University of Rome "La Sapienza"
Department of Ingegneria Informatica, Automatica e Gestionale

Reinforcement Learning

Assignment 2

Sarsa-λ and Linear Q approximation

Gianmarco Scarano

Matricola:
2047315

November 21, 2023

Contents

Chapter 1

Theory

1.1 Q-Learning / SARSA Exercise

1.1.1 Q-Learning

From theory, one knows that the Q-Learning algorithm is explained as follows:

Algorithm 1 Q-Learning

Initialize $Q(s,a), \forall s \in S, a \in A(s); Q(terminal-state, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of the episode):
        Take action A, observe R, S'
        Choose A from S using policy derived from Q (e.g. ϵ-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \cdot \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$;
    until S is terminal

So, simply apply this definition on our problem.

$Q(1,2) = 2 + 0.1 \cdot [3 + 0.5 \cdot 4 - 2]$

Here we take $\max_a Q(S',a) = Q(2,2)$

$Q(1,2) = 2 + 0.1 \cdot [3 + 2 - 2]$
$Q(1,2) = 2 + 0.1 \cdot 3$

Finally, $Q(1,2) = 2.3$.

1.1.2 SARSA

From theory, one knows that the SARSA algorithm is explained as follows:

Algorithm 2 SARSA

Initialize $Q(s,a), \forall s \in S, a \in A(s); Q(terminal-state,\cdot) = 0$
Repeat (for each episode):
    Initialize S
        Choose A from S using policy derived from Q (e.g. ϵ-greedy)
            Repeat (for each step of the episode):
                Take action A, observe R, S'
                Choose A' from S' using policy derived from Q (e.g. ϵ-greedy)
                $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \cdot Q(S^{'}, A^{'}) - Q(S,A)]$
                $S \leftarrow S'; A \leftarrow A';$
            until S is terminal

So, simply apply this definition on our problem.

$Q(1,2) = 2 + 0.1 \cdot [3 + 0.5 \cdot 4 - 2]$

Here we take $Q(S^{'}, A^{'}) = Q(2,2)$

$Q(1,2) = 2 + 0.1 \cdot [3 + 2 - 2]$
$Q(1,2) = 2 + 0.1 \cdot 3$

Finally, $Q(1,2) = 2.3$.

1.2 N-step formula

If we want to prove that the n-step error can also be written as a sum of TD errors if the value estimates don't change from step to step, we can proceed as follows:

We know, from the Assignment PDF, that $G_{t:t+n}$ is equal to:

$$G_{t:t+n} = R_{t+1} + \gamma \cdot R_{t+2} + ... + \gamma^{n-1} \cdot R_{t+n} + \gamma^n \cdot V_{t+n-1}(S_{t+n}) \tag{1.1}$$

If we change this in the general formula and considering that V does not change overtime (namely, we remove the time step over V), one easily obtains:

$$G_{t:t+n} - V_{t+n-1}(S_t) = R_{t+1} + \gamma \cdot R_{t+2} + ... + \gamma^{n-1} \cdot R_{t+n} + \gamma^n \cdot V(S_{t+n}) - V(S_t) \tag{1.2}$$

We know, from theory, that the TD error is defined as follows:

$$\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t) \tag{1.3}$$

Since in (1.2), we are treating only the rewards $(R_{t+1}, R_{t+2}, ..., R_{t+n})$, let's isolate it in (1.3), obtaining that:

$$R_{t+1} = \delta_t - \gamma \cdot V(S_{t+1}) + V(S_t) \tag{1.4}$$

Let's simply replace this in our formula (1.2), which now becomes:

$$\begin{aligned} G_{t:t+n} - V_{t+n-1}(S_t) = \delta_t - \gamma \cdot V(S_{t+1}) + V(S_t) \\ + \gamma \cdot R_{t+2} + ... + \gamma^{n-1} \cdot R_{t+n} \\ + \gamma^n \cdot V(S_{t+n}) - V(S_t), \end{aligned} \tag{1.5}$$

Now, the first three elements of (1.5), can be easily multiplied by any quantity, in this case we decide to multiply them by γ^0, which is equal to 1 (effectively not compromising the formula at all).

Doing this and iterating the same process for $R_{t+1}, R_{t+2}, ..., R_{t+n}$, one obtains:

$$\begin{aligned} G_{t:t+n} - V_{t+n-1}(S_t) = \gamma^0[\delta_t - \gamma \cdot V(S_{t+1}) + V(S_t)] + \\ + \gamma^1[\delta_{t+1} - \gamma \cdot V(S_{t+2}) + V(S_{t+1})] + ... + \\ + \gamma^{n-1}[\delta_{t+n-1} - \gamma \cdot V(S_{t+n}) + V(S_{t+n-1})] + \\ + \gamma^n \cdot V(S_{t+n}) - V(S_t), \end{aligned} \tag{1.6}$$

We can see this whole thing as a summation operation $(\sum)$ except for the last 2 terms $\gamma^n \cdot V(S_{t+n}) - V(S_t)$, in fact we can now rewrite this as:

$$\begin{aligned} G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} [\gamma^{k-t} \cdot \delta_k - \gamma^{k-t+1} \cdot V(S_{k+1}) + \gamma^{k-t} \cdot V(S_k)] + \\ + \gamma^n \cdot V(S_{t+n}) - V(S_t) \end{aligned} \tag{1.7}$$

What we are about to do now is some manipulation of the summation operation in order to cancel out things and re-organize the formula, getting to the final step, which is what we want to prove.

We can split up the whole summation of (1.7) as a sum of the summation operations as follows:

$$\begin{aligned} &\sum_{k=t}^{t+n-1} [\gamma^{k-t} \cdot \delta_k - \gamma^{k-t+1} \cdot V(S_{k+1}) + \gamma^{k-t} \cdot V(S_k)] \\ &= \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot \delta_k - \sum_{k=t}^{t+n-1} \gamma^{k-t+1} \cdot V(S_{k+1}) + \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot V(S_k) \end{aligned} \tag{1.8}$$

If we now take the the middle object from the summation (1.8) at step $t+n-1$, we would get $-\gamma^n \cdot V(S_{t+n})$ which cancels out with the $\gamma^n \cdot V(S_{t+n})$ from (1.7).

Of course, the manipulation of this summation object we've just treated, let us remain with the summation up to step $t+n-2$, so we can rewrite (1.8) as:

$$\sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot \delta_k - \sum_{k=t}^{t+n-2} \gamma^{k-t+1} \cdot V(S_{k+1}) + \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot V(S_k) \tag{1.9}$$

and in the same way, we can now rewrite the whole (1.7) formula as:

$$\begin{aligned} G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot \delta_k - \sum_{k=t}^{t+n-2} \gamma^{k-t+1} \cdot V(S_{k+1}) + \\ + \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot V(S_k) - V(S_t) \end{aligned} \tag{1.10}$$

In formula (1.9), we can observe that, as we did before, the last term of the summation can easily cancel out with $V(S_t)$ if we take it at the first step $k=t$.

In fact, when $k=t$, the last summation operation is equal to $\gamma^0 \cdot V(S_t)$ which cancels out with $V(S_t)$.

Of course, as we did in (1.9), we would now reformulate the last summation to start from $t+1$ instead of just t.

We are then left, with this final formulation:

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot \delta_k - \sum_{k=t}^{t+n-2} \gamma^{k-t+1} \cdot V(S_{k+1}) + \sum_{k=t+1}^{t+n-1} \gamma^{k-t} \cdot V(S_k) \tag{1.11}$$

Concluding this exercise, we now recall a corollary coming from the Theorem of the translation of index variable of the summation operation, which states that:

$$\sum_{k=n}^{m} f(x) = \sum_{k=n+l}^{m+l} f(x-l)$$

one could rewrite $\sum_{k=t}^{t+n-2} \gamma^{k-t+1} \cdot V(S_{k+1}) = \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot V(S_k)$

If we plug this in (1.11), one obtains:

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot \delta_k - \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot V(S_k) + \sum_{k=t+1}^{t+n-1} \gamma^{k-t} \cdot V(S_k) \tag{1.12}$$

Finally, the last two terms cancel out and we are left with only one object which is exactly what we wanted to prove:

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot \delta_k \tag{1.13}$$

Chapter 2

Practice

2.1 Sarsa-λ

I edited the `student.py` file in order to fill in the function `epsilon_greedy_action()` and the `sarsa_lambda()` function as well, effectively computing the Q-Table and the eligibility traces.

With this being said, let's analyze the functions:

- ***epsilon_greedy_action()***:

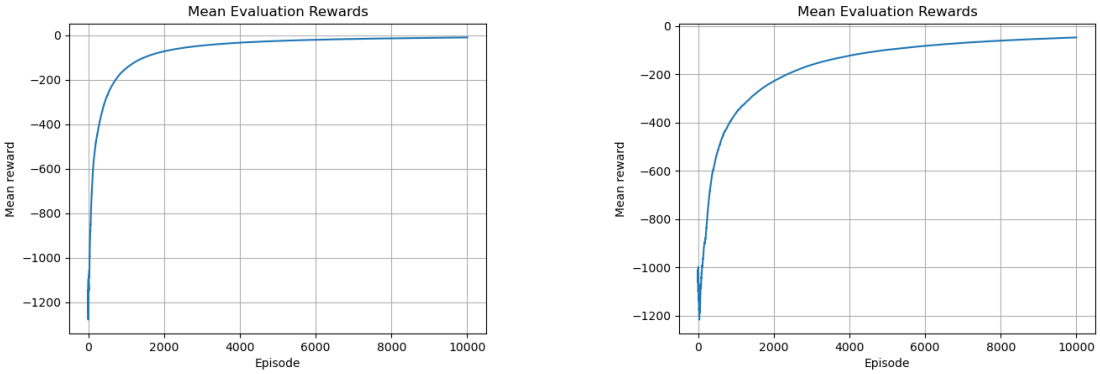
  As we have seen multiple times in class, the epsilon greedy action can be achieved through a simple `if` statement where we take the `argmax` over the Q-Table at that specific state if a random probability (between 0 and 1) is lower than a certain threshold `epsilon`.

- ***sarsa_lambda()***:

  The task here was to write the update step for the table `Q` and traces `E`. I followed the pseudo-code from the slides `07 - n-step bootstrapping` at page 71. Note that we do initialize the Q-Table giving values $[0, 1)$ through a Uniform distribution, instead of starting from 0. This should give some sort of boost to the agent instead of learning from completely nothing. Also, the Eligibility Traces array has been reset at the start of each episode (as in the Sarsa-λ pseudocode. The rest of the formula is pretty straightforward.

- ***BONUS: evaluateQTable()***:

  In order to evaluate our Q-Table as we did in class during the Q-Learning practical, I implemented the `evaluateQTable()` function which evaluates the learnt Q-Table at that specific time-step, by averaging the returns of 10 episodes. This helped me optimizing the parameters for the `lambda` value and also having a visual representation of the values learnt by the agent, as we can see below.



(a) $\lambda = 0.5$ (b) $\lambda = 0.9$

Figure 1: Mean evaluation rewards of 10 episodes at each training episode.

These are the results from 3 runs of the script with $\lambda = 0.5$:

- Mean reward over 10 episodes: 8.2
- Mean reward over 10 episodes: 6.8
- Mean reward over 10 episodes: **9.5**

2.2 Q-Learning TD(λ) via RBF Encoder

In the Q-Learning TD(λ) exercise, I had to update the weights and the eligibility traces through the `update_transition()` function, followed by a proper initialization of the RBFFeatureEncoder class.

- ***update_transition()***:

  In this function I did update the weights and the traces using the formula which can be found on the slide `08 - Linear Approximation` at page 29. In order to get e_{t-1}, I had to update the whole traces matrix by multiplying it by γ and λ. The gradient $\nabla_w Q(s_t, a_t, W)$ is simply taking the features of state s.

  The weight update rule is given as follows:

  $$w_{t+1} = w_t - \alpha \cdot \delta_t \cdot e_t$$

  But, due to the fact that δ is always negative (thanks to the nature of the negative reward of the MountainCar environment), we reformulate the last step of the weight update to be:

  $$w_{t+1} = w_t + \alpha \cdot \delta_t \cdot e_t$$

  such that the difference operator appears before the α term, otherwise we would've been stuck with a + sign. The rest is straightforward.

- ***RBFFeatureEncoder***:

  For the feature encoding part, I joined two `RBFSampler()` taken from the library `sklearn` using the `sklearn.pipeline.FeatureUnion()` function, with a γ value equal to 1.0 and 0.5 respectively. Both the encoders had a number of output features equal to 100, meaning that it we pass a state with shape (1, 2), the final total features encoding would have shape (1, 200). For this reason the function `self.feature_encoder.size` returns a value of 200.

  After initializing the encoder, I initialized a `StandardScaler()` (always from `sklearn`) as well for scaling purposes.

  In order to fit both the `RBFSampler()` and the `StandardScaler()`, I've decided to create an array of experiences which collects around 20000 experiences from the Mountain Car game. Then, very simply I fit both objects on these past experiences in order to have a well-initialized feature encoder and scaler.

  For passing from a state representation to a feature representation, I first call the `scaler.transform()` function which scales my state values and the output of this function will be directly passed to the `encoder.transform()` method.

By running the script, the model achieves from **-99.8** to -115.7 as mean reward (depends on the initial state).

Chapter 3

Collaborations

I discussed this assignment with the following students:

- Giancarlo Tedesco