EXERCISE 12
Eligibility traces and value-function approximations

Tue Herlau
tuhe@dtu.dk

25 April, 2025

Objective: Todays exercise will take a closer look at both value function approximations and value-function based methods. We will look at eligibility traces, which can be though of as a way to interpolate between shallow value-function based methods such as $\mathrm{TD}(0)$ and Monte-Carlo methods, and can be applied to both estimation and control. In addition, eligibility traces allows the so-called backwards view, in which we can overcome the need to keep an annoying buffer of past observations as we had to in n-step estimation last week. (33 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex12.html

Contents

1 Conceptual question

Solve [SB18, Problem 12.1]: *Just as the return can be written recursively in terms of the first reward and itself one-step later [SB18, Equation 3.9], so can the return. Derive the analogous recursive relationship from [SB18, Equation 12.2] and [SB18, Equation 12.1].*

2 Sarsa(λ) (`sarsa_lambda_agent.py`)

The disadvantage of the n-step methods is they require us to keep track of the past state history, they have a fairly complicated update history, and that we have to choose one particular n. The eligibility trace based methods (λ-methods) does away with the n by keeping an eligibility trace. In [SB18] the Sarsa method is introduced in the context of value-function approximations, but we will first consider the case without value function estimators as it is easier to implement and understand.

The eligibility trace is a fairly simple mechanism which just keeps track of previous states visited by the learning method. The method we will implement is given here in the first version of [SB18]: `http://incompleteideas.net/book/first/ebook/node77.html`. Note there is a typo in the book: Whenever the episode terminates, the eligibility trace has to be cleared, i.e. all elements set to zero.

The eligibility trace itself, $e(s, a)$, is indexed the same way as the Q-values $Q(s, a)$, and we will therefore re-use the datastructure for ease.

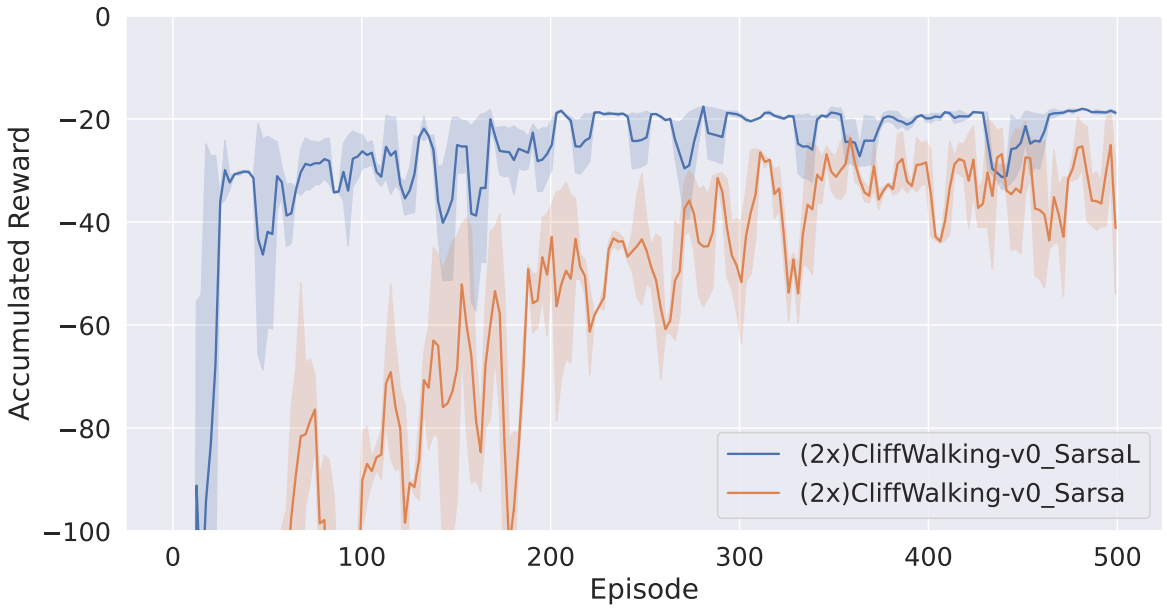
Note that $\mathrm{Sarsa}(\lambda)$ share many similarities with Sarsa, and it is worth comparing to that implementation for ideas. For instance, we can directly re-use the policy function. As an additional note, the implementation referenced above has a slight defect, in that the eligibility trace is not reset. We have included that step in the existing code.

Problem 1 $\mathrm{Sarsa}(\lambda)$

Complete the implementation of the $\mathrm{Sarsa}(\lambda)$ agent and test it on the cliffwalking environment.

For low learning rates, the $\mathrm{Sarsa}(\lambda)$ controller is superior to the regular Sarsa method, however this result, at least on this environment, does not hold for larger learning rates. Can you explain why?

Info: My results, based on 10 runs, are as follows:



2.1 Sarsa(λ) in the gridworld environment (`sarsa_lambda_open.py`)

This is a bit of a strange exercise because it does not involve implementing any code. Instead, run the script and note the first part instantiates a Sarsa(λ) agent in a (open) gridworld environment with a single $+1$ reward in the bottom right corner (same example as we have seen in the lectures). The first part evaluates the Sarsa(λ) agent for a few episodes and save the result as a pdf file, while the second allows you to specify the actions of the agent using the keyboard.

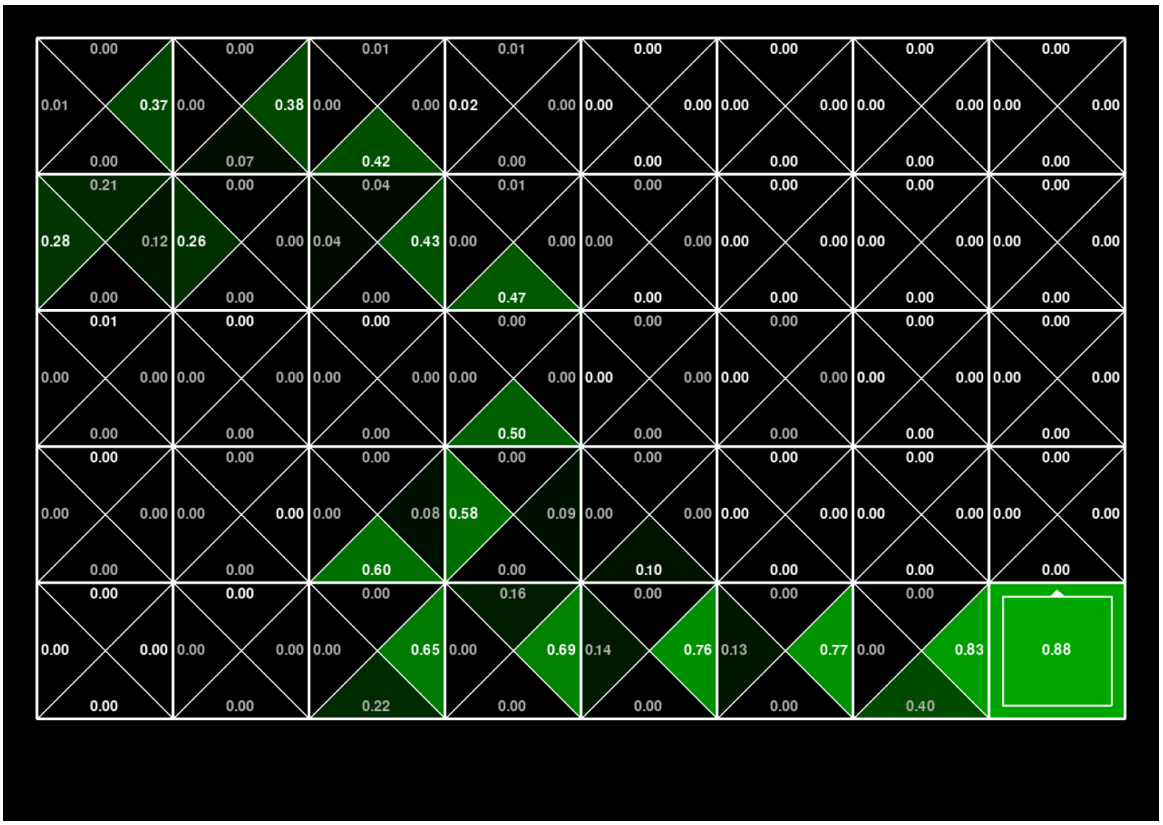
The way this works is (effectively) that we over-write the `def pi`-method with the actions from the keyboard, and these same actions (along with states and rewards) are then passed onto the `train`-function. This works okay-ish for a quick demo, but it is not actually robust: It will sometimes lead to the wrong updates of the Q-values. You can always let the agent play (or stop) by pressing `p`.

Problem 2 Sarsa(λ) *and the open gridworld*

Run the keyboard player. Why does the keyboard player sometimes (but not always!) lead to wrong Q-value updates? (wrong is in the sense that given the trajectory of the agent, we would expect the Q-values to change differently). Can you make an example with keyboard input where the Q-values are updated incorrectly? Is Q-learning susceptible to the same problem?

Discuss what would potentially need to change to fix the issue.

Info: The problem has to do with where in the code the actions are actually generated. The output of the first part of the script will be highly variable, however one run looks as follows:



3 Linear approximators and multi-step methods

We will now combine the n-step Sarsa and $\mathrm{Sarsa}(\lambda)$ methods with linear feature approximators. This is an important step towards combining deep learning and value-function based approximation, however we will postpone this work to next week because of the increased complexity of introducing a deep learning framework.

Implementation-wise $\mathrm{Sarsa}(\lambda)$ with linear approximators is very reminiscent to $\mathrm{Sarsa}(\lambda)$ (above) and the semi-gradient version of Sarsa we saw last week, i.e. it is recommended to complete these exercises first.

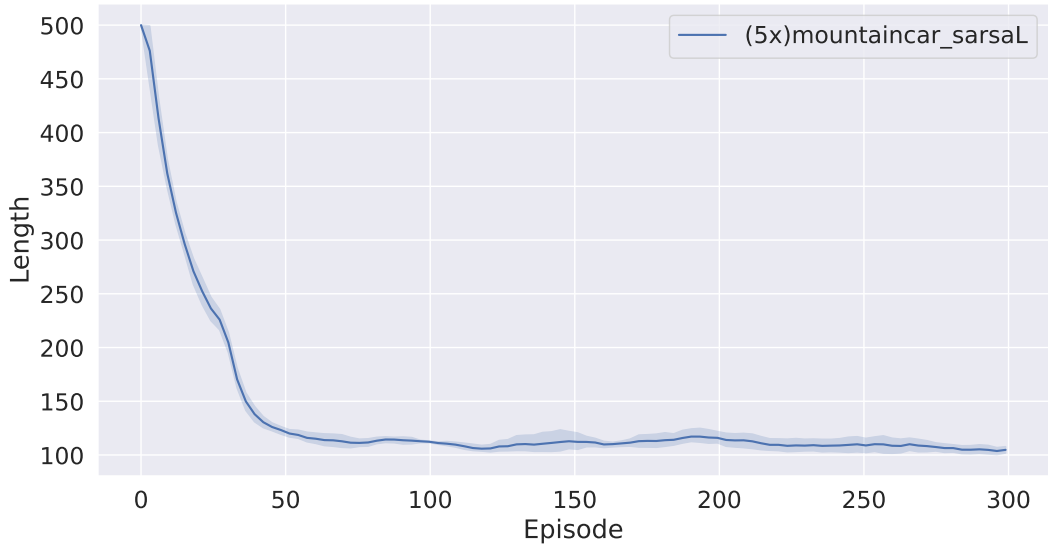
3.1 Semi-gradient $\mathrm{Sarsa}(\lambda)$ (`semi_grad_sarsa_lambda.py`)

We will now generalize the $\mathrm{Sarsa}(\lambda)$ method to include linear feature approximators (binary features). Note there are different (and very similar) variants in [SB18] of this method, and we will particular consider the last algorithm in [SB18, Section 12.7], just before section 12.8. See also todays lecture slides.

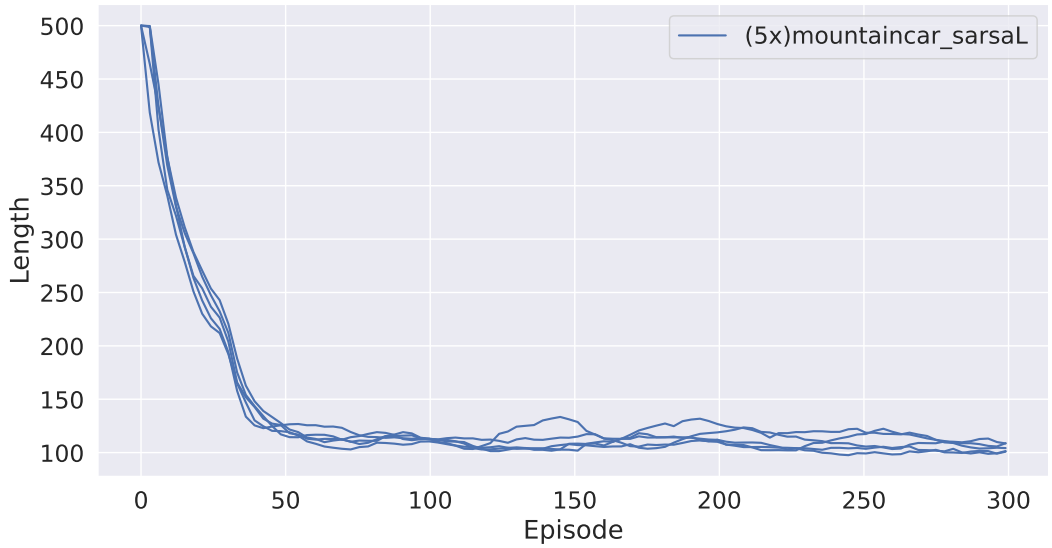
Problem 3 *Semi-gradient* $\mathrm{Sarsa}(\lambda)$

Complete the implementation of the semi-gradient $\mathrm{Sarsa}(\lambda)$ agent and test it on the mountain-car example. Produce plots where multiple runs are averaged and where the individual plots are considered.

Info: I obtain the following result:



Once more it is informative to consider the individual runs reproduced below:



we see a tendency that much of the performance is driven by outliers, and Sarsa-λ appears slightly better than the alternative methods. However, we would need to check multiple environments and settings of parameters to make sure this is a robust effect.

3.2 Semigrad n-step Sarsa (`semi_grad_nstep_sarsa.py`) ★★

For completeness, we will also combine the n-step Sarsa method from the past week with linear value function approximations. This problem is not exam relevant.

If we carefully comparing the method we will implement, [SB18, Section 10.2], with the version from last week ([SB18, Section 7.2]) will reveal the only meaningful changes when we change from tabular to linear methods are when the value functions $\hat{q}_w(s,a)$ are computed and the weights are updated using the gradient $\nabla\hat{q}_w(s,a) = \boldsymbol{x}(s,a)$, i.e. as in the substitution from a tabular update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(G^{(n)} - Q(s,a))$$

to a weight-based update rule:

$$w \leftarrow w + \alpha(G^{(n)} - \hat{q}_w(s,a))\nabla\hat{q}_w(s,a) \tag{1}$$

In other words, we can simply think of this as the n-step algorithm but with these two changes.

There are two ways to proceed: either copy-paste the code from last time (and include these two changes), or combine the existing methods using inheritance and re-use the existing code. As the later is the two-line solution, it is what we will pursue here.

Our solution will inherit from both the `SarsaNAgent` (to get the correct training method) and `LinearSemiGradSarsa` (to get the $\hat{q}$ data-structure) and the full constructor looks as follows:

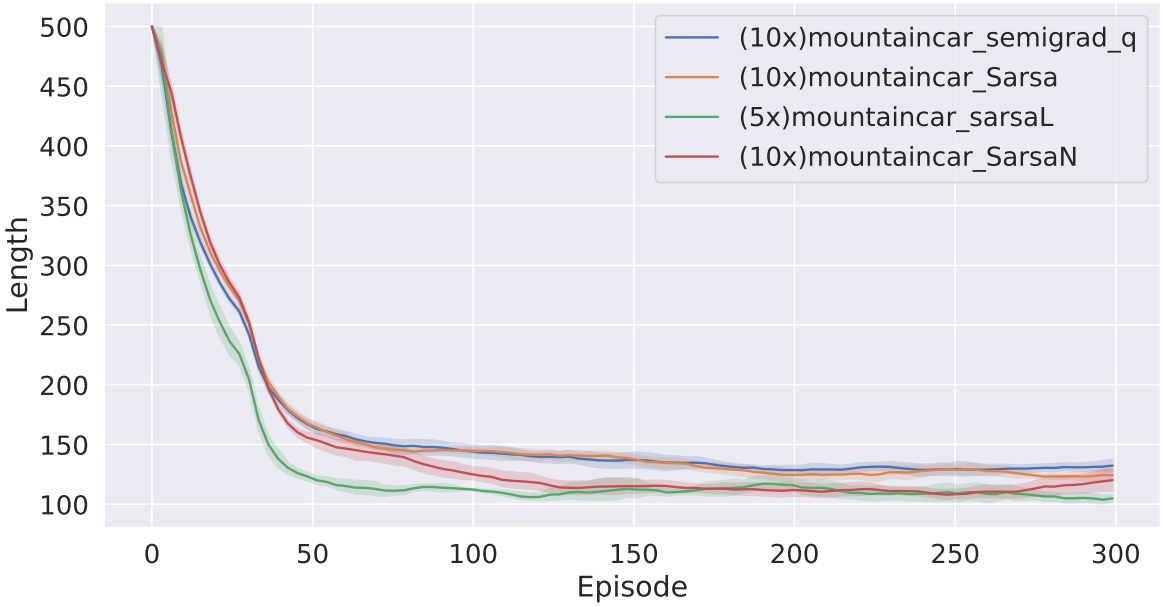
```python
# semi_grad_nstep_sarsa.py
class LinearSemiGradSarsaN(SarsaNAgent, LinearSemiGradSarsa):
    def __init__(self, env, gamma=0.99, alpha=0.5, epsilon=0.1, q_encoder=None, n=1):
        """
        Note you can access the super-classes as:
        >> SarsaNAgent.pi(self, s) # Call the pi(s) as implemented in SarsaNAgent
        Alternatively, just inherit from Agent and set up data structure as required.
        """
        SarsaNAgent.__init__(self, env, gamma, alpha=alpha, epsilon=epsilon, n=n)
        LinearSemiGradSarsa.__init__(self, env, gamma, alpha=alpha, epsilon=epsilon,
        ↪  q_encoder=q_encoder)
```

From this, it is a matter of delegating the calls in `SarsaNAgent` which has to do with either invoking the policy `pi(s)` or accessing/updating the `self.Q[s][a]` -values to functions which we can overwrite and implement using the data structure defined in `LinearSemiGradSarsa`. These methods are `def _q(self, s, a)` (which returns $Q(s,a)$) and `def _upd_q(self, s, a, delta)` (which performs the update to the weight-vector as in eq. (1)). In other words, first re-factor the `SarsaNAgent` to only use these methods, then implement our version of these methods which uses the linear feature approximator in `LinearSemiGradSarsaN`.

Problem 4 Semigrad n-step Sarsa

Complete the implementation of the semi-gradient n-step Sarsa agent from [SB18, Section 10.2] and evaluate it on the Mountain-car task. I did not tune the learning rate α, and other choices might produce superior results.

Info: I get the following results:



However, note these results can probably be improved by tweaking α and seem to be quite driven by poor runs.

4 A closer look at the mountain car example (`mountaincar.py`)

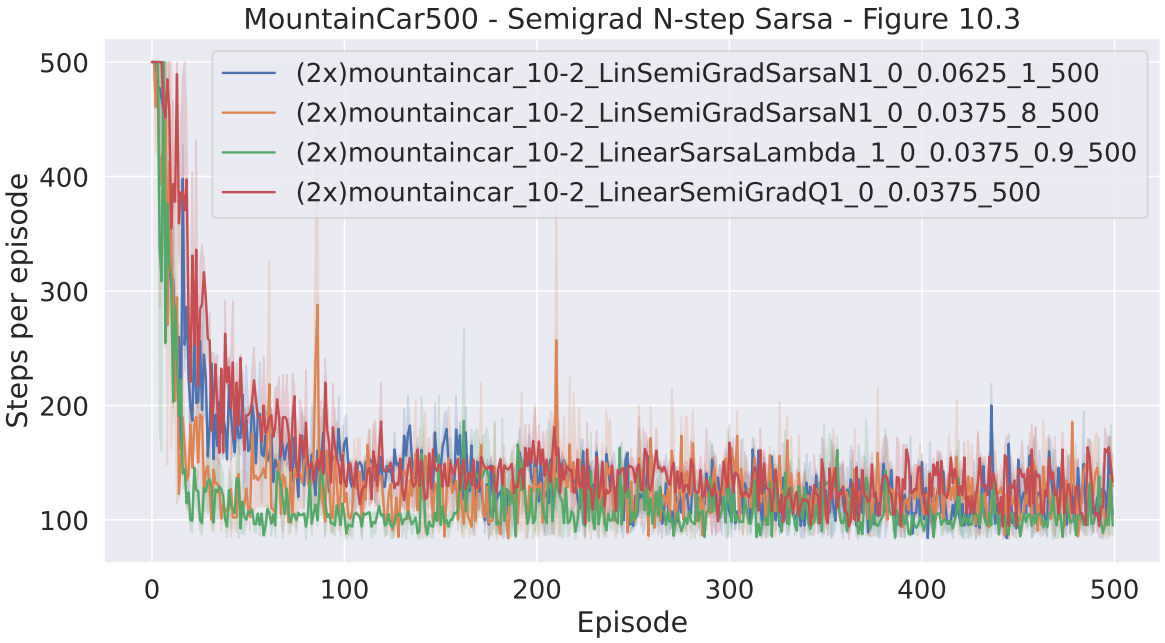
★★

It should now be possible to reproduce the results of MountainCar example. Plots to reproduce four of the figures in chapter 10 can be found in `mountaincar.py`, but for this problem we will focus on a simple comparison of our three methods which use linear interpolation from [SB18, Figure 10.3]. Note the problem is entirely about calling our methods with the correct parameters, however it might be helpful for the projects as it contains code to visualize the value function etc.

Problem 5 *Comparing methods on MountainCar*

Perform comparison of Q, Sarsa, n-step Sarsa, and $\mathrm{Sarsa}(\lambda)$ on the mountaincar task.

Info: I obtain the following result:



Note these results are sensitive to learning rates, etc. and you can likely improve by using different settings; if the results can be severely overturned, please let me know.

References

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (Freely available online).