Off-Policy Correction for Deep Deterministic Policy Gradient Algorithms via Batch Prioritized Experience Replay

Dogan C. Cicek*, Enes Duran*, Baturay Saglam*, Furkan B. Mutlu* and Suleyman S. Kozat*†

*Department of Electrical and Electronics Engineering, Bilkent University, Ankara, Turkey
†*Senior Member, IEEE*
{cicek, enesd, baturay, burak.mutlu, kozat}@ee.bilkent.edu.tr

***Abstract*—The experience replay mechanism allows agents to use the experiences multiple times. In prior works, the sampling probability of the transitions was adjusted according to their importance. Reassigning sampling probabilities for every transition in the replay buffer after each iteration is highly inefficient. Therefore, experience replay prioritization algorithms recalculate the significance of a transition when the corresponding transition is sampled to gain computational efficiency. However, the importance level of the transitions changes dynamically as the policy and the value function of the agent are updated. In addition, experience replay stores the transitions are generated by the previous policies of the agent that may significantly deviate from the most recent policy of the agent. Higher deviation from the most recent policy of the agent leads to more off-policy updates, which is detrimental for the agent. In this paper, we develop a novel algorithm, Batch Prioritizing Experience Replay via KL Divergence (KLPER), which prioritizes batch of transitions rather than directly prioritizing each transition. Moreover, to reduce the off-policyness of the updates, our algorithm selects one batch among a certain number of batches and forces the agent to learn through the batch that is most likely generated by the most recent policy of the agent. We combine our algorithm with Deep Deterministic Policy Gradient and Twin Delayed Deep Deterministic Policy Gradient and evaluate it on various continuous control tasks. KLPER provides promising improvements for deep deterministic continuous control algorithms in terms of sample efficiency, final performance, and stability of the policy during the training.**

***Index Terms*—deep reinforcement learning, experience replay, prioritized sampling, continuous control, off-policy learning**

I. Introduction

Deep Reinforcement Learning techniques have shown notable success on tasks that require sequential decision making. Deep Reinforcement learning agents reach the superhuman-level performance on ATARI Games [1], continuous control tasks [2], board games [3], and real-time strategy games [4]. Coupling Reinforcement Learning with Deep Learning enables the agent to learn a parameterized policy and converge to a nearly optimal policy without visiting each state-action pair [5]. On the other hand, feeding the neural network that generates the policy of the agent by temporally correlated inputs violates the i.i.d assumption of the stochastic gradient-based optimization algorithms. Experience Replay tackles the given problem and breaks the temporal correlation by stacking the transitions to a replay buffer, then picking mini-batches among them randomly [6]. Due to this process, the agent learns from the transitions that are collected from various states of the state space of the task. The works show that the utilization of experience replay provides improvements to the agent in terms of sample efficiency and the stability of the policy [1], [7], [8], [9], [10].

Vanilla Experience Replay (Vanilla ER) algorithm samples transitions from the replay buffer randomly. By uniformly sampling the transitions, the algorithm assumes that the importance of each transition is equal to each other. However, the study shows that the strategy on how the agent's experiences are used during the training drastically affects the performance of the agent [11]. Several algorithms are suggested on how experiences should be replayed by assigning sampling probabilities and yield promising results [12], [13], [14], [15], [16].

In this paper, we introduce a novel experience replay prioritization method, Batch Prioritized Experience Replay via KL Divergence, KLPER. We approach the experience replay prioritization problem by prioritizing the sampled batches of transitions rather than assigning sampling probabilities to the transitions. The main drawback of prioritizing the transitions is that the importance of a transition can significantly change until the transition is sampled again. Therefore, the sampling probabilities of the transitions may not be proportional to their actual importance. In addition, as the policy of the agent changes, the replay buffer contains more off-policy transitions. It has shown that more off-policy updates induce divergence and negatively affect the performance of the agent [12], [17]. Hence, our algorithm forces the agent to learn through the batch of transitions that are more likely generated by the policy of the agent. We assume that each batch has a policy, Batch Generating Policy, that generalizes the past policies of the agent that collected the transitions in the batch. Then, we define the Batch Generating Policy for each batch with respect to the most recent policy of the agent that. We use the KL Divergence between Batch Generating Policy and a multivariate Gaussian distribution with a mean of 0 as a proxy to measure the deviation between the Batch Generating Policy and the most recent policy of the agent.

We evaluate KLPER by coupling it with the Deep Determin-

2375-0197/21/$31.00 ©2021 IEEE
DOI 10.1109/ICTAI52525.2021.00199

istic Policy Gradient and the Twin Delayed Deep Deterministic Policy Gradient algorithms. We compare our algorithm with Prioritized Experience Replay and Vanilla ER algorithms, on OpenAi Gym's continuous control tasks [18].

The main contributions of this paper are summarized as follows:

- Prioritize one batch among certain number of batches that sampled from the replay buffer at each iteration.
- Define Batch Generating Policy with respect to the most recent policy of the agent to obtain the most likely policy that generates the given batch of transitions.
- Develop KLPER, to enable the agent learn through more on-policy updates. KLPER uses KL Divergence between Batch Generating Policy and the most recent policy of the agent to prioritize batches of transitions.
- Demonstrate KLPER on 6 different continuous control tasks. Results yield that our algorithm brings significant improvements on particular tasks in terms of final performance and sample efficiency.

II. Background

In this section, we briefly introduce the reinforcement learning framework. We summarize two off-policy continuous control algorithms that we couple with the KLPER. Then, we cover various experience replay prioritization methods.

A. Reinforcement Learning

In reinforcement learning, the agent tries to find a policy that maximizes the cumulative reward while interacting with the environment. Reinforcement Learning tasks can be formulated as Markov Decision Processes (MDP). At each discrete time step t, the agent observes a state $s_t \in \mathcal{S}$ from the environment, then the agent selects an action $a_t \in \mathcal{A}$, according to its policy $a_t \sim \pi(a|s_t)$. After an action has been selected, the agent receives a reward and the next state, $s'_t \in \mathcal{S}$ with respect to the learning environment dynamics, $P(s', r|s, a)$. The combination of the four elements, (s, a, r, s'), forms a transition, which is appended to a replay buffer that stores the agent's experiences. The main goal of an agent is maximizing its return, discounted cumulative reward, which is defined as $G_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$, where γ is the discount factor.

B. Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is an Off-Policy Deep Reinforcement Learning algorithm that produces deterministic actions on continuous action space. DDPG is an extension of the Deterministic Policy Gradient algorithm [19] which includes function approximation [2]. DDPG consists of two nested network architectures that output the policy of the agent, i.e., Actor, and the estimate values of the state-action pairs, i.e., Critic. The Actor network generates a deterministic action with respect to the state, $a = \psi(s; \phi)$. The Critic network outputs a value representing the estimated return after taking action at state s, $Q(s, a; \theta)$. In the DDPG algorithm, the Actor and the Critic networks are updated sequentially, starting with the Critic. Bootstrapping by directly using the Actor and the Critic networks makes agents prone to divergence [2]. To achieve stability during the training and avoid a divergent policy, target networks, which drastically improved the performance of the DQN algorithm, are used on DDPG [7]. In the DDPG algorithm, there are two networks called the Actor Target and the Critic Target networks which are initialized identically as the Actor and the Critic Networks. Bootstrapping is performed on the Actor Target and the Critic Target networks and their parameters are updated softly updated with respect to the Actor and the Critic networks [2]. The Critic network is trained by using the L2 Loss of the one-step Temporal Difference Error as follows:

$$L = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}[(y - Q(s, a; \theta)^2], \tag{1}$$

where ϕ and θ are the parameters of the Actor network and the Critic network, respectively, y is defined as:

$$y = r + Q'(s', \psi(s'; \phi'); \theta'), \tag{2}$$

where ϕ' and θ' are the parameters of the Actor target network and the Critic network, respectively. The Actor network is optimized by taking the derivative of the objective function with respect to the Actor Network parameters:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim p_\pi}[\nabla_a Q(s, a; \theta)|_{a=\psi(s;\phi)} \nabla_\phi \psi(s; \phi)], \tag{3}$$

Finally, the Target network parameters are softly updated with respect to the Actor and the Critic parameters.

$$\phi' = \tau\phi + (1 - \tau)\phi', \quad \theta' = \tau\theta + (1 - \tau)\theta', \tag{4}$$

where τ is the parameter that controls the rate of the soft update.

C. Twin Delayed DDPG

Twin Delayed Deep Deterministic Policy Gradient (TD3) is an extended version of the DDPG algorithm that significantly improves the performance of its predecessor DDPG [20]. TD3 remedies the problem of overestimation in state-action pairs of the DDPG algorithm. TD3 includes two Critic networks. The minimum Q value that is produced by the Critic Target Networks is set as the target value. In this procedure, called as Clipped Double Q-learning, the target value is calculated as follows:

$$y = r + \gamma \min_{i=1,2} Q'_i(s', \psi'(s'; \phi'); \theta'_i), \tag{5}$$

In addition to using two Critic networks and taking the minimum value generated by these two networks, TD3 makes several modifications on DDPG. TD3 includes Delayed policy update and target policy smoothing, which enables TD3 to outperform DDPG on continuous control tasks [20].

D. Experience Replay Methods

The most primitive version of the Experience Replay mechanism is sampling transitions from the replay buffer uniformly. The idea that some transitions might be more useful or more adversarial than the other ones led to the emergence of new sampling methods [12].

One of the most well-known techniques is Prioritized Experience Replay (PER) [13]. PER increases the sampling probabilities of the transitions that yield more unexpected outcomes for the agent. However, the unexpectedness measure of a transition is not directly reachable. Therefore, PER uses Temporal Difference Error as a proxy to quantify the importance of a transition by assuming a positive correlation between Temporal Difference Error and the unexpectedness of the transition. Temporal Difference Error can be defined as follows:

$$|\delta| =\mid r + \gamma Q\left(s', a'; \theta'\right) - Q(s, a; \theta)\mid, \tag{6}$$

where θ and θ' represent the value and target value network parameters, respectively.

Sharpness of the prioritization can be softened by adjusting the α parameter that combines PER with uniform sampling. Then, the sampling probability of a transition becomes:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{7}$$

Attentive Experience Replay (AER) is an experience replay algorithm that prioritizes transitions by using the state information [15]. AER assumes that states visited by the past policies are not useful for the training process of the agent. These states should be less frequently visited when the agent has a more stable policy. Then, AER feeds the agent with the frequently visited states. The algorithm aims to optimize the agent by using transitions collected by the recent policies of the agent.

We emphasize that the experience replay prioritization can be defined as a learning task. Then, an expert that has a dynamic policy could learn how to optimize the learning progress of the agent and provide an optimal experience replay strategy for the agent. Experience Replay Optimization and Neural Experience Replay Sampler are two approaches that handle the experience replay prioritization by adding a parameterized replay policy [16], [14]. The replay policy component of these methods assigns scores to each transition given the extracted information from the transition such as state, action, the reward of the transition, temporal-difference error, and timestep when the transition is generated. In the following section, we give more details on how we construct our algorithm.

III. KL Experience Replay

In this section, we mention the problems that KLPER aims to tackle. We define the Batch Generating Policy, one of the critical components of the algorithm. Then, we give more details about the batch selection process among candidate batches.

A. Motivation

Increasing sampling probability of the specific transitions over the other ones may lead to undesirable updates on the Actor and the Critic networks. The likelihood of having these unwanted updates is proportional to the heaviness of the prioritization since heavy prioritization causes more Off-Policy Updates [17].

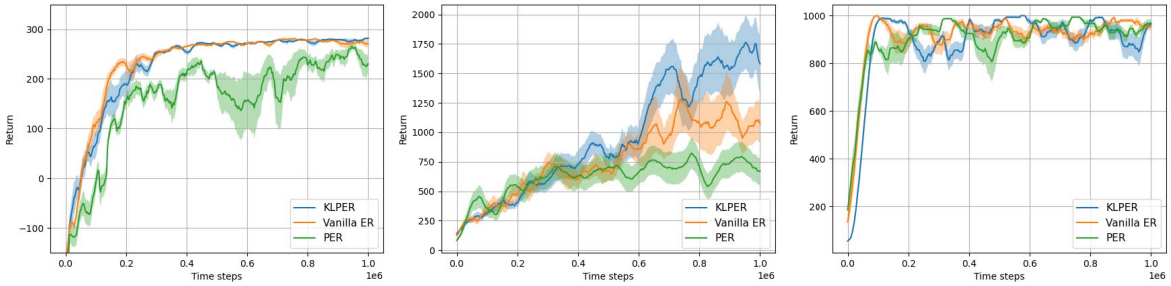
The sampling probability of a transition reflects the importance of the transition. Each transition's contribution to the learning process depends on the policy of the agent and the value network that the agent uses. To quantify the importance of a transition, one measure should be defined. For instance, PER uses Temporal Difference error for that purpose [13]. Furthermore, the importance of the transitions changes during the training since the policy of the agent or the value network that the agent uses is updated after each iteration. Then, to properly arrange the importance of the samples in the buffer, one should span the whole replay buffer and recalculate the sampling probabilities, which is infeasible in terms of computation after some point since the number of samples in the buffer increases rapidly. PER uses a practical solution for the given problem, recalculates a transitions sampling probability when it is sampled [13]. In that case, the expected sampling period of a prioritized transition is calculated as follows:

$$T_s = \frac{1}{P_i b} \tag{8}$$

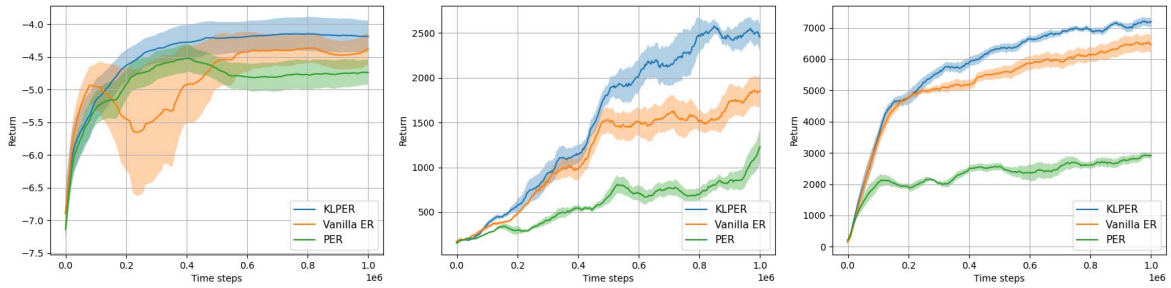
where P_i is the ith transition's sampling probability and b is the mini-batch size. This procedure assumes that the importance of a transition remains the same until it is sampled again. However, a desirable transition may become indifferent or even adversarial at the latter stages of the training for the agent and vice versa [12]. Due to the aforementioned issues, an algorithm that prioritizes transitions may lead to an undesirable training process. Then, the Vanilla ER algorithm that samples transitions may outperform a method that prioritizes the samples in the replay buffer [17].

In reinforcement learning, the deadly triad is defined as the combination of function approximation, bootstrapping, and off-policy learning. An algorithm that includes these three properties may suffer from unbounded value estimates, which deteriorates the learning process of the agent [17]. Function Approximation is the most indispensable component of reinforcement learning among the properties of the deadly triad because when state and action spaces of the reinforcement learning task are huge, then visiting all state-action pairs becomes infeasible, especially in the continuous domain. One may use Monte Carlo learning instead of Temporal Difference learning, then discard the Bootstrapping. However, Monte Carlo learning requires long trajectories that end with a terminal state. Tasks that have no termination conditions cannot be properly solved by Monte Carlo learning. If the On-Policy learning is chosen over Off-Policy learning, the agent cannot use the experiences generated from its past policies. Then, the heavily correlated transitions negatively affect the neural network training [17]. The final component of the deadly triad, Off-Policy learning, can be softened by changing the sampling probabilities of the transitions and the performance of the modified algorithm performs better on the learning tasks [17].

In this paper, we introduce a novel experience replay prioritization algorithm, KLPER, which reduces the off-policyness

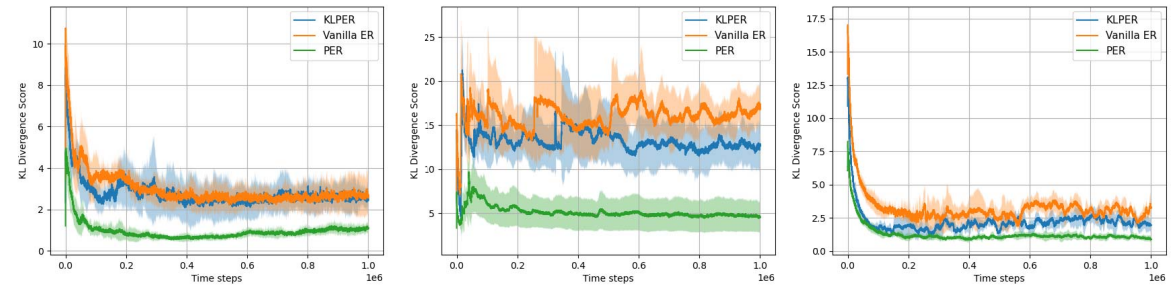


(a) LunarLanderContinuous-v2 (b) Hopper-v2 (c) InvertedPendulum-v2

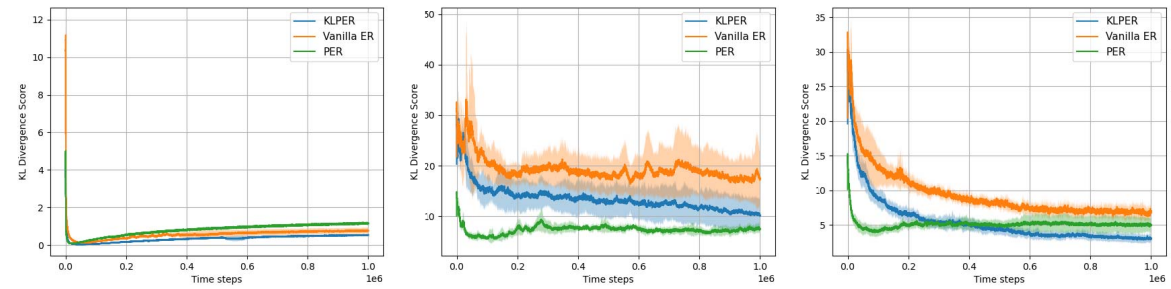


(d) Reacher-v2 (e) Walker2d-v2 (f) HalfCheetah-v2

Fig. 1. Learning Curves of the experience replay methods, KLPER, PER and Vanilla ER on 6 different OpenAI Gym continuous control tasks. The algorithms are coupled with the DDPG. Cumulative reward curves are smoothed for visual clarity. The shaded regions represents half a standard deviation over five trials.



(a) LunarLanderContinuous-v2 (b) Hopper-v2 (c) InvertedPendulum-v2



(d) Reacher-v2 (e) Walker2d-v2 (f) HalfCheetah-v2

Fig. 2. KL Divergence scores that are yielded by Batch Generating Policy and multivariate Gaussian distribution with mean 0 and covariance $0.1\mathbb{I}$ for each algorithm that coupled with the DDPG. Curves are smoothed for visual clarity.

of the updates at each iteration for Deep Deterministic Policy Gradient algorithms. Our approach selects one batch among candidate batches rather than assigning sampling probabilities to the transitions in the replay buffer.

B. Batch Generating Policy

Behavior policy is defined as a mixture of an exploration noise and the target policy of the agent, which is parameterized by the Actor network, for DDPG and TD3 algorithms. The replay buffer of an agent includes transitions gathered by the past policies of the agent. We attempt to find the most likely policy that generates the sampled batch of transitions with respect to the most recent policy of the agent. We call that policy as Batch Generating Policy and denote it as ω. We remark that the policy used for generating each transition becomes intractable after the transition is stored due to the exploration noise term of the behavior policy. Thus, we assume that the Batch Generating Policy is stochastic. We elaborate on how we build the Batch Generating Policy for the remaining part of this subsection.

Feedforwarding the states in the batch of transitions, $\boldsymbol{S}^{b\times m}$, to the Actor network yields the actions, $\hat{\boldsymbol{A}}^{b\times l}$ that the agent act in these states respect to its most recent policy:

$$\hat{\boldsymbol{A}}^{b\times l} = \psi(\boldsymbol{S}^{b\times m}; \phi), \tag{9}$$

where b is the mini-batch size, ψ is the Actor network, l and m are the number of dimensions that action and state space have, respectively. The difference between actions in the batch and actions that would be taken by the agent's most recent policy, $\dot{\boldsymbol{A}}^{b\times l}$, represents the deviation between the current policy of the agent and previous policies of the agent that were used to generate experiences.

$$\dot{\boldsymbol{A}}^{b\times l} := \hat{\boldsymbol{A}}^{b\times l} - \boldsymbol{A}^{b\times l}, \tag{10}$$

where $\boldsymbol{A}^{b\times l}$ is the actions stored in the transitions of the sampled batch. The exploration noise choice affects the behavior policy of the agent. The works show that using Gaussian noise as the exploration noise rather than Ornstein-Uhlenbeck noise [21] does not decrease the performance of the DDPG algorithm [20], [22]. Hence, We choose Gaussian noise,as the exploration noise for both algorithms.

We remark that Batch Generating Policy is stochastic, and we defined it as a probability distribution. The mean of the distribution can be formulated as the difference between the actions of the sampled transitions and the actions that the agent would produce with respect to states of the corresponding transition:

$$\mu_{\omega}^{1\times l} = \frac{1}{b}\sum_{i\in b} \dot{\boldsymbol{A}}_{ij}^{b\times l}, \tag{11}$$

where i is the ith element of the batch and j is the jth dimension of the action space. We define the covariance matrix of the distribution as follows:

$$\Sigma_{\omega}^{l\times l} = \frac{1}{b-1}\sum_{k\in b}(\dot{\boldsymbol{a}}_{k}^{1\times l} - \mu_{\omega}^{1\times l})^{\top}(\dot{\boldsymbol{a}}_{k}^{1\times l} - \mu_{\omega}^{1\times l}), \tag{12}$$

where $\dot{\boldsymbol{a}}_{k}^{1\times l}$ is the kth row of the $\dot{\boldsymbol{A}}^{b\times l}$ matrix. We define the shape of the distribution as the multivariate Gaussian by Maximum Entropy Principle. Finally, we obtain Batch Generating Policy:

$$\omega \sim \mathcal{N}(\mu_{\omega}^{1\times l}, \Sigma_{\omega}^{l\times l}). \tag{13}$$

C. Choosing Batches with KL Divergence

In this section, we give more details on how KLPER scores and chooses one batch among candidate batches.

Firstly, at each training timestep t, KLPER samples N candidate batches before updating the parameters of the Actor and the Critic networks. Then, the algorithm derives Batch Generating Policy for each sampled batch. It uses KL Diverge to measure the similarity between the policy of the agent and the Batch Generating Policy. The Actor networks for the DDPG and TD3 algorithms yield deterministic policies. However, we remark that the transitions are generated by following the behavior policy of the agent. Therefore, we define the target distribution for the KL Divergence as a multivariate Gaussian distribution with mean 0 and variance $\sigma\mathbb{I}$. We refer KL score for each batch with κ and KLPER calculates the KL Score as follows:

$$\kappa = \mathrm{D}_{\mathrm{KL}}(\mathcal{N}(\mu_{\omega}, \Sigma_{\omega}) \| \mathcal{N}(0, \sigma\mathbb{I})), \tag{14}$$

where $\mathbb{I}$ is the identity matrix. KLPER selects the batch that yields the minimum KL score among candidate batches to provide the learning algorithms more on-policy updates at each iteration. We provide KLPER in Algorithm 1.

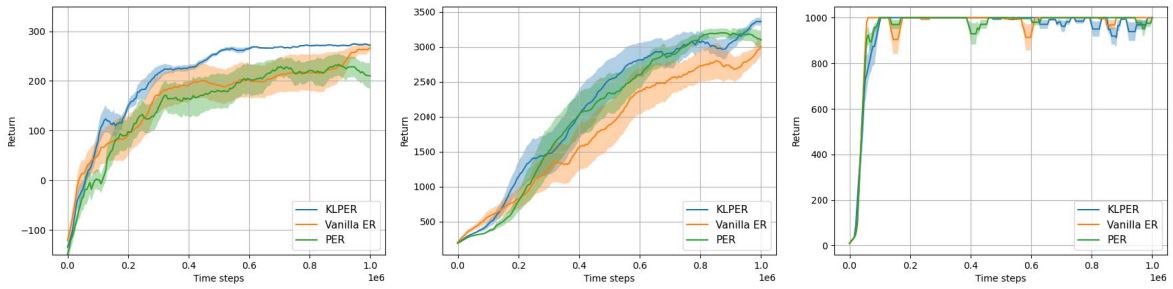
IV. Experiments

In this section, we elaborate on the implementation details on learning algorithms and hyper-parameters. Then, we provide results and comparisons of the KLPER with Vanilla ER and PER. We inspect the results into two categories: results for DDPG and results for TD3.

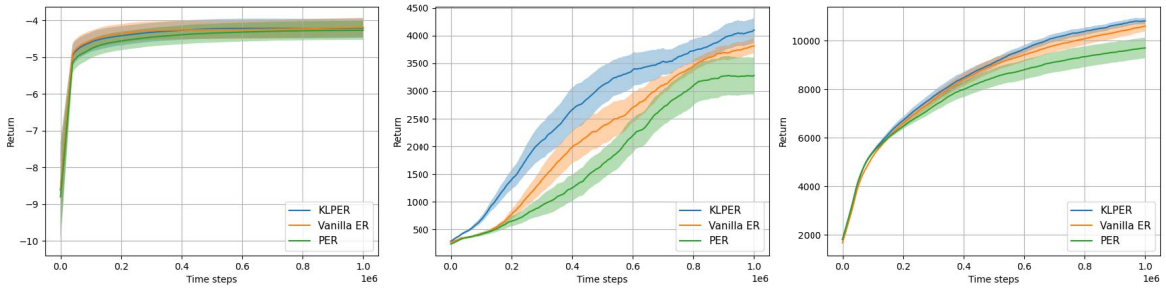
A. Implementation Details

We note that our algorithm compared with Vanilla ER and PER. To make a fair comparison, we have run the algorithms by using 5 different random seeds. For each experience replay method, we use the same architecture and hyperparameters for DDPG and TD3 algorithms. Also, we take the network architectures from the original implementations of the DDPG and TD3 algorithms [2], [20].

DDPG has two hidden layer neural networks for both the Actor and the Critic networks, which include 400 and 300 neurons, respectively. For TD3, we use two hidden layer networks that have both 256 neurons, and the network architecture for Actor and Critic are the same. We use Adam as the optimizer of the networks for both DDPG and TD3 [23]. The learning rates of the Actor and the Critic networks are 1×10^{-4} and 3×10^{-4}, respectively, for the DDPG. The learning rate of both Actor and Critic networks for the TD3 algorithm was 1×10^{-3}. We set the mini-batch size to 64 and 256 for DDPG and TD3. We observe that the DDPG algorithm sticks at the local optima when the number of exploration steps is

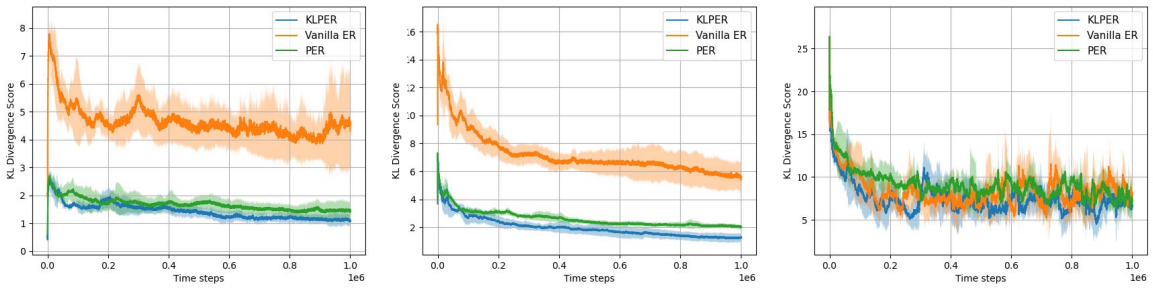


(a) LunarLanderContinuous-v2 (b) Hopper-v2 (c) InvertedPendulum-v2

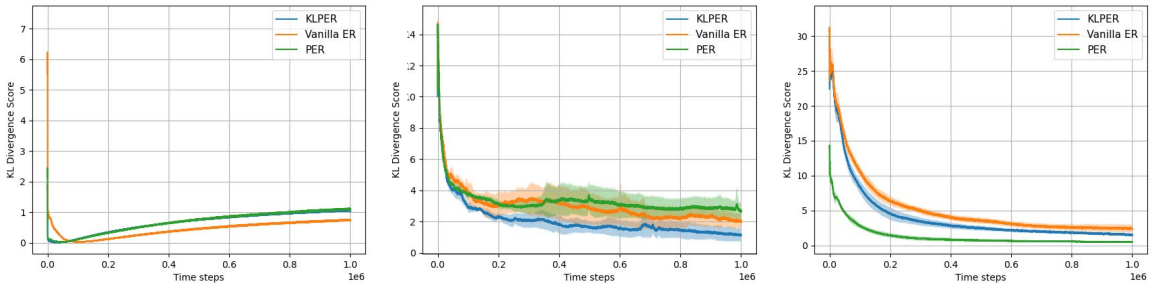


(d) Reacher-v2 (e) Walker2d-v2 (f) HalfCheetah-v2

Fig. 3. Learning Curves of the experience replay methods, KLPER, PER and Vanilla ER on 6 different OpenAI Gym continuous control tasks. The algorithms are coupled with the TD3. Cumulative reward curves are smoothed for visual clarity. The shaded regions represents half a standard deviation over five trials.



(a) LunarLanderContinuous-v2 (b) Hopper-v2 (c) InvertedPendulum-v2



(d) Reacher-v2 (e) Walker2d-v2 (f) HalfCheetah-v2

Fig. 4. KL Divergence scores that are yielded by Batch Generating Policy and multivariate Gaussian distribution with mean 0 and covariance $0.1\mathbb{I}$ for each algorithm that coupled with the TD3. Curves are smoothed for visual clarity.

Algorithm 1 KLPER

Initialize batch size b, replay buffer $\mathcal{D}$
Initialize M for delayed policy updates
Initialize number of candidate batches N
Initialize σ for the target distribution
for $t = 1$ **to** T **do**
  Observe s_t Choose action $a_t \sim \pi_\phi(a|s_t) + \epsilon$
  Observe reward r_t and new state s'_t
  Store transition (s_t, a_t, r_t, s'_t) in $\mathcal{D}$
  Sample N batches from $\mathcal{D}$
  for $n = 1$ **to** N **do**
    Calculate κ_n for the batch (Eq. 14)
  end for
  Select the batch that yields minimum κ
  Update the weights of the Critic network
  if k mod M **then**
    Update the weights of the Actor network
    Update the weights of the Target networks
  end if
end for

insufficient. Therefore, we fill the replay buffer with 10000 transitions gathered by the agent while acting randomly for the DDPG algorithm for each experience replay method. The number of exploration steps set to 25000 for TD3 algorithm.

We choose N, an adjustable parameter for the algorithm, as 4 for DDPG and 8 as TD3. We choose σ as 0.1 for DDPG to couple exploration noise that is used for the DDPG algorithm and target distribution of the KL Divergence. For TD3, we assign σ to 0.2 since TD3 uses SARSA like updates on bootstrapping step of the algorithm by adding a noise term to the action produced by the Actor Target network. This noise component is sampled from a Gaussian Distribution with a 0.2 variance.

We use Proportional PER for the comparison. The alpha and the beta parameters set to 0.6 and 0.4 for Proportional PER [13].

For the comparison of the methods, we run the algorithms on 6 different MuJoCo tasks as the learning environments , which vary in terms of state and action spaces. We evaluate the performances of the algorithms for every 5000 timesteps. During the evaluation episodes, agents perform by using their most recent policy 5 times. We assign average cumulative reward over 5 evaluation episodes as the evaluation score of an agent.

B. Results for DDPG

In this section, we combine KLPER, PER and Vanilla ER with the DDPG algorithm. In Fig. 1, we propose the learning curves.

Fig. 1 shows that our method outperforms Vanilla ER and PER methods for four of the six learning environments in terms of the model's final performance and sample efficiency. Cumulative Rewards of the DDPG Agents that use KLPER as the experience replay method almost monotonically increase during the training process for most of the learning environments. For the LunarLanderContinuous-v2, Vanilla ER and KLPER converge to a nearly optimal policy at the early stages of the training. Results on InvertedPendulum-v2 suggest that there is no significant difference among the methods.

We compare each Batch Generating Policy's deviation from the most recent policy of the agent for each method and provide the results in Fig. 2. As an interesting finding, even though KLPER samples 4 batches and selects the batch that has the least KL score, mostly batches that selected by PER have lower KL scores during the training process. As an explanation for that, we zoom in on the dynamics of the experience replay. The agent fills its replay buffer after the exploration steps by using its behavior policy which is a stochastic variant of its target policy. If the policy of the agent changes considerably after each policy update, then more off policy transitions would be stored to the replay buffer. We conjecture that KLPER leads more prominent policy changes during the training. Consequently, the sampled batches' KL scores are relatively high when compared to PER.

C. Results for TD3

In this section, we combine KLPER, PER, and Vanilla ER with the TD3 algorithm. In Fig. 3, we propose learning curves.

KLPER outperforms the agents that use Vanilla ER and PER in four out of six learning environments in terms of the final performance and sample efficiency. For Reacher-v2, agent performances are almost indistinguishable. For LunarLanderContinuous-v2, the variances of the cumulative rewards collected by the agents that use Vanilla ER and PER are relatively high. On the other hand, all the agents that use KLPER converged to a nearly optimal and robust policy. KLPER provides the same performance as the other methods for the InvertedPendulum-v2 task when coupled with both the learning algorithms, DDPG and TD3. We denote the state and action space dimensions of the learning environments in Table 1. The InvertedPendulum-v2 task has four-dimensional state space and one-dimensional action space. Therefore, we explain that fact as KLPER algorithm may fail to work in low dimensional tasks. However, it works well on high-dimensional action and state spaces.

We investigate KL scores of the sampled batches for each method by following the same procedure as in DDPG and provide the results in Fig. 4. It is surprising that KL Divergence values of batches when PER and KLPER are used are so close to each other in LunarLanderContinuous-v2 and Hopper-v2. However, KLPER agents perform better than PER on both of the environments in terms of the final performance. Prioritizing experience replay leads to relatively smaller policy changes on these learning tasks. For InvertedPendulum-v2, KL score curves are noisy for each method. Then we observe that all the experience replay methods, when combined with TD3, do not ensure a robust policy for InvertedPendulum-v2 task.

TABLE I
STATE AND ACTION SPACE DIMENSIONS OF THE ENVIRONMENTS IN MUJOCO CONTROL SUITE

Environment	LunarLander-v2[1]	Hopper-v2	InvPend-v2[2]	Reacher-v2	Walker2d-v2	HalfCheetah-v2
State Dimension	8	11	4	11	17	17
Action Dimension	2	3	1	2	6	6

[1] Abbreviation for LunarLanderContinuous-v2
[2] Abbreviation for InvertedPendulum-v2

V. CONCLUSION

In this paper, we emphasize the drawbacks of prioritizing transitions, and improve the performance of the agent by reducing the off-policyness of the reinforcement learning algorithms. We introduce the KLPER algorithm that makes prioritization on the batch of transitions rather than prioritizing each transition in the replay buffer. We define Batch Generating Policy to quantize the off-policyness level of a batch. Our algorithm enables agents to have more on-policy updates using KL Divergence between Batch Generating Policy and a multivariate Gaussian distribution with a mean of 0. We combine KLPER with Deep Deterministic Policy Gradient algorithms and test it on continuous control tasks. Results show that KLPER brings promising improvements and outperforms Vanilla ER and PER in terms of sample efficiency and final performance in particular continuous control tasks.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[2] 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," 2019.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, jan 2016.

[4] O. Vinyals, I. Babuschkin, W. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. Agapiou, M. Jaderberg, and D. Silver, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, 11 2019.

[5] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[6] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Mach. Learn.*, vol. 8, p. 293–321, May 1992.

[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*, vol. 518, no. 7540, pp. 529–533, 2015.

[8] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," *CoRR*, vol. abs/1707.01495, 2017.

[9] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015.

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

[11] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška, "Experience selection in deep reinforcement learning for control," *Journal of Machine Learning Research*, vol. 19, no. 9, pp. 1–56, 2018.

[12] S. Zhang and R. S. Sutton, "A deeper look at experience replay," 2018.

[13] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[14] Y. Oh, K. Lee, J. Shin, E. Yang, and S. J. Hwang, "Learning to sample with local and global contexts in experience replay buffer," 2021.

[15] P. Sun, W. Zhou, and H. Li, "Attentive experience replay," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 5900–5907, Apr. 2020.

[16] D. Zha, K.-H. Lai, K. Zhou, and X. Hu, "Experience replay optimization," 2019.

[17] H. van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil, "Deep reinforcement learning and the deadly triad," 2018.

[18] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[19] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," *31st International Conference on Machine Learning, ICML 2014*, vol. 1, 06 2014.

[20] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018.

[21] G. E. Uhlenbeck and L. S. Ornstein, "On the Theory of the Brownian Motion," *Physical Review*, vol. 36, pp. 823–841, Sept. 1930.

[22] S. Fujimoto, D. Meger, and D. Precup, "Off-policy deep reinforcement learning without exploration," 2019.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.