ShinRL: A Library for Evaluating RL Algorithms from Theoretical and Practical Perspectives

Toshinori Kitamura*
Nara Institute of Science and Technology
Nara, Japan
kitamura.toshinori.kt6@is.naist.jp

Ryo Yonetani
OMRON SINIC X
Tokyo, Japan
ryo.yonetani@sinicx.com

Abstract

We present *ShinRL*, an open-source library specialized for the evaluation of reinforcement learning (RL) algorithms from both theoretical and practical perspectives[1]. Existing RL libraries typically allow users to evaluate practical performances of deep RL algorithms through returns. Nevertheless, these libraries are not necessarily useful for analyzing if the algorithms perform as theoretically expected, such as if Q learning really achieves the optimal Q function. In contrast, ShinRL provides an RL environment interface that can compute metrics for delving into the behaviors of RL algorithms, such as the gap between learned and the optimal Q values and state visitation frequencies. In addition, we introduce a solver interface for evaluating both theoretically justified algorithms (*e.g.*, dynamic programming and tabular RL) and practically effective ones (*i.e.*, deep RL, typically with some additional extensions and regularizations) in a consistent fashion. As a case study, we show that how combining these two features of ShinRL makes it easier to analyze the behavior of deep Q learning. Furthermore, we demonstrate that ShinRL can be used to empirically validate recent theoretical findings such as the effect of KL regularization for value iteration [Kozuno et al., 2019] and for deep Q learning [Vieillard et al., 2020a], and the robustness of entropy-regularized policies to adversarial rewards [Husain et al., 2021]. The ShinRL source code can be found on GitHub: https://github.com/omron-sinicx/ShinRL.

1 Introduction

Reinforcement learning (RL) [Sutton and Barto, 2018] has historically been, and still is, a very active topic in machine learning research. Recent years have particularly seen remarkable progress in research on deep RL, where highly-expressive neural networks are used to approximate policy or Q functions to enable complex sequential decision making. Due to its advantages in dealing with high-dimensional state spaces and learning policies that are generalizable to unseen testing environments, the effectiveness of deep RL has been confirmed in a variety of practical applications, such as robot control [Kober et al., 2013], game AI [Mnih et al., 2015], and economics [Zheng et al., 2020], to name a few.

In parallel with research on deep RL for practical tasks, there has been increasing attention paid to efforts to clarify its theoretical basis. Indeed, some state-of-the-art deep RL algorithms can be viewed as an extension of theoretical foundations of RL such as tabular RL (*i.e.*, no function approximation) and dynamic programming (DP; no exploration while assuming that the complete specification

*Work done as an intern at OMRON SINIC X.

[1]TK devised the main conceptual idea, developed the library, and conducted all the experiments presented in the paper. RY aided in shaping the research and worked in collaboration with TK to write the manuscript.

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

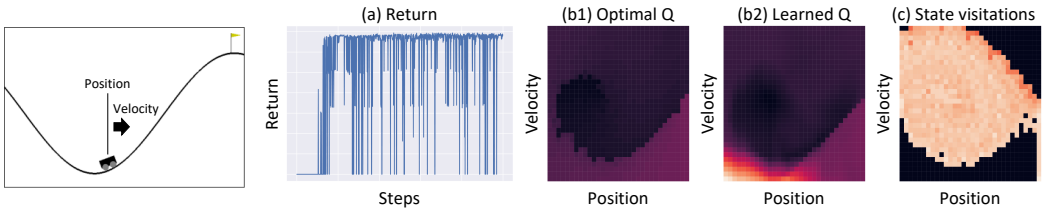

Figure 1: **Analyzing DQL Results on MountainCar using ShinRL.** (a) Return plot; (b1) Visualization of the optimal Q values, (b2) learned Q values, and (c) state visitation frequency.

about state transitions is given). Concrete examples of such correspondences between theoretically justified and practically effective algorithms include: from soft Q learning [Haarnoja et al., 2017] to soft actor-critic (SAC) [Haarnoja et al., 2018], from safe policy iteration [Pirotta et al., 2013] to trust-region policy optimization (TRPO) [Schulman et al., 2015], and from conservative value iteration (CVI) [Kozuno et al., 2019] to Munchausen Deep Q learning [Vieillard et al., 2020a]. In order to better understand a new deep RL algorithm that has been developed, it is critical to identify its theoretical foundation and validate if it works theoretically as expected, typically under a reasonably simplified setting.

To this end, we argue that one indispensable contribution is the development of open-source libraries that can evaluate RL algorithms from both theoretical and practical perspectives in a principled fashion. Despite many RL libraries have been developed so far [Fujita et al., 2021, Castro et al., 2018, Liang et al., 2018], they typically support evaluations of deep RL algorithms only through returns (*i.e.*, cumulative rewards) sampled from episodes. While such evaluations could allow users to systematically compare performances across methods (*e.g.*, Engstrom et al. [2020], Ceron and Castro [2021]), sampled returns are not necessarily useful for assessing if the methods work as theoretically expected. As a motivating example, suppose a scenario where a user performs a deep Q learning (DQL) [Mnih et al., 2015] on the MountainCar environment [Brockman et al., 2016]. The user can utilize existing libraries to implement such experiments easily and confirm that the trained network received a high return as shown in Fig. 1(a). However, *does this empirical success mean that the network really achieved the optimal Q function?* This is a simple but fundamental question to validate the theoretical expectation that the original (*i.e.*, tabular) Q learning has, which is nonetheless hard to answer for deep RL with non-linear function approximation, even empirically from the returns alone.

Motivated by the observations above, we develop a new RL library named *ShinRL*. At its core, we introduce an RL environment interface that can compute a variety of metrics such as optimal and learned Q functions (Fig. 1(b1)(b2)) and state visitation frequency (Fig. 1(c)), which are crucial for analyzing the behavior of RL algorithms but are not currently supported in existing libraries. Additionally, ShinRL provides an RL solver interface to evaluate both theoretically-justified and practical RL algorithms in a consistent fashion. By using this interface, users can easily ablate various extensions from developed RL algorithms, such as by removing function approximation and exploration, to empirically evaluate their theoretically-justified variants. ShinRL is implemented in a standard PyTorch and can be used without expensive computational resources such as high-end CPUs and GPUs to empirical validate theoretical results that are typically confirmed in reasonably simple environments (eg, Vieillard et al. [2020b] and Bellemare et al. [2016]). Nevertheless, as its main components are built on top of OpenAI Gym [Brockman et al., 2016], algorithms once implemented can be immediately available for practical evaluations, such as the ones using Atari [Bellemare et al., 2013] with few modifications.

In this paper, we overview the main features of ShinRL and present how they work in practice. Specifically, we first use ShinRL to effectively analyze the behavior of DQL, by clearly visualizing the effects of exploration, function approximation, and more advanced techniques such as double Q learning [Hasselt, 2010, Van Hasselt et al., 2016]. Furthermore, we demonstrate how ShinRL can be used to empirically validate recent theoretical findings in a systematic fashion, such as the effect of KL regularization for value iteration [Kozuno et al., 2019] and for DQL [Vieillard et al., 2020a], and the robustness of entropy-regularized policies to adversarial rewards [Husain et al., 2021].

2 Background

2.1 Preliminaries

Throughout this paper, we consider an infinite-horizon discounted Markov decision process (MDP) represented by a the tuple $\{\mathcal{S}, \mathcal{A}, P, r, \gamma\}$, where $\mathcal{S}$ is a finite state space, $\mathcal{A}$ is a finite set of actions, $P(s'|s,a)$ is a Markovian transition kernel (where $s, s' \in \mathcal{S}, a \in \mathcal{A}$), $r \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ is a reward function, and $\gamma \in (0,1)$ is a discount factor. The objective of RL is to find the optimal policy π_* that maximizes the discounted return (*i.e.*, cumulative reward) given by: $\pi_* = \mathrm{argmax}_\pi \mathbb{E}_\pi \left[\sum_{t=0}^\infty \gamma^t r\left(S_t, A_t\right)\right]$ where $\mathbb{E}_\pi$ is the expectation over all trajectories induced by policy π For a policy π, the Q function is defined as $Q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^\infty \gamma^t r(S_t, A_t) | S_0 = s, A_0 = a\right]$, the state visitation frequency is defined by $d_\pi(s) = (1-\gamma)\sum_{t=0}^\infty \gamma^t P\left(S_t = s|\pi\right)$. Following Vieillard et al. [2020a], we introduce the component-wise dot product notation $\langle f_1, f_2\rangle = \left(\sum_a f_1(s,a) f_2(s,a)\right)_s \in \mathbb{R}^{\mathcal{S}}$ for some functions $f_1, f_2 \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$. With this, the expectation of a Q function over a policy, the V function, can be expressed simply as $V(s) = \langle \pi, Q\rangle(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}[Q(s,a)]$. Further, we introduce another form to describe state transitions: $Pv = \left(\sum_{s'} P(s'|s,a) v(s')\right)_{s,a} \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ for $v \in \mathbb{R}^{\mathcal{S}}$.

2.2 Dynamic programming, and its extensions to practical RL algorithms

Classical approaches based on dynamic programming (DP), such as value iteration (VI) and policy iteration (PI), aim to find the optimal Q function as the optimal policy can easily be derived from it. In VI, Q function $Q \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ is iteratively updated by applying a Bellman backup: $Q \leftarrow r + \gamma P \max_a Q$, which is guaranteed to reach the optimality as its unique fixed point is Q_*. PI, on the other hand, consists of the following two steps: policy evaluation and policy improvement. The policy evaluation step applies the expected Bellman backup to the Q function: $Q \leftarrow r + \gamma P\langle \pi, Q\rangle$ where its unique fixed point is Q_π. The policy improvement step updates the policy with the Q function as follows: $\pi \leftarrow \mathrm{argmax}_\pi \langle \pi, Q\rangle$. Alternating these steps leads to the optimal Q function Q_*.

Unlike DP-based approaches, RL algorithms typically assume that a state-transition kernel and a reward function are unknown. To achieve the optimal policy or the optimal Q function, they instead require transition samples (s, a, s', r) collected by interacting with the MDP. Nevertheless, many of the RL algorithms are derived from DP. For example, Q-learning [Watkins and Dayan, 1992] can be seen as a variant of VI with exploration, while actor-critic method [Sutton et al., 2000] is a PI variant with exploration and function approximation of Q and π.

Many other deep RL algorithms have also been developed by extending DP. Approximate dynamic programming (ADP) is a framework to theoretically analyze RL algorithms using a DP update scheme [Munos and Szepesvári, 2008, Scherrer et al., 2015]. Specifically, in the ADP framework, the exploration and function approximation are "approximated" as an estimation error, allowing us to analyze how the error propagates to the converged policy. Doing so has revealed that VI and PI are weak to such errors [Munos and Szepesvári, 2008, Scherrer et al., 2015], which further explains the instability of recent deep Q learning algorithms [Mnih et al., 2015, Lillicrap et al., 2015, Fujimoto et al., 2018]. Some studies have then demonstrated the effectiveness of KL regularization against the error [Azar et al., 2012, Ghavamzadeh et al., 2011, Bellemare et al., 2016, Vieillard et al., 2020b, Kozuno et al., 2019], which led to recent KL-regularized deep RL algorithms [Schulman et al., 2015, Vieillard et al., 2020c,a]. Our main motivation is to develop an open-source library that allows users to reproduce and further explore such connections from theoretical results to practical algorithms.

3 ShinRL

As summarized in Fig. 2, ShinRL consists of two main modules: `ShinEnv` as an interface to implement environments modeled by the MDP and `Solver` as an interface for solving the RL tasks (*i.e.*, finding the optimal policy) on the environments with specified algorithms. In order to maximize the simplicity and flexibility of the library, we keep the number of main modules as low as possible in this way, while also implementing some basic RL necessities such as replay buffers, exploration strategies, and samplers, partially by including external libraries such as `cpprb` [Yamada, 2019]. Using `ShinEnv` and `Solver` in combination gives users the ability to evaluate deep RL algorithms as well as their tabular and DP variants through the same interface, making it possible to empirically

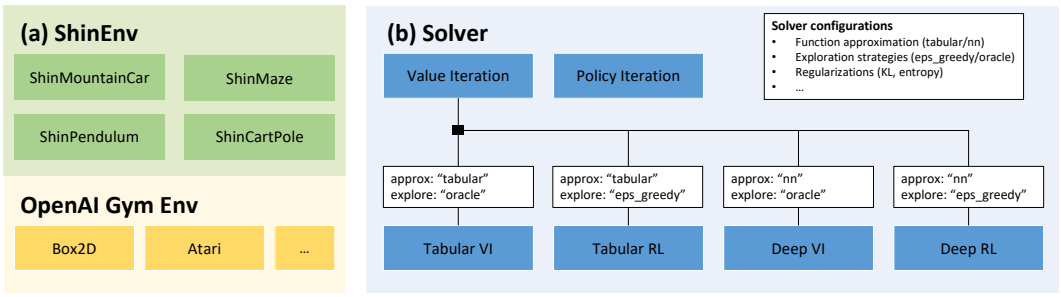

Figure 2: **Overview of ShinRL.**

analyze if developed algorithms work theoretically as expected. In what follows, we describe the design and main features of each module.

3.1 Environments

`ShinEnv` is an interface to implement MDP environments built on top of `Env` class of OpenAI Gym. We design it to extend Gym's classic control environments with a relatively small state space, such as CartPole and MountainCar, to give users access to the "oracle" that can compute exact quantities for returns, Q values, or state visitation frequencies, in an offline fashion. Indeed, when we evaluate a new RL algorithm, we often validate the algorithm on simple and constrained environments before assessing practical performances under challenging settings (*e.g.*, by using Atari and Mujoco) [Vieillard et al., 2020d, Ceron and Castro, 2021]. To this end, we typically collect samples through interactions with environments and only observe estimated returns, typically of high variance, averaged over episodes. On the other hand, such exact quantities with `ShinEnv` can help to get more accurate insights into how the algorithm works.

Under the hood of `ShinEnv`, the oracle performs exhaustive enumeration of all state-action pairs and derives how learned or optimal policies act via sparse matrix calculations. By doing so, `ShinEnv` provides the following methods:

- `calc_q` computes a Q-value table containing all possible state-action pairs (*i.e.*, Q_π) given a policy π. This method accepts some additional input arguments to consider how strongly each reward is affected by KL and entropy regularization imposed on RL algorithms.
- `calc_optimal_q` computes the optimal Q-value table (*i.e.*, Q_*) by exactly performing value iteration for a specified number of finite-horizon using the pre-computed state transition and reward matrices.
- `calc_visit` calculates state visitation frequency table containing all possible states, *i.e.*, d_π for a given policy π.
- `calc_return` is a shortcut for computing exact undiscounted returns for a given policy using state transition and reward tables. This is useful as sampling-based approaches otherwise just gives expected returns typically with high variances.

Any environment can be inherited from OpenAI Gym's `Env` to `ShinEnv` as long as its state space is reasonably small. When the action space is continuous, we discretize the space with a user-defined number of bins to execute the above-mentioned methods while the environment itself can accept the original continuous actions. Table 1 shows some default environments we already implemented. While we developed `ShinCartPole`, `MountainCar`, and `Pendulum` by inheriting respective OpenAI Gym's `Env` classes, we create `ShinMaze` from scratch as an environment that solves an easy 2D maze where agents need to arrive at predefined goal locations while avoiding obstacles, like the one implemented and evaluated in Fu et al. [2019]. For some environments, we also support state spaces given by raw input images, which enforces solvers presented in the next section to automatically use convolutional neural networks when approximating policy or Q functions.

Table 1: **Default Environments Implemented in ShinEnv.**

Environment	Discrete action	Continuous action	Image observation	Tuple observation
`ShinMaze`	✓	✗	✗	✓
`ShinCartPole`	✓	✓	✗	✓
`ShinMountainCar`	✓	✓	✓	✓
`ShinPendulum`	✓	✓	✓	✓

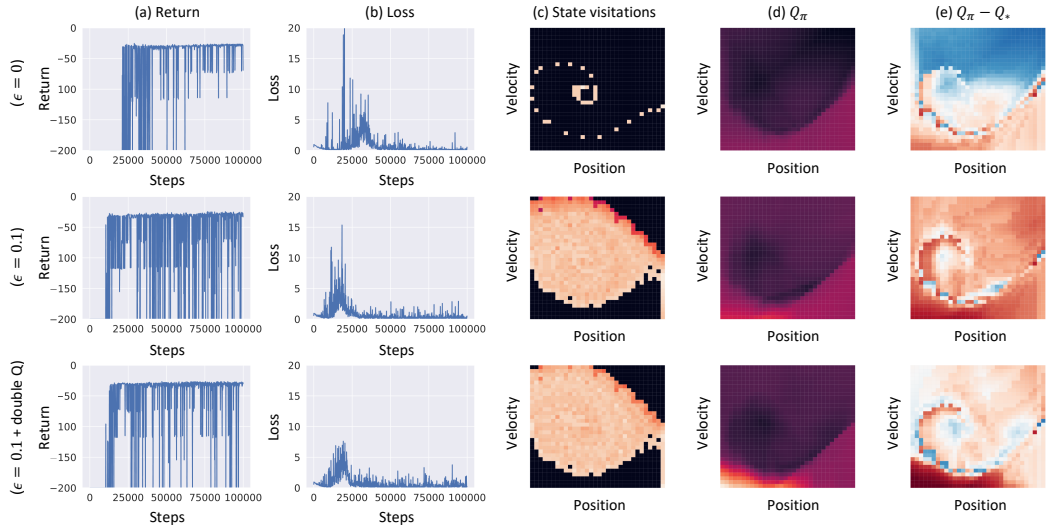

Figure 3: **Comparison of DQL with Different Settings on ShinMountainCar.** Column (e) shows the gap between the optimal Q values Q_* and learned ones Q_π, where negative and positive values (*i.e.*, overestimation and underestimation) are highlighted in red and blue.

3.2 Solvers

`Solvers` is an interface on which a variety of DP and RL algorithms can be implemented. As summarized in Fig. 2(b), `Solvers` has a hierarchical structure based on how methods are theoretically related with each other as introduced in Sec. 2.2. More concretely, the current version supports VI and its extensions such as KL-regularized VI (also known as Dynamic Policy Programming [Azar et al., 2012]) and conservative VI [Kozuno et al., 2019], as well as tabular Q learning, deep Q learning [Mnih et al., 2015], and Munchausen RL [Vieillard et al., 2020a] that are all extended from VI. We also implement PI as well as actor-critic [Konda and Tsitsiklis, 2000] and soft actor-critic [Haarnoja et al., 2017] as variants of PI.

Importantly, all of these algorithms can be used in a consistent fashion by toggling function approximation and other extensions such as exploration strategies and KL and entropy regularizers when instantiating them. For example, by disabling function approximation while enabling exploration, deep RL methods will reduce to their tabular RL variants. Alternatively, disabling both function approximation and exploration turns the methods back into VI or PI. This is very unlike existing libraries that extensively but exclusively support rapid-prototyping of deep RL algorithms.

4 Case Studies

4.1 Delving into the results of DQL

Deep Q learning (DQL) [Mnih et al., 2015] is a popular approach that still has much room for improvement. While deep networks used to approximate the Q function are generally highly expressive, they also need to be trained from diverse transition samples and therefore require a well-tuned exploration strategy in practice.

First, let us introduce how ShinRL can visualize the effectiveness of exploration strategies on the `ShinMountainCar` environment. As the original DQL can be seen as an extension of Value Iteration with neural network approximation and epsilon-greedy exploration, DQL can be built by passing `nn` to `approx` and `eps_greedy` to `explore` in the configuration. Instantiating the environment and performing the DQL on it can be done simply in a few lines as shown below.

```
import gym
from shinrl.solvers.vi.discrete import ViSolver

# instantiate an environment
env = gym.make("ShinMountainCar-v0")

# instantiate deep Q learning-based solver
config = ViSolver.DefaultConfig(approx="nn", explore="eps_greedy")
solver = ViSolver.factory(config)
solver.initialize(env, config)

# run the solver
solver.run()
```

The epsilon-greedy exploration strategy highly depends on its value of epsilon, *i.e.*, how likely the agent takes random actions at each step. To understand how this epsilon affects DQL's behaviors, we run two DQL solvers with different constant values set to epsilon, $\epsilon = 0.0$ and $\epsilon = 0.1$, and observe their state visitation frequencies. This can be done can with `calc_visit` function as follows, which take just about 10ms in a CPU environment[2].

```
# compute state-action visitation table using learned policy
policy = solver.history.tbs["ExplorePolicy"]
visit = env.calc_visit(policy)
```

Figure 3(a) and (b) present plots for returns and losses, and corresponding state visitation tables are visualized in (c). A bit surprisingly, both of the solvers finally solved the task (defined by the return arrived at -20), and their losses are almost comparable. Nevertheless, they demonstrate a clear difference in state-visitation frequencies, where the solver with $\epsilon = 0$ leads to a poor exploration policy that can potentially visit a limited set of states even after a large number of steps, while the solver with $\epsilon = 0.1$ can visit almost all possible states the agent can reach. Now we are interested in how this difference in state visitation frequencies affects the quality of learned Q functions. Here, we visualize the difference between learned and optimal Q values as follows:

```
optimal_q = env.calc_optimal_q()
learned_q = solver.history.tbs["Q"]
diff_q = learned_q - optimal_q
```

As shown in Fig. 3(e), Q values are inaccurate in many places due to underestimation under $\epsilon = 0$ and overestimation under $\epsilon = 0.1$. Overestimation is particularly a known phenomenon and can be alleviated via double-Q learning [Hasselt, 2010, Van Hasselt et al., 2016] that learns two Q functions with different sets of samples. The bottom row of Fig. 3 shows that the double-Q trick indeed improves the accuracy of the learned Q values, except for some state-action pairs that were not visited during learning. This further implies the importance of better dealing with out-of-distribution actions such as done in offline RL [Levine et al., 2020], which we leave for future work.

4.2 Comparing VI, KL-regularized VI, CVI, and Munchausen DQL

As we introduced in Sec. 2.2, VI theoretically becomes robust to estimation errors, which typically arise due to function approximation and exploration, by imposing KL regularization [Vieillard et al., 2020b]. Furthermore, involving entropy as well as KL regularizations turns VI to conservative VI (CVI) [Kozuno et al., 2019], which inspires the formulation of Munchausen DQL (M-DQL) [Vieillard et al., 2020a] that extends conventional deep Q learning with these regularizations to improve the stability. In this case study, we demonstrate how these theoretical findings of VI, KL-regularized VI, CVI, DQL, and M-DQL can be confirmed empirically and systematically using ShinRL.

[2]Confirmed with Intel(R) Core(TM) i9-11900H @ 2.50GHz.

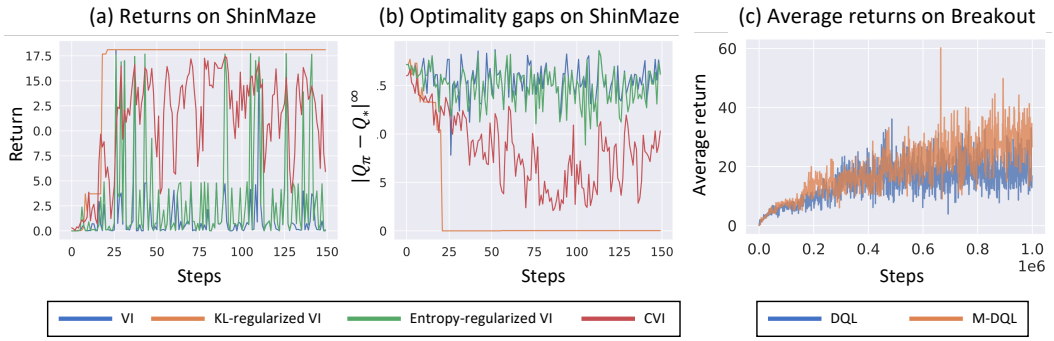

Figure 4: **Comparisons of VI Variants with Various Regularizations.** (a) Return plots and (b) optimality gaps given by $\|Q_{\pi_k} - Q_*\|_\infty$ for VI, KL-regularized VI, entropy-regularized VI, and CVI on the ShinMaze environment. (c) DQL and M-DQL on the Breakout environment.

To observe how VI changes its behavior with regularizations, let us start from formulations. Consider the following DP update schemes:

$$\begin{cases} \pi_{k+1} & = \mathrm{argmax}_\pi \left(\langle \pi, Q_k \rangle - \tau \mathrm{KL}(\pi \| \pi_k) + \lambda \mathcal{H}(\pi) \right), \\ Q_{k+1} & = r + \gamma P \langle \pi_{k+1}, Q_k - \tau \mathrm{KL}(\pi_{k+1} \| \pi_k) + \lambda \mathcal{H}(\pi_{k+1}) \rangle + \epsilon_k, \end{cases} \tag{1}$$

where τ and λ are coefficients for KL and entropy regularization, respectively. $\epsilon_k \sim \mathcal{N}(0, \sigma)$ is a zero-mean Gaussian error vector with standard deviation σ at k-th iteration, which models either function approximation and/or exploration errors. Evaluating this regularized DP with ShinRL is quite simple by just toggling the configurations of `ViSolver`, where the parameters τ, λ, and σ are respectively specified by `kl_coef`, `er_coef`, and `noise_scale`. For example, CVI that comes with both KL and entropy regularizations can be instantiated as follows:

```
# Instantiate VI
cvi_config = ViSolver.DefaultConfig()
cvi_config.update(
    {
        "approx": "tabular",  # use tabular method
        "explore": "oracle",  # use oracle for exploration
        "noise_scale": 1,  # scale of simulated noise
        "kl_coef": 0.1,
        "er_coef": 0.3,
    }
)
cvi_solver = ViSolver.factory(cvi_config)
cvi_solver.initialize(env, cvi_config)
cvi_solver.run()
```

Figure 4 shows (a) returns as well as (b) the optimality gap defined by $\|Q_* - Q_\pi\|_\infty$ on the `ShinMaze` environment. KL-regularized VI is indeed robust against noise empirically and can easily reach the optimality. On the other hand, entropy regularization is less important than KL regularization, which is also explained by Vieillard et al. [2020b]. Nevertheless, the next section will show the effectiveness of entropy regularization when used in the SAC algorithm [Ceron and Castro, 2021].

Now the task is to compare DQL and M-DQL. They are deep RL algorithms that should be evaluated in more challenging environments to best show their performances. To this end, ShinRL fully supports OpenAI Gym, and can call the `minatar` environment [Young and Tian, 2019] that is a lightweight testbed inspired by Atari games.

```
from shinrl import utils
env = utils.make_minatar("breakout")

# M-DQL
config = ViSolver.DefaultConfig(
    "approx": "nn",
```

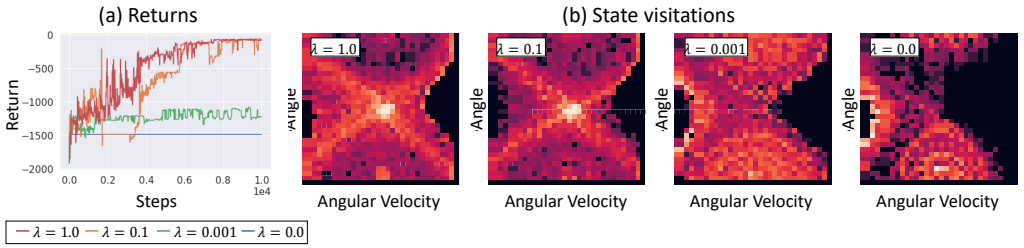

Figure 5: **SAC with Different Coefficients λ for Entropy Regularization:** (a) Return plots and (b) State visitation frequencies (higher frequencies highlighted in red) on the ShinPendulum environment.

```
    "explore": "eps_greedy",
    # the solver reduces to standard DQL by setting kl_coef and
    er_coef to zero.
    kl_coef=0.027,
    er_coef=0.003
)
mdql_solver = ViSolver.factory(config)
mdql_solver.initialize(env, config=config)
mdql_solver.run()
```

Figure 4 (c) depicts return plots for the Breakout environment, demonstrating that M-DQL outperforms DQL thanks to KL and entropy regularizations.

4.3 Evaluating robustness of the SAC algorithm to adversarial rewards

Another family of algorithms that ShinRL supports extensively is policy iteration (PI), which is the foundation of many recent deep RL algorithms. For example, the SAC algorithm [Haarnoja et al., 2018] is an extension of PI with function approximation, exploration, and entropy regularization. Some recent work shows that the SAC algorithm can learn a robust policy, both empirically [Haarnoja et al., 2018] and theoretically [Husain et al., 2021], thanks to its entropy regularizer. In this case study, we first investigate how SAC changes its robustness, in particular to adversarial rewards presented by Husain et al. [2021], with different entropy regularization coefficient λ in the `ShinPendulum` environment. In ShinRL, SAC can be implemented on the top of `PiSolver` as follows:

```
# Instantiate SAC
config = PiSolver.DefaultConfig()
config.update(
    {
        "approx": "nn",  # use neural network approximation
        "explore": "eps_greedy",  # use epsilon-greedy policy for
    exploration
        "er_coef": 0.1,
    }
)
sac_solver = PiSolver.factory(config)
sac_solver.initialize(env, config)
sac_solver.run()
```

Note that by setting `er_coef` to 0, the method reduces to the vanilla actor-critic algorithm [Konda and Tsitsiklis, 2000]. As done in the experiments of Husain et al. [2021], we consider the following adversarial reward r_{adv} by slightly modifying the original implementation of the pendulum:

$$r_{\text{adv}} = \begin{cases} r(s,a) + \epsilon & \text{if } r(s,a) \leq -5 \\ r(s,a) & \text{otherwise,} \end{cases} \tag{2}$$

where ϵ is sampled from the normal distribution $\mathcal{N}(4.9, 0.1)$. This reward design promotes the agent to stay the pendulum around its initial state while the optimal behavior is still swinging it up.

Figure 5(a) shows the learning curves of SAC with different coefficients for entropy regularization. We confirm that reasonably increasing the regularization strength improves the robustness against adversarial rewards as confirmed by Husain et al. [2021].

Furthermore, we validate the finding of Haarnoja et al. [2017] that empirically confirms the improvement of exploration quality as the entropy regularization becomes stronger. By using `count_visit` function, we can visualize the frequencies of state-action pairs stored in a replay buffer, making it possible to assess if the exploration is sufficient during the training. As shown in Fig. 5(b), we confirm that a wider range of state-action pairs are visited as λ becomes higher.

5 Conclusion

We presented ShinRL, an open-source library that can evaluate RL algorithms from both theoretical and practical perspectives in a principled fashion. As shown in our case studies, ShinRL can be used to analyze the behavior of deep RL algorithms through the lens of Q-value tables and state visitation frequencies, which are not immediately available in existing RL libraries. Further, we empirically confirm recent theoretical findings of KL regularization and entropy regularization for RL [Kozuno et al., 2019, Vieillard et al., 2020a, Husain et al., 2021] using our flexible RL solver interface. Future work will seek to extend the library to deal with a wider variety of tasks and algorithms not only on RL but also imitation learning and offline RL.

Acknowledgments

The authors would like to thank Masashi Hamaya for helpful feedback on the manuscript.

References

Tadashi Kozuno, Eiji Uchibe, and Kenji Doya. Theoretical Analysis of Efficiency and Robustness of Softmax and Gap-Increasing Operators in Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning*, pages 2995–3003, 2019.

Nino Vieillard, Olivier Pietquin, and Matthieu Geist. Munchausen Reinforcement Learning. In *Advances in Neural Information Processing Systems*, pages 4235–4246, 2020a.

Hisham Husain, Kamil Ciosek, and Ryota Tomioka. Regularized Policies are Reward Robust. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 64–72, 2021.

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2nd edition, 2018.

Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement Learning in Robotics: A Survey. *International Journal of Robotics Research*, 32(11):1238–1274, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-Level Control through Deep Reinforcement Learning. *Nature*, 518(7540):529–533, 2015.

Stephan Zheng, Alexander Trott, Sunil Srinivasa, Nikhil Naik, Melvin Gruesbeck, David C Parkes, and Richard Socher. The AI Economist: Improving Equality and Productivity with AI-Driven Tax Policies. *arXiv preprint arXiv:2004.13332*, 2020.

Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement Learning with Deep Energy-based Policies. In *Proceedings of the International Conference on Machine Learning*, pages 1352–1361, 2017.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *Proceedings of the International Conference on Machine Learning*, pages 1861–1870, 2018.

Matteo Pirotta, Marcello Restelli, Alessio Pecorino, and Daniele Calandriello. Safe Policy Iteration. In *Proceedings of the International Conference on Machine Learning*, pages 307–315, 2013.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust Region Policy Optimization. In *Proceedings of the International Conference on Machine Learning*, pages 1889–1897, 2015.

Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. ChainerRL: A Deep Reinforcement Learning Library. *Journal of Machine Learning Research*, 22(77):1–14, 2021.

Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A Research Framework for Deep Reinforcement Learning. *arXiv preprint arXiv:1812.06110*, 2018.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning*, pages 3053–3062, 2018.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO. In *Proceedings of the International Conference on Learning Representations*, 2020.

Johan Samir Obando Ceron and Pablo Samuel Castro. Revisiting Rainbow: Promoting More Insightful and Inclusive Deep Reinforcement Learning Research. In *International Conference on Machine Learning*, volume 139, pages 1373–1383, 2021.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

Nino Vieillard, Tadashi Kozuno, Bruno Scherrer, Olivier Pietquin, Rémi Munos, and Matthieu Geist. Leverage the Average: an Analysis of KL Regularization in Reinforcement Learning. In *Advances in Neural Information Processing Systems*, pages 12163–12174, 2020b.

Marc G Bellemare, Georg Ostrovski, Arthur Guez, Philip Thomas, and Rémi Munos. Increasing the Action Gap: New Operators for Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1476–1483, 2016.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Hado Hasselt. Double Q-Learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, page 2094–2100, 2016.

Christopher JCH Watkins and Peter Dayan. Q-Learning. *Machine learning*, 8(3-4):279–292, 1992.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

Rémi Munos and Csaba Szepesvári. Finite-Time Bounds for Fitted Value Iteration. *Journal of Machine Learning Research*, 9(27):815–857, 2008.

Bruno Scherrer, Mohammad Ghavamzadeh, Victor Gabillon, Boris Lesner, and Matthieu Geist. Approximate Modified Policy Iteration and its Application to the Game of Tetris. *Journal of Machine Learning Research*, 16:1629–1676, 2015.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous Control with Deep Reinforcement Learning. In *Proceedings of the International Conference on Learning Representations*, pages 1–14, 2015.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. In *Proceedings of International Conference on Machine Learning*, pages 1587–1596, 2018.

Mohammad Gheshlaghi Azar, Vicenç Gómez, and Hilbert J Kappen. Dynamic Policy Programming. *Journal of Machine Learning Research*, 13(1):3207–3245, 2012.

Mohammad Ghavamzadeh, Hilbert Kappen, Mohammad Azar, and Rémi Munos. Speedy Q-Learning. In *Advances in Neural Information Processing Systems*, pages 2411–2419, 2011.

Nino Vieillard, Bruno Scherrer, Olivier Pietquin, and Matthieu Geist. Momentum in Reinforcement Learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 2529–2538, 2020c.

Hiroyuki Yamada. cpprb, 1 2019. URL `https://gitlab.com/ymd_h/cpprb`.

Nino Vieillard, Olivier Pietquin, and Matthieu Geist. Deep Conservative Policy Iteration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 6070–6077, 2020d.

Justin Fu, Aviral Kumar, Matthew Soh, and Sergey Levine. Diagnosing Bottlenecks in Deep Q-Learning Algorithms. In *Proceedings of the International Conference on Machine Learning*, pages 2021–2030, 2019.

Vijay R Konda and John N Tsitsiklis. Actor-Critic Algorithms. In *Advances in Neural Information Processing Systems*, pages 1008–1014, 2000.

Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *arXiv preprint arXiv:2005.01643*, 2020.

Kenny Young and Tian Tian. Minatar: An Atari-Inspired Testbed for Thorough and Reproducible Reinforcement Learning Experiments. *arXiv preprint arXiv:1903.03176*, 2019.