SAPIENZA
UNIVERSITÀ DI ROMA

Department of Computer, Control and Management
Engineering Antonio Ruberti

Assignment 2

Reinforcement Learning

Master program in Artificial Intelligence and Robotics

Professor:
Roberto Capobianco

TAs:
Andrea Fanti
Michela Proietti

Student:
Lavalle Leonardo 1838492

Academic Year 2023/2024

Contents

1 Theory

1.1

Temporal Difference *tabular* updates are defined in this way:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s', \bullet) - Q(s,a)) \tag{1}$$

How do we select action $\bullet$? Depending on this choice, we essentially have two algorithms: *(i)* **Sarsa**, where the target action is selected according to policy π (*on-policy*); *(ii)* **Q-learning**, where the action is greedy with respect to Q (*off-policy*). In particular the selected action is $\max_{a'} Q(s', a')$.

Given the following Q-table at time t:

$$Q_t(s,a) = \begin{pmatrix} Q_t(s_1,a_1) & Q_t(s_1,a_2) \\ Q_t(s_2,a_1) & Q_t(s_2,a_2) \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{2}$$

with $\alpha = 0.1$ and $\gamma = 0.5$.

Update the Q-table according to both *Sarsa* ($a' = \pi_\epsilon(s') = a_2$) and *Q-learning*, after the experience depicted in Figure 1 at time $t+1$.

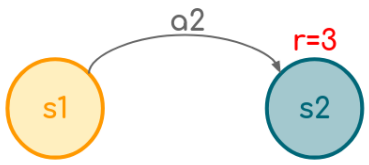

Figure 1

Sarsa:

$$\begin{aligned} Q_{t+1}(s_1,a_2) &= Q_t(s_1,s_2) + \alpha(r + \gamma Q_t(s_2, \pi_\epsilon(s_2)) - Q_t(s_1,a_2)) \\ &= Q_t(s_1,s_2) + \alpha(r + \gamma Q_t(s_2,a_2) - Q_t(s_1,a_2)) \\ &= 2 + 0.1 \cdot (3 + (0.5 \cdot 4) - 2) \\ &= 2 + (0.1 \cdot 3) \\ &= 2 + 0.3 \\ &= 2.3 \end{aligned}$$

Q-learning:

$$\begin{aligned} Q_{t+1}(s_1,a_2) &= Q_t(s_1,s_2) + \alpha(r + \gamma \max_{a'} Q_t(s_2,a') - Q_t(s_1,a_2)) \\ &= Q_t(s_1,s_2) + \alpha(r + (\gamma \cdot 4) - Q_t(s_1,a_2)) \quad (a_2 \text{ maximizes the value}) \\ &= 2 + 0.1 \cdot (3 + (0.5 \cdot 4) - 2) \\ &= 2 + (0.1 \cdot 3) \\ &= 2 + 0.3 \\ &= 2.3 \end{aligned}$$

The target action both for *Sarsa* and for *Q-learning* is the same (a_2). For this reason the updated Q-table is for both cases:

$$Q_{t+1}(s,a) = \begin{pmatrix} 1 & 2.3 \\ 3 & 4 \end{pmatrix} \tag{3}$$

1.2

Prove that:

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \gamma^{k-t}\delta_k \tag{4}$$

where $G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$ and δ_k is the TD error at time step k.
The reasoning started by taking a look to the n-step TD algorithm illustrated in Figure 2.

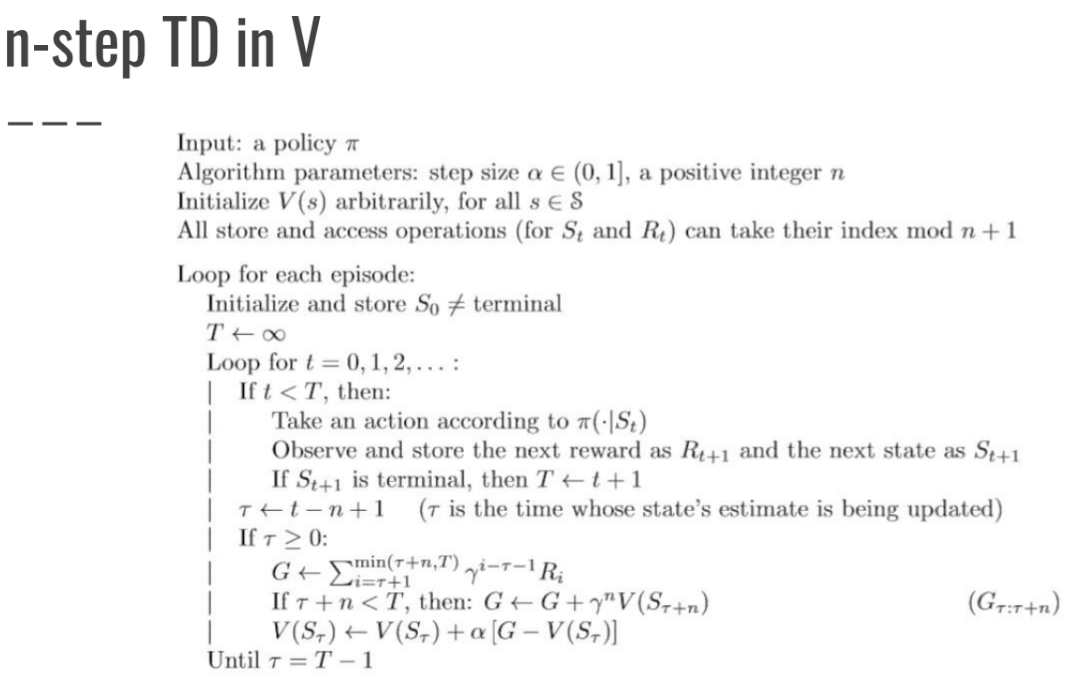

Figure 2

In the algorithm, the definition of $G_{t:t+n}$ is exactly the one written in equation 4, but with τ instead of t ($G_{\tau:\tau+n}$). If we see carefully, t refers to the iterations of the algorithm and τ (equals to $t-n+1$) refers to the time whose state's estimate is being updated. In order to stick to algorithm's nomenclature, from now on we will substitute t with τ.
Indeed:

$$G_{\tau:\tau+n} - V_{\tau+n-1}(S_\tau) = \sum_{k=\tau}^{\tau+n-1} \gamma^{k-\tau}\delta_k \tag{5}$$

and since $\tau = t - n + 1$, we can substitute $\tau + n - 1$ with t. Therefore, the equation becomes:

$$G_{\tau:\tau+n} - V_t(S_\tau) = \sum_{k=\tau}^{t} \gamma^{k-\tau} \delta_k \tag{6}$$

We simply changed variables names. Let's prove equation 6!
For illustration purposes, we highlight and name the two parts of the equation in this way:

$$G_{\tau:\tau+n} - V_t(S_\tau) = \sum_{k=\tau}^{t} \gamma^{k-\tau} \delta_k$$
$$\mathbf{1} = \mathbf{2}$$

We then expand them:

$$R_{\tau+1} + \gamma R_{\tau+2} + ... + \gamma^{n-1} R_{\tau+n} + \gamma^n V_t(S_{\tau+n}) - V_t(S_\tau) = \delta_\tau + \gamma\delta_{\tau+1} + \gamma^2\delta_{\tau+2} + ... + \gamma^{n-1}\delta_t$$

The aim is now manipulating **1** in order to make it equal to **2**:

$$\begin{aligned}
&\mathbf{1} + \gamma V_t(S_{\tau+1}) - \gamma V_t(S_{\tau+1}) + \\
&+\gamma^2 V_t(S_{\tau+2}) - \gamma^2 V_t(S_{\tau+2}) + \\
&+\gamma^3 V_t(S_{\tau+3}) - \gamma^3 V_t(S_{\tau+3}) + \\
&+ ... + \gamma^{n-1} V_t(S_{\tau+n-1}) - \gamma^{n-1} V_t(S_{\tau+n-1}) = \mathbf{2} \quad \text{(adding and substracting)} \\
&\underbrace{(R_{\tau+1} + \gamma V_t(S_{\tau+1}) - V_t(S_\tau))}_{\delta_\tau} + \\
&\underbrace{\gamma(R_{\tau+2} + \gamma V_t(S_{\tau+2}) - V_t(S_{\tau+1}))}_{\gamma\delta_{\tau+1}} + \\
&\underbrace{\gamma^2(R_{\tau+3} + \gamma V_t(S_{\tau+3}) - V_t(S_{\tau+2}))}_{\gamma^2\delta_{\tau+2}} + \\
&+ ... + \underbrace{\gamma^{n-1}(R_{\tau+n} + \gamma V_t(S_{\tau+n}) - V_t(S_{\tau+n-1}))}_{\gamma^{n-1}\delta_{t=\tau+n-1}} = \mathbf{2} \\
&\delta_\tau + \gamma\delta_{\tau+1} + \gamma^2\delta_{\tau+2} + ... + \gamma^{n-1}\delta_t = \mathbf{2} \quad \text{q.e.d.}
\end{aligned}$$

2 Code

2.1

Sarsa-λ algorithm is clearly depicted in Figure 3.

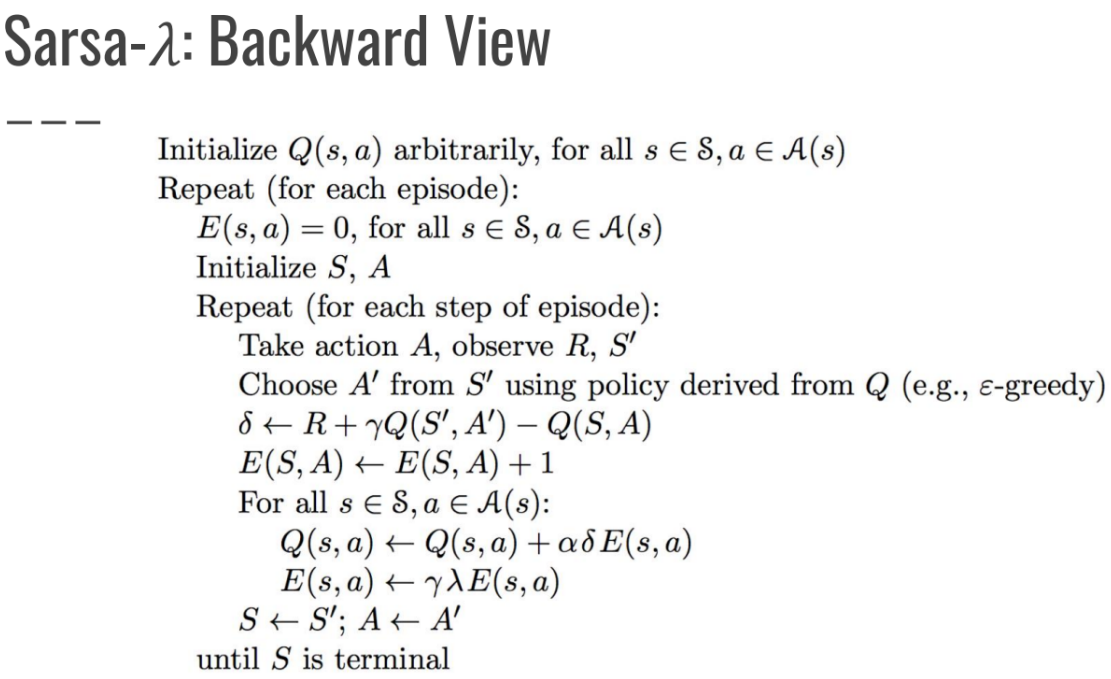

Figure 3

It can be noticed how, for each starting episode, the *eligibility traces* table $E(s,a)$ is re-initialized to zero. The first challenge to me was understand in which line of the code putting $E(s,a) \leftarrow \gamma\lambda E(s,a)$. This because in other slides of the course material the $E(s,a)$ update was indicated before the $Q(s,a)$ update. After a thorough inspection, I recognize that doing:

```
# first version
E *= (gamma * lambda_)
E[state, action] += 1
Q += (alpha * td_error * E)
```

or doing (like the algorithm in Figure 3):

```
# second version
E[state, action] += 1
Q += (alpha * td_error * E)
E *= (gamma * lambda_)
```

is exactly the same in terms of *eligibility traces* computation, and indeed the outcome of the algorithm in general. The second version, basically anticipates the computation of $E(s,a)$ w.r.t. the first version. They produces the same updates because, at the last iteration (when $done == True$), the update is actually not used by the second

version of the algorithm and doesn't influence in any way the Q- table. In fact, by running 1 million of episodes at testing time, the mean rewards are pretty the same: 7.12 for the first version and 7.07 for the second one!
I also tried an incorrect version of the algorithm (always changing position of $E(s, a) \leftarrow \gamma\lambda E(s, a)$):

```
# incorrect version
E[state, action] += 1
E *= (gamma * lambda_)
Q += (alpha * td_error * E)
```

The results are quite surprising because the mean reward after 1 million episodes is of 6.5: not so bad!
The submitted solution for the assignment employs the first version.

2.2

The first thing to implement was the RBF encoder. Without losing time, I immediately implemented it by using the *sklearn* version. Among the hyperparameters to set, the most important one whose decision had a preponderant weight on the outcome of the algorithm was the dimensionality of the computed feature space (*feature_size* in my code).
We could implement either the *forward* or the *backward* view version of the Q-learning TD(λ) with linear approximation algorithm. I decided to implement the *forward* view version. After many trials, I set *feature_size* of RBF encoder equals to 20, since it was the number of features which produced the best results compared to all the other tried. The implemented *forward* view algorithm performed very well: after 10.000 testing episodes the mean reward was -146.63! The produced linear model (*model.pkl*) is actually the one choosen for the submission.
I was also curious about the *backward* view version and I implemented it. After only 50 training episodes, it reaches a mean reward of -150.99 (in 10.000 testing episodes). This to demonstrate its effectiveness and speed in learning.