Experience Replay More When It's a Key Transition in Deep Reinforcement Learning

Anonymous authors
Paper under double-blind review

Abstract

We propose a experience replay mechanism in Deep Reinforcement Learning based on Add Noise to Noise (AN2N), which requires agent to replay more experience containing key state, abbreviated as Experience Replay More (ERM). In the AN2N algorithm, we refer to the states where exploring more as the key states. We found that how the transitions containing the key state participates in updating the policy and Q networks has a significant impact on the performance improvement of the deep reinforcement learning agent, and the problem of catastrophic forgetting in neural networks is further magnified in the AN2N algorithm. Therefore, we change the previous strategy of uniform sampling of experience transitions. We sample the transition used for experience replay according to whether the transition contains key states and whether it is the most recently generated, which is the core idea of the ERM algorithm. The experimental results show that this algorithm can significantly improve the performance of the agent. We combine the ERM algorithm with Deep Deterministic Policy Gradient (DDPG), Twin Delayed Deep Deterministic policy gradient (TD3) and Soft Actor-Critic (SAC), and evaluate algorithm on the suite of OpenAI gym tasks, SAC with ERM achieves a new state of the art, and DDPG with ERM can even exceed the average performance of SAC under certain random seeds, which is incredible.

1 Introduction

Deep reinforcement learning (RL) has shown its promising feature for decision-making in various computer games, such as atari (Mnih et al., 2013; 2015), go (Schrittwieser et al., 2020) and starcraft (Vinyals et al., 2019). However, most successes have been exclusively in simulation largely due to poor sample efficiency of typical Deep RL algorithm and other challenges. Reinforcement learning can be divided into model based RL and model free RL in the light of its data efficiency. Model free RL is usually subdivided into off-policy RL and on-policy RL. Although Model based RL requires less sampled data, it needs to build a world model and predict the next state based on a lot of prior work, demonstrated in Hafner et al. (2019). Although the on-policy RL algorithm do without establishing a world model and has good stability, the performance improvement is slow due to limiting the update step of the policy, a large number of sampled trajectory data are required in the training process (Schulman et al., 2015; 2017). Off-policy RL is between Model based RL and On policy RL in terms of data efficiency, and the research in this field is enduring (Watkins & Dayan, 1992; Hessel et al., 2018; Barth-Maron et al., 2018) on account of it's relatively high data efficiency and world model free.

Experience Replay (Lin, 1992) is an important part of improving the data efficiency of Off-policy RL, which stores experience in a replay buffer and break the temporal correlations by mixing data, therefore, empirical data can be used multiple times to update the networks. However, most of the current work is to uniformly sample transitions from buffer, such as Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015), Soft Actor-Critic (SAC) (Haarnoja et al., 2018) and Twin Delayed Deep Deterministic policy gradient (TD3) (Fujimoto et al., 2018) and many other algorithms (Van Hasselt et al., 2016; Mnih

et al., 2016; Andrychowicz et al., 2017; Dabney et al., 2018; Liu et al., 2020). However, this approach replays experience transitions at the same frequency, regardless of their significance. Schaul et al. (2016) develops a framework for prioritizing experience, so as to replay important transitions more frequently, and therefore learn more efficiently. Prioritized experience replay (PER) method samples transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error. However, prioritization introduces bias, which needs to be corrected with importance sampling. Meta-reinforcement learning (meta-RL) algorithms enable agents to learn new skills from small amounts of experience (Rothfuss et al., 2018; Mishra et al., 2018), Rakelly et al. (2019) develops an off policy meta-RL algorithm that disentangles task inference and control, but its performance is still far behind the mainstream off policy RL.

In this paper, We first analyzed the state change of the agent from the beginning of interaction with the environment to the convergence of the policy, and found that the state of the agent is different at different stages. Agent will rarely transfer to terrible states[1] when the agent in a good [2]. Therefore, the proportion of the most recently generated empirical transitions should be appropriately increased, so that the agent pays more attention to learning the recent empirical transitions, In this way, the state of the agent is gradually transferred from a poor state to a better state, similar to the gradual update from a poor strategy to a better policy in Trust region policy optimization (TRPO) (Schulman et al., 2015) or Proximal policy optimization (PPO) (Schulman et al., 2017).

Inspired by the Add Noise to Noise (AN2N) Algorithm (Guo & Gao, 2021), We divide the states into two categories. These states that have been explored with added noise are called key states[3], and the rest are non-key states. For the sake of improving the agent's performance in key states in time, we have increased the probability of sampling new key states, making it more likely to participate in agent's policy updates, we call the process as ERM (Experience Replay More), and combine it with off policy RL algorithms commonly used in continuous control tasks, such as SAC, obtained faster learning and state-of-the-art performance.

2 Preliminaries

We consider a reinforcement learning setup consisting of an agent learning policies to maximize the expected reward when interacting with the environment (Sutton & Barto, 2018). At each timestep t, the agent receives an observation $o_t \in \mathcal{O}$, selects action $a_t \in \mathcal{A}$ with respect to its policy π: $\mathcal{O} \to \mathcal{A}$. After taking the action a_t in environment E, agent receives a reward r_t and the next observation o_{t+1}. The practical problem is usually a partial Markov decision process (POMDP), only part of the observation information could be obtained. To simplify the problem, we assumed the environment is fully-observed, so $s_t = o_t, \mathcal{S} = \mathcal{O}$.

In reinforcement learning, the action-value function $Q^\pi(s, a)$ is uesed to approximate the expected sum reward of the ation a in state s, defined as following:

$$Q^\pi(s, a) = \mathbb{E}_{s_t \sim p_\pi, a_t \sim \pi}\left[\sum_{t=0}^{+\infty} \gamma^t R(s_t, a_t)\right] \tag{1}$$

Where $\gamma \in [0, 1]$ is the discount factor, $\mathbb{E}_{s_t \sim p_\pi, a_t \sim \pi}$ is the expectation over the distribution of the trajectories $(s_0, a_0, s_1, a_1, \dots)$.

The mean value of Q^π in the same state s called the value function V^π, defined as $V^\pi(s) = \mathbb{E}_{a \sim \pi(\cdot|s)}[Q^\pi(s, a)]$. We express the action-value function Q^π in the form of Bellman equation (Bellman & Kalaba, 1965):

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim p_\pi}[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^\pi(s_{t+1}, a_{t+1})]] \tag{2}$$

[1]Agent with poor decision-making ability always be

[2]Agent with good decision-making ability always be

[3]The key state is a state where the agent has not performed well in the past

In this paper, we need to be familiar with DDPG, TD3 and SAC algorithms, here, we mainly introduce DDPG as the basis. DDPG applied two different fully connected neural networks to approximate the action-value function $Q(s, a|\theta^Q)$ and policy function $\mu(s|\theta^\mu)$, DDPG introduces action-value target network $\theta^{Q^{'}}$ and policy target network $\theta^{\mu^{'}}$, so as to Stable the policy update. Consequently, gradient descent is used to optimize the network weights by minimizing the loss:

$$L(\theta^Q) = \mathbb{E}_{s_t \sim p_{\mu(s_t|\theta^\mu)}, a_t \sim \mu(s_t|\theta^\mu)}[(Q(s_t, a_t|\theta^Q) - y_t)^2] \tag{3}$$

Where

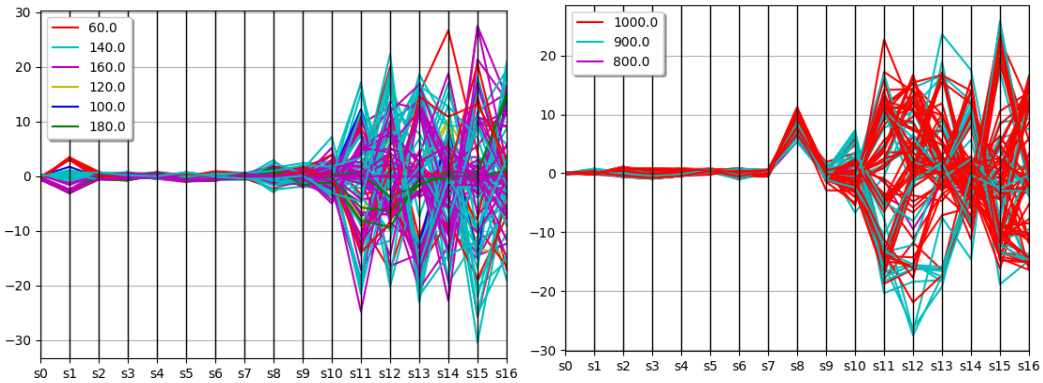
$$y_t = r(s_t, a_t) + \gamma Q^{'}(s_{t+1}, \mu^{'}(s_{t+1}|\theta^{\mu^{'}})|\theta^{Q^{'}}) \tag{4}$$

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s \sim p_{(s_t|\theta^\mu)}} \left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s \sim p_{(s_t|\theta^\mu)}}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s_t|\theta^\mu)|s = s_t] \end{aligned} \tag{5}$$

Where equation 4 derived from equation 2, the weights of target networks are updated periodically to slowly track the learned networks: $\theta^{'} \leftarrow \tau\theta + (1-\tau)\theta^{'}$ with $\tau \ll 1$, which alleviates the fluctuation in the agent's learning process. The policy is updated by equation 5, following the chain rule to the expected sum return $Q(s, a|\theta^Q)$ with respect to parameters θ^μ. TD3 addresses the problem that DDPG is prone to overestimating the Q function. An additional Q-function is added, and the predicted smaller Q value is used as the calculation of TD-error. At the same time, which also reduces the update frequency of the policy function. The most significant difference between SAC and DDPG is the introduction of policy entropy $H(\pi(\cdot|\mathrm{s}_t))$, so the objective function equation 1 is rewritten as:

$$Q^\pi(s, a) = \mathbb{E}_{s_t \sim p_\pi, a_t \sim \pi}\left[\sum_{t=0}^{+\infty} \gamma^t R(s_t, a_t) - \alpha \log(\pi(a_t|s_t))\right] \tag{6}$$

Where α is temperature parameter, which adjusts the optimization target, agent pays more attention to exploration if increase the coefficient α.



(a) States between 2e4 and 2.4e4 steps (b) States between 5.6e5 and 5.64e5 steps

Figure 1: In the HalfCheetah-v2 environment, the state of the agent in different training stages is compared. The abscissa is the collected information at different positions of the agent, and the ordinate is the specific value of the state, the difference is obvious at states s1, s8 and s12. Different colors of legend indicate different Q values.(a) Start collecting at 2e4.(b) Start collecting at 5.6e5.

3 Experience Replay More When It's a Key Transition

The addition of experience replay improves the sample efficiency of off policy RL. Many off policy algorithms usually sample transitions uniformly from the experience buffer, which po-

tentially considers the transitions generated at different times to be of the same importance, we will discuss this in section 3.1. Prioritized Experience Replay (PER) is an optimization of the uniform sampling method based on the TD-error value of transitions, so as to learn the samples more efficiently, however, PER essentially still does not consider whether the transitions generated at different times are equally important to the current agent's policy.

3.1 States are Different at Different Stages

Taking the HalfCheetah-v2 simulation environment as an example, we recorded the state information of an agent using the DDPG algorithm at different stages, starting from 2e4 and 5.6e5 to collect the state and the corresponding Q value, collecting 10 complete episodes for each simulation, and then we uniformly sample 100 sets of data from the collected data for drawing, as shown in the Fig. 1. The legend in the upper left corner represents the Q value, which is discretized in the subregions to reasonably reduce the number of Legends. Subgraph (a) is discrete in units of 20, and subgraph b is discrete in units of 100. It can be found when the agent interacts with the environment, not only the policy is gradually improving[4], but the state is also changing accordingly.

After the agent's policy converges and is in a different state set from the previous one, more data similar to the agent's recent state should be collected to train the neural network to strengthen the current policy.If the data of the training strategy network and the action state network are quite different from the agent's recent state, it will not only help the agent's policy improvement to a small extent, but will also forget the newly learned network weights due to the catastrophic forgetting of the neural networks. Therefore, For the purpose of further improving the sample efficiency, experience replay should be switched from uniform sampling to targeted sampling.

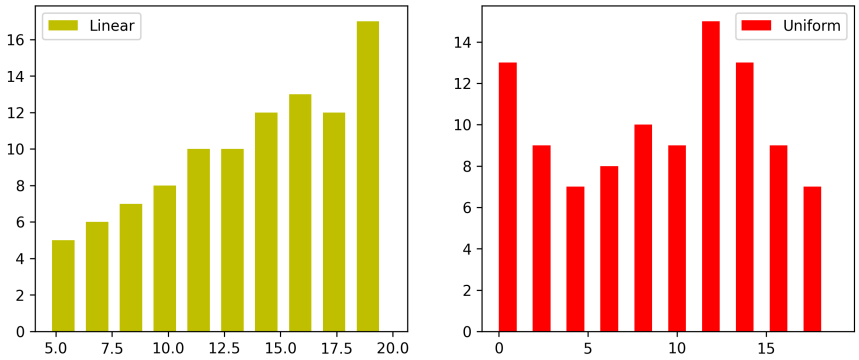


Figure 2: Compare the sampling results of the linear distribution sampling function and the uniform distribution sampling function. The left picture shows the linear distribution sampling, and the right picture shows another. The sampling interval is $[0, 20]$, and the sampling number is 100.

3.1.1 Replaying More Key Transitions

The AN2N algorithm will record the state s_d that is in a dilemma during the evaluation process, and when interacting with the environment, if the current state s_c and s_d are considered to be highly similar, an additional exploration noise $\mathcal{N}_a$ will be added. In this paper, the state s_c with additional noise is called the key state s_k, and the transition containing s_k is called the key transition $tran_k$. The agent's policy is relatively fragile in s_k, if it has not learned how to get out of this dilemma before, agent is more likely to fall into a series of bad states after s_k, which greatly reduces the agent's overall performance.

[4]Better decision-making ability can get a higher Q value

Therefore, whether the agent can learn a good policy in s_k is very important to improve the performance of the agent.

It can be seen from equation 2 that reward of the key state s_k will affect the Q value of the state at the previous moment through the iteration of the dynamic equation, nevertheless, if the data used to update the Q value network is collected from the experience buffer by uniform sampling, there will be the following three problems:

- Since most of the data generated by AN2N are non-key states, the probability of key states s_k being sampled in the experience buffer is small.
- The key state s_k is time-sensitive. The new key state needs to participate in the training of the network as quickly as possible. Otherwise, if agent's state undergoes a relatively change, its role in improving the participation strategy will decline.
- Since the state is gradually changing, more recent the generated experience, more similar its distribution is to the distribution of the latest experience generated currently, and the more it satisfies the assumption of independent and identical distribution (iid).

In response to the first question, we designed two experience buffers to store key transition and non-key transition respectively, and use $\min(Prt_{AN2N}, K_t)$ to adjust the proportion of sampling key transition, where Prt_{AN2N} is the proportion of the number of key transitions generated by AN2N, K_t is linearly related to the simulation times of the agent. The above ensures that key transitions can be sampled strictly according to the proportion from the experience buffer. For the second and third questions, we will linearly increase the probability of new transition being sampled. Two sampling functions will sample 100 from $[0, 20]$ to compare the difference of The linear distribution sampling function and the uniform distribution more vividly, the result Shown in Fig. 2. The pseudo code of ERM algorithm is shown in algorithm 1.

Algorithm 1: ERM

```
Input: Sampling ratio Prt_AN2N, K_t, Replay buffer R_non-key, R_key, batch size bs_1, bs_2,
       bs_sum and AN2N parameters
Randomly initialize critic network Q(s, a|θ^Q) and actor μ(s|θ^μ) with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights θ^Q' ← θ^Q, θ^μ' ← θ^μ
for episode e ∈ {1,...,M} do
    Initialize a random process N for action exploration
    Receive initial observation state s_1
    for t ∈ {1,...,T} do
        Execute AN2N action a_t and observe reward r_t and observe new state s_{t+1}
        if (AN2N exploring more) then
            Store key transitio (s_t, a_t, r_t, s_{t+1}) in R_key
        else
            Store transition (s_t, a_t, r_t, s_{t+1}) in R_non-key
        end
        bs_1 = min(Prt_AN2N, K_t)
        bs_2 = bs_sum − bs_1
        Sample bs_1 and bs_2 transitions with linear distribution in R_key and R_non-key for
          training
        Run DDPG, SAC or TD3 etc. Algorithm
    end
end
```

4 Experiments

In this section, we describe the performance of the combinations of Experience Replay More algorithm (ERM) with different off policy RL algorithms across a variety of continuous control tasks. As is shown in Fig. 3, consistent with the benchmark, we use the mainstream Mujoco physics engine (Todorov et al., 2012) as the simulation environment to test

the performance of the algorithm in the HalfCheetah-v2, Swimmer-v2, Walker2d-v2, and Hopper-v2 tasks, as Mujoco offers a unique combination of speed, accuracy and modeling power, and it is also the first full-featured simulator designed from the ground up for the purpose of motion control. For the specific introduction of the task environment is shown in Table 2 in Appendix A.

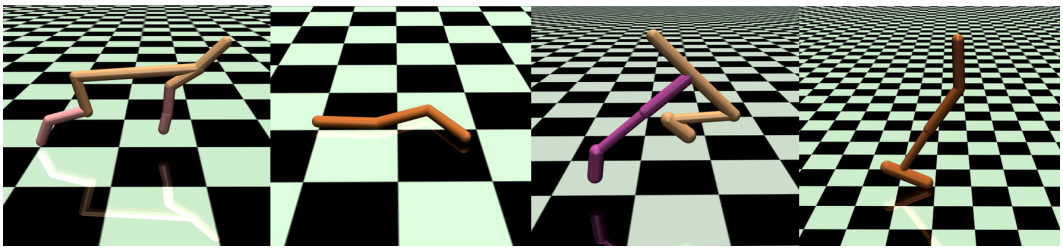
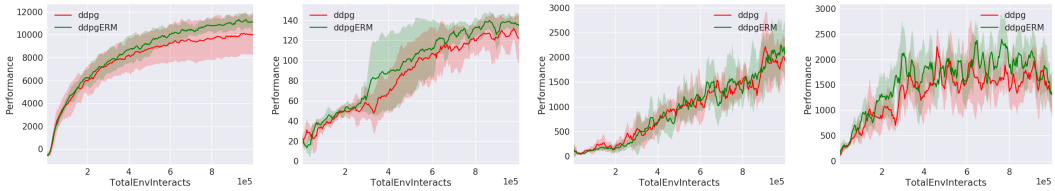


Figure 3: Samples of Mujoco tasks. In order from the left: HalfCheetah-v2, Swimmer-v2, Walker2d-v2, Hopper-v2.

We present the ERM, which builds on the Add Noise to Noise algorithm (AN2N) by classifying transitions and sampling experience with linear distribution functions, described in section 3, to increase the stability and performance with sampling efficiency. In each task, we run our algorithm and periodically fix the policy to test it without exploration noise. The goal of our experimental evaluation is to understand how the sample complexity and stability of our method compares with prior off-policy RL algorithms. In all tasks, we run experiments using state description (such as joint angles and positions) for five times, which fix random seeds in 0, 5, 10, 15, 20 respectively. The results of ERM combined with different off policy RL algorithms are analyzed below.

4.1 DDPG with ERM

For the implementation of DDPG with ERM, we use a two layer fully connected network of 256×256 hidden nodes respectively, with rectified linear units (ReLU) between each layer for both the policy and action state networks, and a final tanh unit following the output of the policy network for limiting amplitude. After a certain number of steps, the networks are trained with a mini-batch of a totall 100 transitions, sampled uniformly from a replay buffer containing the entire history of the agent. See Appendix A for more experimental details.



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

Figure 4: Performance curves for a selection of domains using DDPG and DDPG with ERM: DDPG (red), DDPG with ERM (green).

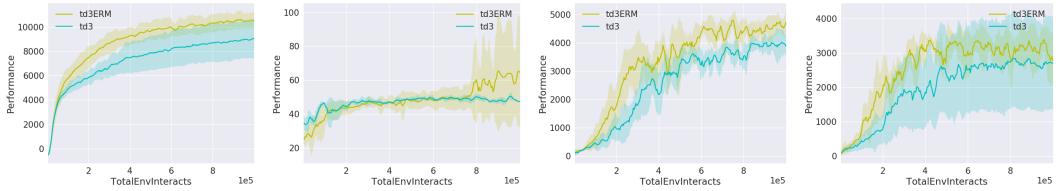
DDPG uses two actor-networks and critic-networks respectively to approximate the policy and action-state value. In the test stage, it records the reward of each state of the agent, and then calculates all the action state value of the trajectory when the episode finished, save a batch transitions whose totall reward is minimal. When the agent interacts with the environment, it uses the policy of superimposing disturbance noise, if the current state is similar to key state, add a disturbance noise on small noise, otherwise, only use a small noise. Besides, we store the key tansitions and non-key transitions separatly, which is

convenient for sampling two kinds of transitions in a appropriate proportion with linear distribution sampling function. The pseudo code of DDPG with ERM is shown in algorithm 2 in Appendix B.

We compare the DDPG with ERM algorithm with DDPG baseline, illustrated in Fig. 4, In four tasks, the performance of DDPG with ERM is higher. In the HalfCheetah task, the method combined with ERM has significantly lower variance while maintaining higher performance, indicating that the algorithm we proposed with DDPG has better stability and sample efficiency.

4.2 TD3 with ERM

TD3 is an optimized version on the basis of DDPG, and the structure is very similar to the DDPG algorithm. The main difference is that three improvements are proposed for the problems of DDPG in engineering practice: 1.Clipped Double-Q Learning. TD3 uses two Q networks with the same structure to predict the state action value at the same time, but when calculating TD-error, only the Q value with the smallest prediction participates in the calculation, so as to alleviate the overestimation of DDPG. 2.Delayed Policy Updates. Since updating the policy frequently is prone to make the policy unstable, therefore, TD3 updates the policy with a lower frequency. 3.Target Policy Smoothing. When calculating target Q, a small range of noise is added to the policy to make the calculated target Q value more robust. Therefore, TD3 with ERM and DDPG with ERM can maintain the same network structure and hyper-parameters. See Appendix A for more details.



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

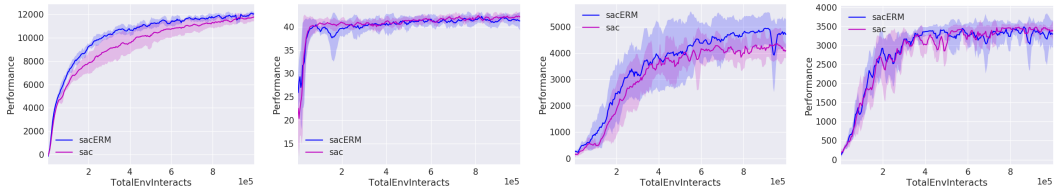
Figure 5: Performance curves for a selection of domains using TD3 and TD3 with ERM: TD3 (cyan), TD3 with ERM (yellow).

In the same four tasks, we tested the performance of TD3 with ERM and TD3. As illustrated in Fig. 5. In all the test tasks, the performance improvement of TD3 with ERM is very obvious compared to TD3, especially in HalfCheetah and Walker2d tasks. At the same time, in the HalfCheetah and Hopper tasks, the method of combining TD3 and ERM has significantly lower variance while maintaining higher performance, indicating that the algorithm we proposed is also very stable and sample efficient on TD3.

4.3 SAC with ERM

Compared with TD3, SAC is more different from DDPG, but it is still an Actor-Critic structure. The main differences are as follows: 1.The policy entropy is added to the objective function, which can more fully explore the action space, but there is also an additional temperature coefficient that adjusts the entropy proportion of the policy. 2.SAC does not directly outputting the deterministic strategy, but the mean value and variance of the policy, and then sampling policy, so it is necessary to adjust the size of the variance to achieve a noise-like addition. See Appendix A for the specific hyper-parameter settings of SAC.

The SAC algorithm is currently one of the most commonly used off policy RL methods in academia and industry owing to its good performance, stability and sample efficiency. We tested the performance of SAC with ERM and SAC in four tasks, as shown in Fig. 6, we found that in half of the tasks, the performance of SAC with ERM still has a performance improvement compared to sac, while the performance of the remaining tasks was flat, in-

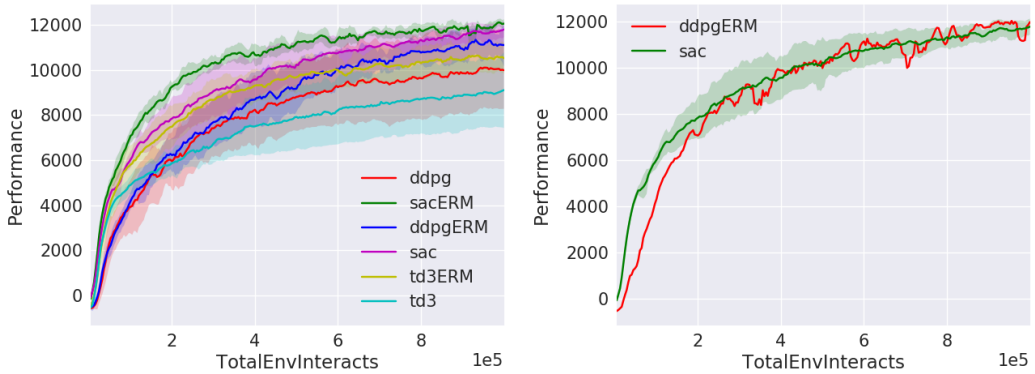


(a) HalfCheetah-v2 (b) Swimmer-v2 (c) Walker2d-v2 (d) Hopper-v2

Figure 6: Performance curves for a selection of domains using SAC and SAC with ERM: SAC (purple), SAC with ERM (blue).

dicating that the algorithm we proposed also has great stability and sample efficiency on SAC.

In addition, we summarize the results of the above several algorithms in the HalfCheetah simulation environment. As shown in the sub-graph (a) in Fig. 7. We run agent with each algorithm for 1 million time steps with evaluations every 4000 time steps, where each evaluation reports the average reward over 10 episodes with no exploration noise. Our results are reported over 5 random seeds of the Gym simulator. It can be seen that the performance of SAC with ERM is the best, surpassing state of the art (sac), and its convergence speed is also the fastest, indicating that it has higher sample efficiency compared with other off policy RL algorithms.



(a) HalfCheetah-v2 (b) HalfCheetah-v2

Figure 7: Performance curves for HalfCheetah-v task using different algorithms: (a)DDPG (red), DDPG with ERM (blue), TD3 (cyan), TD3 with ERM (yellow), SAC (purple), SAC with ERM (green) (b) SAC (green), DDPG with ERM (red).

We take the best results of the five random seed tests in DDPG with ERM, whose random seed is 5, we compare it with the average performance of SAC, as shown in the subgraph (b) in Fig. 7. It can be found that although the convergence speed of ddpg with ERM is slower in the early stage, its performance has approached or even exceeded the average performance of SAC in the later stage. This improvement is very incredible for DDPG.

We display the statistical data of all the experimental results in Table 1. The first column indicates the algorithm or policy, among which Random indicates that the agent uses a random policy to interact with the environment. The numbers in the table represent the average cumulative rewards obtained by the corresponding algorithm or policy in the environment. The numbers in bold are the highest performance scores. It can be seen that the SAC with ERM algorithm has the highest score, followed by the DDPG with ERM algorithm. ERM has the greatest effect on improving the performance of TD3. From the statistical average

cumulative reward results, ERM is helpful to the performance improvement of DDPG, TD3 and SAC.

Table 1: Mean value of the total reward of agent in different tasks

Environment	HalfCheetah-v2	Swimmer-v2	Walker2d-v2	Hopper-v2
Random	-283±29	1 ± 4	2 ± 2	19 ± 6
DDPG	7790±2058	84 ± 26	920 ± 550	1313 ± 867
DDPGERM	8415±1161	**94 ± 24**	933 ± 578	1548 ± 861
SAC	9452±984	41 ± 2	3163 ± 951	2856 ± 502
SACERM	**10219 ± 645**	40 ± 3	**3564 ± 1216**	**2883 ± 543**
TD3	7240±1455	58 ± 31	2568 ± 733	1939 ± 1442
TD3ERM	8795±1023	48 ± 13	3305 ± 966	2553 ± 852

The most time-consuming part of the algorithm is the calculation of the state similarity. For this reason, we accelerate the process by using matrix operation, which expands the current state dimension to $\boldsymbol{S}_c$ and calculates the similarity with the matrix $\boldsymbol{S}_k$ composed of all key states. Therefore, in the case of a small increase in time consumption, the performance of the algorithm is significantly improved, especially on the HalfCheetah task.

5 Conclusion

This work divides the states of the agent into key states and non-key states, and analyzes the reasons why the recent generated key states need to be sampled and trained as soos as possible. For the purpose of sampling key transitions more accurately, we introduce two experiences memory stores the key transitions and non-key transitions respectively, and set a adjustable coefficient to determine the proportion of transitions sampled from two experience memories. Besides, we analyze the advatages of sampling the key transitions if we make use of linear distribution function from two perspectives : 1 The distribution of transitions sampled recently are more similar to the distribution of latest experience generated, making transitions more satisfie the assumption of independent and identical distribution (iid); 2. The recent generated experience can be used for the training of the action state network and policy network of the agent more quickly, so as to make up for the lack of the policy in time. Finally, on the basis of AN2N, the combination of ERM method and DDPG, TD3 or SAC algorithm has a very obvious performance improvement on tasks such as HalfCheetah. The performance of the DDPGERM algorithm with some random seeds can even exceed the average performance of SAC, and SACERM has also outperforms the current state of the art.

Acknowledgments

We would like to thank Feng Pan, Weixing Li, Xiaoxue Feng, Yan Gao and many others at Institute of Pattern Recognition and Intelligent System of BIT for insightful discussions and feedback.

References

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 5055–5065, 2017.

Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, TB Dhruva, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. In International Conference on Learning Representations, 2018.

Richard Bellman and Robert E Kalaba. Dynamic programming and modern control theory, volume 81. Citeseer, 1965.

Rémi Coulom. Reinforcement learning using neural networks, with applications to motor control. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2002.

Will Dabney, Mark Rowland, Marc G Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

Tom Erez, Yuval Tassa, and Emanuel Todorov. Infinite horizon model predictive control for nonlinear periodic tasks. Manuscript under review, 4, 2011.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In Jennifer Dy and Andreas Krause (eds.), Proceedings of the 35th International Conference on Machine Learning, volume 80 of Proceedings of Machine Learning Research, pp. 1587–1596. PMLR, 10–15 Jul 2018. URL http://proceedings.mlr.press/v80/fujimoto18a.html.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Proceedings of the fourteenth international conference on artificial intelligence and statistics, pp. 315–323. JMLR Workshop and Conference Proceedings, 2011.

Youtian Guo and Qi Gao. Exploring more when it needs in deep reinforcement learning. arXiv preprint arXiv:2109.13477, 2021.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In International conference on machine learning, pp. 1861–1870. PMLR, 2018.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In International Conference on Learning Representations, 2019.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In Thirty-second AAAI conference on artificial intelligence, 2018.

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

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. arXiv preprint arXiv:1509.02971, 2015.

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

Yao Liu, Adith Swaminathan, Alekh Agarwal, and Emma Brunskill. Provably good batch off-policy reinforcement learning without great exploration. Advances in Neural Information Processing Systems, 33:1264–1274, 2020.

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In International Conference on Learning Representations, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 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.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In International conference on machine learning, pp. 1928–1937. PMLR, 2016.

Kate Rakelly, Aurick Zhou, Chelsea Finn, Sergey Levine, and Deirdre Quillen. Efficient off-policy meta-reinforcement learning via probabilistic context variables. In International conference on machine learning, pp. 5331–5340. PMLR, 2019.

Jonas Rothfuss, Dennis Lee, Ignasi Clavera, Tamim Asfour, and Pieter Abbeel. Promp: Proximal meta-policy search. In International Conference on Learning Representations, 2018.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In ICLR (Poster), 2016.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. Nature, 588(7839): 604–609, 2020.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In International conference on machine learning, pp. 1889–1897. PMLR, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. MIT press, 2018.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 5026–5033. IEEE, 2012.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence, volume 30, 2016.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. Nature, 575 (7782):350–354, 2019.

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

Paweł Wawrzyński and Ajay Kumar Tanwani. Autonomous reinforcement learning with experience replay. Neural Networks, 41:156–167, 2013.

A Experiment Details

The experiment uses a unified parameter configuration for all algorithms, the two hidden layers are composed of 256 nodes and the neural networks use the rectified non-linearity (Glorot et al., 2011) for all hidden layers. The activation function of the actor output layer uses a tanh for bounding the actions. Adam (Kingma & Ba, 2014) optimizes the weights of the neural networks with a learning rate of 10^3 for the both of actor and critic networks, discount factor set to 0.99. For the soft target updates, we set $\tau = 0.005$. At the beginning, the agent uses a random policy to generate 10^4 transitions for the initial training of the network, the proportion of sampling key transitions is fixed by $\min(Prt_{AN2N}, K_t)$, where $Prt_{AN2N} = 0.4 - 0.2 * \frac{t}{totaltime}$, $K_t = 50 * \frac{t}{totaltime}$. We train the networks with minibatch sizes of 100 for all of the tasks, and use a replay buffer size of 10^6. For DDPG or TD3 with ERM, we set two kind noise $\mathcal{N}_{\text{big}} = 0.4$ and $\mathcal{N}_{\text{small}} = 0.05$, For SAC with ERM, different from the policy generation method of DDPG and TD3, SAC generates the mean and variance of the policy, and then sample a specific action. Therefore, we set the expansion and reduction coefficients of the variance to 1.5 and 0.5.

We test our algorithm in the Mujoco task environment. The state dimension, observation dimension, and action dimension of the tasks are shown in Table 2, and the detailed description of the tasks are shown in Table 3.

Table 2: Dimensionality of the MuJoCo tasks environment: the dimensionality of the underlying physics model dim(s), number of action dimensions dim(a) and observation dimensions dim(o).

Task name	HalfCheetah-v2	Swimmer-v2	Walker2d-v2	Hopper-v2
dim(s)	18	27	18	14
dim(a)	6	8	6	4
dim(o)	17	111	41	14

Table 3: The description of the Mujoco tasks environment

Task name	Brief description
HalfCheetah-v2	The agent should move forward as quickly as possible with a cheetah like body that is constrained to the plane (Wawrzyński & Tanwani, 2013).
Swimmer-v2	This task involves a 3-link swimming robot in a viscous fluid, where the goal is to make it swim forward as fast as possible (Coulom, 2002).
Walker2d-v2	The agent's goal is making a two-dimensional bipedal robot walk forward as fast as possible (Erez et al., 2011).
Hopper-v2	The agent's goal is making a two-dimensional one-legged robot hop forward as fast as possible (Erez et al., 2011).

B Algorithm

Appendix B includes two parts: the first part is the overall structure diagram of the Experience Replay More (ERM) algorithm 8. The core of the entire algorithm is to classify and store the data stream, and sample the experience with a linear distribution function. The second part is the pseudo code of the algorithm. ,: algorithm pseudo code of DDPG with ERM in Algorithm 2, algorithm pseudo code of TD3 with ERM in Algorithm 3 and algorithm pseudo code of SAC with ERM in Algorithm 4.

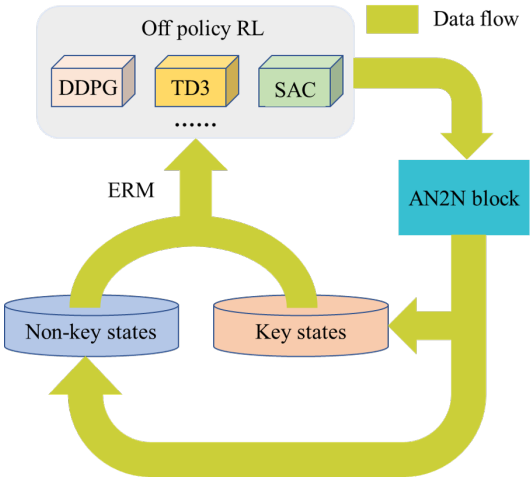


Figure 8: The overall structure diagram of the Experience Replay More (ERM) algorithm.

Algorithm 2: DDPG with ERM

Input: Sampling ratio Prt_{AN2N}, K_t, Replay buffer $R_{non-key}$, R_{key}, batch size bs_1, bs_2, bs_{sum} and AN2N parameters

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ

Initialize target network $Q^{'}$ and $\mu^{'}$ with weights $\theta^{Q^{'}} \leftarrow \theta^Q$, $\theta^{\mu^{'}} \leftarrow \theta^\mu$

for episode $e \in$ {1,...,M} do

  Initialize a random process $\mathcal{N}$ for action exploration

  Receive initial observation state s_1

  for $t \in$ {1,...,T} do

    Execute AN2N action a_t and observe reward r_t and observe new state s_{t+1}

    if (AN2N exploring more) then

      Store key transitio (s_t, a_t, r_t, s_{t+1}) in R_{key}

    else

      Store transition (s_t, a_t, r_t, s_{t+1}) in $R_{non-key}$

    end

    $bs_1 = \min(Prt_{AN2N}, K_t)$

    $bs_2 = bs_{sum} - bs_1$

    Sample bs_1 and bs_2 transitions with linear distribution in R_{key} and $R_{non-key}$ for training

    if t mod u then

      Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

      Set $y_i = r_i + \gamma Q^{'}(s_{i+1}, \mu^{'}(s_{i+1}|\theta^{\mu^{'}})|\theta^{Q^{'}})$

      Update critic by minimizing the loss:$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

      Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i$$

      Update the target networks:

$$\theta^{Q^{'}} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q^{'}}$$

$$\theta^{\mu^{'}} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{Q^{\mu^{'}}}$$

    end

  end

end

Algorithm 3: TD3 with ERM

Input: Sampling ratio Prt_{AN2N}, K_t, Replay buffer $R_{non-key}$, R_{key}, batch size bs_1, bs_2, bs_{sum} and AN2N parameters

Randomly initialize critic networks $Q(s, a|\theta_1^Q)$, $Q(s, a|\theta_2^Q)$, and actor $\mu(s|\theta^\mu)$ with weights θ_1^Q, θ_2^Q and θ^μ

Initialize target network $Q^{'}$ and $\mu^{'}$ with weights $\theta_1^{Q^{'}} \leftarrow \theta_1^Q$, $\theta_2^{Q^{'}} \leftarrow \theta_2^Q$, $\theta^{\mu^{'}} \leftarrow \theta^\mu$

for episode $e \in$ {1,...,M} do

  Initialize a random process $\mathcal{N}$ for action exploration

  Receive initial observation state s_1

  for $t \in$ {1,...,T} do

    Execute AN2N action a_t and observe reward r_t and observe new state s_{t+1}

    if (AN2N exploring more) then

      Store key transitio (s_t, a_t, r_t, s_{t+1}) in R_{key}

    else

      Store transition (s_t, a_t, r_t, s_{t+1}) in $R_{non-key}$

    end

    $bs_1 = \min(Prt_{AN2N}, K_t)$

    $bs_2 = bs_{sum} - bs_1$

    Sample bs_1 and bs_2 transitions with linear distribution in R_{key} and $R_{non-key}$ for training

    if t mod u then

      Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

      Get $\tilde{a} \leftarrow \mu(s|\theta^{\mu^{'}}) + \epsilon, \quad \epsilon \sim \text{clip}\ (\mathcal{N}(0, \tilde{\sigma}), -c, c)$

      Set $y_i = r_i + \gamma \min_{j=1,2} Q^{'}(s_{i+1}, \mu^{'}(s_{i+1}|\tilde{a})|\theta_j^{Q^{'}})$

      Update critic by minimizing the loss:$L = \min_{\theta_j} \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta_j^Q))^2$

      if t mod $(d \times u)$ then

        Update the actor policy $\mu(s|\theta^\mu)$ using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta_1^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|s_i$$

        Update the target networks:

$$\theta_j^{Q^{'}} \leftarrow \tau\theta_j^Q + (1-\tau)\theta_j^{Q^{'}}$$
$$\theta^{\mu^{'}} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{Q^{\mu^{'}}}$$

      end

    end

  end

end

Algorithm 4: SAC with ERM

Input: Sampling ratio Prt_{AN2N}, K_t, Replay buffer $R_{non-key}$, R_{key}, batch size bs_1, bs_2, bs_{sum} and AN2N parameters, Temperature parameter α

Randomly initialize critic networks $Q(s, a|\theta_1^Q)$, $Q(s, a|\theta_2^Q)$, and actor $\mu(s|\theta^\mu)$ with weights θ_1^Q, θ_2^Q and θ^μ

Initialize target network $Q^{'}$ and $\mu^{'}$ with weights $\theta_1^{Q^{'}} \leftarrow \theta_1^Q$, $\theta_2^{Q^{'}} \leftarrow \theta_2^Q$

for episode $e \in$ {1,...,M} do

  Initialize a random process $\mathcal{N}$ for action exploration

  Receive initial observation state s_1

  for $t \in$ {1,...,T} do

    Execute AN2N action a_t and observe reward r_t and observe new state s_{t+1}

    if (AN2N exploring more) then

      Store key transitio (s_t, a_t, r_t, s_{t+1}) in R_{key}

    else

      Store transition (s_t, a_t, r_t, s_{t+1}) in $R_{non-key}$

    end

    $bs_1 = \min(Prt_{AN2N}, K_t)$

    $bs_2 = bs_{sum} - bs_1$

    Sample bs_1 and bs_2 transitions with linear distribution in R_{key} and $R_{non-key}$ for training

    if t mod u then

      Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

      Set $y_i = r_i + \gamma(\min_{j=1,2} Q^{'}(s_{i+1}, \mu(s_{i+1}|\theta^\mu)|\theta_j^{Q^{'}}) - \alpha \log \mu(s_{t+1}|\theta^\mu))$

      Update critic (soft Q-function) by minimizing the loss:

$$L = \frac{1}{N} \sum_i \sum_j (y_i - Q(s_i, a_i|\theta_j^Q))^2$$

      Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i ((\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|s_i$$
$$-\nabla_a \log \mu(s_t|\theta^\mu)) - \nabla_{\theta^\mu} \log \mu(s_t|\theta^\mu)|s_i)$$

      Update the target networks:

$$\theta_j^{Q^{'}} \leftarrow \tau\theta_j^Q + (1-\tau)\theta_j^{Q^{'}}$$

    end

  end

end