Reinforcement Learning Assignment

COM3240 Adaptive Intelligence
https://github.com/charalambosG-9/com3240_assignment

Charalambos Georgiades[a]

[a]*Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield S1 4DP*

March, 2023

Abstract

SARSA and Q-Learning are two model-free reinforcement learning algorithms that aim to learn optimal policies for decision-making in an environment. In this paper, we introduce and implement the two algorithms to train an agent that learns to play chess while also evaluating their performances and suitability for this type of an application. We also analyze how the performance of these algorithms changes as their parameters are tuned.

1. Introduction

The idea behind reinforcement learning was identified as "a learning system that wants something, that adapts its behavior in order to maximize a special signal from its environment" [3]. It is one of three basic machine learning paradigms, the other two being supervised and unsupervised learning. It differs from the more widely studied supervised learning in that there is no presentation of input/output pairs and that on-line performance is important. Instead, after choosing an action, the agent is told the immediate reward and the subsequent state but is not told which action would have been in its best long-term interests. It is necessary for the agent to gather useful experience about the possible system states, actions, transitions and rewards to act optimally [1].

The first and oldest of the two algorithms analysed in this paper is Q-Learning, an off-policy TD control algorithm introduced in 1989 [4] with a convergence proof being presented in 1992 [5]. An off-policy algorithm uses data collected from exploratory behavior to update the value function which in our case it gathers using epsilon-greedy policy. In Deep Reinforcement Learning (DLR), the goal of the agent is to maximize its total reward and the algorithm does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward.

The second and newest algorithm is SARSA (State–Action–Reward–State–Action) which was proposed by Rummery and Niranjan as Modified Connectionist Q-Learning (MCQ-L) in their paper "On-line Q-learning using connectionist systems" [2] before Richard S. Sutton proposed the current name in his aforementioned paper [3]. Contrary to the Q-Learning algorithm, SARSA is an on-policy algorithm meaning that it learns the optimal policy by using the same policy for exploration and for updating the value function. This means that the agent learns about the consequences of its own actions and the target policy and behavior policy are the same. The name reflects the fact that the main function for updating the Q-value depends on the current state of the agent, the action the agent chooses, the reward the agent gets for choosing this action, the state that the agent enters after taking that action, and finally the next action the agent chooses in its new state.

As stated previously, the main difference between the two algorithms is that SARSA is an on-policy algorithm, while Q-Learning is an off-policy algorithm. SARSA is generally more stable than Q-Learning and less prone to overestimation biases, but it can take longer to converge. Q-learning on the other hand, is generally more efficient in exploring the state-action space, but it may be more sensitive to the choice of the exploration-exploitation trade-off and may suffer from overestimation biases.

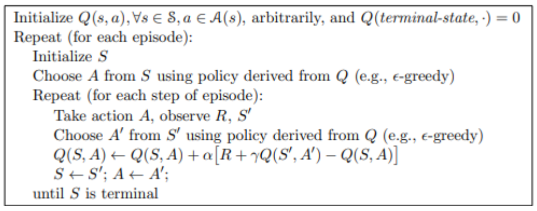
Both algorithms will be implemented to train a deep reinforcement learning agent that learns to play chess on a reduced board of size 4x4 and only 3 pieces: a king, a queen and an opponent's king. The agent will try to checkmate the opposing king while it moves randomly across the board. The agent is rewarded with 1 point if a game ends with checkmate and 0 points if it ends in a stalemate. In such a deterministic environment with a relatively small number of exploration options, we expect the Q-Learning algorithm to converge to the optimal faster than SARSA but given enough number of episodes, both algorithm should present relatively similar end results.

2. Methodology

2.1. Implementing SARSA

The SARSA algorithm was employed first to create the chess playing agent. Initially, the weights and biases of a hidden layer and the output layer are chosen randomly and the starting values of epsilon for the epsilon-greedy policy, the value affecting how fast epsilon decreases, the discount factor, learning rate and number of episodes parameters are initialized. Depending on if eligibility traces are enabled, the degree of influence of past state-action pairs is initialized and the learning rate is modified. The code iterates over the specified number of episodes where in each one the agent selects actions based on the current state of the game and updates the Q-values of the state-action pairs based on the reward it received. An episode ends after the game ends in a tie or a victory which award 0 and 1 points respectively.

At the beginning of the episode, the Q-values are calculated using forward propagation so that the first action can be selected using epsilon-greedy policy. A for-loop is then initiated that



```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Figure 1: *SARSA: An on-policy TD control algorithm (Source: Sutton and Barto [3]).*

repeats until the end of the episode. It first performs forward propagation to calculate the Q-values of all possible actions from the current state. Subsequently, the code transitions to the next state based on the selected action, receives the reward for that action and checks if the episode has ended before updating the eligibility trace. If the episode is done, the weights and biases for the hidden and output layers are updated using backward propagation and so is the eligibility trace. The reward and number of steps are saved so that an exponential moving average can be calculated. If the episode is not done, forward propagation is performed again to obtain the Q-values of all possible actions from the next state, an action is selected based on an epsilon-greedy policy with the Q-values obtained, and the Q-values are updated based on the reward received and the Q-values obtained from the next state. The weights and eligibility trace are also updated. The code then transitions to the next state, updates the selected action and increments the action counter.

2.2. Changing the discount factor and rate of decay of epsilon

Two parameters that significantly affect the performance of the algorithm are the discount factor γ and the rate of decay of epsilon β.

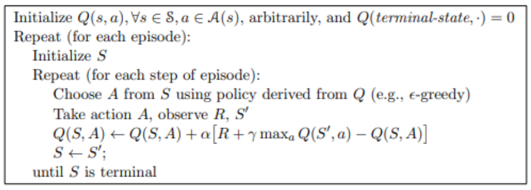
The discount factor γ determines the importance of future rewards. In the given algorithm, the discount factor affects the update rule for the Q-values. Since it determines the degree to which the agent should value immediate vs future rewards. A high γ value would result in the agent being more patient and value long-term rewards more, while a low one would make the agent more short-sighted and value short-term rewards more.

The rate of decay of epsilon or β, is used to control the rate at which the exploration probability epsilon decreases as the number of episodes increases. In other words, it controls the trade-off between exploration and exploitation with a higher value favoring exploitation and a lower value exploration. In our implementation, the value of epsilon for a specific episode is calculated by dividing the original value by 1 plus β multiplied by the number of the current episode n.

$$epsilon = epsilon_0/(1 + \beta * n) \qquad (1)$$

2.3. Implementing Q-Learning

Three modifications were made to the code to accommodate for the addition of the Q-Learning algorithm. The first modification made is forward propagation does not occur outside of the for-loop but the next move is always chosen at the start of the for-loop based on the forward propagation which occurs there. The second modification is the subsequent action chosen is not made by the epsilon-greedy policy but based on which state will yield the maximum reward. The final modification is that the selected action is not updated at the end of the for-loop since it will be re-calculated at its beginning.



```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

Figure 2: *Q-Learning: An off-policy TD control algorithm (Source: Sutton and Barto [3]).*

3. Results

3.1. Evaluating the SARSA algorithm

The performance evaluation for the SARSA algorithm can be found below. To ensure that the results were accurate and consistent, the agent was trained to play chess for 100000 episodes 8 times and the mean values from all of the runs were calculated. The process was repeated once more without eligibility traces to establish their effect on the performance of the algorithm. From the graphs obtained and shown in **Figure 3** it is obvious that our implementation of the SARSA algorithm is correct and works as intended. Initially, we can see a higher average number of moves and a lower reward average but as time progresses and the epsilon value decreases, we can see that the average number of moves per game slowly decreases and that the average reward converges towards 1. The mean of rewards of all 100000 games was 0.873 while the mean number of steps was 3.813. If the eligibility traces have been deactivated, the algorithm takes significantly longer to converge and the means drop down to 0.721 and 6.341 respectively. It is also worth mentioning that implementing eligibility traces to the algorithm not only improves the overall performance but also reduces the overall time required for the algorithm to finish.

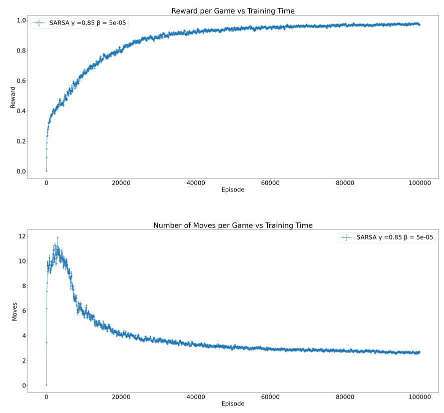


Figure 3: *SARSA EMA for rewards and number of steps vs training time*

3.2. Effect of altering the parameters

To investigate if altering the parameters β and γ would cause the effects that we speculated on the SARSA algorithm, we re-trained the agent several times using the same starting weights and biases while altering one or both of the aforementioned parameters each time.

Rewards/Moves	$\beta = \mathbf{0.0005}$	$\beta = \mathbf{0.00005}$
$\gamma = \mathbf{0.75}$	0.900/3.538	0.852/3.581
$\gamma = \mathbf{0.85}$	0.919/3.578	0.880/3.881

Table 1: *Table comparing the results of the varying parameters.*

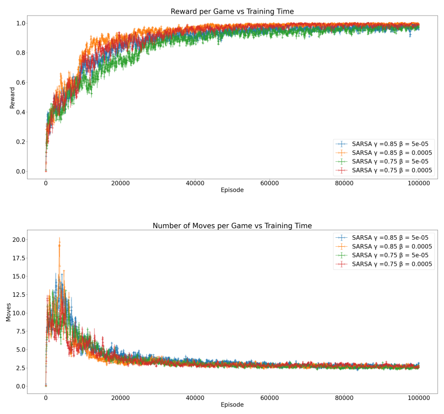


Figure 4: *SARSA EMA for rewards and number of steps vs training time with varying β and γ values.*

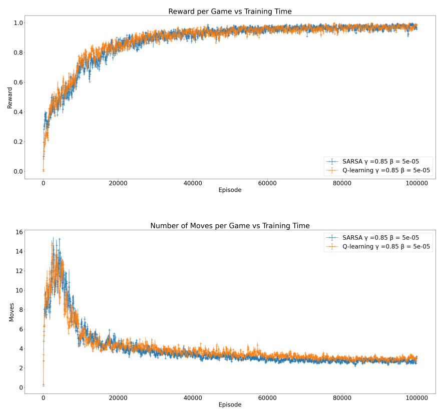


Figure 5: *SARSA and Q-Learning EMA for rewards and number of steps vs training time*

The results from tests conducted seem to support our previous statements. When the variable β is increased, the algorithm converges faster to the optimum and in less moves. This is because the value of epsilon decreases in a much faster rate which in result favours exploitation rather than exploration. In **Figure 5** we can clearly see the impact that β makes on the algorithms, especially in the earlier stages where the number of the episode is still relatively small. The discount factor γ also affects the algorithm in the way that we predicted. A higher γ also results in the algorithm converging faster as it values long term rewards more which in turn also increases the number of moves needed in the earlier stages.

3.3. SARSA vs Q-Learning

Having also implemented Q-Learning, we tasked both algorithms with training an agent over 100000 games with the same starting weights and biases to compare their performance.

	ET	**MoR**	**MoS**
SARSA	Yes/No	0.880/0.686	3.881/6.406
Q-Learning	Yes/No	0.885/0.708	4.057/6.194

Table 2: *Comparison between the results of the two algorithms.*

The two algorithms provide very similar results, making it extremely difficult to contrast them. The graphs and table do confirm our suspicions that the Q-Learning algorithm would be more suitable for this project, however SARSA performs only marginally worse. There are a few possible explanations as to why this might be the case. The first one is that a large enough number of episodes were supplied, thus both algorithms were able to successfully converge. The second possible explanation is that the epsilon value decreases too fast thus reducing the amount of exploration done by SARSA.

4. Conclusions

In this paper, we have successfully implemented two Deep Reinforcement Learning algorithms, SARSA and Q-Learning, and compared their performance when it comes to training an agent to play chess. We also analyzed the effect certain parameters have on the performance of these algortihms.

References

[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore, *Reinforcement learning: a survey*, 1996.

[2] G. A. Rummery and M. Niranjan, "On-line q-learning using connectionist systems", (1994).

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction* (MIT Press, 1998).

[4] C. J. Watkins, "Learning from delayed rewards", PhD thesis (King's College, Cambridge, 1989).

[5] C. J. Watkins and P. Dayan, "Q-learning", Machine Learning **8**, 279–292 (1992) `10.1007/BF00992698`.

Appendix A. Replicating the results

```python
repeat_multiple_times = 1 # THE NUMBER OF TIMES THE EXPERIMENT IS REPEATED (TO
    GET MORE ACCURATE RESULTS)

# Set to 1 for reproducibility
if repeat_multiple_times == 1:
    np.random.seed(93)
```

Listing 1: Replicating the results.

To replicate the results in this paper make sure that the repeat_multiple_times variable is set to 1.

Appendix B. Code Snippets

```python
h1 = np.matmul(W1, X_next) + b1

# Apply the ReLU activation function
x1 = np.maximum(0, h1)

# Compute Qvalues
Qvalues1 = np.matmul(W2, x1) + b2

a1, _ = np.where(allowed_a_next == 1)

if learning_type == "SARSA":
    a1_agent = EpsilonGreedy_Policy(Qvalues1, epsilon_f, a1)
else:
    # Since EpsilonGreedy_Policy retrieves the index of
    # the maximum of the Qvalues, we can use it for Q-learning
    a1_agent = EpsilonGreedy_Policy(Qvalues1, 0, a1)

# Backward propagation
delta2 = R + gamma * Qvalues1[a1_agent] - Qvalues[a_agent]

eta_delta2 = eta * delta2

W2[a_agent] = W2[a_agent] + eta_delta2 * x1
b2[a_agent] = b2[a_agent] + eta_delta2

delta1 = np.dot(W2[a_agent], delta2) * (x1 > 0)

W1 = W1 + eta * np.outer(delta1, X)
b1 = b1 + eta * delta1

if eligibility_trace:
    W2 = W2 + eta_delta2 * np.outer(e, x1)
    b2 = b2 + eta_delta2 * e
    e = gamma * lamb * e
```

Listing 2: The difference in updating the SARSA and Q-Learning algorithms

```python
R_save[n] = (1 - eta) * R_save[n - 1] + eta * np.copy(R)
N_moves_save[n] = (1 - eta) * N_moves_save[n - 1] + eta * np.copy(i)
```

Listing 3: Exponential moving average based on the equations in the slides.

The exact equation found in the slides is:

$$Q_{n+1} = (1 - \eta)Q_n(s, a) + \eta r \tag{B.1}$$

```python
plt.rcParams.update({'font.size': 24})
plt.figure(figsize=(30, 12))
for i, R_save in enumerate(all_R_save):
    errors = 2 * np.std(R_save) / np.sqrt(N_episodes)
    plt.errorbar(np.arange(N_episodes), R_save.flatten(), errors, 0, elinewidth
   =2, capsize=4, alpha=0.8, errorevery=50)
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.title("Reward per Game vs Training Time")
plt.legend(legends)
plt.savefig('EMA_Reward.png')

plt.rcParams.update({'font.size': 24})
plt.figure(figsize=(30, 12))
for i, N_moves_save in enumerate(all_N_moves):
    errors = 2 * np.std(N_moves_save) / np.sqrt(N_episodes)
    plt.errorbar(np.arange(N_episodes), N_moves_save.flatten(), errors, 0,
   elinewidth=2, capsize=4, alpha=0.8, errorevery=50)
plt.xlabel('Episode')
plt.ylabel('Moves')
plt.title("Number of Moves per Game vs Training Time")
plt.legend(legends)
plt.savefig('EMA_Moves.png')
```
Listing 4: Plotting the results with error bars.