Merging Reinforcement Learning and Inverse Reinforcement Learning via Auxiliary Reward System

1st Wadhah Zeyad Tareq
Computer Engineering
Yıldız Technical University
Istanbul, Turkey
wadhah.zeyad.t.tareq@std.yildiz.edu.tr

2nd Mehmet Fatih Amasyali
Computer Engineering
Yıldız Technical University
Istanbul, Turkey
amasyali@yildiz.edu.tr

***Abstract*—In recent years, learning from demonstration has become one of the promising methods in robotics and interactive systems. Learning from demonstration is a model by which an agent learns by observing an expert. The expert could be a pre-trained agent or human. The main problem with learning from demonstrations is the difference between the reward representation in the demonstrations and the actual environment. During the construction of the demonstrations, it is easy to add new rewards to enhancement the agent's performance. In contrast, it is not easy to do that in an actual environment. This work is built upon our previous work to solve this problem. In previous work, the agent uses Reinforcement Learning algorithms to learn how to play video games from demonstrations. The agent was supplied with an external reward to solve the problem of missing rewards in the hard exploration environments. In this work, Inverse Reinforcement Learning uses to extract the external rewards from the demonstration and make them available during the interaction period. The results showed that inverse learning enables the agent to interact with the environment after the pre-training. Furthermore, the performance of the agent becomes more stable.**

***Index Terms*—deep reinforcement learning, inverse reinforcement learning, prioritized double deep q-networks, atari games**

I. INTRODUCTION

Deep reinforcement learning has succeeded in many sequential decision-making problems such as Atari games and robotic control. One of the challenges to applying reinforcement learning is the missing of environment reward which leads to makes the exploration difficult [1]. The reward is the only criterion that evaluates the efficiency of the reinforcement learning agent. Designing a reward function to assess the agent behavior depending on the environment is complex and impossible. One successful solution to learning the reward function is inverse reinforcement learning (IRL) [2].

IRL solves the reward engineering problem by obtaining the reward function from the expert's demonstrations [3]. An expert can be a professional human player or an agent who has been previously trained using one of the learning algorithms. IRL agents aim is to find out the reward function that explains the behavior of the expert. Once the reward function is found, the agent starts using the standard reinforcement learning methods to find out the optimal policy expectedly to behave as well as the expert. Reward function enables the agent to interact with the environment and improve its behavior without extra demonstrations [4].

In traditional Imitation Learning (IL), the agent is used to directly learn the expert's behavior. IRL enables the agent to learn the strategies which lead to that behavior. Understanding the strategy increases the robustness of the new behavior and allows the agent to handle new challenges such as new initial states or any change in the environment. Another advance of IRL is the ability of the agent to explore the environment via online learning using the standard RL algorithms. The exploration enables the agent to visit new states without the need for further demonstrations. Being robust and adapting to the change is essential to an agent running in a dynamic world [5]. Unlike deep RL, few IRL algorithms have been developed to play video games. In this paper, we build on our previous work [6]. The earlier work represented the first phase, and the current work represented the second phase. In phase one, the agent is trained to learn from the demonstrations without interacting with the environment. In phase two, the agent uses the IRL principles to interact with the environment and enhance its performance.

II. RELATED WORK

There are two parts of related work. The first part is about the deep RL in video games, especially Atari games. The second is about the IRL works in different areas.

The first most popular algorithm implemented Deep RL in Atari games is the deep Q-network algorithm (DQN) [7]. DQN merged the Q-learning [8] with the convolutional neural network (CNN) and tested it on many Atari games. The result showed that the DQN agent was able to reach the expert performance on many games. After that, the RL community has made many changes and extensions to the DQN algorithm to improve its performance and stability. Using the Double Q-learning algorithm [9] rather than the Q-learning algorithm was the first change to the DQN. The Double DQN (DDQN)

[10] showed that the DQN suffers from significant overestimation, and the Double Q-learning reduces that overestimation which leads to better performance. Prioritized experience replay (PER) [11] replaced random sampling with sampling experience by probability. This change allowed the DQN to sample experiences that have more information to learn than the other.

The dueling network [12] suggests decoupling the Q-learning into two estimators: state and action. The state estimator tells us it is (or is not) a valuable state regardless of the effect of the actions at that state. A3C [13] presented asynchronous four standard reinforcement learning algorithms and showed that parallel actor-learners stabilize training, allowing all four methods to train neural network controllers successfully. Distributional Q-learning [14] learns a categorical distribution of discounted returns instead of estimating the mean. Noisy DQN [15] uses stochastic network layers for exploration. Finally, Rainbow [16] studied the efficiency of the six previous extensions in one algorithm. Their algorithm provided an agent with performance better than each extension separately.

Deep Q-learning from demonstrations (DQfD) [17] was the first algorithm used demonstrations in Atari games. The DQfD included two learning phases. In phase one, the agent learning by imaging the demonstrations. In phase two, the agent is learning by interacting with the environment. The self-generated data and the demonstrations are used in the learning during phase two. The DQfD was the first work to enable RL agents to score in environments where the reward is rare or missed. Later, many works [18] – [20] attempt to use demonstrations to enhancement the performance of RL in such environments.

The second part of the related works is about the IRL. The IRL first appeared to solve the problem of the new state in the Imitation Learning (IL) approaches. In IL, when the agent visited a new state, it became impossible to return to the demonstrated states. IRL was first introduced by Ng, and Russell [2]. They proposed to solve the reward engineering problem by inferring the reward function from the expert demonstrations. The main challenge was the ambiguity. In the same demonstration, many reward functions could produce the expert policy. In [21], they developed a method that produces a policy that gets the same reward as the expert policy rewards. The later works are Bayesian IRL and Maximum Entropy IRL. Bayesian IRL [22] adopted the ambiguity by calculating the distributed rewards rather than focusing on the given function. This method produces the same guarantee as in [21]. On the other hand, Maximum Entropy IRL [23] builds a reward function that meets the expert features. This function guarantees the higher-entropy stochastic policy and allows the generalization in the environments using many different planning dynamics.

As mentioned before, few IRL algorithms were developed to play video games. The work by Uchibe [24] used logistic regression to classify the transitions into two groups: expert and non-expert. The classifier is used instead of the reward function to train standard deep RL algorithms. The results showed that their performance rarely outperforms the IL. The other work is CNN-AIRL [25]. They modified the Adversarial IRL algorithm [3]. Their modification includes adding CNN to the AIRL baseline, normalizing the environment reward, and increasing the size of the discriminator dataset. Additionally, they represented the state with low-dimensional by autoencoder architecture built especially for video games. Their algorithm achieves good performance on the simple Catcher video game. They applied their algorithm to the Enduro game, and the algorithm performance was lower than the expert performance.

III. Background

A. Markov Decision Processes

RL researchers adopt the Markov Decision Process (MDP) formalism for their works. MDP framework is a tuple (S, A, T, γ, R) where S is a set of states. Each state represents the status of the environment at that time. A is a set of actions. Actions are something that an agent can do in the state. T is a transition probability from one state to another. γ is a discount factor with a value between 0 and 1. The discount factor controls the dependability of the far future rewards. If $\gamma = 0$, the agent will only learn from the immediate reward. R is a signal that represents the reward of the agent. The reward evaluates the agent's action. For each state, there is a policy π determining which action the agent must select. The agent's primary goal is to find the policy that enables it to choose the optimal action, which increases the cumulative reward. The Q value for an (s, a) pair estimates the expected future reward when the agent follows that policy. The Q^{π} represents the Q value with that policy. The optimal Q value Q^* (s, a), which achieve maximal reward in one episode, is determined by solving the Bellman equation [17] [22]:

$$Q^*(s,a) = E\left[R + \gamma \sum_{s_{t+1}} P(s_{t+1}|s,a) max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\right] \tag{1}$$

From the above equation, the optimal policy can be found by:

$$\pi(a) = argmax_{a \in A} Q^*(s,a) \tag{2}$$

DQN [7] approximates the Q value for all available actions in a state by using a deep neural network. The network's input is the stack of several states (to determine the direction and speed of the moving objects in the game), and the output is the Q value for each action. DQN used a separate network to calculate the target values. This network is updated after a specific number of steps by copying the weights of the regular network. The aim here is to stability the Q target values. DDQN [9] used the regular network to select the best action and the target network to calculate the target Q value for that action. The DDQN loss is [17]:

$$J_{DDQN} = (R(s,a) + \gamma Q(s_{t+1}, a_{t+1}^{max}; \theta^-) - Q(s,a;\theta))^2 \tag{3}$$

Where θ is the weights of the regular network, the θ^{-} is the weights of the target network, and $a_{t+1}^{max} = argmax_a Q(s_{t+1}, a; \theta)$.

PER [11] set a priority for each sample. These priorities define which sample must be selected first. In the classical DQN and DDQN, the samples were selected randomly. Due to the randomness, some important samples may be deleted before being selected. Also, the PER is sampling the necessary samples more frequently. The probability of sampling a particular transition *i* is proportional to its priority:

$$P_i = \frac{p_i^{\alpha}}{\sum_k p_k^{\alpha}} \tag{4}$$

Where p_i is the last temporal difference error (the difference between the new and old prediction) calculated for the i sample plus a small positive constant. This small value ensures that all samples are selected with some probability. The α is a hyper-parameter used to reintroduce some randomness in the experience selection for the replay buffer. When the α is equal to 0, all samples are selected randomly. When the α is equal to 1, only necessary samples with the highest probability are selected.

B. Deep Q-learning from demonstrations

The general structure of our work is like the DQfD structure [17]. DQfD contains two phases: the pre-training phase and the interacting phase. The pre-training phase aims to learn as much as possible before start interacting with the environment. In this phase, the agent imitates the demonstrator to satisfy the Bellman equation. During this phase, the agent updates the network by applying four losses: the 1-step double Q-learning loss, an n-step double Q-learning loss, a supervised large margin classification loss, and an L2 regularization loss on the network weights and biases. In the interacting phase, the agent acts on the environment by selecting action with its learned policy. In each time step, the agent saves current states, action, reward, and next state as self-generated data. Once the memory is full, the agent pulls out the oldest self-generated data and keeps the demonstration data without any change. The agent updates its network with a mixed mini-batch of demonstration and self-generated data. DQfD used PER [11] to control the ratio between demonstration and self-generated data while learning to improve the algorithm's performance.

C. Inverse Reinforcement Learning

The main difference in IRL is the expert's demonstrations. In IRL, the demonstration sampled from the MDP without the reward. These samples represent the expert policy π_E or the behavior that the expert follows to solve a particular problem. IRL aims to find a reward function that explains the expert behavior. IRL algorithm receives as inputs a set of the expert sampled transitions $D_E = (s, a, s^{'})$ and if available, a set of non-experts sampled transitions $D_{NE} = (s, a, s^{'})$. The goal of an IRL algorithm is to compute the reward R [4]:

$$\forall s \in S, \qquad argmax_{a \in A} Q_R^{*}(s, a) = Supp(\pi_E(.|s)) \tag{5}$$

Where Q_R^{*} represents the optimal score function, and π_E represents the expert policy. It is important to separate the expert sampled transitions from the non-experts sampled transitions to prevent the last one from being optimal, which is essential to imitate the expert behavior. To achieve that, one must search for a significant reward, which is a reward for which the expert policy is optimal and for which only expert actions are optimal or at least a subset of expert actions [4]:

$$\forall s \in S, \qquad argmax_{a \in A} Q_R^{*}(s, a) \subset Supp(\pi_E(.|s)) \tag{6}$$

The IRL researches, thus, included methods to find significant rewards [21], [23]. Once the reward is determined, the $MDP = (S, A, T, \gamma, R)$ must be solved to compute the agent's policy, which is a problem as such.

IV. Learning from Auxiliary Rewards Method

Our agent relies on the extra rewards for learning through two stages. The first stage is known as phase one or the pre-training phase. The second stage is known as phase two or the interacting phase. In the following two subsections, the details of these two phases will be explained.

A. The Pre-Training Phase

Our work for this phase was published previously [6]. In this section, phase one is briefly described. For further details about the implementation, evaluation, and comparisons, the readers can check out our previous article. In this phase, the agent learns to play video games using human demonstrations. Two classical RL algorithms were used DDQN [9] and PER [11]. Our contribution is to use auxiliary rewards rather than environment rewards. The auxiliary rewards were involved within the demonstrations, and the agent learned to play video games depending on these demonstrations only. There is no interacting or self-generated data during this phase.

The auxiliary reward is represented by 0 or 1. A human player selects the correct action in each step and sets that action's reward to 1. The reward of the rest actions is 0. Assigning actions with 0 is essential to give all actions realistic values. These absolute values prevent the network from updating toward ungrounded variables. Five Atari games were used to benchmark the performance of the agent. Three of these games are considered hard exploration games (Environment rewards are miss or rare), and two are simple games. Our agent results in the hard exploration games exceeded the results of many baseline algorithms including DQfD. Fig. 1 shows the agent results in all five games [6].

B. The Interacting Phase

Once the pre-training phase is complete, the agent starts acting on the environment by selecting action using its learned policy, collecting self-generated data, and learning from that data. The objectives of phase two are to explore new states and make the agent performance stable. The agent performance in some games is unstable. However, once the agent started interacting with the environment, the learned policy performance decreased. The reason here is the reward distribution. In phase

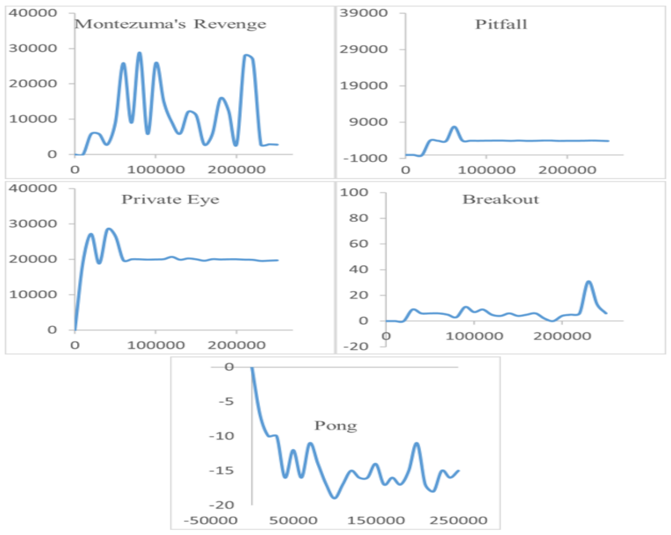

Fig. 1. Phase one scores averaged over ten episodes for each weight in 25 different weights starting from 10000 to 250000 steps of training for the five games: Montezuma's Revenge, Pitfall, Private Eye, Breakout, and Pong.

one, the agent received a 0 or 1 reward for each action. On the other hand, phase two depends on environment rewards. These rewards have different time intervals variations from game to game. Fig. 2 shows the performance of the agent in Montezuma's Revenge game after starting interacting.

Fig. 2 proves that the agent performance gets effect by the newly generated data. The agent runs on the environment for 50000 steps which is enough to show the problem. To solve this problem, the concept of IRL is used to extract new auxiliary rewards that help the agent during phase two. All details will explain in the next section.

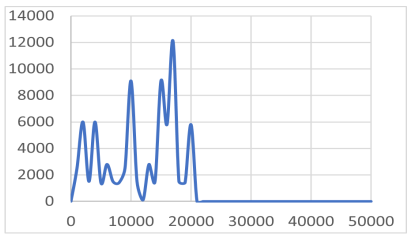

Fig. 2. The performance of our agent in phase two for Montezuma's Revenge game. The agent starts learning from self-generated data after performing 20,000 steps.

V. Learning Using IRL

IRL's main concept is learning the reward function from the demonstration data. A new CNN is built to serve as a Reward Function (RF) to take advantage of that. The RF dataset is a particular transition from the phase one demonstration. These transitions are the transitions with rewards equal to 1. The current state in that transition is considered as an input to the RF. A zeros array with a size equal to action space size is a target output for the RF. Only the correct action in that transition is assigned with 1 in the target output. The processor of building the RF dataset is shown in Fig. 3. The S_t refer to the current state, the S_{t+1} refer to the next state, (a) and (r) represent the selected action and the environment reward respectively. The RF has the same architecture as the agent regular CNN architecture from phase one. The main differences are the output activation function and the loss function. In the agent's CNN, a linear activation function is used. In the reward function, the SoftMax activation function is used. For the loss function, the Categorical cross-entropy loss is used rather than Huber loss.

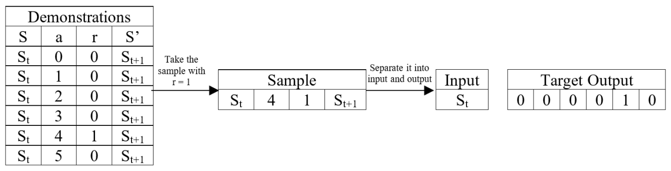

Fig. 3. Reward function dataset building procedure.

Fig. 4 shows the differences between the standard MDP used in classical RL algorithms and our new framework. In the new framework, the agent will select the action, and the reward function will provide a reward for the chosen action. This reward will be used alongside the environment reward. Fig. 5 compares our architecture with the DQfD architecture for phases one and two.

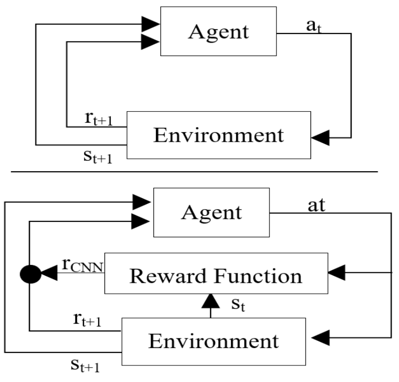

Fig. 4. The above plots show the standard MDP for solving RL problems. The down plots show our new framework by adding the RF.

VI. Results and Discussions

Our integrated approach has only been tested on two games: Montezuma's Revenge and Breakout. Montezuma's Revenge is a challenging exploration game, while Breakout is a simple game with routinely environmental feedback. The RF part has been tested on all five games from phase one. The reason for testing the integrated approach on two games only is the training time that is required.

A. Reward Function Results

This part discusses the results of RF where this part contains the training and testing results for the RF CNN. The reward function trained with the dataset extracted from the

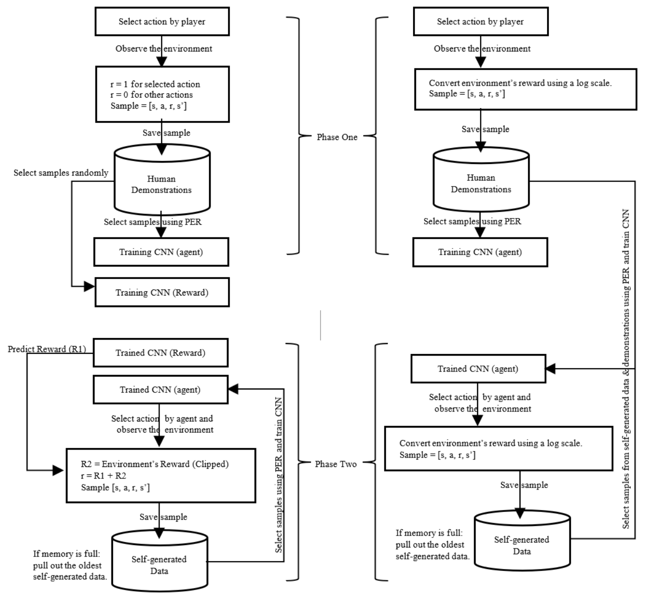

Fig. 5. The left plots show our architecture. The right plots show the DQfD architecture.

demonstration for 20.000 update steps. After that, the RF was tested on a new test set. This test set differs from the training set. The test set builds by playing each game for one episode. The episode starts from the initial state and finishes when all available lives become equal to zero. In general, one episode is enough to visit most states in the training test. These test sets find out if the reward function suffers from overfitting or underfitting. The details of the training dataset, test dataset, and test accuracy are shown in Tab. I. Furthermore, the RF learning F1 scores for each action are shown in Tab. II.

TABLE I
DATASET DETAILS AND TEST ACCURACY

Game	Training Dataset		Test Dataset		Test Accuracy
	Size	Episode	Size	Episode	
MR	60877	5	9705	1	0.8646
Pitfall	132844	5	25135	1	0.9148
Private Eye	39098	5	8282	1	0.8443
Breakout	31297	9	3282	1	0.6453
Pong	26783	3	6245	1	0.7969

TABLE II
F1 SCORE RESULTS

Accuracy	0	1	2	3	4	5	6	7	Weighted
MR	0.86	0.63	0.91	0.88	0.85	0.85	0.78	0.74	0.86
Pitfall	0.93	0.64	0.78	0.93	0.	0.59	0.76	0.	0.91
Private Eye	0.41	0.	0.54	0.87	0.92	0.	0.55	0.57	0.83
Breakout	0.68	0.	0.29	0.29	-	-	-	-	0.56
Pong	0.87	0.64	0.51	0.6	0.	0.	-	-	0.81

RF has low performance on both Breakout and Pong games. The reason is these games have an infinite state space. Furthermore, the size of the training set for both games is small. In such games, the RF output is not very important due to the availability of environment reward, which can replace the reward function output.

B. Interacting Results

The result of this phase is reported while the agent is running in the environment. The agent started with an empty buffer. This buffer will store the agent's self-generated data. The environment provides the current state for both the agent and the RF. As in DQfD, our agent selects action using the trained policy from the pre-trained phase. The RF role is to generate rewards array output depending on the current state. The current action reward and the environment clipped reward will add together and save with the current state, action, and next state. The agent uses these transitions for training.

In each game, the convolutional neural network is trained on a single GPU for 500k steps. We used the Nvidia GeForce GTX 1060 graphics card (6 GB memory version). For comparison, our agent, the DQfD agent, and the standard Prioritized Double DQN (DDQN) agent were trained on the same device. The standard PDDQN is the algorithm used to build our approach. The PDDQN agent differs from our agent because it does not have demonstration data, pre-training phase, and reward function. The PDDQN agent took about one week to finish the training for each game. The DQfD agent took about four days for each game, while our agent required 20 days to complete the training for each game. The reason for the difference in training time is due to the number of predications in the algorithms. In each step, our agent makes two predictions, the first prediction is selecting the correct action, and the second is predicting the reward for that action. On the other hand, the DQfD and PDDQN agents predict the correct action only. The difference in training time is the only disadvantage in our work. Fig. 6 Showing the online results for the three algorithms in Montezuma's Revenge and Breakout games.

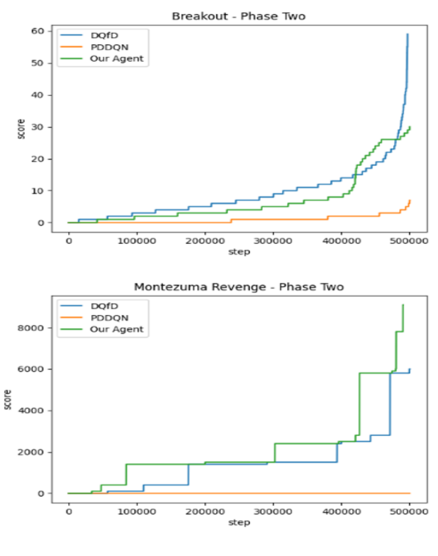

Fig. 6. Online rewards for the Montezuma's Revenge and Breakout games. The zero rewards between episodes are removed. Scores are from the Atari game, regardless of the internal representation of reward used by the agent.

In Montezuma's Revenge game, our agent performance is good. The high score of our agent is about 9000 scores, while the DQfD agent scored 6000 only. As in all standard reinforcement learning algorithms, the PDDQN agent score is zero. In the Breakout game, our agent performance is poor compared with the DQfD agent. Our agent was able to score 30 only, while the DQfD agent scored 60. The reason here is the infinite state space in the simple games. For example, in the Breakout game, the state changes depending on the ball. So, it is impossible to train the RF with all possible states. The breakout game is a simple challenge for all classical reinforcement learning algorithms [7], [10] – [17]. All these works obtained results ranging from 300 to 600 after 200 million steps of training. For 500 steps, our agent performance is better than the performance of the PDDQN agent. Our approach can score higher in both games if trained with millions of steps. The last plot is the ratio between the demonstration and the self-generated data in the mini-batch. As mentioned before, the DQfD agent trained its current network using both demonstrations data and self-generated data. The ratio for both Montezuma's Revenge and Breakout is shown in Fig. 7. Unlike our agent, the DQfD agent relies on demonstrations all-time of training. Furthermore, it uses the demonstrations only at the beginning of the training without returning to the self-generated data. Our agent uses the self-generated data only, which is the main concept in learning from interaction and makes it different from imitation learning.

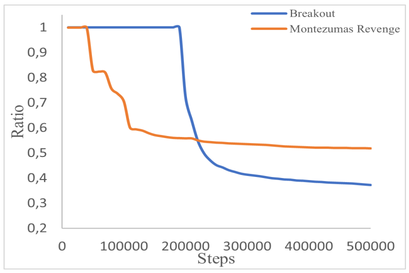

Fig. 7. The ratio of how often the demonstration data was sampled versus the self-generated data in the DQfD algorithm.

VII. Conclusion

We have introduced an integrated approach that uses Reinforcement Learning, Learning from Demonstration, and Inverse Reinforcement Learning for solving the hard-exploration problem. Our approach solves the problem of sparse and/or deceptive rewards by using external rewards resulting in higher scores compared with prior works. The external reward system opens up a large number of new research directions including experimenting with different environments and different methods with one limitation: the availability of human demonstration.

References

[1] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "Go-Explore: a New Approach for Hard-Exploration Problems," arXiv preprint arXiv:1901.10995, 2019.

[2] A. Ng, and S. Russell, "Algorithms for inverse reinforcement learning," In ICML, Vol. 1, 2000.

[3] J. Fu, K. Luo, and S. Levine, "Learning robust rewards with adversarial inverse reinforcement learning," ICLR, 2018.

[4] B. Piot, M. Geist, and O. Pietquin, "Bridging the gap between imitation learning and inverse reinforcement learning," IEEE Transactions on Neural Networks and Learning Systems, vol. 28, no. 8, 2017.

[5] T. Munzer, B. Piot, M. Geist, O. Pietquin, and M. Lopes, "Inverse reinforcement learning in relational domains," in Proc. Of the 24th International Joint Conference on Artificial Intelligence. 2015.

[6] W. Z. Tareq, and M. F. Amasyali, "A New Reward System Based on Human Demonstrations for Hard Exploration Games," CMC-Computers, Materials and Continua, vol. 70, no. 2, pp.2401–2414, 2022.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[8] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. thesis, University of Cambridge England, 1989.

[9] H. V. Hasselt, "Double Q-learning," Advances in Neural Information Processing Systems, vol. 23, pp.2613–2621, 2010.

[10] H. V. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in Proc. of the 30th AAAI Conf. on Artificial Intelligence, Arizona, USA, vol. 30, pp. 2094–2100, 2016.

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in Proc. of the Int. Conf. on Learning Representations, San Juan, Puerto Rico, 2016.

[12] Z. Wang, T. Schaul, M. Hessel, H. V. Hasselt, M. Lanctot, et al., "Dueling network architectures for deep reinforcement learning," in Proc. of the 33rd Int. Conf. on Machine Learning, New York, NY, USA, vol. 48, 2016.

[13] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, et al., "Asynchronous methods for deep reinforcement learning," in Proc. of the 33rd Int. Conf. on Machine Learning, New York, NY, USA, vol. 48, 2016.

[14] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," in Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, vol. 70, 2017.

[15] M. Fortunato, M. G. Azar, B. Piot, J. Menick, M. Hessel, et al., "Noisy networks for exploration," in Proceedings of the International Conference on Learning Representations (ICLR), 2018.

[16] M. Hessel, J. Modayil, H. V. Hasselt, T Schaul, G. Ostrovski, et al., "Rainbow: Combining improvements in deep reinforcement learning," in Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI-18), vol. 32, no. 1, pp. 3215–3222, 2018.

[17] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, et al., "Deep q-learning from demonstrations," in Proceedings of the 32nd AAAI Conference on Artificial Intelligence, vol. 32, no. 1, pp. 3223–3230, 2018.

[18] T. Salimans, and R. Chen, "Learning montezuma's revenge from a single demonstration," in Proceedings of the 32nd Conference on Neural Information Processing Systems NIPS, Montréal, Canada, 2018.

[19] Y. Aytar, T. Pfaff, D. Budden, T. L. Paine, Z. Wang, et al., "Playing hard exploration games by watching YouTube," in Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS), Montréal, Canada, 2018.

[20] T. Pohlen, B. Piot, T. Hester, M. G. Azar, D. Horgan, et al., "Observe and look further: Achieving consistent performance on atari," arXiv preprint arXiv:1805.11593, 2018.

[21] P. Abbeel, and A. Ng, "Apprenticeship learning via inverse reinforcement learning," In ICML, 2004.

[22] D. Ramachandran, and E. Amir, "Bayesian inverse reinforcement learning," In IJCAI, 2007.

[23] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey, "Maximum entropy inverse reinforcement learning," In Proc. of the 23rd AAAI Conf. on Artificial Intelligence, California, USA, vol. 8, pp. 1433–1438, 2008.

[24] E. Uchibe, "Model-free deep inverse reinforcement learning by logistic regression," Neural Processing Letters, vol. 47, no. 3, pp. 891–905, 2018.

[25] A. Tucker, A. Gleave, and S. Russell, "Inverse reinforcement learning for video games," In Proceedings of the Workshop on Deep Reinforcement Learning at NeurIPS, 2018.