Q-Learning and SARSA for Chess

The University of Sheffield

Hyun Han - 160152230

COM 3240 - Eleni Vasilaki

Abstract

Deep reinforcement learning is implemented in chess game via Q-learning and SARAS algorithms. The reward per game and the number of moves per game are plotted to observe the relations between them. Exponential moving average is used to reduce to make the graph less noisy. Gamma and betta values are experimented with different values. Moreover, SARASA is experimented to compare it with Q-Learning. Lastly, the exploding gradients problem is introduced and the solutions follow.

1 Q-Learning vs SARSA

SARSA is an on-policy reinforcement learning algorithm in machine learning, where given the current state S, an action A is taken to output a reward R for the agent and move to the next state S' and takes another action. It updates the Q-value based on its policy to find the next action. which is a different point compare to Q-Learning. The algorithm assumes that the choice made is not maybe the best choice, so it seeks for the safer way. In general, it demonstrates a better performance than Q-learning. On the other hand, Q-learning is an off-policy and model free reinforcement learning algorithm. The algorithm use a greedy policy to choose an action that returns the maximum Q-value for the state, so it can learn the optimal policy directly. The algorithm is likely to choose more risky way, because it believes it makes the best choice per iteration. A disadvantage is that it has higher per-sample variance, so it may have a problem of converging. Model free algorithms take advantage of the trial and error approach to find the optimal Q-values, so it is not essential to have space enough to store all the combination of states and actions.

2 Q-Learning Implementation

The input layer is set to 50, as it has 3 pieces on 4x4 board and +2 for the degree of freedom of the opponent king, as shown in Figure 1. And the output layer is set to 32, which is the number of possible actions. Weights and biases for the input layer and hidden layer are initialised and the weights are rescaled by the total number of connection between the two considered layers.

The allowed moves have different Q values and the index of the one with the highest Q-value is used again to return the index of its move. Figure 2 shows the implementation of epsilon greedy policy. If a random number is larger than a given threshold that gets smaller per iteration to have less probability to explore, the greedy policy is selected. If the number is smaller, a random action is picked.

The implementation of calculating Q-values of neurons in the output layer and the activation of nodes of the first layer can be seen in Figure 3. The weight is updated, only if the Heaviside

function returns one instead of zero. The weights are multiplied by a corresponding neuron and its bias is added. In addition, rectified linear units are used to modify minus values to zero.

Weights and biases should be updated to train the network, using backpropagation technique. The existing formula for calculating delta, weight and bias change is applied, as can be seen in Figure 4. The weight is updated, only if the Heaviside function (written in if statement in the code) returns one instead of zero. The weights between output and hidden layer are updated, and then the neurons between hidden layer and input layer are considered to multiply with their weights. The same methods are used for checkmate and draw, but if the match continues, the maximum of next Q value is considered in calculating delta value for the backpropagation from output layer to hidden layer. The weight and bias are updated accordingly.

Each time a game ends, the reward is stored and applied by moving exponential average technique with an alpha value of 0.0001. On the same note, if a game is not finished, the counter for move increases, each time a player make a move. The moves per game are also applied by the same smoothing technique above. Figure 5 plots the rewards per game and moves per game graph for Q-Learning (gamma = 0.85, beta = 0.00005). As shown in the figure, the graph shows a steep increase from 0 to 0.7 as an average reward score within 20,000 iterations. In the end, it reaches almost reaches 1, which means it is always been checkmated. However, the average moves per game shows volatile changes, as shown in Figure 6. It increases drastically after 60k iterations and then start decreasing after 70k, eventually reaching around 60 moves per game

3 Different γ and β

Gamma is a variable used in calculating delta for backpropagation from output layer to hidden layer. The default value 0.85 is modified to 0.15 in this experiment, which means the influence of the next Q-value has a smaller effect to the delta value. The Beta value determines the magnitude of discounting speed of epsilon per iteration. If the value is high then, the algorithm has less probability to explore but higher probability to exploit. For this experiment, the beta is changed from 0.00005 to 0.005, a hundred times bigger value. Hence, with these changed gamma and beta, the algorithm is likely to have less impact from the next Q-value and to focus on more exploitation than exploration for the epsilon greedy policy. As shown in Figure 7, the average rewards per game end up reaching 0.6, which is lower performance than the default gamma and beta setting experimented before. It is expected to perform better with higher influence for the next Q-value and more exploration stage. The moving average of moves per game increases up to 90 at about 20k iterations, but goes down to a single digit number in the end after a bit of fluctuation in the middle.

4 SARAS Implementation

It needs to apply epsilon greedy policy once again to obtain the best Q-value of the next move. Apart from this point, the other processes are exactly same as Q-Learning process. Figure 9 shows the moving average of rewards go beyond 0.8 same as Q-Learning, but it shows a more gradual increase over 100k iterations. A narrow change of moving average of moves per game is show in Figure 10. It reaches the highest moves per game at 20k iterations, and it gradually decreases afterwards down to around 6. This explains the reason SARSA takes less time to return a result than Q-Learning.

5 Exploding Gradient

In deep neural network, gradients used for updating weights get bigger, so it essentially explodes gradients inside layers. This problem makes the network so unstable that prevents it from learning, and result in NaN weigh values which cannot be no longer changed or updated. Some solutions can resolve this problem. Firstly, the network model can be redesigned. Truncated backpropagation through time process a sequence one time step at a time and every k1 times steps it runs backpropagation through time for k2 time steps. Secondly, exploding gradients can be reduced by using the Long Short-Term Memory that can remove the inherent instability in the network and is a solution for sequence reduction. Thirdly, the size of gradients can be limited to a given threshold, which is known as gradient clipping. Lastly, the size of network weights can be changed and a penalty to the network loss function or large weight values can be re-evaluated.

RMSprop, an optimisation algorithm for neural network to deal with exploding or diminishing values of the weight, is a solution to the problem introduced here. The moving average of the squared gradients for each weight is the key idea of RMSprops. The gradient is divided by square root the mean square. In this way, the problem of varying gradients widely in magnitude can be resolved. Figure 11 shows the moving average of rewards per game in SARAS with RMSpro. The maximum score was around 0.275. It is assumed maybe that the RMSprop is not working. The V shape moving average of moves peer game can be seen in Figure 12. The maximum moves are 300 and the minimum moves are still over 100. It is expected that the RMSprop is not applicable for this case, but because of the time constraint, a further experiment is limited.

6 Conclusion

The performance of Q-Learning and SARSA is similar in this 4x4 board chess game, but SARSA performs faster, showing lower moves per game. Reducing gamma and increasing beta shows worse performance. RMSprop needs to be improved. It is expected that the calculation used is not correct.

Appendix

```
n_input_layer = 50  # Number
n_hidden_layer = 200  # Numbe
n_output_layer = 32  # Number

W1=np.random.uniform(0,1,(n_hidden_layer,n_input_layer))
W1=np.divide(W1,np.matlib.repmat(np.sum(W1,1)[:,None],1,n_input_layer))
W2=np.random.uniform(0,1,(n_output_layer,n_hidden_layer))
W2=np.divide(W2,np.matlib.repmat(np.sum(W2,1)[:,None],1,n_hidden_layer))
bias_W1=np.zeros((n_hidden_layer,))
bias_W2=np.zeros((n_output_layer,))
```

Figure 1: Define the size of layers, weights and bias

```
#eps-greedy policy implementation
greedy = (np.random.rand() > epsilon_f)
if greedy:
    #a_agent = allowed_a[np.take(Q, allowed_a).tolist().in
    a_agent = allowed_a[np.argmax(np.take(Q, allowed_a))]
else:
    a_agent = np.random.choice(allowed_a)          #pick
```

Figure 2: Epsilon greedy policy

```
# Neural activation: input layer -> hidden layer

out1 = np.maximum(0, W1.dot(x)+bias_W1)

Q = np.maximum(0, W2.dot(out1)+bias_W2)
# act1 = np.dot(W1,x) + bias_W1
```

Figure 3: Calculating Q values and hidden layer neurons.

```
# bias_W1 += (eta * out1delta)

out1delta = np.dtype(np.complex128)
#Backpropagation, same as before but this time the n
if ((np.dot(W2[a_agent], out1)) > 0):
    out2delta = (R - Q[a_agent])
    W2[a_agent] += (eta * out2delta * out1)
    bias_W2 += (eta * out2delta)
    if (np.sum(np.dot(W1, x)) > 0):
        out1delta = np.dot(W2[a_agent], out2delta)
        W1 += (eta * np.outer(out1delta, x))
        bias_W1 += (eta * out1delta)
```

Figure 4: Backpropagation and Q-Learning to update the parameters in the network.

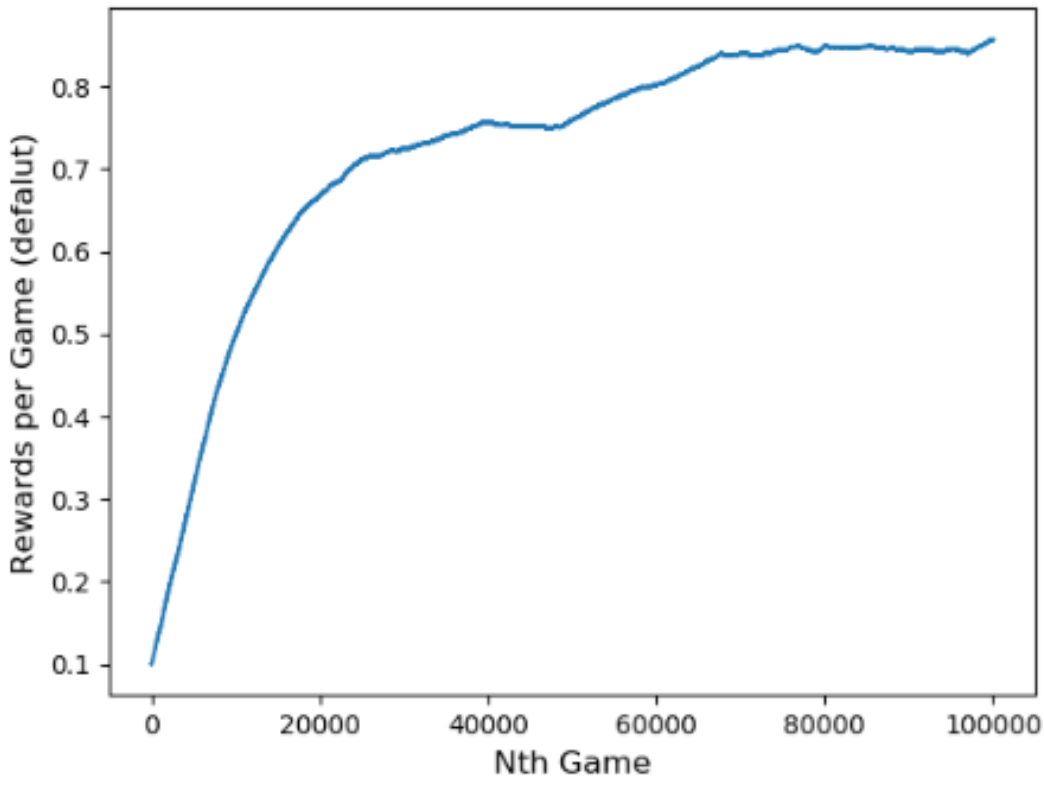

Figure 5: Moving average of rewards per game in Q-Learing

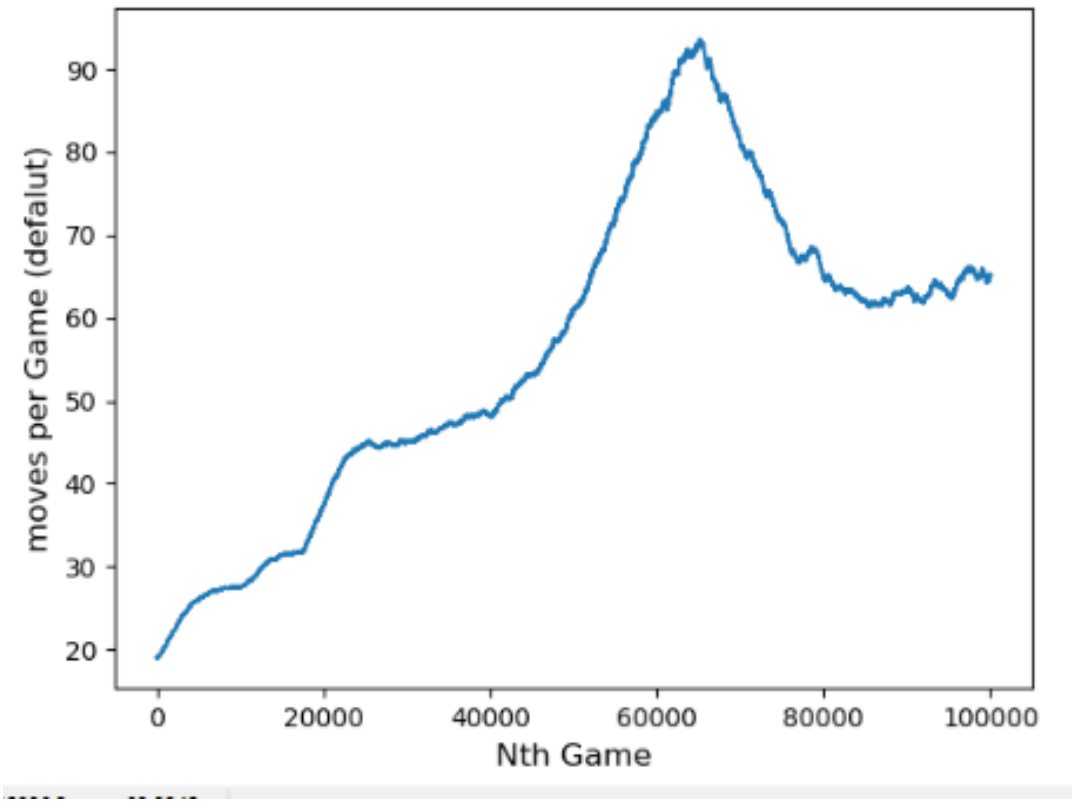

Figure 6: Moving average of moves per game in Q-Leering

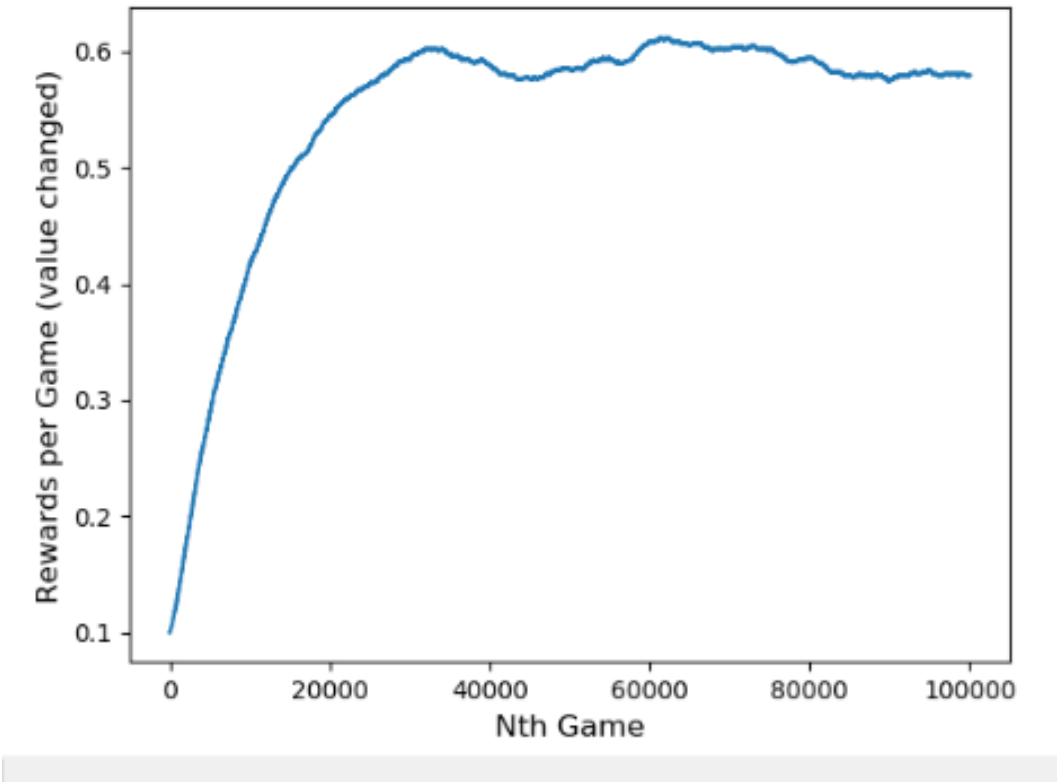

Figure 7: Moving average of rewards per game in Q learning (gamma = 0.15, beta = 0.00005)

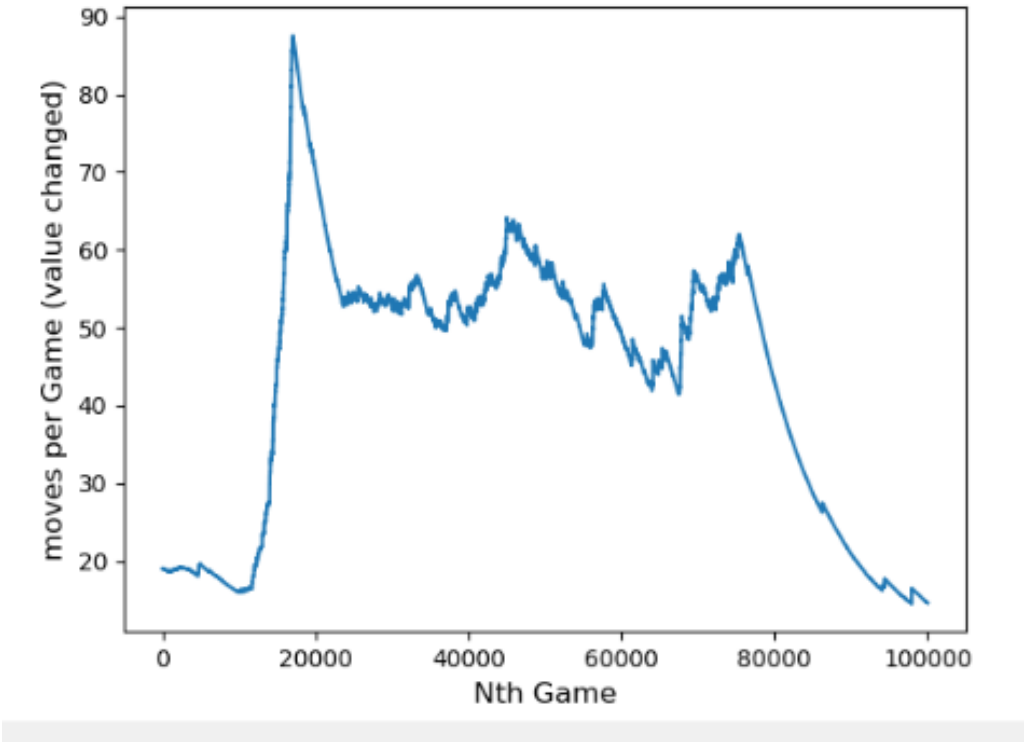

Figure8: Moving average of moves per game in Q-Leering(gamma = 0.15, beta = 0.00005)

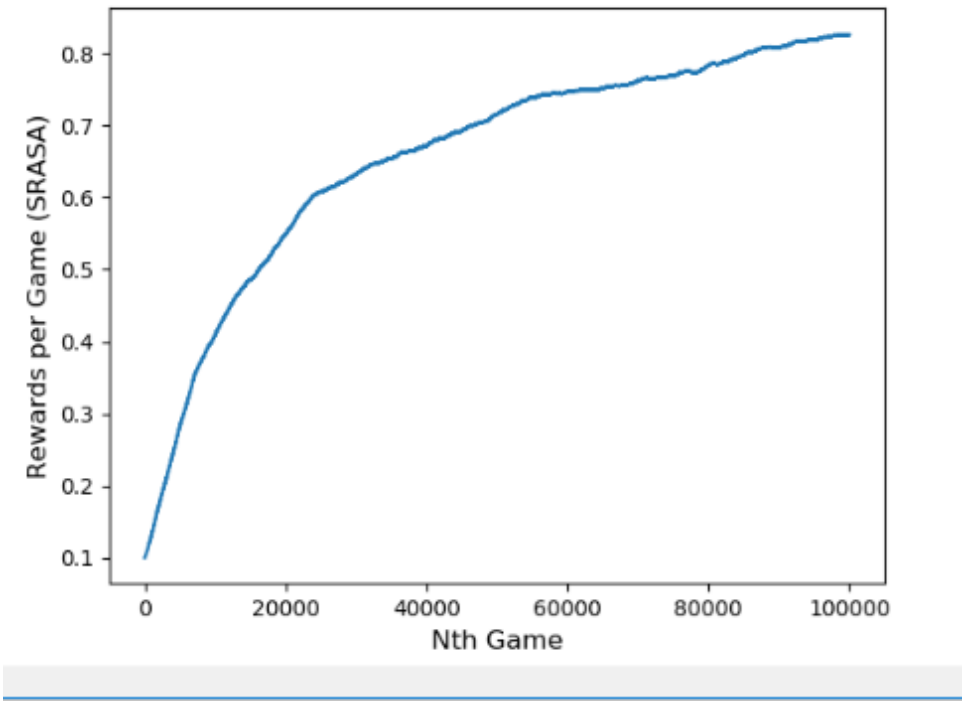

Figure9 : Moving average of rewards per game in SARSA

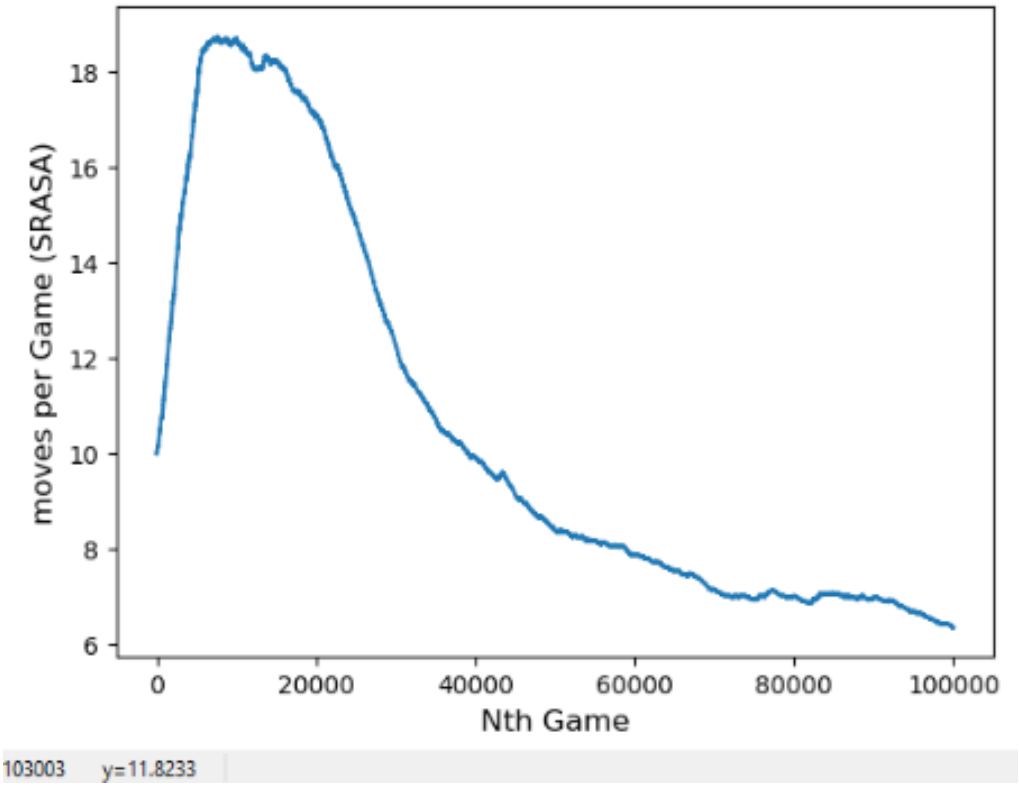

Figure 10: Moving average of moves per game in SARSA

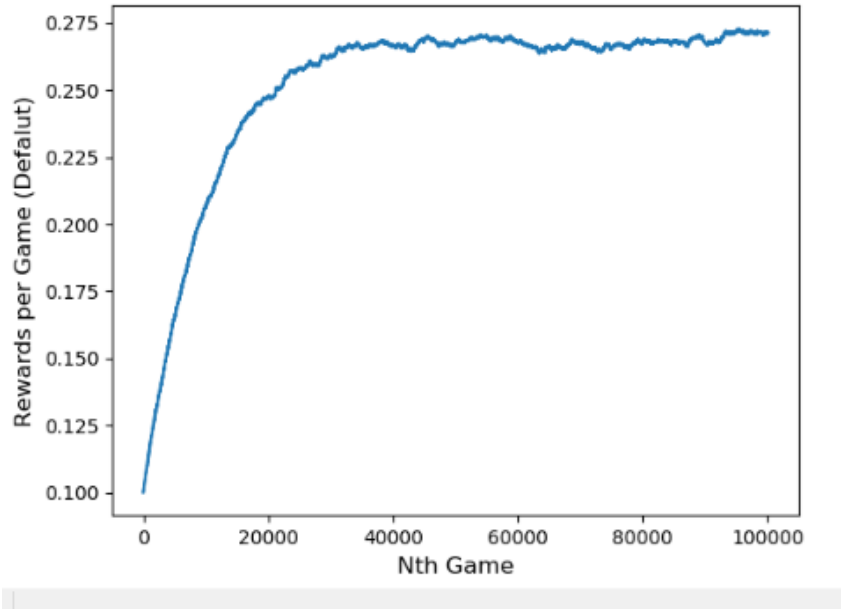

Figure 11: The moving average of rewards per game in SARSA with RMSprop

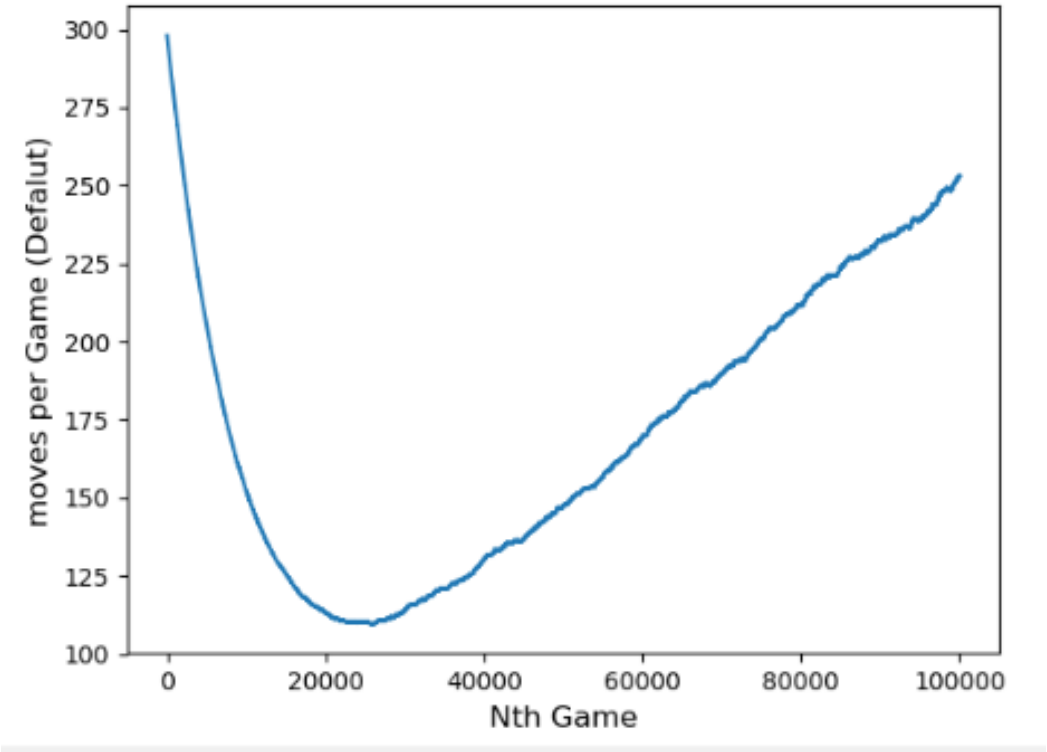

Figure12: The moving average of moves per game in SARSA with RMSprop