Backward Q-learning based SARSA (BQSA): balance between fast convergence and better final performance in Cliff-Walking experiment.

1st Aidar Shakerimov
Nazarbayev University
Nur-Sultan, Kazakhstan
aidar.shakerimov@nu.edu.kz

2st Dmitriy Li
Nazarbayev University
Nur-Sultan, Kazakhstan
dmitriy.li@nu.edu.kz

***Abstract*—BQSA, an algorithm that combines optimal behaviour of Q-Learning and fast convergence of SARSA, was proposed by Wang, Li and Lin, 2013. We investigate that reasons for why the algorithm obtain fast convergence as SARSA and good final performance as Q-learning and try to reproduce the experiment as it was described in the original paper in Cliff-Walking environment.**

I. Introduction

Reinforcement learning techniques keep obtaining more and more attention from Machine Learning community as they successfully accomplish variety of tasks that require knowledge gained through the process significantly better than human [1]. Newest developments of RL algorithms usually combine with deep learning: descending the ability to independently learn from interacting with an environment from reinforcement learning perspective and ability to process huge state spaces from deep learning perspective [2]. A recent great achievement was done by an agent that bet human level in most of Atari games in General Video Game Playing competition [3]. Another glorious achievements of RL approach was the first time AI winning against a professional human-player in famous ancient game called Go, one the hardest challenges for artificial intelligence [1].

In a core of any RL algorithm lies the policy, the way how to behave at a given situation [3], and the issue how to efficiently update this policy is one of relevant questions nowadays [2]. Two competing basic Reinforcement Learning techniques: Q-learning and SARSA illustrate different approaches to update their behavior policies which cause different advantages and disadvantages. While Q-Learning uses a certain policy to choose the action in the current state and greedy policy for the action in the next state when updating, SARSA uses one policy for both actions [4]. In other words, when updating the Q-table SARSA realistically assumes that in the next state the policy will be the same, and Q-Learning assumes the best case scenario in the future, that the agent will choose the most beneficial action.

The behaviour of these algorithms is well displayed through Cliff-Walking environment, a 4x12 or 5x12 grid world where an agent is tasked to reach goal using an optimal sequence of transitions [4], [5]. The environment has a goal state located close to the "cliff", a bunch of terminal states with negative reward, so that the fastest path to the goal lies along the "cliff" (see Fig 1): For each step the agent receives a small negative reward and when the agent walks into a cliff tile it receives a big negative reward and goes back to the starting tile [5]. In this scenario at the end of the learning, Q-Learning acts a little greedily, i.e. it walks right next to the cliff tile, while SARSA tries to avoid this area and walks slightly upper (see Fig 2a and Fig 2b). As a result of walking close to the cliff, Q-Learning agent, due to the random factor of action selection policy, sometimes falls off the cliff and thus the cumulative reward per episode may vary greatly. SARSA has less fluctuations but suffers from suboptimal behavior. Thus if the agents were to eventually use greedy policy Q-Learning would provide better results. Since SARSA has faster convergence but has suboptimal behavior, and Q-Learning allows for optimal behavior but lacks in convergence speed [4], [5], it is possible to say that both of these algorithms would perfectly complement each other [5]. Combining the benefits of Q-Learning and SARSA a new algorithm was proposed by Wang, Li and Lin, 2013 [5] called Backward Q-learning based SARSA. BQSA allows the agent to converge quickly using features borrowed from SARSA and then use features from Q-learning to optimize the behavior by approaching a better route.

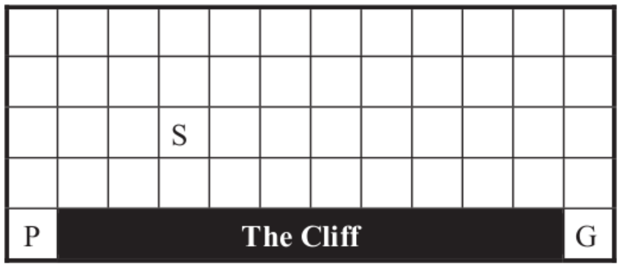
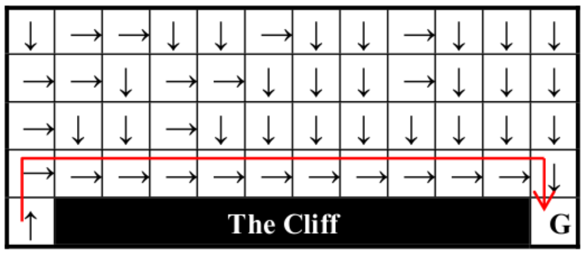
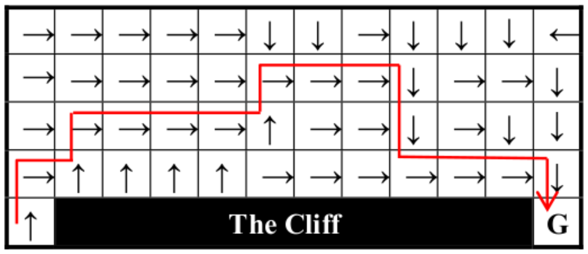


Fig. 1: Cliff-Walking environment

In this project we try to reproduce and discuss the Cliff-



(a) Q-Learning



(b) SARSA

Fig. 2: Different agents in Cliff-Walking environment

Walking experiment with BQSA agent, as described in Wang, Li and Lin, 2013. As a deliverable of the reproduction we present our pseudocode of the BQSA that learns to solve Cliff-Walking. The sections below are: description of the methods we use, the reproduction pseudocode, the discussion and summary.

II. Method

Here we discribe Q-Learning, SARSA and BQSA, three algorithms which were used in the Wang, Li and Lin, 2013 to solve Cliff-Walking experiment. However, unlike the authors, we replace Boltzmann softmax policy with epsilon-greedy exploration policy. That is done due to the fact that the paper does not provide the data required for using Boltzmann softmax policy. For example, the initial value of the temperature factor as well as its annealing criteria are missing, which makes the precise replication of the experiments impossible. On the other hand, since the algorithms are given the same exploration policy, we consider the comparison of the algorithms' performances to be independent of the exploration policy that is chosen. Although the numerical results of such reproduction may differ from the paper's results, the overall behavior of an agent and results ratio should stay the same.

A. Q-Learning

We already stated that Q-Learning (1) considers the best possible scenario for the next state when updating the Q-value, despite the fact that it does not always choose the best action in the current state. It is said that its policy how the agent chooses the current action and the policy how the action in the next state is chosen for the update function are different [4]. In our case the behavior policy is epsilon-greedy and the update policy is greedy (see Algorithm 1).

$$Q(S,A) = Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)] \quad (1)$$

Algorithm 1 Q-Learning algorithm

```
Initialize $Q$
for each episode do
    Choose random state $s_t$ or initialize $s_t$
    while episode is not finished do
        Choose an action $a_t$ from state $s_t$ using $\epsilon$-greedy
        Execute the action $a_t$, receive immediate reward $r_{t+1}$, observe the new state $s_{t+1}$ and see if the episode is finished
        Update $Q(s_t, a_t)$ according to Eq. (1)
        $s_t = s_{t+1}$
    end while
    Decay exploration rate
end for
```

B. SARSA

Unlike Q-Learning, SARSA(2) is an on-policy algorithm, i.e. it has the same behavior and update policies. SARSA assumes all the risks that the future steps may provide and hence behaves in a more safe way.

$$Q(S,A) = Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)] \quad (2)$$

Algorithm 2 SARSA algorithm

```
Initialize $Q$
for each episode do
    Choose random state $s_t$ or initialize $s_t$
    Choose an action $a_t$ from state $s_t$ using $\epsilon$-greedy
    while episode is not finished do
        Execute the action $a_t$, receive immediate reward $r_{t+1}$, observe the new state $s_{t+1}$ and see if the episode is finished
        Choose an action $a_{t+1}$ from state $s_{t+1}$ using $\epsilon$-greedy

        Update $Q(s_t, a_t)$ according to Eq. (2)
        $s_t = s_{t+1}$
        $a_t = a_{t+1}$
    end while
    Decay exploration rate
end for
```

C. BQSA

BQSA results from modifying SARSA with a method called Backward Q-Learning [5]. With this addition the current state and action, reward and the next state of each step are stored in some data structure to be used again at the end of the episode. When the episode is finished the records are retrieved starting from the last placed quadruple and Q-values are updated according to Q-Learning update function (1). The updates are conducted backwardly so that it allows to use already updated values of next state in the current state. This addition could be integrated into other algorithms [5], but we will take a close look at SARSA-based Backward Q-Learning.

In this algorithm SARSA is chosen as the base algorithm because it finds successful behavior faster. At each transition the agent performs an update of the state-action value in SARSA manner but before this it stores the transition (current state, current action, immediate reward and next state) in to a memory buffer. Then after terminal state is reached, these transitions are taken from the memory buffer and their state-action values are reevaluated from the perspective of Q-Learning. Using the Q-learning in backward update makes those transitions that lead to a more optimal path are gradually changing to be more appealing [5]. As a result of such a combination the behavior of the agent is becoming more optimal, like in Q-Learning, while keeping faster convergence from SARSA. The formula for the backward update is the following:

$$Q(S,A) = Q(S,A) + \alpha_b[R + \gamma_b \max_a Q(S',a) - Q(S,A)] \quad (3)$$

In order to vary the impact of Backward method (3), learning rate and discount factor are different from the ones in the base algorithm (2). The original paper empirically found that the configuration where $\alpha = 0.5$ and $\gamma = 0.95$ provides the best results [5]. The original paper used the pseudocode similar to Algorithm 3. Here M represents the data structure for keeping the records, and N is the number of records in M.

Algorithm 3 BQSA algorithm

```
Initialize $Q$ {1}, $M$ {2} and set $\alpha_b$ and $\gamma_b$ {3}
for {4} each episode do
    Choose random state $s_t$ or initialize $s_t$ {5}
    Choose an action $a_t$ from state $s_t$ using $\epsilon$-greedy {6}
    while {7} episode is not finished do
        Execute the action $a_t$ {8}, receive immediate reward
        $r_{t+1}$ {9}, observe the new state $s_{t+1}$ {10} and see if
        the episode is finished {11}
        Choose an action $a_{t+1}$ from state $s_{t+1}$ using $\epsilon$-greedy
        {12}
        Record the four events: $M \leftarrow s_t, a_t, r_{t+1}, s_{t+1}$ {13}
        Update $Q(s_t, a_t)$ according to Eq. (2) {14}
        $s_t = s_{t+1}$
        $a_t = a_{t+1}$
    end while{15}
    for {16} $j = N$ to 1 do
        Backward update $Q(s_t^j, a_t^j)$ according to Eq. (3) {17}
    end for
    Initialize all M values {18}
    Decay exploration rate {19}
end for{20}
```

D. *Epsilon-greedy exploration policy*

In order to include an exploration in learning of the agents we use epsilon-greedy action selection policy. It takes into consideration an exploration rate that is initialized with some number in interval [0, 1). The exploration rate determines the propability with which an agent will take a random action for exploration of the outcomes instead of using the most beneficial one. It decays episodically in order to decrease exploration and increase exploitation of learned actions [6].

III. Reproduction

We reproduced the pseudocode of the BQSA algorithm performing inside a Cliff-Walking environment. The implementation of the BQSA pseudocode (Algorithm 3) is explained in comments sub-section below. The number of the comment is referenced in the pseudocode as number in curly braces in the commented line.

A. *Comments*

1) Initialize the Q-table as a two-dimensional array with the row number equal to the amount of states in the environment (60 states: 12 x 5 cells) and the number of columns equal to the amount of available actions (4: up, down, right and left). Fill each entry of the table with zeros.
2) Initialize a memory buffer for transitions recording as list of 4-tuples.
3) Set learning rate for immediate Q-value update and for backward update to be equal to 0.5 and discount factor for both updates to be equal to 0.95. According to Wang, Li and Lin, 2013 the values are obtained experimentally to show the best output.
4) An agent will run a series of episodes until the goal counter will be equal to 300. However, before starting the loop, we need to perform several important steps:
    - States will represent an integer value of the agent's location in the environment in the following way: 12 * row's number + column's number. Specify the goal state = the 59th state, and the cliff states = state from the 49th to the 58th.
    - Specify the starting state for the run: a random state in interval from 0 to 48 inclusively.
    - Initialize a goal counter in order to indicate the end of the learning. The learning ends when the number of goals reaches 300.
    - Initialize caches (lists) for storing the cumulative rewards and lengths of episodes in order to monitor the on-line performance and convergence status of the agent.
5) Place the agent to the starting state
6) Run the epsilon-greedy function that takes the q-table, the current state and exploration rate as arguements and after generating a random nunber in interval [0, 1) returns either an action with best Q-value if the generated random number is higher than the exploration rate, either some random action if it is lower.

7) An agent will run an episode until the terminal state is reached: state = the goal state or cliff state. However, before starting the loop, we need to initialize a variable for accumulation of rewards in the episode and a variable to count transitions. This is important for monitoring the performance and convergence situation of the agent for each episode.
8) Move the agent in accordance with the selected action, Increment the transitions counter.
9) Obtain the reward in accordance with the next state: if the next state is the goal state, reward=10; if it is the cliff state, then reward = - 10, else reward = -1. Add the reward to the cumulative reward of the episode.
10) Convert the new location into integer variable denoting the next state.
11) If the next state is the goal state or a cliff state, the next is the terminal state.
12) Select the next action using next state in epsilon-greedy function.
13) Add the state, the action, the reward and the next state as a 4-tuple into the memory buffer.
14) Retrieve the Q-value from the Q-table using the state as a row's number and action as a column's number and perform the update according to the Eq. 2.
15) At the end of an episode append the accumulated reward and length of the episode into the cache in order to include it into the overall statistics.
16) Start a reverse loop for backward Q-learning update until the memory buffer becomes empty.
17) Pop a transition tuple (state, action, reward, next state) from the memory buffer. Update the Q-value of the transition in accordance to the Q-learning principle (see Eq. 3)
18) Reinitialize memory buffer.
19) Decay the value of the exploration rate. The decay factor should be obtained by experiments, those values providing best results will be chosen.
20) Return the Q-table (this will represent the generated behaviour policy that is ready for testing); the list of cumulative rewards (then we can extract average value in order to show on-line performance and standart deviation in order to show closeness to convergence) and list of lengths of episodes (we can count the sum of all lengths in order to receive the total time of learning).

IV. Discussion

We suppose that the main problem that our research meets is that we cannot replicate the experiment precisely. This is caused by the fact that authors do not provide any code implementation and, therefore, we reproduce the experiment ourselves based on our own understanding of the paper. As a consequence, we are responsible for verification of the accuracy of our reproduction. In order to prove the relevance of our reproduction we aim three important points to achieve: 1) Explain all implementation decisions as we did in the section reproduction description and in the appendix. 2) Compare the behavior of the agents and approximate results that we obtain with those presented in the paper. Capture the on-line performance and convergence during the learning: record cumulative rewards per episode, fluctuations of cumulative reward and total time of learning phase. Then test the final quality of generated policy by running a greedy agent. Use the same Q-table obtained in the learning, run a greedy agent that starts from each available state once (49 episodes in total), takes actions according to the Q-table in greedy manner until the episode termination. Count goals and total steps amount at the end of the testing to evaluate the quality of the learned policy. According to the paper, BQSA should provide less fluctuations, better cumulative rewards and give a faster convergence during the learning, a successful and optimal final policy in tests. Detailed pseudocode of testing agent can be found in the presentation in appendix.

After the verification of the reproduction we are planning to contribute the Wang, Li and Lin, 2013 research by studying the impacts of BQSA components towards convergence and final performance in environments with varying granularity of the state space. This research will increase the level of understanding how does the theoretical components of BQSA such as SARSA, backward Q-learning and their combined variant, SARSA with reversed policy update [6] influence the convergence speed and the final performance of the learning. In addition to that, we will analyze the efficency of the approach in environments with different complexity of state spaces. In order to accomplish that, we add three objectives to our project: 1) to introduce/propose a middle-step-algorithms between SARSA/Q-learning and BQSA using the idea of reversed policy update from Vu and Tran, 2020 [6]; 2) Evaluate convergence and final quality of the policies generated by different levels of algorithms in environment with different granularity of state space; 3) to summarize the impacts of each component of BQSA in terms of convergence and final performance. The final deliverables of our project would then include: 1) a proposed middle-step methods between Q-learning/SARSA and BQSA using idea of reversed policy update; 2) An implementation of Python code of Q-learning, SARSA, BQSA and middle-step algorithms in different environments using Gym library; 3) demonstration of visualization of performances of the agents in environments with varying granularity of state spaces using the matplotlib libraries.

V. Summary

The process of the project has revealed to us several important ideas that contributed our overall research experience. First of all, stability of the progress of the research depended on the clarity of the basic knowledges. Understanding of the principles of Q-learning and SARSA was crucial for starting research. Secondly, balancing between relevance and reproducibility is important. Most of new methods in RL introduce integration of deep learning which is hard to understand and reproduce on the UG level. On the other hand, simpler methods may be older and have less space for contribution. Thirdly, a division of the labor in research may lead to considerable gaps in each team-mate's understanding of the subject. It is important to push each other's understanding periodically. In overall, the reinforcement learning approach is a fast-developing influential state-of-art topic that can be implemented in personal computer software, but as any research topic it requires stable and clear understanding.

REFERENCES

[1] Koch, Christof. "How the computer beat the go player." *Sci Am Mind* 27 (2016): 20-23.

[2] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

[3] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529-533.

[4] Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[5] Wang, Yin-Hao, Tzuu-Hseng S. Li, and Chih-Jui Lin. "Backward Q-learning: The combination of Sarsa algorithm and Q-learning." *Engineering Applications of Artificial Intelligence* 26.9 (2013): 2184-2193.

[6] Vu, Tai, and Leon Tran. "FlapAI Bird: Training an Agent to Play Flappy Bird Using Reinforcement Learning Techniques." *arXiv preprint arXiv:2003.09579* (2020).

APPENDIX

Link to our GitHub code: https://github.com/aidarshakerimov/Backward-Q-Learning