Introduction to Reinforcement Learning

Luis Torrejón machado
luis.torrejonmachado@uzh.ch
(Student Id: 21-721-097)

https://github.com/luis-tm/reinforcement-learning

I. Introduction

The goal of reinforcement learning is to train a system, called an agent, to complete a task by interacting with the environment in which the task takes place. The agent obtains rewards through this interaction, which allows it to learn the optimal way to complete the task. Therefore, reinforcement learning algorithms seek to maximize rewards when performing a task [1]. However, agents do not know the rewards a priori, so agents have to estimate them through experience [2]. These reward estimates are called Q – values and the behavior of the agent while interacting with the environment is called policy (π).

Within the existing reinforcement learning methods, there are the so-called time-difference (TD) learning methods, which allow predicting quantities that depend on future values. These methods allow agents to learn from environments where rewards are not immediate, but where they need to perform a sequence of actions before getting a final reward. A key feature of these methods is that they do not need a model of the environment's dynamics, but they can learn from raw experience. Within this category are the algorithms known as Q – learning and SARSA. In this assignment, an agent is trained by combining SARSA and Q – Learning algorithms with deep learning to checkmate a king using a king and a queen on a board of 4x4 dimensions. Deep learning will help us to obtain the different Q – values as the output of an artificial neural network, thanks to their well-known function approximation properties [3, 4].

II. SARSA and Q – Learning

SARSA and Q – learning algorithms are structurally quite similar, though they show a different behavior. Agents using both SARSA and Q – Learning start in a given initial state and choose a possible action derived from the initial estimated Q-values following a behavior policy π (e.g., epsilon – greedy). Chosen the action, agents take it to observe the next state and the Q – values which corresponds to this next state. The general form of the updating rule can be expressed as follows:

$$Q_{old} \leftarrow Q_{old} + \alpha[learned\ value - Q_{old}], \qquad (1)$$

where α is the learning rate and the *learned value* considers not only the reward after performing one action, but also the estimated reward of a future step. The difference between SARSA and Q – learning lies in the calculation of the *learned value*.
In SARSA algorithm, reward estimates are updated by the following expression:

$$Q(S_t, A_t) \leftarrow \\ Q(S_t, A_t) + \alpha[R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)], \qquad (2)$$

where S_t and A_t are the state and the action of the agent at the time step t, and γ is the discount factor, which represents the importance of future rewards for the agent. The general formula considers the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, which gives the algorithm its name.
On the other hand, Q – Learning updates the estimated rewards as follows:

$$Q(S_t, A_t) \leftarrow \\ Q(S_t, A_t) + \alpha\left[R_t + \gamma \max_{\mathrm{a}} Q(S_{t+1}, a) - Q(S_t, A_t)\right] \qquad (3)$$

SARSA therefore follows the same policy for learning as it does for actions selection (behavior policy). However, Q – learning learns independently of the agent's actions, since it is acting by following a behavior policy π, but it is learning by choosing the highest reward (greedy policy). In other words, SARSA is an on-policy algorithm and Q – learning is off-policy. Regarding the update rule both algorithms follow, we can see that Q – learning is more optimistic, directly learning the optimal policy while SARSA learns a near – optimal policy at the same time it explores [5]. This exploit – explore trade-off in SARSA will depend on the value of ϵ, if we choose an epsilon-greedy policy. In view of this, we can say that SARSA is a more conservative algorithm. Due to the optimism of Q - Learning, situations where there are negative rewards close to the optimal path may make this algorithm a bad choice, with SARSA being a better option. The cliff walking from [2] is an example which clearly illustrates this difference between SARSA and Q – Learning.

Let us now see how these algorithms fit in with deep learning. Since the goal of our agent is to estimate Q - values, we can use an artificial neural network to approximate them. The update rules defined in (2) and

(3) allow us to define loss functions to minimize for SARSA:

$$E(w) = \frac{1}{2}[r + \gamma Q(s', a'; w) - Q(s, a; w)]^2 \qquad (4)$$

and for Q – Learning:

$$E(w) = \frac{1}{2}\left[r + \gamma \max_{a} Q(s', a; w) - Q(s, a; w)\right]^2 \qquad (5)$$

The loss functions are defined as function of the parameters of the network w, whose approximation is the objective of the deep algorithms. From (4) and (5) we can derive the updating rules for the weights and the bias terms of the neural network by applying backpropagation. Considering a single hidden layer and a RELU activation function, the resulting formulas are as follows:

$$\Delta w_{ij}^2 = \eta(\delta_{model}^2)H(h_i^2)x_j^1, \qquad (6)$$

$$\Delta w_{jl}^1 = \eta\left(\delta_j^1\right)x_l^0, \qquad (7)$$

$$\Delta b_{ij}^2 = \eta(\delta_{model}^2)H(h_i^2), \qquad (8)$$

$$\Delta b_{jl}^1 = \eta\left(\delta_j^1\right), \qquad (9)$$

$$\delta_j^1 = \delta_j^1 w_{ij}^2 H\left(h_j^1\right), \qquad (10)$$

where δ_{model}^2 depends on the used algorithm with value ($learned\ value$ - Q_{old}), x^n corresponds to the output of the layer n and H is the Heaviside function, obtained by deriving the RELU activation function.

III. MODELS AND METHODS

Deep SARSA and deep Q – Learning algorithms have been implemented in Python Jupyter notebooks to train the agent to checkmate the enemy king. Both models include an artificial neural network with a single hidden layer containing 200 units. The weight matrices for the network have been randomly initialized following a uniform distribution with values between -1 and 1 and normalized by the number of neurons contained in the adjacent layers. The bias terms for both layers have been initialized to zero vectors.

Q – values are obtained as output of the artificial neural network which is fed with an input vector with contains the following degrees of freedom of the game: the position for each piece on the board, the possible movements for the enemy king, and a digit for the possible outcomes: draw or checkmate. The dimension of the output layer corresponds with the number of possible actions which the agent can make. After each step, the weights and bias terms of the neural network are updated following the formulas (6) – (10). For both SARSA and Q-Learning, an ϵ-greedy policy has been followed to choose the agent actions from the Q-values. All training loops have been run for 100000 games. Each game is played until the agent checkmates the enemy king or until there is a draw. The environment gives a reward of 1 in case of checkmate and 0 otherwise. Table 1 shows the implemented pseudocode for both algorithms. Implementation details are shown in the appendix.

Deep SARSA Algorithm

```
1. For each game:
   1. Get initial state of the environment
   2. Obtain initial Q – values from the network
   3. Pick an action following $\epsilon$-greedy policy
   4. While not checkmate/draw:
      1.Take the chosen action
      2. Obtain Q – values from the network
      3. Pick a new action following $\epsilon$-greedy policy
      4. Update parameters of the network
      5. Save values for the next iteration
```

Deep Q – Learning Algorithm

```
1. For each game
   1. Get initial state of the environment
   2. While not checkmate/draw:
      1. Obtain Q – values from the network
      2. Pick an action following $\epsilon$-greedy policy
      3. Take the chosen action
      4. Obtain Q – values from the network
      5. Update parameters of the network
      6. Save values for the next iteration
```

Table 1. Implemented pseudocode for deep SARSA and deep Q-Learning. This code runs in each of the episodes in which the agent has been trained.

The model developed in the first place has been deep SARSA. Following the suggested hyperparameters, the initial values were $\epsilon_0 = 0.2$, $\beta = 0.00005$, $\gamma = 0.85$ and $\eta = 0.0035$. For SARSA, the value of γ was modified while maintaining the values for ϵ_0, β and η. Same was done with β, maintaining in this case ϵ_0, γ and η. Finally, Q – learning was implemented in order to compare the performance with SARSA. The comparison was done with the suggested parameters, since it was found that the result was optimal.

The obtained results have been represented in two different types of plots: reward and number of steps both as a function of the episodes. Raw representations of the data were very noisy and an exponential moving average using pandas has been applied.

For the following discussion, results from both SARSA and Q-Learning are presented in the first place, to discuss later the effects of changing the hyperparameters γ and β in SARSA.

IV. RESULTS

Results for both SARSA and Q-Learning algorithms are shown in Figure 2. The reward evolution is quite similar for both, with SARSA presenting a smoother learning due of its exploratory behavior. After 40000 episodes there is no significant difference between them, and the average reward for both reaches a plateau after 55000 episodes with value 0.97. More different is the evolution of the number of steps: though both models reach a minimum of steps after 30000 episodes, Q-learning shows a larger number of steps before this threshold and a much sharper drop. Comparing both representations, it is seen that both models have a drop in the number of steps after the same number of episodes, and after that both models approach its maximum reward average. The similar behavior between both models seems to be produced by the small dimension the board. I would expect a worse behavior for Q-Learning for environments with a larger number of pairs state-action, while SARSA most likely would maintain the smooth profile due to its exploratory behavior.

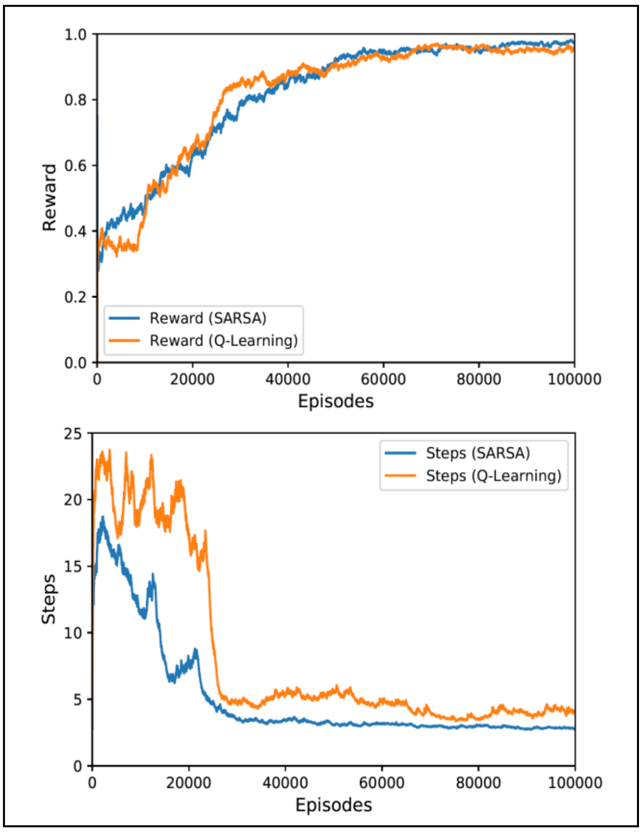

Figure 2. Reward and number of steps as function of the episodes for SARSA and Q-learning algorithms ($\epsilon_0 = 0.2$, $\beta = 0.00005$, $\gamma = 0.85$ *and* $\eta = 0.0035$).

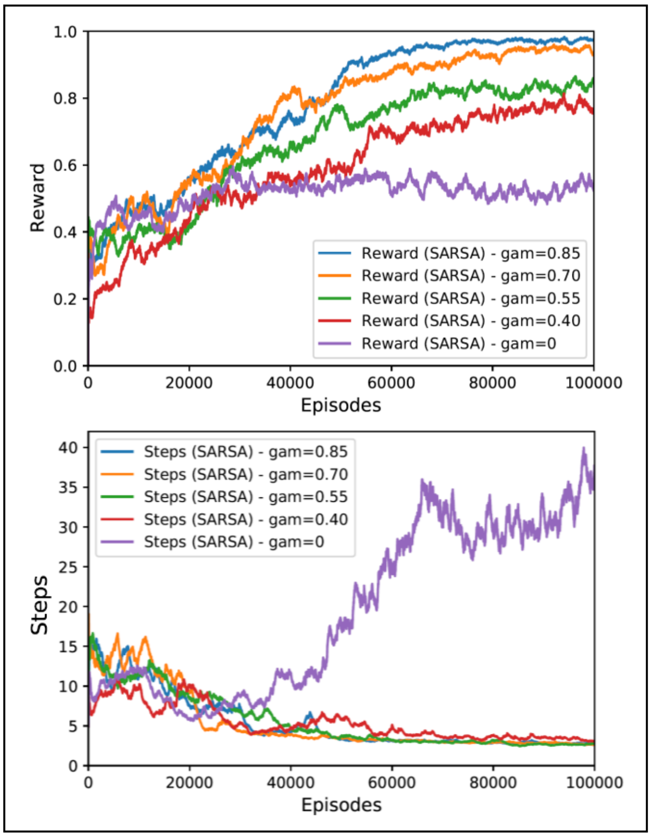

Figure 3. Reward and number of steps as function of the episodes for different gamma values for SARSA algorithm ($\epsilon_0 = 0.2$, $\beta = 0.00005$ *and* $\eta = 0.0035$).

Now we focus on how the discount factor γ affects the training of our agent. Figure 3 shows the results obtained for different values of the hyperparameter. We can see that by decreasing the value of γ, the maximum average reward achieved during training is shifted from the optimal value 0.97 when γ is 0.85 to 0.52 when γ has the minimum possible value, 0. By setting the discount factor to 0, we are eliminating the factor $Q(S_{t+1}, A_{t+1})$ from the updating rule of the model (2), which means that the agent does not consider future rewards, but only the immediate ones. In other words, out agent will only learn from movements with immediate checkmate. The effect of removing future rewards has also a great impact in the number of steps. Decreasing γ to 0.4 has a slight effect on the steps, but it does still converge as greater values. However, when $\gamma = 0$, the number of steps diverges. The obtained result for the reward for this extreme case is a surprisingly good result for our agent considering the nature of chess, in which pieces are moved considering future combinations. This fact can be explained by the small size of the board which reduces the criticality of not seeing future rewards due to the proximity of the checkmate from any state.

For the discussion of the effect of beta on the algorithm result, let's analyze the definition of ϵ:

$$\epsilon = \frac{\epsilon_0}{1+n\cdot\beta}, \quad (11)$$

where n is the number of episodes and ϵ_0 the initial value for ϵ. Therefore, higher values of β decrease ϵ at a lower number of episodes, implying less exploratory behavior from the beginning of the training. On the contrary, smaller values of β will approach ϵ to ϵ_0. When $\beta = 0$, ϵ will be constant independently of the number of episodes. This analysis is consistent with the results shown in Figure 4. We can see the smoother behavior of the curves when $\beta = 0$, because of the conservative behavior of the agent. However, the obtained reward is less than with the other β values because it continues exploring, even when all Q-values have been estimated. On the other hand, with the highest value of β, the reward reaches its maximum before due to the tendence of the agent to exploit. Because of the dependency of ϵ with the number of episodes, we can see how the reward lines for the non-zero β values meet when the number of episodes is high enough. Regarding the number of steps, the algorithm works better when $\beta = 0.00005$, because of its more exploratory behavior at the first games (after a large number of episodes, the agent will not explore anymore).

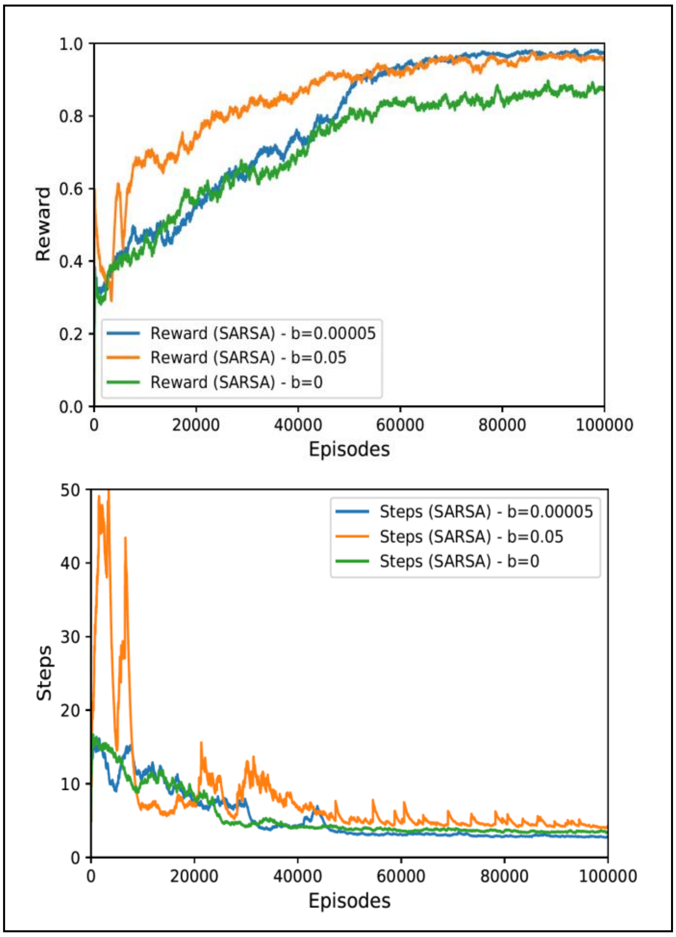

Figure 4. Reward and number of steps as function of the episodes for different beta values for SARSA algorithm. ($\epsilon_0 = 0.2,\ \gamma = 0.85$ *and* $\eta = 0.0035$).

V. REPRODUCIBILITY OF THE RESULTS

The implemented codes are available in the folder /notebooks of the following GitHub repository:

- https://github.com/luis-tm/reinforcement-learning

The folder contains separated projects for the algorithms SARSA and Q-Learning. To make the results reproducible, a seed has been added when initializing the parameters of the neural network. Running all cells of the Jupyter notebooks is enough to obtain the results and their graph representation for the optimal hyperparameter ($\epsilon_0 = 0.2$, $\beta = 0.00005$, $\gamma = 0.85$ and $\eta = 0.0035$). In case you want to get results for other hyperparameters, just modify them in cell 5 and run the whole notebook.

VI. CONCLUSIONS

In this assignment deep SARSA and deep Q-Learning have been implemented to train an agent to checkmate in a reduced chess game. After the whole training and for optimal parameters, both algorithms show a similar result in terms of gained reward. However, SARSA presents better results for the number of steps needed to checkmate. Within these results, we can conclude that SARSA behaves better for our environment. For higher dimensional boards, I would expect a greater difference between both algorithms, again with SARSA showing better results. The exploratory behavior of SARSA will make the agent to learn before the different states reaching convergence at a lower number of games.

Also, the influence of the discount factor and β have been analyzed. The results have shown that by decreasing γ the agent learns worse than with high values, since small γ "blind" the agent to see future values, which is critical for learning in our environment. Regarding β, the results have shown an optimal behavior with 0.00005. This value allows the agent to learn in the first episodes of the training, changing its behavior to exploit more when the number of episodes is large enough.

REFERENCES

[1] Eleni Vasilaki, Course material: "Introduction to Reinforcement Learning", http://staffwww.dcs.shef.ac.uk/people/E.Vasilaki/introduction-to-reinforceme.html

[2] Richard S. Sutton and Andrew G. Barto, “Reinforcement Learning: An Introduction”, http://incompleteideas.net/book/the-book-2nd.html

[3] “A beginner’s guide to deep reinforcement learning”, https://wiki.pathmind.com/deep-reinforcement-learning

[4] Mnih et al, “Playing Atari with Deep Reinforcement Learning”, https://deepmind.com/research/publications/ 2019/playing-atari-deep-reinforcement-learnin

[5] Ahmed Raafat, “Q – Learning vs SARSA (Model-Free RL algorithms”, https://www.linkedin.com/pulse/q-learning-vs-sarsa-model-free-rl-algorithms-ahmed-raafat/

VII. APENDIX

A. CODE IMPLEMENTATION

The resulting code for the assignment has maintained the provided skeleton. It has been modified the initialization of the parameters of the model and implemented the corresponding code for the training loop and the result plotting.

The initialization of the network for both algorithms is done in the same way. To make the results reproducible, a seed has been added before initializing the parameter of the artificial neural network.

```
np.random.seed(1)
…
# initialization of the network
…
W1 = np.random.uniform(-1,1,(N_h,
N_in))/(N_in+N_h)
W2 = np.random.uniform(-1,1,(N_a,
N_h))/(N_in+N_h)

bias_W1 = np.zeros((N_h,))
bias_W2 = np.zeros((N_a,))
```

For the epsilon – greedy policy, it has been used the code provided in the laboratories: given a list of Q – values and an epsilon, the index for a Q – value of the list is returned. Epsilon defines the probability that the agent explores.

```
# define the epsilon - greedy policy
def EpsilonGreedy_Policy(Qvalues, epsilon):

    N_a=np.shape(Qvalues)[0]
    rand_value=np.random.uniform(0,1)
    rand_a=rand_value<epsilon

    if rand_a==True:
        # random action with epsilon
        a=np.random.randint(0,N_a)
    else:
        # Greedy with probability epsilon
        a=np.argmax(Qvalues)
    return a
```

The following snippet of code shows how the previous policy is used in the implementation for SARSA and Q – Learning to choose an action, and the corresponding Q – value. First, the Q – values are obtained from the output of the neural network given the X (given by the environment).

```
x0 = X
h1 = np.dot(W1,x0) + bias_W1
x1 = np.maximum(h1, 0)
h2 = np.dot(W2,x1) + bias_W2
x2 = np.maximum(h2, 0)
```

The variable x2 contains the Q – values for the current parameters of the network. Then, we can use this value and the previous definition of the policy to choose an action and the corresponding Q – value:

```
# pick an action from the allowed ones
following the epsilon-greedy policy
allowed_a_index,_=np.where(allowed_a==1)
a = EpsilonGreedy_Policy(x2[allowed_a_index],
epsilon_f)
taken_action = allowed_a_index[a]
…
Q_a = x2[taken_action]
```

Now we show the core part of the implementation: the update rules for the parameters of the network, which follow the formulas defined in (6) – (10).

For the SARSA algorithm:

```
#update network parameters only for the
action neuron (SARSA)
delta = R + gamma*Q_a2 - Q_a

dW2 = delta*np.heaviside(h2[taken_action],
0)*x1
dW1 =
np.outer(delta*W2[taken_action,:]*np.heavisid
e(h1, 0), X)

W2[taken_action,:] = W2[taken_action,:] +
eta*dW2
W1 = W1 + eta*dW1

bias_W2[taken_action] = bias_W2[taken_action]
+ eta*delta*np.heaviside(h2[taken_action], 0)
bias_W1 = bias_W1 +
eta*delta*W2[taken_action,:]*np.heaviside(h1,
0)
```

For the Q-Learning algorithm, the definition of delta has been implemented in the following way:

```
#update network parameters only for the
action neuron (Q - learning)
delta = R + gamma*np.max(x2_next) - Q_a
```

This implementation is executed for each step which is not final. For final steps, the Q – value is equal to 0, and the implementation of the updating rule has been adapted to that.