applied sciences

MDPI

Article

Self-Adaptive Priority Correction for Prioritized Experience Replay

Hongjie Zhang, Cheng Qu, Jindou Zhang and Jing Li *

School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China; zhanghongjie@mail.ustc.edu.cn (H.Z.); qucheng@mail.ustc.edu.cn (C.Q.); jindou@mail.ustc.edu.cn (J.Z.)

* Correspondence: lj@ustc.edu.cn

Received: 3 September 2020; Accepted: 28 September 2020; Published: 2 October 2020

Abstract: Deep Reinforcement Learning (DRL) is a promising approach for general artificial intelligence. However, most DRL methods suffer from the problem of data inefficiency. To alleviate this problem, DeepMind proposed Prioritized Experience Replay (PER). Though PER improves data utilization, the priorities of most samples in its Experience Memory (EM) are out of date, as only the priorities of a small part of the data are updated while the Q network parameters are updated. Consequently, the difference between storage and real priority distributions gradually increases, which will introduce bias into the gradients of Deep Q-Learning (DQL) and make the DQL update toward a non-ideal direction. In this work, we propose a novel self-adaptive priority correction algorithm named Importance-PER (Imp-PER) to fix the update deviation. Specifically, we predict the sum of real Temporal-Difference error (TD-error) of all data in EM. Data are corrected by an importance weight, which is estimated by the predicted sum and the real TD-error calculated by the latest agent. To control the unbounded importance weight, we use truncated importance sampling with a self-adaptive truncation threshold. The conducted experiments on various games of Atari 2600 with Double Deep Q-Network and MuJoCo with Deep Deterministic Policy Gradient demonstrate that Imp-PER improves the data utilization and final policy quality on discrete states and continuous states tasks without increasing the computational cost.

Keywords: deep reinforcement learning; experience replay; importance sampling; DDQN; DDPG

1. Introduction

As a pivotal field of machine learning, deep reinforcement learning (DRL) has surpassed the human level in many fields, such as video games [1], robot control [2,3], traffic signal control [4], chess tasks [5], and speech recognition [6]. The typical process for DRL is as follows. The agent receives state of the environment and associated rewards, predicts action. The environment will perform the action and feedback of the new state and the reward. Repeat the loop until the target is reached or the environment ends. The goal of DRL is to learn how to control the environment system to maximize cumulative rewards. There is a big difference from supervised learning. In supervised learning, samples are prepared early and labeled by human experts. In DRL, samples are generated dynamically, and there are no clear labels, only sparse reward signals. The quality of the sample cannot be guaranteed, which makes training more difficult. Even for a simple video game, DRL still requires millions of samples to complete training. Traditionally, through the Experience Replay (ER) technology [7,8], DRL can remember and reuse the previously generated samples to improve data utilization. In the ER setting, the samples generated by the agent will be stored in the EM and repeatedly reused, which will improve the sample utilization rate while reducing the correlation between samples, making training more stable. Recently, DeepMind proposed Prioritized Experience

Replay (PER) [9], which further improves data utilization compared to ER. In PER, the probabilities of replaying data are proportional to their priorities. The Temporal-Difference error (TD-error) provides a way to measure these priorities [10]. Therefore, data in the replay EM are proportional to their absolute TD-errors. It makes the agent focus on valuable data and speeds up the training process. Subsequently, many works have been proposed to improve DRL based on PER, such as Distributed PER [11], DDPG + PER [12], twice sampling PER [13], ERO [14], ReF-ER [15], and Rainbow [16].

In Deep Q-Learning (DQL) or Deep Deterministic Policy Gradient (DDPG), the TD-error is the difference between the target Q value and the estimated Q value, where Q value is the state-action value function. The goal of DQL or DDPG is to put the estimated Q value closer to the target Q value. When updating the value function, the TD-errors of data in EM [17] will change randomly. However, the TD-errors stored in EM are not updated in real-time. In this work, we refer to the TD error stored in the EM as the stored TD error or the stored priority. The real-time TD-error is called real TD-error or real priority. In the original PER, only the TD-errors of the replayed data are updated. As a result, most of the data will be sampled with the wrong priority. It makes DQL or DDPG update in a non-ideal direction [9], where the ideal update direction of DQL or DDPG is represented by the gradients calculated using samples under real priority distribution. Meanwhile, due to a large amount of data in EM, the cost of updating the data priorities of the entire EM is intolerable. Therefore, effectively correcting data priority is a challenge. The intuitive solution to this problem is approximating the real priority of the entire EM. Bai et al. proposed an active sampling method named ATDC-PER [18], which predicts the real priorities of data in EM according to their stored TD-errors and replay periods. However, it requires to update the stored TD-error of the entire EM, which will be very time-consuming.

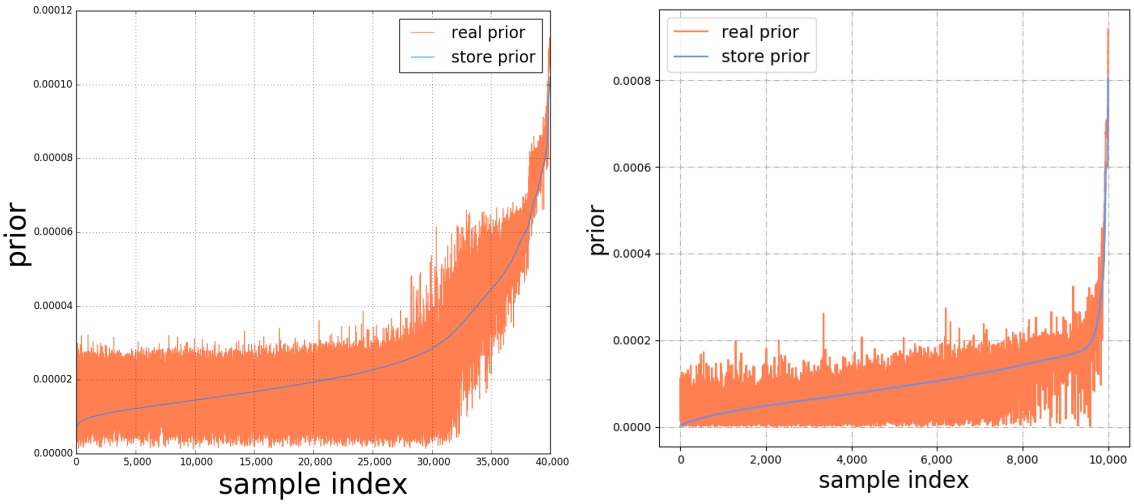
As mentioned above, the delayed TD-error makes the agent update in the wrong direction, thereby reducing data utilization. Existing methods try to calculate the real TD-errors of the data and update the entire EM in real time. These methods are computationally expensive and difficult to apply in practice. How to quickly correct the update direction of the agent to keep it consistent with the theoretical direction of real TD-error is an important issue.

Unlike previous methods, this work solves this problem from another perspective. The proposed method only corrects the replayed samples to fix the update direction of DQL or DDPG. As the learning rate of DQL is usually low, Q value will not change much in the short term. Although there are differences between the real TD-error and stored TD-error, the overall distributions of both are similar (Figure 1). This motivated us to find a solution to correct the priority deviations according to the similarity. Specifically, We use the importance of the replayed samples to achieve a correct estimate of the gradient. To achieve this goal, there are two main challenges must be overcome: One is how to accurately calculate the real priorities with a low computation cost, and the other is how to reduce the time overhead of the correct gradient's estimation and improve data utilization.

In this work, we propose a self-adaptive priority correction algorithm Importance-PER (Imp-PER) based on Importance Sampling (IS) [19,20]. First, Imp-PER uses a regression model to predict the sum of TD-error of all data in EM according to the stored priorities. Then, a mini-batch of data is sampled from PER according to the stored priorities. After that, the real sampling probabilities of data are estimated by the predicted sum and the real TD-error values calculated by the latest Deep Q-Network (DQN) or DDPG. Finally, in the DQN or DDPG loss function, data is multiplied by importance weight, which is estimated by the real sampling probabilities and the stored sampling probabilities. The main contributions of this work are summarized as follows.

- We design a linear model to predict the sum of real TD-error and estimate the real probability of the current batch of data. Compared to predicting the real probability of each sample in EM, the cost is much less.
- By using IS technology to estimate the DQN or DDPG loss function for the real priority distribution, we improve the data utilization and eliminate the cost of updating the entire EM priority.

- We conduct experiments on various video games of Atari 2600 with DQN and MuJoCo with DDPG from OpenAI Gym [21] to evaluate our algorithm. Empirical results show that Imp-PER can significantly improve data utilization on discrete state and continuous state tasks without introducing additional computational costs. The experiments also illustrate that Imp-PER is compatible with value-based and policy-based DRL.



(**a**) store vs real priorities of game Pong (**b**) store vs real priorities of task Walk2d

Figure 1. The similarity of distributions of real and stored priorities in Atari game Pong and MuJoCo walk2d. The x-axis is the index of the sample, which is sorted according to the stored TD-error.

2. Related Work

In this section, we discuss the related literatures on priority experience replay and importance sampling which will be used in our priority correction.

2.1. Priority Experience Replay

Priority experience replay was proposed in 2016 by DeepMind [9]. Due to its efficient data utilization, considerable works have been proposed to extend and optimize it. Initially, the PER is used to improve the DQN and its family (Double DQN [22], Duling DQN [23], Distributed DQN [11], and Rainbow [16]) algorithms. TD-error is an effective metric of priority and it is easy to obtain in DQNs. Not only does the value-based algorithm use PER, but researchers also try PER on policy-based algorithms, such as DDPG + PER [12]. The metrics used in priority are the same as DQN because DDPG also needs to optimize TD loss. By using PER, DRL's data utilization and policy quality have been significantly improved.

In PER, as TD-error is not the only choice of priority metrics, the selection of priority metrics will influence the training. To address this drawback, Zhao et al. developed a framework for a twice active sampling method in the deep Q-learning [13], which considered the cumulative reward and TD-error together. First of all, they sampled a batch of data proportional to their cumulative rewards. Then, a mini-batch of data was sampled according to their TD-errors. It got better performance on many Atari games. A similar idea was also used in HVPER [24]. HVPER considered the TD-error, value of Q, and data usage together. Different training stages would focus on different importance estimators, and dynamically adjust the weight of each estimator to realize an adaptive importance estimation method. To stabilize the PER, Hu et al. added the latest transitions into the training batch [25]. In addition, they also considered the reward into priority, which balances the TD-errors and rewards. In addition, Wang et al. used reward-shaping to give an additional reward in priority,

which further improved data utilization [26]. Zhu et al. integrated Upper Confidence Bound (UCB) with PER [27]. Using such a method, when choosing the next batch of data, the UCB balanced the exploration and exploitation of data. Compared to the original PER, UCB reached a better final policy. Recent research called Experience Replay Optimization (ERO) which dynamically adjusted the sampling policy to adapt variety tasks [14]. The ERO used the transition's reward, the TD-error, and the current timestep to estimate the priority of data. A similar priority was used to lifelong learning on multi-tasks DRL [28]. Some information theory-based methods were used in DRL, such as entropy-based optimal sampling strategy [29]. Different from the above methods, ReF-ER tried to control the deletion of data from EM but not to improve the sampling strategy [15]. Recently, Sun et al. proposed Attentive Experience Replay (AER), which ranked the data according to the agent's state distribution [30]. AER outperformed PER on many MujoCo tasks. In order to further improve data utilization, Bu et al. not only used priority sampling, but also based on priority when deleting old samples [31]. Bu et al. proposed double prioritized state recycled (DPSR) to sample and replace data in EM, which keeping a high-quality replay memory. Most of the latest state of the art research focuses on how to design better priority indicators, which basically used delayed TD-errors. However, our work is different from the above, we mainly solve the delay of TD-errors to improve data utilization. The above researches can use our method to further improve their performances.

2.2. Importance Sampling Techniques

Importance sampling (IS) is effective in reducing the variance of the estimations in Monte Carlo simulations. It also faces the problem of unbounded variance. The high variance in stochastic gradient descent makes the DRL training unstable and damages the final policy of the agent. To address the shortcoming, researchers proposed a variety of methods. Yu et al. proposed a novel method to bound IS weights [32]. They defined a safe region according to its value of IS. Especially, the safe region is the value lower than the threshold r. The integral of the safe region is used to estimate the IS of the whole region. A similar idea is used in another method (the truncated importance sampling method), which considers the optimal truncation rate is $\sqrt{n}$, where n is the batch size of the data [33]. Using the truncated importance sampling, the variance and bias of IS can be balanced. In some special circumstances, there is a special solution to solve the variance issue. Philip et al. proposed a simple estimator named importance sampling with unequal support (US) [34]. It aimed to handle the variance when the supports of the sampling and target distributions differ.

When the batch size becomes larger, the variance of importance sampling decreases. However, due to sampling costs, we cannot use unlimited batch sizes. Lately, the effective sample size (ESS) has been deeply studied. Martino et al. proposed a novel ESS function to measure the difference between two probability functions [35]. Sourav et al. got the best of ESS through theoretical analysis [36]. They thought that $exp(KL(p||q))$ is a sufficient batch size, where KL represents the Kullback–Leibler divergence of distribution p and q. Our priority correct algorithm relies on these researches heavily. Different from the above researches, we analyze the distribution of importance weights during reinforcement learning training, and set the threshold quantitatively to balance the bias and variance.

3. Proposed Approach

3.1. Preliminary

The goal of Reinforcement Learning (RL) is to maximize the cumulative reward of sequential decision tasks. The task is always modeled as a Markov Decision Process (MDP) problem, which is represented by a 5-tuple (S, A, P, R, γ). In the 5-tuple, S represents the state space, A represents the action space, P is the state transition function, $P : S \times A \rightarrow S$, R is the reward function, $R : S \times A \times S \rightarrow \mathbb{R}$, and γ is the discount factor, $\gamma \in (0, 1)$. The cumulative reward is $G = \sum_{t=0}^{\infty} \gamma^t r_t$,

where r_t represents the reward at current state. DQN will update the neural network weight $\boldsymbol{\theta_t}$ through Equation (1).

$$\boldsymbol{\theta_{t+1}} = \boldsymbol{\theta_t} + \eta(y - Q(s_t, a_t; \boldsymbol{\theta_t}))\nabla_\theta Q(s_t, a_t; \boldsymbol{\theta_t}) \tag{1}$$

where η is the learning rate and y is the target Q value.

$$y = r_t + \gamma Q(s_{t+1}, argmax_{a'} Q(s_{t+1}, a'; \boldsymbol{\theta_t}); \boldsymbol{\theta_t}) \tag{2}$$

In this work, we use an improved DQN named Double DQN (DDQN) [22] and DDPG [12]. In DDQN, the definition of target value y shown in Equation (3), which solves the over-optimistic value estimation problem. The same idea is used in DDPG. However, the DDPG uses a policy network to output actions.

$$y = r_t + \gamma Q(s_{t+1}, argmax_{a'} Q(s_{t+1}, a'; \boldsymbol{\theta_t}); \boldsymbol{\theta^-}) \tag{3}$$

From the above definition, we know that the value-based RL algorithm initially minimizes the TD error, which is defined as $\delta = Q(s_t, a_t; \boldsymbol{\theta_t}) - y$. In other words, δ represents how surprising the sample is [37]. DeepMind used the TD-error δ to define the priorities of samples in PER. For each data in EM, the stored probability of sampling $p(x)$ is defined by Equation (4) [9].

$$p(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{4}$$

where $p_i = |\delta_i| + \epsilon$ is the stored priorities of data i and $|\delta_i|$ are the stored TD-errors. The parameter α represents how much prioritization is used. However, priority experience replay can introduce bias into the evaluation of loss functions. To anneal the bias, PER uses importance sampling weights Equation (5).

$$v_i = \left(\frac{1}{N} \cdot \frac{1}{p(i)}\right)^\beta \tag{5}$$

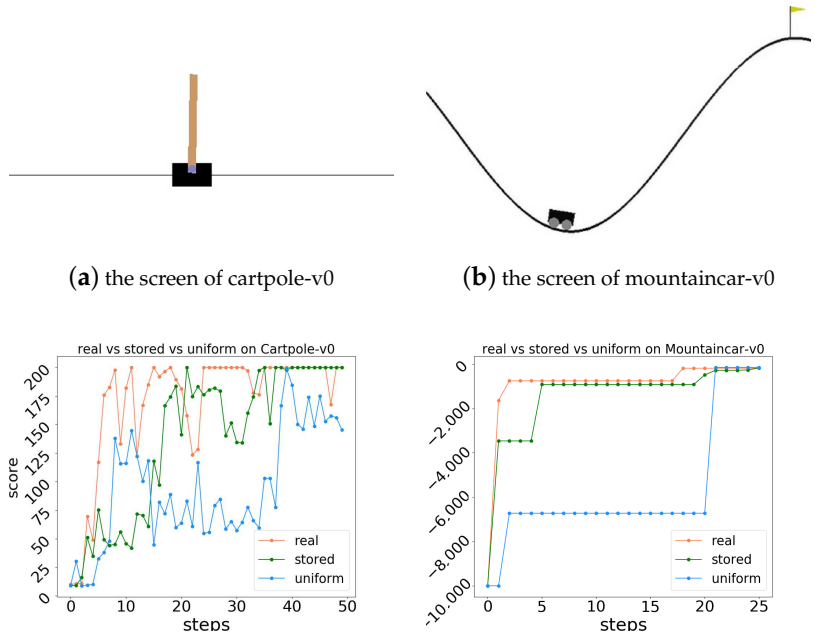
where β controls the correct strength. To reduce training fluctuations, v is usually normalized by $1/max_i v_i$. Therefore, the final loss function of DDQN or DDPG is defined by Equation (6).

$$loss(\boldsymbol{\theta}) = E_{x\sim p(x)}\left[\left(\frac{p(x)}{min_i p(i)}\right)^{-\beta}(Q(x,\boldsymbol{\theta}) - y)^2\right] \tag{6}$$

In this work, we define the real priority of data i as $q_i = |\tilde{\delta}_i| + \epsilon$, where the $\tilde{\delta}_i$ is estimated by real-time Q value function. The real sampling probability distribution is $q(i) = q_i^\alpha / \sum_k q_k^\alpha$. Figure 1 shows the probability distribution of real and stored priorities. The data are sorted from small to large according to their stored priorities. The difference in priorities is obvious.

We also train a DQN on two simple tasks—Cartpole and Mountain-car—based on different priorities. We use a 3-layers fully-connected neural network to represent the Q function. The result is illustrated in Figure 2, where the x-axis represents episodes and the y-axis represents the corresponding score. It can be noticed that DQN with real priority has higher efficiency.

However, it is costly to obtain real priority when training with big neural networks on complex tasks. Our problem is very clear: how to estimate the value of the DDQN or DDPG's loss function for the real distribution without increasing the computational cost.



(a) the screen of cartpole-v0

(b) the screen of mountaincar-v0

(c) training DQN on Cartpole-v0 with different priorities

(d) training DQN on Mountain-car-v0 with different priorities

Figure 2. (**a**) Shows the screen of task cartpole. (**b**) Shows the screen of task mountaincar. (**c**) Shows the performance of DQN based on real, store and uniform priorities on task cartpole. (**d**) Shows the performance of DQN based on real, store and uniform priorities on task mountaincar. The orange, green, and blue curves represent real, store, and uniform priority, respectively.

3.2. Overview

The core idea of the proposed algorithm is to use importance weights to correct the update direction of DQN or DDPG based on the priority of data. The key challenges are how to accurately predict the real priorities and correct priorities at the lowest possible computational cost. In this section, we only discuss the DQN algorithm. The DDPG extends DQN and applies it to the continuous state-action space. They have a similar training framework. Figure 3 shows the architecture of the Imp-PER algorithm on DDQN. First, the proposed algorithm needs to get the stored sampling probability $p(x)$ and predict the real sampling probability $q(x)$ of each replayed samples, which is handled by the TD sum model. Then, it calculates the importance weights w_i based on $p(x)$ and $q(x)$. Last, the loss function of DQL is corrected by the importance weight and trained as original PER, as shown in the DDQN's definition. The other modules are the same as the original PER. Therefore, the Imp-PER is general and can be used in any DRL algorithm based on PER.

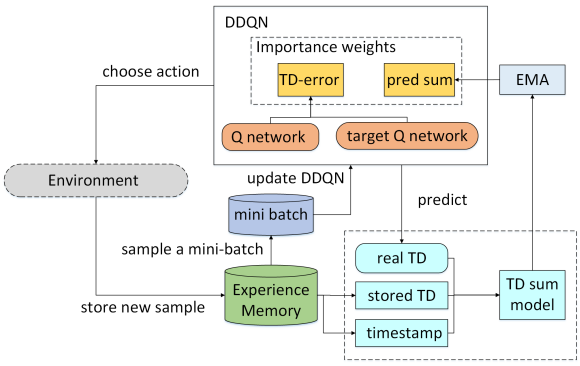
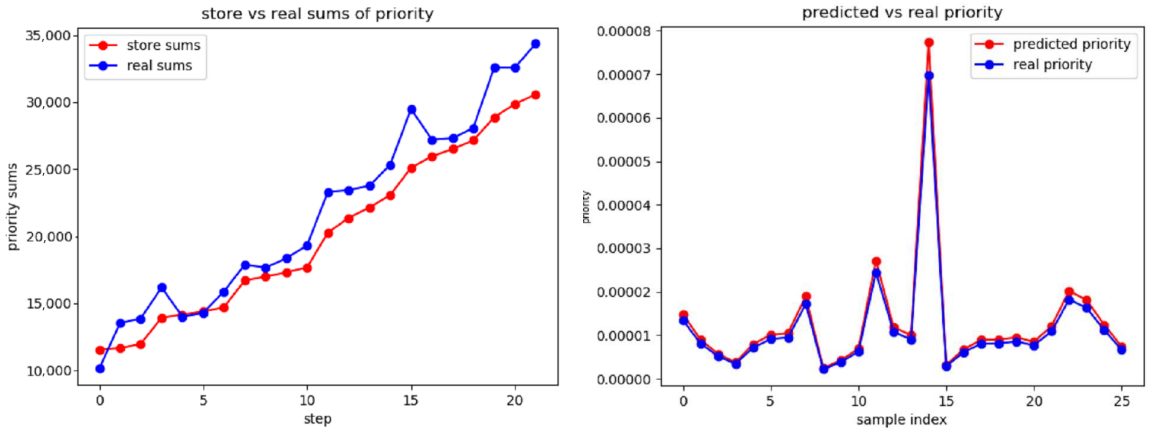


Figure 3. The architecture of the Imp-PER on DDQN.

3.3. Real Probability Prediction

To predict the probability $q(i)$ of each data i, it is necessary to estimate real TD-error of data i and predict the sum of all priorities in EM, $sum = \sum_i q_i^\alpha$, where the q_i is the TD-error of data i. The real TD-error of data i can be estimated by the current DDQN, which is defined in the preliminary section. In our algorithm, we only need to calculate the real TD-error for the data that are replayed. The real TD-error can be obtained directly from the DDQN loss function, as $|\tilde{\delta}_i| = |Q(x;\boldsymbol{\theta}) - y|$. Therefore, estimating the real TD-error does not incur additional computational costs. The key problem is how to predict the sum of real TD-error in EM.

Bai et al. found that there is an approximately linear relationship between real priority and stored priority [18]. However, it is not necessary to predict the priorities of all data in EM. What we care about is the relationship between the sum of real and stored priorities in EM. Although real and stored priority distributions are different (Figure 1), their sums are strongly correlated and change slowly over time, which is shown in Figure 4a. This correlation is also easy to explain because the TD-error will not change significantly in the short term, as the DQN is trained with a sufficiently small learning rate.



(**a**) store vs real sum of priorities (**b**) predicted vs real priority

Figure 4. Using a simple linear model to predict the sum of real priorities. Based on the predicted value, we can estimate the real priority of each sample.

In this work, we use a simple model to predict the real sum based on stored sum. Meanwhile, the timestamps of data in EM also have a strong correlation with the sum of priorities. As a result, we use a linear model to predict the real sum of priorities. Precisely, there are two features in our model, one is the sum of stored priorities $x^{(1)}$, which is the root of sum-tree in PER. The other one is the sum of timestamps $x^{(2)}$. The target value is the sum of real priorities z. The linear model is defined by Equation (7).

$$\begin{aligned} \hat{z} &= W_1 x^{(1)} + W_2 x^{(2)} + W_3 \\ x^{(1)} &= \sum_{i=1}^{n} (|\delta_i| + \epsilon)^\alpha \\ x^{(2)} &= \sum_{i=1}^{n} \tau_i \end{aligned} \tag{7}$$

where W_i is the parameters of our model. To solve the linear model, we only need to minimize the Mean Square Error (MSE) defined in Equation (8).

$$J(W) = \frac{1}{m} \sum_{i=1}^{m} (z_i - \hat{z}_i)^2 \tag{8}$$

where m is the size of training data and z_i is the real probability. Imp-PER uses the historical sum of data as training data, which is shown in Figure 4a, to fit the linear model. Based on the predicted sum z and real TD-error $\tilde{\delta}_i$, it is able to estimate the predicted probability of each data in the mini-batch, $q(i) = q_i^\alpha / z$. Figure 4b shows the predicted and real probabilities of data during training on Atari game Pong. It indicates that our method can accurately predict the real probabilities of data.

As we mentioned in Equation (6), the original importance weights v must be normalized by $max_i v_i$. We should estimate the minimum real value min_q. Using the same function Equation (7), we could get the value min_q. It is costly to calculate the min_q every time before estimate the loss of DDQN. The next subsection gives the solution to handle it.

3.4. Execution Optimization

To improve prediction accuracy while reducing extra time overhead, we perform several optimizations in the Imp-PER algorithm. Specifically, we updated the linear model per K steps. However, in the first few steps, we lack training data (sum of all data). In each update, we only obtained one training data, which is computed from current EM. Therefore, in the first few steps, we split the whole EM into M fragmentation. Each fragmentation X_i can be treated as a smaller EM. The training set X is defined by Equation (9).

$$\begin{aligned} X &= (X_1, X_2, \cdots, X_M) \\ X_i &= \left(\sum_{j=1}^{N} (|\delta_{ij}| + \epsilon)^\alpha, \sum_{j=1}^{N} \tau_{ij} \right) \end{aligned} \tag{9}$$

where N is the capacity of fragmentation EM.

On the other hand, to reduce the cost of predicting sum and a minimum of p, Imp-PER predicts the sum every D steps. From our preliminary experiments, we found that the sum and minimum of priority change smoothly. Figure 4a shows the evidence. In this work, we adopt the Exponential Moving Average to smooth the predicted value, which is defined by Equation (10).

Where ρ_1 and ρ_2 are smoothing parameters. To verify the prediction effect, we performed experiments on the pong task. As the training progresses, the sum and min_q prediction will become more accurate.

$$\begin{aligned} \tilde{z} &= \rho_1 \tilde{z} + (1 - \rho_1) z \\ min_q &= \rho_2 min_q + (1 - \rho_2) \frac{min_{td}}{\tilde{z}} \end{aligned} \tag{10}$$

3.5. Importance Weighted PER

Unlike ATDC-PER [18], the Imp-PER will not use the predicted value to update EM. Based on the predicted and stored probabilities of data, Imp-PER uses the importance weights to correct the bias of priorities. From the definition in preliminary, the real probability distribution is $q(x)$ and the stored probability distribution is $p(x)$. A mini-batch of data is sampled from EM based on $p(x)$. Our target is to estimate the loss function $f(x)$ under distribution $q(x)$. Using the importance weight, we can do that by Equation (11).

$$E_{x \sim q(x)}[f(x)] = E_{x \sim p(x)} \left[\frac{q(x)}{p(x)} f(x) \right] \tag{11}$$

From the definition of expectation, It is easy to prove the correctness of Equation (11). According to Equation (6), the importance weighted loss is defined by Equation (12).

$$loss(\boldsymbol{\theta}) = E_{x \sim p(x)} \left[\frac{q(x)}{p(x)} \left(\frac{min_i q(i)}{q(x)} \right)^\beta (Q(x;\boldsymbol{\theta}) - y)^2 \right] \tag{12}$$

Figure 5 illustrates the update process. Training DQN with uniform sampling from the EM usually converges to a locally optimal solution. It mostly samples the data with low magnitudes TD-errors, which gives a slower direction. PER based on real priority always chooses the data with high magnitudes TD-error, which gives the steepest direction to minimize Q loss. In most cases, it will converge to a position closer to the global solution. However, Due to outdated TD errors, PER based on storage priority sometimes gives the wrong direction. It significantly increases the steps to reach an optimal solution. In this work, we use the importance sampling to correct the direction of the update, which makes the wrong direction closer to the steepest direction.

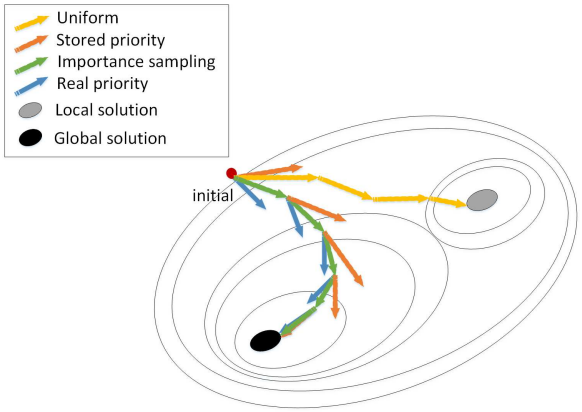
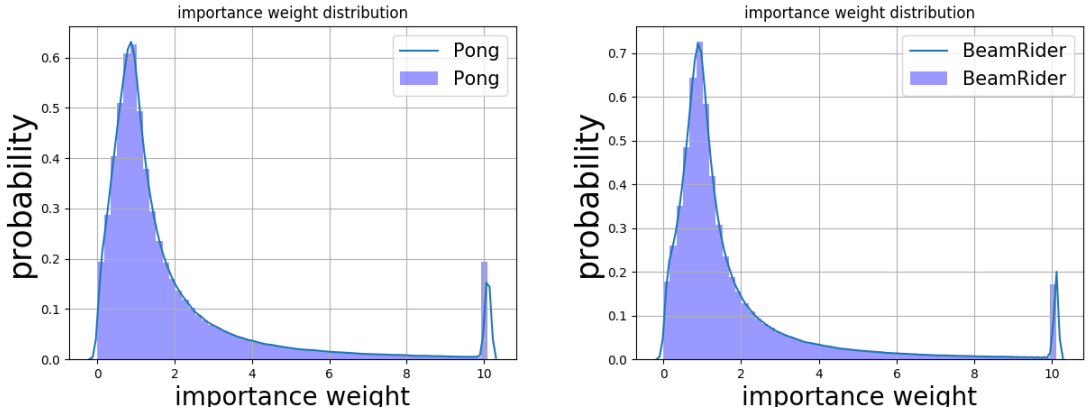


Figure 5. The convergence of uniform sampling, PER based on store priority, PER based on real priority, and PER with importance sampling in the training process, respectively. The contour lines constitute the space curve formed by the Q loss of all experiences.

However, the importance weights $w = q(x)/p(x)$ has an unbounded variance, which would affect the training stability. Figure 6 shows the distribution of importance weight on some Atari tasks. It is easy to observe that about 2% of the importance weights are greater than 10.



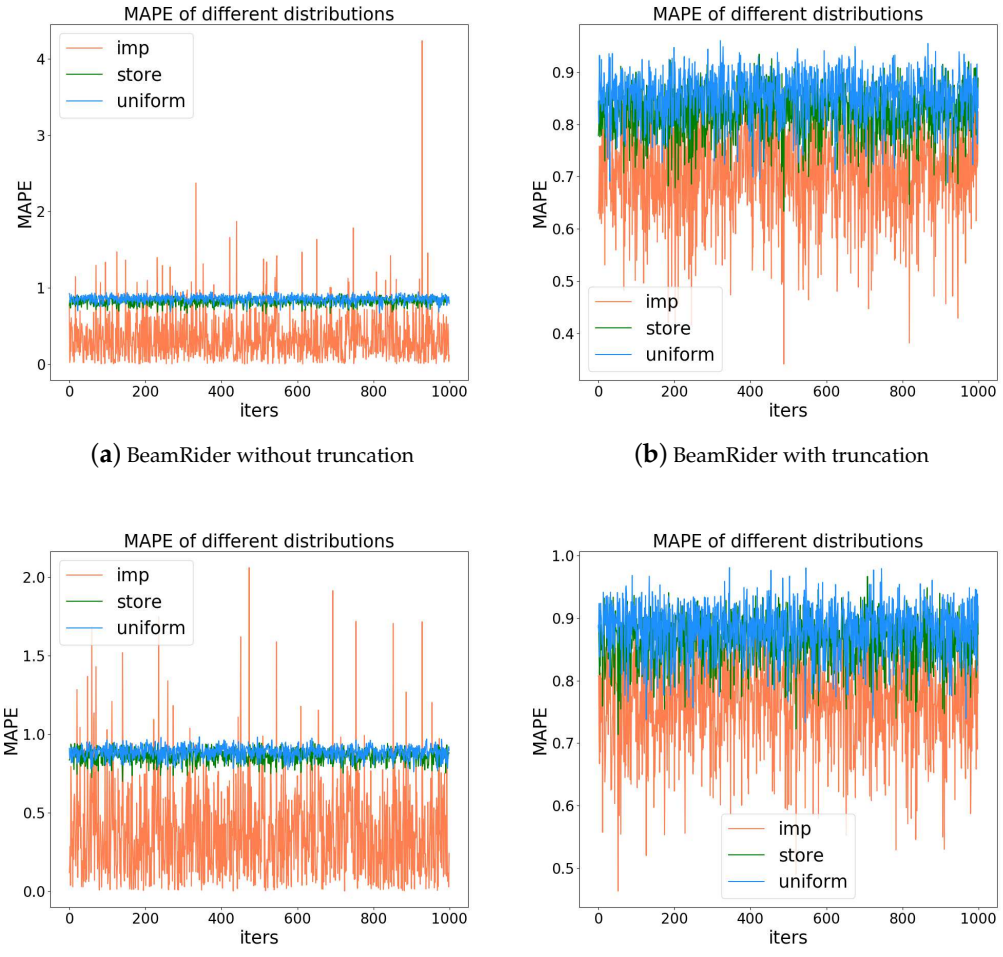
(**a**) the importance weight of Pong (**b**) the importance weight of BeamRider

Figure 6. The distribution of importance weight on Atari tasks. The bars represent statistical histograms, and the curves are the corresponding kernel density estimates.

Like the original PER, we can normalize weights by $1/max_i w_i$ so that they only scale the update downwards [9]. However, experiments on multiple tasks show that the method will reduce the training efficiency in most cases. Another way to reduce the variance is to directly truncate the importance weight to limit it between $[\tau_{min}, \tau_{max}]$. Where the τ_{min} and τ_{max} are fixed values calculated in advance. Truncating importance weights will introduce bias. Ionides proved that truncation at rate $\sqrt{n}$ is a good general choice [33], where n represents the size of mini-batch. For example, in our experiment the

batch size is set to 32, then the truncated importance weight belongs to $[0.0, 5.657]$. It can also be seen from Figure 6 that such a setting is reasonable.

For better understanding, we extract the stored and real priorities during the training process of BeamRider and verified the effect of truncation. We regard the loss function of DQN based on real priority as the target value y, and calculate the error between the predicted value $\hat{y}$ and y at different priorities. Here, we use Mean Absolute Percentage Error (MAPE) to measure the accuracy Equation (13). Figure 7 shows the errors based on uniform distribution, storage priority, and importance correction. The left column shows the comparison without truncation. The y-axis represents MAPE, and the x-axis represents the number of tests. Obviously, through the importance correction, the estimated value of the loss function is closer to the true value (based on the true priority). The right column shows the comparison with truncation $\tau_{min} = 0$ and $\tau_{max} = 5.657$. It can be found that the variance decreases but the bias increases. The experiment shows that setting $\tau_{max} = \sqrt{n}$ is a good trade-off between variance and bias.



(**a**) BeamRider without truncation

(**b**) BeamRider with truncation

(**c**) RoadRunner without truncation

(**d**) RoadRunner with truncation

Figure 7. The Mean Absolute Percentage Error (MAPE) of uniform, stored, and importance sampling. Each experiment runs 1000 times. The red, green, and blue curves represent imp, store, and uniform, respectively.

3.6. Algorithm Description

Algorithm 1 shows the whole training process of Double DQN with Imp-PER. In our preliminary experiments, we find it is unnecessary to update the linear model at each iteration. We train the linear model every K steps. Lines 6–9 show the linear model training process. The most costly step

is to calculate the real priority of each data (line 8), which needs to sweep the whole EM by DDQN. The parameter K has little effect on the accuracy of the model prediction. In our experiments, K is set to 10^5, which greatly reduces the time overhead of training the linear model. Lines 17–25 are the main process of Imp-PER. First, we sample a mini-batch of data. The data j is sampled according to probability $p(j)$ (line 18), which is the stored priority. Then, real TD-error is computed by DDQN (line 19). We use the current TD-error $\tilde{\delta}_j$ to update the stored priority of data j (line 20). By using the real $\tilde{\delta}_j$ and predicted sum $\tilde{z}$, we could compute the real probability q_j (line 21). Different from the original PER, the IS weight uses real probability q_j instead of stored probability p_j (line 22). Then, our IS weight is computed based on p_j and q_j (line 23). IS weight must be truncated by $[\tau_{min}, \tau_{max}]$. The two IS weights are used to update DDQN (line 24). As we mentioned before, the DDPG algorithm has a similar training framework. DDPG + PER [12] shows the details of DDPG's implementation.

Algorithm 1 Double DQN with Imp-PER.

Input: Linear model update period K, sum predict period D, budget T, mini-batch size k, learning rate η, discount factor γ, importance strength β, exponential smoothing ρ, smoothing index α, target network update L

Output: the parameter of DDQN $\boldsymbol{\theta}$

1: choose action $a_0 \sim \epsilon - greedyQ(s_0, a; \boldsymbol{\theta})$
2: **for** t = 1 to T **do**
3: execute a_{t-1} and observe s_t, r_t.
4: store tuple $(s_{t-1}, a_{t-1}, s_t, r_t)$ into EM with maximal priority $p_t = max_{i<t} p_i$
5: **if** t%K==0 **then**
6: calculate feature $x^{(1)} = \sum_{i=1}^{n} (|\delta_i| + \epsilon)^{\alpha}$
7: calculate feature $x^{(2)} = \sum_{i=1}^{n} \tau_i$
8: predict real TD-error $\tilde{\delta} = Q(s_t, a_t; \boldsymbol{\theta_t}) - y$
9: estimate parameters w of Linear model based on Equation (8)
10: **end if**
11: **if** t%D==0 **then**
12: calculate feature $x^{(1)} = \sum_{i=1}^{n} (|\delta_i| + \epsilon)^{\alpha}$
13: calculate feature $x^{(2)} = \sum_{i=1}^{n} \tau_i$
14: predict sum z and minimum td-error based on Equation (7)
15: smooth sum $\tilde{z}$ and min_q according to Equation (10)
16: **end if**
17: **for** j = 1 to k **do**
18: sample data $j \sim p(j)$ according to Equation (4)
19: compute current TD-error $\tilde{\delta}_j = Q(s_j, a_j; \boldsymbol{\theta_t}) - y_j$
20: update priority $p_j \leftarrow |\tilde{\delta}_j|$
21: compute real priority probability $q_j = \frac{(|\tilde{\delta}_j| + \epsilon)^{\alpha}}{\tilde{z}}^{-\beta}$
22: compute org IS weight $v_j = \left(\frac{q_j}{min_q}\right)$
23: new IS weight $w_j = clip(\frac{q_j}{p_j}, [\tau_{min}, \tau_{max}])$
24: $\Delta \leftarrow \Delta + w_j \cdot v_j \cdot \tilde{\delta}_j \cdot \nabla_{\theta} Q(s_j, a_j; \boldsymbol{\theta_t})$
25: **end for**
26: update weights $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \cdot \Delta$, reset $\Delta = 0$
27: **if** t%L==0 **then**
28: update target Q network $\boldsymbol{\theta}^{-} = \boldsymbol{\theta}$
29: **end if**
30: choose next action $a_t \sim \epsilon - greedyQ(s_t, a; \boldsymbol{\theta})$
31: **end for**

4. Experiments

In this section, we conduct experiments to evaluate Imp-PER. Our goal is to improve data utilization in DDQN and DDPG. There are three questions we need to verify: ***Q*1**: Compared to the

original PER, can Imp-PER effectively improve data utilization? ***Q2***: How much additional time does our method cost? ***Q3***: Is Imp-PER compatible with value-based and policy-based DRL? First, we give detailed settings for our experiments. Then, we will discuss the two questions.

4.1. Experimental Setting

We evaluated the performance of Imp-PER in 10 Atari 2600 games using the arcade learning environment [38]. The target value-based DRL algorithm is DDQN. The target policy-based DRL algorithm is DDPG. The compared algorithms are listed as follows.

- **Vanilla-ER**: the original experience replay, where each data is sampled uniform from experience memory.
- **Org-PER**: the original proportional prioritized experience replay. [9], where each data is sampled according to stored probability.
- **ATDC-PER**: the latest priority correction algorithm, which uses machine learning to predict the priorities of all data in EM [18].

To be fair, all comparison algorithms use the same network structures and are implementated based on OpenAI's baseline (DDQN) [39] and DDPG [12], which are defined in Figures 8 and 9.

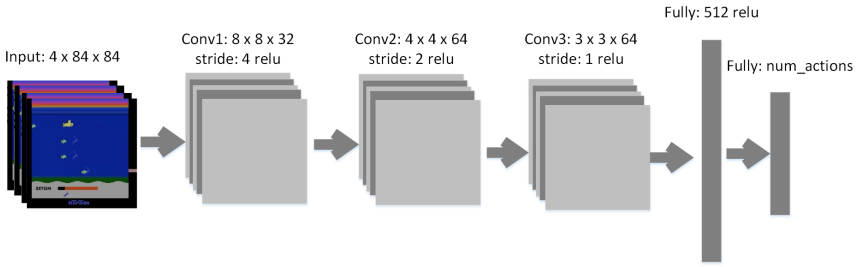


Figure 8. The network architecture of DDQN.

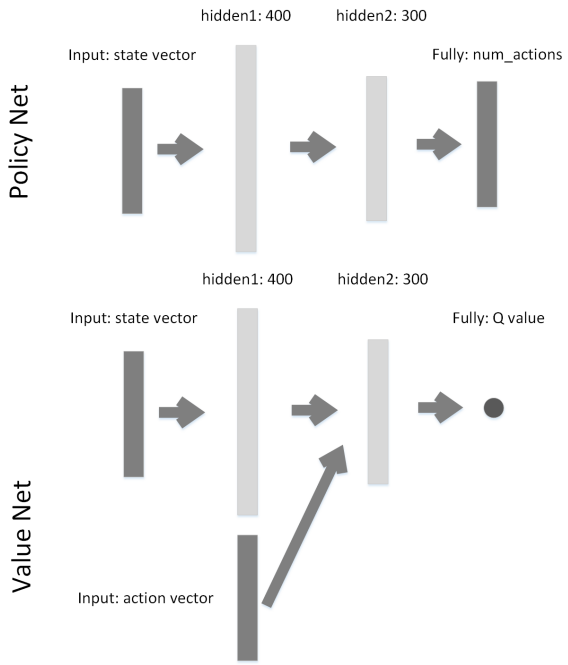


Figure 9. The network architecture of DDPG.

DDQN configuration.

All compared algorithms are implemented in python using Tensorflow 1.10.0 version [40]. The network architecture of DDQN contains three convolutional layers and two fully connected

layers. The network architecture is the same as DQN in research [1]. The main hyperparameters of all compared algorithms are shown in Table 1. Besides, the training period of the prediction model in ATDC-PER and Imp-PER is set to 10^5. In our method, other hyperparameters need to be set. The period of sum prediction is $D = 10^4$ and the exponential smoothing are $\rho_1 = \rho_2 = 0.3$. The priorities are updated only when data are replayed. The ATDC-PER will update the priorities of the whole EM at each iteration. Adam optimizer is used to update the parameters of DDQN and the epsilon is set to 0.0001. The gradients are norm clipped by 10. All experiments are evaluated on a server with 48 Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz processors and 4 GeForce GTX-1080 Ti 12GB GPU.

Table 1. The main hyperparameters for every baseline. There are some other hyperparameters that are the same as ATDC-PER. All tasks use the same hyperparameter settings.

Hyparameter	Memory	α	β	Batch Size	γ	η	Target Q	ε
Value	10^6	0.6	0.4→1.0	32	0.99	0.0001	40,000	0.0→1.0

DDPG configuration.

All compared algorithms are implemented in python using Tensorflow 1.10.0 version [40]. The network architecture of DDPG contains a policy network and value network. Both network architectures are the same as DDPG in research [12]. The main hyperparameters of all compared algorithms are shown in Table 2. Besides, the training period of the prediction model in Imp-PER is set to 10^4. The period of sum prediction is $D = 10^4$ and the exponential smoothing are $\rho_1 = \rho_2 = 0.3$. Adam optimizer is used to update the parameters of DDPG. The gradients are norm clipped by 5. The target networks are updated by momentum with 0.01. All experiments are evaluated on a server with 48 Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz processors and 4 GeForce GTX-1080 Ti 12GB GPU.

Table 2. The main hyperparameters for every baseline. There are some other hyperparameters that are the same as DDPG + PER [12]. All tasks use the same hyperparameter settings.

Hyparameter	Memory	α	β	Batch Size	γ	η	Noise σ	Noise θ
Value	10^4	0.6	0.4→1.0	32	0.99	0.0001	0.2	0.15

4.2. Performance Comparison

Accurately predicting priorities is critical to estimating importance weights. First, we compare the accuracy of priority prediction with ATDC-PER. Then, we analyze the time cost of our algorithm compared to ATDC-PER and Org-PER. At the end of this section, we compare the learning speed and data utilization of all algorithms.

4.2.1. Accuracy of Priority Prediction

First of all, we need to verify the accuracy of TD-error's sum. In our experiment, the training period of the predicted model is 10^5. During this period, the prediction model should keep accuracy. Figure 10 shows the accuracy of predicted TD-error's sum on game BeamRider and Pong. The green curve represents the predicted TD-error's sum which using the last updated model. The blue curve represents the predicted TD-error's sum which uses the current updated model. The prediction accuracy of the model is guaranteed.

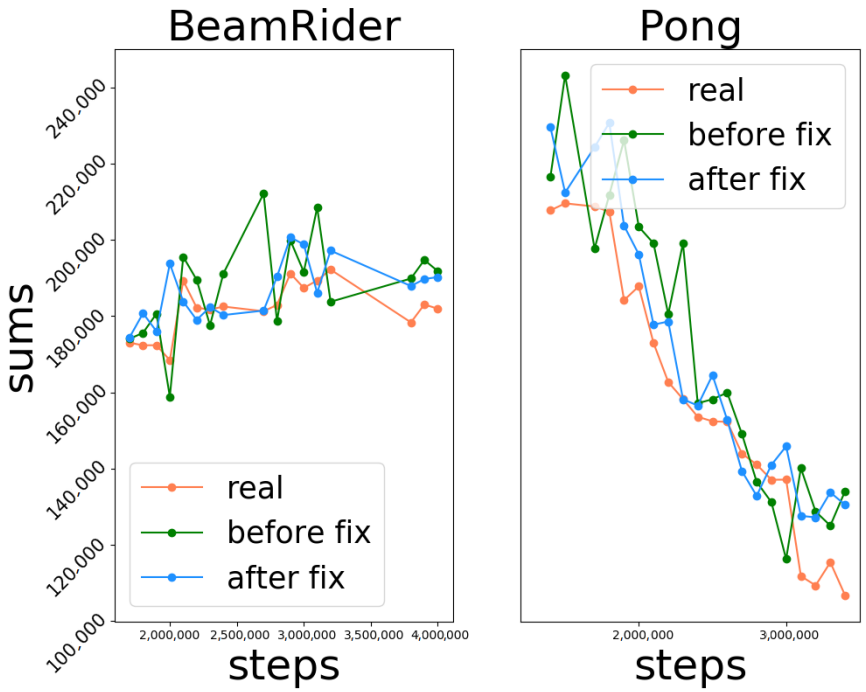


Figure 10. The accuracy of predicted TD-error's sum on game BeamRider and Pong. The coral, green, and blue curves represent real, before fix model, and after fix model, respectively.

Based on this prediction model, we can directly calculate the true priority. In this work, we use the Mean Absolute Percentage Error (MAPE) [41] to estimate accuracy, which is defined as Equation (13). As we only need the priorities of the current mini-batch, the m in Equation (13) is set to the batch size.

$$MAPE = \sum_{i=1}^{m} \left| \frac{y_i - \hat{y}_i}{y_i} \right| \times \frac{100}{m} \tag{13}$$

where y_i is the real value and $\hat{y}_i$ is the predicted value. We split the training process into three stages, and we sample 2000 data at each stage. The mean MAPE of 2000 data represents the accuracy in each stage. Table 3 shows the result of each stage. Our algorithm Imp-PER has very high accuracy compared to ATDC-PER. There are two reasons for this: One is that the sum of prioritizing is very predictable compared to the priority of each data, which is shown in Figure 4a. The other one is that we calculate the real TD-errors of data with absolute accuracy. These two reasons determine the accuracy of the final priority prediction, which is shown in Figure 4b. Meanwhile, as the training progresses, the prediction accuracy of both algorithms is gradually improving.

Table 3. The MAPE of each method on game Pong. Each cell represents the mean value of the corresponding period. Bold font shows the best performance.

Methods	$\mathbf{0-2(e^7)}$	$\mathbf{2-4(e^7)}$	$\mathbf{4-6(e^7)}$
ATDC-PER	76	54	48
Imp-PER	**6**	**3**	**0.7**

4.2.2. Time Cost Analysis

Before comparing the data utilization, we analyze the time cost in each algorithm. All-time measurements are done under the same server configuration. In each iteration, there are three main processes here. The first one is the sample, which needs to search on the sum-tree. When the capacity

of EM goes larger, the sampling time, whose time complexity is $O(logN)$, becomes a bottleneck. The second one is PER update, which is the same time complexity as sampling. The last one is the DDQN or DDPG update, which is executed on GPU. As we mentioned before, it is costly to correct all priorities by using current DDQN or DDPG. We measure the time cost to correct all priorities of EM (capacity is 10^6). All data must be predicted by DDQN on GPU, it needs 150+ s. We can see that the update cost is very high.

The time cost is shown in Table 4. The second column shows the total time cost at each iteration. ATDC-PER will spend $10\times$ time compare to Org-PER. In the third column, we can see ATDC-PER needs more time to correct the priorities. However, our method only spends a little more time. Because we do not need to correct the priorities of all data in EM. On the other hand, the time to update PER is the same, which is only updates the priorities of the current mini-batch of data. In the last column, our algorithm Imp-PER spend a little more time to update DDQN or DDPG. This extra time is used to calculate the importance weights. Except for these times, our method and ATDC-PER both need to train the priority prediction model. The time cost of ATDC-PER is 160+ s, while Imp-PER spends about 20 s. To further reduce training time, the prediction model is trained per 10^5 steps. Therefore, the time cost of the prediction model is ignored. However, the total time also contains some small processing, including the cost of eliminating outliers and so on.

Table 4. The time cost in each algorithm (s) on Atari with DDQN. The total time also includes other processes which are not the main factors.

Method	Total	Sampling	Update	DDQN
Org-PER	0.014	0.005	0.0028	0.006
ATDC-PER	0.150	0.075	0.0028	0.006
Imp-PER	0.015	0.005	0.0029	0.007

Although ATDC-PER can effectively improve data utilization, its execution time is too long to make it widely used. Our algorithm Imp-PER could effectively improve data utilization without significant additional time overhead. From the time analysis, Imp-PER is only 10% slower than Org-PER. Compared with the improved data utilization and the improved policy quality, this time cost is completely acceptable. Specifically, data utilization and policy quality improvement refer to using the same data to get a higher score (policy quality). In subsection X and subsection Y, we made a detailed comparison of related indicators of data utilization and policy quality. In general, Imp-PER uses about 10% more training time, and gets about 30% improvement in data utilization and 15% (atari) and 51% (mujoco) final policy quality improvements, making the PER technology more effective and practical. The next section provides more analysis details.

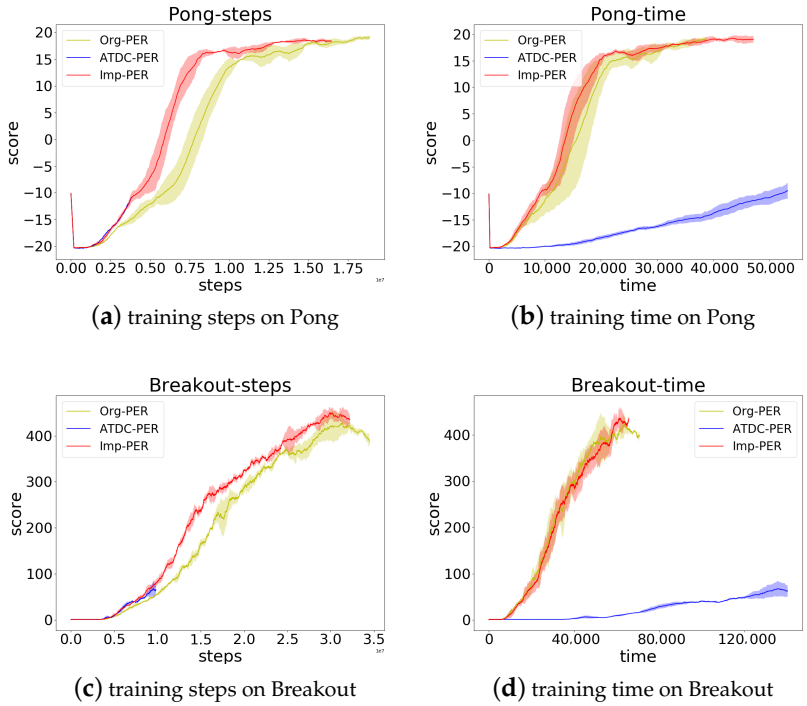
4.2.3. Data Utilization Comparison

The data utilization is defined as the number of training data to reach a fixed score. Each game is run five times using different random seeds until DDQN or DDPG convergence. The mean score is used, and the shaded area represents a mean $+/-$ standard deviation.

Figure 11a shows the score with training steps on Pong (DDQN). The x-axis is the training step, which also represents data utilization. The y-axis represents the mean score for the recent 100 episodes. Our Imp-PER has the same data utilization as ATDC-PER, which is better than the Org-PER. The data utilization is improved by about 30% on game Pong. Figure 11b shows the training curves with the time cost. The x-axis represents the time (seconds). From Section 4.2.2, ATDC-PER is $10\times$ slower than Org-PER and Imp-PER is 10% slower than Org-PER. Figure 11b shows the same results as our analysis. Although Imp-PER is slower than Org-PER in each iteration, we use less training time to convergence because of higher data utilization. The same result on game Breakout is shown in Figure 11c,d.

The results of the other eight Atari games are shown in Figure 12. We ignore the ATDC-PER in the other games and only compare Imp-PER to Org-PER. The first column and third column are

data utilization on each game. Correspondingly, the second and fourth column is time spent on each game. In most games, our Imp-PER has higher data utilization than Org-PER. From the results, we can see that Imp-PER has almost no additional time overhead, which reduces the whole training time on most games.



(**a**) training steps on Pong (**b**) training time on Pong

(**c**) training steps on Breakout (**d**) training time on Breakout

Figure 11. The performance compares on Pong and Breakout with DDQN. We also show the performance of the ATDC-PER algorithm. ATDC-PER spends more than 10x wall time and we stop them at maximum wall time.

In different stages of training, our algorithm has different degrees of data utilization improvement compared to Org-PER. Imp-PER has better improvement during the early stages. Table 5 shows the average scores achieved by Org-PER and Imp-PER in different training stages (0.50×10^7, 1.00×10^7, and 1.50×10^7, steps respectively). In order to Intuitively show the performance improvement of each stage, we estimate the relative improvement of scores. Figure 13 shows the results. Each stage is represented by a different color.

Table 5. The scores achieved by Org-PER and Imp-PER. Each value is estimated by 5 random seeds. The results shows the average scores at 0.50×10^7, 1.00×10^7, and 1.50×10^7 training steps respectively. Bold font shows the best performance.

Game	Org-PER			Imp-PER		
	0.50×10^7	1.00×10^7	1.50×10^7	0.50×10^7	1.00×10^7	1.50×10^7
Pong	−12.1	13.5	18.6	**7.3**	**16.6**	**19**
BankHeist	151.8	279.2	400	**178.2**	**332**	**466.6**
Boxing	11.9	34.6	55.3	**17.3**	**38**	**60**
Breakout	10	63.27	174.9	**16.65**	**93.24**	**249.95**
FishingDerby	−87.15	−64.29	−20	**−77.85**	**−40**	**−7.86**
Freeway	8.3	13.57	19.28	**8.33**	**18.6**	**25**
RoadRunner	2000	18,400	26,800	**4000**	**21,200**	**30,000**
Robotank	3.64	7.3	13.1	**4.62**	**10.4**	**21.5**
Tutankham	187	**183**	**187**	**229**	166.1	156.4
Bowling	**25**	**22**	20.3	**25**	20.5	**24.5**

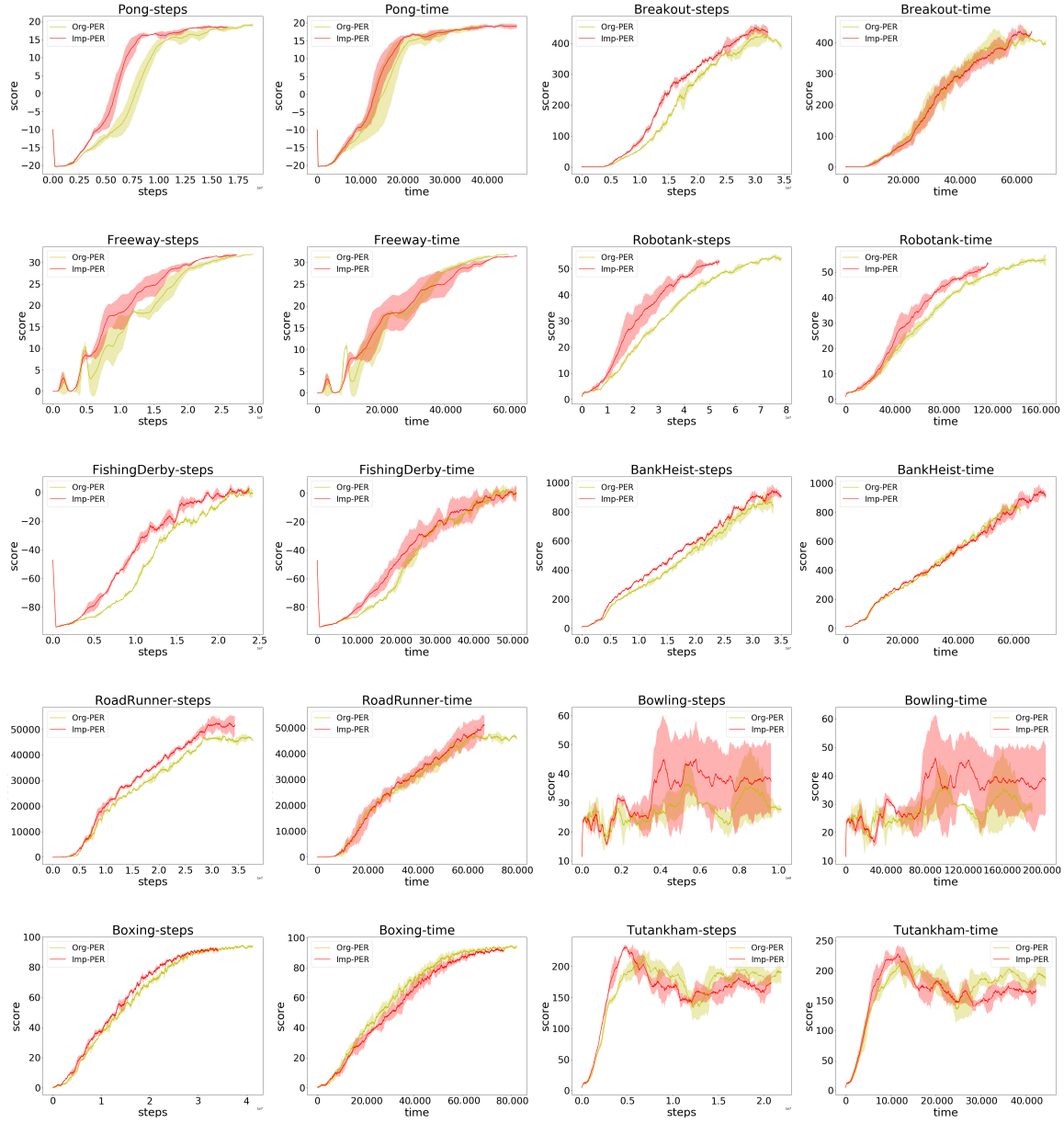


Figure 12. The performance compared to other games. Each task is trained with five different random seeds. The shaded area shows the standard deviation. The first column and third columns are data utilization on each game. Correspondingly, the second and fourth columns are time spent on each game.

The DDPG experiments are conducted on six continuous control tasks based on MuJoCo environments: Walker2d, InvertedPendulum, InvertedDoublePendulum, Hopper, Reacher, and Humanoid. Figure 14 shows the score with training steps on MuJoCo. The x-axis is the training episode, which represents data utilization. The y-axis represents the mean score for the recent 150 episodes. The results show that the Org-PER could improve data utilization compare to Vanilla-ER, and Imp-PER improves the data utilization further. On the other hand, Imp-PER could achieve a higher score compared to baselines (except for Reacher and Humanoid).

From the results of DDPG in Figure 14, the Imp-PER always outperforms the Org-PER algorithm. Same as DDQN on Atari games, the Imp-PER has better improvement at the early training stages. However, the performance of PER is degraded on task Humanoid. Although Imp-PER is better than Org-PER, it is lower than the Vanilla-ER algorithm. We guess that it is because we only use TD-error as a priority. From the analysis of ERO [14], the TD-error is not the best priority metric; sometimes

the TD-error metrics will harm the training process. All the hypotheses need to be verified in our future work.

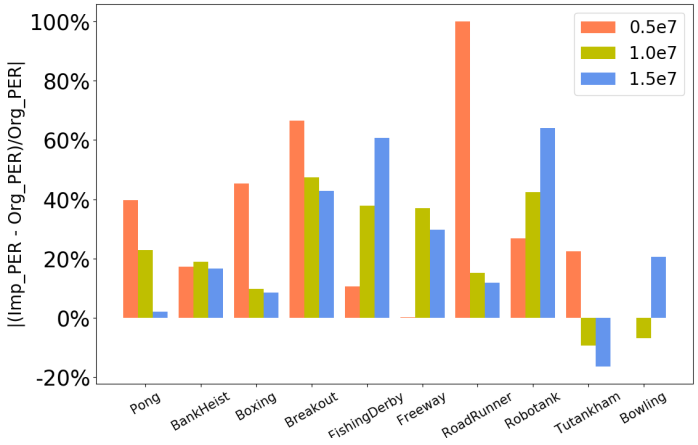
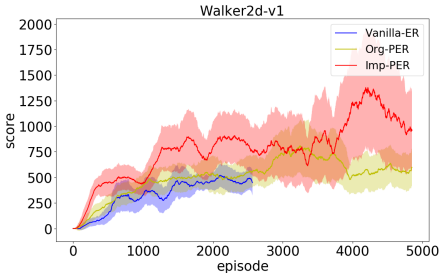
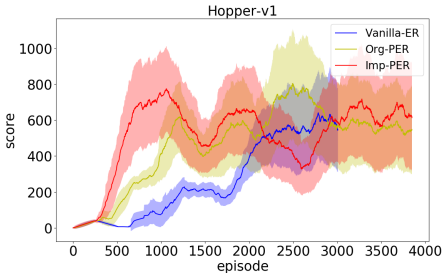


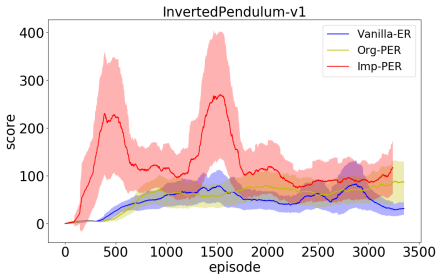
Figure 13. The relative scores achieved by Imp-PER under 5 random seeds. The results shows the relative scores at 0.50×10^7, 1.00×10^7, and 1.50×10^7 training steps, respectively.



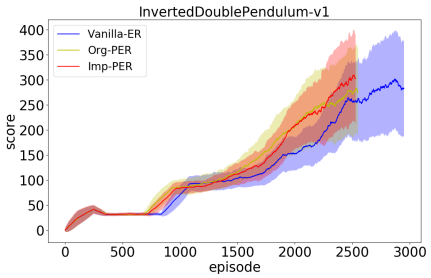
(**a**) training steps on Walker



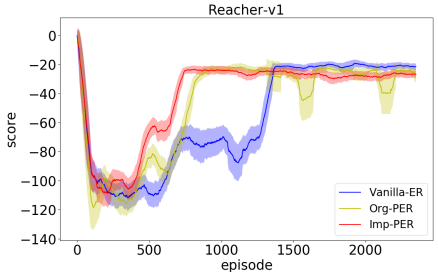
(**b**) training steps on Hopper



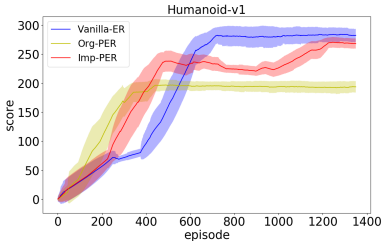
(**c**) training steps on InvertedPendulum



(**d**) training steps InvertedDoublePendulum



(**e**) training steps on Reacher



(**f**) training steps on Humanoid

Figure 14. The performance compares on MuJoCo with DDPG. Each task is run five times using different random seeds until DDPG convergence.

4.2.4. Final Policy Quality

The optimal policy reflects the final training result of the agent. Ensuring the quality of optimal policy is as important as improving the data utilization. We save the trained Q network at regular intervals. Then, we use the policy to interact with the environment with the exploration rate $\varepsilon = 0$. The highest recorded score represents the quality of the optimal policy. At the beginning of the test, the agent randomly interacts with the environment 1 to 31 times without action, providing a random initial state for the test. As the score range of each game is different, we use Equation (14) to calculate the relative score based on the actual score.

$$score_{normalized} = \frac{score_{agent} - score_{random}}{|score_{human} - score_{random}|} \tag{14}$$

where $score_{agent}$ is the score got by a trained agent. $score_{human}$ and $score_{random}$ represent the score achieved by human and random policy, respectively. If the normalized score is greater than 100%, it means that the agent surpasses the human level.

First, we compare the best scores got by different methods on Atari with DDQN. Table 6 shows the final score of each method. Imp-PER not only has higher data utilization but also is not less than the Org-PER algorithm in the final score. On the 7/10 of tasks, our final score was higher than Org-PER. On game Boxing, Robotank, and Tutankham, the score of our method is lower than Org-PER. We guess that TD-error is not a good estimation of priority on these tasks. If we use real TD as a priority, it will limit the training efficiency of DQN. However, it is just conjecture, and we will conduct in-depth research on this issue in the future.

Table 6. Final score and normalized score comparison of all methods on Atari. The best scores are shown in bold font. The uniform and human score are from deepmind [9]. Bold font shows the best performance.

Game	Uniform	Human	Raw Score			Normalized Score %		
			DDQN	Org-PER	Imp-PER	DDQN	Org-PER	Imp-PER
BankHeist	21.7	644.5	728.3	870.6	**951.6**	113.4	136.3	**149.3**
Bowling	35.2	**146.5**	50.4	53.5	60.5	13.6	16.4	**22.7**
Boxing	−1.5	9.6	81.7	**90.6**	90.3	749.5	**829.7**	827.0
Breakout	1.7	31.8	450	420.5	**451.3**	1489.3	1391.3	**1493.6**
FishingDerby	−77.1	**5.1**	3.2	3.1	3.4	97.6	97.5	**97.9**
Freeway	0.1	25.6	28.8	32	**32.2**	112.5	125.0	**125.8**
Pong	−20.7	9.3	**21**	20.7	**21**	**139**	138	**139**
RoadRunner	11.5	7845	48,330.5	50,113	**54,370.6**	616.8	639.5	**693.9**
Robotank	2.2	11.9	47.8	**53.7**	52.1	470.1	**530.9**	514.4
Tutankham	12.7	138.3	125.7	**207**	190	89.9	**154.6**	141.1

We also calculated the median and mean normalized scores to prove the performance of different methods on Atari with DDQN. Table 7 shows the result. Our method is better than both DDQN and Org-PER. Comparing to Org-PER, our method improve about 3% on the median and 15% on the mean normalized score.

Table 7. Median and mean normalized score of all games (%). Bold font shows the best performance.

Metrics	DDQN	Org-PER	Imp-PER
Median	113.4	138	**141.1**
Mean	389.2	405.9	**420.5**

Next, we compare the best scores got by different methods on MuJoCo with DDPG. In these tasks, we don't have the human score. In this work, we only compare the best scores achieved by Random,

Vanilla-ER, Org-PER and Imp-PER. Table 8 shows the result. Our method Imp-PER gains about 51% policy improvement compare to Org-PER.

Table 8. Final score and normalized score comparison of all methods on MuJoCo. The best scores are shown in bold font. The Random represents a random policy. The relative improvement represents the improved score compare to Org-PER. Bold font shows the best performance.

Tasks	Random	Vanilla-ER	Org-PER	Imp-PER	Relative Improvement
Walker2d	−4.34	500	800	**1360**	70%
Hopper	11	650	800	**950**	18.8%
InvertedPendulum	7.76	120	105	**275**	161.9%
InvertedDoublePendulum	54.11	300	275	**312**	13.5%
Reacher	−90	**−9**	−10	−10	0%
Humanoid	102.01	**284**	195	271	38.97%

In summary, the experimental results show that our method Imp-PER can improve the learning effect of the agent by improving the sampling efficiency in the experience memory, and make the agent converge to better final policy on both value-based and policy-based DRL.

5. Conclusions and Future Work

In this work, we analyzed the priority delay problem in PER and explained the negative impact of this problem on agent learning through theoretical and experimental data. The existing methods solve the problem by calculating and correcting the TD-errors of the entire EM, which the computational cost introduced is huge, and it is not applicable. We propose a simple and general framework Imp-PER. It is a cost-free method compared to existing methods. First, we design a linear model to predict the sum of real TD-error and estimate the real probability of the current batch of data. Compared to predicting the real probability of each sample in EM, the cost is much less. Next, by using IS technology to estimate the DQN or DDPG loss function for the real priority distribution, we improve the data utilization and eliminate the cost of updating the entire EM priority. The experimental results show that our Imp-PER could improve data utilization on most games and only 10% slower than Org-PER. Specifically, our algorithm could improve both value-based and policy-based RL methods. And Imp-PER also handles discrete state and continuous state tasks. Finally, our optimal policy is better than the original PER about 15% on the mean with DDQN and about 51% on the mean with DDPG.

The PER has a pivotal position in DRL and is widely used in DRL training. From a practical point of view, we improve data utilization while ensuring the training speed, which can solve practical problems more quickly. In terms of scientific research, our research analyzes the priority problem of PER in detail, and at the same time corrects the training of the agent from a new perspective. The Imp-PER framework can be used in any DRL algorithm based on PER, such as SAC with PER, ACER, Ape-x.

To conclude this article, we present here some issues for future study. First, Imp-PER cannot handle the rank-based prioritized experience replay, which is a variant of PER. The rank-based PER has a better performance in some tasks. We have already started research on this aspect. Second, we use clip technology to reduce the variance of importance weights, but introduce bias. How to ensure the bias and reduce the variance is still a challenge. These will be part of our future work.

Author Contributions: Conceptualization, H.Z. and C.Q.; methodology, H.Z., J.Z. and J.L.; software, H.Z. and C.Q.; writing, H.Z., J.Z. and J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: We would like to thank Network and Information Center, USTC for providing the GPU servers to run our experiments.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529–533. [CrossRef] [PubMed]
2. Wu, H.; Song, S.; You, K.; Wu, C. Depth Control of Model-Free AUVs via Reinforcement Learning. *IEEE Trans. Syst. ManCybern. Syst.* **2018**, *49*, 2499–2510. [CrossRef]
3. Moreira, I.; Rivas, J.; Cruz, F.; Dazeley, R.; Ayala, A.; Fernandes, B. Deep Reinforcement Learning with Interactive Feedback in a Human–Robot Environment. *Appl. Sci.* **2020**, *10*, 5574. [CrossRef]
4. Gregurić, M.; Vujić, M.; Alexopoulos, C.; Miletić, M. Application of Deep Reinforcement Learning in Traffic Signal Control: An Overview and Impact of Open Traffic Data. *Appl. Sci.* **2020**, *10*, 4011. [CrossRef]
5. Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. Mastering the game of Go without human knowledge. *Nature* **2017**, *550*, 354–359. [CrossRef]
6. Chung, H.; Lee, S.J.; Jeon, H.B.; Park, J.G. Semi-Supervised Speech Recognition Acoustic Model Training Using Policy Gradient. *Appl. Sci.* **2020**, *10*, 3542. [CrossRef]
7. Lin, L.-J. *Reinforcement Learning for Robots Using Neural Networks*; Technical Report; Carnegie Mellon University, School of Computer Science: Pittsburgh, PA, USA, 1993. Available online: https://apps.dtic.mil/dtic/tr/fulltext/u2/a261434.pdf (accessed on 3 July 1993).
8. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018.
9. Schaul, T.; Quan, J.; Antonoglou, I.; Silver, D. Prioritized experience replay. In Proceedings of the International Conference on Learning Representations 2016, San Juan, Puerto Rico, 2–4 May 2016.
10. Van Seijen, H.; Sutton, R.S. Planning by prioritized sweeping with small backups. In Proceedings of the International Conference on Machine Learning 2013, Atlanta, GA, USA, 17–19 June 2013; pp. 361–369.
11. Horgan, D.; Quan, J.; Budden, D.; Barth-Maron, G.; Hessel, M.; Van Hasselt, H.; Silver, D. Distributed prioritized experience replay. In Proceedings of the International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, 30 April–3 May 2018.
12. Hou, Y.; Zhang, Y. *Improving DDPG via Prioritized Experience Replay*; Technical Report; no. May. 2019. Available online: https://course.ie.cuhk.edu.hk/ierg6130/2019/report/team10.pdf (accessed on 5 October 2019).
13. Ying-Nan, Z.; Peng, L.; Wei, Z.; Xiang-Long, T. Twice sampling method in deep q-network. *Acta Autom. Sin.* **2019**, *45*, 1870–1882.
14. Zha, D.; Lai, K.H.; Zhou, K.; Hu, X. Experience replay optimization. In Proceedings of the International Joint Conference on Artificial Intelligence 2019, Macao, China, 10–16 August 2019; pp. 4243–4249.
15. Novati, G.; Koumoutsakos, P. Remember and forget for experience replay. In Proceedings of the International Conference on Machine Learning 2019, Long Beach, CA, USA, 10–15 June 2019; pp. 4851–4860.
16. Hessel, M.; Modayil, J.; Van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M.; Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence 2018, New Orleans, LA, USA, 2–7 February 2018.
17. Longji, L. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Mach. Learn.* **1992**, *8*, 293–321.
18. Chenjia, B.; Peng, L.; Wei, Z.; Xianglong, T. Active sampling for deep q-learning based on td-error adaptive correction. *J. Comput. Res. Dev.* **2019**, *56*, 262–280.
19. Hesterberg, T.C. Advances in Importance Sampling. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 1988.
20. Owen, A.B. Monte Carlo Theory, Methods and Examples. 2013. Available online: https://statweb.stanford.edu/~owen/mc/ (accessed on 15 October 2019).
21. Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. Openai gym. *arXiv* **2016**, arXiv:1606.01540. Available online: https://arxiv.org/abs/1606.01540 (accessed on 1 October 2019).
22. Van Hasselt, H.; Guez, A.; Silver, D. Deep reinforcement learning with double q-learning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
23. Wang, Z.; Schaul, T.; Hessel, M.; Hasselt, H.; Lanctot, M.; Freitas, N. Dueling network architectures for deep reinforcement learning. *arXiv* **2015**, arXiv:1511.06581. Available online: https://arxiv.org/abs/1511.06581 (accessed on 1 October 2019).

24. Cao, X.; Wan, H.; Lin, Y.; Han, S. High-value prioritized experience replay for off-policy reinforcement learning. In Proceedings of the 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI), Portland, OR, USA, 4–6 November 2019; pp. 1510–1514.
25. Hu, C.; Kuklani, M.; Panek, P. Accelerating Reinforcement Learning with Prioritized Experience Replay for Maze Game. *SMU Data Sci. Rev.* **2020**, *3*, 8.
26. Wang, X.; Xiang, H.; Cheng, Y.; Yu, Q. Prioritised experience replay based on sample optimisation. *J. Eng.* **2020**, *13*, 298–302. [CrossRef]
27. Fei, Z.; Wen, W.; Quan, L.; Yuchen, F. A deep q-network method based on upper confidence bound experience sampling. *J. Comput. Res. Dev.* **2018**, *55*, 100–111.
28. Isele, D.; Cosgun, A. Selective experience replay for lifelong learning. In Proceedings of the National Conference on Artificial Intelligence 2018, New Orleans, LA, USA, 2–7 February 2018; pp. 3302–3309.
29. Zhao, D.; Liu, J.; Wu, R.; Cheng, D.; Tang, X. Optimistic sampling strategy for data-efficient reinforcement learning. *IEEE Access* **2019**, *7*, 55763–55769. [CrossRef]
30. Sun, P.; Zhou, W.; Li, H. Attentive experience replay. In Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence 2020, New York, NY, USA, 7–12 February 2020; pp. 5900–5907.
31. Bu, F.; Chang, D.E. Double Prioritized State Recycled Experience Replay. *arXiv* **2020**, arXiv:2007.03961.
32. Yu, T.; Lu, L.; Li, J. A weight-bounded importance sampling method for variance reduction. *Int. J. Uncertain. Quantif.* **2019**, *9*, 3. [CrossRef]
33. Ionides, E.L. Truncated importance sampling. *J. Comput. Graph. Stat.* **2008**, *17*, 295–311. [CrossRef]
34. Thomas, P.S.; Brunskill, E. Importance sampling with unequal support. In Proceedings of the National Conference on Artificial Intelligence 2016, Phoenix, AZ, USA, 12–17 February 2016; pp. 2646–2652.
35. Martino, L.; Elvira, V.; Louzada, F. Effective sample size for importance sampling based on discrepancy measures. *Signal Process.* **2017**, *131*, 386–401. [CrossRef]
36. Chatterjee, S.; Diaconis, P. The sample size required in importance sampling. *Ann. Appl. Probab.* **2018**, *28*, 1099–1135. [CrossRef]
37. Andre, D.; Friedman, N.; Parr, R. Generalized prioritized sweeping. In Proceedings of the Advances in Neural Information Processing Systems 1998, Denver, CO, USA, 30 November–5 December 1998; pp. 1001–1007.
38. Bellemare, M.G.; Naddaf, Y.; Veness, J.; Bowling, M. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.* **2013**, *47*, 253–279. [CrossRef]
39. Dhariwal, P.; Hesse, C.; Klimov, O.; Nichol, A.; Plappert, M.; Radford, A.; Schulman, J.; Sidor, S.; Wu, Y.; Zhokhov, P. *Openai Baselines*; GitHub Repository; GitHub: San Francisco, CA, USA, 2017.
40. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. Tensorflow: A system for largescale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
41. De Myttenaere, A.; Golden, B.; Le Grand, B.; Rossi, F. Mean absolute percentage error for regression models. *Neurocomputing* **2016**, *192*, 38–48. [CrossRef]