Exploration Bonuses Based on Upper Confidence Bounds for Sparse Reward Games

Naoki Mizukami[1(✉)], Jun Suzuki[2], Hirotaka Kameko[1], and Yoshimasa Tsuruoka[1]

[1] Graduate School of Engineering, The University of Tokyo, Tokyo, Japan
{mizukami,kameko,tsuruoka}@logos.t.u-tokyo.ac.jp
[2] NTT Communication Science Laboratories, NTT Corporation, Kyoto, Japan
suzuki.jun@lab.ntt.co.jp

Abstract. Recent deep reinforcement learning (RL) algorithms have achieved super-human-level performance in many Atari games. However, a closer look at their performance reveals that the algorithms fall short of humans in games where rewards are only obtained occasionally. One solution to this sparse reward problem is to incorporate an explicit and more sophisticated exploration strategy in the agent's learning process. In this paper, we present an effective exploration strategy that explicitly considers the progress of training using exploration bonuses based on Upper Confidence Bounds (UCB). Our method also includes a mechanism to separate exploration bonuses from rewards, thereby avoiding the problem of interfering with the original learning objective. We evaluate our method on Atari 2600 games with sparse rewards, and achieve significant improvements over the vanilla asynchronous advantage actor-critic (A3C) algorithm.

1 Introduction

Atari games are one of the most commonly used benchmark environments for deep RL algorithms [1–3]. Bellemare et al. [1] categorized Atari games by two properties on their difficulty levels for RL algorithms. The first property is the ease of exploration, which categorizes games into two types: *easy exploration* and *hard exploration*. In easy exploration games, the agent can find a high-scoring policy using simple local exploration strategies such as ϵ-greedy. Hard exploration games are further categorized based on the density of the rewards. Recent deep RL algorithms have achieved human-level performance in games with dense rewards.

However, hard exploration games with sparse rewards remain difficult for the current deep RL algorithms. In such games, the agent rarely reaches a rewarding state if it uses purely random exploration strategies that are employed in many current deep RL algorithms. One potential solution to this problem is to use a more sophisticated exploration strategy that forces the agent to select actions that would lead to novel or unknown states. The MBIE-EB algorithm [4] embodies such an exploration strategy using exploration bonuses based on state visit

M. H. M. Winands et al. (Eds.): ACG 2017, LNCS 10664, pp. 165–175, 2017.
https://doi.org/10.1007/978-3-319-71649-7_14

counts. It satisfies a PAC-like theoretical guarantee and has proven to be effective in a simple Markov decision process (MDP) setting. However, the same approach is not directly applicable to the Atari domain, since games have a huge pixel-based state space, which greatly reduces the opportunities for the agent to visit identical states more than once. Recent studies on deep RL address this problem with such methods for state generalization as hashing or Bayesian models [1,3].

Despite such research efforts, the performance of deep RL algorithms on sparse reward games still falls short of expert human players. At least two elements can be improved in the existing approach: (1) how to compute exploration bonuses and (2) how to use them. Exploration bonuses in the previous work are calculated by separately using the information on each state. However, we argue that an agent should explore by considering the relative priority between states regarding which should be explored first. In this paper, we present novel exploration bonuses based on the Upper Confidence Bound (UCB) [5] algorithm, which was originally developed for the multi-armed bandit (MAB) problem. Our exploration bonuses are defined based on the relative priority between states instead of actions.

We address the second issue, i.e., how to use exploration bonuses, by presenting novel RL architecture in which the agent is trained with two separate policies—one for exploration and another for exploitation—using two different types of rewards. This step is motivated by the fact that an agent rarely receives rewards in games with sparse rewards, and thus mixing real rewards with exploration bonuses in a single policy is likely to produce suboptimal performance in the evaluation phase due to the dominant effect from the exploration bonuses.

To evaluate our proposed methods, we implement them by extending the Asynchronous Advantage Actor-critic (A3C) [6] algorithm, which is a representative deep RL method that has been successfully applied to Atari games and has outperformed Deep Q-networks (DQN) [2] except in hard exploration games. We carry out experiments using several Atari games, focusing on hard exploration games for which the vanilla A3C struggles to compete with humans. The experimental results show that our methods significantly improve the performance of hard exploration games, and achieve state-of-the-art scores on "Private Eye" and "Solaris".

2 Related Work

Mnih et al. [2] proposed a method called "Deep Q-Network" (DQN) that leverages deep neural networks for training the Q-function to play Atari games. One notable DQN property is that it directly trains the Q-function only from the actual game screens that human players basically observe without other additional information except the current scores, which we refer to "rewards" in RL literature.

To further improve the performance and training speed of DQN, Mnih et al. [6] proposed the A3C algorithm, which trains deep neural networks similar to DQN, but in multiple threads, and asynchronously updates the network

Algorithm 1. Training procedure of A3C for each actor-learner thread

// T: global shared counter that initialized by 0
// θ_π, θ_v: global shared parameters, θ'_π, θ'_v: thread specific parameters
// $t_{\text{interval}} \leftarrow$ interval for updating parameters, *i.e.*, 5
Initialization: $t \leftarrow 1$, and $s_t \leftarrow$ Initial state
repeat
  $d\theta_\pi \leftarrow 0$, $d\theta_v \leftarrow 0$, $\theta'_v \leftarrow \theta_v$ and $\theta'_\pi \leftarrow \theta_\pi$ // Initializations for each iteration
  $t_s \leftarrow t$
  repeat
    Perform action a_t according to policy $\pi(a_t|s_t;\theta'_\pi)$
    Receive reward r_t and next state s_{t+1}
    $t \leftarrow t+1$
    $T \leftarrow T+1$ // synchronous addition for all threads
  until s_t is terminal state **or** $t - t_s = t_{\text{interval}}$
  $R \leftarrow 0$ **If** s_t is terminal state, or $R \leftarrow v(s_t;\theta'_v)$ **otherwise**
  foreach $i \in \{t-1, \ldots, t_s\}$ **do**
    $R \leftarrow \gamma R + r_i$
    $d\theta_\pi \leftarrow d\theta_\pi + d\theta'_\pi$ obtained by Eq. (1) // accumulate gradients $d\theta_\pi$ with respect to θ'_π
    $d\theta_v \leftarrow d\theta_v + d\theta'_v$ obtained by Eq. (2) // accumulate gradients $d\theta_v$ with respect to θ'_v
  end for
  Update asynchronously θ_π using $d\theta_\pi$, and θ_v using $d\theta_v$
until $T \geq T_{\max}$

parameters for faster training. One appealing property of A3C is that it explicitly divides the Q-function (used in DQN) into policy and value functions, and separately updates the parameters. This separation basically leads to a richer behavior space. However, this separation also often leads to deficient exploration. To prevent this degradation, A3C generally incorporates an entropy regularization term into the objective function of the training since it encourages the agent to explore novel or unknown states during training.

Algorithm 1 summarizes the training procedure of A3C. Let R be the weighted sum of the observed reward in a certain interval. Let π and v represent the policy and the output of the value function, respectively. Similarly, let θ'_π and θ'_v be parameters for π and v, respectively. Moreover, $H(\theta'_\pi)$ denotes the entropy term, and β_{a3c} is a coefficient that controls the strength of the entropy term. Then A3C updates the network based on $d\theta'_\pi$ and $d\theta'_v$, which are calculated as follows:

$$d\theta'_\pi = \nabla_{\theta'_\pi} \log(\pi(a_i|s_i;\theta'_\pi))(R - v(s_i;\theta'_v)) + \beta_{\text{a3c}} \nabla_{\theta'_\pi} H(\theta'_\pi), \tag{1}$$

$$d\theta'_v = \frac{\partial (R - v(s_i;\theta'_v))^2}{\partial \theta'_v}. \tag{2}$$

The recent deep RL methods developed based on Atari games utilize raw pixels obtained directly from the game screen as states. This setting makes state space $\mathcal{S}$ extremely large. For example, in a typical setting a state is represented by a set of the game screens of four consecutive time steps. Each game screen consists of 84×84 pixels with 256 grades for each pixel in general [2]. Therefore, the naive state space becomes $256^{84\times84\times4}$. Consequently, the probability of obtaining identical states more than once is very small. This is the main reason why a naive count-based exploration method does not work well.

Some exploration methods in deep RL literature have recently been developed. For example, Tang et al. [3] proposed a count-based exploration method that utilizes a simple hash function. Hash function $\phi : \mathcal{S} \to \mathcal{Z}$ maps state space $\mathcal{S}$ to finite integer space $\mathcal{Z}$, where $\mathcal{Z} = \{1, \ldots, N\}$. For example, let $h \in \mathcal{Z}$ be an integer that is converted by hash function ϕ from state $s \in \mathcal{S}$: $h = \phi(s)$. Let $n(h)$ denote the occurrence of index h obtained from hash function ϕ. $n(h)$ is then used to compute a reward bonus based on the classic count-based exploration theory. For example, reward bonus r_{hash} was previously defined as follows [3]:

$$r_{\text{hash}} = \frac{\beta_{\text{hash}}}{\sqrt{n(h)}}, \tag{3}$$

where $\beta_{\text{hash}} \in \mathbb{R}_{\geq 0} = \{0, 1, 2, \ldots\}$ is the bonus coefficient. For every time step t, $n(h_t)$ is increased by one if the state obtained at t is indexed by h_t.

Bellemare et al. [1] proposed an exploration bonus using pseudo-counts. A state pseudo-count, derived from a current *recoding probability* ρ and $\rho'(s)$ that new state s occurs, is defined as

$$\tilde{n}(s) = \frac{\rho}{\rho'(s) - \rho}, \tag{4}$$

where $\tilde{n}(s)$ is the pseudo-count of state s. Reward bonus r_{psc} is defined as

$$r_{\text{psc}} = \frac{\beta_{\text{psc}}}{\sqrt{\tilde{n}(s) + 0.01}}, \tag{5}$$

where β_{psc} is the bonus coefficient. This method significantly improved the performance of the agent especially in Montezuma's revenge, which is one of the hardest Atari games for the current deep RL algorithms available in Arcade Learning Environment (ALE) [7].

To choose an action that considers the relative relationship between each action, Lai and Robbins [5] use the total number of trials in the MAB problem. They proposed upper confidence bounds (UCB) and an algorithm that chooses the action that maximizes the UCB score at time t. The UCB score is defined as

$$UCB = r(a_t) + \sqrt{\frac{2\log(t)}{n(a_t)}}, \tag{6}$$

where r is the estimated reward and $n(a_t)$ is the number of times action a_t has been chosen. The second term represents the value of the information about the action. The UCB algorithm is guaranteed to choose the best action with infinite trials.

3 Proposed Method

This section explains our proposed method. Our proposal is a methodology to effectively explore states with positive rewards, which is particularly suitable for

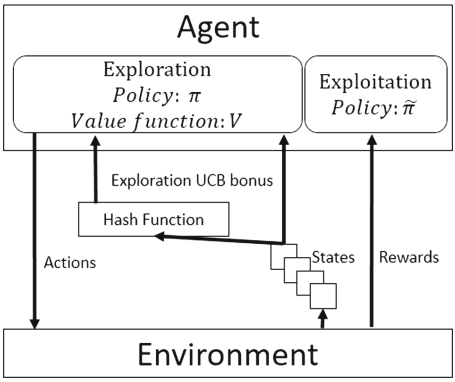

Fig. 1. Overview of proposed method

sparse reward games. Figure 1 illustrates a brief overview that summarizes the modules and their relations in our proposed method. First, A3C, as explained in the previous section, is our starting point. Basically, our proposed method enhances the A3C-based RL framework by incorporating a strong exploration strategy.

Our method has two major improvements from the baseline A3C. It incorporates a module for calculating exploration bonuses from state information (Sect. 3.1). Then, the calculated bonuses are utilized as pseudo rewards for updating the network parameters. It also incorporates two distinct policies, which are trained separately by true rewards and exploration bonuses (Sect. 3.2). These policies are then used in different purposes; the policy trained by exploration bonuses is used for exploring unseen states in the training phase, and the other is used for achieving states with positive rewards mainly in the evaluation phase.

3.1 Exploration UCB Bonus Using Hashing

We define our exploration bonus at t, namely, e_t, as follows:

$$e_t = \beta_{\mathrm{ucb}} \sqrt{\frac{\log(t)}{n(h_t)}}, \tag{7}$$

where β_{ucb} is a coefficient. Conceptually, $n(h_t)$ represents the counts of states indexed by h_t, which is the index of the state at time t: s_t. Initially, counts $n(h)$ for all h are set to zero. Then, for every time step t, $n(h_t)$ is increased by one. Thus, relation $n(h) \in \mathbb{R}_{\geq 0}$ holds for all h. Clearly, e_t is influenced by the UCB score shown in Eq. 6. We modified the original UCB score to fit the situation for evaluating which states we should select instead of actions. We refer to the exploration bonus defined by Eq. 7 as the exploration UCB bonus.

Intuitively, if the state indexed by h is unseen or seen with a small number of occurrences, where $n(h)$ is small, then e_t takes a relatively large value. In

Algorithm 2. Procedure for obtaining index h given state s

Input: State s
$key \leftarrow 0$
$\boldsymbol{b} \leftarrow g(s)$ // g: state pre-processor that maps state s into a D-dimensional vector
$\boldsymbol{z} \leftarrow \boldsymbol{A}\boldsymbol{b}$ // $\boldsymbol{A} \in [-1, 1]^{k \times D}$: a matrix for random projection
for $i = 1$ **to** k **do**
  $key \leftarrow key \times 2$
  if $z_i > 0$ **then** // $\boldsymbol{z} = (z_1, \ldots, z_k)$
    $key \leftarrow key + 1$
  end if
end for
$h \leftarrow key \,\%\, p_r$ // p_r: integer for determining the maximum index number
Output: h

contrast, e_t becomes relatively small if the state indexed by h appears many times, where $n(h)$ is large. Moreover, even if $n(h)$ takes a large number, e_t might take a relatively large value due to the effect of the ratio between the numerator and the denominator. This situation may occur if no state with index h was selected in the last certain time steps since the numerator monotonically increases along with time step t.

Next, we describe how our method converts each state s into corresponding index h for calculating $n(h)$. Algorithm 2 shows the procedure for obtaining h from s. This procedure basically utilizes the technique of locality-sensitive hashing (LSH) [8], which is essentially identical to a previously described idea [3]. We assume matrix $\mathbf{A} \in \mathbb{R}^{k \times D}$ with random initialization based on a continuous uniform distribution $[-1, 1]$. State s is converted into D-dimensional vector $\boldsymbol{b}$ by pre-defined conversion function $g(\cdot)$. More precisely, we define $g(\cdot)$ to convert the matrix form of the last input screen into a vector representation by concatenating every column in the matrix. We also concatenate the difference between the first and last input screens in the vector representation as explained above. Consequently, the total length of vector $\boldsymbol{b}$ becomes $D = 14,112 (= 84 \times 84 \times 2)$. Then vector $\boldsymbol{b}$ is (randomly) projected into a k-dimensional vector $\boldsymbol{z}$ by (randomly initialized) transformation matrix A. Finally, we obtain a binary code (hash key) from $\boldsymbol{z}$ by converting each element in $\boldsymbol{z}$ into 1 if the element takes a positive value, or 0 otherwise.

In addition, the value of k controls the granularity; since higher values reduce collisions, they are more likely to distinguish states. Ideally, the hash keys of the current and subsequent states should always be different, which makes a large k more desirable. However, the memory requirement increases in proportion to the value of k. To reduce the memory requirement, we introduce p_r, which represents the maximum number of indexes. We set $p_r = 999983$ and $k = 128$.

3.2 Training Two Types of Policies

This section explains the overall learning procedure of our proposed method. Algorithm 3 describes how to train the agent by our method, which basically trains it in the same fashion as our baseline method A3C. First, the agent chooses an action based on the current policy and receives a state, a reward, and an

Algorithm 3. Training procedure of A3C + UCB for each actor-learner thread

// T: global shared counter that initialized by 0
// $\theta_{\tilde{\pi}}$, θ_π, θ_v: global shared parameters, $\theta'_{\tilde{\pi}}$, θ'_π, θ'_v: thread specific parameters
// $t_{\text{interval}} \leftarrow$ interval for updating parameters, *i.e.*, 5
// $I_{\text{th}} \leftarrow$ # of threads for using $\tilde{\pi}$ instead of π, *i.e.*, 3 (basically selecting a small number)
Initialization: $t \leftarrow 1$, and $s_t \leftarrow$ Initial state
repeat
  $d\theta_{\tilde{\pi}} \leftarrow 0$, $d\theta_\pi \leftarrow 0$, $d\theta_v \leftarrow 0$, $\theta'_{\tilde{\pi}} \leftarrow \theta_{\tilde{\pi}}$, $\theta'_\pi \leftarrow \theta_\pi$ and $\theta'_v \leftarrow \theta_v$
  $t_s \leftarrow t$
  repeat
    Perform action a_t according to $\tilde{\pi}(a_t|s_t;\theta'_{\tilde{\pi}})$ **if** threadID $\leq I_{\text{th}}$, $\pi(a_t|s_t;\theta'_\pi)$ **otherwise**
    Receive reward r_t and next state s_{t+1}
    $t \leftarrow t+1$
    $T \leftarrow T+1$ // synchronous addition for all threads
  until s_t is terminal state **or** $t - t_s = t_{\text{interval}}$
  $R \leftarrow 0$ **if** s_t is terminal state, **or** $R \leftarrow v(s_t;\theta'_v)$ **otherwise**
  foreach $i \in \{t-1, \ldots, t_s\}$ **do**
    $R \leftarrow \gamma R + e_i$
    $R' \leftarrow \gamma R' + r_i$
    $\tilde{R} \leftarrow clip(R', -1, 1)$
    $d\theta_{\tilde{\pi}} \leftarrow d\theta_{\tilde{\pi}} + d\theta'_{\tilde{\pi}}$ obtained by Eq. (8) // accumulate gradients $d\theta_{\tilde{\pi}}$ with respect to $\theta'_{\tilde{\pi}}$
    $d\theta_\pi \leftarrow d\theta_\pi + d\theta'_\pi$ obtained by Eq. (2) // accumulate gradients $d\theta_\pi$ with respect to θ'_π
    $d\theta_v \leftarrow d\theta_v + d\theta'_v$ obtained by Eq. (1) // accumulate gradients $d\theta_v$ with respect to θ'_v
  end for
  Update asynchronously $\theta_{\tilde{\pi}}$ using $d\theta_{\tilde{\pi}}$, θ_π using $d\theta_\pi$, and θ_v using $d\theta_v$
until $T \geq T_{\max}$

exploration UCB bonus. We clip the rewards in the range $[-1, 1]$. A notable difference from A3C and other similar exploration methods, such as [1,3], is that we use two distinct policies, and train them independently either by true rewards or exploration bonuses.

In general, exploration bonuses are used as additions of rewards in the existing methods [1,3], namely $R = r + e$, where r represents the true reward, e represents the exploration bonus, and R is used for updating the agent's network instead of r alone. Their methods may cause two problems regarding the appropriate training of the policy and value functions. The first problem is that the agent may inappropriately explore in the training phase. This is because their methods tend to lead the agent to explore states with a positive reward that they found rather than a novel state. Their method therefore can result in a locally optimum strategy with which the agent receives positive rewards many times in the same way. The second problem is that trained policy may not choose actions that receive positive rewards in the evaluation phase. This is because the exploration bonus e in training may work as noise from the viewpoint of the evaluation. In fact, since the number of updates in training is limited, exploration bonuses take non-zero values, and so the noise influence cannot be ignored.

Based on the above arguments, we introduce two distinct policies for rewards and exploration bonuses. We call the first an *exploration policy* π, which is trained with exploration bonuses and is employed in the training phase. We call the second an *exploitation policy* $\tilde{\pi}$, which is trained with rewards from the environment and is employed mainly in the evaluation phase. Although these two policies are separately trained, they share the convolutional neural networks

to interact with each other for sharing important information on rewards and exploration bonuses.

Let $\theta'_{\tilde{\pi}}$ be a set of parameters for $\tilde{\pi}$. Then, similar to Eqs. 1 and 2, the agent's network is also updated based on $d\theta'_{\tilde{\pi}}$, which is calculated as follows:

$$d\theta'_{\tilde{\pi}} = \begin{cases} \nabla_{\theta'_{\tilde{\pi}}} \log(\tilde{\pi}(a_i|s_i;\theta'_{\tilde{\pi}}))\tilde{R} & \text{if } \tilde{R} > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Even though an agent receives a reward once, it is difficult to choose the same action again. To overcome this problem, if an agent obtains the highest score, it memorizes the past states and actions. This history is used for training data with probability ϵ. We call this the best score policy and set ϵ to 0.1.

4 Experiments

We conducted our experiments on the ALE, which provides a simulator for Atari 2600 games, and has been utilized as a standard benchmark environment to evaluate recent deep RL algorithms.

4.1 Setting

Our focus is games of hard exploration with sparse reward categorized by a previous work [1]. We investigated the effectiveness of our proposed method on the following six games: "Freeway," "Gravitar," "Montezuma revenge" (Montezuma), "Private eye" (Private), "Solaris," and "Venture" (see [1])[1]. We selected one additional game for evaluation, "Enduro" since it also requires the agent to have a strong exploration strategy since the performance of the baseline A3C was zero.

Unless otherwise noted, we basically followed the experimental settings used in A3C experiments [6]. For example, the network architecture is identical to previous research [9] and consists of two convolutional layers (16 filters of size 8×8 with stride 4, and 32 filters of size 4×4 with stride 2, respectively), and one fully connected layer (256 hidden units). Moreover, ReLU [10] was selected as activation functions for all hidden units. The final output layer has two types of outputs. One is a single linear output unit for evaluating the value function. The other is a softmax output unit for representing the probabilities for all actions by one entry per action.

Table 1 summarizes the training and evaluation configurations. In our experiments, we set a few settings differently from previous works [6]. First, we trained the agents for 200 million time steps to match the experimental conditions of the most related exploration technique [1] and evaluated their performance at every one million time steps. This means that we evaluated the performance 200 times during an entire training procedure. Then at each performance evaluation, the agents were evaluated 30 times with different initial random conditions in the "no-op performance measure" (see this work for an example [11]). Additionally, we trained the agents with 56 threads in parallel instead of 16 [6].

[1] We removed the evaluation of "Pitfall" since it ran abnormally in our environment.

Table 1. Summary of configurations used for training and evaluation

Training	Training steps	200 million steps (800 million frames)
	Threads	56
	Optimization algorithm	RMSProp with a decay factor of 0.99
	Discount factor γ	0.99
	Coefficient of entropy regularizer β_{a3c}	0.01
Evaluation	Evaluation intervals	Every 1 million steps
	Evaluations	30 episodes/per each

4.2 Results

Table 2 shows the results of our experiments. We also listed the results of the baseline and current top-line methods for comparison. For explanation convenience, we categorized the results into four categories (a), (b), (c), and (d). The rows of category (a) show the results of our experiments. The main purpose of these rows is for comparing our baseline method (A3C), a previously proposed exploration method (A3C + psc), and our proposed method (A3C + UCB) in fair conditions; all the results were evaluated by our implementation. The second category (b) shows previous results [1]. Note that we only picked results whose base algorithm was A3C as well as our proposed method for comparison with those obtained by our implementation. Category (c) shows the results of two recently developed exploration methods, psc and SimHash-based methods[2]. Finally, category (d) shows the results of the following current top-line deep RL

Table 2. Results of our proposed method and comparison with baseline and current top-line methods. Boldface numbers indicate best result on each game.

Cat.	method	Enduro	Freeway	Gravitar	Montezuma	Private	Solaris	Venture
(a)	A3C + UCB (Proposed)	28.5	23.0	406.7	126.7	**7643.5**	**4622.7**	163.3
	A3C (baseline: our impl.)	0.0	0.0	283.3	3.3	160.1	3287.3	0.0
	A3C + psc (our impl.)	181.7	22.1	311.7	0.0	1855.7	2612.0	0.0
(b)	A3C (baseline: reported [1])	0.0	0.0	201.3	0.2	97.4	2102.1	0.0
	A3C + psc [1]	694.8	30.4	238.7	273.7	99.3	2270.2	0.0
(c)	DDQN + psc (taken from [3])	–	29.2	–	**3439**	–	–	369
	TRPO + picel-SimHash [3]	–	31.6	468	0	–	2897	263
	TRPO + BASS-SimHash [3]	–	28.4	604	238	–	1201	616
	TRPO + AE-SimHash [3]	–	33.5	482	75	–	4467	445
(d)	DQN [2]	301.8	30.3	306.7	0.0	1788.0	–	380.0
	DDQN [11]	319.5	31.8	170.5	0	670.1	–	93.0
	Gorila [12]	114.9	11.7	**1054.5**	4.2	748.6	–	**1245.3**
	Bootstrapped DQN [13]	1591.0	**33.9**	286.1	100.0	1812.5	–	212.5
	Dueling network [14]	**2258.2**	0.0	588.0	0.0	–	2250.8	497.0

[2] Note that the detailed scores of DDQN+psc were not reported in the original paper [1]. However, we obtained them from a previous work [3].

algorithms (excluding A3C): DQN [2], double DQN (DDQN) [11], Gorila [12], Bootstrapped DQN [13], and Dueling network [14]. Note that these methods basically have no special exploration strategy.

4.3 Discussions

Note the following observations in Table 2:

1. The previous A3C results [1] and those of our implementation (our impl.) are very close. This implicitly supports that our implementation with our experimental setting works in a way that resembles previous studies.
2. A3C + UCB (Proposed) consistently outperformed the baseline A3C. This supports that our UCB-based exploration strategy helps discover new states with positive rewards that the baseline A3C can hardly reach.
3. A3C + UCB (Proposed) achieved average scores of **7643** and **4622** for "Private eye" and "Solaris", respectively. To the best of our knowledge, these are the best reported scores for the corresponding games.

Additionally, we investigated behavior of the agent trained with A3C + UCB on Private Eye to identify the essential advantage for achieving state-of-the-art performance. We found that Private Eye has a sort of trap that impedes finding better states. First, several states with small rewards, i.e., $r = 100$, are located near the starting point. Then even though a small number of states with much larger rewards, i.e., $r = 5000$, exist, they are located far from the starting point. To reach such states, the agent repeatedly encounters states with negative rewards, i.e., $r = -1$. Therefore, it is easy to imagine that agents tend to avoid such states with negative rewards and explore states with small rewards. Consequently, an agent will struggle to discover states with large positive rewards if it is not equipped with a strong exploration strategy. In contrast, the agent trained with our method, A3C + UCB, successfully conquered such obstacles and reached states with large rewards based on the power of our UCB-based exploration strategy. This provides strong evidence that our method works well even in hard exploration environments.

5 Conclusions

In this paper, we proposed an effective exploration strategy based on Upper Confidence Bounds (UCBs) that are suitable for recent deep RL algorithms. Conceptually, our exploration UCB bonus can be interpreted as a score estimated from a combination of the visit counts of a state and the degree of training progress. We also proposed a mechanism that effectively leverages exploration bonuses. Our method incorporates two types of policies, namely, exploration and exploitation, both of which are simultaneously trained. These policies force the agent to explore a novel state in the training phase and receive large rewards in the evaluation phase. As a result, the proposed method significantly improved the performance of A3C and other similar exploration methods. In addition, our agent achieved the highest score on "Private Eye" and "Solaris" in Atari games.

References

1. Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., Munos, R.: Unifying count-based exploration and intrinsic motivation. In: Advances in Neural Information Processing Systems, NIPS, vol. 29, pp. 1471–1479 (2016)
2. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
3. Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, X., Duan, Y., Schulman, J., De Turck, F., Abbeel, P.: # exploration: a study of count-based exploration for deep reinforcement learning. arXiv preprint arXiv:1611.04717 (2016)
4. Strehl, A.L., Littman, M.L.: An analysis of model-based interval estimation for Markov decision processes. J. Comput. Syst. Sci. **74**, 1309–1331 (2008)
5. Lai, T.L., Robbins, H.: Asymptotically efficient adaptive allocation rules. Adv. Appl. Math. **6**, 4–22 (1985)
6. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: Proceedings of the 33rd International Conference on Machine Learning, JMLR (2016)
7. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: an evaluation platform for general agents. J. Artif. Intell. Res. **47**, 253–279 (2013)
8. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 459–468. IEEE (2006)
9. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. In: NIPS Deep Learning Workshop, NIPS (2013)
10. Nair, V., Hinton, G.E.: Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning, Omnipress, pp. 807–814 (2010)
11. Van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double Q-learning. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence, AAAI, pp. 2094–2100 (2016)
12. Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., Silver, D.: Massively parallel methods for deep reinforcement learning. In: ICML Deep Learning Workshop (2015)
13. Osband, I., Blundell, C., Pritzel, A., Van Roy, B.: Deep exploration via bootstrapped DQN. In: Advances in Neural Information Processing Systems, NIPS 29, pp. 4026–4034 (2016)
14. Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., de Freitas, N.: Dueling network architectures for deep reinforcement learning. In: Proceedings of the 33rd International Conference on Machine Learning, JLMR, pp. 1995–2003 (2016)