Experiments of Advanced Model-Free Algorithms on Value and Policy-Based Reinforcement Learning
CS489 Reinforcement Learning Project

Zenan Li
emiyali@sjtu.edu.cn

Abstract

Real world problems such as video games and physics simulation are soliciting effective methods for model-free control. Generally, value-based algorithms such as DQN and its advanced variants can deal with discrete action space efficiently. However, they are inherently unsuited to environments with continuous action space since the max operator is intractable for continuous functions. Therefore, policy-based algorithms are developed to learn the policy directly. One step further, Actor-Critic algorithms are proposed to reduce variance in policy-based training, while prior works such as SAC strive to introduce off-policy learning schema to promote sample efficiency. In this paper, we implement and evaluate the value-based algorithm: D3QN on three of Gym Atari environments. Besides, we implement two policy based algorithms: PPO (on-policy) and SAC (off-policy) and test them on four of Gym MuJoCo environments. Apart from that, we also conduct ablation studies to verify the significance of each module in D3QN, and introduce prioritized experience replay (PER) to further boost SAC's performance. Experiment results show that our algorithms can achieve satisfying results on all these 7 environments.

1 Introduction

Value-based algorithms have emerged as effective methods for reinforcement learning (RL) problems with discrete action space. Generally, they strive to fit action value function or state value function, e.g. Monte-Carlo, TD learning for model-free policy evaluation and SARSA [8], Q-learning [17] for model-free control. With the quick development of deep learning, neural networks have been introduced into reinforcement learning to further boost its performance. Series of algorithms based on deep Q-network (DQN) [7] have been proposed, including double DQN (DDQN) [15], which uses target networks to re-estimate the greedy action value to avoid over-estimation, and dueling DQN [16], which learns state value and action advantage function separately to promote the representative power. Typically, we can combine DDQN with the dueling network architecture to form dueling DDQN (D3QN), which further improves its model-free control performance.

In this paper, we will implement and evaluate D3QN on three of Gym Atari [1] environments: Breakout, Boxing, and Pong. We make adaptive modifications to the hyperparameters and network architectures in order to better fit the testing environment. We also do ablation studies to verify the significance of double and dueling architecture. The experiment results show that our D3QN can achieve satisfying scores on all these three environments.

Despite the great success of value-based RL algorithms, they are inherently unsuited to continuous action space (since the max operator is intractable for continuous functions). Therefore, policy-based RL is proposed to solve continuous control problems. The key idea is to optimize over an actor directly using policy gradient [13], however, original policy gradient suffers from high variance. Actor-Critic algorithms [4] such as proximal policy optimization (PPO) [12] introduce an additional module named critic to estimate state or action value, which can dramatically reduce variance (though introduce bias) and stabilize the training procedure. The biggest drawback for policy gradient methods is sample inefficiency: since policy gradients are estimated from rollouts. Although Actor-Critic methods use value approximation instead of rollouts, its on-policy style remains sample inefficient. Taking the idea from Q-learning, prior works such as deep deterministic policy gradient (DDPG) [6] and soft Actor-Critic (SAC) [2, 3] strive to introduce off-policy mode to policy-based RL.

In this paper, we implement and evaluate two traditional policy-based RL algorithms: PPO (on-policy) and SAC (off-policy) on four of Gym MuJoCo [14] environments: Hopper, Humanoid, HalfCheetah, and Ant. Generally, the experiment results show that SAC is much more sample efficient than PPO, which always achieves higher scores on these four environments. Besides, we also introduce prioritized experience replay (PER) [9] to further boost SAC's performance.

2 Environment Setup

As we have discussed above, we use Gym Atari and MuJoCo environments as benchmarks to evaluate value-based and policy-based RL algorithms, respectively. In this section, we will briefly introduce the environments and setup procedures.

2.1 Gym Atari

Gym Atar [1] is a simulation environment of video games, which provides more than 100 original games and different versions of them in Atari-2600 platform. Generally, the action space of all Atari games is a subset of a discrete set of 18 legal actions. The observation space (return state) is the RGB (or gray) image that is displayed to the human player. The exact reward dynamics depend on the specific environment and game design. Considering the these properties and the diversity of the environments, Atari is an ideal platform to evaluate value-based RL algorithms.

In order to setup the Atari environment, we need to use pip tools to install `atari-py` package in our virtual environment. After that, we need to download `Roms.rar` from the website [1] and extract it to a certain folder. Then, we run:

```
python -m atari_py.import_roms <path to folder>
```

by which the ROMs will be copied into the `atari-py` installation directory. To this end, we can run the Atari simulator smoothly.

2.2 Gym MuJoCo

MuJoCo [14] is the abbreviation of Multi-Joint dynamics with Contact. It is a physics engine for facilitating research and development in robotics, biomechanics, graphics and animation. Generally, the action space of MuJoCo is a bounded continuous vector, and the observation space is also a vector, whose dimension and and physics meaning depend on specific environments. The reward dynamics is specially designed for each environment. Because of its integrity and diversity, MuJoCo has become a widely adopted benchmark for policy-based RL algorithms.

As in October 2021 DeepMind has acquired MuJoCo, it is open sourced and become free for everyone in 2022. So we no longer need to buy the license but can start the installation directly. Frist, we need to download `mujoco210` from the website [2] and extract it into the folder *~/.mujoco/mujoco210*. After that, we need to use pip tools to install `mujoco-py` package in our virtual environment. To this end, we can run the MuJoCo engine after we import `mujoco-py` in our code. Note that some errors may occur during the environment building procedure. It is usually because that MuJoCo currently only supports GCC with version before 9. Besides, we also need to install some missing packages according to the error logs.

[1] `http://www.atarimania.com/rom_collection_archive_atari_2600_roms.html`.

[2] `https://mujoco.org/download/mujoco210-linux-x86_64.tar.gz`.

3 Value-Based Method: D3QN

As the most widely adopted value-based RL algorithm, DQN [7] has endured many trials and spawned many advanced variants. According to the tutorial given by David Silver in ICML 2016, the advanced modifications to DQN can be mainly classified into three directions: DDQN [15], Dueling DQN [16], and PER [9]. In this section, we will combine the former two techniques to form D3QN, and test its performance on three of Gym Atari environments. We also do ablation studies to verify the significance of each component in D3QN.

3.1 About Environment

Specifically, we choose the three environments: *BreakoutNoFrameskip-v4, PongNoFrameskip-v4, and BoxingNoFrameskip-v4* as shown in Figure 1 to test our algorithms. Therefore, we would like to introduce the basic ideas on the implementation and usage of them in this section.

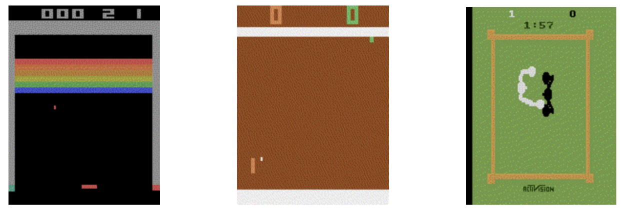

(a) Breakout-v4 (b) Pong-v4 (c) Boxing-v4

Figure 1: An overview of selected Atari environments.

Wrapper class. Gym provides a Wrapper class [3] for better adaptation to Atari 2600 games, which is inherited from the Env base class. In this way, we can customize the configurations and interaction method when using different Atari environments by passing environment-specific parameters to methods in the Wrapper class.

Environment rules. The Atari game environment is a entirely deterministic environment, i.e. the state transition function is deterministic. However, an agent performs well in the deterministic environment may be highly sensitive to perturbations, so randomness is added into the environment rules to mitigate this issue. Specifically, instead of always simulating the action passed to the environment, there is a small probability that the previously executed action is used instead. Besides, the agent may have multiple lives in the games we choose. We will mark an episode to be *done* every time the agent losses its life.

Reward and observation clipping. The reward for each action is divided into {-1,0,+1} to prevent the influence of different reward settings across different environments. Besides, the original observed image is converted from 210×160 to 84×84, and the RGB image is converted to a gray scale image, which would be stored into the replay buffer after transformation.

Frame staking. Using only one frame of image to represent current state may not provide enough information for the agent. Frame staking technology combines the previous k frames to form the state space. Typically, this can provide context for the observation and lead the agent to take correct actions. Specifically, we take $k = 4$ in our experiment.

3.2 Algorithm Description

First, about the vanilla DQN. It is built on the idea of Q-learning, using a neural network (NN) to approximate the Q function $Q(s, a; \theta)$, where θ denotes the parameter of the NN. Besides, it is usually combined with two key techniques: experience replay and fixed target network. The former technique proposes to store the agent's experience transitions (s_t, a_t, r_t, s_{t+1}) into a replay buffer and samples a batch after each environment step to update the network. This enables batch updating for Deep Q-learning and promotes the sample efficiency significantly. Besides, it can also mitigate

[3] `https://github.com/openai/baselines/blob/edb52c22a5e14324304a491edc0f91b6cc07453b/baselines/common/atari_wrappers.py`

the dependence for the training data and stabilize the training procedure. Specifically, the loss of DQN is calculated by the TD error. However, the TD target: $r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta)$ can be very unstable if we update the Q-net in every iteration. Therefore, the fixed target network is proposed to tackle this issue by updating the target network after a few iterations, and the loss function can be formulated as:

$$L(\theta) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s', a'; \overline{\theta}) - Q(s, a; \theta)\right)^2\right], \tag{1}$$

where $\overline{\theta}$ is the parameter of the fixed target network, and the expectation term is approximated by the average value of the sampled experience batch.

Improvement 1: Dueling DQN. The key insight of the dueling network architecture is that for many states, it is unnecessary to estimate the value of each action choice. To bring this insight into fruition, we modify the Q-net architecture, as illustrated in Figure 2.

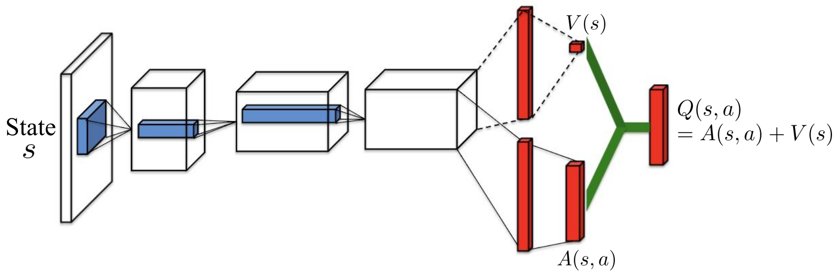


Figure 2: Dueling DQN network architecture [16].

The lower layers are convolutional (to process image inputs in Atari) as in the original DQNs. However, instead of following the convolutional layers with a single sequence of fully connected layers, we instead use two streams of fully connected layers. The streams are constructed such that they have the capability of providing separate estimates of the value and advantage functions. Finally, the two streams are combined to produce a single output Q function:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha). \tag{2}$$

Typically, the two streams can learn to pay attention to different information during the training procedure. For example, in an autonomous driving scenario, the value stream learns to pay attention to the road, while the advantage stream learns to pay attention *only* when there are cars immediately in front, so as to avoid collisions.

However, Equation 2 is yet another parameterized estimate of the true Q function. It is unidentifiable in the sense that given Q we cannot recover V and A uniquely, which will lead to bias and poor practical performance. To address this issue, [16] proposes to normalize the advantage function with zero mean value:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s, a'; \theta, \alpha)), \tag{3}$$

where $\mathcal{A}$ denotes the action space of the environment. We adopt Equation 3 in our final implementation for its promising performance.

Improvement 2: Double DQN. The idea of DDQN is proposed in [15] to solve the over-estimation problem of vanilla DQN. The target Q function $r + \gamma \max_{a'} Q(s', a'; \overline{\theta})$ tends to select the action that is over-estimated. To mitigate this issue, DDQN proposes to select the target action by the policy net, while evaluate its value by the target net:

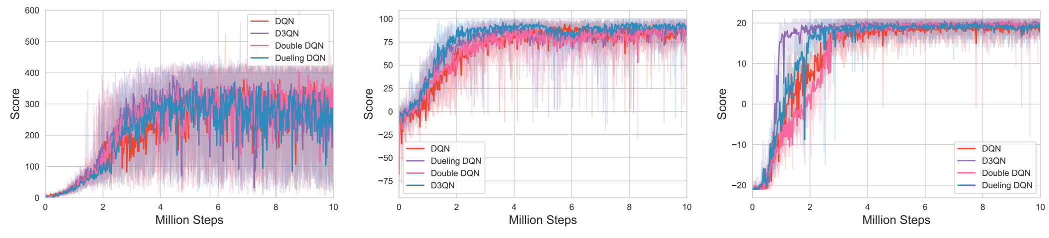
$$Q(s, a; \theta) \leftarrow r + Q'(s', \arg\max_{a'} Q(s', a'; \theta); \overline{\theta}). \tag{4}$$

In Equation 4, we decouple the process of action choosing and value estimation, which can significantly eliminate the over-estimation issue. For example, if Q over-estimate a', so it is selected. Then Q' will give it proper value. Besides, the action overestimated by Q' will not be selected by Q.

Typically, we can see that dueling DQN and double DQN are orthogonal techniques that pay attention to different problems. Therefore, we can combine them to form Dueling Double DQN (D3QN) to achieve even better performance. Specifically, we adopt the dueling net architecture in Figure 2 to form our Q-net, and we use Equation 4 to calculate the loss function.

3.3 Experiments Analysis

We conduct experiments on Gym Atari in this section to validate the effectiveness of our value-based algorithms, also to verify the significance of each component in D3QN.

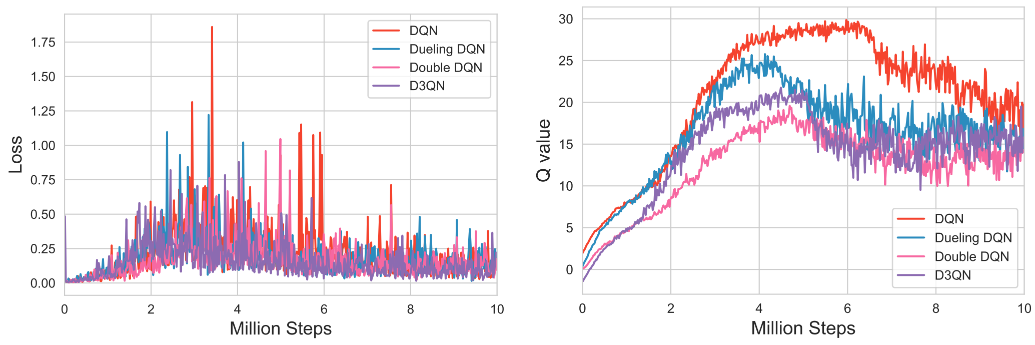


(a) Breakout-v4 (b) Boxing-v4 (c) Pong-v4

Figure 3: Reward (score) of SAC agents trained on Gym Atari.

Average evaluation rewards. The main results can be found in Figure 3. Specifically, we train 4 algorithms on the 3 environments respectively, with each performing 10 evaluation rollouts every 10,000 environment steps. The solid curves corresponds to the mean and the shaded region to the minimum and maximum returns over the 10 evaluation rollouts.

Generally, the four algorithms all achieve satisfying scores on the three testing environments. As we can see, the **asymptotic (final) performance** of the four algorithms are almost on a par with each other. In comparison, D3QN shows the best **sample efficiency**, which learns and converges faster than the three other algorithms, especially on **PongNoFrameskip-v4**. Besides, we can find that both double and dueling architecture can promote the sample efficiency of DQN, proving the effectiveness of the two proposed techniques.



(a) Loss (b) Q Value

Figure 4: Loss and Q value on BoxingNoFrameSkip-v4.

Loss and Q value. We also visualize the loss and Q value in Figure 4 to have more intuition of how dueling and double DQN taking effects during training.

Typically, Figure 4(a) shows that the loss of DQN is much higher and more unstable than the peer algorithms. Note that it is normal to encounter steep increases in the loss curves, as the agent keeps exploring new transitions that bring about new knowledge (with large loss). In this sense, we can say that dueling and double DQN can promote the sample efficiency as they stabilize the loss curves.

As for the Q value, Figure 4(b) reveals the over-estimated problem [15] of DQN (the red line is always higher though the four curves tend to converge to a same value asymptotically). We can also see that the Q value of the double DQN is the smallest, proving its capability of mitigating over-estimation, which is consistent with our discussion in Section 3.2. The better learned Q value is another reason of the superiority of D3QN.

4 Policy-Based Method: PPO and SAC

Although value-based RL algorithms have achieved remarkable performance in tasks such as Atari games, the inherent drawbacks has hindered its development:

- First, value-based methods are inherently unsuited to problems with continuous action space since the max operator is intractable for continuous functions.
- Second, non-linear value function approximation like NNs is unstable and brittle w.r.t. their hyperparameters.

Therefore, policy-based methods are proposed to overcome these problems by learning the policy directly based on policy gradient [13]. As original policy gradient suffers from high variance, Actor-Critic [4] algorithms such as PPO [12] introduce an additional critic to reduce the variance during training (though raise bias accordingly). Besides, inspired by the idea of Q-learning, prior works such as DDPG [6] and SAC [2] strive to introduce off-policy learning schema for Actor-Critic, which can promote its sample efficiency significantly. In this paper, we implement PPO and SAC (with PER) separately, then test and compare them on four of Gym MuJoCo environments.

4.1 About Environment

Specifically, we choose four environments from Gym MuJoCo: *Hopper-v2, Humanoid-v2, HalfCheetah-v2, and Ant-v2* to serve as benchmarks for the policy-based models in our experiments, which can be found in Figure 7. So here we would like to give a brief introduction about the implementation and usage of them.

Environment configuration. XML configuration files are used in MuJoCo for users to specify or customize the details of physical simulation, including physical models, velocity change and so on. For example, one can use `<acutator>` to define the joints that can perform motion, use `<body>` to define all simulator components.

Environment rules. As a physics simulation environment, the state spaces of MuJoCo consists of two parts that are flattened and concatenated together: a position of a body part or joint and its corresponding velocity. The action space is a bounded continuous vector that represents the driving force on the key parts of the object. The reward strategy is specially designed for each environment to learn better control policy. Typically, the agent will start from a fixed state with a uniform noise added to each dimension, and an episode will terminate when the physics object is no longer alive (according to some predefined conditions).

4.2 Algorithm Description

First, about policy gradient. Suppose we have an actor network $\pi(a|s, \phi)$ parameterized with ϕ, e.g. an Gaussian distribution with learnable mean and standard deviation. Denote the total reward of an episode $\tau = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}$ as $R(\tau) = \sum_{t=1}^{T} r_t$. The target of policy gradient is to maximize $\overline{R}_\phi = \mathbb{E}_{\tau\sim\pi(\tau;\phi)}[R(\tau)]$. Omitting the complex mathematical induction, the policy gradient yields that:

$$\begin{aligned}
\nabla_\phi \overline{R}_\phi &= \mathbb{E}_{\tau\sim\pi(\tau;\phi)}[R(\tau)\nabla_\phi \log \pi(\tau;\phi)] \\
&\approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla_\phi \log \pi(a_t^n|s_t^n;\phi) \\
&\approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}(\sum_{t'=t}^{T_n}\gamma^{t'-t}r_{t'}^n - b)\nabla_\phi \log \pi(a_t^n|s_t^n;\phi).
\end{aligned} \tag{5}$$

where $b \approx \mathbb{E}[R(\tau)]$ is an added baseline, to avoid that $R(\tau^n)$ is always positive (or negative). We also introduce $\gamma^{t'-t}$ to assign suitable credits to the actions.

Second, about the PPO algorithm. As the original policy gradient suffers from high variance due to the rollout estimates, an additional critic is introduced in Actor-Critic (AC) for value estimation. Specifically, the critic uses the approximation $Q^\pi(s_t^n, a_t^n) = \mathbb{E}[r_t^n + V^\pi(s_{t+1}^n)] \approx r_t^n + V^\pi(s_{t+1}^n)$ and only estimates the state value. In this sense, the policy gradient can be modified as:

$$\nabla_\phi \overline{R}_\phi = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}(r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n))\nabla_\phi \log \pi(a_t^n|s_t^n;\phi). \tag{6}$$

which can reduce the variance drastically (though introduce some bias). Typically, the actor is optimized using policy gradient, while the critic is optimized by Monte-Carlo or the TD error.

PPO [12] (originates from TRPO [10]) strives to promote AC's sample efficiency by enabling learning from previous experience. Specifically, it uses the importance sampling to modify the policy gradient:

$$\begin{aligned}
\nabla_\phi \overline{R}_\phi &= \mathbb{E}_{(s_t,a_t)\sim\pi_\phi}[A(s_t,a_t;\phi)\nabla_\phi \log \pi(a_t^n|s_t^n;\phi)] \\
&= \mathbb{E}_{(s_t,a_t)\sim\pi_{\phi'}}\left[\frac{\pi(s_t,a_t;\phi)}{\pi(s_t,a_t;\phi')}A(s_t,a_t;\theta)\nabla_\phi \log \pi(a_t^n|s_t^n;\phi)\right] \\
&= \mathbb{E}_{(s_t,a_t)\sim\pi_{\phi'}}\left[\frac{\nabla_\phi\pi(s_t,a_t;\phi)}{\pi(s_t,a_t;\phi')}A(s_t,a_t;\theta)\right]
\end{aligned} \tag{7}$$

Equation 7 implies that we can use experience sampling from policy $\pi_{\phi'}$ to optimize π_ϕ, which can promote the sample efficiency significantly. We adopt generalized advantage estimation (GAE) [11] to calculate $A(s_t, a_t; \theta)$ for its promising performance. Its key idea is similar to TD(λ), but we omit it here since it's not our main concern in this paper. However, we need to constraint the difference between $\pi_{\phi'}$ and π_ϕ for practical use. Specifically, in PPO1, we introduce an additional KL-loss between $\pi_{\phi'}$ and π_ϕ. Here we adopt PPO2, which is easier to implement while achieving competitive performance:

$$J_\pi(\phi) = \sum_{(s_t,a_t)} \min\left(\frac{\pi(s_t,a_t;\phi)}{\pi(s_t,a_t;\phi')}A(s_t,a_t;\theta), \text{clip}\left(\frac{\pi(s_t,a_t;\phi)}{\pi(s_t,a_t;\phi')}, 1-\epsilon, 1+\epsilon\right)A(s_t,a_t;\theta)\right) \tag{8}$$

As we can see, it clips the probability ratio to prevent unreasonable gradient steps. In practical use, PPO will use a policy net to interact with the environment and fill the replay buffer. When the replay buffer is full, it will start optimizing the agent for a number of epochs. Then the replay buffer will be cleared and we will use the new policy net to sample experience for the next iteration. Thus, the experienced transitions are only used in one iteration, and the size of the replay buffer is rather small to ensure that π_ϕ will not go too far away from $\pi_{\phi'}$.

In conclusion, PPO is essentially an on-policy algorithm, for it optimizes a first-order approximation of the expected return while carefully ensuring that the approximation loss does not deviate too far from the underlying objective. To further promote the sample efficiency of policy-based algorithms, soft Actor-Critic (SAC) [2] is proposed based on the idea of entropy maximization. Here we will discuss its improvements in details.

Improvement 1: Maximum Entropy Reinforcement Learning (MERL). The objective for standard reinforcement learning is to find the policy that can maximize the cumulative reward:

$$\pi^*_{\text{std}} = \arg\max_\pi \sum_t \mathbb{E}_{(s_t,a_t)\sim\pi}[r(s_t,a_t)], \tag{9}$$

while the objective for MERL is:

$$\pi^*_{\text{MaxEnt}} = \arg\max_\pi \sum_t \mathbb{E}_{(s_t,a_t)\sim\pi}[r(s_t,a_t) + \alpha H(\pi(\cdot|s_t))] \tag{10}$$

where α is a hyperparameter named temperature to control the significance of the policy entropy. Different from in PPO, where we introduce an entropy loss to encourage the actor for more exploration, in MERL we directly put the entropy into the objective, which will change the Bellman equation completely. Experiments have proved that MERL can perform well in many continuous control scenarios. It also performs more stable and owns excellent immunity to interference compared to other off-policy algorithms such as DDPG [6].

Improvement 2: Energy Based Policy (EBP). In SAC, energy-based models (EBMs) [5] are adopted to represent policy:

$$\pi(a_t|s_t) \propto \exp(-\mathcal{E}(s_t, a_t)), \tag{11}$$

different from the previously used Gaussian distribution, EBM owns the promising power to approximate any probability model [5]. Therefore, the policy model in SAC is more general and can learn more useful information during the training procedure. Specifically, the energy function in SAC is connected with the Q function:

$$\mathcal{E}(s_t, a_t) = -\frac{1}{\alpha} Q_{\text{soft}}(s_t, a_t) \Rightarrow \pi(a_t|s_t) \propto \exp(\frac{1}{\alpha} Q_{\text{soft}}(s_t, a_t)), \tag{12}$$

where $Q_{\text{soft}}(s_t, a_t)$ can be modeled by a neural network.

Improvement 3: Soft Policy Iteration (off-policy). Although we can use policy gradient to solve Equation 10, here we have more clever methods to simplify the process. Typically, the objective can be re-written as:

$$\mathcal{J}(\pi) = \sum_{t=1}^{T} \gamma^{t-1} \mathbb{E}_{(s_t, a_t) \sim \pi}[r(s_t, a_t) + \alpha H(\pi(\cdot|s_t))]. \tag{13}$$

In soft Q-learning (SQL), the soft V function is defined as:

$$V_{\text{soft}}(s) = \alpha \log \int \exp(\frac{1}{\alpha} Q_{\text{soft}}(s, a)) da. \tag{14}$$

To find the optimal policy π that maximizes Equation 13, we can calculate the derivative (set to zero) of it w.r.t. $\pi(a_t|s_t)$, which yields that:

$$\pi(a_t|s_t) = \exp(\frac{1}{\alpha}(Q_{soft}(s_t, a_t) - V_{\text{soft}}(s_t))), \tag{15}$$

where $Q_{\text{soft}}(s_t) = r(s_t, a_t) + \gamma \mathbb{E}[V_{\text{soft}}(s_{t+1})]$. We can conclude that Equation 15 is consistent with the proposed form in Equation 12. The results implies that the optimal policy can be derived if we have known the soft value functions. Inspired by this, we can use a similar policy iteration training procedure as in Q-learning to optimize the agent, which enables **off-policy** training mode in SAC. Since Equation 14 is intractable, SAC proposes a soft bellman equation to conduct value iteration:

$$\begin{aligned} Q^{\pi}_{\text{soft}}(s, a) &= \mathbb{E}_{s' \sim p(s'|s,a)}[r(s, a) + \gamma V^{\pi}_{\text{soft}}(s')] \\ &= \mathbb{E}_{s' \sim p(s'|s,a), a' \sim \pi}[r(s, a) + \gamma(Q^{\pi}_{\text{soft}}(s', a') + \alpha H(\pi(\cdot|s')))]. \end{aligned} \tag{16}$$

Therefore, SAC can conduct Q value iteration according to Equation 16. The critic loss function is:

$$J_Q(\theta) = \mathop{\mathbb{E}}_{\substack{(s_t, a_t, s_{t+1}) \sim \mathcal{D} \\ a_{t+1} \sim \pi_\phi}} \left[\frac{1}{2}(Q(s_t, a_t; \theta) - (r(s_t, a_t) + \gamma(Q(s_{t+1}, a_{t+1}; \theta) - \alpha \log(\pi(a_{t+1}|s_{t+1}; \phi)))))^2 \right]. \tag{17}$$

Though we can calculate the policy probability directly according to Equation 12, however, we cannot directly sample actions from the EBP. Therefore, we use an Gaussian distribution instead to interact with the environment. Afterwards, we will optimize the Gaussian distribution towards the EBP. Specifically, we try to minimize the KL divergence:

$$\pi_{new} = \arg\min_{\pi \in \mathcal{N}} D_{\text{KL}} \left(\pi(\cdot|s_t) || \frac{\exp(\frac{1}{\alpha} Q^{\pi_{old}}_{\text{soft}}(s_t, \cdot))}{Z^{\pi_{old}}_{\text{soft}}(s_t)} \right), \tag{18}$$

where $\mathcal{N}$ denotes the set of parameterized Gaussian distribution, $Z^{\pi_{old}}_{\text{soft}}(\cdot)$ is the function to normalize the probability distribution. One step further, we can induce the actor loss function:

$$\begin{aligned} J_\pi(\phi) &= D_{\text{KL}} \left(\pi(\cdot|s_t; \phi) || \exp\left(\frac{1}{\alpha} Q(s_t, \cdot; \theta) - \log Z(s_t) \right) \right) \\ &= \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\phi} \left[\log \left(\frac{\pi(a_t|s_t; \phi)}{\exp(\frac{1}{\alpha} Q(s_t, a_t; \theta) - \log Z(s_t))} \right) \right] \\ &= \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\phi} \left[\log \pi(a_t|s_t; \phi) - \frac{1}{\alpha} Q(s_t, a_t; \theta) + \log Z(s_t) \right]. \end{aligned} \tag{19}$$

As we can see, the action a_t in the expectation is **sampled from the current policy** but not from the replay buffer. This is the key difference between SAC and those algorithms based on policy gradient, which enables **off-policy** training and promotes the sample efficiency greatly. Since the action a_t is sampled from $\pi(\cdot|s_t;\phi)$, we can introduce the reparameterization technique to make the backpropagation process differentiable. Specifically, we have:

$$a_t = f(\epsilon_t, s_t; \phi) = f^{\mu}(s_t; \phi) + \epsilon_t \odot f^{\sigma}(s_t; \phi), \tag{20}$$

where ϵ_t is sampled from the standard Gaussian distribution. Therefore, the final actor loss is transformed as:

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D},\epsilon\sim\mathcal{N}_{\text{std}}}[\alpha \log \pi(f(\epsilon_t, s_t; \phi)|s_t; \phi) - Q(s_t, f(\epsilon_t, s_t; \phi))]. \tag{21}$$

Till this end, we can summarize the data flow pattern of SAC in Figure 5. Notably, we parameterize two soft Q-functions, with parameters θ_i, and train them independently to optimize $J_Q(\theta_i)$. We then use the minimum of the soft Q-functions for Equation 17 and Equation 21, which has been found to significantly speed up training [3], especially on harder tasks.

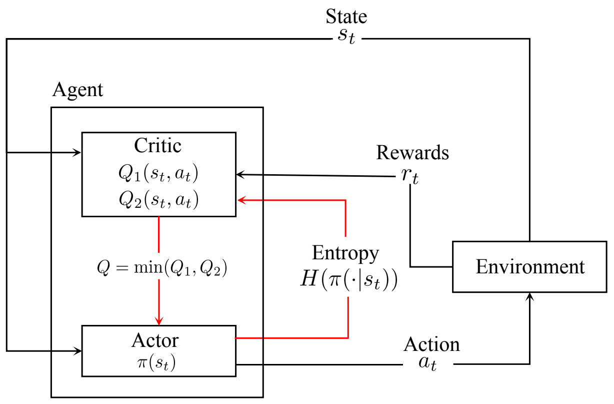


Figure 5: Data flow diagram of SAC.

Improvement 4: Automating Entropy Adjustment. The temperature α controls the significance of entropy in the objective. However, the optimal α may differ for different RL tasks, or in different stages of a same problem. As this is an important hyperparameter that will determine the performance of SAC. In [3], an algorithm has been proposed to tune the temperature automatically. Specifically, the authors formulate it as a constrained optimization problem (maximize the expected return while maintain the entropy to be larger than a threshold) and induce the loss function for the temperature:

$$J(\alpha) = \mathbb{E}_{a_t\sim\pi(a_t|s_t;\phi)}[-\alpha \log \pi(a_t|s_t; \phi) - \alpha H_0]. \tag{22}$$

In conclusion, the training procedure of SAC is summarized in Algorithm 1.

Finally, about the prioritized experience replay (PER). It is proposed in [9] to sample experienced transitions from the replay buffer w.r.t. their priority. Typically, the priority is defined as the TD error of the transition pair. In our experiment, the priority of the i-th sample is calculated as:

$$p_i = \frac{1}{2}\sum_{l=1}^{2} |Q(s_i, a_i; \theta_l) - (r(s_i, a_i) + \gamma(Q(s'_i, a'_i; \theta_l) - \alpha \log(\pi(a'_i|s'_i; \phi))))|. \tag{23}$$

Afterwards, the probability of sampling the i-th data is:

$$P(i) = \frac{p_i^{\beta_1}}{\sum_j p_j^{\beta_1}}, \tag{24}$$

where β_1 is a hyperparameter that controls how much the priority value affects the sampling probability. As we can see, a transition pair with larger TD error is more likely to be chosen. PER introduces

Algorithm 1: The training procedure of SAC.

```
Input: Original network parameters $\theta_1, \theta_2, \phi$.
1  Initialize target network $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$;
2  Initialize replay buffer $\mathcal{D} \leftarrow \emptyset$;
3  for each iteration do
4      for each environment step do
5          Sample action from the actor $a_t \sim \pi_\phi(a_t|s_t)$;
6          Sample transition from the environment $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$;
7          Store the transition in the replay buffer $\mathcal{D} \leftarrow \mathcal{D} \cup \{s_t, a_t, r_t, s_{t+1}\}$;
8      for each gradient step do
9          Sample a batch of data $\mathcal{B}$ from the replay buffer $\mathcal{D}$;
10         Calculate the critic loss $J_Q(\theta_i)$ for $i \in \{1, 2\}$, the actor loss $J_\pi(\phi)$, and the entropy loss
            $J(\alpha)$ w.r.t. the data batch $\mathcal{B}$;
11         $\theta_i \leftarrow \theta_i - \lambda_Q \nabla_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$; # Update the critic.
12         $\phi \leftarrow \phi - \lambda_\pi \nabla_\phi J_\pi(\phi)$; # Update the actor.
13         $\alpha \leftarrow \alpha - \lambda \nabla_\alpha J(\alpha)$; # Adjust temperature.
14         Update target network parameters $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$;
Output: Trained parameters $\theta_1, \theta_2, \phi$.
```

bias because it changes the sample distribution in an uncontrolled fashion. We can correct this bias by using importance sampling weights:

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^{\beta_2}, \tag{25}$$

that fully compensates for the non-uniform probabilities $P(i)$ if $\beta_2 = 1$. Typically, w_i is combined in the training procedure to weight the loss of the i-th sample. We also normalize weights by $1/\max_i w_i$ to stabilize training.

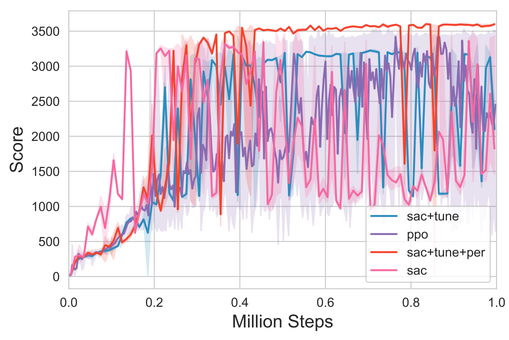
4.3 Experiment Analysis

We conduct experiments on Gym MuJoCo in this section to validate SAC's superiority over on-policy algorithms (e.g. PPO), also to verify the effectiveness of automating entropy adjustment and PER.

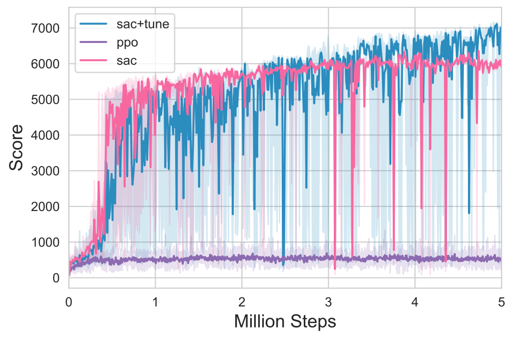
Average evaluation rewards. The main experimental results are shown in Figure 6. Specifically, we train 4 algorithms on the 4 environments respectively, with each performing 10 evaluation rollouts every 10,000 environment steps. The solid curves corresponds to the mean and the shaded region to the minimum and maximum returns over the 10 evaluation rollouts. Generally, the shaded area is large due to the instability of the testing environments.

The results show that, overall, SAC performs comparably to the baseline method (PPO) on the easier tasks such as *Hopper-v2*, and outperforms them on harder tasks with a large margin, both in terms of **sample efficiency (learning speed)** and **asymptotic (final) performance**. For example, PPO fails to make remarkable progress on *Humanoid-v2*, *Ant-v2*, and *HalfCheetah-v2*, a result that is consistent with prior works [3]. In comparison, SAC turns out to learn considerably faster than PPO as a sequence of the large batch sizes PPO needs to learn stably on more high-dimensional and complex tasks, proving the effectiveness of MERL and the off-policy training mode. The results also indicate that the automatic temperature tuning scheme works well across all the environments, and thus effectively eliminates the need for tuning the temperature. Finally, we can learn from Figure 6 that the PER technique can promote the learning speed of SAC obviously, as the red line achieves the highest score on the three tested environments. This result is consistent with the claim in [9] that emphasizing on previous large-error transitions can guide the agent to learn more effective experience, thereby promoting the sample efficiency. However, the PER technique inevitably raises the memory requirements. Note that the 'sac+tune+per' curve is missing on *Humanoid-v2*, since our device goes out-of-memory during training.

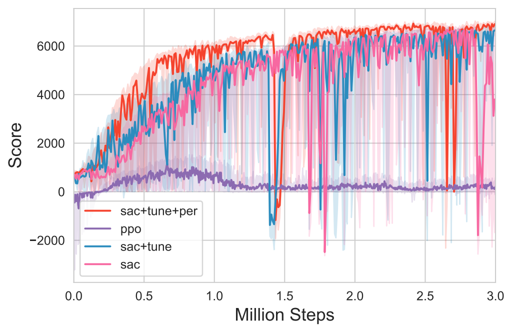
Simulated results. We also test the trained SAC agents and render the simulated results for more intuition about the promising power of SAC. As shown in Figure 7(a), the hopper can learn to jump



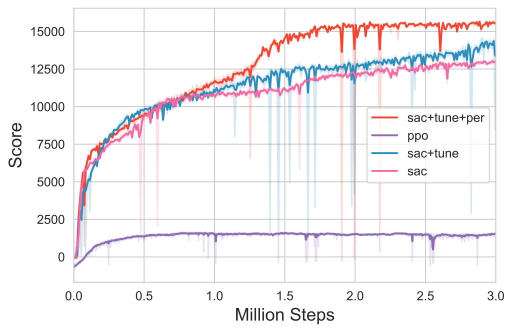
(a) Hopper-v2



(b) Humanoid-v2

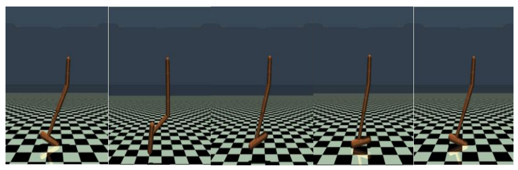


(c) Ant-v2

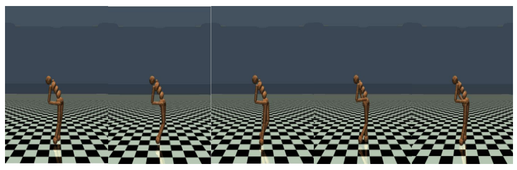


(d) HalfCheetah-v2

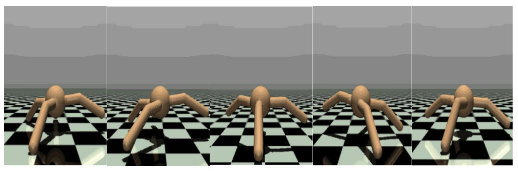
Figure 6: Reward (score) curves on Gym MuJoCo.



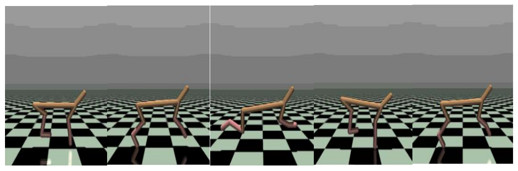
(a) Hopper-v2



(b) Humanoid-v2



(c) Ant-v2



(d) HalfCheetah-v2

Figure 7: Snapshots of SAC agents trained on Gym MuJoCo.

on the ground continuously, the human also learns to alternate leg lifts to move forward. For *Ant-v2* and *HalfCheetah-v2*, both agents are able to crawl at high speeds, corresponding with the high score they achieved in Figure 6.

5 Conclusion

In this paper, we do model-free control using both value-based and policy-based algorithms. Specifically, we implement and evaluate D3QN on three of Gym Atari environments with discrete action space. Besides, we also implement two representative policy-based algorithms: PPO (on-policy) and SAC (off-policy) and test them on four of continuous control environments in Gym MuJoCo. We also conduct ablation studies on D3QN to verify the significance of its components, and introduce PER to further boost SAC's performance.

The experiment results show that our proposed methods can achieve satisfying scores on all these 7 environments. Besides, the analysis results also suggest that 1) both double and dueling structures are necessary to achieve D3QN's performance, 2) the off-policy learning schema and PER can promote the sample efficiency of policy-based algorithms significantly.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

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

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

[4] V. Konda and J. Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[5] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang. A tutorial on energy-based learning. *Predicting structured data*, 1(0), 2006.

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

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

[8] G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*, volume 37. Citeseer, 1994.

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

[10] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[11] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

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

[13] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

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

[15] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[16] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

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

A Implementation Details

We present our implementation details here for reproducibility. For the environment requirements, we implement all our models with Python 3.8, Pytorch 1.8 and Gym 0.24.0. Besides, we need to install `opencv-python` and `atari-py` packages for value-based algorithms, and `mujoco-py` for policy-based algorithms. All of our experiments are run on a NVIDIA RTX 3090. Our OS platform is Linux 20.04.

A.1 Details for Value-Based Methods

For the value-based methods, we adopt the same setting for the three testing environments.

Typically, the architecture of the Q network is as follows:

```python
class QNet(nn.Module):
    def __init__(self, h, w, action_dim, device):
        super(QNet, self).__init__()
        self.conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4, bias=
            False)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=
            False)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=
            False)
        self.fc1 = nn.Linear(64*7*7, 512)
        self.fc2 = nn.Linear(512, action_dim)
```

As can be seen from the code above, we use CNNs to process the input state image, along with two linear layers to output the action value. The dueling network shares the same network architecture except that `fc2` is divided into a state value layer and a advantage layer:

```python
class DuelingQNet(nn.Module):
    def __init__(self, h, w, action_dim, device):
        super(DuelingQNet, self).__init__()
        self.conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4, bias=
            False)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=
            False)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=
            False)
        self.fc1 = nn.Linear(64 * 7 * 7, 512)
        self.adv = nn.Linear(512, action_dim)
        self.val = nn.Linear(512, 1)
```

Next, the training details and hyperparameter settings are listed as follows:

- We use the smooth L1 loss to compute the loss between the policy and target Q networks.
- We use mini-batch gradient descent to optimize all the models, and the batch size is 32. The Adam optimizer is adopted for optimization.

- We train the Q network for 10,000,000 steps, optimize the model for one batch after every 4 environment steps, and update the target network for every 10,000 steps.
- The size of the replay buffer is 200,000, and the optimization procedure starts when the replay buffer owns 500 pieces of experiences.
- We use the ϵ-greedy algorithm for exploration. Typically, ϵ is initialized as 1.0, and linearly drop to 0.1 after 1,000,000 steps.
- We evaluate the training agent for every 25,000 environment steps. Specifically, we greedily rollout 10 episodes to calculate the mean and min, max scores.
- The activation function is ReLU.
- The learning rate is set as 6.25e-5, and the discounted factor γ is set as 0.99.

A.2 Details for Policy-Based Methods

For the policy-based methods, we also adopt the same setting for the four testing environments.

First, for the network architecture of PPO algorithm:

```
class PPOActor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size=256):
        super(PPOActor, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.mu = nn.Linear(hidden_size, action_dim)
        self.log_std = nn.Parameter(torch.zeros(1, action_dim))

class VCritic(nn.Module):
    def __init__(self, state_dim, hidden_size):
        super(VCritic, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.v = nn.Linear(hidden_size, 1)
```

Note that we use one learnable scalar for each dimension of the log standard deviation of Gaussian action distribution. The training details and hyperparameter settings are listed as follows:

- We use the MSE loss to compute the loss of the critic.
- We use mini-batch gradient descent to optimize the actor as well as the critic, and the batch size is 64. The Adam optimizer is adopted for optimization.
- The number of total training steps depends on the specific environments. Typically, we run 10,000,000 steps for Humanoid, 3,000,000 steps for Ant and Halfcheetah, and 1,000,000 steps for Hopper. And we optimize the model for 10 epochs after every 2048 steps.
- The size of the replay buffer is 2048, exactly the same as the period of model optimization.
- We evaluate the training agent for every 2048 steps. Specifically, we rollout 10 episodes to calculate the mean and min, max scores.
- The activation function is Tanh.
- For the hyperparameters, the learning rate is set as 3e-4, the discounted factor γ is set as 0.95, the hidden layer size is 256, and the coefficient of entropy is 0.1. The clipping ratio ϵ of PPO is set as 0.2, and the λ for GAE is set as 0.95.

Second, for the network architecture of SAC:

```
class SACActor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size=256):
        super(SACActor, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.mu = nn.Linear(hidden_size, action_dim)
        self.log_std = nn.Linear(hidden_size, action_dim)

class QCritic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size=256):
        super(QCritic, self).__init__()
        self.fc1 = nn.Linear(state_dim+action_dim, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.q = nn.Linear(hidden_size, 1)
```

Different from PPO in which the critic needs to estimate the state value, the critic in SAC is designed to approximate the Q function. Besides, we use a linear layer to approximate the log standard deviation in SAC instead. The training details and hyperparameter settings are listed as follows:

- We use the MSE loss to compute the loss of the critic.
- We use mini-batch gradient descent to optimize the model, and the batch size is 256. The Adam optimize is adopted for optimization.
- The number of training steps is set the same as in PPO. However, we will optimize the model for one batch every step after the initialize 10,000 steps. Besides, the target network is soft updated after every environment step.
- The size of the replay buffer is 1,000,000.
- We evaluate training agent every 10,000 steps. Specifically, we rollout 10 episodes to calulate the mean, min and max scores.
- The activation function is ReLU. And we use Tanh to normalize the output action into [-1,1].
- For the hyperparameters, the learning rate is set as 3e-4, the discounted factor γ is set as 0.99, the hidden layer size is 256, and the coefficient of entropy is 0.2 (if we use the fixed strategy). The soft updated ratio τ is set as 0.005. Finally, the β_1 for PER is set as 0.6, while β_2 is set as 0.4. Besides, β_2 will increase 0.001 after every environment step until 1.0.

Acknowledgements

First, sincerely thank Prof. Zou for her guidance in reinforcement learning and excellent teaching, this project is interesting and I have learned a lot from this experience. It's really exciting when I render out the simulated results on the 7 testing environments and see the excellent performance of the trained agents. Second, thank TA Tianshi Zhang for his help in Atari environment understanding. Finally, thank you for reading till this end, hope this paper can inspire you to some extend.