Chess Game – Reinforcement Learning

Karin Thommen
Department of Informatics
University of Zurich
Switzerland
karin.thommen@uzh.ch

I INTRODUCTION

There are many developments happening in the field of artificial intelligence and researchers try to find a way such that systems are able to learn and adapt human-like behaviors. One part of the trend within artificial intelligence and machine learning is reinforcement learning (Sutton & Barto, 2020, p. 4). This is another machine learning method beside supervised and unsupervised learning.
There exist multiple algorithms in the field of reinforcement learning. However, this paper focuses on SARSA and Q-learning.
The goal is to summarize the functionality of both algorithms and their advantages as well as their disadvantages. Moreover, both algorithms are implemented in the context of a chess game where the template code of the game is given and must be adapted by adding the learning algorithms. Afterwards, the results of each algorithm get analyzed and compared in the results section.

II THEORETICAL BACKGROUND

Reinforcement learning is a machine learning method where an agent learns to solve problems by interacting with the environment and receiving rewards for its actions (Russell & Norvig, 2020, pp. 789-790). The goal of an agent which is trained on reinforcement learning is to maximize the expected sum of rewards (Russell & Norvig, 2020, pp. 789-790). This method has some advantages over the other ways to train an agent. This includes the fact that the computer scientists who are training the agent must not know every detail about the environment in which the agent has to perform its actions since there is no need for labels (Russell & Norvig, 2020, pp. 789-790). Hereby, the system does not know which actions it should perform but must discover by trial-and-error (Sutton & Barto, 2020, pp. 1-2). One of the main challenges is that the learner must decide whether to exploit experiences from before to get rewards or to explore to optimize the actions in the future (Sutton & Barto, 2020, p. 2).
There exist different reinforcement learning algorithms and approaches on how a system can be developed and trained.
One approach is temporal-difference learning where an agent can learn from raw experience without a model of the environment and updates estimates without waiting for a final outcome (Sutton & Barto, 2020, p. 119). For this work two different temporal-difference learning algorithms are used: the Q-learning algorithm and the SARSA algorithm. The goal is to implement a chess game with an agent which works with reinforcement learning and uses SARSA or the Q-learning algorithm.
To understand and explain further equations and descriptions of the codes, I will define some variables which are used in this report. They can be found in the appendix.

III METHODS

3.1 SARSA Algorithm

First, the structure of the SARSA algorithm is discussed. This algorithm is an on-policy temporal-difference learning control method (Sutton & Barto, 2020). Sutton & Barto (2020, p.100) describe such algorithms the following way: “On-policy methods attempt to evaluate or improve the policy that is used to make decisions [...]”.
First, a table with so-called Q-values for each state-action pair is arbitrarily initialized. Afterwards, there is a loop in range of the defined number of episodes that the learning process has to run through. The following explanations are now described in application of the chess game and are based on the algorithm description of Sutton and Barto (2020, p. 130). Inside the loop, the current state is initialized (i.e., the chessboard and its pieces), and an action is chosen from the Q-table using the policy, which is an epsilon-greedy policy in our case.
Afterwards, another loop follows which terminates as the chess game is over (i.e., checkmate or draw happens). During the game, the agent performs the chosen action and observes its reward. Depending on the resulting next step, another action is derived from the Q-table, using the policy. Afterwards, the Q-table is updated on the current state and action.

The equation on updating the Q-table in the case of SARSA goes as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \eta\, [r_{t+1} + \gamma Q(S_{t+1}, A_{t+1} - Q(S_t, A_t)]$$

(Sutton & Barto, 2020, pp. 129, variables adapted according to template code), (Vasilaki, 2022)

The name SARSA is coming from the updating rule that uses the following elements of events: $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ (Sutton & Barto, 2020, p. 129). The Q-table is only updated until before the final state because for the terminal state the next action and state is zero (Sutton & Barto, 2020, p. 129). Overall, SARSA takes the action a in state s following the policy (Sutton & Barto, 2020, p. 129).

3.2 Q-Learning Algorithm

Now take a closer look at the Q-learning algorithm. In contrast to SARSA, Q-learning is an off-policy method. Sutton & Barto (2020, p.100) describe this term as follows: "[...] Off-policy methods evaluate or improve a policy different from that used to generate the data."

As for SARSA, the following description of the algorithm is based on Sutton and Barto (2022, p. 131). The structure of the algorithm is similar to the one of SARSA, as first a Q-table is arbitrarily initialized, and it goes through a defined number of episodes. For each episode, the state is initialized, and another loop follows until the game is over. An action is chosen out of the Q-table, using the policy. The action is taken, the resulting reward is acknowledged, and the next state becomes known. This is followed by the update of the Q-table which is the fundamental discrepancy from SARSA:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \eta \left[r_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

(Sutton & Barto, 2020, pp. 129, variables adapted according to template code), (Vasilaki, 2022)

3.3 Comparison of the algorithms

The main difference between SARSA and Q-learning is that SARSA is an on-policy and Q-Learning an off-policy method. Moreover, SARSA performs an action while it follows the current policy to update the Q-values (Sutton & Barto, 2020). In contrast, Q-learning is choosing a greedy action. Greedy action means that it is doing the action that gives the maximum Q-value for the current state (Gautam, 2022). In Q-Learning the optimal action-value function is approximated, without considering the policy (Sutton & Barto, 2020, p. 131).

The difference of the two algorithm is described in the cliff walking example in the book of Sutton and Barto (2020). They showed how SARSA and Q-learning perform on the task of finding their way from one point to another along a cliff. For all steps, there is a reward of -1. If the agent steps off the cliff, it gets a reward of -100 (Sutton & Barto, 2020, p. 132). It follows, that the optimal way is to follow the path along the cliff without stepping over because it would be the shortest way Nevertheless, it would be safer to get from one point to another with further distance from the cliff (Vasilaki, lecture slides, 2022). This is the method that SARSA chooses (Sutton & Barto, 2020, p. 132). In contrast, Q-learning learns the optimal policy and goes along the cliff with the risk of falling off which leads to a bigger punishment in the reward (Sutton & Barto, 2020, p. 132).
So the overall performance of the agent based on Q-learning will be worse (Sutton & Barto, 2020, p. 132). This is due to the fact that Q-learning learns the optimal policy (Sutton & Barto, 2020, p. 132). Summarizing, it can be said that Q-learning tends to ignore the higher negative reward, while SARSA tries to optimize the reward and is avoiding the risk of getting a punishment.
This example only works if epsilon, the probability to take a random action in epsilon-greedy policy, is large enough. As smaller epsilon gets, the methods will converge on using the optimal policy (Sutton & Barto, 2020, p. 132).
One advantage of Q-learning is that the agent does not has to know the environment and the effect of an action but is nevertheless able to compare the utility of a chosen action (S.Manju & M.Punithavalli, 2011).
The main drawback is that Q-learning does not improve much on the rewards and needs much time to reach optimal Q-values (S.Manju & M.Punithavalli, 2011). In contrary, SARSA will follow a more optimal policy and learn faster because it takes the next action into account (Arabnejad, Pahl, Jamshidi, & Estrada, 2017).

3.4 Implementation

The implementation is based on the template code which was given by the assignment requirements. Moreover, the code snippets from the random agent and the code from the Lab sessions of the course "Introduction to Reinforcement Learning" by Prof. Eleni Vasilaki were used as a basis for the implementation.
The main part of the implementation stays the same for both algorithms, Q-Learning and SARSA. First, the neural net is initialized as well as the function for the epsilon greedy policy.

Afterwards, the hyperparameters get defined and the number of episodes which is set to 100’000.
In the part of the initialization the Q-table gets generated with random values. Moreover, a Q-look-up-table is generated. How the table is accessed can be found in the appendix.

At the start of each episode the epsilon is decayed by the given decaying speed β and the game is initialized with a starting state. Moreover, the possible actions for this specific state are saved in a variable. While the game is not done, the agent looks up the possible actions for the specific state and the corresponding Q-values. The Q-values, epsilon and the allowed actions get passed to the function which is responsible to choose an action according to the epsilon greedy policy. Due to the chosen action, the agent makes the next step and outputs the next states, features, futures reward and if the game is done.
While the game is not finished, the agent keeps updating the Q-table according to the equations of SARSA and Q-Learning.
When the episode ends, the reward and the number of moves that were needed to end the game, get saved in variables.
To visualize the results, plots are produced according to the saved variables after the training process. The visualizations of the performance of the algorithms on the default parameters are discussed in the results section of this report.

IV RESULTS

To reduce the noisiness of the plots, an exponential moving average with a smoothing factor $\alpha = 0.001$ is used on all the visualizations. The results are produced over 100’000 episodes and the comparison of SARSA and Q-learning is based on the training on the same default parameters.

4.1 Results with SARSA

Figure 1 shows the reward per episode.
As we can see in the visualization, the reward levels off at the beginning, drops slightly after around 30’000 episodes, increases slowly but is levelling off at the same amount of reward overall. As the reward increases, the needed number of

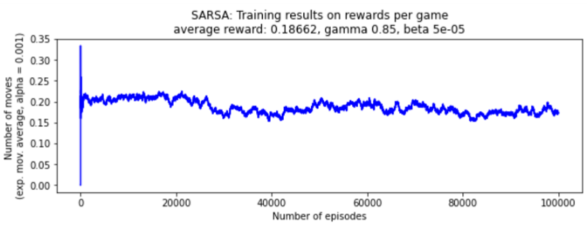


Figure 1 Results of SARSA: average reward over episodes (exponential moving average of a = 0.001), beta = 0.00005, gamma = 0.85

steps to end a game by draw or checkmate decrease. This can be shown by the second plot on figure 2.
It follows that even if the reward and the number of games won is low, the agent is able to reduce the number of steps in order to end a game.

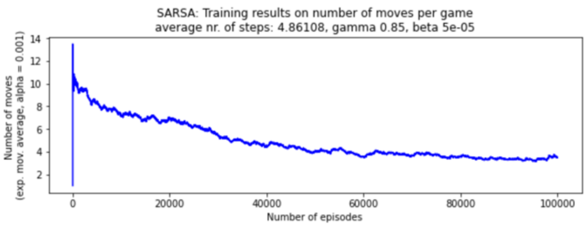


Figure 2 Results of SARSA: number of steps per episodes (exponential moving average of a = 0.001), beta = 0.00005, gamma = 0.85

4.2 Results with Q-Learning

The agent based on Q-learning achieved an average reward of 0.1683 and has used 5.1795 steps on average.
Figure 3 shows that the agent reached the average value for the reward after a few episodes. Afterwards, the reviews stay on a similar level and reaches its plateau.

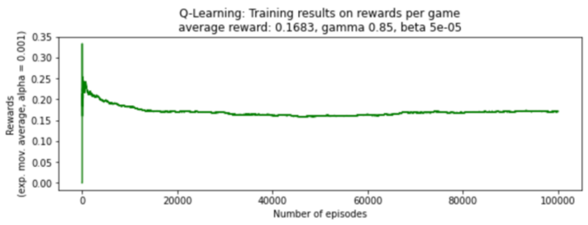


Figure 3 Results of Q-learning: average reward over episodes (exponential moving average of a = 0.001), beta = 0.00005, gamma = 0.85

In contrast, the number of steps gets reduced with increasing training time. Starting with around 10 moves to end a game, the agent is able to decrease this number to a value around 4.

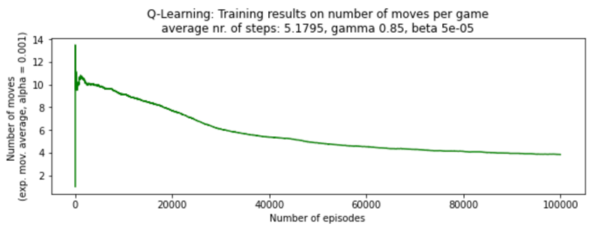


Figure 4 Results of Q-learning: number of steps per episodes (exponential moving average of a = 0.001), beta = 0.00005, gamma = 0.85

4.3 Comparison of both algorithms

Now the results of both algorithms should be compared. It is noticeable that SARSA performs better regarding the rewards since the average reward with 0.1866 is higher than the reward that the Q-learning agent received over the 100’000 episodes. There are other runs with different parameters that show similar results.

Moreover, SARSA seems to learn slowly but increases the average reward with increasing learning time. Q-learning in contrast reaches the plateau after a few episodes and increasing training time has not much impact on the process of optimizing the reward. Even with the exponential moving average, SARSA is a lot wigglier than Q-learning. This shows that SARSA does more exploration than Q-learning due to the on-policy method.

Both algorithms can reduce the moves they use to end a game in draw or checkmate over training time. However, SARSA is able to reduce the needed number of steps more than Q-learning does over the training time.

4.4 Impact of discount factor γ and decaying speed β

As a next step the impact of the hyperparameter will be analyzed. For better understanding the discount factor and the decaying speed of epsilon, I have explored different values for these parameters. The different results for SARSA and Q-Learning were stored in a table to compare them in a further step. The default values are marked in gray color.

The first table shows the results for different parameters used on the SARSA algorithm. The best average reward of 0.24 was achieved if the discount factor γ was set to 0.9 and was combined with a decaying speed β of 0.2 which is much higher than the initial β = 0.00005. The average number of steps which was needed to end the game was not on its lowest achievable value for this setting. For the training where the number of steps was the lowest, the reward was quite low which could mean that the agent brought the game to a draw quickly.

discount factor γ	**beta β**	**average reward**	**average number of steps**
0.85	0.00005	0.18662	4.86108
0.1	0.00005	0.20929	3.91514
0.001	0.00005	0.21537	3.861
0.9	0.01	0.2268	5.62125
0.85	0.01	0.21678	5.22858
0.2	0.9	0.19247	2.94067
0.9	0.2	0.24468	5.30311
0.9	0.9	0.16349	5.56193
0.8	0.00001	0.17649	4.71375
0.2	0.01	0.23043	3.3385

Table 1 Different parameters for SARSA

The second table shows the results for the exploration of different variables with Q-learning in a similar manner as in table 1.

discount factor γ	**beta β**	**average reward**	**average number of steps**
0.85	0.00005	0.1683	5.1795
0.1	0.00005	0.22861	4.13519
0.001	0.00005	0.21403	3.84629
0.9	0.01	0.24452	5.87946
0.85	0.01	0.23801	4.84546
0.2	0.9	0.17012	2.92138
0.9	0.2	0.18873	5.61714
0.9	0.9	0.15327	5.3519
0.8	0.00001	0.15127	5.38389
0.2	0.01	0.22938	3.26541

Table 2 Different parameters for Q-learning

Regarding both tables, we can see that Q-learning is able to get higher results with a higher discount factor γ and a higher β. SARSA got the highest reward in similar settings. Since the agent is not achieving the best performance, the results of the training on the mentioned parameters are different but on a lower level.

V Conclusion

Overall, it can be said that the decaying speed of epsilon β and the discount factor γ have significant impact on the performance of both algorithms, Q-Learning and SARSA.
The agent does not perform well which is due to the missing connection to the deep neural network. To improve the agent and its ability to achieve checkmate in a chess game, SARSA and Q-Learning should be implemented in deep way such that the process of learning and the update of the Q-values happens based on a neural net.
Moreover, I trained the agent several times to test different hyperparameters to explore their performance. It was noticeable that the performance of the agent is different from one training loop to another.
I assume that this is due to the randomness of the initial states. Consequently, the speed of learning or the increase of the reward over training time seems to depend on the initial states of the chessboard and the pieces that were given to the agent. For this reason, I set a seed such that the results are more or less reproducible.
Moreover, during debugging I faced the problem of wrongly initialized games. In these cases, there were more pieces on the chessboard, than there should be. This means that sometimes there were for example two opponent's kings, or two queens initially placed. In these cases, the agent threw an error, stating that the state cannot be found in the Q-table. As a solution, I have implemented an if-statement that states whether the sum of the initially placed pieces is 6 (1 – King, 2 – Queen, 3 – Opponent's King).

VI References

Arabnejad, H., Pahl, C., Jamshidi, P., & Estrada, G. (2017). A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling. *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 64-73.

Gautam, A. (2022). *Introduction to Reinforcement Learning (Coding SARSA) — Part 4, Medium*. Abgerufen am März 2022 von https://medium.com/swlh/introduction-to-reinforcement-learning-coding-sarsa-part-4-2d64d6e37617#:~:text=SARSA%20is%20an%20on%2Dpolicy,stands%20for%20the%20acronym%20SARSA.

Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A modern approach* (Vol. 4th edition). Boston: Pearson.

S.Manju, & M.Punithavalli. (2011, 02). An Analysis of Q-Learning Algorithms with Strategies of Reward Function. *International Journal on Computer Science and Engineering, Vol. 3*.

Sutton, R. S., & Barto, A. G. (2020). *Reinforcement Learning: An Introduction* (Vol. second edition). London, England: The MIT Press.

Vasilaki, E. (2022). Introduction to Reinforcement Learning. *Lecture Slides, Spring Semester February 2022.*

VII Appendix

7.1 Glossary

a: single action
s: state
r: reward
q: Q-value
Q: Q table
t: time step ($t + 1$ indicates next time step)
A_t: action or state at given time step
S_t: state at given time step
$Q(S_t, A_t)$: value in Q-table for state-action pair
γ: discount-rate parameter
ϵ: probability to take a random action in epsilon-greedy policy.
η: learning rate
β: decaying speed of epsilon
α: smoothing value for exponential moving average

7.2 Parts of the code

7.2.1 Epsilon Greedy Policy – Function:

This function is different from the Labs code because in the task of performing in a chess game, only specific actions are allowed in a specific state. It follows, that only Q-values from allowed actions can be chosen by the described function. So the Q-values for actions to be not considered are changed to very small numbers.

```
def EpsilonGreedy_Policy(Qvalues, allowed, epsilon):
    N_a=np.shape(Qvalues)[0]
    rand_value=np.random.uniform(0,1)
    rand_a=rand_value<epsilon

    if rand_a==True:
        a=np.random.randint(0,N_a)

        while not a in allowed:
          # if a is not an allowed action, do as
            much as random choices
          # until an allowed action is choosen
          a=np.random.randint(0,N_a)

    else:
        a=np.argmax(Qvalues)
        while not a in allowed:
          # if the max Q value is not an allowed
            action, set value temporary
          # to a very low value and choose the next
            max q value instead.
          #  repeat until the highest q value of an
             allowed action is found
          Qvalues[a] = -10000
          a = np.argmax(Qvalues)

    return a
```

7.2.2 Accessing and updating the Q-table

The Q-table consists of two parts. One part is the Q-table where the Q-values for a specific state and action (state-action pairs) are stored. The second part is a look-up table that saves all states in a dictionary and assigns an index to them. Each index stands for a specific row of the Q-table. In this case it is possible to look up specific states in the Q-table to choose an action.

```
while Done==0:  ## START THE EPISODE

   # GET STATE NUMBER
   k = list(Q_lookup.keys()) #look up the keys
         (index of specific state)
   v = list(Q_lookup.values())#look up the
         values (states)
   S_lookup = S.tolist() # convert the state
         to a list for comparison

   if S_lookup in v: # check if the given
         state of the game is already saved

         position = v.index(S.tolist())# get
                position of the state in the list
         state = k[position] # access state
                number via position in list

   else:  # state is not in list
         Q_lookup[known] = S_lookup # save the state
                to the list
       state = known #set current index as state nr.
       known += 1

   a,_=np.where(allowed_a==1) # find allowed actions

   Qvalues = Qtable[state] # get values at specific
         state, not needed, just for debugging

   a_agent = EpsilonGreedy_Policy(Qvalues, a,
         epsilon_f) # choose current action with
                epsilon greedy

   S_next,X_next,allowed_a_next,R,Done=
         env.OneStep(a_agent) # do step, find next
                state and allowed actions
```

7.3 Reproducibility of the results

As mentioned above, each training result in slightly different values for the average reward and number of steps because of the randomness factor of the initialization of the chessboard and the pieces. For this case, a seed is used, such that the results are reproducible. The code is runnable via the submitted Notebook.

7.3.1 How to run the code

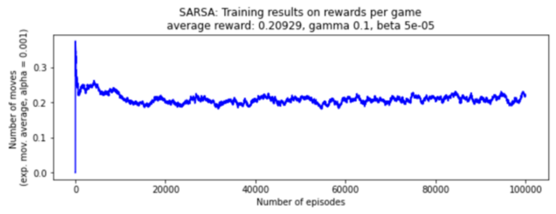
1. Import the .py files given in the assignment
2. Import all libraries
3. Initialize the environment
4. Do Initialization-Part
5. Run Sarsa-Part for Sarsa OR Q-Learning Part

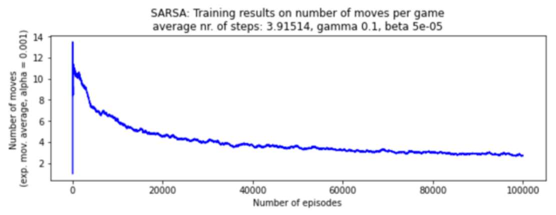
Remark: If you want to run Q-Learning after running Sarsa, the initialization has to be done again in order to reset everything.

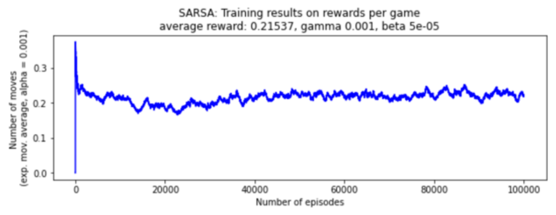
The code can be found here:
https://github.com/KarinTho/introRL-assignment

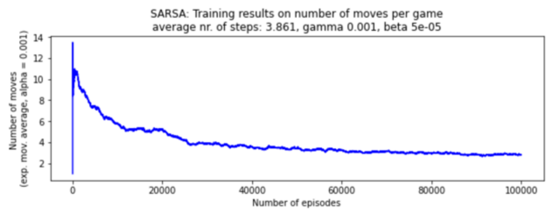
7.4 Plots of the different runs for different parameters

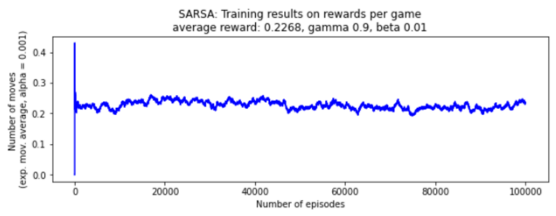
7.4.1 SARSA

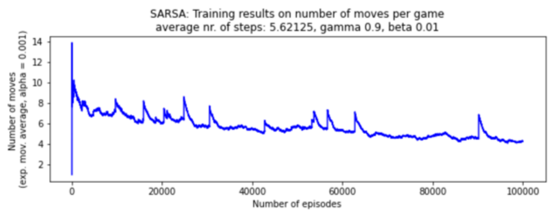


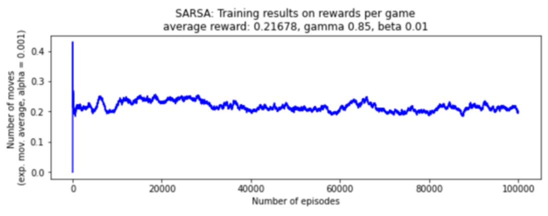


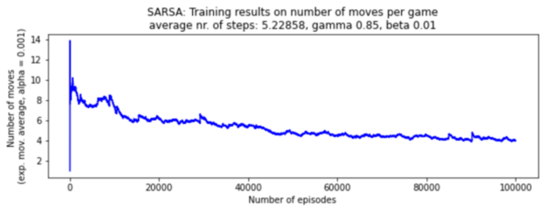


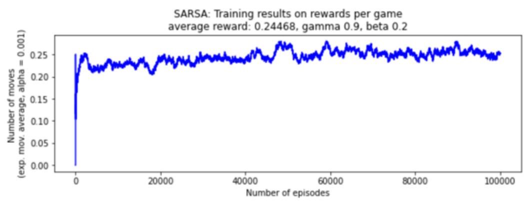


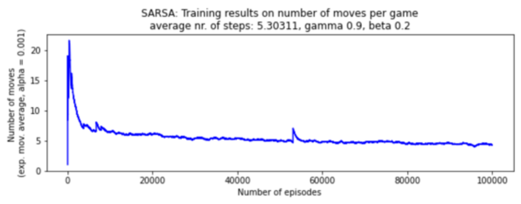


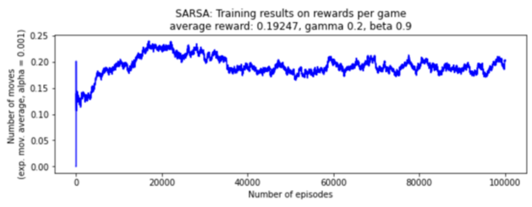


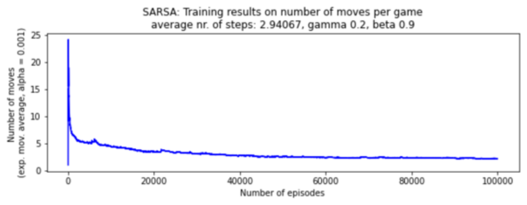


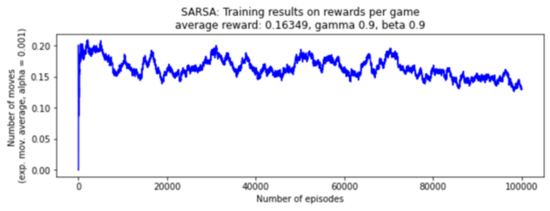


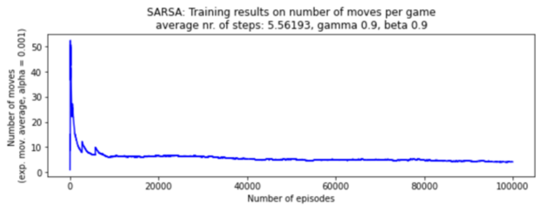


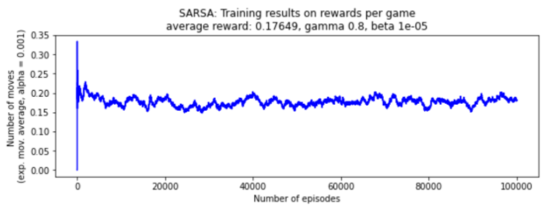


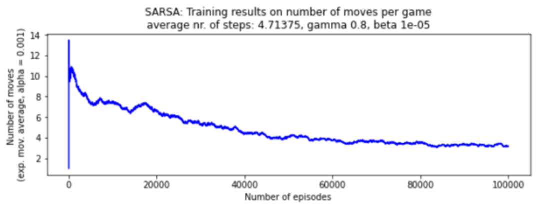


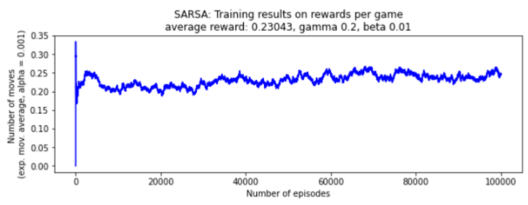


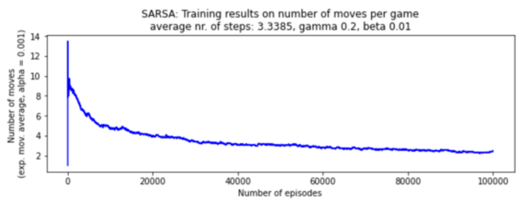




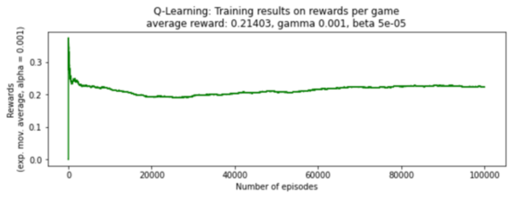


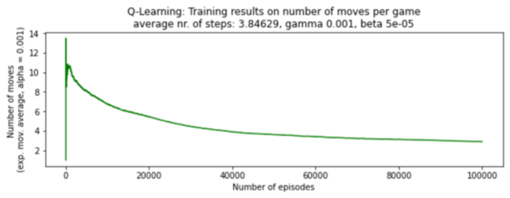


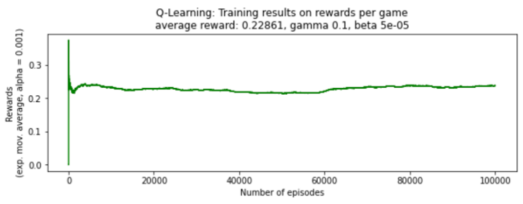


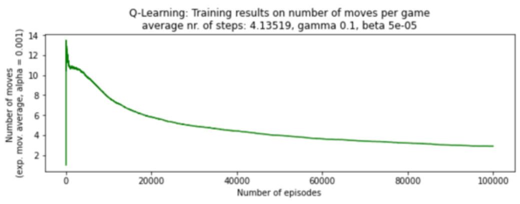


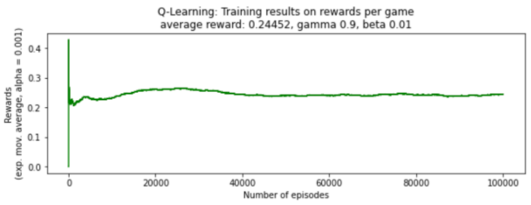
7.4.2 Q-Learning

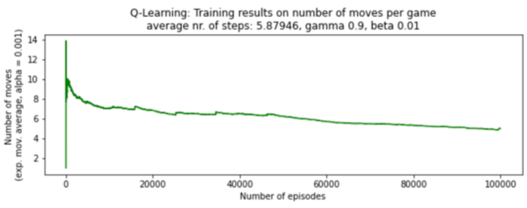


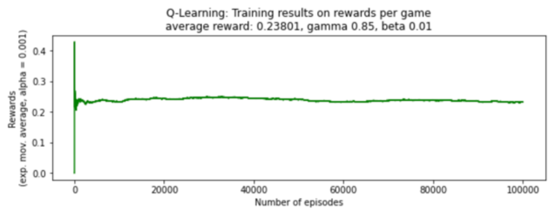


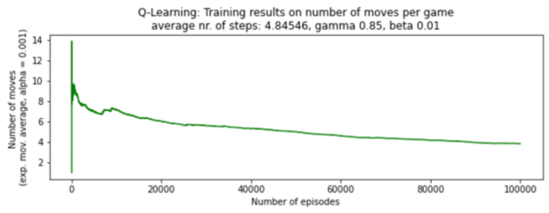


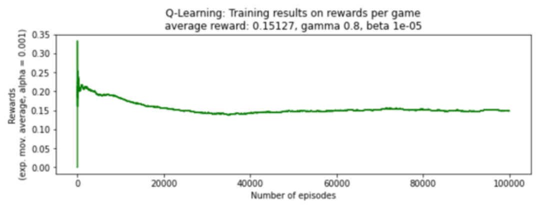


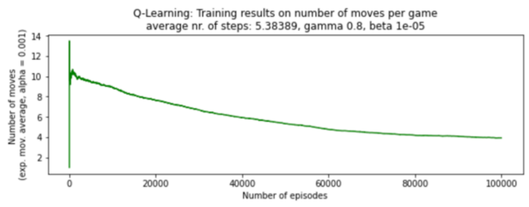


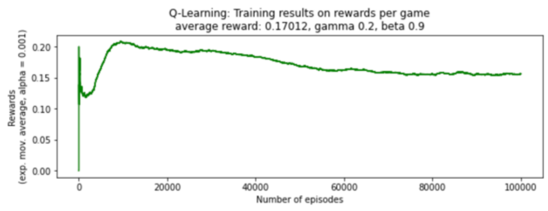


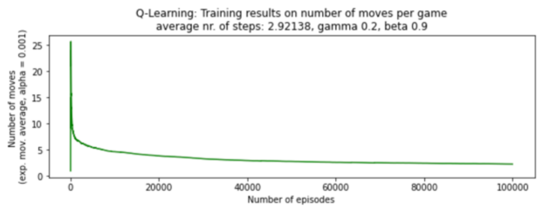

Q-Learning: Training results on rewards per game
average reward: 0.23801, gamma 0.85, beta 0.01
Rewards
(exp. mov. average, alpha = 0.001)
Number of episodes

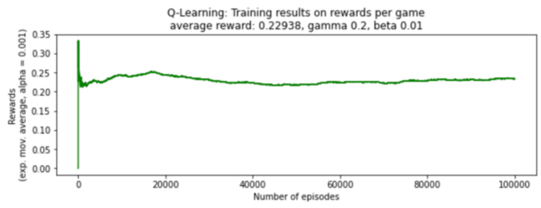

Q-Learning: Training results on number of moves per game
average nr. of steps: 4.84546, gamma 0.85, beta 0.01
Number of moves
(exp. mov. average, alpha = 0.001)
Number of episodes

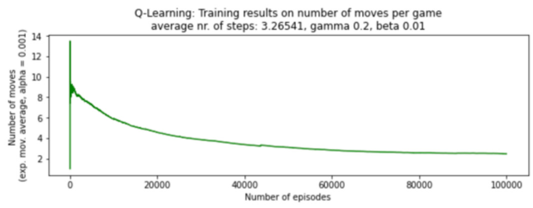

Q-Learning: Training results on rewards per game
average reward: 0.15127, gamma 0.8, beta 1e-05
Rewards
(exp. mov. average, alpha = 0.001)
Number of episodes

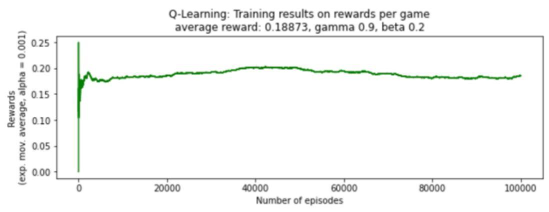

Q-Learning: Training results on number of moves per game
average nr. of steps: 5.38389, gamma 0.8, beta 1e-05
Number of moves
(exp. mov. average, alpha = 0.001)
Number of episodes

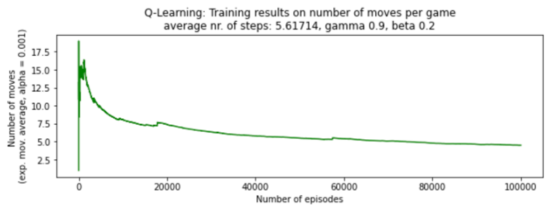

Q-Learning: Training results on rewards per game
average reward: 0.17012, gamma 0.2, beta 0.9
Rewards
(exp. mov. average, alpha = 0.001)
Number of episodes

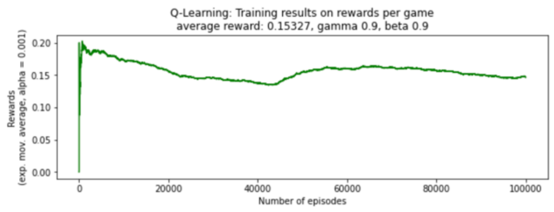

Q-Learning: Training results on number of moves per game
average nr. of steps: 2.92138, gamma 0.2, beta 0.9
Number of moves
(exp. mov. average, alpha = 0.001)
Number of episodes

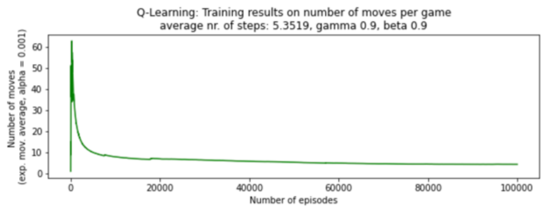

Q-Learning: Training results on rewards per game
average reward: 0.22938, gamma 0.2, beta 0.01
Rewards
(exp. mov. average, alpha = 0.001)
Number of episodes


Q-Learning: Training results on number of moves per game
average nr. of steps: 3.26541, gamma 0.2, beta 0.01
Number of moves
(exp. mov. average, alpha = 0.001)
Number of episodes


Q-Learning: Training results on rewards per game
average reward: 0.18873, gamma 0.9, beta 0.2
Rewards
(exp. mov. average, alpha = 0.001)
Number of episodes


Q-Learning: Training results on number of moves per game
average nr. of steps: 5.61714, gamma 0.9, beta 0.2
Number of moves
(exp. mov. average, alpha = 0.001)
Number of episodes


Q-Learning: Training results on rewards per game
average reward: 0.15327, gamma 0.9, beta 0.9
Rewards
(exp. mov. average, alpha = 0.001)
Number of episodes


Q-Learning: Training results on number of moves per game
average nr. of steps: 5.3519, gamma 0.9, beta 0.9
Number of moves
(exp. mov. average, alpha = 0.001)
Number of episodes