EXERCISE 11
Model-Free Control with tabular and linear methods

Tue Herlau
tuhe@dtu.dk

25 April, 2025

Objective: Value-function based methods can often converge faster than Monte-Carlo methods and form the basis of several recent successes of reinforcement learning. We will start by investigating two important control methods which use action-value functions: Sarsa learning, which is on-policy, and Q-learning which is off policy. In the later part of the exercise we will investigate n-step TD learning, which combines advantages of MC and TD(0) learning, a subject which will lead to eligibility traces next week.

In the last part of todays exercise we will look at value function approximations which are essential for scaling up reinforcement learning. These introduce a number of complications and design choices we will return to in two weeks, but to avoid much of that discussion we will focus on the important case of linear approximators.

(22 lines of code)

Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex11.html

Contents

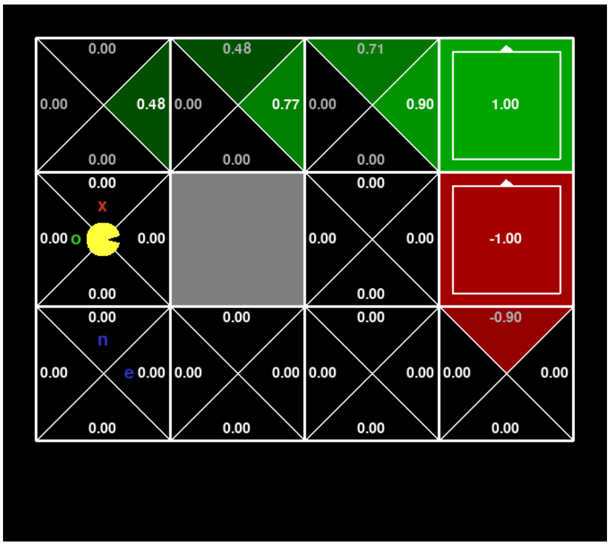


Figure 1: A gridworld environment. The figure shows the current Q-values for Sarsa

1 Exam question: Q-learning and Sarsa

Suppose that Sarsa (using discount factor $\gamma = 1$, a learning rate $\alpha = 0.9$ and an exploration rate of $\varepsilon = 0.1$) is applied to the Gridworld shown in fig. 1. The living reward is 0, and the dynamics is deterministic. Recall that pacman stay at the current state if he choose an action which moves him into a wall. The current state is indicated by Pacman.

(a.) Suppose that in the next step of the Sarsa algorithm, the agent takes (and execute) the action `North` in the current position s. Upon taking this action, the Q-value associated with the red cross $Q(s, \texttt{North})$ will be updated by Sarsa.

What are the possible values of $Q(s, \texttt{North})$ after this step? (list all of them).

(b.) In the previous question, we took one step, performed a single action `North`, and updated one Q-value.

In this question, suppose again that the agent starts in the position indicated in fig. 1 and assume Sarsa is applied to update the Q-values shown in the figure.

Different sequences of future actions will result in different Q-values being updated. What are the minimum number of steps which are required before the Q-value associated with the green circle can take a value different than 0, and what actions will the agent take in this case? Give your answer as a list of actions.

(c.) Sarsa learning has clearly not converged in the example shown above. However, assume we apply a more realistic version of Sarse where $\gamma = .95$, and where importantly α decrease to 0 at a rate satisfying the stochastic approximation conditions [SB18, Eq. (2.7)] for convergence so that the Q-values converge to their *true values* under Sarsa.

Consider the two Q-value associated with moving `North` and `East` Q_n, Q_e, indicated by the blue n, e-letters in fig. 1. After convergence, it must be the case that either they will have the same value, $Q_n = Q_e$, or one will be greater than the other: $Q_n > Q_e$ or $Q_n < Q_e$. State which is the case and provide a clear and specific argument for your answer.

2 Tabular control methods

In this section we will consider the basic (tabular) basic versions of Q and Sarsa learning, and later extend sarsa with linear function approximators (we will return to Q learning when we discuss deep reinforcement learning).

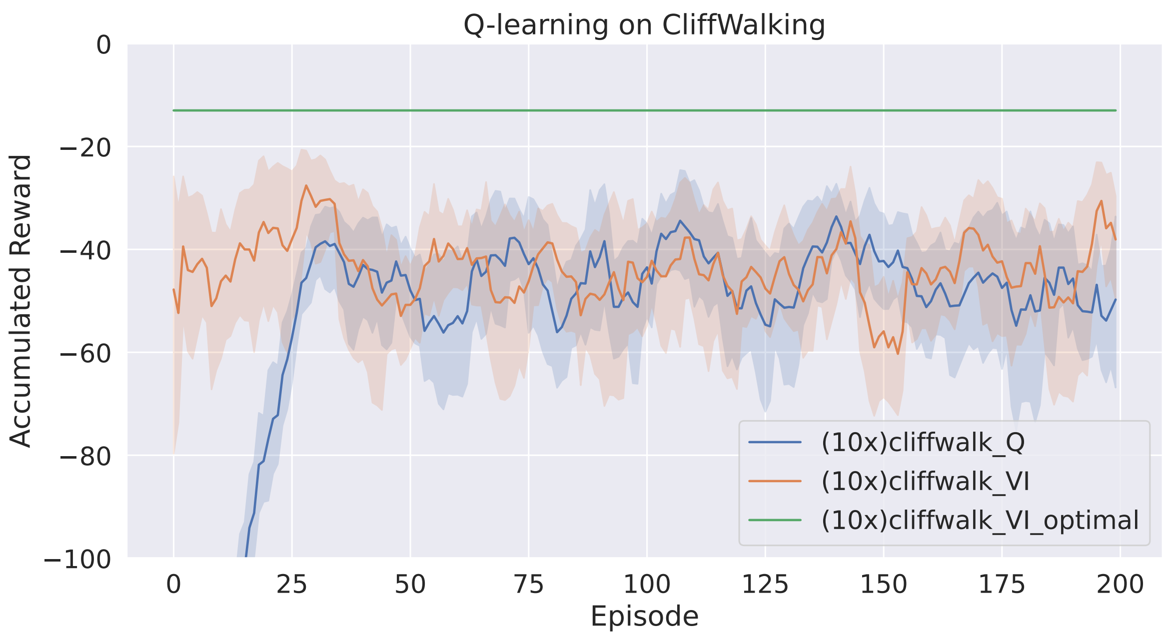
2.1 Q-Learning (`q_agent.py`)

Our implementation will be based on [SB18, Section 6.5], i.e. we update one Q-value at a time and use ε-greedy exploration. Note we will be extending the `Agent`-class we worked with last week and I encourage the reader to look at the code for this class for ideas, for instance the `Agent.pi_eps`-function.

Problem 1 Q-learning

Complete the implementation of the Q-learning agent and test it on the cliff walking task. The obtained result should be comparable to the figure in [SB18, Example 6.6], however I think that this figure is averaged over quite a few runs. I am not entirely sure what α should be, so I selected $\alpha = 0.5$ because this was the value one page back; are other values better?

Info: Writing the Q-learning is about 2-lines of code, however it takes a little work to make it robust to the various use cases we can imagine such as action spaces which depend on state etc. I strongly recommend using the build-in `Agent.Q`-data-structure to store the Q-values as this will make the code significantly more robust.



2.2 Sarsa (`sarsa_agent.py`)

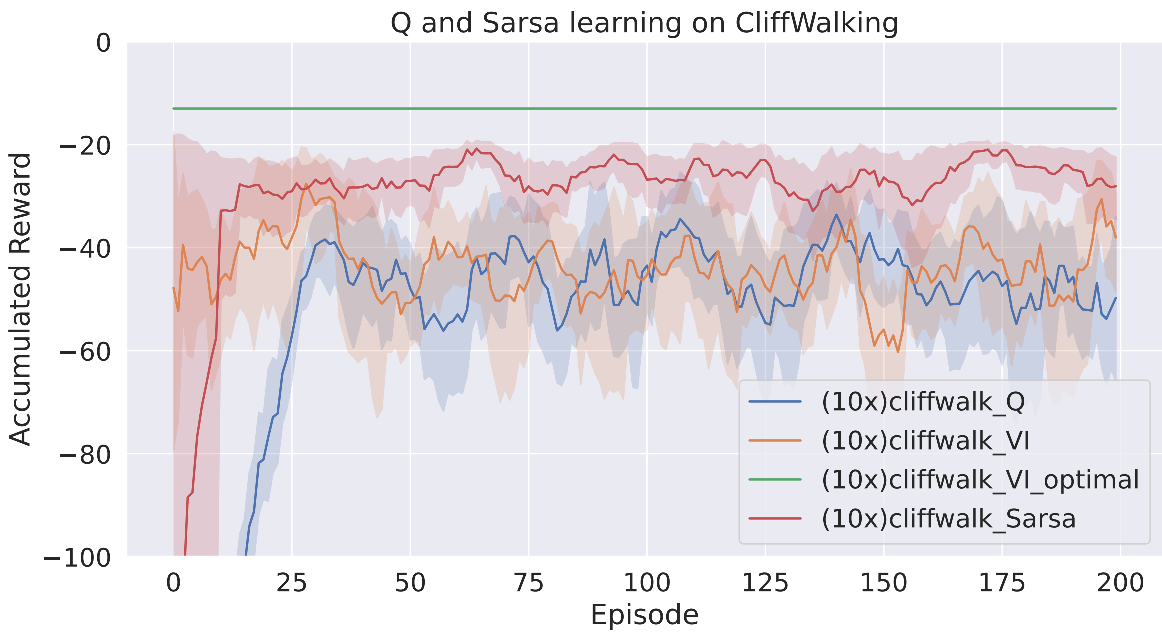
The next method we will consider will be Sarsa for on-policy control as in [SB18, Section 6.4]. Be warned Sarsa is a bit more tricky to implement as it requires us to know the future actions (here, A') "*in advance*" compared to Q-learning.

In practice, this means we must generate the A' values in the `Agent.train` function (where we know S'), store the action, and then simply return the action when π is called. This works well for all steps except the first ($t = 0$) where `agent.train` has not yet been called, and so in the case where $t = 0$ we must actually generate the action similar to Q-learning when the policy is called.

Problem 2 *Sarsa-control*

Complete the implementation of the Sarsa agent and reproduce the result in the figure in [SB18, Example 6.6]; however note this is with the caveat that I am not entirely sure about the parameters/runs used to generate the figure.

Info:



3 Linear feature encoding

In this section we will consider approximations of the value or action-value function. This means that we represent the action-value function $q(s, a)$ using an approximation

$$q(s, a) \approx \hat{q}(s, a, \mathbf{w})$$

depending on a weight-vector $\boldsymbol{w}$ which we have to learn. As discussed in [SB18, Chapter 11] there are many choices available for the function appropriator, and since we do not wish these choices to get in the way of the learning algorithms we will as a first step consider the case where $\hat{q}(s, a, \mathbf{w})$ is linear; as a benefit, this also allows us to be consistent with the methods in [SB18, Chapter 12]. In other words we assume:

$$q(s, a) \approx \hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)^\top \boldsymbol{w} \tag{1}$$

here, $\boldsymbol{x} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}^d$ is a function which maps each state, action pair into a d-dimensional feature vector. We have to choose this function based on one of the methods described in [SB18, Section 9.5]. Note some of the algorithms we will see momentarily are based on the gradient, but this is easy to compute in a linear representation:

$$\nabla \hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)$$

There are a multitude of ways to construct the function $\boldsymbol{x}(s, a)$ (as you can tell in Todays reading from [SB18]), but since the details are not that interesting we will delegate this functionality to a special class. You can find more information in the comments to today's code or in the online documentation.

3.1 Episodic semi-gradient Q-learning (`semi_grad_q.py`)

Even though the book focuses on Sarsa (on policy) learning with approximations, we will first discuss Q-learning as it is simpler and we can re-use our Q-learning code when we later build the Sarsa implementation.

the ideas are easier introduced in the context of Q-learning. The actual change to the basic Q-learning method is one line, namely that instead of doing this update:

$$Q(S, A) \leftarrow Q(S, A) + \alpha\delta, \quad \delta = \left[R + \gamma \max_a Q\left(S', a\right) - Q(S, A)\right]$$

we do this:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}\left(S, A, \mathbf{w}\right)$$

To implement this, we need an actual feature approximator. For that purpose I have made a small tile encoder using the same tile-encoding scheme as Sutton. This means that $\boldsymbol{x}(s, a)$ will be binary. Using the encoder is very easy, the important parts are these:

```python
# semi_grad_q.py
class LinearSemiGradQAgent(QAgent):
    def __init__(self, env, gamma=1.0, alpha=0.5, epsilon=0.1, q_encoder=None):
        """ The Q-values, as implemented using a function approximator, can now be accessed as follows:

        >> self.Q(s,a) # Compute q-value
        >> self.Q.x(s,a) # Compute gradient of the above expression wrt. w
        >> self.Q.w # get weight-vector.

        I would recommend inserting a breakpoint and investigating the above expressions yourself;
        you can of course al check the class LinearQEncoder if you want to see how it is done in practice.
```

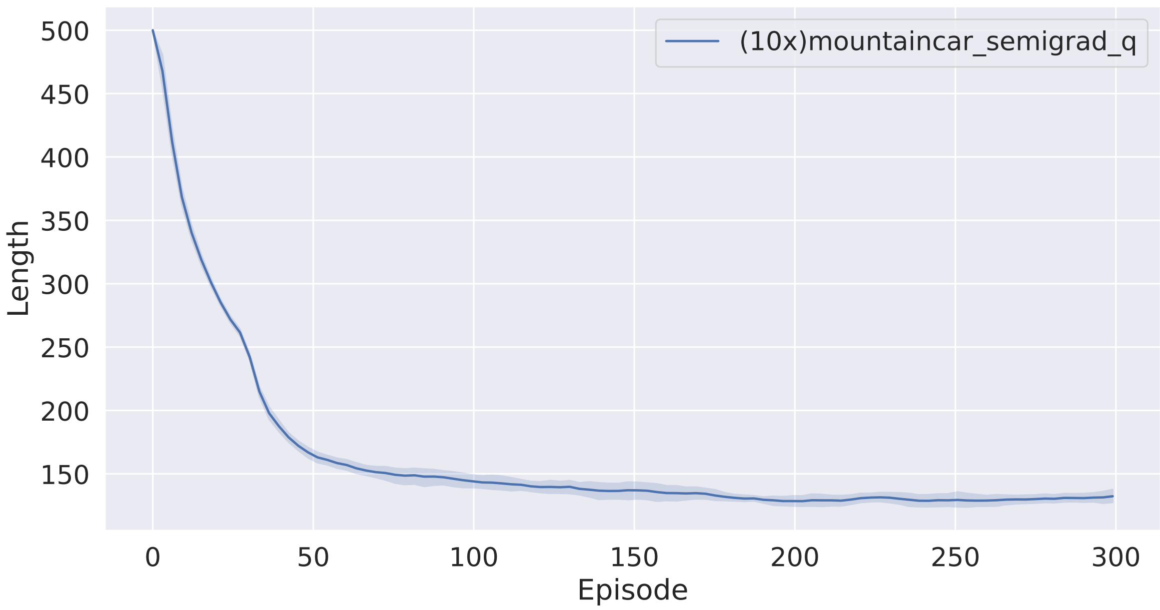
```python
        """
        super().__init__(env, gamma, epsilon=epsilon, alpha=alpha)
        self.Q = LinearQEncoder(env, tilings=8) if q_encoder is None else q_encoder
```

Note in particular the example in the comments: this will be how we access the $\boldsymbol{x}$-feature vector and compute the q-values.

Problem 3 *Semi gradient Q-learning*

Complete the implementation of the linear semi-gradient Q-learning agent. It might be of help comparing the semi-gradient version of Sarsa ([SB18, Section 10.1]) to the tabular version to better understand the new notation.

Info: The script tries to solve the MountainCar example (same setup as [SB18, Figure 10.2]), and you might consider reducing the number of runs while you debug your code. I obtain the following result:



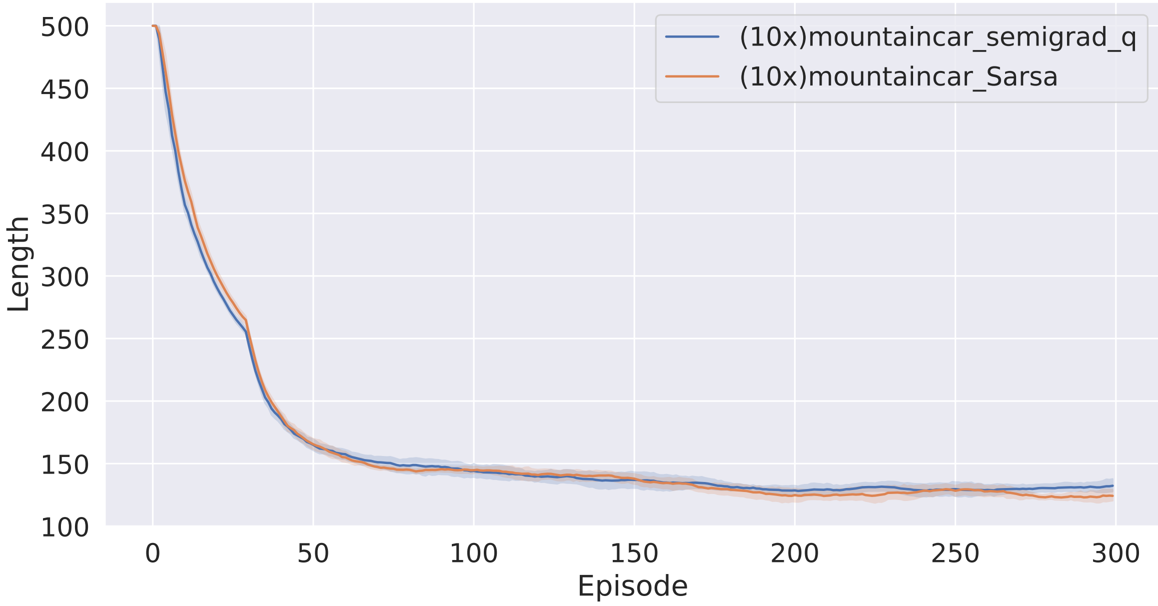
3.2 Episodic semi-gradient Sarsa (`semi_grad_sarsa.py`) ✶✶

Our second task will be episodic semi-gradient Sarsa [SB18, Section 10.1]. The code is nearly identical to the basic Sarsa method we just implemented with the feature-approximator idea added back in, and I would therefore recommend starting with the tabular Sarsa solution.

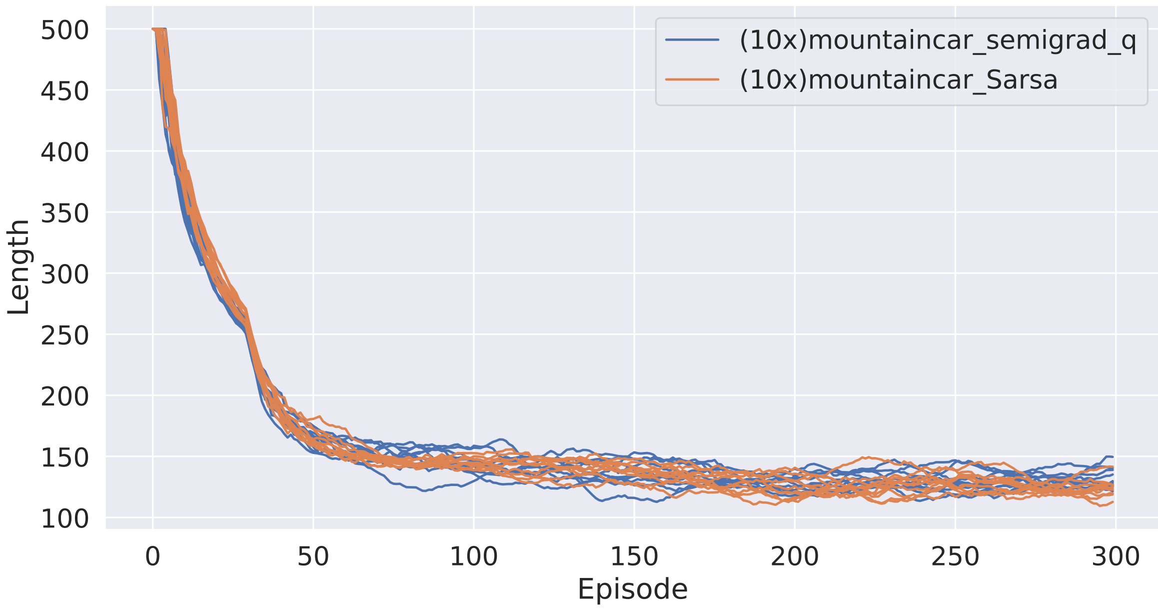
Problem 4 *Semi gradient Sarsa-learning*

Complete the implementation of the linear semi-gradient Sarsa-learning agent from [SB18, Section 10.1]).

Info: I have added a check to see if the weights diverge which happened to me in some of the problems. It might also be of use to add a regularization term, however I have not tested this idea.



Often, the averaging can hide important details about the individual runs, such as whether the average performance is highly driven by outliers. I have included options to turn off averaging and get the individual runs as shown below:



4 n-step Sarsa (`nstep_sarsa_agent.py`) ✸✸

This method is included for completeness, but I feel it is reasonable to understand n-step methods theoretically since the implementation is annoying. The final method will be Sarsa for on-policy control as in [SB18, Section 7.2].

Be warned Sarsa is a bit more tricky to implement as it requires us to know the future actions (here, A_{t+k}) "*in advance*" relatively to Q-learning in order to compute the return, however it is in my opinion very worthwhile to implement this method in order to properly understand eligibility traces which will be the subject next week.

In our implementation, we store the previous $n+1$ values of S, A, R in lists as a buffer so we can compute the return. Then, once that buffer has enough elements (the check is already in the code as given), we can perform the update of the Q-value for the *past* time step $\tau = t - n + 1$ (as in the pseudo code). This requires us to compute the return, starting in τ and counting n steps into the future, but is identical to the pseudo code.

Note that when the environment terminates (i.e. `done=True`) we still have n missing updates we must perform on the buffer elements, however these updates should be similar to that for a given τ and thus the code is re-used.

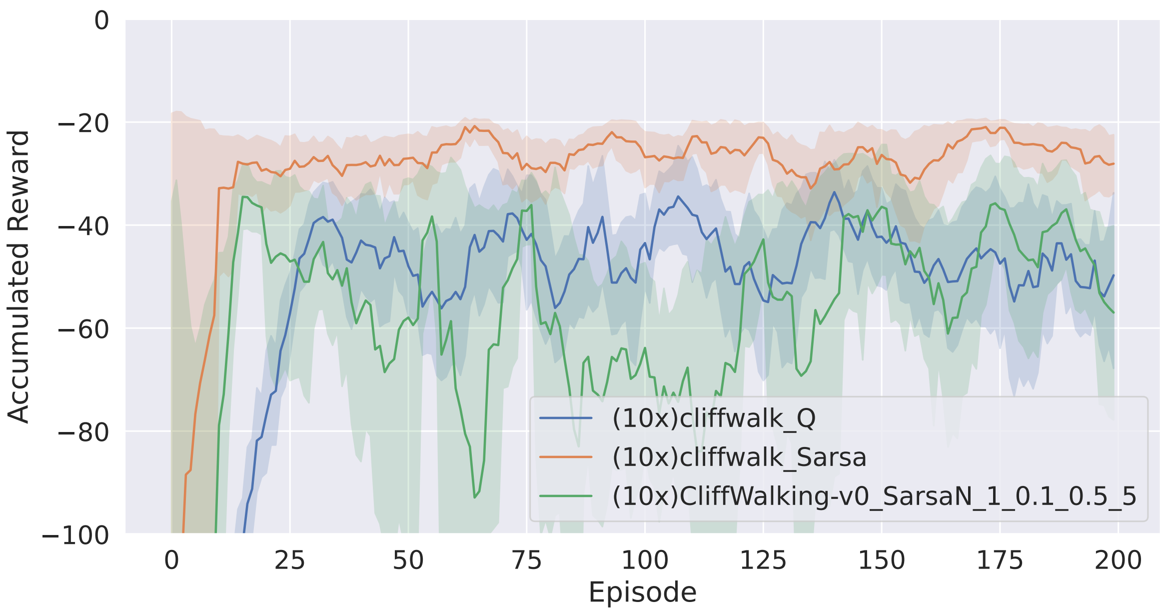
This is complicated, and you are likely to make a bug along the way. However noting that $n = 1$ corresponds to regular Sarsa, we can check that we compute the same updates in this case. I have inserted such a check and I strongly recommend debugging the code for the $n = 1$ case to ensure it works before moving on to $n > 1$.

Note some methods are abstracted out to make the semi-gradient easier to implement, but if you want to follow this path is up to you.

Problem 5 *n-step Sarsa-control*

Complete the implementation of the n-step Sarsa agent and reproduce figures comparable to [SB18, Example 6.6] using $n = 5$; Note I am not sure about the settings of the parameters and you might be able to find better parameters or a nicer example.

Info: With the caveat I am not sure about the settings for the simulations I get the following results. I think what they indicate is that n-step Sarsa perhaps learn a bit faster but with more variance relative to Sarsa (which is quite natural considering what we know about MC learning)



References

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