Proceedings of the Estonian Academy of Sciences,
2020, **69**, 3, 186–196
https://doi.org/10.3176/proc.2020.3.02
Available online at www.eap.ee/proceedings

COMPUTER SCIENCE

Empirical explorations of strategic reinforcement learning: a case study in the sorting problem

Ching-Sheng Lin[a*], Jung-Sing Jwo[a,b], Cheng-Hsiung Lee[a], and Ya-Ching Lo[a]

[a] Master Program of Digital Innovation, Tunghai University, Taichung 40704, Taiwan
[b] Department of Computer Science, Tunghai University, Taichung 40704, Taiwan

Received 14 November 2019, accepted 14 April 2020, available online 4 June 2020

Abstract. Recent advances in deep learning and reinforcement learning have made it possible to create an agent that is capable of mimicking human behaviours. In this paper, we are interested in how the reinforcement learning agent behaves under different learning strategies and whether it is able to complete the task similar to human performance in principle. To study the effect of different reward types, two reward schemes which include immediate reward and pure-delayed reward are introduced. To build a more human-like agent when interacting with the environment, we propose a goal-driven design that forces the agent to achieve a level close to human ability and a training mechanism that learns only from good trajectories. Q-learning is one of the most popular reinforcement learning algorithms and we employ it for our study. As the sorting problem is a classical topic in theoretical computer science with widespread applications, it is used for the empirical evaluation. We compare our results against the algorithmic solutions.

Key words: reinforcement learning, Q-learning, reward schemes, goal-driven design, good trajectories, sorting problem.

1. INTRODUCTION

The goal of reinforcement learning is to map from states to actions in order to maximize a reward or achieve a goal. Unlike supervised learning problems where each instance with a correct label is given to the learner, the reinforcement learning agent has to try and discover which optimum state-action sequences will yield the best performance for the desired task. At each time step, the agent perceives the current state and takes an action accordingly. In return, the environment transits to a new state based on the agent's action and provides an immediate reward or a one-time reward until the final time step. Since actions of the agent affect not only the reward but also the future state, all these pose the challenge of developing efficient and effective algorithms for reinforcement learning problems.

Q-learning is a well-known reinforcement learning algorithm that the agent learns to act optimally in order to successively approximate the action-value function [1]. The action-value is defined as the sum of the received reward for taking a particular action at a particular state and the expected future reward thereafter. It is desirable for the agent to try all actions in all states sufficiently under some exploration scheme. The strength of Q-learning is that it does not require any model of the environment to perform iterative updates and it has been proved that Q-learning converges with probability one. In reinforcement learning tasks, reward not only defines the objective but also formulates the decision-making process. A properly chosen reward can guide the agent towards desired behaviours, on the contrary, a poorly chosen reward may cause the agent to fail the learning process or move away from the objective [2,3]. Reward schemes, in general, can be classified into two categories including

* Corresponding author, cslin612@thu.edu.tw

immediate reward and pure-delayed reward although some tasks fall in between these two extremes [4]. In the class of immediate reward problems, the environment assigns a value to each action taken at a particular state. The card game snap is an example of this type. In the case of pure-delayed problems, the agent will not receive any rewards for every step of the action but there will be a reward given at the end to indicate a success or a failure. Playing the board game, backgammon, is an example that can be characterized as a pure-delayed problem.

The convergence time of reinforcement learning is a serious concern and developing methods for speeding it up is important. For example, if the game level is too hard, directly learning from the most difficult level may take a long time [5]. Shaping is the idea, originated from behaviourist psychology, to reduce the learning curve for goal-directed exploration and fast training [6,7]. It gradually increases the complexity of the task, so that the agent is allowed to learn easier versions of the task first and use the obtained skills to accelerate learning while the tasks become progressively harder. The learning from Easy Missions mechanism is one of the shaping methods to make the robot deal with easier situations at the early stages and later on navigate in more difficult situations [8].

The main aim of this paper is to explore the use of reinforcement learning to improve the average performance empirically for the problem with already known complexity. Learning from good trajectories can make the agent mimic good experiences and lower the computational costs by updating the parameters for those successful cases only. The goal-driven design allows our explicit goals to be exercised gradually. We propose a training method combining the good trajectories adoption and goal-driven design to balance the speed of convergence time and the quality of results. In addition, since the choice of reward is considered one of the major influences on the quality of policies solved by reinforcement learning, we use two extremes (immediate reward and pure-delayed reward) to investigate how they will affect the timeliness and accuracy of the training task. Sorting, a fundamental data operation, has been applied to many computing tasks and has already attracted intensive interest since its introduction. Among all comparison-based sorting algorithms, the performance cannot be better than $O(n\ log\ n)$ in the average or worst case. To illustrate our approach and to lay the groundwork, we consider the sorting task in which an agent is asked to perform under the designated strategies. The remainder of this paper is organized as follows. Section 2 reviews the related work of this paper. Section 3 describes our approach and detailed strategies over which the agent operates. Experimental results are presented and discussed in Section 4. In Section 5, we summarize conclusions and future work.

2. RELATED WORK

In the reinforcement-learning problem, an agent takes sequential actions for the corresponding states of the environment in an attempt to maximize a reward signal. There are two main methods for solving such a problem, model-based and model-free learning. Model-free approaches directly learn the policy by the trial-and-error interaction without modelling the behaviour of the underlying environment whereas model-based approaches learn to build an explicit model of the environment and then compute the optimal policy relying on the derived model [9]. The details of the model-based approaches are out of the scope of this study and their reviews are omitted for simplicity. There are various algorithms for model-free approaches, but most are classified into one of two families, either value-based methods or policy-based methods, according to the goal of the training.

Value-based methods fit an optimized policy by primarily learning a value function which includes the state value function and the action value (Q-value) function [10]. A state value function is used to determine the expected reward the agent can receive in a given state whereas an action value function is to assess how well the agent performs an action for a given state [11]. Q-learning [1] and SARSA [9] are two well-known and extensively studied value-based methods. SARSA is an on-policy algorithm which fits the Q-value to the current policy depending only on the past states visited and actions taken. On the other hand, Q-learning which is an off-policy algorithm attempts to directly find the Q-value for the optimal policy rather than the policy that was used to generate the data. Because of using the maximum action value to approximate the expected values of actions, Q-learning is less robust due to overestimating the Q-value. Double Q-learning, a double estimator method, is proposed to use two estimators for uncoupling the selection and evaluation of an action, so that it can eliminate the harm caused by the overestimation in Q-learning [12]. In recent years, deep learning methods have gained significant attention and have been successfully applied to a wide range of areas. It is, therefore, natural to adopt deep learning methods for reinforcement learning problems as the rich representation of the deep network could enable the traditional reinforcement learning algorithms to perform more effectively. A recent development is the combination of the Convolutional Neural Network (CNN) [13] and Q-learning to develop a reinforcement learning agent called deep Q-networks (DQN) [14]. It has shown to be effective in Atari games with large state-action spaces and can achieve at or even beyond human level. An extension of DQN is to replace the CNN layers with Long Short-Term Memory networks [15] to address especially for the Partially-Observable Markov Decision Process

(POMDP). The resulting deep recurrent Q-Network [16] is capable of integrating information over time such that it shows similar performance on Atari games but exhibits better performance on POMDP domains. Similarly, there are also versions of double SARSA [12] and deep SARSA [17].

In contrast to value-based methods, policy-based methods try to learn the policy directly instead of maintaining the state/action value function to select the best action accordingly. In practice, the policy is represented explicitly by a parametric probability distribution $\pi_\theta(a|s)$ such that action a is chosen in state s based on the current policy. The objective is to update the parameter θ over the time in order to derive the optimal policy π^* and obtain the largest reward [18]. A major advantage of such methods is their capability of handling large action spaces or continuous action spaces by computing the probability for each action or learning the probability distribution [11]. The REINFORCE algorithm, a well-known policy gradient algorithm, uses Monte Carlo sampling to update the parameter θ via gradient ascent [1]. A natural policy gradient which refers to the steepest descent direction is proposed to replace the gradient with the natural gradient for the purpose of speeding learning [19].

Actor-critic methods combine the advantages of valued-based methods and policy-based methods to optimize both the policy and the value function where the actor refers to the learnt policy and the critic to the learned value function. Since the gradient of policy-based methods is usually estimated by simulation, this may result in high variance and make the algorithm converge too slowly. Adding a critic will not only reduce the variance but also deliver faster convergence [20,21]. Building on top of the traditional actor-critic model, Advantage Actor-Critic (A2C) develops an advantage function to evaluate the policy instead of using the value function [22]. An advantage function represents a relative action value defined by the difference between the action value function and state value function (i.e., $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$). It is used to capture how better an action is compared to the average performance of the policy at a given state in terms of the expected reward [23]. Asynchronous Advantage Actor-Critic (A3C) is an asynchronous version of A2C. The A3C algorithm has multiple actors executing different policies to stabilize training in parallel as more actors are allowed for more exploration. Moreover, although deep reinforcement learning algorithms based on experience replay have achieved success in many challenging domains, asynchronous update in A3C is able to efficiently reduce memory and computation cost per real interaction [24].

Although Convolutional Neural Networks have gained significant popularity and success, most usable network architectures heavily depend on expert knowledge and experience. A meta-modelling algorithm based on reinforcement learning, MetaQNN, is trained to search connections between convolution, pooling, and fully connected layers through an ε-greedy Q-learning strategy with experience replay [25]. MetaQNN is able to yield good performance on small datasets such as CIFAR-10 but is computationally expensive for big datasets due to the search of a huge space. To address this issue, a block-wise network generation pipeline, BlockQNN, automatically designs the network architecture using a fast Q-learning framework where the state s represents the status of the current layer and the action a is the decision for the next successive layer [26]. The focus of BlockQNN is switched to learn the entire topological structure of network blocks to improve the performance rather than designing the entire network. Since on-policy and off-policy techniques have their own advantages, recent methods have been developed to make use of both on-policy and off-policy learning. While policy gradient methods offer stable learning but require collection of large amounts of on-policy experiences, an approach combining policy gradient and Q-learning (PGQ) is proposed to take advantage of off-policy data by drawing experience from a replay buffer [27]. The PGQ approach achieves better performance than DQN and A3C on Atari games. Q-Prop is a sample-efficient policy gradient method which trains an off-policy Q critic as a general control variate to reduce on-policy gradient variance by using Taylor expansion [28]. In addition to the improvement of sample efficiency compared to state-of-the-art policy gradient methods, Q-Prop has outperformed other actor-critic techniques in humanoid locomotion tasks. Among the existing imitation learning methods, Deep Q-learning from Demonstrations (DQfD) is proposed to pre-train the network in DQN by leveraging small sets of demonstration data from a human expert and includes a margin loss which encourages the expert's actions to have higher Q-values than other actions [29]. Once the pre-training is completed, the agent starts to interact with the environment and explores a much larger state space. The experiments are conducted on 42 Atari games and DQfD achieves state-of-the-art results for 11 games.

Compared to prior studies, we are more interested in guiding the reinforcement learning to quickly reach the task. Our novel goal-driven design gradually relaxes the constraints imposed on the agent and forces it to achieve close to human ability level. Moreover, to speed up the convergence time, the goal-driven design is also accompanied with a training mechanism that learns only from good trajectories so as to reduce the computational costs for updating the parameters.

3. METHODOLOGY

In this section, we illustrate our approach with a case study in the sorting problem and present an algorithm for learning from good trajectories. The sorting task is characterized as a reinforcement learning process and our proposed learning technique is considered as an agent to sort the list. At each time step t, the agent observes state s_t representing the current sorting result of the given list and takes an action to exchange the values in the positions i and j. After performing an action, the agent may receive an immediate reward to assess the action or the reward may be delayed.

3.1. Q-learning

Q-learning, a form of model-free reinforcement learning, was proposed for Markov decision processes [1]. It directly provides agents with the capability of learning to act optimally for pairs of states and actions without relying on an explicit model of the Markov process. The core of the algorithm is a Q-value iteration derived from the Bellman Equation [30] given by

$$\mathrm{Q}(s_t, a_t) \leftarrow \mathrm{Q}(s_t, a_t) + \alpha(r_t + \gamma \max_a \mathrm{Q}(s_{t+1}, a) - \mathrm{Q}(s_t, a_t)), \quad (1)$$

where $\mathrm{Q}(s_t, a_t)$ is the action value function of a state-action pair (s_t, a_t) at time step t, α is the learning rate, γ is the discount factor, and r_t is the reward value of taking action a_t given state s_t. $\mathrm{Q}(s_{t+1}, a)$ denotes the action value of the next possible state by choosing optimal action a to maximize the value. However, the above greedy method with pure exploitation (i.e., to choose the action yielding the highest value) may potentially cause the agent to run into local optima quickly. Therefore, the agent need to be capable of incorporating exploration (i.e., to select an action which may not be the optimum for the given state). In practice, ε-greedy is often the first choice to balance the trade-off between exploitation and exploration [31].

In this study, we apply Q-learning as our base reinforcement learning algorithm in the sorting problem. The state is defined as a list of numbers. For the task of sorting six numbers, there will be 720 states in total. The action is to denote the swap of values in position *i* and position *j*. Thus, there are 15 actions to sort six numbers. The Q-learning algorithm calculates a Q-value based on the current sorting result (i.e., state) and the exchange of two numbers (i.e., action). This Q-value indicates the expected values the agent may receive by selecting the action.

3.2. Rewards

In reinforcement learning, the goal of a task is characterized in terms of the rewards and the agent utilizes the received reward as a guidance to produce suitable behaviours. An agent's ultimate objective is to maximize the expected cumulative reward it receives over time instead of focusing on the short-term rewards while interacting with its environment. The reward at each time step t is denoted as r_t and its cumulative reward is defined as

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots\ldots. \quad (2)$$

The formulation of the cumulative reward may be problematic for continuing tasks because it could diverge to infinity. Moreover, all rewards are equally considered, no matter how far away in the future they are. The discount factor, γ, is introduced to prevent the cumulative reward from increasing to infinity and control the weights between future rewards against the current reward. The discounted cumulative reward is defined as

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots\ldots. \quad (3)$$

In the sorting problem, we consider two types of rewards, immediate reward and pure-delayed reward. For the immediate reward, a reward will be given at each time step depending on whether the action improves the number of items in the correct position. For example, if the state of six numbers is '1,3,2,4,6,5', where two numbers are in the correct position, an action of swapping two numbers results in '1,2,3,4,6,5', where four numbers are in the correct positions and the agent will receive a positive reward. On the contrary, if the action causes the ranking worse, a negative reward will be given. For the pure-delayed reward scheme, a reward will only be assigned at the end to indicate a successful or unsuccessful sorting. The purpose of this design is to investigate whether an immediate reward can reduce the number of required trials or a pure-delayed reward could actually reflect the long-term goal best by obtaining rewards in the far future.

3.3. Goal-driven design

Although Reinforcement Learning has been used for autonomous tasks, the prominence largely depends on whether it can be scaled up to do larger and harder tasks. Especially, learning to obtain a good performance in the challenging tasks often takes a very long time. In this paper, we propose a method based on relaxing the constraints to approach the goal more rapidly and learn from easier tasks. With relaxing the constraints currently imposed on the agent, we give the learner more freedom and less restriction to discover useful and effective policies. What we do is essentially adjust the termination condition to allow better execution in an attempt to enhance success for the missions. Generally, it is

applicable to know which learning condition is easier to fulfil than others in order to reach the goal given that a priori knowledge of the problem is available. In the sorting problem, we consider the constraint on the number of actions (i.e. swaps between two numbers) taken. We initially put severe constrains on the problem and gradually relax the constraints over the course of learning until the goal is achieved or time runs out. Doing this has the great merit of making the problem simpler by increasing the allowed number of swaps.

3.4. Good trajectories

The agent learns to improve its skill from observable histories called trajectories where each trajectory is a state-action sequence of length h denoted as $\langle(s_1,a_1), (s_2,a_2)...(s_h,a_h)\rangle$. A good trajectory, in this study, is a sequence in which the agent is able to reach the goal state successfully within a predefined number of steps. By contrast, if the agent does not finish the task promptly, we consider it a bad trajectory. Although good trajectories may not be the optimal solution and there is no guarantee that learning from good trajectories will make the agent have performance comparable to or better than that of an expert, at least it is clear that the agent should try to imitate those aspects of the teacher agents. On the contrary, bad trajectories are even more ambiguous because it is not obvious and sometimes difficult to differentiate whether the entire trajectory was wrong or some parts of the trajectory were correct.

Figure 1 shows the flowchart of our training process. The input of the algorithm is a training set (denoted as Training_Set) which is randomly selected from $n!$ lists where each training sample contains the state and constraint. At each iteration, the agent will interact with the training sample to perform the sorting task. If the training sample is unable to be sorted, we will relax the constraint of the sample as described in the previous section and save the sample into the replay set (denoted as Replay_Set) without updating the Q-table; otherwise, the sample will be removed from the Training_Set and the Q-table will get updated. At the end of each iteration, elements in the Replay_Set will be the new target training samples. The training process keeps iterating until all training samples have been trained successfully.

4. EXPERIMENT AND RESULTS

In order to determine the efficacy of our proposed approach, a case study in the sorting problem is presented. We conducted a series of experiments to observe the performance under the learning configurations as well as to compare the results with other algorithmic solutions.

4.1. Experimental setup

We construct two experimental tasks that aim at evaluating our learning strategies to sort n numbers. The first task is to apply two reward schemes, which is specified in Subsection 3.2 on the sorting problem with loose constraints. A Q-learning agent is asked to reach the goal state within n^2 actions. Given a training sample (i.e. a list of numbers), we are interested in the number of episodes that the agent can approach a 90% successful rate for the latest 100 episodes. This task is designed to assess the effect on the different reward schemes. For the second task, the agent is expected to interact with the environment based on the training flowchart in Fig. 1. We start this task with a strict constraint that the agent is required to finish sorting within n actions. Given a training example, we are interested whether the agent can have a 90% successful rate for the latest 100 episodes within 45 000 episodes. If the agent is able to fulfil the requirement, we will update the Q-table and remove this training example from the training set; otherwise, the Q-table will stay unchanged and the constraint will be relaxed to $n+1$ actions at the next iteration. The goal of this task is to investigate the feasibility about forcing the agent to learn from good trajectories and easier missions. In the next section, we explain our performance according to the designated tasks and compare our results to Quicksort.

4.2. Experimental results

To evaluate our approach, we employ the proposed Q-learning agent to sort $n!$ lists where each list is a permutation of n numbers. As a case study, we demonstrate our results for n equal to 6, 7, and 8. In the training step for both tasks described in Subsection 4.1, we randomly select 40 lists as our training set for each value of n.

For the first task, we illustrate the number of training steps needed to take in order to sort each training example within n^2 actions for different values of n across two different reward schemes in Figs 2–4. We note that the performance of immediate reward is significantly over pure-delayed reward. The comparison of the average training steps is outlined in Table 1. As n increases, their differences tend to increase progressively as well. For $n = 6$, there are $6! = 720$ states while it explodes to 40 320 states for $n = 8$. So, even with large state spaces, the agent is still able to reach an average of n^2 actions to sort after acceptable training steps. In this experiment, we keep updating the Q-table no matter the agent fails or succeeds in the sorting, i.e., we learn from both good and bad experiences.

For the second task, the detailed training results at each iteration based on the algorithm in Fig. 1 are reported in Tables 2–4. We note that, in general, immediate reward

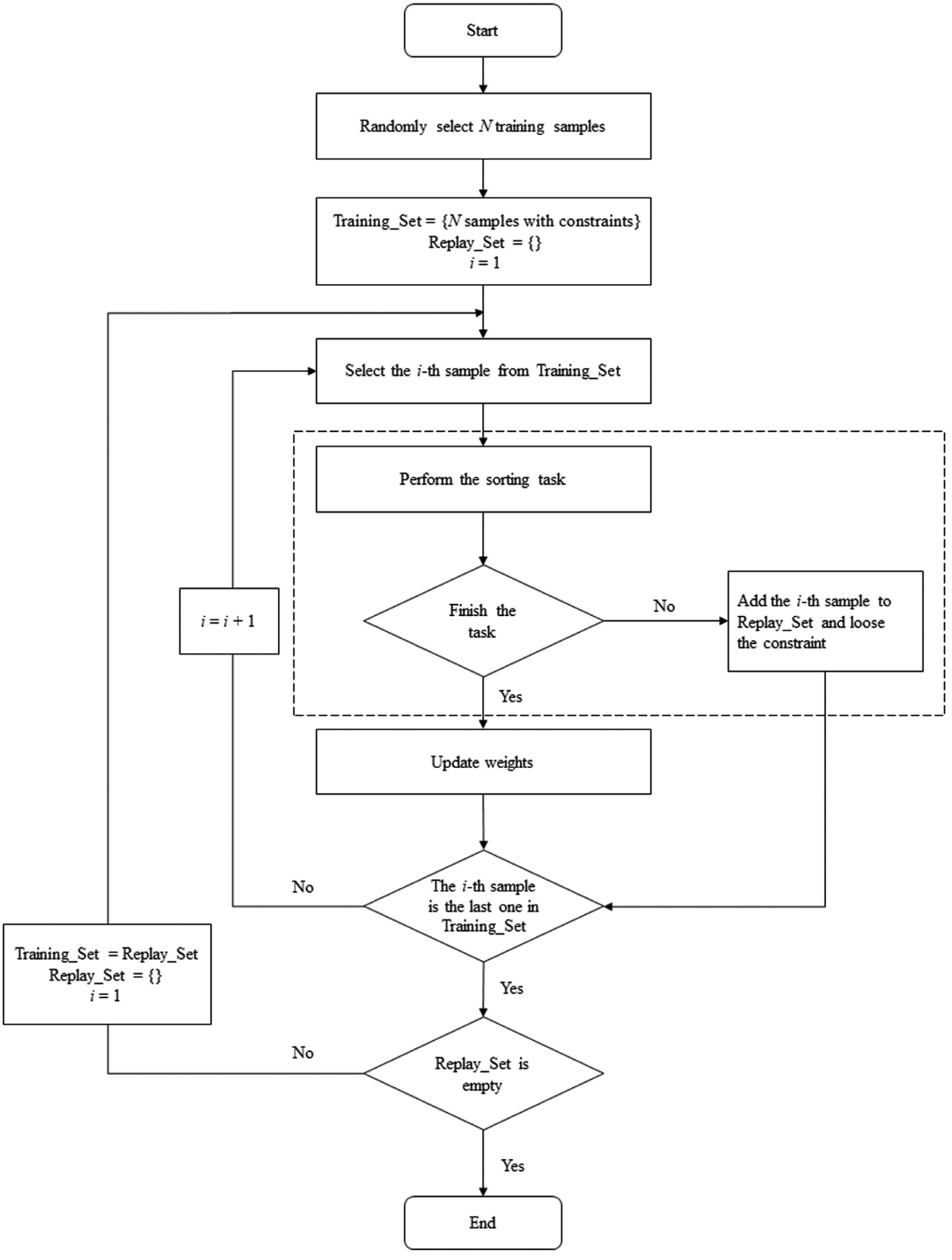

Fig. 1. Training flowchart where the dashed rectangle represents the goal-driven design.

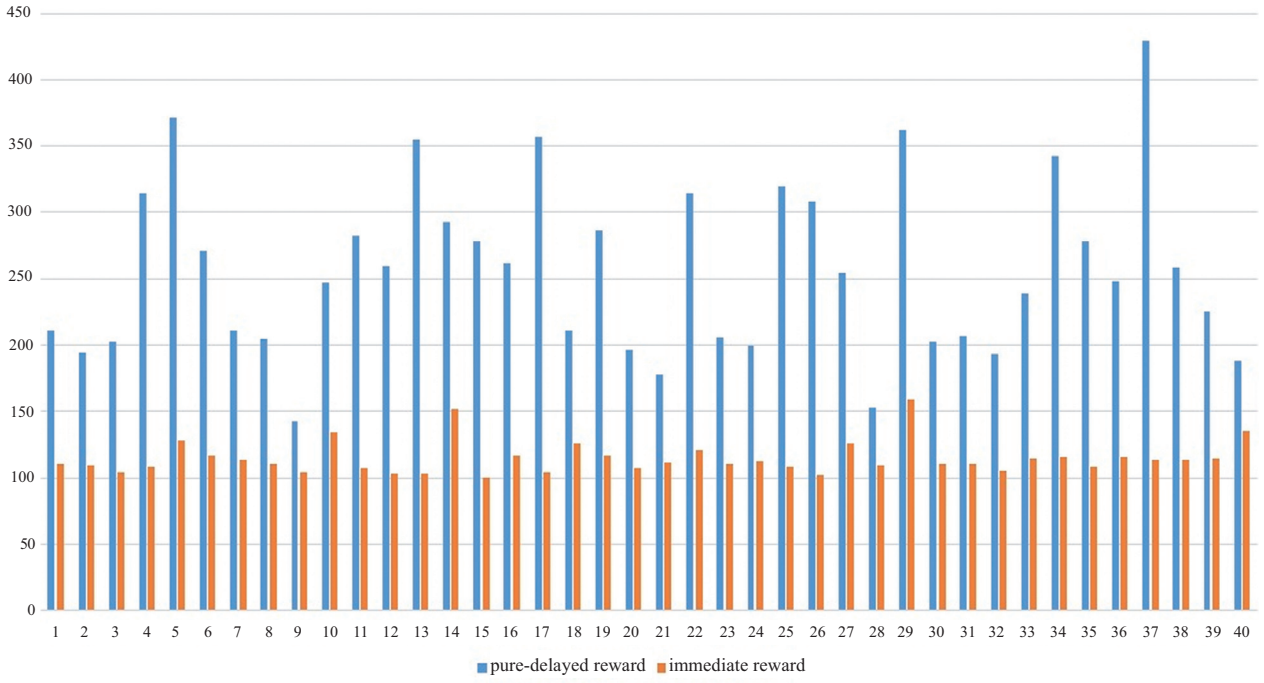

Fig. 2. Distribution of the number of training steps for two reward schemes when $n = 6$.

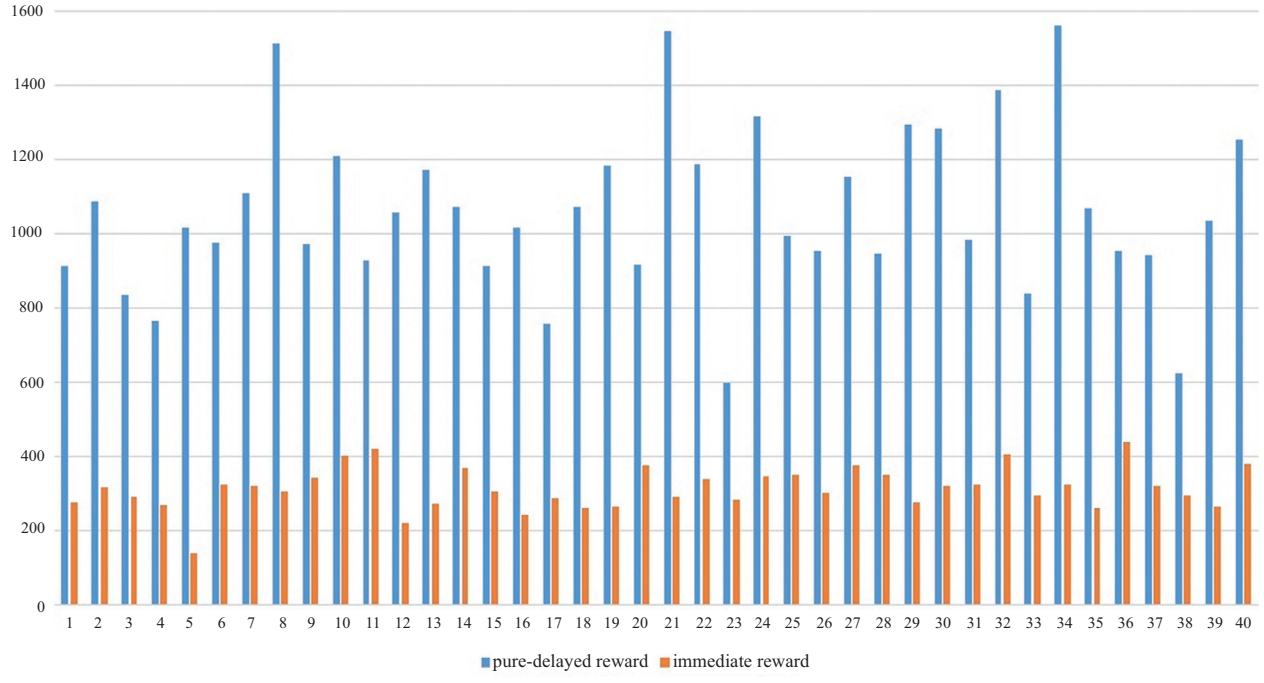

Fig. 3. Distribution of the number of training steps for two reward schemes when $n = 7$.

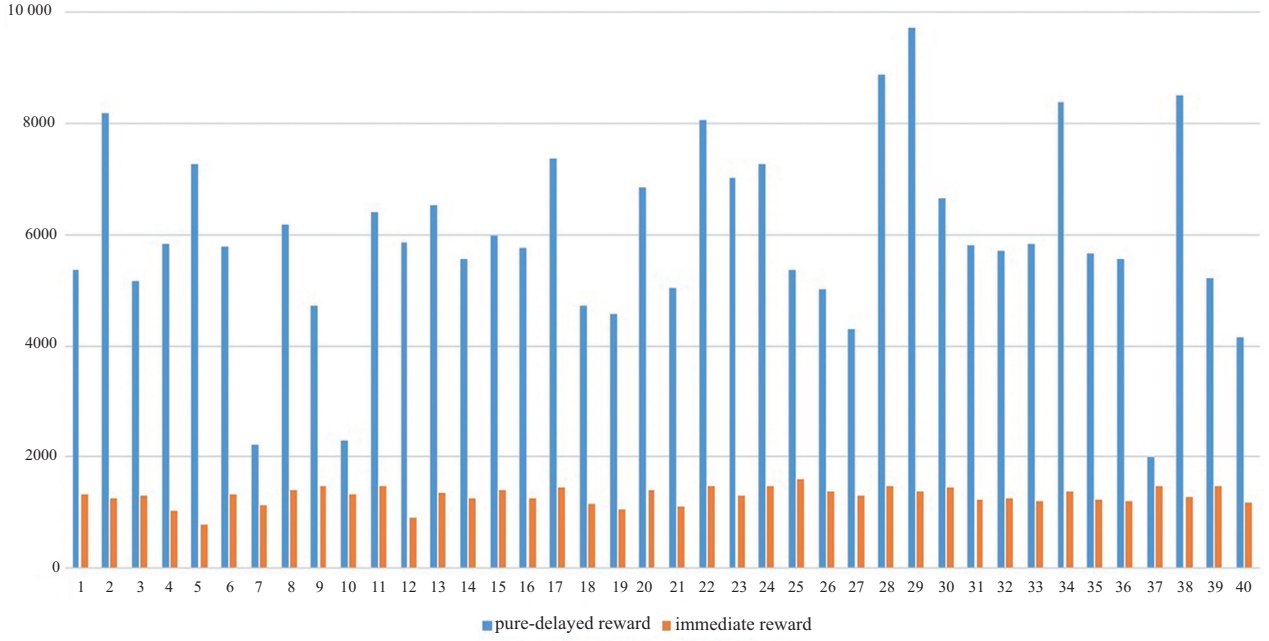

Fig. 4. Distribution of the number of training steps for two reward schemes when $n = 8$.

Table 1. The average training steps for two reward schemes when n equals 6, 7, and 8

n	Pure-delayed reward	Immediate reward
6	256.5	114.7
7	1061.1	314.4
8	5919.4	1297.6

Table 2. The number of training iterations for two reward schemes when n equals 6

Iteration	Pure-delayed reward		Immediate reward	
	Success	Failure	Success	Failure
1	11	29	14	26
2	13	16	13	13
3	14	2	12	1
4	2	0	1	0

Table 3. The number of training iterations for two reward schemes when n equals 7

Iteration	Pure-delayed reward		Immediate reward	
	Success	Failure	Success	Failure
1	15	25	14	26
2	13	12	10	16
3	5	7	13	3
4	4	3	3	0
5	2	1		
6	1	0		

Table 4. The number of training iterations for two reward schemes when n equals 8

Iteration	Pure-delayed reward		Immediate reward	
	Success	Failure	Success	Failure
1	12	28	13	27
2	7	21	7	20
3	6	15	12	8
4	4	11	4	4
5	7	4	4	0
6	1	3		
7	3	0		

requires a smaller number of iterations to finish the training procedure except when $n = 6$, where both immediate reward and pure-delayed reward require four iterations. For $n = 6$ and given the immediate reward scheme, there is only one list that requires 10 (i.e., n + number of iteration) actions to finish the training procedure. For $n = 7$ and $n = 8$ with the immediate reward scheme, all 40 lists can be trained to sort within 11 (i.e., $7 + 4$) actions and 13 (i.e., $8 + 5$) actions, respectively.

To examine whether or not our approach offers support for the empirical data when compared to the existing solution, we assess our results against Quicksort. After the training process, we conduct the experiments to sort $n!$ lists for 50 times while n equals 6, 7, and 8. Table 5 displays the comparative analysis of maximum and average for

Table 5. The performance comparison between two reward schemes and Quicksort for 50 experiments of different values of n. From the aspect of average performance, the immediate reward approach achieves the best results which are highlighted in bold

	Maximum	Average
$n = 6$		
Pure-delayed reward	123.24	11.06
Immediate reward	18.48	**5.03**
Quicksort	15	10.3
$n = 7$		
Pure-delayed reward	87.9	10.19
Immediate reward	30.54	**6.67**
Quicksort	21	13.49
$n = 8$		
Pure-delayed reward	126.38	11.88
Immediate reward	46.96	**9.18**
Quicksort	28	20.49

Quicksort, pure-delayed reward, and immediate reward given different values of n. From the aspect of average performance, immediate reward achieves promising results and it only needs less than half of the values obtained by Quicksort for all cases of n. On average, it only takes 5.03 actions to sort six numbers, 6.67 actions to sort seven numbers, and 9.18 actions to sort eight numbers. Although pure-delayed reward did not yield a good average value while attempting to sort six numbers, it still holds better performance for sorting seven and eight numbers compared to Quicksort. The detailed results of 50 experiments for n equal to 6, 7, and 8 are shown in Fig. 5, Fig. 6, and Fig. 7, respectively. In conclusion, our training algorithm for both reward schemes is capable of improving the behaviour by learning from good experiences and goal-driven strategy despite its training overhead.

5. CONCLUSION AND FUTURE WORK

In this paper, we described in detail our learning method that enables the reinforcement learning agent to improve its behaviours. We introduced human-like strategies for the agent to adopt while gaining experience with the environment. Two different reward schemes (pure-delayed reward and immediate reward) were chosen so as to guide the learning process and access their impacts on the training. As a result of our experiment, immediate reward took much less steps to complete the training process than pure-delayed reward. To shorten the learning time and avoid unnecessary trial and error, we adopted rapid and easy task learning by relaxing the constraints of maximum allowable actions. To exploit and imitate past good experiences, we propose to learn from good trajectories in which the agent is able to reach the goal state efficiently. Our case study was conducted on the sorting problem. On average, the number of comparisons for immediate reward is about half the number of comparisons necessary for Quicksort. The empirical results indicate that there is a substantial reduction in the number of steps to solve the problem.

The future work is to investigate the selection of training samples and evaluate its influence on the sorting task. In our current approach, we randomly selected 40 lists as our training set without applying instance selection techniques. We plan to study whether the property of training samples influences the outcome of the training and accuracy of the performance by considering the difference of choosing between difficult states (i.e., most digits are in the wrong positions) and easy states (i.e., most digits are in the correct positions) as the training targets. Moreover, we will also investigate whether the number of training samples has significant impact on training processes, and deal with the automatic determination of the number of training samples. Another

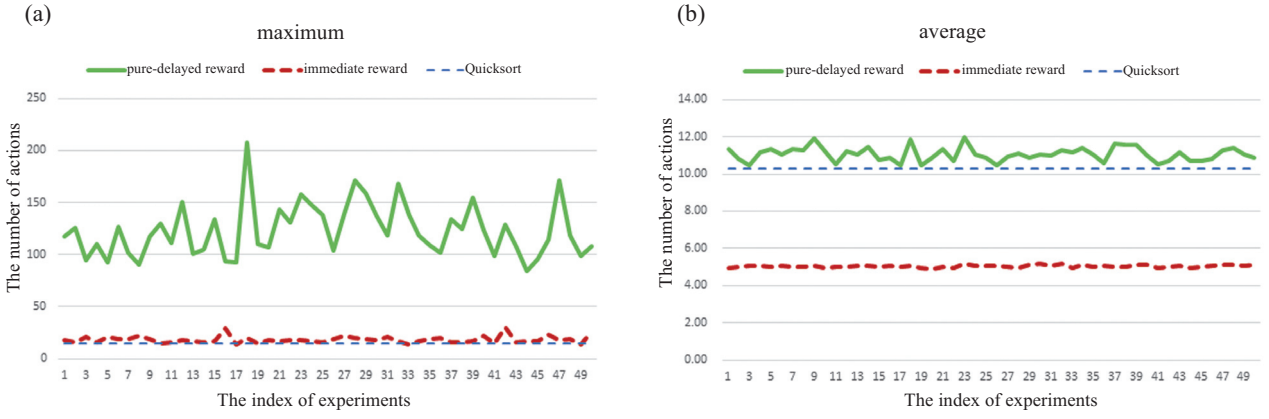

Fig. 5. Detailed results of maximum (a) and average (b) values to sort 6! lists for 50 times regarding pure-delayed reward, immediate reward, and Quicksort.

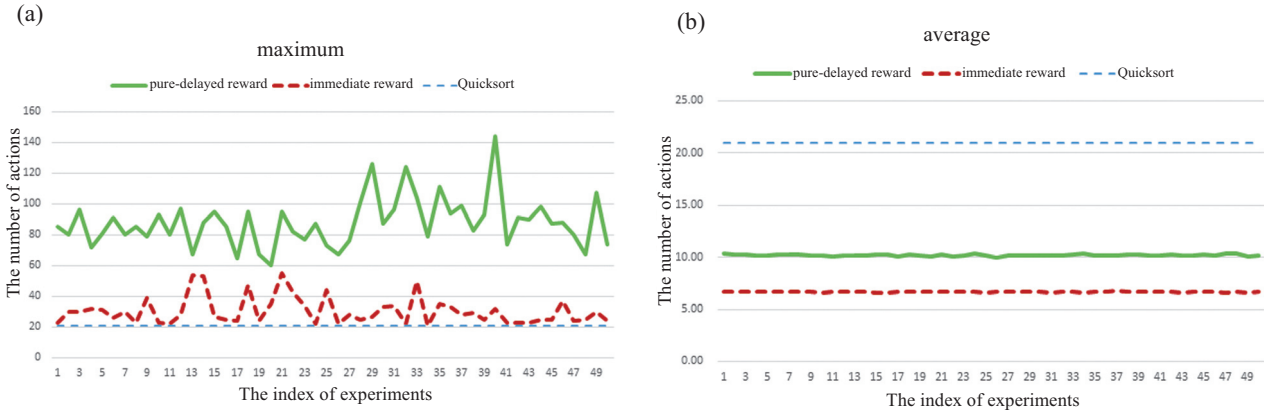

Fig. 6. Detailed results of maximum (a) and average (b) values to sort 7! lists for 50 times regarding pure-delayed reward, immediate reward, and Quicksort.

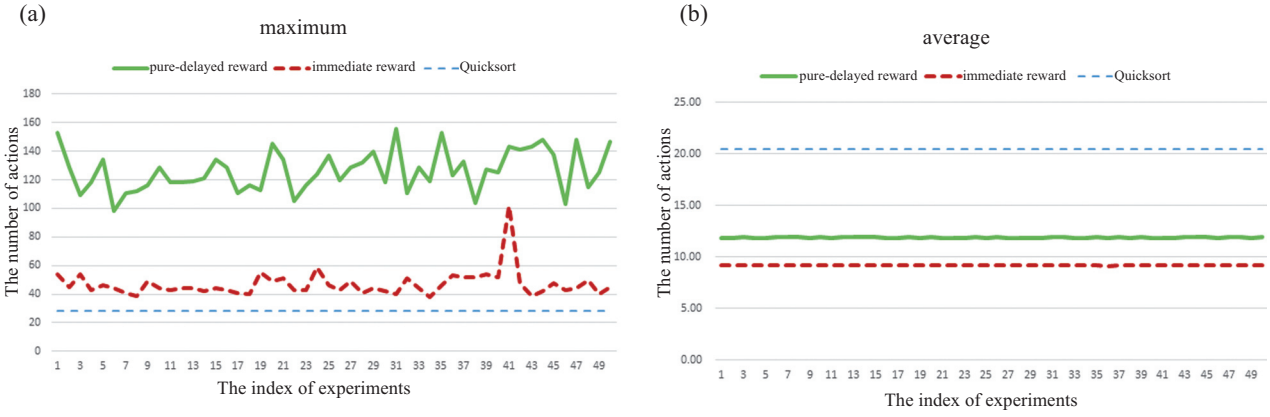

Fig. 7. Detailed results of maximum (a) and average (b) values to sort 8! lists for 50 times regarding pure-delayed reward, immediate reward, and Quicksort.

avenue of future work is to improve the efficiency of the training process. As we can see from Table 1, it takes more than 1000 training steps to complete the training task for n equal to 8 with loose constraints (n^2 actions). It would be necessary to accelerate learning in order to handle larger n. Transfer learning has been used to speed up learning through the adaptation of previously learned behaviours by the inter-task mapping. In the sorting problem, instead of randomly initializing weights for larger n, we will explore its state similarity with previously learned tasks and initialize its action-value by leveraging the knowledge from prior learned and similar states.

ACKNOWLEDGEMENT

The publication costs of this article were partially covered by the Estonian Academy of Sciences.

REFERENCES

1. Watkins, C. J. C. H. and Dayan, P. Q-learning. *Mach. Learn.*, 1992, **8**(3–4), 279–292.
2. Ng, A. Y. and Jordan, M. I. *Shaping and Policy Search in Reinforcement Learning*. University of California, Berkeley, 2003.
3. Devlin, S. and Kudenko, D. Theoretical considerations of potential-based reward shaping for multi-agent systems. In *The 10th International Conference on Autonomous Agents and Multiagent Systems, Volume 1*, 2011, 225–232.
4. Gaskett, C. *Q-Learning for Robot Control*. Australian National University, 2002.
5. Andrade, G., Ramalho, G., Santana, H., and Corruble, V. Extending reinforcement learning to provide dynamic game balancing. In *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games, 19th International Joint Conference on Artificial Intelligence (IJCAI)* (Aha, D. W., Muñoz-Avila, H., and van Lent, M.). Edinburgh, Scotland, 2005, 7–12.

6. Erez, T. and Smart, W. D. What does shaping mean for computational reinforcement learning? In *2008 7th IEEE International Conference on Development and Learning, Monterey, California*. IEEE, 2008, 215–219.
7. Konidaris, G. and Barto, A. Autonomous shaping: knowledge transfer in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning* (Cohen, W. W. and Moore, A., eds). Association for Computing Machinery, NY, 2006, 489–496.
8. Asada, M., Noda, S., Tawaratsumida, S., and Hosoda, K. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Mach. Learn.*, 1996, **23**(2–3), 279–303.
9. Rummery, G. A. and Niranjan, M. *On-Line Q-Learning Using Connectionist Systems*. Department of Engineering, University of Cambridge, England, 1994.
10. Tangkaratt, V., Abdolmaleki, A., and Sugiyama, M. Guide actor-critic for continuous control. 2017, arXiv:1705.07606.
11. Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, USA, 2011.
12. Hasselt, H. V. Double Q-learning. In *Advances in Neural Information Processing Systems* (Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S., and Culotta, A., eds). 2010, 2613–2621.
13. Kim, Y. Convolutional neural networks for sentence classification. 2014, arXiv:1408.5882.
14. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing Atari with deep reinforcement learning. 2013, arXiv:1312.5602.
15. Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 1997, **9**(8), 1735–1780.
16. Hausknecht, M. and Stone, P. Deep recurrent Q-learning for partially observable MDPs. In *2015 AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents*. 2015, 1–9.
17. Zhao, D., Wang, H., Shao, K., and Zhu, Y. Deep reinforcement learning with experience replay based on SARSA. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2016, 1–6.
18. Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*. 2000, 1057–1063.
19. Kakade, S. M. A natural policy gradient. In *Advances in Neural Information Processing Systems*. 2002, 1531–1538.
20. Konda, V. R. and Tsitsiklis, J. N. On actor-critic algorithms. *SIAM J. Control Optim.*, 2003, **42**(4), 1143–1166.
21. Marbach, P. and Tsitsiklis, J. N. Simulation-based optimization of Markov reward processes. *IEEE Trans. Autom. Control*, 2001, **46**(2), 191–209.
22. Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*. 2016, 1928–1937.
23. Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. A brief survey of deep reinforcement learning. 2017, arXiv:1708.05866.
24. Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. 2015, arXiv:1511.05952.
25. Baker, B., Gupta, O., Naik, N., and Raskar, R. Designing neural network architectures using reinforcement learning. 2016, arXiv:1611.02167.
26. Zhong, Z., Yan, J., Wu, W., Shao, J., and Liu, C. L. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2018, 2423–2432.
27. O'Donoghue, B., Munos, R., Kavukcuoglu, K., and Mnih, V. PGQ: Combining policy gradient and Q-learning. 2016, arXiv:1611.01626.
28. Gu, S., Lillicrap, T., Ghahramani, Z., Turner, R. E., and Levine, S. Q-Prop: Sample efficient policy gradient with an off-policy critic. 2017, arXiv:1611.02247.
29. Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., et al. Deep Q-learning from demonstrations. In *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018, 3223–3230.
30. Dietterich, T. G. Hierarchical reinforcement learning with the MAXQ value function decomposition. *J. Artif. Intell. Res.*, 2000, **13**, 227–303.
31. Li, Y. Deep reinforcement learning: an overview. 2017, arXiv:1701.07274.