

Including new third factors in learning rules for reinforcement learning tasks

CS-479 Mini Project 2

Spring Semester 2025

Grading and submission

Grading The mini-project is **mandatory** and counts towards 30% of the final grade. You will be graded on the results you obtained and your answers to the questions, and a fraction of the grading is also dedicated to the quality and clarity of the report.

Teams You will have to work in teams of two students. To find a project partner if you don't have one yet, you can use the EdStem forum (<https://edstem.org/eu/courses/2085/discussion>).

Final report Your submission should be the interactive notebook .ipynb. Submit the notebook with all the plots and answers, but running it should reproduce all your results and plots (if we run it after placing it in the same folder with the environment code that we provided). The code itself will not be graded, only your answers to the questions. Clearly label your answers to the questions and the plots that form part of the answer. Write **concisely** in complete English sentences and present figures carefully (axis labels, legends etc.). Write **concisely**: go straight to the point in answering the question.

Submission You will have to submit your .ipynb notebook on the Moodle page. The file name should have the structure `MPX.NameMember1.NameMember2.ipynb`, where X is the miniproject number (either 1 or 2). If you work in teams both of you should submit the same file. On Moodle, make sure you press the submit button before the deadline.

Deadline The deadlines to submit notebook are:

- May 26th 2025, at 11.55 pm (fraud-detection slots: May 27th and 28th, 2025);
- June 2nd 2025, at 11.55 pm (fraud-detection slots: June 3rd and 4th, 2025).

The early deadline allows you to do the fraud detection before the exam preparation period, which you might find convenient.

Regulations and fraud detection After the submission, we will perform a fraud detection interview which consists of a few questions about your report and your code. You will need to attend **in-person** (both members of the team). Choose your deadline accordingly.

The mini-project is graded and the same rules as for exams apply (“Ordonnance sur le contrôle des études à l’EPFL”). In short: EPFL takes any form of cheating and plagiarism very serious. The mini-project has to be your own work. You are not allowed to share code or answers across teams. However you can discuss ideas between teams and ask questions on the Ed forum. If you split the work between the two team members, both team members have to understand all parts of the mini-project. Note: The goal of this meeting is fraud detection; it is neither an exam nor a presentation.

1 Introduction

In this project, we aim to show how third factors enhance model-based and model-free RL algorithms in an environment with traps. In this environment, we will have in each ‘non-trap’ state four actions that result in the following transitions:

- One action results in a ‘neutral’ transition, i.e. no state change.
- Two actions result in transition to a ‘trap’ state.
- One action results in a transition to the next state.

We can number each of the actions 1 to 4, and then we randomly assign the possible transitions to the actions. The schematic overview of the environment can be seen in the notebook file. The start state is 1 and the goal is then to reach the goal state G . The next states are ordered as $1 \rightarrow 2 \rightarrow \dots \rightarrow 7 \rightarrow G$. The trap states are 8, 9, 10, and in each trap state we have three actions that will result in a transition to either 8, 9 or 10, again randomly assigned. Then there will be one action per trap state that results in a transition to the start state, 1.

Note that due to the trap states, an agent that takes random moves will on average take a large number of actions before it completes one episode, as it has a large probability of taking an action that transitions to a trap state.

First we will be interested in eligibility traces, which you have seen in class. This is a classic addition to basic RL algorithms like SARSA which, as you will find, can greatly improve the speed of learning. This is achieved by better exploiting the exploration of all visited state-action pairs after the first reward is received.

The second concept that we will explore is novelty. Novelty can be thought of as how rarely a state has appeared in previous observations. It can help steer the exploration even before any reward is achieved (which can take a long time in this environment) towards unexplored states, which are potentially more promising than known states (that haven’t led to reward). Thus, novelty can improve the speed of learning from the very first episode.

To test our third factors in various types of algorithms, we will use SARSA and Q-learning as model-free methods, and Tabular Dyna- Q (shortly put: Q -learning with a replay buffer) and Value Iteration as model-based methods.

2 RL Algorithms

You will be able to see the template and detailed instructions in the Notebook. All the questions are also included in the notebook for your convenience.

2.1 SARSA and Q-Learning

Please implement SARSA and Q-learning algorithms as taught in the lecture, based on the template we provide in the notebook. For the next step, you should also be able to add eligibility traces for SARSA. You should add the option to initialize Q values with a chosen value.

1. Implement the SARSA and Q-Learning algorithms. Run simulations specified in the notebook. Provide plots of the number of steps per episode (i.e. episode length) during training.
2. How many steps does the first episode take (on average)?
3. How many episodes until "convergence"? Here by "convergence" we mean that the episode length becomes stable. You are welcome to define your own quantitative criterion for 'convergence'.
4. How many episodes are needed to update the Q-values of the start state?
5. For the optimistic initialisation, try to set $\epsilon = 0$ and see how that affects the episode lengths. Give an intuition why optimistic initialisation works, and why performance is so poor without it.

2.2 Tabular Dyna-Q

Tabular Dyna-Q

```
Initialize  $Q(s, a)$  and  $Model(s, a)$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ 
Do forever:
  (a)  $S \leftarrow$  current (nonterminal) state
  (b)  $A \leftarrow \epsilon\text{-greedy}(S, Q)$ 
  (c) Execute action  $A$ ; observe resultant reward,  $R$ , and state,  $S'$ 
  (d)  $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
  (e)  $Model(S, A) \leftarrow R, S'$  (assuming deterministic environment)
  (f) Repeat  $n$  times:
     $S \leftarrow$  random previously observed state
     $A \leftarrow$  random action previously taken in  $S$ 
     $R, S' \leftarrow Model(S, A)$ 
     $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
```

1. Implement this algorithm. Run simulations specified in the notebook. Provide plots of the number of steps per episode during training.
2. What does setting $\epsilon = 0$ do for Dyna-Q, with and without optimistic initialisation? What is the effect of each?
3. Run the algorithm with different numbers of replays (a replay is one sampled state-action pair previously experienced by the agent). How does the number of replays affect the final performance of the algorithm?
4. Does the number of replays affect the speed of the convergence? And why? If yes, try to quantify how, for example by trying to fit the decay rate of the episode lengths and plotting the rates for different numbers of replays.

2.3 Value Iteration

If we know all the transitions and rewards in an environment, we can solve the Bellman equation to get the value of each state. Using these values, we can construct an optimal policy. Recall that the Bellman equation has the following form if we are playing with a policy $\pi(s, a)$:

$$Q(s, a) = \sum_{s'} P_{s \rightarrow s'}^a \left[R_{s \rightarrow s}^a + \gamma \sum_{a'} \pi(s', a') Q(s', a') \right]$$

If we assume the optimal policy $\pi^*(s) = \operatorname{argmax}_{a'} Q(s, a')$, then the Bellman equation takes the following form:

$$Q^*(s, a) = \sum_{s'} P_{s \rightarrow s'}^a \left[R_{s \rightarrow s}^a + \gamma \max_{a'} Q^*(s', a') \right]$$

Recall, that we can calculate Q -values from V -values and vice versa:

$$V(s) = \sum_a \pi(s, a) Q(s, a)$$

$$Q(s, a) = \sum_{s'} P_{s \rightarrow s'}^a [R_{s \rightarrow s'}^a + V(s')]$$

So the above equations for the Bellman equation for a policy $\pi(s, a)$ and optimal policy $\pi^*(s)$ can also be written in terms of V -values:

$$V(s) = \sum_a \pi(s, a) \sum_{s'} P_{s \rightarrow s'}^a [R_{s \rightarrow s'}^a + \gamma V(s')]$$

$$V^*(s) = \max_a Q^*(s, a) = \max_a \sum_{s'} P_{s \rightarrow s'}^a [R_{s \rightarrow s'}^a + \gamma V^*(s')]$$

If we know the transition probabilities and rewards, we can solve the Bellman equation and find the optimal policy. An algorithm to do so, is called value iteration and is listed below:

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

- $\Delta \leftarrow 0$
- Loop for each $s \in \mathcal{S}$:
- $v \leftarrow V(s)$
- $V(s) \leftarrow \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$
- $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that

$$\pi(s) = \operatorname{argmax}_a \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$$

First we assume that all transitions occur with equal probability, in this case: $\frac{1}{11}$ as there are 11 possible states. With this assumption, and assuming that $R(G) = R(11) = 1$, we can solve the Bellmann equation exactly for the optimal policy. **Calculate this and fill it in.** With this initialisation of the values, we start playing the episodes using the approximate optimal policy as calculated from the current value estimates. Then at each step, we can update the transition probabilities $p(s'|s, a) = \frac{1}{11}$ to 1 for the state we end up in and 0 for the rest, as the environment is deterministic. After this update of transition probabilities, we run one iteration of value iteration.

1. Explain your calculation of the initial values.
2. What exactly are the offline and online parts in this setup?
3. Is this on-policy or off-policy?
4. Implement and run the algorithm. Provide plots of the number of steps per episode during training.

3 The new third factors

3.1 Eligibility Trace

1. Run the simulations specified in the notebook. Provide plots of the number of steps per episode during training.
2. How many steps does the first episode take (on average)?
3. How many episodes until "convergence"? Here by "convergence" we mean that the episode length becomes stable. You are welcome to define your own quantitative criterion for 'convergence'.
4. How many episodes are needed to update the Q-values of the start state (in theory and in practice)?

3.2 Novelty

Novelty can be defined as a state being new, original or unusual. In the case of RL, we consider a state novel when we have not seen it much. In the environment we consider here, the states that we have not seen much are usually the states leading to the goal. The reason for that is that during initial play we will frequently visit the trap states and not the states leading to the goal.

To this end, we will define a novelty function that we can later use for learning. We will use the definition listed in the paper by Xu et al., 2016

We can define visit counts of a state s at a step t as $C_s^{(t)}$. Using this, we can define a sort of empirical frequency of observing state s at step t as (if you are interested in more information, see the paper):

$$p_N^t(s) = \frac{C_s^{(t)} + 1}{(\sum_{s'} C_{s'}^{(t)}) + 11} \quad (1)$$

With this definition, we can then define novelty as:

$$N^{(t)}(s) = -\log p_N^{(t)}(s) \quad (2)$$

With this novelty function defined, there are a few approaches we can take to incorporate it. Either we can add it as an extra reward signal, or we have a more complex setup with 2 policies. In the former we can simply adapt the rewards and use our existing algorithms. In the latter we would use a novelty-seeking policy in the first episode and a reward-seeking policy in subsequent episodes.

For the first part, let's adapt the environment to add the novelty to the reward in such a way that we do not have to adapt our algorithms. You can see the override of the class in the notebook.

1. Run the SARSA, Dyna- Q , and Value Iteration algorithms in the environment with Novelty reward. Simulation requirements are specified in the notebook. Provide plots of the number of steps per episode during training.
2. For all three algorithms, how many steps does the first episode take (on average)?
3. For all three algorithms, how many episodes until convergence?
4. For all three algorithms, how does adding novelty impact the performance in terms of both the length of the first episode and the number of episodes to convergence? Compare the effect of novelty on the three algorithms.
5. For SARSA and Dyna- Q , how do novelty and optimistic initialisation interact and affect training?

4 Summary Questions

1. What is the effect of doing optimistic initialisation?
2. How do novelty and optimistic initialisation relate? Do they achieve similar goals?
3. How do you think would the results change if we had run all the algorithms on a stochastic environment? Rethink your answers to the previous two questions in this case.

We hope you enjoy the project! Good luck!