Q-Learning and SARSA: A comparative study on Roulette-v0

Shahid Gulzar Padder
Eötvös Loránd University
Faculty of Informatics
Budapest

***Abstract*—Reinforcement Learning is finding its way in broader dimensions with the advent of time. Q Learning and SARSA are the building blocks for solving any problem in Markovian domain and thus enabling agent to perform optimally. To dive deeper into in this field classic games and Atari games play a significant role. In order to maximise score of an agent various parameter settings are performed. My approach is simple but robust in order to see the effect of varying the hyper parameters and other settings in order to achieve convergence. I have implemented Q and SARSA algorithms for solving Roulette-v0. I have also attached Jupyter notebooks for my approch.**

***Index Terms*—Reinforcement Learning, Q-learning, SARSA, AI Gym, Roulette-v0 solution.**

I. Introduction

Q LEARNING and SARSA are important algorithms and exhibit properties of asynchronous dynamic programming. These two Reinforcement Learning methods do not require information about the model. These two methods enable an agent to perform optimally in Markovian domain by facing the consequences of actions rather than building maps of the domain [1]. My motivation to select this problem: because it seems simple in working but it has a wide application ranging from robotics to marketing and much more.Policy can affect orientation of robot/customer base or next step in a virtual game etc. We first need to understand the environment we are dealing with. In this environment agent bets on a number and receives a positive reward if and only if the number is not zero and its parity matches the agent's bet. Moreover, agent can walk away from table and thus ending the episode. Fig 1 depicts a roulette game example.

Reinforcement learning(RL) finds many applications in diverse fields but there is still contradiction between exploration and exploitation strategy for action selection policy. On one hand SARSA has faster convergence but on the other hand Q-Learning has better overall performance. SARSA gets stuck in local minimum while Q learning takes longer time to learn.[2] Coming to a stronger and firm conclusion is out of scope for this brief assignment but these methods were used during solving the Roulette-v0 environment.These two methods do not require any model knowledge. Unlike Monte-Carlo we don't need to wait for end of complete episode. In these methods update is made after each step. For both cases we have ϵ-greedy policy which is necessary for exploration. In both cases for every episode from current state s, action a is taken from *s* to new state *s'* keeping track of reward *r*. While ϵ-greedy policy is followed given by current Q-value function the action a is taken. And here we need to update *Q(s,a)* In SARSA, it is done by choosing an another action a' while following same current policy given above and $r+\gamma Q(s',a')$ as target.(2)

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a))] \quad (1)$$

SARSA is also known as *on-policy* learning as new action a' is chosen using same ϵ-greeedy policy similar to action a, the one which generated *s'* On the other hand in Q learning it is done by picking the **greedy action** a^g, In other words the action which maximizes the Q-value function in new state *Q(s',a)*:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a^g) - Q(s,a))] \quad (2)$$

or similarly

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma max_u Q(s',a^g) - Q(s,a))] \quad (3)$$

Q-learning is also known by the name *off-policy* learning as the new action a^g is greedy in nature, not using current policy.

II. Literature Review

- Gary in [3] has used Monte-Carlo method to solve the environment to solve the Roulette-V0 by setting the ϵ to 5 percent and γ to 1. Gary uses a million episodes to train the agent which requires a lot of computational power and time. On the other hand I have used only 30000 episodes to train the agent to get value function and results. In other words my approach is better than that of Gary in terms of utilizing resources. Gary also came to a conclusion that the best way to win roulette-v0 is not to play at all or in other words agent walks away if the loses are more than the wins. But when he plays the other possibilities, he loses 97.5 percent of times using Monte-carlo policy. But our approach got more than 50 percent of winning only after 25 games.
- Tanemaki in [4] has solved Roulette-v0 by Q learning algorithm with epsilon greedy/softmax policies. He can choose in between with a flag. The best thing which I liked about his algorithm he controlled exploration rate

Shahid Gulzar Padder is with the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary, e-mail: shahidgulzar4u@gmail.com
András Lőrincz is with the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

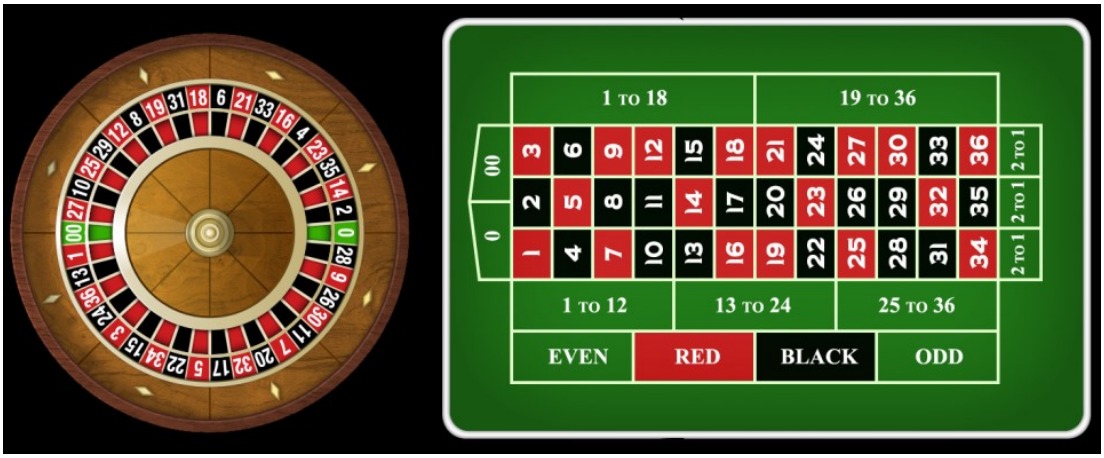

Fig. 1. Roulette example

by some reward statistics. At the end author is able to get four plots of steps, terminal rewards, cumulative rewards and epsilon respectively and it can be seen from the plots that terminal and cumulative rewards flatten to values near to zero after around 15000 episodes because of learning the optimal value function. This method is quite similar to mine and after 100 episode the average reward here is 4.34 with a run time of three seconds. For my case if α =0.1 (Table I) the average reward after 25 games was 7.0 but it might have varied after 100 games.

- Jacke in [5] has also chosen Q learning to solve the environment of Roulette-v0 and his approch is quite similar to mine. He has used a single hyperparameter sensibility but on the other hand I have extended this approach to three different sensibilities i.e: α, ϵ and γ. Jacke has evaluated it for 40000 and he has also came to the conclusion that best policy is to walk away if there are more loses than wins.
- Blessed in [6] has again implemented the solution using Q learning. Likewise he has defined the the +1 reward for correctly betting and -1 reward for incorrectly betting. For walking away from the table there is reward of zero. Therefore this method also solves roulette problem with similar approach that you can optimal value function if there are chances of losing more and you walk away in this situation.
- Hussien in [7] has used basic approach of using random seeds to get a final reward from the step function. He is just randomly selecting any action and observing the rewards. There is no strategy or policy followed by him. On the other hand our approach uses a comparative study of both Q learning and SARSA which are building blocks of Reinforcement learning. In his approach the reard keeps oscillating between -1.0 and 1.0.

III. METHODS

- For my environment solution I have chosen Q-Learning and SARSA as these methods are discussed in lecture and require no model.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Fig. 2. Q Learning algorithm (Sutton Barto)

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

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
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Fig. 3. State-action-reward-state-action **SARSA** algorithm (Sutton Barto)

- Principle in Q learning is a fixed strategy.Strategy that we use differs from the strategy that we evaluate.
- Q learning is an online algorithm. It exploits actual experiences only i.e; older experiences are not used.
- On the other hand SARSA converges faster in general. Same strategy for exploration and exploitation.
- Lets explain both algorithms briefly.

Q-Learning is an off policy RL algorithm. Choosing a strategy π we choose a greedy action in most of the time (Exploitation). We choose another action (Exploration). Q in Q-learning denotes quality. In other words how a given action **a** is fruitful in gaining some future reward. Immediate rewards are finite. Fig. 2 is Q learning algorithm [8]. In the next few steps I will show how we implement Q-learning algorithm.

- **Creating Q-table**: Whenever Q learning is performed we create a Q-table or which follows the shape of [state, action] and firstly we initialize values to zero. Updating

and storing of *q-values* takes place after an episode. This table is very important for the agent to choose the best action according to the q-value.

- **Q learning and updates**: In the next step agent interacts with the given environment and thus making updates to the state action pairs in the environment.
  Taking action: *Exploration and Exploitation* An agent interacts with a given environment in of the two given ways. The first way is to use q-table as the reference and select the action based on the maximum value of those actions. This way is known as *Exploitation* because we use already available information to make a particular decision. On the other hand the second method is to take actions randomly and this approach is called *Exploration*. We do not consider here the max future reward but we select action randomly.Exploration is equally important as exploitation.We cannot discover new states if we always rely on exploitation. We need to have a balance between these two using ϵ and thus setting the value of how much you need to explore/exploit. After every step or an action updates occur and end when an episode is finished. Finishing refers to terminal point by agent. It can be anything ranging from ending of a game to completion of the desired objective. It is an obvious fact that the agent will not learn too much after a single episode but after various steps and episodes it will finally converge and learn the optimal q-values.
  There are three basic steps: i) Agent in state(s1) starts, then takes an action (a1) and a reward(r1) is received. ii) In the second step refers to q-table and selects either highest value(max) or random(epsilon) action. iii) In the third step updation of q-values takes place. We haven't yet mentioned about **Learning rate** α, **Discount factor** γ and **Reward** R as stated in eq(1,2,3).
  Learning Rate: Learning rate, also referred α, in simple words can be defined as how much take up new value as compared to the old value.
  Discount factor: Gamma or γ is known as discount factor. Its primary aim is to make a balance between immediate and future reward.
  Reward: After completing a certain action at a given state reward is received. A reward can come into existence at any time-step or specifically at the terminal time-step.

The second algorithm the that I have used is **SARSA**.*State-action-reward-state-action* (SARSA) is another famous algorithm to learn **MDP** policy. Whole algorithm is depicted in the Fig 3. It is clear from the name that the updation of q-value depends on the present state **s** , the action agent takes **a**, the reward **R** after choosing this action, the state **s'** in which agent enters after taking action **a**, and lastly action **a'** which agent chooses in new state.

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a))] \quad (4)$$

It is known as on-policy learning algorithm because SARSA interacts with a given environment and updates the policy π on the basis of actions taken.

Hyperparameters:

Learning Rate: Learning rate, also referred α, in simple words can be defined as how much take up new value as compared to the old value. 0 learning rate means will not learn anything and 1 will make it consider most recent information

Gamma: Gamma or γ is known as discount factor. Its primary aim is to make a balance between immediate and future reward. 0 factor makes agent opportunistic because of considering only current rewards. On the other hand 1 factor will make wait for the long term bigger reward.

Initial conditions (Q(s0,a0)): As we know that SARSA is an iterative algorithm, it assumes an initial condition before first updation occurs. A lower initial value can encourage exploration.

These two methods do not require any model knowledge. Unlike Monte-Carlo we don't need to wait for end of complete episode. In these methods update is made after each step. For both cases we have ϵ-greedy policy which is necessary for exploration. In both cases for every episode from current state s, action *a* is taken from *s* to new state *s'* keeping track of reward *r*. While ϵ-greedy policy is followed given by current Q-value function the action *a* is taken. And here we need to update *Q(s,a)*

IV. Experiments, Results and Discussion

My main aim of the software task is based on Python scripts and in these scripts I have done a comparative study of **SARSA** and **Q-learning**. But for my approach I have chosen to play around the hyperparameters. I have divided the sensibility into three categories.

- **EPSILON/** ϵ **sensibility**: Analysis of sensibility to the exploration rate
- **ALPHA/** α **sensibility**: Analysis of sensibility to the learning rate
- **DISCOUNT/** γ **sensibility**: Analysis of sensibility to discounting rate

I changed my mode of sensibility for both SARSA and Q-learning. I have used code from github but the author has done it only for Discount/ γ *but I have extended it to* ϵ *and* α *as well*. In the implementation the maximum number of episodes for training are 30000.

Firstly we formulate the probabilities of the agent to receive positive reward. For instance the agent bets on 0 and the number after rolling is zero then the agent will get positive reward and thus the agent will have probability of 1/37 for receiving the reward. On the other hand if the agent chose to bet on any number other than 0, then to achieve a positive reward the got number shouldn't be 0 and the agents bet should match the parity of rolled number. Other than 0 agent's probability for getting a positive reward is 18/37 when betting on any number.

In both the cases the chances of losing are more or in other probability of receiving negative rewards is more than that of positive. Thus the agent should learn the policy of walking away before the actual game will start.

Roulette-v0 environment has 37 state-action pairs. As it can be seen that it is not too big so we can use tabular methods like SARSA/Q to solve this type of problem. We traind agent for

TABLE I
ALPHA (α) SENSIBILITY FOR 25 GAMES ON BETTING 23

Game number	Agents Reward	Game status
1	-1.0	LOST
2	0.0	WON
3	1.0	WON
4	0.0	LOST
5	1.0	WON
6	2.0	WON
7	3.0	WON
8	2.0	LOST
9	3.0	WON
10	2.0	LOST
11	3.0	WON
12	4.0	WON
13	5.0	WON
14	4.0	LOST
15	3.0	LOST
16	4.0	WON
17	5.0	WON
18	4.0	LOST
19	5.0	WON
20	4.0	LOST
21	5.0	WON
22	4.0	LOST
23	5.0	WON
24	6.0	WON
25	7.0	WON

30000 episodes. Fig. 4 shows the learnt action values by the agent during training. Here,we can see we have more chances of loosing than winning. In this figure optimal action is 37 with action value 0.0. This refers to the case of walking away from the table. When acting on optimal policy agent will walk away from the table. For Alpha α= 0.1 and 0.2 the number of games won were 16 and 13 respectively out of 25 games and the reward total was +7.0 and +1.0 respectively. If I increased α more than this agents chooses to walk away thus learning the optimal policy. Similarly for ϵ and γ the number of games won were 14 and 14 for ϵ=0.4 and γ=0.9 respectively. Fig. 5 shows the rewards collected with advent of trails in Q learning.

V. CONCLUSION

I have been successful in converging my given environment and I tried to check all the possibilities of hyper parameters. All data about convergence, number of iterations required, number of episodes etc. is given in the Table I, II, III and IV respectively. Moreover, when agent learns that there can be more loss to bet then he chooses to walk away thus learning the optimal value function. It was challenging to converge the environment with low computational power. In future I would like to improve my solution to faster convergence using Deep Q learning and Deep neural networks. Currently, I had low computational power and time constraints.

TABLE II
ALPHA (α) SENSIBILITY FOR 25 GAMES ON BETTING 22

Game number	Game Status	Total reward
1	WON	1.0
2	LOST	0.0
3	WON	1.0
4	LOST	0.0
5	WON	1.0
6	LOST	0.0
7	LOST	-1.0
8	LOST	-2.0
9	WON	-1.0
10	LOST	-2.0
11	WON	-1.0
12	WON	0.0
13	WON	1.0
14	LOST	0.0
15	WON	1.0
16	WON	2.0
17	LOST	1.0
18	WON	2.0
19	LOST	1.0
20	WON	2.0
21	LOST	1.0
22	WON	2.0
23	LOST	1.0
24	WON	2.0
25	LOST	1.0

TABLE III
EPSILON (ϵ) SENSIBILITY FOR 25 GAMES ON BETTING 35

Game number	Game Status	Total reward
1	WON	1.0
2	WON	2.0
3	LOST	1.0
4	LOST	0.0
5	WON	1.0
6	LOST	0.0
7	WON	1.0
8	LOST	0.0
9	WON	1.0
10	LOST	0.0
11	LOST	-1.0
12	WON	0.0
13	WON	1.0
14	WON	2.0
15	LOST	1.0
16	WON	2.0
17	WON	3.0
18	LOST	2.0
19	WON	3.0
20	WON	4.0
21	LOST	3.0
22	LOST	2.0
23	WON	3.0
24	WON	4.0
25	LOST	3.0

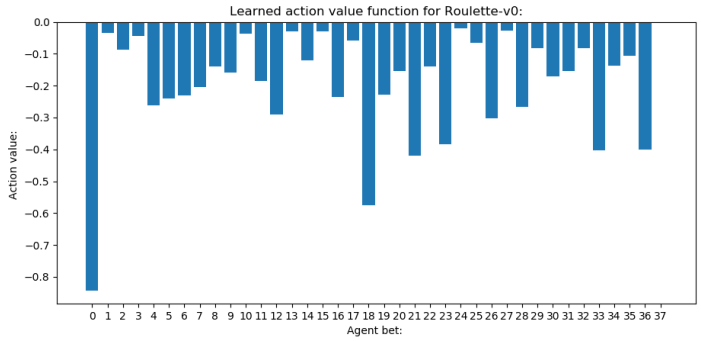

Fig. 4. Learned action-value function

Fig. 5. Rewards v/s Trials

TABLE IV
GAMMMA (γ) SENSIBILITY FOR 25 GAMES ON BETTING 31

Game number	Game Status	Total reward
1	WON	1.0
2	WON	2.0
3	WON	3.0
4	LOST	2.0
5	LOST	1.0
6	WON	2.0
7	LOST	1.0
8	WON	2.0
9	WON	3.0
10	LOST	2.0
11	LOST	1.0
12	LOST	0.0
13	WON	1.0
14	LOST	0.0
15	WON	1.0
16	WON	2.0
17	LOST	1.0
18	LOST	0.0
19	LOST	-1.0
20	WON	0.0
21	WON	1.0
22	LOST	0.0
23	WON	1.0
24	WON	2.0
25	WON	3.0

ACKNOWLEDGMENT

The author would like to thank Prof. András Lőrincz for guiding us whole semester and giving us a chance to get hands on Reinforcement Learning.

REFERENCES

[1] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
[2] Y.-H. Wang, T.-H. S. Li, and C.-J. Lin, "Backward q-learning: The combination of sarsa algorithm and q-learning," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 9, pp. 2184–2193, 2013.
[3] G. Butler, "Roulette with monte carlo policy," *https://medium.datadriveninvestor.com/learning-machine-learning-roulette-with-monte-carlo-policy-db1b3b788230*, 2018.
[4] Tanemaki, "Algorithm on roulette-v0," *https://gym.openai.com/evaluations/*, 2016.
[5] E. Jacke, "Q-learning for the roulette-v0," *https://github.com/eekjacke/Roulette-v0/blob/main/Roulette-v0.ipynb*, 2021.
[6] B. Cobra, "Simple q-learning agent for openai gym roulette-v0 environment." *https://github.com/BlessedCobra/roulette*, 2020.
[7] J. Hussien, "Openai-gym-envs," *https://github.com/JoHussien/OpenAI-GYM-envs/blob/master/Roulette-v0.ipynb*, 2020.
[8] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.