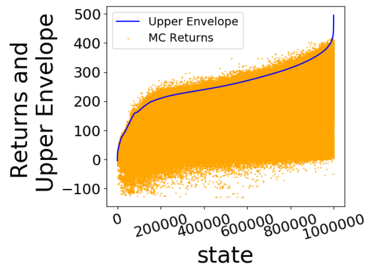
(f) HalfCheetah $\sigma = 0.5$ 2nd



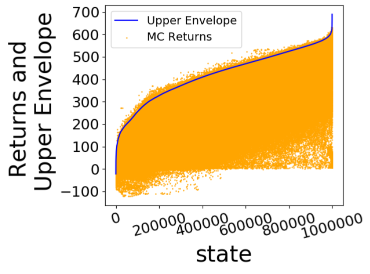
(g) Hopper $\sigma = 0.1$ 1st



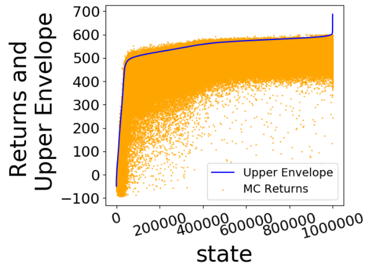
(h) Hopper $\sigma = 0.1$ 2nd



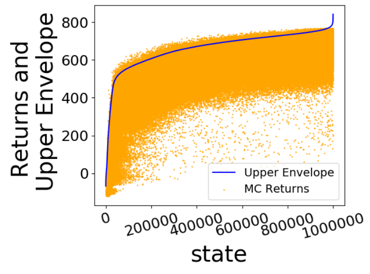
(i) Walker2d $\sigma = 0.1$ 1st



(j) Walker2d $\sigma = 0.1$ 2nd



(k) HalfCheetah $\sigma = 0.1$ 1st



(l) HalfCheetah $\sigma = 0.1$ 2nd

Figure A.9: Typical Upper Envelopes for BAIL. For each figure, states are ordered from lowest $V(s_i)$ upper envelope value to highest. Thus the upper envelope curve is monotonically increasing. Each curve is trained with one million returns, shown with the orange dots. Note that the upper envelope lies above most data points but not all data points.

Bibliography

Agarwal, R., Schuurmans, D., and Norouzi, M. (2020). An optimistic perspective on offline reinforcement learning. In *International Conference on Machine Learning*.

Ahmed, Z., Le Roux, N., Norouzi, M., and Schuurmans, D. (2019). Understanding the impact of entropy on policy optimization. In *International Conference on Machine Learning*, pages 151–160.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, O. P., and Zaremba, W. (2017). Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058.

Anschel, O., Baram, N., and Shimkin, N. (2017). Averaged-DQN: Variance reduction and stabilization for deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, pages 176–185.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866.

Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 449–458. PMLR.

Bellemare, M. G., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 1479–1487, Red Hook, NY, USA. Curran Associates Inc.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-dynamic programming*, volume 5. Athena Scientific Belmont, MA.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

Chen, X., Wang, C., Zhou, Z., and Ross, K. W. (2021). Randomized ensembled double q-learning: Learning fast without a model. In *International Conference on Learning Representations*.

Chen, X., Zhou, Z., Wang, Z., Wang, C., Wu, Y., Deng, Q., and Ross, K. W. (2019). BAIL: best-action imitation learning for batch deep reinforcement learning. *CoRR*, abs/1910.12179.

Chou, P.-W., Maturana, D., and Scherer, S. (2017). Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 834–843. JMLR. org.

Dabney, W., Ostrovski, G., Silver, D., and Munos, R. (2018a). Implicit quantile networks for distributional reinforcement learning. In Dy, J. G. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pages 1104–1113.

Dabney, W., Rowland, M., Bellemare, M. G., and Munos, R. (2018b). Distributional reinforcement learning with quantile regression. In McIlraith, S. A. and Weinberger, K. Q., editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*.

Dalal, G., Dvijotham, K., Vecerik, M., Hester, T., Paduraru, C., and Tassa, Y. (2018). Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757*.

De Bruin, T., Kober, J., Tuyls, K., and Babuška, R. (2015). The importance of experience replay database composition in deep reinforcement learning. In *Deep reinforcement learning workshop, NIPS*.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338.

Dulac-Arnold, G., Mankowitz, D. J., and Hester, T. (2019). Challenges of real-world reinforcement learning. *CoRR*, abs/1904.12901.

Eisenach, C., Yang, H., Liu, J., and Liu, H. (2018). Marginal policy gradients: A unified family of estimators for bounded action spaces with applications. *arXiv preprint arXiv:1806.05134*.

Ernst, D., Geurts, P., and Wehenkel, L. (2005a). Tree-based batch mode reinforcement learning. *J. Mach. Learn. Res.*, 6:503–556.

Ernst, D., Geurts, P., and Wehenkel, L. (2005b). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(Apr):503–556.

Farajtabar, M., Chow, Y., and Ghavamzadeh, M. (2018). More robust doubly robust off-policy evaluation. *arXiv preprint arXiv:1802.03493*.

Faußer, S. and Schwenker, F. (2015). Neural network ensembles in reinforcement learning. *Neural Process. Lett.*, page 55–69.

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135.

Fu, J., Kumar, A., Nachum, O., Tucker, G., and Levine, S. (2020). D4RL: datasets for deep data-driven reinforcement learning. *CoRR*, abs/2004.07219.

Fu, J., Kumar, A., Soh, M., and Levine, S. (2019). Diagnosing bottlenecks in deep q-learning algorithms. *arXiv preprint arXiv:1902.10250*.

Fujimoto, S., Meger, D., and Precup, D. (2018a). Off-policy deep reinforcement learning without exploration. *CoRR*, abs/1812.02900.

Fujimoto, S., van Hoof, H., and Meger, D. (2018b). Addressing function approximation error in actor-critic methods.

Fujita, Y. and Maeda, S.-i. (2018). Clipped action policy gradient. *arXiv preprint arXiv:1802.07564*.

Ghasemipour, S. K. S., Schuurmans, D., and Gu, S. S. (2020). Emaq: Expected-max q-learning operator for simple yet effective offline and online RL. *CoRR*, abs/2007.11091.

Goodfellow, I., Bengio, Y., and Courville, A. (2016a). *Deep Learning*, chapter Regularization for Deep Learning. MIT Press. http://www.deeplearningbook.org.

Goodfellow, I. J., Bengio, Y., and Courville, A. (2016b). *Deep Learning*. MIT Press, Cambridge, MA, USA. http://www.deeplearningbook.org.

Gu, S., Lillicrap, T., Sutskever, I., and Levine, S. (2016). Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838.

Haarnoja, T., Tang, H., Abbeel, P., and Levine, S. (2017). Reinforcement learning with deep energy-based policies. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1352–1361. JMLR. org.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018a). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*.

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. (2018b). Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905.*

Hasselt, H. (2010). Double q-learning. In Lafferty, J., Williams, C., Shawe-Taylor, J., Zemel, R., and Culotta, A., editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc.

Hasselt, H. v., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press.

Hausknecht, M. and Stone, P. (2015). Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143.*

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence.*

Hernandez, D. and Brown, T. B. (2020). Measuring the algorithmic efficiency of neural networks. *CoRR*, abs/2005.04305.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence.*

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. (2018). Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933.*

Hou, Y., Liu, L., Wei, Q., Xu, X., and Chen, C. (2017). A novel ddpg method with prioritized experience replay. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 316–321. IEEE.

Irpan, A., Rao, K., Bousmalis, K., Harris, C., Ibarz, J., and Levine, S. (2019). Off-policy evaluation via off-policy classification. *arXiv preprint arXiv:1906.01624*.

Islam, R., Henderson, P., Gomrokchi, M., and Precup, D. (2017). Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*.

Janner, M., Fu, J., Zhang, M., and Levine, S. (2019). When to trust your model: Model-based policy optimization. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*.

Jaques, N., Ghandeharioun, A., Shen, J. H., Ferguson, C., Lapedriza, A., Jones, N., Gu, S., and Picard, R. (2019). Way off-policy batch deep reinforcement learning of implicit human preferences in dialog. *arXiv preprint arXiv:1907.00456*.

Jiang, N. and Li, L. (2015). Doubly robust off-policy value evaluation for reinforcement learning. *arXiv preprint arXiv:1511.03722*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kingma, D. P. and Welling, M. (2014). Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

Kostrikov, I., Nair, A., and Levine, S. (2021a). Offline reinforcement learning with implicit q-learning. *CoRR*, abs/2110.06169.

Kostrikov, I., Tompson, J., Fergus, R., and Nachum, O. (2021b). Offline reinforcement learning with fisher divergence critic regularization. *CoRR*, abs/2103.08050.

Kumar, A., Fu, J., Tucker, G., and Levine, S. (2019a). Stabilizing off-policy q-learning via bootstrapping error reduction. *arXiv preprint arXiv:1906.00949*.

Kumar, A., Fu, J., Tucker, G., and Levine, S. (2019b). Stabilizing off-policy q-learning via bootstrapping error reduction. *CoRR*, abs/1906.00949.

Kumar, A., Zhou, A., Tucker, G., and Levine, S. (2020). Conservative q-learning for offline reinforcement learning. *CoRR*, abs/2006.04779.

Kuznetsov, A., Shvechikov, P., Grishin, A., and Vetrov, D. (2020). Controlling overestimation bias with truncated mixture of continuous distributional quantile critics. In *Proceedings of the 37th International Conference on Machine Learning*, pages 5556–5566.

Lan, Q., Pan, Y., Fyshe, A., and White, M. (2020a). Maxmin q-learning: Controlling the estimation bias of q-learning. *CoRR*, abs/2002.06487.

Lan, Q., Pan, Y., Fyshe, A., and White, M. (2020b). Maxmin q-learning: Controlling the estimation bias of q-learning. In *8th International Conference on Learning Representations*.

Lange, S., Gabel, T., and Riedmiller, M. (2012). Batch reinforcement learning. In *Reinforcement learning*, pages 45–73. Springer.

Langlois, E., Zhang, S., Zhang, G., Abbeel, P., and Ba, J. (2019). Benchmarking model-based reinforcement learning. *arXiv preprint arXiv:1907.02057*.

Lee, K., Laskin, M., Srinivas, A., and Abbeel, P. (2021). SUNRISE: A simple unified framework for ensemble learning in deep reinforcement learning. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021*, volume 139, pages 6131–6141.

Lee, S., Purushwalkam, S., Cogswell, M., Crandall, D. J., and Batra, D. (2015). Why M heads are better than one: Training a diverse ensemble of deep networks. *CoRR*.

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

Levine, S. and Koltun, V. (2013). Guided policy search. In *International Conference on Machine Learning*, pages 1–9.

Levine, S., Kumar, A., Tucker, G., and Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *CoRR*, abs/2005.01643.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2019). Continuous control with deep reinforcement learning.

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

Luenberger, D. G. and Ye, Y. (2008). *Linear and Nonlinear Programming*, chapter Penalty and Barrier Methods. Springer.

McClelland, J. L., McNaughton, B. L., and O'reilly, R. C. (1995). Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102(3):419.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Nachum, O., Norouzi, M., Xu, K., and Schuurmans, D. (2017). Bridging the gap between value and policy based reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2775–2785.

Novati, G. and Koumoutsakos, P. (2018). Remember and forget for experience replay. *arXiv preprint arXiv:1807.05827*.

Peng, X. B., Kumar, A., Zhang, G., and Levine, S. (2019). Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. *arXiv preprint arXiv:1910.00177*.

Puterman, M. L. (2014). *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Ratcliff, R. (1990). Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97(2):285.

Rawlik, K., Toussaint, M., and Vijayakumar, S. (2013). On stochastic optimal control and reinforcement learning by approximate inference. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

Robins, A. (1995). Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146.

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Schulze, C. and Schulze, M. (2018). Vizdoom: Drqn with prioritized experience replay, double-q learning and snapshot ensembling. In *Proceedings of SAI Intelligent Systems Conference*, pages 1–17. Springer.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

Sohn, K., Lee, H., and Yan, X. (2015). Learning structured output representation using deep conditional generative models. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Strehl, A., Langford, J., Li, L., and Kakade, S. M. (2010). Learning from logged implicit exploration data. In *Advances in Neural Information Processing Systems*, pages 2217–2225.

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

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction.* The MIT Press, second edition.

Swaminathan, A. and Joachims, T. (2015). Batch learning from logged bandit feedback through counterfactual risk minimization. *Journal of Machine Learning Research*, 16(1):1731–1755.

Thomas, P. and Brunskill, E. (2016). Data-efficient off-policy policy evaluation for reinforcement learning. In *International Conference on Machine Learning*, pages 2139–2148.

Thrun, S. and Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, pages 255–263. Hillsdale, NJ.

Todorov, E. (2008). General duality between optimal control and estimation. In *2008 47th IEEE Conference on Decision and Control*, pages 4286–4292. IEEE.

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*, pages 5026–5033.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T. P., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5.

Vuong, Q., Zhang, Y., and Ross, K. W. (2018). Supervised policy update for deep reinforcement learning. *arXiv preprint arXiv:1805.11706*.

Wang, C., Wu, Y., Vuong, Q., and Ross, K. (2020). Striving for simplicity and performance in off-policy drl: Output normalization and non-uniform sampling. *ICML*.

Wang, Q., Xiong, J., Han, L., Liu, H., Zhang, T., et al. (2018). Exponentially weighted imitation learning for batched historical data. In *Advances in Neural Information Processing Systems*, pages 6288–6297.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016). Sample efficient actor-critic with experience replay. *CoRR*, abs/1611.01224.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Publication.

Wu, Y., Chen, X., Wang, C., Zhang, Y., Zhou, Z., and Ross, K. W. (2021). Aggressive q-learning with ensembles: Achieving both high sample efficiency and high asymptotic performance. *CoRR*, abs/2111.09159.

Wu, Y., Tucker, G., and Nachum, O. (2019). Behavior regularized offline reinforcement learning. *CoRR*, abs/1911.11361.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2017). Understanding deep learning requires rethinking generalization. *ICLR*.

Zhao, R. and Tresp, V. (2019). Curiosity-driven experience prioritization via density estimation. *arXiv preprint arXiv:1902.08039*.

Ziebart, B. D. (2010). *Modeling purposeful adaptive behavior with the principle of maximum causal entropy*. PhD thesis, figshare.

Ziebart, B. D., Maas, A., Bagnell, J. A., and Dey, A. K. (2008). Maximum entropy inverse reinforcement learning.