Reinforcement Learning

Carlos Matheus Barros da Silva, *Computer Engineering Bachelor Student of ITA*
Prof. Marcos Ricardo Omena de Albuquerque Máximo

***Abstract*—This paper evaluates two *Temporal Difference Learning* techniques, *SARSA* and *Q-Learning*.**

Both techniques have been implemented and passed by some tests. On the end It was avaliated the learning with a car that needs to follow a track.

It was observed that both implementation worked as expected. The *Q-Learning* having a faster learning, converging to a local maximum with much less iteration than the *SARSA* algorithm.

***Index Terms*—Temporal Difference Learning, SARSA, Q-Learning, Reinforcement Learning**

I. Introduction

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

It differs from supervised learning in that labelled input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

The environment is typically formulated as a Markov decision process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible.

Temporal difference (TD) learning refers to a class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function. These methods sample from the environment, like Monte Carlo methods, and perform updates based on current estimates, like dynamic programming methods.

While Monte Carlo methods only adjust their estimates once the final outcome is known, TD methods adjust predictions to match later, more accurate, predictions about the future before the final outcome is known. This is a form of bootstrapping, as illustrated with the following example:

"Suppose you wish to predict the weather for Saturday, and you have some model that predicts Saturday's weather, given the weather of each day in the week. In the standard case, you would wait until Saturday and then adjust all your models. However, when it is, for example, Friday, you should have a pretty good idea of what the weather would be on Saturday – and thus be able to change, say, Saturday's model before Saturday arrives."

Temporal difference methods are related to the temporal difference model of animal learning.

II. Algorithms Implementation

Both Techniques implementation can be seen on the file *reinforcement_learning*. The essence of the implementation can be seen on the Code 1.

```
def compute_greedy_policy_as_table(q):
    """
    Computes the greedy policy as a table.

    :param q: action-value table.
    :type q: bidimensional numpy array.
    :return: greedy policy table.
    :rtype: bidimensional numpy array.
    """
    policy = np.zeros(q.shape)
    for s in range(q.shape[0]):
        policy[s, greedy_action(q, s)] = 1.0
    return policy


def epsilon_greedy_action(q, state, epsilon):
    """
    Computes the epsilon-greedy action.

    :param q: action-value table.
    :type q: bidimensional numpy array.
    :param state: current state.
    :type state: int.
    :param epsilon: probability of selecting a random
     action.
    :type epsilon: float.
    :return: epsilon-greedy action.
    :rtype: int.
    """
    if np.random.binomial(1, epsilon) == 1:
        action = np.random.choice([a for a in range(len(q[
     state]))])
    else:
        values_ = q[state]
        action = np.random.choice([action_ for action_,
     value_ in enumerate(values_) if value_ == np.max(
     values_)])

    return int(action)


def greedy_action(q, state):
    """
    Computes the greedy action.

    :param q: action-value table.
    :type q: bidimensional numpy array.
    :param state: current state.
    :type state: int.
    :return: greedy action.
    :rtype: int.
    """
    values_ = q[state]
    return np.random.choice([action_ for action_, value_ in
      enumerate(values_) if value_ == np.max(values_)])


class RLAlgorithm:
    """
    Represents a model-free reinforcement learning
     algorithm.
    """
    def __init__(self, num_states, num_actions, epsilon,
     alpha, gamma):
        """
        Creates a model-free reinforcement learning
     algorithm.

        :param num_states: number of states of the MDP.
        :type num_states: int.
        :param num_actions: number of actions of the MDP.
        :type num_actions: int.
        :param epsilon: probability of selecting a random
     action in epsilon-greedy policy.
        :type epsilon: float.
        :param alpha: learning rate.
        :type alpha: float.
        :param gamma: discount factor.
        :type gamma: float.
        """
        self.q = np.zeros((num_states, num_actions))
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma

    def get_num_states(self):
        """
        Returns the number of states of the MDP.

        :return: number of states.
        :rtype: int.
        """
        return self.q.shape[0]

    def get_num_actions(self):
        """
        Returns the number of actions of the MDP.

        :return: number of actions.
        :rtype: int.
        """
        return self.q.shape[1]

    def get_exploratory_action(self, state):
        """
        Returns an exploratory action using epsilon-greedy
     policy.

        :param state: current state.
        :type state: int.
        :return: exploratory action.
        :rtype: int.
        """
        return epsilon_greedy_action(self.q, state, self.
     epsilon)

    def get_greedy_action(self, state):
        """
        Returns a greedy action considering the policy of
     the RL algorithm.

        :param state: current state.
        :type state: int.
        :return: greedy action considering the policy of
     the RL algorithm.
        :rtype: int.
        """
        raise NotImplementedError('Please implement this
     method')

    def learn(self, state, action, reward, next_state,
     next_action):
        raise NotImplementedError('Please implement this
     method')


class Sarsa(RLAlgorithm):
    def __init__(self, num_states, num_actions, epsilon,
     alpha, gamma):
        super().__init__(num_states, num_actions, epsilon,
     alpha, gamma)

    def get_greedy_action(self, state):
        """
        Notice that Sarsa is an on-policy algorithm, so it
     uses the same epsilon-greedy
        policy for learning and execution.

        :param state: current state.
        :type state: int.
        :return: epsilon-greedy action of Sarsa's execution
     policy.
        :rtype: int.
        """
        return epsilon_greedy_action(self.q, state, self.
     epsilon)

    def learn(self, state, action, reward, next_state,
     next_action):
        target = 0.0
        q_next = self.q[next_state]
        best_actions = np.argwhere(q_next == np.max(q_next)
     )
        for action_ in self.q[state]:
            action_ = int(action_)
            if action_ in best_actions:
                target += ((1.0 - self.epsilon) / len(
     best_actions) + self.epsilon / len(self.q[state])) *
     self.q[next_state][action_]
            else:
                target += self.epsilon / len(self.q[state])
     * self.q[next_state][action_]
        target *= self.gamma
        action = int(action)
        self.q[state][action] += self.alpha * (reward +
     target - self.q[state][action])


class QLearning(RLAlgorithm):
    def __init__(self, num_states, num_actions, epsilon,
     alpha, gamma):
        super().__init__(num_states, num_actions, epsilon,
     alpha, gamma)

    def get_greedy_action(self, state):
        return greedy_action(self.q, state)

    def learn(self, state, action, reward, next_state,
     next_action):
        self.q[state][action] += self.alpha * (reward +
     self.gamma * np.max(self.q[next_state]) - self.q[state
     ][action])
```

Code 1. Code of both *Temporal Difference Learning* techniques: *SARSA* and *Q-Learning*

III. Temporal Difference Techniques Analysis

A. Initial Test Analysis

1) SARSA: In order to initially evalute *SARSA* technique, it was run the file *test_rl.py* setted to test *SARSA*.

With this setup it was obtained the output represented on the Code 2.

```
Action-value Table:
[[-1.10987995 -1.01087793 -1.10989846]
 [-1.10991555 -1.10987994 -1.1100879 ]
 [-1.11939826 -1.10989833 -1.11224751]
 [-1.11190989 -1.1101107  -1.13647653]
 [-1.13202054 -1.11271984 -1.1421604 ]
 [-1.10981013 -1.11113142 -1.10978179]
 [-1.10939041 -1.10944667 -1.10890761]
 [-1.10912798 -1.10976546 -1.10007692]
 [-1.10007692 -1.10890761 -1.01087793]
 [ 0.         -0.10890761 -0.10987811]]
Greedy policy learnt:
[L, L, L, L, L, R, R, R, R, S]
```

Code 2. Output of *SARSA* test, It is denoted the Action-value Table and the Greedy policy learnt

In this test was observed a slow converge of *SARSA* in a way that running this same test multiple times might result um some Greedy policy slightly differents.

2) Q-Learning: In order to initially evaluate *Q-Learning* technique, it was run the file *test_rl.py* setted to test *Q-Learning*.

With this setup it was obtained the output represented on the Code 3.

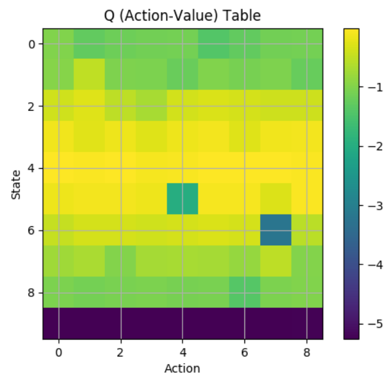

Fig. 1. *SARSA* Action Value Table at the end of more 700 iterations.

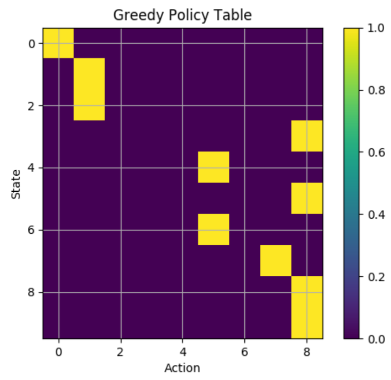

Fig. 2. *SARSA* Greedy Polucy Table at the end of more 700 iterations.

```
Action−value Table:
[[−1.99          −1.          −2.9701       ]
 [−2.96714069   −1.99         −3.91352577]
 [−3.74296146   −2.9701       −4.08464779]
 [−4.40221178   −3.94039885   −4.71148159]
 [−5.19580059   −4.89647895   −4.89670467]
 [−4.26602435   −4.91073711   −3.94039888]
 [−3.40003391   −4.35595654   −2.9701      ]
 [−2.9645173    −3.91633443   −1.99        ]
 [−1.99          −2.9701       −1.          ]
 [ 0.            −0.99         −0.99        ]]
Greedy policy learnt:
[L, L, L, L, L, R, R, R, R, S]
```

Code 3. Output of *Q-Learning* test, It is denoted the Action-value Table and the Greedy policy learnt

In this test was observed a faster converge of *Q-Learning* technique, in a way that it is less frequent than *SARSA* to get a variation on the Greedy policy by the end of the test.

It was observed as well that the *Q-Learning* algorithm is, as well, slightly faster than the *SARSA*, in a way that more iterations are completed in *Q-Learning* in a given time.

B. Car Test Analysis

1) SARSA: In order to initially evalute *SARSA* technique, it was run the file *main.py* setted to test *SARSA*.

With this setup it was obtained the output represented by the graphs from Image 1 to Image 3.

With this experiment and observing the Image 3, it is possible to see that the *SARSA* technique has a slow convergence having a high variance even with hundreds of iterations already, what can be good to prevent bias.

2) Q-Learning: In order to initially evalute *Q-Learning* technique, it was run the file *main.py* setted to test *Q-Learning*.

With this setup it was obtained the output represented by the graphs from Image 4 to Image 6.

With this experiment and observing the Image 6, it is possible to see that the *Q-Learning* technique has a faster con-

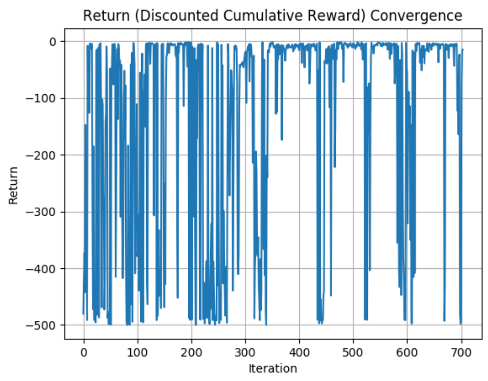

Fig. 3. *SARSA* Return converge graph with more 700 iterations.

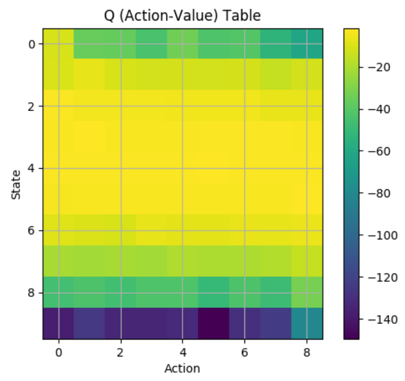

Fig. 4. *Q-Learning* Action Value Table at the end of more 700 iterations.

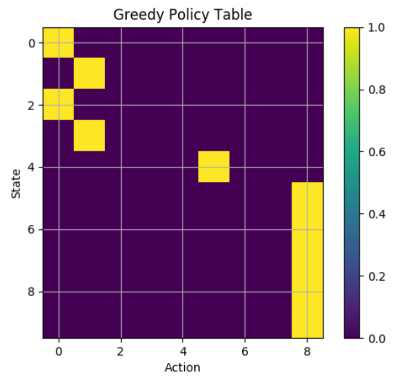

Fig. 5. *Q-Learning* Greedy Polucy Table at the end of more 700 iterations.

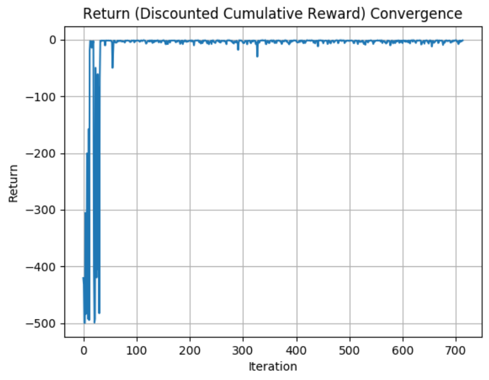

Fig. 6. *Q-Learning* Return converge graph with more 700 iterations.

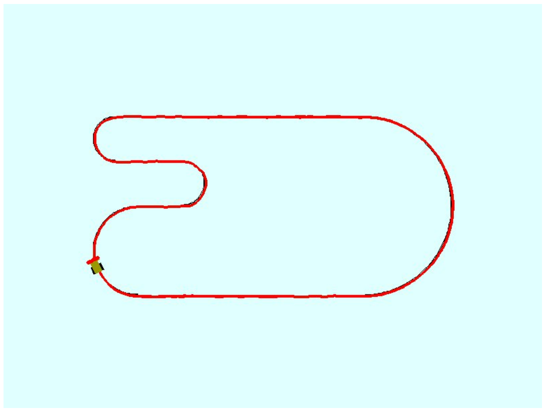

Fig. 7. *Q-Learning* solution after more 700 iterations.

vergence when compared with *SARSA* having a low variance with time, what this is good because it learns fast a good approach, but it can have a higher bias.

In fact, *Q-Learning* has a so high learning rate that at about 10% of the number of iterations of *SARSA* it was perform already better, this is, at iteration 70 *Q-Learning* was performing better than *SARSA* at iteration 700.

At the end, *Q-Learning* was with a very smooth approach. It can be seen in Image 7.

IV. Conclusion

It was clear, therefore, that both implementation worked as expected. The *Q-Learning* having a faster learning, converging to a local maximum with much less iteration than the *SARSA* algorithm.

This makes *Q-Learning* having a low variance but it might have a higher bias. While *SARSA* have a higher viriance with a lower bias.