Reinforcement Learning Report

Iuliia Prozorova
20-746-921
iuliia.prozorova@uzh.ch
University of Zurich

Siddhant Sahu
20-744-579
siddhant.sahu@uzh.ch
University of Zurich

Adriana Mohap
14-725-808
adriana.mohap@zzm.uzh.ch
University of Zurich

***Abstract*—In this study two model free and value based reinforcement learning algorithms are implemented, one on-policy SARSA and one off-policy Q-Learning, on a simplified version of Chess: King-Queen-King Chess Endgame problem. The empirical comparison of the differences in these algorithms and the reasons behind a slightly faster convergence of Q-Learning compared to SARSA are presented. Additionally, variants of these algorithms, namely SARSA with the Adagrad optimization and Q-Learning with Experience Replay, are implemented and a comparative study is made. Moreover, the behaviour of the algorithms under different reward schemes is analyzed, and an exhaustive study on different configurations of hyperparameters is conducted. The code, results and plots can be found in the GitHub repository here: https://github.com/amohap/reinforcement-learning.**

I. Introduction

Reinforcement learning is a type of machine learning training method in which an agent learns iteratively through trial-and-error interactions with its environment [5]. Due to the nature of reinforcement learning problem statements, it is highly applicable to games involving at least one agent interacting with its environment with a final desired outcome. To this end, this report explains the implementation of the king-queen-king endgame in chess using various reinforcement learning algorithms as examples.

The report is structured as follows: The rest of this section is devoted to establishing notation and describing the Markov decision process (MDP) in general, which lays the foundation to understand specific algorithms in greater detail. In section II the different implemented reinforcements learning algorithms applied to the chess game are briefly mentioned and information regarding general implementation is given. The following section III displays the theoretical aspects of three different reinforcement learning techniques along with their specific implementation. Section IV summarizes the results of SARSA and Q-learning applied to the chess endgame with varying hyperparameters. The two sections V and VI are devoted to the explanation of edge cases, different reward schemes applied to the problem statement, and how to overcome certain challenges that arise when dealing with deep reinforcement learning algorithms. Finally, section VII concludes this report with some closing remarks.

Markov Decision Process - This is a decision process in which the cost and transition functions depend only on the current state of the system and the current action and not on history [9].

The way we describe our environment is through a MDP. Additionally, in our problem we assume that our environment is fully observable and that the agent's state is same as that of the environment's state. This is enables us to use the Markov property i.e. what happens next characterized completely by the current state and not the previous states.

More formally, the Markov property is defined as $\mathbb{P}(S_{t+1}|S_t) = \mathbb{P}(S_{t+1}|S_1, ..., S_t)$. The state representation captures all relevant information from the history and once the current state is known, the history can be thrown away. To further extend this, the MDP can now be formally defined. In an MDP, all states are considered to be Markovian.

Definition 1. *A MDP - Markov Decision Process is a quintuple, tuple of five quantities.* $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$

- $\mathcal{S}$ *is the set of states.*
- $\mathcal{A}$ *is the set of actions.*
- $\mathcal{P}$ *is a state transition probability matrix.*
- $\mathcal{P}^a_{SS'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$
- $\mathcal{R}$ *is the reward function,* $\mathcal{R}^a_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$
- γ *is the discount factor* $\gamma \in [0, 1]$

In the king-queen-king endgame it is assumed that the model is unknown since we do model free control. So $\mathcal{P}$, the state transition probability matrix is not known.

Definition 2. *A policy* $\pi : S \mapsto A$ *is a distribution over actions given states,* $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$

Policies can be stochastic or deterministic. For the former, a distribution over all possible sets of actions is received in the action space $\mathcal{A}$, whereas in the later only one 'best' action is obtained.

II. Methods

For the king-queen-king endgame in chess, an environment was provided, which is able to initialize chess games with a given board size and to perform updates of the entire system. In the chess environment the default reward scheme is implemented and modified in section V. In order to reproduce the results shown in this report, a random seed (2022) was set in the beginning of each script. There are four main scripts that were implemented:

- sarsa.py
- q_learning.py
- ex_replay.py
- sarsa_adagrad.py

These can be run independently, or alternatively when optimizing the hyperparameters (see section IV) the corresponding bash scripts can be executed. For example, in order to find the optimal hyperparameters for SARSA, specify the range of the parameters in the $measurements\_sarsa.sh$ bash file (line $3-6$) and then execute the script. However, to give a conclusive overview about all algorithms implemented, the jupyter notebook *Assignment - default parameters.ipynb* contains some additional information. All the algorithms were implemented from scratch without the help from external libraries apart from numpy and pandas for visualizations.

III. Reinforcement Learning Algorithms

The three algorithms that were developed throughout the course of this study are the following: SARSA, Q-learning and Experience Replay. First, the theory of the three algorithms is described along with their advantages and their disadvantages. Then, the implementation of the algorithms is explained in greater detail.

A. Theory

Reinforcement Learning Terms – According to Sutton [13] there exist four distinct components for reinforcement learning problems where an agent interacts with the environment: a policy, a reward signal, a value function, and a model of the environment. An agent follows a specific type of strategy, which is called *policy*. The *reward (signal)* is sent by the environment as a single number and the agent's objective is to maximise the total rewards received over time. The *value function* indicates the expected value for long runs. Hence, a state's value is the total amount of reward an agent can expect to accumulate over the future, starting from that state. The *model of the environment* is an optional component and imitates certain behavior of the environment. Due to this fact, it is possible to make predictions about the environments future states.

Temporal Difference Learning – Temporal difference (TD) learning can be applied to Markovian problem statements in order to learn how to accurately predict future outcomes for environments where the ultimate reward is not immediately observable. As opposed to conventional prediction-learning methods (comparing predictions with outcomes), it is possible to learn from temporally successive predictions [12]. The following two algorithms, SARSA and Q-learning, are based on TD learning.

SARSA – Originally, SARSA appeared as Modified Connectionist Q-Learning (MCQ-L) and is an on-policy algorithm based on TD learning [10]. However, Richard Sutton established the name SARSA, which stands for state-action-reward-state-action representing the following quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ [13]. The update rule for SARSA is the following in non-terminal states:

$$\begin{aligned} Q(S_t, A_t) \leftarrow & \, Q(S_t, A_t) \\ & + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \end{aligned}$$

Q-values are action values, which are estimates of the expected return for each action taken from each non-terminal state. There are two hyperparameters α and γ. The first one, is the learning rate while γ represents a discount factor for future rewards. SARSA is an on-policy learning algorithm, which means that it will follow whatever behavioral policy it has chosen, compute the next action A_{t+1} corresponding to the first action A_t (which generated S_{t+1}) defined by the same policy.

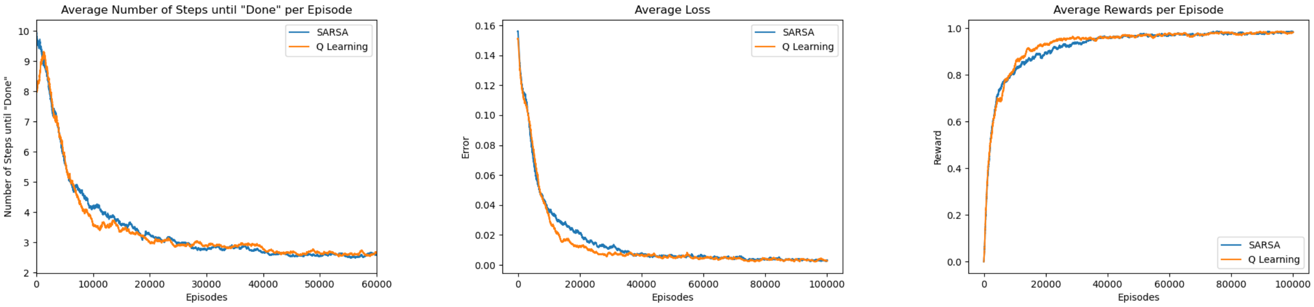
Q-learning – The goal of Q-learning is to estimate the Q-values for an optimal policy (off-policy learning), which are the expected rewards for an action taken in a given state and to maximise the total reward over the entire learning process [15]. That means that the Q-values are always calculated in the same way since the optimal action-value function is approximated, independent of the policy being followed compared to SARSA. The update rule for Q-learning is as follows:

$$\begin{aligned} Q(S_t, A_t) \leftarrow & \, Q(S_t, A_t) \\ & + \alpha[R_{t+1} + \gamma \underset{a}{max} Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned}$$

As opposed to SARSA, only the new state is updated while the new action is not updated.

SARSA vs. Q-learning – The algorithmic difference between SARSA and Q-learning is closely related to the policy. While SARSA is always on-policy, Q-learning learns optimally off-policy when computing the subsequent reward. When a greedy strategy is chosen, SARSA and Q-learning deliver exactly the same results. However, if an epsilon-greedy policy is chosen, then the results will be different since SARSA learns according to this policy (exploration vs. exploitation). One of the main advantages of Q-learning is that the algorithm is model-free and just takes an action-value function as an input and returns the expected utility according to the optimal policy. For SARSA, ϵ (decaying trend) would need to be optimized so that it achieved the optimal action choice. Nevertheless, with SARSA it is possible to explore different paths, whilst Q-learning always performs the same action. Due to the fact that off-policy learning has a higher per-sample variance than SARSA, convergence is not always guaranteed. Hence, SARSA is a more conservative algorithm, which performs some exploratory moves. In the classic "cliff walking" example, the differences between SARSA and Q-learning become apparent for an extreme negative reward scheme with its close optimal path. SARSA tries to avoid this path and will learn this path only very slowly if exploration is reduced. On the other hand, Q-learning will risk to have more negative rewards during exploration. Hence, the result is that Q-learning will on average have a lower reward than SARSA in this setting.

Experience Replay – Experience replay was first mentioned in [7] explaining that an experience is a quadruple (x, a, y, r) representing an action, executed in a state x resulting in a new state y and reinforcement r. Reusing experiences means that the agent memorizes past experiences and presents them



(a) Number of moves

(b) Loss

(c) Reward

Fig. 1: SARSA vs. Q-Learning

to the learning algorithm typically in mini-batches. Sampling experiences can be done with different mechanisms (uniform random or prioritized). Learning through experience replay is performed off-policy, because current parameters are different from those used to generate the sample. There are three major advantages of experience replay. First, data efficiency can be increased because in each step, experience is potentially used in many weight updates. Second, experience replay breaks temporal correlations of neural network learning updates. Learning subsequent samples violates the i.i.d assumption, whereas experience replay ensures that samples are uniformly random sampled from mini-batches. This reduces the variance of the updates and increases stability of learning process. Third, the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. To summarize, it breaks correlation arising from samples being taken from the same policy [11] [14] [1].

B. Implementation

SARSA – The training loop for SARSA is organized as follows. The episodes are the number of games that are played until a *Done* state is reached. For each step per episode, an action is chosen based on the ϵ-Greedy policy which is implemented in the function EpsilonGreedyPolicy. It takes Q-values, allowed actions and epsilon as arguments and returns the action based on the policy. Once the action is chosen, it is performed by calling corresponding function OneStep from the environment. Based on the updated Q-values, corresponding weights are masked and updated while performing backpropagation for the implemented neural network. The training continues until a *Done* state is reached or when the number of steps is too high. In this implementation the threshold was set at 50 steps.

Q-learning – The implementation is very similar to the one of SARSA. However, it is different in the point that the action from the next state is obtained off-policy. It is important to notice that whenever 0 is passed as the input argument for ϵ to the EpsilonGreedyPolicy function, it behaves just in a greedy way. That means that the the probability of making a random move becomes 0 and the algorithm will always choose greedily. Therefore, despite the fact that the EpsilonGreedyPolicy function is invoked in Q-learning, the action $a1$ is computed off-policy.

Experience Replay – Due to the advantages of experience replay outlined in the theory section, it was decided to implement it to reason about any possible improvements. Mini-batches of experiences are stored as episodes. A hash table (dictionary) was utilized because it enables fast data access by key. In this implementation, a dictionary of dictionaries was implemented. The outer-dictionaries store the number of episodes as keys and the respective data as values. Each sub-dictionary contains the episode and the respective number of steps. Firstly, randomization was performed across steps within the same episode and secondly, across steps and episodes within 20 subsequent episodes. Weight updates are performed in the following way: a random number of step/episode is drawn from a uniform distribution and the corresponding dictionary is retrieved. Then, the weights are updated based on the retrieved episode-step data. When sampling from memory, the Q-value is recomputed off-policy according to the current network state and then, weight updates are performed as usual. The database is cleared every 20 episodes so that a new mini-batch can be created. A more sensible approach would be to incrementally pop old experiences and push new ones. The disadvantage would be that it is computationally more expensive as all keys in the hash table would need to be updated. Stack data structure could ensure fast data retrieval. However, random episode retrieval would become slow. Therefore, a simplified version of the batch update was implemented. Overall, no significant improvement could be observed. This could be due to sub-optimal batch updates or sub-optimal selection of the episodes. While a uniform distribution was chosen, a priority approach might perform better. Moreover, the parameters were not explicitly trained for experience replay. For reference, in the previous commits there is an old implementation of incremental experience replay, inspired by [6]. The incremental experience replay implementation was performing worse converging at an average reward of 0.6. This is an indicator that randomization in the form of mini-batches improves learning. Furthermore, it was observed that experience replay is more prone to gradient explosion under certain reward schemes as opposed to SARSA or Q-learning. This is probably due to the fact that the loss was high when experiences are sampled randomly. We concluded that the incremental update ensures smoother transitions for the loss while random drawing of samples creates fluctuations.

IV. Results

In order to compare the different results for SARSA and Q-learning, the hyperparameters were optimized for both. Grid-search over a defined hyperparameter space was performed as displayed below in Table I.

TABLE I: Hyperparameter Search Space.

Parameter Name	Default Value	Search Interval
epsilon	0.2	[0.1, 0.2, 0.9]
beta	0.00005	[0.00005, 0.0005, 0.005]
gamma	0.85	[0.8, 0.85, 1]
eta	0.0035	[0.0035, 0.005, 0.007]

The choice of the parameter search space was motivated by the following reasoning: eta needs to be fairly small in order to prevent overflow issues. However, it cannot be too small, otherwise learning would take too long and convergence might not be reached after all. Gamma is the discount parameter, and it was decided to try out to discount less than the default value and to try once without discounting. Epsilon and beta are closely related to exploration and exploitation. Since epsilon was not identified to be the source of overflow issues, a broad range of values could be tested. However, notice, that an initialization of epsilon 0.9 means much exploration and requires a faster decay parameter. This is why a relatively large beta was also added that exploitation happens eventually. More exploration in the initial phase, and moving towards exploitation with time ensures convergence. However, it would require longer training time.

A. SARSA

In both "Fig. 2" "Fig. 3", the average reward per game and the number of moves until *Done* is displayed for different hyperparameter combinations in SARSA. Epsilon and eta stayed constant throughout training, and only beta and gamma varied according to the defined grid above in Table I. For the runs with different configurations of eta and epsilon, please refer to the appendix.

a) Discount Factor γ: With no discounting ($\gamma = 1$), SARSA converges very slowly especially when β is relatively large (bright green line in "Fig. 2"). This is due to the fact, that long and short games are viewed equally from an agent's perspective, and the rewards appear similarly irrespective of the time dimension. The agent receives no information about any time preferences. In addition, the agent is not penalized for performing more moves. Therefore, the average number of steps per game fluctuates at around 8 (with sufficiently large β values) as opposed to the configuration with smaller γs where the number of steps converges to 2. Therefore, it is advised to lower the hyperparameter γ for this type of algorithm and game.

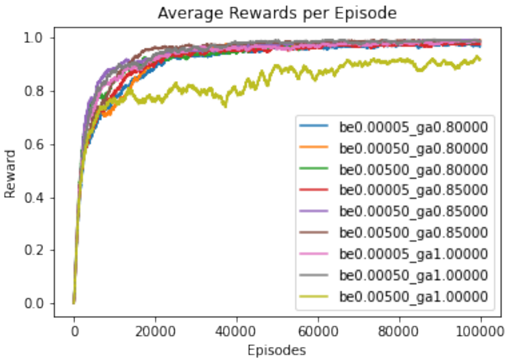


Fig. 2: Rewards for SARSA with different β and γ.

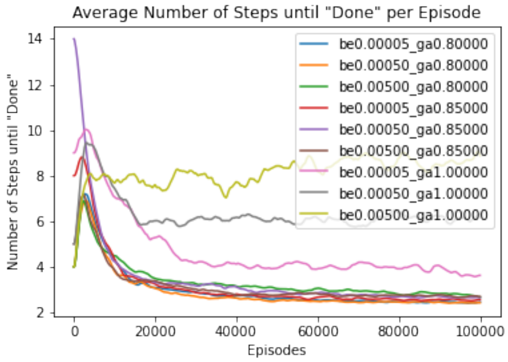


Fig. 3: Number of Moves for SARSA with different β and γ.

b) Speed β: The parameter β controls the degree of exploration. For small β values exploration is encouraged for a longer time period (it decays slower), and convergence of reward is achieved slower than with higher values. This is to be expected because the agent explores for a longer time instead of exploiting already discovered successful strategies. Nevertheless, all of those parameters achieve convergence after $20'000$ episodes and the difference is minor because SARSA by its nature is an exploratory algorithm. It is also important to notice, that with high value of γ, exploration ensured by low β encourages the agent to learn how to play with smaller number of steps. In "Fig. 3" one can observe that the pink line ($\beta = 0.00005$) as opposed to bright green ($\beta = 0.005$) converges to just 4 moves, even though the γ parameter is large. So small β (more exploration) compensates for large γ (less information).

B. Q-Learning

Since the results are very similar for Q-learning compared to SARSA, the corresponding Figures (10 and 11) are attached in the appendix.

a) Discount Factor γ: Similarly, for higher γ values, more steps will be performed. Furthermore, the reward convergence values are lower too. For Q-learning with $\gamma = 1$ the rewards are even lower than for SARSA.

b) Speed β: As opposed to SARSA, Q-learning makes off-policy moves, that is why varying β has a smaller impact in this algorithm. In “Fig. 10” and “Fig. 11” it is visible, that neither reward nor number of moves significantly improves when varying β for $\gamma = 1$. Therefore, little exploration decay β does not compensate for the lack of time-related information.

C. Comparison SARSA vs Q-Learning

In order to compare SARSA to Q-learning with the default hyperparameter setup, the rate of convergence for the rewards, the loss, and the number of steps are visually displayed below in “Fig. 1” and “Fig. 1a”. The loss is attached in the appendix along with the plots including SARSA implemented with Adagrad, and the experience replay implementation.

V. Administration of Reward

In the Chess environment script, the administration of reward was modified for the purpose of this task. The following combinations were considered:

1) Varying reward for staying in the game
2) Varying reward for checkmate vs. draw

The reward schemes were tested for the configurations present in Table II.

TABLE II: Administration of Rewards.

Reward Scheme	Reward Draw	Reward Checkmate	Reward Step
Scheme - 1	-1	1	0
Scheme - 2	0	1	0
Scheme - 3	0	1	0.1
Scheme - 4	1	0	0.1

SARSA **–** Schemes 1 and 2 performed similarly in terms of the convergence speed for the reward as shown in “Fig. 4”. The number of steps per episode in scheme 2 (the default reward scheme) performed on average slightly better. This behavior is expected because both schemes do not penalize or reward for staying in the game as it is demonstrated in “Fig. 5”. Scheme 1 penalizes for ending the game in a draw, which is why the agent was motivated to stay in the game longer but ultimately avoiding a draw situation. It becomes apparent especially when comparing the two schemes in Q-learning where initially scheme 2 outperforms scheme 1 (referring to the cliff walking experiment). In scheme 3, the agent was rewarded for staying in the game, which was supposed to reward exploration. The number of steps per episode increased since the agent is being awarded for each step. However, the reward decreases to an average of below 0.25 because it is easier for the agent to stay in the game as opposed to trying to checkmate. Scheme 4 was a contradictory reward scheme where the agent is rewarded for loosing. This was implemented in order to verify that learning could also be achieved for this kind of reward scheme. It is shown that learning is achieved very quickly with little number of steps. The average reward is high from the beginning, showing that achieving draw is simpler than checkmating.

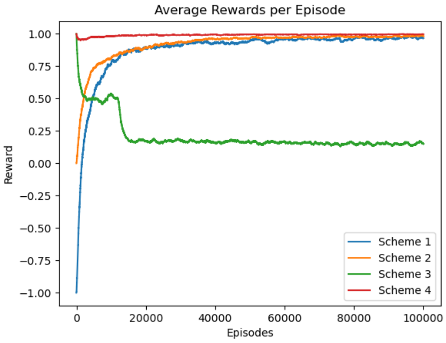


Fig. 4: Rewards for SARSA.

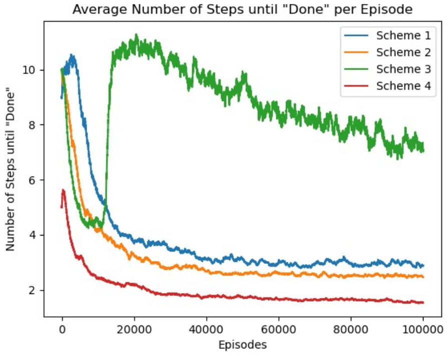


Fig. 5: Number of Moves in SARSA.

Q-learning **–** Similar patterns are observed for Q-learning displayed in “Fig. 15” and “Fig. 16” in the appendix. However, scheme 3 converges to the optimum as opposed to its application in SARSA. A possible explanation is the fact that Q-learning generally explores less than SARSA by not taking random actions when moving off-policy. Therefore, encouraging further exploration by awarding 0.1 for each move is beneficial for Q-learning in contrast to SARSA.

VI. Exploding Gradients

In general, the vanishing and/or exploding gradient problem makes it difficult to learn parameters of the first layers in an n-layer network. This problem becomes worse as the number n of layers in the architecture increases [4]. There are a multitude of reasons causing this issue: poor weight initialization, large learning rates, and/or the increasing (decreasing) results from matrix multiplications applied by the backpropagation algorithm. To overcome this problem various different techniques have been proposed, such as optimization algorithms or the implementation of non-linearities applied as activation functions such as Rectified Linear Units (ReLU) or Sigmoid. Sigmoid applied on the hidden layer might lead to vanishing gradients that is why it was avoided. When applied on the output layer both functions ensure that the output is within a certain range. The range of Sigmoid makes sense in our case due to the applied default reward scheme and improves stability and convergence. The baseline implementation of SARSA and Q-learning is sensitive to weight initialization. By using sensible weight initialization - in this implementation Xavier was used [3] - overflow could be prevented at all times.

A. Adagrad Implementation and Results

In the baseline implementation of SARSA, the learning rate is not adapted during training. If the learning rate eta is relatively too high, gradients explode (or vanish) over time. For example, if the learning rate is initialized with high values (e.g. 0.035), overflow occurs in the first few episodes regardless of activation function and other configurations, indicating that this parameter is critical for the issue of exploding gradients. Optimizers like Adam or Adagrad include adaptive learning rates. They are scaled based on the steepness of the slope in each parameter that they provide [8] [2]. Since Adam was presented in one of the previous labs, Adagrad was implemented instead. Adagrad is implemented in the following way. Gradients get accumulated and are further used to re-scale the learning rate. The parameters with the largest partial derivative have a more substantial decrease in their learning rate. This ensures that the explosion of gradients does not occur and that the learning rate is properly re-scaled. We store the accumulated gradient information in the vector "diagonal". The inverse of the square root of this diagonal is further computed and is used as a re-scaling factor for the learning rate.

Exploding gradients can be observed when large error gradients accumulate and result in very large updates to the neural network. It is achieved with a reward scheme, which has a big reward gap (e.g. ranging between 0 and 10). With such a reward scheme, poorly initialized weights and ReLU applied to the output layer, SARSA overflows immediately even with a small learning rate. In “Fig. 6” the red line indicates the timestamp of the overflow. It can be observed that the gradients become large in absolute values. Adagrad helps in this case: (“Fig. 7”) not only could it optimize without exploding gradients but also it allowed to increase the learning rate substantially. The known disadvantage of Adagrad is that the learning rate can decrease significantly, which then in turn slows down convergence. Nevertheless, in this implementation, it did not seem to be an issue because the algorithm could always converge within the first 10000 steps. Additionally, it was insensitive to different weight initialization schemes. This implies that it indeed gives some stability margin, adds to robustness, and prevents exploding gradients.

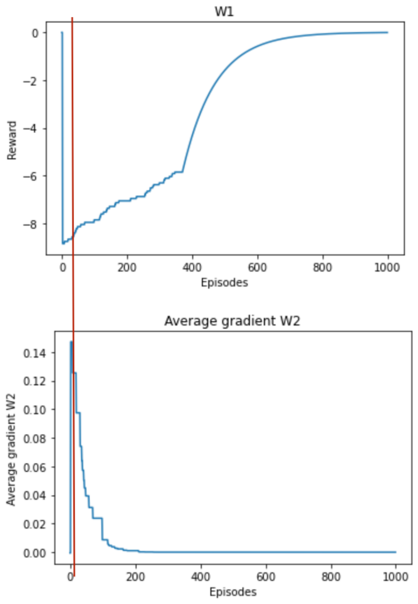


Fig. 6: Example of Overflow (marked in red).

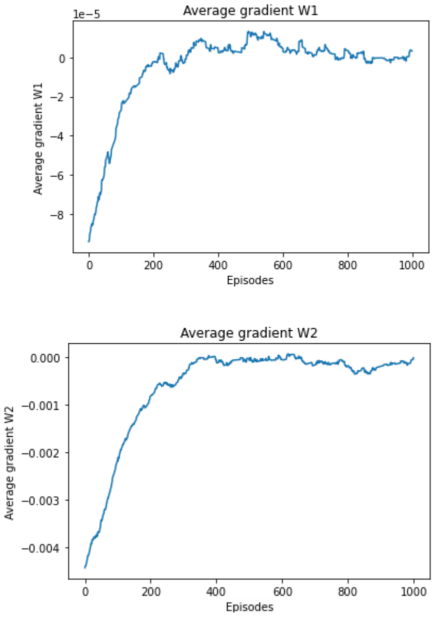


Fig. 7: Smooth Transition of Gradients.

VII. Conclusion

With this report, various reinforcement algorithms were explored theoretically and also implemented from scratch. It could be shown that SARSA and Q-learning can be successfully applied to the king-queen-king endgame of chess. To conclude, both algorithms performed equally good when training a two-layered neural network for sufficient amount of episodes. Furthermore, another reinforcement learning algorithm could be implemented - namely experience replay. By introducing non-linearities to the network and choosing a sensible weight initialization, exploding gradients were mitigated. The choice of hyperparameters is crucial when training the neural network and therefore, needs careful tuning. For future work, it could be of interest to explore other deep reinforcement learning algorithms to solve chess related tasks or to extend this framework to a game of chess with more game pieces.

References

[1] V. Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[2] S. Bock and M. Weiß. “A proof of local convergence for the Adam optimizer”. In: *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2019, pp. 1–8.

[3] S. K. Kumar. “On weight initialization in deep neural networks”. In: *CoRR* (2017).

[4] T. Kurbiel and S. Khaleghian. “Training of Deep Neural Networks based on Distance Measures using RMSProp”. In: *CoRR* abs/1708.01911 (2017).

[5] M. L. Littman L. P. Kaelbling and A. W. Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[6] L. Lin. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[7] L. Lin. “Self-improving reactive agents based on reinforcement learning, planning and teaching”. In: *Machine learning* 8.3 (1992), pp. 293–321.

[8] A. Agnes Lydia and F. S. Sagayaraj. “Adagrad—an optimizer for stochastic gradient descent”. In: *International Journal of Information and Computer Science* 6.5 (2019), pp. 566–568.

[9] M. v. Otterlo and M. Wiering. “Reinforcement learning and markov decision processes”. In: *Reinforcement learning*. Springer, 2012, pp. 3–42.

[10] G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. Citeseer, 1994.

[11] L. Busoniu S. Adam and R. Babuska. “Experience replay for real-time reinforcement learning control”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.2 (2011), pp. 201–212.

[12] R. S. Sutton. “Learning to predict by the methods of temporal differences”. In: *Machine learning* 3.1 (1988), pp. 9–44.

[13] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[14] K. Tuyls T. De Bruin J. Kober and R. Babuška. “The importance of experience replay database composition in deep reinforcement learning”. In: *Deep reinforcement learning workshop, NIPS*. 2015.

[15] C. Watkin and D. Peter. “Q-learning”. In: *Machine learning* 8.3 (1992), pp. 279–292.

Appendix

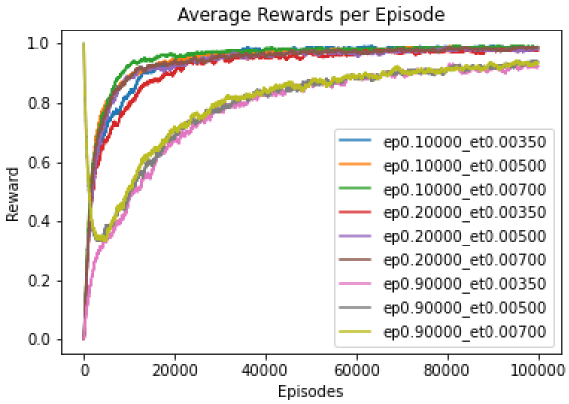


Fig. 8: Rewards for η and ϵ: Sarsa.

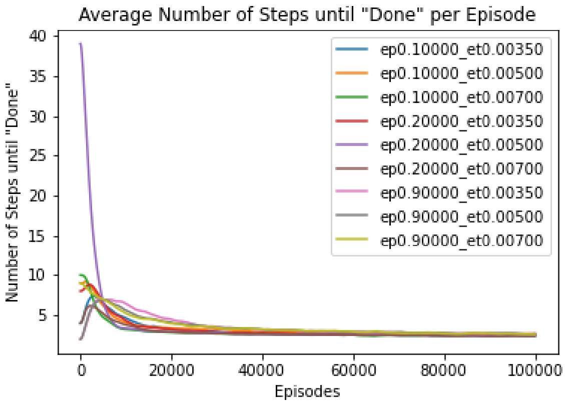


Fig. 9: Number of Moves for η and ϵ: Sarsa.

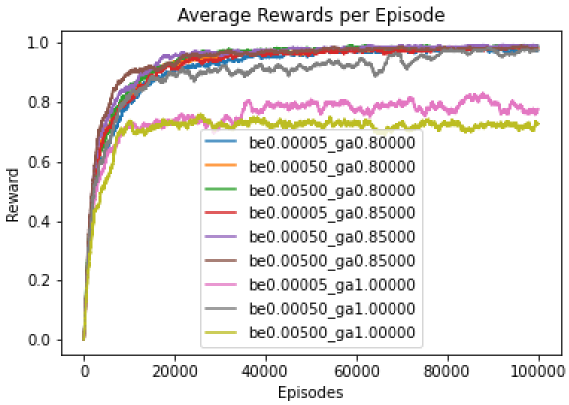


Fig. 10: Rewards for β and γ: Q-learning.

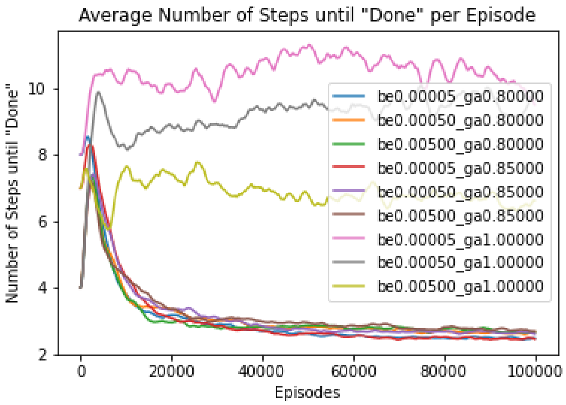


Fig. 11: Number of Moves for β and γ: Q-learning.

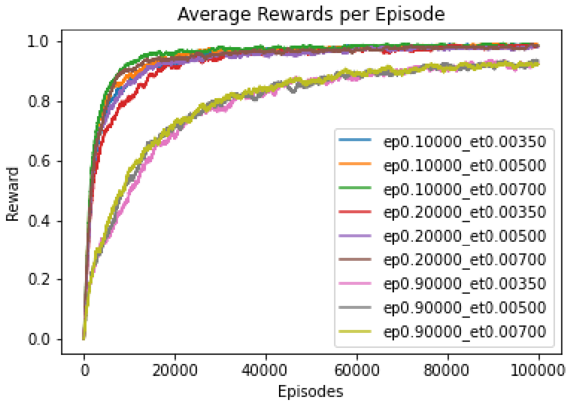


Fig. 12: Rewards for η and ϵ: Q-learning.

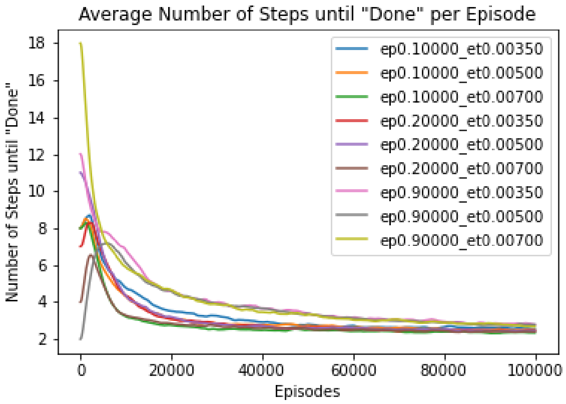


Fig. 13: Number of Moves for η and ϵ: Q-learning.

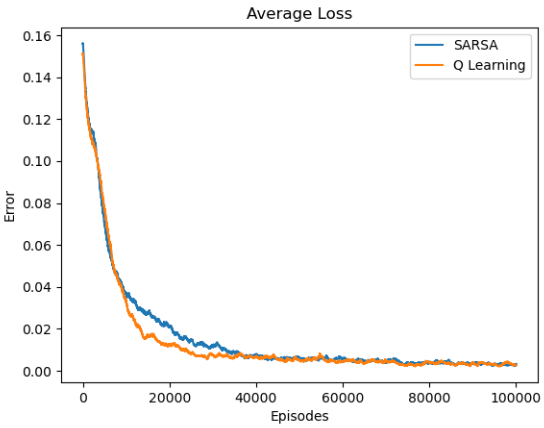


Fig. 14: Loss Sarsa vs. Q-learning.

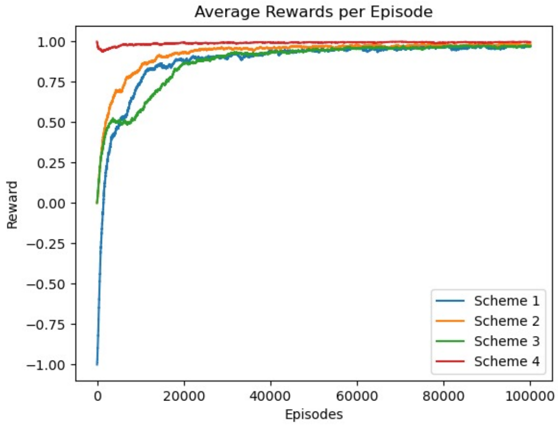


Fig. 15: Average Rewards for Q-learning.

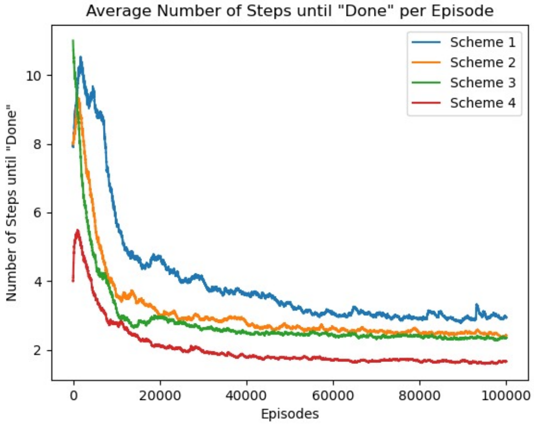


Fig. 16: Number of Moves in Q-learning.

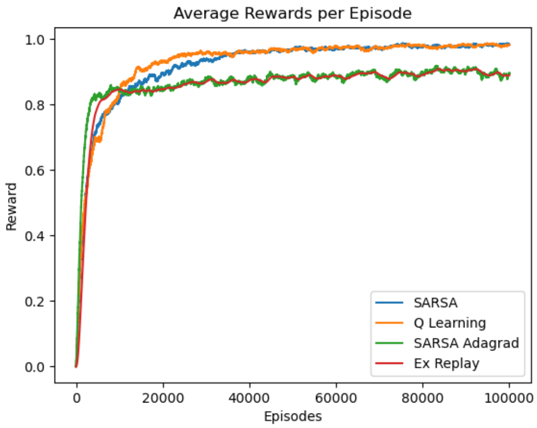


Fig. 17: Reward Comparison.

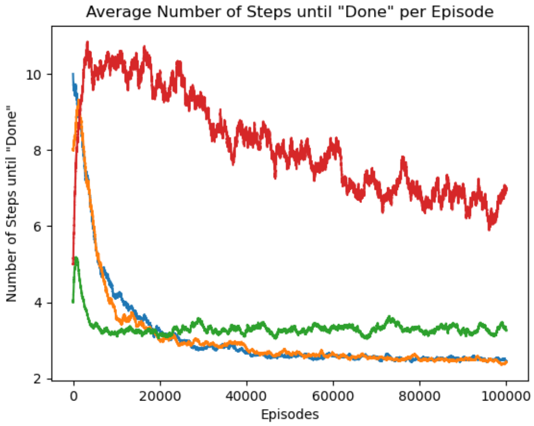


Fig. 18: Number of Moves Comparison.

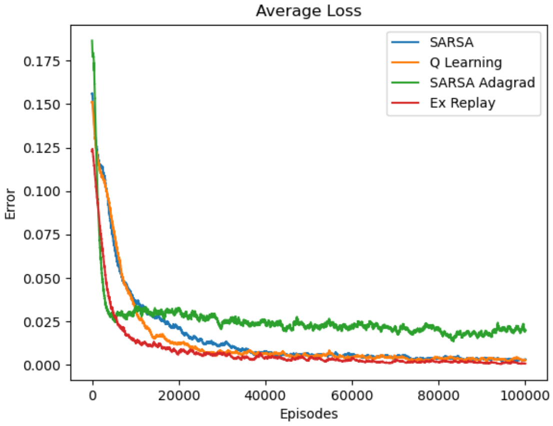


Fig. 19: Loss Comparison.