IRL Report

*https://github.com/WZishan/Intro_RL_chessAssignment

Weiyi Wang
weiyi.wang@uzh.ch
19-764-786

Zishan Wei
zishan.wei@uzh.ch
21-737-671

Wenqing Chang
wenqing.chang@uzh.ch
20-752-259

I. Introduction

Reinforcement learning [4] is an area where the agent interacts with the environment to learn to take actions which maximize the cumulative reward. Unlike supervised learning, reinforcement learning uses rewards as signals and focuses on finding a balance between exploration and exploitation. The focus of this project is to apply reinforcement learning algorithms to learn a chess game. We implemented SARSA and Q-learning algorithms.

The rest of this report is organized as follows. Section 2 introduces the reinforcement learning models (SARSA and Q-learning) we used. Section 3 describes the experience replay technique. Section 4 presents an overview of the environment. The results for SARSA, Q-learning and after solving exploding gradient problem are presented in Section 5. Section 6 analysed the results for different parameters, the state representation and administration rewards. Section 7 concludes this project.

II. Models

A. SARSA

State–action–reward–state–action (SARSA) [9] [5] is an algorithm for learning a Markov decision process policy, used in the reinforcement learning area of machine learning. This name simply reflects the fact that the main function for updating the Q-value depends on the current state of the agent "S_1", the action the agent chooses "A_1", the reward "R" the agent gets for choosing this action, the state "S_2" that the agent enters after taking that action, and finally the next action "A_2" the agent chooses in its new state. The acronym for the quintuple $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ is SARSA.

A SARSA agent interacts with the environment and updates the policy based on actions taken, hence this is known as an on-policy learning algorithm. The Q value for a state-action is updated by an error, adjusted by the learning rate alpha. Q values represent the possible reward received in the next time step for taking action a in state s, plus the discounted future reward received from the next state-action observation.

B. Q-leaning

Q-leaning [8] is a model-free reinforcement learning algorithm based on the state-action value (also known as Q) function: $Q : S \times A \to \mathbb{R}$.

When Q-Learning is performed, a Q-table is created and the values are arbitrarily initialized. At time step t, the agent in state s_t can choose an action a_t which maximizes the expected future rewards according to the Q-table or choose an action randomly following epsilon-greedy policy. Then, the agent observes a reward r_t, enters a new state s_{t+1} and updates the Q-table following equation (2).

The algorithm ends when the agent reaches a terminal state s_f. $Q(s_f, a)$ is set to the reward value r observed in state s_f and never updated.

C. Difference of Q-learning and SARSA

From the update equation of SARSA and Q-learning as shown in equation (1) and equation (2), we can conclude that the biggest difference between Q-learning and SARSA [11] is how Q-value is updated after each action. Q-learning is off-policy, and SARSA is on-policy. SARSA [6] uses the Q-value for the next step with action drawn from it following ϵ-greedy policy. In contrast, Q-learning uses the maximum Q-value over all possible actions for the next step.

$$\begin{gathered} Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \\ \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \end{gathered} \tag{1}$$

$$\begin{gathered} Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \\ \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \end{gathered} \tag{2}$$

As a on-policy method, SARSA has the advantages and disadvantages that a on-policy method has:

- attempt to evaluate or improve the policy that is used to make decisions,
- often use soft action choice,
- commit to always exploring and try to find the best policy that still explores,
- may become trapped in local minima,
- more conservative (e.g. cliff walking).

Q-learning (Off-policy methods):

- directly learns the optimal policy,
- evaluate one policy while following another, e.g. tries to evaluate the greedy policy while following a more exploratory scheme,

- the policy used for behaviour should be soft,
- policies may not be sufficiently similar,
- has higher per-sample variance than SARSA, may suffer from converging problems
- may be slower (only the part after the last exploration is reliable), but remains more flexible if alternative routes appear,
- more aggressive (e.g. cliff walking).

III. Experience Replay

Experience Replay is a replay memory technique used in reinforcement learning. Deep neural networks are easily overfitting current episodes, if the input data are highly correlated. To solve the problem, experience replay is proposed in 1993 [7], [1].

Experience Replay stores each experience in a memory including state transitions, actions and rewards, which are necessary data to perform Q-learning and update neural networks [3]. At each time step t of data collection, the transitions (a_t, s_t, r_t, s_{t+1}) are added to a circular buffer called the replay buffer [2]. Then a mini-batch of transitions are randomly sampled from the replay buffer to update the neural networks. This mechanism enables the algorithm to remember and keep track of his past experiences [13].

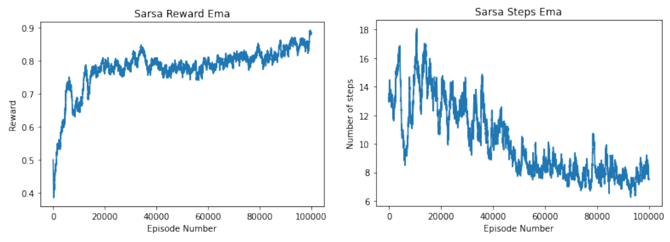
Experience replay learning can be regarded as a kind of meta-learning. There are several analogous studies in the context of RL: learning to generate data for exploration. [10], learning intrinsic reward to maximize the extrinsic reward on the environment. [14], adapting discount factor. Experience replay enables reinforcement learning agents to memorize and reuse past experiences, just as humans replay memories for the situation at hand [12].

IV. Environment

The environment is a simplified version of a chess game. It consists of a 4×4 chessboard with three pieces, 1 King, 1 Queen and 1 Opponent's King. The goal of this game is to checkmate the Opponent's King.

The three pieces are initialized in random locations of the board. In the initial positions, the pieces are not causing any threats. The features representing the states are denoted by a vector s(t), whose dimensionality is $3N^2 + 10$, where N×N is the size of the board and 3 is the number of the pieces involved in this task. The +10 term in the dimension of s(t) contains: (i) the degree of freedom of the opponent King (8 options in 1-hot-encoding), i.e. the number of available squares where the Opponent's King is allowed to move, and (ii) whether the Opponent's King is threatened or not (2 options in 1-hot-encoding). The action of the agent is a one-hot encoded array of 32 possible actions for the Queen and the King. General movement rules are as follows:

- The King can move 1 block at a time.
- The Queen can move in any one straight direction: forward, backward, right, left, or diagonally, as far as possible as long as she does not move through any of the other pieces on the board.



(a) Reward per episode (b) Steps per episode

Fig. 1: Results of SARSA.

- None of the pieces is allowed to exit the board.

The game can end either in a checkmate, where the Opponent's King cannot move and is threatened, or in a draw, where the Opponent's King cannot move but it is not threatened. The agent will get a reward 1 if the game ends in a checkmate or reward 0 if the game ends in a draw.

V. Experiments

A. *SARSA with value approximation*

In this project, we use SARSA algorithm to simulate the process of chess game. Also, we use a one layer neural network with 200 hidden neurons to approximate the Q value. First, we initialize the weights and bias with random numbers. Then we update them through formats below. The weights of last layer will be update through equation (3), the weights of other layers will be update using equation (4) and equation (5).

$$\Delta w_{ij}^{(n)} = \eta \left(R + \gamma Q\left(s', i'\right) - Q(s, i)\right) H\left(h_i^{(n)}\right) x_j^{(n-1)} \quad (3)$$

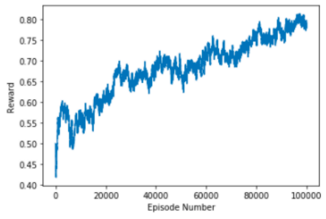
$$\delta_j^{(k-1)} = \sum_i \delta_i^{(k)} w_{ij}^{(k)} H\left(h_j^{(k-1)}\right) \quad (4)$$

$$\Delta w_{jl}^{(k-1)} = \eta \delta_j^{(k-1)} x_l^{(k-2)} \quad (5)$$

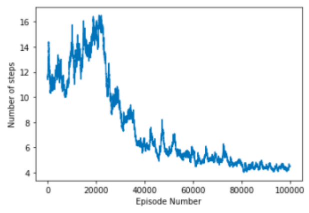
To avoid the coincidence of the experiment, we set 10000 episodes and produce two plots that show the reward per game and the number of moves per game vs training episode.

According to the rule, we get reward 0 if we loss the game and get reward 1 if we win the game. No other rewards in the process. Through the Fig. 1a, we can conclude that with the increase of training time, the average reward increased. It shows that the Q value is getting close to the real action value by training and the agent has learnt how to improve the possibility of winning. But still, we can see that reward stable at around 0.7 which still not reach our goal. We hope that the agent should keep winning at the end which means the reward is supposed to close to 1.

The Fig. 1b plots the average steps it takes to end the game. We may find that the steps decreased with the increase of training time. It also showed that the real value of every action is well measured by our Q value. But still, it takes around 11



(a) Reward per episode



(b) Steps per episode

Fig. 2: Results of Q-learning.

steps to finish the game. And from 15 steps to 11 steps it seems that it didn't improve much. We assume that is because the weights explode during learning. We tried to fix this problem by using Adam.

B. *Q-learnig*

We implement Q-learning algorithm to simulate the process of chess game. The Q value is approximated using the same one layer neural network as shown in SARSA. The weights are initialized randomly and updated after each action following the equation (3)-(5).

The agent learns the game in around 80000 episodes. The average reward is 0.68225 and the average number of moves to checkmate is 7.53242. Fig. 2a shows the average reward achieved for each episode. We could find that the reward is increasing and has not reached the steady state. The final reward reached 0.8. Hence, given more training epochs, we may still improve the results for Q-learning. In Fig. 2b, the number of steps to end the game reached around 4 and remains steady after 80,000 episodes.

Compared to SARSA, the result plots for Q-learning is very noisy using the same weighting factors α for exponential moving average. As illustrated in Fig. 1 and Fig. 2, the number of episodes the agent took to learn the game is larger in Q-learning (80,000) than in SARSA (20,000). Furthermore, Q-learning suffers from convergence problems whilst SARSA stabilized quickly. The average reward for Q-learning is smaller than SARSA. However, the reward of SARSA only reached 0.7 while Q-learning reached 0.8 and is still increasing. For the steps the agent took to end the game, the average number of moves is 10 for SARSA and 7 for Q-leaning. In addition, from Fig. 1b and Fig. 2b, the steps for Q-leaning converged around 4 moves after 80,000 episode while SARSA only decreased to 10. Hence, we can conclude that the results of Q-learning is better than SARSA.

C. *Adams*

1) Exploding Gradients: Exploding gradients and vanishing gradients are actually a same common issue caused by unstable weights update in neural network. Exploding gradients, as implicitly shown in its name, refers to large error gradients accumulate in updating the weights and would result in large gradients, which would cause extreme increase of weights. This would lead to unstable and not learning networks. To fix vanishing and exploding gradient, a common solution would be first changing the error gradients and then feed them to back-propagation. So instead of vanilla gradient descent, gradient should be either scaled or clipped first. Based on this idea, several methods were proposed. In this project we considered Adam.

2) RMSProp: To introduce about Adam, we first start with RMSProp. Root Mean Square Propagation adjusts learning rate automatically as in the formula below.η is the initial learning rate, g_t id the gradient at time t. Equation (6) calculates v_t, which is the exponential average of squares of gradients. We use it for automatically updating learning rate. Using exponential average would weigh recent gradient updates more than older ones.

$$v_t = \rho v_{t-1} + (1-\rho) * g_t^2 \tag{6}$$

$$\Delta w_t = -\frac{\eta}{\sqrt{v_t + \epsilon}} * g_t \tag{7}$$

$$w_{t+1} = w_t + \Delta w_t \tag{8}$$

3) Adam: Adam, Adaptive Moment Estimation is a combination of Momentum and RMSProp. Momentum uses previous steps gradients to help decide the gradient direction. This would help gradient descent converge and find the minima faster. Formula (9)-(12) explains Adam algorithm.

$$v_t = \beta_1 * v_{t-1} - (1-\beta_1) * g_t \tag{9}$$

$$s_t = \beta_2 * s_{t-1} - (1-\beta_2) * g_t^2 \tag{10}$$

$$\Delta w_t = -\eta \frac{v_t}{\sqrt{s_t + \epsilon}} * g_t \tag{11}$$

$$w_{t+1} = w_t + \Delta w_t \tag{12}$$

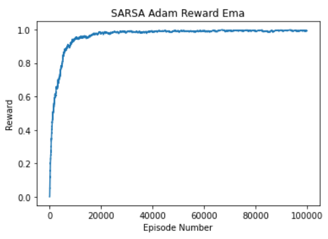
s_t is the exponential average of squares of gradients along w_j. Adam computes both the exponential average of the gradient and the squares of the gradient. As in formula (11), the learning step is decided with multiplying the learning rate by the average of the gradient, and then divide it by the root mean square of the exponential average of square of gradients, which are what was done in the momentum algorithm.

4) Adam Implementation and Performance: We used Adam for our problem. The following two subsections are the comparisons of the performance of the original network and the network with Adam.

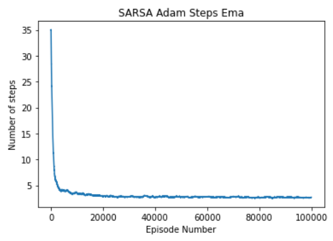
D. *SARSA-Adams*

By applying Adams in calculating the weights of SARSA model, we produce 2 plots of average steps it takes and average rewards it gets to compare with the original model. Before analyzing the plot, by theory, it is expected that with Adam, the network would be more stable and would converge faster. Below are the plots of SARSA with Adam.

It is shown in Fig. 3a that the turning point of reward is at around 10,000, which is a lot faster than SARSA without Adam (only converge at 20,000 episodes). Also the Reward

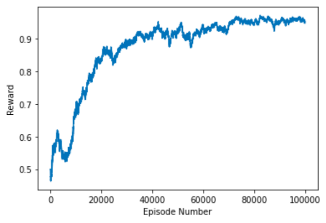


(a) Reward per episode

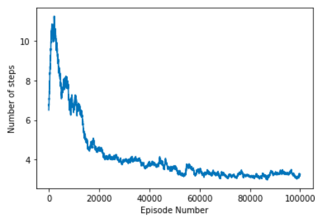


(b) Steps per episode

Fig. 3: Results of SARSA with Adam.



(a) Reward per episode



(b) Steps per episode

Fig. 4: Results of Q-learning with Adam.

converges and stablelized at 1 on episode 20,000, yet SARSA without Adam could only reach 0.7.

It is shown in Fig. 3b that SARSA with Adam quickly stabilized and converged at around 5,000 episodes. Compared with SARSA without Adam, which is much more wiggly and have several large increases before 20,000 episodes, the improved performance is amazingly smooth and converges fast. The original SARSA could only converge to 5 steps at around 100,000 episodes, yet with Adam, it converges below 5 steps and averaged around 2 from almost the beginning.

E. *Q-learning-Adams*

For Q-learning, the performance improved using Adam. The results plot (Fig. 4) is smoother with the same weighting factors for exponential moving average compared to Fig. 2. The average steps to end the game is 4.17616 and the average reward is 0.87446, which are better than Q-learning without Adam. In Fig. 4a, the final reward converged at 1 while without Adam the reward does not converge within 100,000 episodes. The number of steps to end the game converged around 2 steps, yet the steps only reached 4 without Adam (see Fig. 4b). Additionally, the turning point of the reward and steps is in around 20,000 episodes, which decreased significantly. However, the convergence time is still longer than SARSA.

VI. Experiments of different parameters

A. *Discount factor γ and the speed β of the decaying trend of ϵ*

1) SARSA: We observed from previous experiment that the average reward and steps tend to stable when episode equals 30,000. To save the time of training, we trained 30000 episodes to get the average reward and steps of different value of γ and β.

Average Reward		γ		
		0.8	0.85	0.9
β	0.00001	0.8416	0.8751	0.8763
	0.00005	0.8333	0.8798	0.8911
	0.0001	0.8663	0.8823	0.9113

TABLE I: SARSA average reward.

The table I shows the average reward of applying γ from 0.8 to 0.9 and β from 0.00001 to 0.0001. We can find that with the increase of γ the reward also increased and so does β. We can conclude that when γ=0.9 and β=0.0001 the reward is the highest.

Average Steps		γ		
		0.8	0.85	0.9
β	0.00001	3.76	3.73	3.74
	0.00005	4.68	3.36	3.82
	0.0001	3.66	3.36	3.72

TABLE II: SARSA average steps.

The table II shows the average steps of applying γ from 0.8 to 0.9 and β from 0.00001 to 0.0001. However, there is no obvious patterns in steps. We can find that when β = 0.00001 and γ = 0.00005 the step is the highest. When β = 0.00005 or β = 0.0001 with γ = 0.85, the step is the least.

2) Q-learning: Since Q-learning suffers from convergence problem without Adam, we did not use the results (see in Appendix B) for analyse. To save the time of training, we used Adam and trained 60,000 episodes to get the average reward and steps of different value of γ and β. We set γ to 0.8, 0.85, 0.9 and β to 0.00001, 0.00005, 0.0001. The results of different combinations of γ and β are shown in Table III and Table IV.

From Table III, it is clear that the average reward increased as γ increases and so does β. We can find that we obtain the best average reward with $\gamma = 0.9$ and $\beta = 0.0001$. This coincide with the results of SARSA.

Average Reward		γ		
		0.8	0.85	0.9
β	0.00001	0.780417	0.779783	0.83955
	0.00005	0.7852	0.825	0.8644
	0.0001	0.808917	0.84715	0.88225

TABLE III: Q-learning with Adam average reward.

Table IV demonstrated that the average steps decreased as β increases. There is no obvious trend for γ. $\gamma = 0.9$

outperforms all the others with different β. With $\gamma = 0.9$ and $\beta = 0.0001$, we obtain the smallest steps.

Average Steps		γ		
		0.8	0.85	0.9
β	0.00001	5.031517	5.3471	4.952717
	0.00005	4.830117	4.79925	4.5579
	0.0001	4.53138	4.63568	4.11478

TABLE IV: Q-learning with Adam average steps.

B. State Representation

The original state representation is stated in the Section Environment. Our idea of changing the state representation is adding the degree of freedom (DoF) of the agent King and agent Queen to the feature matrix X. We tried three combinations of new state representations, (i) only add DoF of agent King, (ii) only add DoF of agent Queen, (iii) add both DoFs.

We trained the model with and without Adam for three cases and again the models with Adam showed great improvements, with more steady curves. Especially as shown in Fig. 9, the model with all the DoF as input features, this would lead to larger model and more parameters, which means more prone to gradient explode.

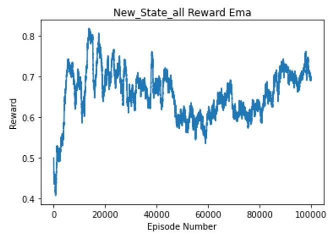
As displayed in Fig. 5 (a) and (b), the model performed badly with low reward and more steps. The steps at episode 100,000 is around 7.5, which is close to a random agent's performance. However, after adding Adam, the advantage of having all DoF is shown, the Reward of the model could converge to 1, which is similar to the performance of SARSA with Adam. Yet the steps could start with very low value as 5.5 at the beginning, which is better than the original model.

Fig. 6 and Fig. 7 compared performance of the model after adding K DoF or Q DoF, the model fluctuates more with Q DoF, as Q adds more parameters thank K. However, the convergence appeared similar. By Comparing Fig. 5 and 6 and 7, we drew the conclusion that by only adding the K DoF is a great enough state representation improvement compared with original SARSA with Adam. Adding more parameters gives more fluctuation than improvement.

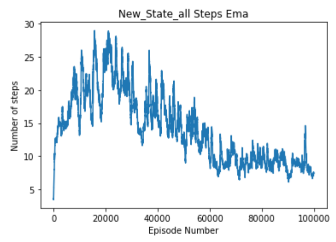
C. Reward

The original Reward just reward the agent 1 when checkmates and reward 0 when draw and when every step that doesn't finish the game. To encourage the agent to checkmate the Opponent in less steps, we decided to add punishment reward every step the agent takes.We set -0.01 reward every step that wasn't checkmate or draw, and keep other reward the same.

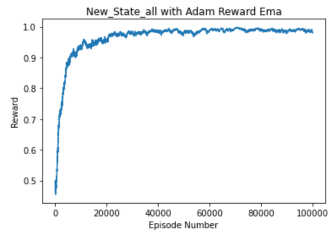
Fig. 8 shows the comparison between original reward and our reward. The original reward start with very high steps around 35, however, with punishment, new reward start with



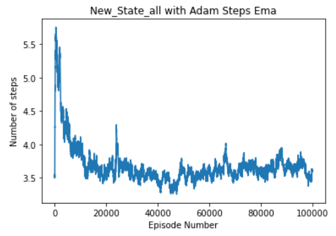
(a) All DoF Reward



(b) All DoF Steps

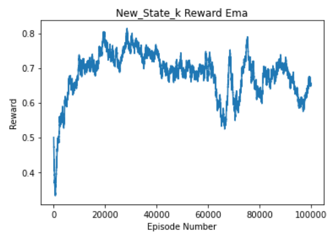


(c) All DoF Reward with Adam

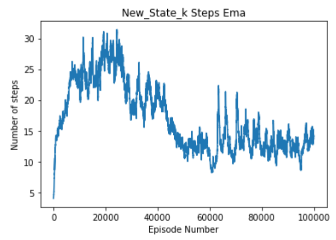


(d) All DoF Steps with Adam

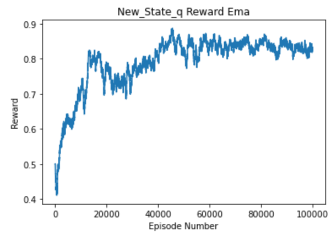
Fig. 5: New state with all DoF trained with/without Adam



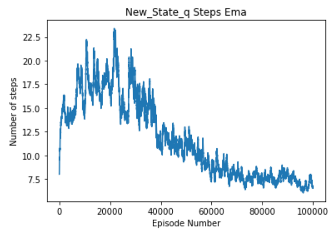
(a) K DoF Reward without Adam



(b) K DoF Steps without Adam



(c) Q DoF Reward without Adam



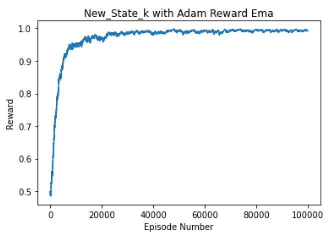
(d) Q DoF Steps without Adam

Fig. 6: New state with K or Q DoF trained without Adam

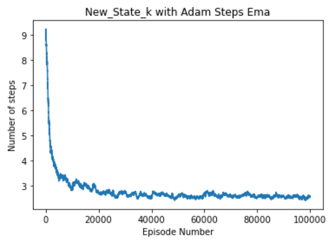
much less steps around 11, which is a reasonable improvement. However, with the new reward, the model fluctuates more, which indicates the punishment of -0.1 every step is not preferred. This would limits the agent ability of exploration, maybe increase the punishment linearly with respect to the steps number would be a better reward administration.

VII. Conclusion

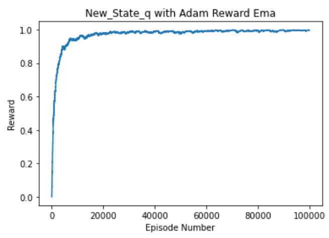
In this project, we aim to simulate a chess game with reinforcement learning algorithm. We implemented SARSA and Q-learning algorithms and applied Adam to avoid gradient exploding. To conclude, before applying Adam, the performance of Q-Learning is better than SARSA in both reward and steps 2 metrics despite of the learning time. After applying Adam, the performance of both models improved a



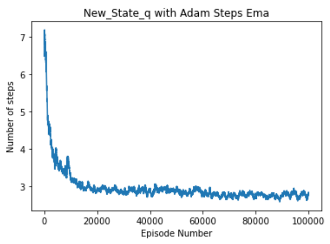
(a) K DoF Reward with Adam



(b) K DoF Steps with Adam

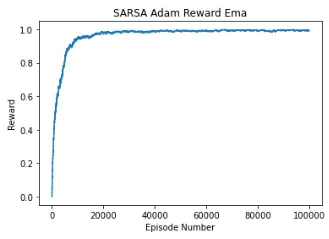


(c) Q DoF Reward with Adam

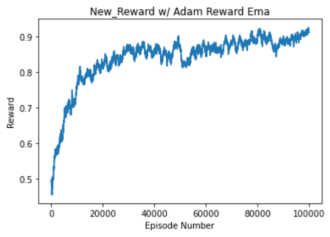


(d) Q DoF Steps with Adam

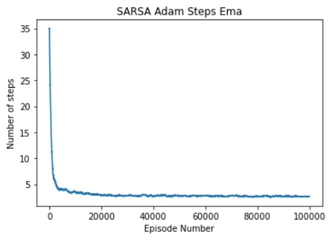
Fig. 7: New state with K or Q DoF trained with Adam



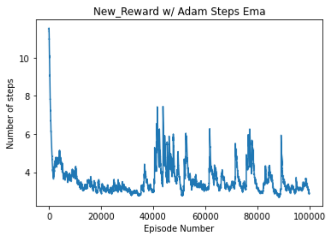
(a) Original R Reward with Adam



(b) New R Reward with Adam



(c) Original R Steps with Adam



(d) New R Steps with Adam

Fig. 8: Original/New R trained with Adam

lot and SARSA shows a better result than Q-learning. The reward and steps of SARSA stable at a better value and less episodes than Q-learning. The improvement shows that the network suffers from gradient explode which could be solved by Adam. We also trained the agent with different parameters: the discount factor γ and the speed β of the decaying trend of ϵ. Furthermore, we changed the state representation by adding the King and Queen's action and added punishment rewards. Adding agent King degrees of freedom to state representation would improve model performance. Administrating reward by add punishment linearly with respect to number of steps would help model converge faster.

References

[1] Sander Adam, Lucian Busoniu, and Robert Babuska. Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(2):201–212, 2012.

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[3] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay. *CoRR*, abs/2007.06700, 2020.

[4] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[5] Mohd Azmin Samsuden, Norizan Mat Diah, and Nurazzah Abdul Rahman. A review paper on implementing reinforcement learning technique in optimising games performance. In *2019 IEEE 9th International Conference on System Engineering and Technology (ICSET)*, pages 258–263, 2019.

[6] Yin-Hao Wang, Tzuu-Hseng S. Li, and Chih-Jui Lin. Backward q-learning: The combination of sarsa algorithm and q-learning. *Engineering Applications of Artificial Intelligence*, 26(9):2184–2193, 2013.

[7] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *CoRR*, abs/1611.01224, 2016.

[8] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[9] Marco A Wiering and Jürgen Schmidhuber. Fast online q(λ). *Machine Learning*, 33:105–115, 2004.

[10] Tianbing Xu, Qiang Liu, Liang Zhao, and Jian Peng. Learning to explore with meta-policy gradient. *arXiv preprint arXiv:1803.05044*, 2018.

[11] Zhi-xiong XU, Lei CAO, Chen Xiliang, Chen-xi LI, Yong-liang ZHANG, and Jun LAI. Deep reinforcement learning with sarsa and q-learning: A hybrid approach. *IEICE Transactions on Information and Systems*, E101.D:2315–2322, 09 2018.

[12] Daochen Zha, Kwei-Herng Lai, Kaixiong Zhou, and Xia Hu. Experience replay optimization. *arXiv preprint arXiv:1906.08387*, 2019.

[13] Shangtong Zhang and Richard S. Sutton. A deeper look at experience replay. *CoRR*, abs/1712.01275, 2017.

[14] Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. *Advances in Neural Information Processing Systems*, 31, 2018.

Appendix

A. Reproduce Guide

Almost all the code are in jupyter notebook named Assignment. To reproduce our experiment results, simply run the jupyter notebook cells in sequence.

We also edited Chess_env.py into a new file named Chess_env_new.py, which is modified with new reward administration and new state representation.

The method Initialise_game was increased with two new boolean inputs, new_feature_k1, new_feature_q1, each in control of adding K DoF and Q DoF, when set True, add new feature, when set False, not. If both set True, then would add both K and Q DoF into X.

The method OneStep was increased with three boolean inputs, except from new_feature_k1 and new_feature_q1, new_reward is in control of implementing new reward administration when set True.

B. Results of discount factor γ and the speed β of the decaying trend of ϵ for Q-learning

Average Reward		γ		
		0.8	0.85	0.9
β	0.00001	0.60158	0.63424	0.67807
	0.00005	0.60354	0.68225	0.7334
	0.0001	0.62252	0.63655	0.72514

TABLE V: Q-learning average reward.

Average Steps		γ		
		0.8	0.85	0.9
β	0.00001	8.7164	6.71375	8.21581
	0.00005	7.2135	7.53242	8.48592
	0.0001	6.9318	7.30768	8.57999

TABLE VI: Q-learning average steps.

C. Meeting Log

Meeting 1
Form: Zoom Meeting

- Sarsa finished by Zishan Wei.
- Report part of assignment 1 and 2 assigned to Zishan Wei.
- Assignment 4 and 5 code and report assigned to Weiyi Wang.
- Assignment 6 and 7 code and report assigned to Wenqing Chang.
- Decided next meeting on Sunday Evening.

Meeting 2
Form: Zoom Meeting

- Zishan Wei updated Sarsa code and reported part of assignment 1 and 2.
- Wenqing Chang updated Adam code and Adam with old Sarsa code.
- Weiyi Wang updated Q-learning code.
- Discussed possible solutions to new reward administration and new state representation.

Meeting 3
Form: In-person Meeting

- Debugging and fixed network not learning problem of Q-learning and SARSA.
- Weiyi Wang updated report with Introduction, Experience Replay and Environment.
- Zishan Wei updated report with Experience Replay Technique.
- Wenqing Chang updated report with Adam, State and Reward.

D. Group Work Distribution

Code

Zishan Wei

- SARSA (Assignment 3)
- Parameters adjust and analyse (Assignment 4)

Weiyi Wang

- Q-learning (Assignment 5)
- Q table test

Wenqing Chang

- Adam (Assignment 7)
- New state representation (Assignment 6)
- New reward (Assignment 6)

Report

Zishan Wei

- SARSA model
- SARSA experiments
- Experience Replay

Weiyi Wang

- Introduction
- Environment
- Q-learning model
- Q-learning experiments
- Q-learning Adam experiments

Wenqing Chang

- Adam
- SARSA Adam experiments
- New state representation
- New reward
- Reference