Evaluation of Sarsa(λ) Learning Agent

Padraic Cashin [*], Ruihao Zhou [†] David Lahtinen [‡], Zubin Kapadia [§],
[*] ASU ID: 1214153888
[†] ASU ID: 1213439264
[‡] ASU ID: 1207725034
[§] ASU ID: 1213238024

I. Introduction

One of the most important reinforcement learning technique in Artificial Intelligence is Q-learning and its goal is to learn itself and get a desired policy which will help agent to make best choice under that circumstances. Q-learning does not require a model of the environment and can handle problem without requiring adaptations, only with with stochastic transitions and rewards. By successfully implementing the agent in our individual project 3, we have already had a good understanding of this method, however, its not the best method which means it also has some limitation. For example, it may be slower and remains inflexible if alternative routes does not appear. Also, policies may not be sufficiently similar. In this group project, we try to implement a on-policy algorithm called SARSA(λ) and Q-learning into our project three at the same time, and find out whats the difference between these two popular method. In the end, we can get straightforward result of this comparison.

II. Background

The Sarsa algorithm belongs to an On-Policy algorithm for TD-Learning. The speciality of this algorithm compared to Q-Learning is that in the next state, maximum reward doesnt play an role for updating the Q-values. Instead, it uses the next action and the same policy that determined the original action to get new reward. The name Sarsa actually comes from the four first letter of Q(s, a, r, s', a'). s and a represent original state and action, r is the reward observed in the following state, s', a' are the new state-action pair. As you can see in the following pictures, two action selection steps needed for determining the next state-action pair along with the first. The parameters a and r wont change as they do in Q-Learning. The nature of the policy's dependence on Q determine the convergence of the properties of Sarsa algorithm. For example, one could use ϵ-soft or ϵ-greedy policies. [2]

III. Sarsa(λ) Implementation

We chose to start with the codebase provided by UC Berkeleys Pacman-themed AI tutorial. As the SARSA-lambda algorithm requires an eligibility trace we implemented this as a python dict with default value of 0, same as our Q table. Unlike the Q-table, the eligibility trace must be cleared between episodes, so we implemented a check for final step in the function which updated our Q table to clear the eligibility trace. To test and demo the sarsa-lambda agent, we used a test that was built-in, and used by UC berkeley which consisted of a small, gridworld maze for our agent to run around in. What we found was that while the Default q agent first had to random walk to the goal, then to the square in front of the goal, and so on for each episode, the SARSA-lambda agent converged very quickly. After a single random walk to the goal, the Q table updated almost immediately to the optimal policy for the default starting position. While we probably could have improved the results by implementing an ϵ-greedy function, our results for the basic set learning rate worked very well.

```
Initialize $Q(s,a)=0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for each episode do
    $E(s,a)=0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
    Initialize $S, A$
    for step in an episode do
        Take action $A$ and observe $S', R$
        Choose $A'$ based on $S'$ and current policy
        $\delta \leftarrow R + \gamma Q(S',A') - Q(S,A)$
        $E(S,A) \leftarrow (1-\alpha)E(S,A) + 1$
        for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ do
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
            $E(s,a) \leftarrow \gamma\lambda E(s,a)$
        end
        $S \leftarrow S'$
        $A \leftarrow A'$
    end
    Until $S$ is terminal
end
```

Algorithm 1: Sarsa(λ) Algorithm with Dutch Tracing. Algorithm was provided by Sutton et al. [2]

IV. Results

Once the Sarsa(λ) agent had been integrated into the UC Berkley Pacman framework [1], *gridword.py* and *pacman.py* were used to test out the efficacy of the Sarsa(λ) agent compared to the Q-learning agent already present.

For the this paper, we evaluated policy/value convergence in *gridworld.py*, using the BookGrid world. Convergence was found by exponentially increasing the number of training episodes until the policy became stationary. The default values for each agent were: $\epsilon = 0.05, \gamma = 0.8, \alpha = 0.2\, \lambda = 0.8$ and $noise = 0.2$.

```
def update(self, state, action, nextState, reward):
    nextAction = self.getAction(nextState)

    delta = reward + (self.discount *
        self.values[(nextState, nextAction)]) -
        self.values[(state, action)]

    self.eligibility[(state, action)] = (1 - self.alpha) *
        self.eligibility[(state, action)] + 1

    for k, v in self.values.iteritems():
        trace = self.eligibility[k]
        self.values[k] = v + (self.alpha * delta * trace)
        self.eligibility[k] = trace * self.discount * self.y

    if not self.getLegalActions(nextState):
        self.eligibility = util.Counter()
```

Fig. 1: Implementation of `update` for Sarsa(λ) agent. Since `update` is called by the agent each time an aciton is selected, the function implements the inner loop of the algorithm detailed in Algorithm 1.

We also created two custom grid layouts viz. CustomGrid and LargeMazeGrid to test and compare the two agents. The LargeMazeGrid is a 20x20 grid with multiple dead ends and a single reward. We ran both the agents for 100 episodes (with default values and 0 noise) and the Sarsa(λ) agent out performed the Q-learning agent on this grid. For the Sarsa(λ) agent policy convergence was noticed after 15 episodes, whereas for the Q-learning agent it was after 65 episodes. Average time taken per episode for the Sarsa(λ) agent was 0.18s, while the Q-learning agent took 0.29s.

The CustomGrid is a 5x5 grid, with a small reward near the start point and a large reward further away, with a couple hazards along the way. We ran both the agents for 1000 episodes (with default values and 0 noise). Both the agents performed identically on this grid. With the small reward being near the start, both the agents deviated towards it and didn't stray much from that path.

To evaluate each agent in a more complex environment, we used *pacman.py* with smallGrid world. Each agent was trained for $[100, 2000]$ episodes and then evaluated for another 20 episodes. The default settings for each agent was: $\epsilon = 0.05, \gamma = 0.8, \alpha = 0.2$ and $\lambda = 0.8$.

Table I shows a summary of the experimental results. In general we found that the Sarsa(λ) agent out performed the Q-learning agent in the grid world. Sarsa(λ) was able to converge on an optimal policy 3840 training episodes before the Q- learning agent. On the other hand, the Sarsa(λ) agent took approximately $500\times$ longer to train in the *pacman.py* environment, most likely due the number of $(state, action)$ pairs that exist in the environment. If we are willing to wait for the training rounds, the Sarsa(λ) agent was able to reach a consistent 100% win rate before the Q-learning agent.

In the *gridworld.py* experiments we found that Sarsa(λ) performed better than Q-learning due its eligibility trace allowing for each training episode to update more of the Q-values than Q-learning's single step functions. Figure 4 clearly illustrates the difference. After only 10 training episodes, a majority of the Q-values have been updated at least once by the Sarsa(λ) agent; while the Q-learning agent has only updated a few of its Q-values.

Agent	Time(s)/100 eps	50% Win Rate	Value convergence
Sarsa(λ)	1678.92s	1000 eps	256 eps
Q-Learning	3.54s	800 eps	4096 eps
Approximate Q-Learning	1.92s	800 eps	N/A
Approximate Sarsaλ)			N/A

TABLE I: Summary of measured differences between Sarsa(λ) and Q-learning agents. Time per 100 episodes (seconds) was taken from 20 trials using the *pacman.py* smallGrid world. Similarly, the agents were trained on an increasing number of episodes until they achieved a 50% win rate in the *pacman.py* smallGrid world. Value convergence was found by exponentially increasing the number of training episodes until the Q-values and action policy became stationary. Value convergence was run using *gridworld.py* on the BookGrid world.

The ability to update the Q-values associated with each $(state, action)$ in the eligibility trace allows for good decisions be effect the value of the prior decisions, and thus speeds up policy convergence. It also, means that Sarsa(λ) is susceptible to noise, as a series of poor or random choices that lead to a positive outcome are *all* updated as though those decisions were good or meaningful to the final outcome. This leads the agent to get temporarily stuck in policies which are not optimal, meaning take longer to achieve the goal.

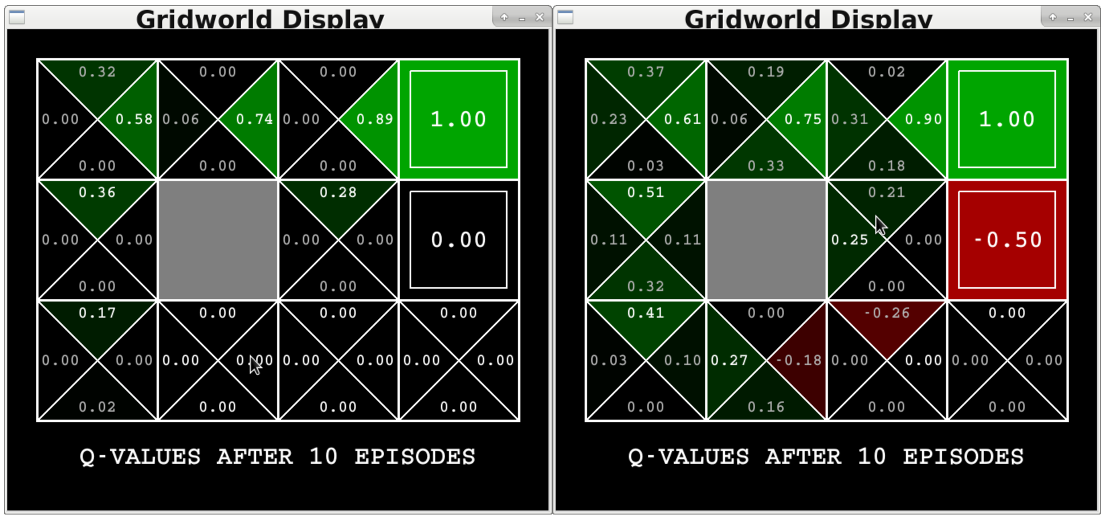
The other trade off of using the eligibility trace each time the agent picks an action, is the time needed to loop through all of the Q-values in the environment. For small simple environments such as *gridworld.py*, the trade off between training time and number of training episodes puts Sarsa(λ) ahead of Q-learning. However, in the more complex *pacman.py* environment the Sarsaλ) agent slows down considerably. Even though Figure 5 shows that Sarsa(λ) was able to reach peak win rate with fewer training episodes than Q-learning, the difference in real world training time was several hours.

V. Conclusion

From what we have already implemented, we can easily draw two conclusions:

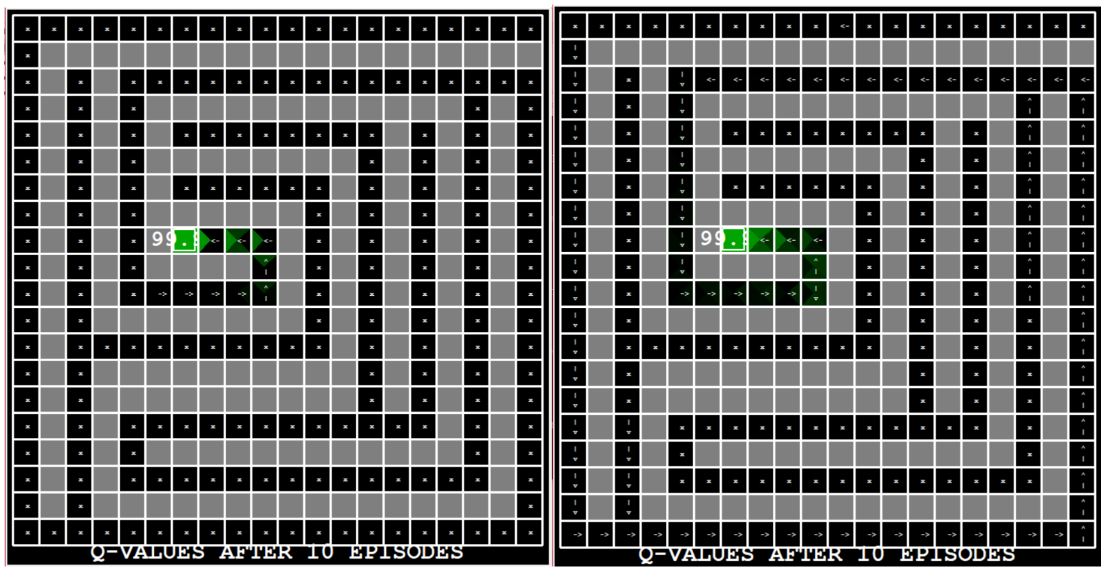
1. Q-learning maybe suffers from problems converging as a result due to it has higher per-sample variance than SARSA, however SARSA learns a near-optimal policy while exploring. So if we decide to use SARSA, we need propose a good strategy to decay in -greedy action choice, which can become fiddly hyper parameter to adjust.

2. SARSA is more conservative than Q-learning, because SARSA will approach convergence allowing for all kinds of penalties while agent explores its move while the Q-learning ignore them. In another words, if there is risk of a large negative reward close to the optimal path, Q-learning will perform more aggressive and will trigger that reward risk. On



(a) Q-learning agent Q values (b) Sarsa(λ) agent Q values

Fig. 2: Q values of the Q-learning and Sarsa(λ) agents after 10 episodes of training. The Sarsa(λ) agent is able to evaluate more of the state space than the Q-learning agent. Each agent was run for 10 episodes on the BookGrid world with a noise rate of 0.2, and $\epsilon = 0.5$.



(a) Q-learning agent Policy (b) Sarsa(λ) agent Policy

Fig. 3: Policies of the Q-learning and Sarsa(λ) agents after 10 episodes on the LargeMazeGrid. Sarsa(λ) agent's policy is almost on the verge of converging as opposed to that of the Q-learning agent.

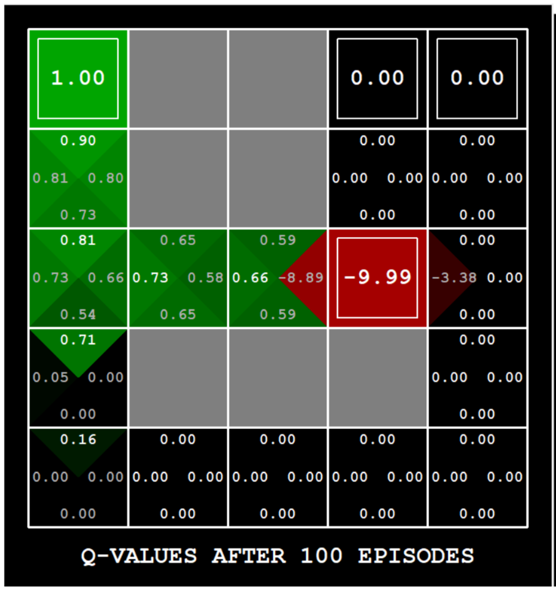
the contrary, SARAS will try to avoid it and slowly learn to use it.

All in all, Q-learning learn optimal policy directly and SARAS learn online. If you want to train agent in low-cost and fast0iterating environment, choose Q-learning. And if you care about reward risks while learning, SARAS will be the best choice compared Q-learning.

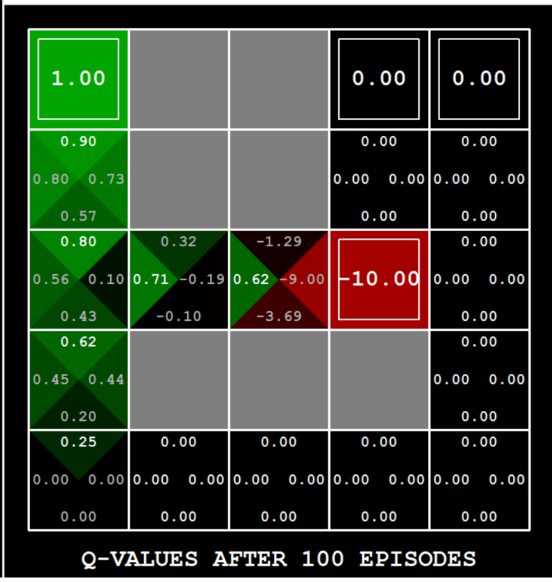
References

[1] D. Klein, J. DeNero, and P. Abbeel. Uc berkeley cs188 intro to ai, 2014.

[2] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

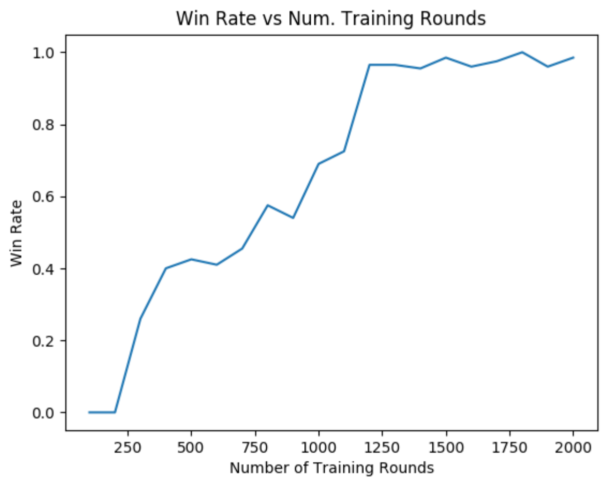


(a) Q-learning agent Q values

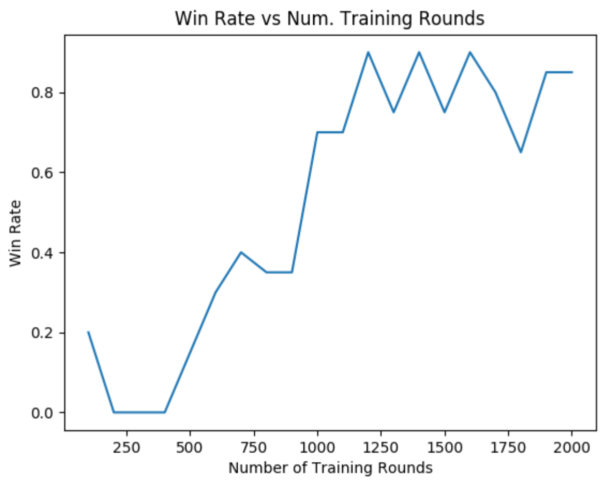


(b) Sarsa(λ) agent Q values

Fig. 4: Q values of the Q-learning and Sarsa(λ) agents after 100 episodes on the CustomGrid. Identical results are found for both the agents on this grid.



(a) Q-Learning Agent



(b) Sarsa(λ) Agent

Fig. 5: Win rate vs training episodes for Q-learning and Sarsa(λ) agents. Each agent was trained using pacman smallGrid environment. Win rate was taken from 20 post training episodes. Each agent was trained with $\epsilon = 0.05, \gamma = 0.8, \alpha = 0.2$. The Sarsa($\lambda$) agent has a default $\lambda = 0.8$.