Dynamic Adjustment of Exploration Rate for PPO Algorithm to Relief Rapid Decline of Episode Rewards

Shingchern D. You[1,*], Chao-Wei Ku[1,2], and Chien-Hung Liu[1]

[1]Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei, Taiwan
[2]Taiwan Semiconductor Manufactory Co., Taiwan
Email: scyou@ntut.edu.tw (S.D.Y.); weichaoku@gmail.com (C.W.K.); cliu@ntut.edu.tw (C.H.L.)
[*]Corresponding author
Manuscript received May 20, 2023, revised July 27, 2023; accepted August 30, 2023; published February 2, 2024.

***Abstract*—Proximal Policy Optimization (PPO) is a widely used algorithm in reinforcement learning. We observe that the agent may repeatedly select actions in a fixed sequence in some environments, leading to rapid decline of rewards. After that, the reward remains very low for a prolonged training period. Consequently, the training efficiency reduces. In this paper, we propose an approach to dynamically adjust the constant in the entropy term in the objective function of the PPO algorithm to encourage the agent to explore. Our experimental results show that the proposed algorithm is effective to relief this detrimental situation of rapid decline of episode rewards.**

***Keywords*—entropy, Proximal Policy Optimization (PPO), exploration rate, reinforcement learning**

I. Introduction

With its wide applications, machine learning techniques have received much attention. According to Wikipedia [1], traditionally there are three categories of machine learning algorithms, supervised learning, unsupervised learning, and reinforcement learning. For supervised learning paradigms, inputs and outputs (labels) of examples are given to the computer to learn the relationships between the inputs and outputs. Therefore, supervised learning is used for classification and regression problems. For unsupervised learning paradigms, input data do not have labels. Thus, the main applications of this type of learning are for clustering and dimensionality reduction. As to the reinforcement learning, a built-in agent interacts with the dynamic environment to perform a goal [2] based on the received rewards. There are many successful applications of reinforcement learning, such as playing video games [3], modeling human behavior in dynamic tasks [4], decision of offloading from smart phones to clouds [5].

In terms of the agent implementation, it can be a simple, table-based program, or a much more complicated program (or hardware), such as Convolutional Neural Networks (CNN), or Long Short-term Memory (LSTM) networks. To use a reinforcement learning algorithm, we need to choose a suitable agent. The agent, in most cases, is related to the environment it interacts with. For example, in the video-game environment, a CNN-based agent is widely used.

During exploring the environment, the agent receives rewards as feedback. In this case, the agent is expected to maximize the expected return (defined as cumulative future rewards) by choosing a sequence of appropriate actions. How to maximize the cumulated rewards is an optimization problem, where various types of algorithms have been developed to solve. One widely used type is the value-based algorithms, whereas another type is the policy-based algorithms. The former uses a value function to estimate the expected return for any chosen action, and picks the action with the maximum (expected) return. One well-known algorithm in this type is the Deep Q-Learning (DQN) algorithm, which has been shown to play video games with skill levels comparable to, if not exceeding, that of human experts [3]. Despite its success, the DQN algorithm "fails on many simple problems," such as those requiring continuous control [6].

The agent in a policy-based algorithm learns an optimal policy directly, which in turn provides the probability of each action. The chosen action is random, according to the associated probability, in nature. To learn the policy, the agent interacts with the environment, and collects a sequence of actions, states (i.e., instantaneous environment) and rewards, called a trajectory. By exploring the environment, the agent gradually learns a (sub-)optimal policy.

The original policy-based algorithm (REINFORCE) suffers from low training efficiency because a trajectory is used only once for training. To re-use the collected trajectories, the Proximal Policy Optimization (PPO) algorithm was developed. Due to its superior performance, this algorithm was chosen as a default learning algorithm in the OpenAI gym [7].

When applying the PPO algorithm to real-world problems, sometimes certain actions may not be useful, or even allowed, such as buying stocks without sufficient fund. To this end, Tan *et al.* [8] proposed the use of action masks to remove invalid actions from the action list. While this approach is effective, it nevertheless reduces the candidates in the action list. Consequently, it is more likely to observe "valleys with low rewards" during training, although the original PPO algorithm also has the same problem. This type of "valleys with low rewards" situation is an adverse effect, as it reduces the training efficiency. In this paper, we provide an in-depth study about this problem and propose an approach to solve this problem. We also conduct experiments to confirm the usability of the proposed algorithm.

This paper is organized as follows: Section II is the description of the PPO algorithm and the description of the problem. Section III is the proposed approach. Section IV describes the experiments and results. Finally, Section V is the conclusion.

doi: 10.18178/jacn.2024.12.1.288

II. PPO Algorithm and Reward Rapid Decline

A. PPO Algorithm

To be self-contained, we briefly describe the PPO algorithm. The description given here closely follows the paper by Tan *et al.* [8]. Initially, the agent randomly chooses actions to interact with the environment in order to collect trajectories

$$T = \{(s_t, a_t, y'_{a_t}(s_t), r_t): 1 \le t \le M\} \quad (1)$$

Each element in T is a 4-tuple, where s_t is the state at step (or time) t, a_t is the used action $1 \le a_t \le K$, $y'_{a_t}(s_t)$ is the probability of choosing action a_t, and r_t is the received reward. In terms of implementation, the agent usually is a variation of neural-network model, such as CNN or CNN plus LSTM. In this case, the state information (from environment) is the inputs to the network, and y'_k, $1 \le k \le K$, are the outputs from the network to represent the probability that action k is chosen. For example, if $y'_3 = 0.8$, then action 3 has 80 % of chance to be selected. If action 3 is indeed used to interact with the environment at step t, then $a_t = 3$.

Once we have the trajectory, we can compute the return R_t from $t = 1$ till M. To simplify the discussion, we let one trajectory contain exactly one episode. The return in each step can be computed as

$$R_t = G_t + v(s_t) \quad (2)$$

where G_t is an advantage function and $v(s_t)$ is an estimation of the state value function $V_\pi(s_t)$. The function $V_\pi(s_t)$ is the expected (average) return from state s_t to the end of the episode with a policy π. In the actual implementation, $v(s_t)$ is estimated by a neural network. Both the estimation network and the agent network can share the same network model except the output layers.

The N-step advantage function is computed as

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots + \gamma^N r_{t+N} - v(s_t) \quad (3)$$

where γ is a discount factor. We use $\gamma = 0.9$ in the experiments. If we further define

$$\delta_t = \big(r_t + \gamma\, v(s_{t+1}) - v(s_t)\big), \quad (4)$$

then G_t can be expressed as

$$G_t = \delta_t + \gamma\lambda\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots + (\gamma\lambda)^N\delta_{t+N} \quad (5)$$

where λ is a smoothing factor ($\lambda = 0.95$ in the experiments).

The training process adjusts the network parameter θ through a gradient search to maximize the following objective function

$$J(\theta) = J_1(\theta) + J_2(\theta) + J_3(\theta) \quad (6)$$

The first term $J_1(\theta)$ computes the policy-gradient term with a clipping. The policy-gradient term is computed as $p_t(\theta)G_t$, where G_t is given in Eq. (5) and

$$p_t(\theta) = \frac{y'_{a_t}(s_t)}{y_{a_t}(s_t)}. \quad (7)$$

In Eq. (7), $y_{a_t}(s_t)$ is the actual agent-network output at state s_t on action a_t when reusing a previous trajectory T, whereas $y'_{a_t}(s_t)$ is previously obtained value in T. When computing the gradients, $y'_{a_t}(s_t)$ is treated as a constant. To avoid excessively large $p_t(\theta)$, a clip function is used. Therefore,

$$J_1(\theta) = \min(p_t(\theta)G_t, \text{clip}(p_t(\theta), 1-\epsilon, 1+\epsilon)G_t) \quad (8)$$

where clip$(\cdot)$ is a clipping function and ϵ is its parameter. We use $\epsilon = 0.2$ in the experiments.

The $J_2(\theta)$ term is the mean-squared error of the advantage term, computed as

$$J_2(\theta) = -C_1(v(s_t) - R_t)^2 \quad (9)$$

The $J_3(\theta)$ term computes the entropy for all actions, as follows

$$J_3(\theta) = -C_2 \textstyle\sum_k y_k(s_t) \log\big(y_k(s_t)\big). \quad (10)$$

Higher value of $J_3(\theta)$ indicates that all actions have comparable probabilities. Therefore, if C_2 is large, this term has a higher weight to determine the gradient direction. Consequently, the agent is encouraged to explore.

In summary, a simplified version of the PPO algorithm is given as follows. This simplified version uses a stochastic gradient ascent search. It can be easily modified to perform mini-batch gradient ascent.

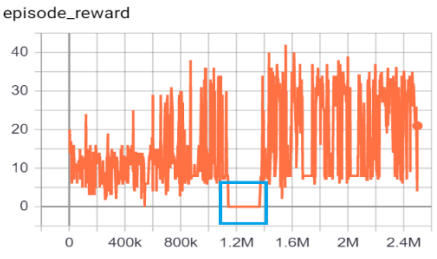
Algorithm for A simplified version of the PPO

For iter = 1 … ITER
  Collect a trajectory T of one episode with M steps
  Compute advantage G and return R

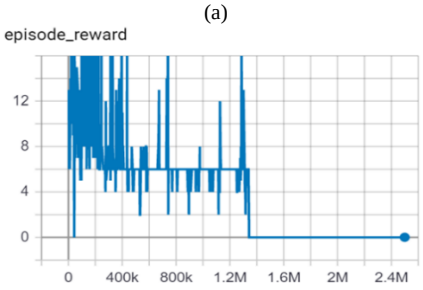
  // Perform stochastic gradient ascent
  For epoch = 1 … EPOCH
    For i = 1 … M
      Compute agent outputs y_k, $1 \le k \le K$
      Compute estimated output v
      Compute gradient of objective function J
      Update network parameter θ

B. The Problem of Rapid Decline of Reward

To illustrate this problem, we use the "mouse in a maze" [8] as an example. Fig. 1(a) shows the episode reward with respect to the training episode. It is clear that the reward has rapid decline around 1.2 M training steps, and falls into a "valley with low rewards" for a period of 200 k steps. In some cases, the agent may never leave this low-reward situation till the end of the training, such as the one shown in Fig. 1(b).



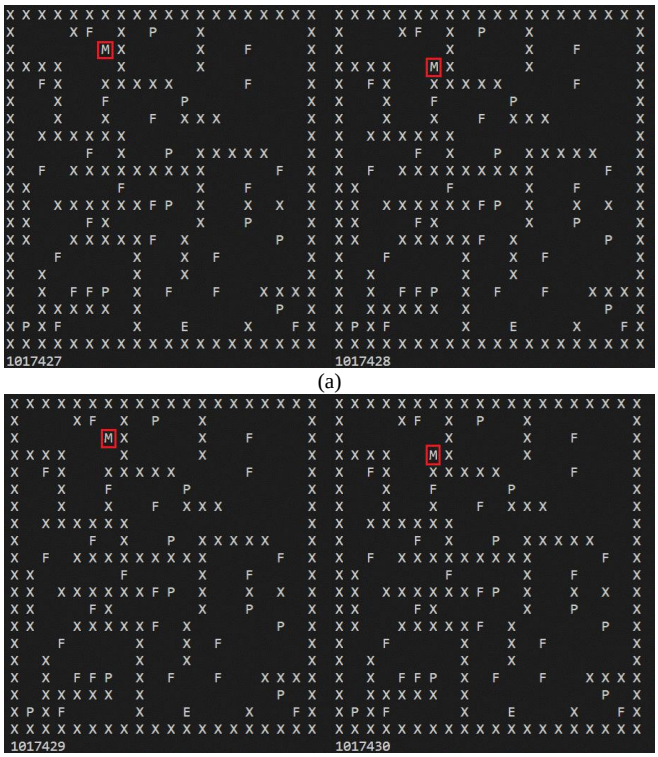
(a)



(b)

Fig. 1. Episode reward of the game "mouse-in-a-maze."

When closely examining the behavior of the agent in the low-episode-reward period, we find that the agent repeatedly moves upward by one step, downward by one step, and repeats these two steps many, many times, as shown in Fig. 2.



(a)

(b)

Fig. 2. Movement of the mouse from step index 1,017,427 to 1,017,430. The meanings of the characters are "X" for wall, "M" for mouse, "F" for food, "P" for poison, and "E" for exit.

We now consider the impact of a repeated action sequence on the objective function in Eq. (6). With a repeated sequence of actions, the first term (policy gradient) tends to go to zero because G_t is very small. The second term also tends to zero as the difference between $v(s_t)$ and R_t is very small. Therefore, the first two terms provide little or no gradient information for the optimizer to follow. The only useful term is the entropy term. From Eq. (10) we know that if the agent has comparable probability y_k on all actions, this term is large. Therefore, the gradient search tends to make relatively even probabilities among all actions. In a sense, this term determines the exploration probability of the agent. Therefore, if the constant C_2 is large, the agent is encouraged to explore more aggressively. However, in the reference implementation [7], the constant C_1 is set to 0.5, whereas C_2 is only 0.01. Therefore, when the agent is in the low-episode-reward period, a small C_2 makes this term insignificant in performing the gradient ascent. Table I shows some numerical values produced during the period of low episode rewards. It is clearly seen that the objective-function term is small, indicating that the agent does not have a clear gradient direction to follow. With this observation, one way to mitigate this problem is to increase the value of C_2 for more aggressive exploration. However, arbitrarily assigning a large value to C_2 is not a good strategy. If the agent already learns to solve a particular problem, it also closely follows its previous actions. In this case, further exploration is not desirable. In short, a better strategy is to dynamically adjust this constant based on some metrics.

III. Proposed Approach

From Fig. 1 we know that when the episode reward sharply drops, it remains low for a long period. Therefore, a simple and straightforward approach is to continuously monitor the episode reward, and then to increase the constant C_2 in case that the rapid decline of reward is detected.

The proposed approach uses a smoothed cumulative return Z_t as the basic function to detect the decline. Specifically, let

$$Z_t = (1-\alpha)Z_{t-1} + \alpha r_t \tag{11}$$

where r_t is the received reward, α is a constant, and $Z_0 = 0$. In the experiments, we set $\alpha = 0.001$. To determine if Z_t rapidly declines in the present state, we further define a local maximum of past Z_t as

$$Z_t^{Mx} = \max_{\tau \in [0, t-n_0 \times S_b]} (Z_\tau) \tag{12}$$

where n_0 is the number of mini-batches and S_b is the size of a mini-batch. The term $n_0 \times S_b$ is used to determine the ending point of the maximum operation. In the experiments, we use $S_b = 128$. The value n_0 does not significantly affect the results, and can be set to any reasonable value, such as 5. With the local maximum Z_t^{Mx}, we define

$$\tilde{C}_t = \max\left(0.01, \frac{0.1(Z_t^{Mx} - Z_t)}{(Z_t^{Mx} + 10^{-5})}\right) \tag{13}$$

as a substitute for C_2. If the present return Z_t is close to Z_t^{Mx}, the term $\frac{0.1(Z_t^{Mx} - Z_t)}{(Z_t^{Mx} + 10^{-5})}$ is close to 0. Therefore, $\tilde{C}_t$ is 0.01, which is the default value of C_2. Consequently, the proposed algorithm degenerates to the original version. On the other hand, if $Z_t \to 0$, then $\frac{0.1(Z_t^{Mx} - Z_t)}{(Z_t^{Mx} + 10^{-5})}$ is close to 0.1, and then $\tilde{C}_t \approx 0.1$. In this case, the agent is encouraged to explore more aggressively than the original version does. Finally, to avoid the situation that $Z_t^{Mx} \to 0$, we set an upper limit to $\tilde{C}_t$. Thus, the constant C_2 used in Eq. (10) is set as

$$C_2 = \min(k_0, \tilde{C}_t) \tag{14}$$

In the experiments, we set $k_0 = 0.1$. Please note that though not explicitly shown, C_2 is a function of training step t.

Table 1. Some numerical values Recorded during the period of low episode rewards

Step index	Policy gradient	Advantage network	Entropy	Objective function
1429888	−7.45E-09	5.45E-04	7.14E-01	−6.87E-03
1429920	1.83E-03	4.93E-04	7.09E-01	−5.01E-03
1429952	1.88E-03	4.56E-04	7.06E-01	−4.94E-03
1429984	2.74E-04	4.43E-04	7.04E-01	−6.54E-03
1430016	3.73E-08	2.05E-02	7.50E-01	2.73E-03
1430048	−3.62E-03	1.81E-02	7.43E-01	−1.98E-03
1430080	−6.81E-03	1.41E-02	7.34E-01	−7.10E-03

IV. Experiments and Results

This section covers the hardware specifications and software versions of experiments, the used learning environments (video games), and experimental results.

A. Hardware and Software Used in Experiments

The experiments are conducted on a personal computer

equipped with two graphic cards. The detailed specification of the computer is given in Table 2. The experiments are conducted based on the stable baselines 2 [9] with action mask [10]. The programs are run on a docker to have a higher flexibility and portability. The used software versions are given in Table 3.

Table 2. Experimental hardware

Component	Specification / Name
Processor	Intel(R) Core(TM) i9–9900X CPU @ 3.50 GHz
Mother board	Gigabyte X299 DESIGNARE EX-CF
Graphic card	Nvidia GeForce GTX 1080 Ti * 2
RAM size	32GB

Table 3. Experimental software

Component	Version
OS	Ubuntu 20.04
Python	3.6.9
Docker Engine Version	20.10.7
Docker Image	ntutselab/stable-baselines-gpu
NVIDIA Driver	460.80
Tensorflow	1.14
CUDA	11.2

B. *Learning Environments for Conducted Experiments*

To evaluate the performance and potential side effects of the proposed approach, we use the following video games as the learning environments for the agent to learn.

1) Mouse in a maze. This is a self-developed game already used in [6]. If the mouse eats a fruit, the reward is 2. If it eats a poison, the reward is −1 and the episode ends. If the mouse reaches the exit, the reward is 1 and the episode also ends. Any action that leads to hit a wall is ignored (no penalty) in the original version and is an invalid action in the proposed algorithm. Again, the maximum number of steps in an episode is also 1,000 steps. Fig. 2, given previously, shows the snapshots of the game.
2) Empty room. This game is a simplified version of the mouse in a maze. The only reward given to the agent is when the agent reaches the exit. As the reward is sparse, this experiment is a good test for the proposed approach. A snapshot of this environment is given in Fig. 3.

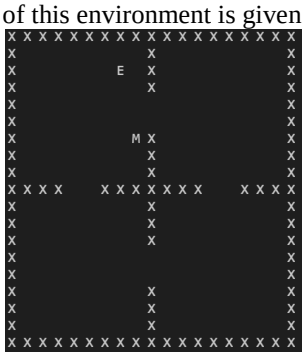


Fig. 3. Empty room.

3) Atari games. We also conduct experiments on several Atari games. These games do not suffer from the rapid decline of episode rewards. We choose these games to check if the proposed approach has any downsides for normal, reward-rich tasks. According to Bellemare [11], there are four different categories of games in Atari game set: Human-optimal, Score exploit, Dense reward, and Sparse reward. In the experiments, one game is chosen form each of the categories. In addition, the first two are easy exploration, and the last two are hard exploration. The chosen games are: Breakout, Seaquest, MsPacman, and Freeway. The snapshots of the games are given in Fig. 4 to Fig. 7.

Fig. 4. Breakout.

Fig. 5. Seaquest.

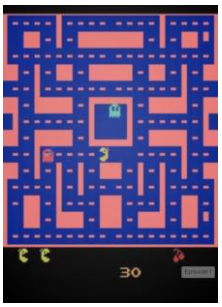


Fig. 6. MsPacman

Fig. 7. Freeway

The chosen agent for the mouse in a maze and Empty room is the MlpPolicy provided by the stable baseline. The agent for Atari games is the CNNPolicy with the hyperparameters given in RL-Baselines-Zoo [12].

C. *Experimental Results for Mouse in a Maze*

We use this experiment to observe whether the proposed approach can dynamically adjust the coefficient C_2, and also

observe the change of the episode reward with respect to training steps.

Fig. 8 and Fig. 9 show two different trials during training. There is no obvious rapid decline of reward in Fig. 8 (a). Therefore, the variation of C_2 over training steps fluctuates between 0.01 to greater than 0.05 in Fig. 8 (b). On another trial, as shown in Fig. 9 (a), the reward exhibits a small low-episode-reward period (highlighted with a red box). However, this period is much smaller when compared with that in Fig. 1(a). Fig. 9 (b) shows the change of C_2 over time, where the maximum value of C_2 reaches almost 0.1. With these observations, the proposed approach behaves as expected.

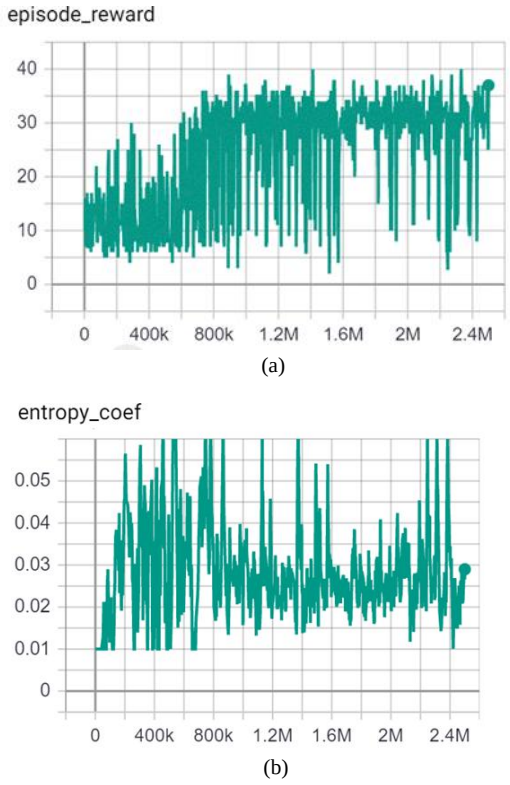


Fig. 8. Coefficient C_2 with respect to training steps.

To have quantitative comparison, we repeat the training 50 times (i.e., 50 trials) and present the results in Table IV. The first parameter, average episode reward calculates the average reward per episode over the 50 trials. This parameter, in a sense, measures the efficiency of the training, as a higher value indicates a more efficient training (learning). The second parameter picks the maximum reward in each trial, and takes the average. This parameter is another useful parameter to measure how well the agent can behave during the learning process. The third parameter counts the average number of episodes in the low-episode-reward periods, and the final one counts the number of trials with observable low-reward instances. It is clear from Table IV that the proposed approach is better, especially in the first parameter "average episode reward." In addition, the small number of episodes in low-episode-reward periods also signifies that the proposed algorithm works as expected.

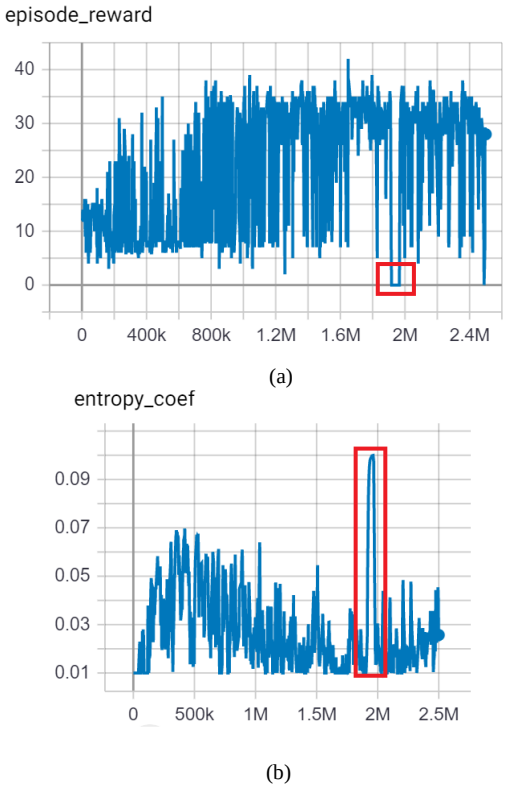


Fig. 9. Episode reward with respect to training steps.

Table 4. Experimental results for mouse in a maze

Parameter	Original	Proposed
Average Episode Reward	10.70	17.48
Avg of Max Reward in 50 Trials	33.06	40.42
Avg Low Reward Episodes per Trial	234.38	11.95
Trials with Low Rewards	27/50	2/50

D. *Experimental Results for Empty Room*

We use the "empty room" learning environment to test if the agent can constantly find a path to the exit. It is quite often to observe that the agent occasionally finds a path to exit, but forgets the learnt knowledge in subsequent learning steps, such as the one shown in Fig. 10. It clearly shows that the agent only finds a path to exit with two occasions.

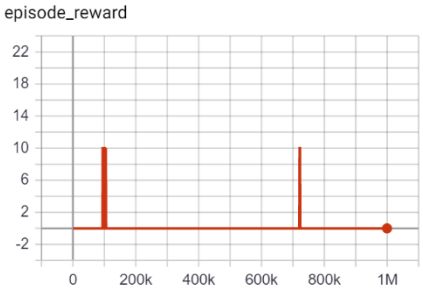


Fig. 10. Episode reward with respect to training steps.

When applying the proposed approach to this problem, the reward plot is shown in Fig. 11, where the low-reward period is still pretty long, but the coefficient C_2 increases to its maximum for higher exploration rates. In this experiment, although the agent cannot retain the learnt knowledge, the proposed approach meets its original goal: to encourage the agent to explore more when encountered low-reward situations.

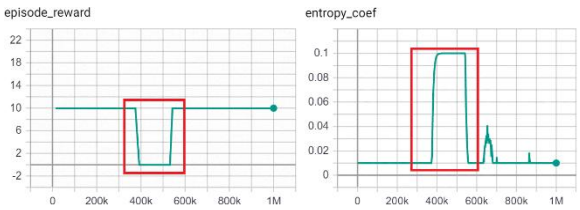


Fig. 11. Episode reward with respect to training steps.

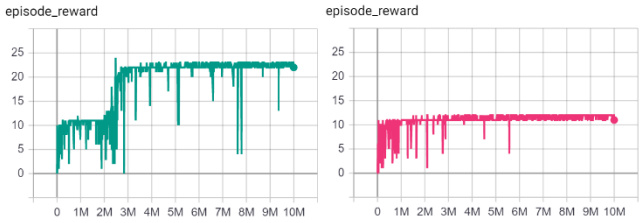


Fig. 12. Episode reward with respect to training steps in Seaquest.

We follow the discussion in Section IV.C and report the quantitative results in Table V. It is shown that the proposed approach also significantly improves the training performance. Note that this experiment only repeats 10 trials. Therefore, the value of "Trials with Low Reward" is not given.

Table 5. Experimental results for empty room

Parameter	Original	Proposed
Average Episode Reward	4.22	8.55
Max Reward	10	10
Avg Low Reward Episodes	775.63	467.37

E. Experimental Results for Atari Games

The Atari games have rich backgrounds, and different screenshots are considered as different states. Therefore, the agent is unlikely to revisit the same states many times in these video games. Consequently, the agent rarely exhibits rapid reward decline. The main purpose of this experiment is to observe any undesirable side effects. In this experiment, to save space, we thus only provide the results of average episode reward in Table VI. It is observed that the proposed approach does not negatively affect the performance in three games, and slightly reduce the training performance in the Seaquest game.

Table 6. Average episode reward in Atari games

Game	Original	Proposed
Breakout	9.59	9.14
Seaquest	16.56	13.70
MsPacman	28.42	27.09
Freeway	28.20	28.85

To understand the problem of the proposed approach performs in Seaquest, we repeat the experiment several times. We observe that this game can lead to two distinct episode reward plots when using the original PPO algorithm: one has a terminal reward approaching 23, whereas the other one is close to 10, as shown in Fig. 12. For both situations, we check the C_2 values in our approach. We find that our approach does not change C_2 at most of the time. Therefore, the lower average episode reward is mainly due to different initial seeds used in the PPO algorithm for generating random numbers. Unfortunately, the seeds are implicitly implemented, and the user is unable to control the seeds. Therefore, we are unable to report results with the exactly the same simulation conditions. Still, with our observations, we conclude that the proposed approach does not have any obvious side effects.

F. Limitations and Future Work

In our experiments, we showed that the proposed approach can reduce the detrimental low-reward effects appeared during training. Nevertheless, increasing exploration rate, in a sense, is equivalent to reducing exploitation. Consequently, the learnt knowledge is not applied as frequently as the original algorithm although the proposed algorithm is designed to minimize this situation when the rewards are not declined rapidly.

In the proposed algorithm, there are some hyperparameters to be determined by the user, such as α in Eq. (11) and k_0 in Eq. (14). We plan to investigate the impact of these hyperparameters on the performance of the proposed approach in the future.

Finally, although the proposed approach has been tested on various environments, the number of tested environments is still very limited. Further experiments and the corresponding analysis are necessary to confirm the generalizability of this approach to other environments. As the experiments are time consuming, this part is also our future work.

V. Conclusion

We propose an approach to dynamically control the exploration rate in the PPO algorithm. Doing so prevents the agent from entering a prolonged low-reward, low-efficient training period. The proposed approach is tested on several training environments. The results show that the proposed approach can improve the training efficiency for simple and reward-sparse environments. For other environments, the proposed approach reduces to the original version. Overall, the proposed approach can directly replace the original PPO algorithm without any noticeably adverse effects.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

SDY initiated the problem, directed the progress of the research, and wrote the paper; CWK conducted experiments; CHL co-investigated the research and proofread the paper; all authors had approved the final version.

References

[1] Machine_learning. [Online]. Available: https://en.wikipedia.org/wiki/Machine_learning
[2] R. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd Ed., Cambridge, MA: MIT Press, 2018.
[3] V. Mnih, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.
[4] V. Dutt, "Explaining human behavior in dynamic tasks through reinforcement learning," *Journal of Advances in Information Technology*, vol. 2, no. 3, pp. 177-188, August, 2011.

[5] N. Muslim, S. Islam, and J. C Grégoire, "Reinforcement learning based offloading framework for computation service in the edge cloud and core cloud," *Journal of Advances in Information Technology*, vol. 13, no. 2, pp. 139–146, April 2022
[6] J. Schulman, *et al*., "Proximal policy optimization algorithms," arXiv preprint arXiv:1707.06347, 2017.
[7] OpenAI Ltd. Gym toolkit software [Online]. Available: https://gym.openai.com/
[8] C. Y. Tang, C. H. Liu, W. K. Chen, and S. D. You, "Implementing action mask in proximal policy optimization (PPO) algorithm," *ICT Express*, vol. 6, no. 3, pp. 200–203, 2020.
[9] Stable Baselines. [Online]. Available: https://github.com/hill-a/stable-baselines
[10] Stable Baselines ActionMask. [Online]. Available: https://github.com/NTUT-SELab/stable-baselines/tree/ActionMask
[11] Bellemare, *et al*., "Count-based exploration with neural density models," in *Proc. of the 34th International Conference on Machine Learning*, 2017.
[12] RL Baselines Zoo: a Collection of Pre-Trained Reinforcement Learning Agents. [Online]. Available: https://github.com/araffin/rl-baselines-zoo