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 et al. in their paper. We investigated the reasons for why the algorithm improves performance of SARSA, why certain methodology was used in the paper and finally we tried reproducing the experiment as it was described in the original paper in Cliff-Walking environment.**

I. Introduction

In Reinforcement Learning there are two very important algorithms that most novices learn: Q-Learning and SARSA. They are similar to the point that one would not be wrong to call SARSA a modification of Q-Learning or vice versa. As a matter of fact, there is only one difference, and it impacts the behaviour of the agent significantly. While Q-Learning(1) 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(2) uses one policy for both actions. 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.

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

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

In Sutton and Barto's book [1] the behaviour of these algorithms is explored through Cliff-Walking environment, a 4*12 grid where an agent is tasked to reach goal tile in the lower-right corner from the tile in the lower-left corner, but the entire lower row, except for the starting and final tiles are cliff tiles. 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. In this scenario 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 on the upper row. As a result of walking close to the cliff, Q-Learning agent due to the random factor of ϵ-greedy policy falls off the cliff and thus the cumulative reward per episode may vary greatly. SARSA has less fluctuations but suffers from suboptimal behaviour. Thus if the agents were to eventually use greedy policy Q-Learning would provide better results.

Considering the drawbacks and benefits of Q-Learning and SARSA a new algorithm was proposed that combines the positive aspects of them. BQSA (Backward Q-Learning based SARSA) [2] is SARSA modified with Backward Q-Learning update. As the SARSA agent learns the Q-values, 4-tuples of state, action, reward and next state are stored in a data structure to update Q-table at the end of the episode using Q-learning update. This additional component learns the Q-values starting from the last entry to the first, thus it is called Backward. Usually SARSA has faster convergence, because it has less possibility of receiving negative reward but has suboptimal behaviour, and Q-Learning allows for optimal behaviour but lacks in convergence speed. BQSA allows the agent to converge quickly and then use Backward update to optimize the behaviour by approaching a better route. We try replicating the experiments in the original paper by testing the algorithms on several metrics, which are Length of Episode, Cumulative reward, Total Steps and Success rate. The experiment is conducted in three environments: Cliff-Walking, MountainCar and CartPole.

II. Method

Our main focus here is on three algorithms: Q-Learning, SARSA and BQSA. We also tried to include all of the other algorithms used in the original paper. For all the algorithms Boltzmann softmax policy is used, where T is the temperature factor, which determines the ratio of exploration/exploitation (smaller T leads to more exploitation):

$$Pr(a_i) = \frac{exp(Q(s_t,a_i)/T)}{\sum_{k=1}^{n} exp(Q(s_t,a_k)/T)} \quad (3)$$

The temperature factor is annealed in order to provide higher exploration rate at the beginning of a learning process and higher exploitation of optimal choices later [3].

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 behaviour policy(how the agent chooses the current action) and update policy(how the action in the next state is chosen for the update function) are different. In our case the behaviour policy is Boltzmann softmax and the update policy is greedy.

B. SARSA

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

Algorithm 1 BQSA algorithm

```
Initialize $Q$, $M$ and set $\alpha_b$ and $\gamma_b$
for each episode do
    Choose random state $s_t$ or initialize $s_t$
    Choose an action $a_t$ from state $s_t$ using Boltzmann Eq. (3)
    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 Boltzmann Eq. (3)
        Record the four events: $M \leftarrow s_t, a_t, r_{t+1}, s_{t+1}$
        Update $Q(s_t, a_t)$ according to Eq. (2)
        $s_t = s_{t+1}$
        $a_t = a_{t+1}$
    end while
    for $j = N$ to 1 do
        Backward update $Q(s_t^j, a_t^j)$ according to Eq. (4)
    end for
    Initialize all M values
    Anneal the temperature factor
end for
```

C. BQSA

BQSA results from modifying SARSA with a method called Backward Q-Learning [2]. 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, 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 behaviour faster. Then after the episode the Q-values are reevaluated from the perspective of Q-Learning, and thus action-value pairs that lead to a more optimal policy are gradually changing to be more appealing. As a result of such a combination the behaviour of the agent is becoming more optimal, like in Q-Learning, while keeping faster convergence from SARSA.

The Backward update is different from SARSA update in this algorithm. In order to vary the impact of Backward method learning rate and discount factor different from the ones in the base algorithm were used. The original paper empirically found that the configuration where $\alpha = 0.5$ and $\gamma = 0.95$ provides the best results [2]. The formula for the backward update is the following:

$$Q(S, A) = Q(S, A) + \alpha_b [R + \gamma_b \max_a Q(S', a) - Q(S, A)] \quad (4)$$

The original paper used the pseudocode similar to Algorithm 1. Here M represents the data structure for keeping the records, and N is the number of records in M.

III. Reproduction Description

We reproduced the pseudocode of the BQSA algorithm performing inside a Cliff-Walking environment.

Algorithm 2 Our implementation of BQSA

```
Initialize $Q$, $M$ and set $\alpha$, $\gamma$, $\alpha_b$ and $\gamma_b$
Initialize $gcount = 0$ to score the number of times goal was reached
Initialize $reward_cache$ and $step_cache$ to keep the cumulative rewards and lengths of each episode
while $gcount < 300$ 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
        Record the four events: $M \leftarrow s_t, a_t, r_{t+1}, s_{t+1}$
        Update $Q(s_t, a_t)$ according to Eq. (2)
        $s_t = s_{t+1}$
        $a_t = a_{t+1}$
    end while
    Append cumulative reward and length of episode to corresponding lists
    while $M$ is not empty do
        Pop a quadruple from $M$
        Backward update $Q(s_t^j, a_t^j)$ according to Eq. (4)
    end while
    Initialize all M values
    Decay exploration rate
end while
return $Q$, $reward_cache$, $step_cache$
```

A. Environment

The environment represents a 12 x 5 grid with 4 available actions (up, down, left, right). The bottom right tile of the grid represents the goal state, and the entire bottom row, except for goal state and the tile in the bottom left corner are cliff states. By default agent receives a reward of -1 each timestep, for reaching the goal state the reward is 10 and for falling off the cliff the reward is -10.

B. Implementation Details

In the environment state is represented as two numbers, and to simplify the Q-table dimensionality state is calculated as: $12*Y+X$, where Y - row number(y-position) and X - column number(x-position). As such, the final state becomes 59 and the cliff states become $49:58$. To track the performance of the agents cumulative reward and length of each episode are recorded in two lists. Data structure for memory buffer is a list, which has a pop method that works similarly to a stack.

C. ϵ-greedy

We decided to use ϵ-greedy as the policy of our agents due to absence of data required for Boltzmann softmax policy: temperature factor initial value and temperature annealing criteria are missing in the paper. In short ϵ-greedy works like this: with some probability ϵ the agent is going to explore, i.e. choose the action randomly, otherwise the action is chosen greedily.

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 donot 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) Verify all implementational decisions trought the careful explanations as we did in the section reproduction description and in the appendix. 2) 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. Compare the results with the expected behavior according to the paper (see figures). 3) Test the final quality of generated policy by running a greedy agent. Evaluate the optimality of the performed actions (BQSA should provide optimality) in terms of total performed steps. 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. 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 th elevel of understanding how does the theoretical components of BQSA such as SARSA, backward Q-learning and their combined variant, SARSA with reversed policy update (Vu and Tran, 2020) influence the convergence speed and the final performace of the learning. In addition to that, we will analyze the efficency of the approach in environmens with diferent compexities 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; 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.

References

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

[2] 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.

[3] Sakaguchi, Yutaka, and Mitsuo Takano. "Reliability of internal prediction/estimation and its application. I. Adaptive action selection reflecting reliability of value function." Neural Networks 17.7 (2004): 935-952.

V. Implementation Details

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