Reinforcement Learning and Decision Making, Coursework

MLMI candidate H801D

Preamble

We apply various model-based (policy iteration, value iteration) and model-free methods (Sarsa, Q-learning) to a grid world exploration task. General reference is [SB18].

Initial code and task description were provided in [HLGAP21]. All experiments were run on MATLAB. Complete code can be found in Appendix B. 1999 words.

1 Model-based methods

1.1 Value Iteration

We use *value iteration* (see [SB18, Section 4.4]) to iteratively approximate the optimal value function V_* using the following update rule:

$$V_{k+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left(R(s,a) + \gamma V_k(s') \right), \tag{1}$$

where A, S are the (discrete) action and state spaces, $P(s'|s,a)$ is the probability of transitioning to state $s' \in S$ from state $s \in S$ with action $a \in A$, the reward for action a in state s is $R(s,a)$ and γ is the *discount rate*[1]. We give a simple proof of convergence (which was not discussed in the Lectures) for the algorithm in Appendix A. A very similar argument works for the *policy evaluation* algorithm used in Subsection 1.2.

In Code 1 below, we use the "in-place" version of the algorithm (which has been reported to converge faster, see [SB18, Section 4.1]), where we update the value of $V(s)$ as we loop over the states $s \in S$, rather than computing $V_{k+1}(s)$ for all $s \in S$ using formula (1) and the previous values $V_k(s')$ before updating the value function.

```
v = zeros(model.stateCount, 1);
for i = 1:maxit
    % Used for convergence criterion
    v_comparison = v;
    % Perform the Bellman update
    % for each state
    for s = 1:model.stateCount
        Qs=zeros(4,1);
        for a=1:4
            Qs(a)= model.P( s, :, a )*...
            (model.R(s,a)+ model.gamma*v);
        end
        [best_Qs_a,~]=max(Qs);
        v(s)=best_Qs_a;
    end
    % Exit early if v is not changing much
    if max(abs(v-v_comparison))<tolerance
        break;
    end
end
pi=policyFromV(v,model);
v=policyEvaluation(pi,model,10000,tolerance,v);
```

Code 1: Value iteration algorithm.

The 'policyFromV' function computes the greedy policy π_{greedy} s.t.

$$\pi_{greedy}(s) := \operatorname*{argmax}_{a \in A} \sum_{s' \in S} P(s'|s,a) \left(R(s,a) + \gamma V(s') \right)$$

for all $s \in S$ associated to a given value function V. On the last line of Code 1, we compute $V_{\pi_{greedy}}$ directly from π_{greedy} with policy evaluation (see Subsection 1.2) instead of using the function V

[1] Formulations can vary depending on the problem; for example, the reward is not always deterministic.

[2] This is not a problem here, as convergence is very fast, nor would it be computationally efficient in a "real world" context.

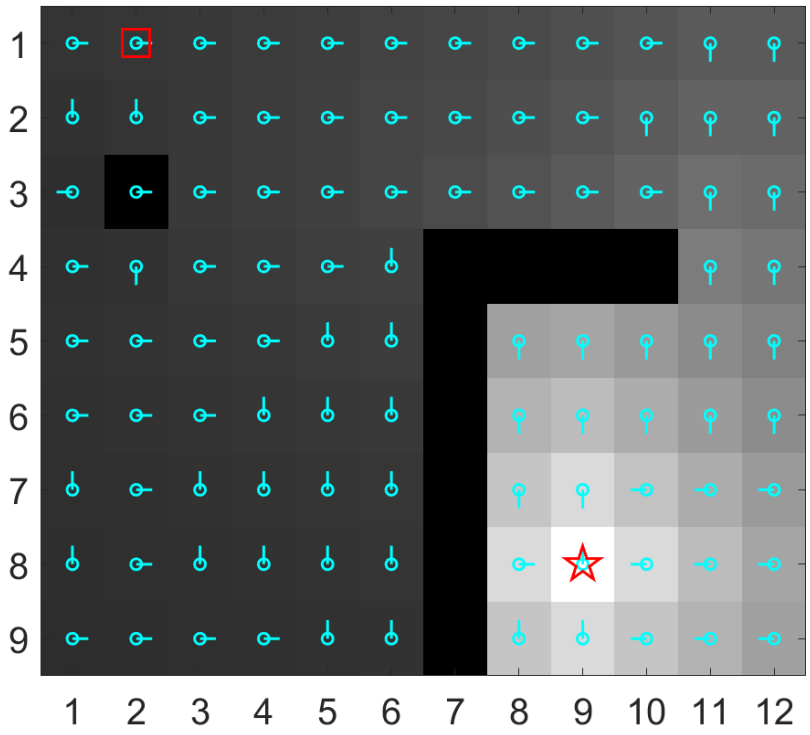

Figure 1: Results of the value iteration algorithm (or equivalently of the policy iteration algorithm) on the grid world environment (with a tolerance of 10^{-7}). The starting and goal states are indicated by the red square and star respectively, the final policy by the blue arrows, and the final value function by the shading (lighter shading indicates higher values).

previously computed, as V might be very different from $V_{\pi_{greedy}}$ if 'maxit' was too small[2].

Note that $||V_{k+2} - V_{k+1}||_\infty \leq \gamma ||V_{k+1} - V_k||_\infty$ (where $|| - ||_\infty : \mathbb{R}^S \to \mathbb{R}^S$ is the ∞-norm), for arguments similar to those in Appendix A, which justifies our stopping criterion: if one iteration brings little change, so will the following ones.

The algorithm stops for a tolerance threshold of 10^{-7} after only 68 iterations on the 'grid-world' model; $V_{\pi_{greedy}}$ and π_{greedy} are illustrated in Figure 1 obtained with the provided 'PlotVP' function). The end state is not shown, though it is necessary for our implementation (as otherwise the agent would keep indefinitely collecting reward after having reached the goal state).

We see that the optimal policy takes the shortest paths towards the objective, except around the pitfall location $(2, 3)$, where it tries to move away from it. This is due to the probability of "bad transition" of 0.3, which can result in the agent accidentally falling in the pitfall when moving parallelly to it (*e.g.* from $(2, 2)$ to $(3, 2)$), but also allows it to move away from $(1, 3)$ despite aiming for the wall. This phenomenon disappears when setting the probability of good transition to 1, as illustrated in Figure 2.

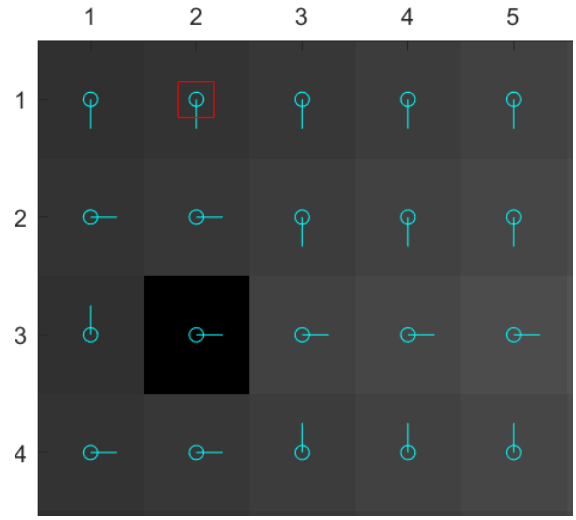

Figure 2: Results of the value iteration algorithm on the grid world environment with modified probability of good transition of 1, zoomed in around the "pitfall" state.

1.2 Policy Iteration

We apply *policy iteration*, another Dynamic Programming (DP) method. It alternates between greedily updating a deterministic policy π w.r.t a value function V by setting as before

$$\pi(s) := \underset{a \in A}{\operatorname{argmax}} \sum_{s' \in S} P(s'|s,a) \left(R(s,a) + \gamma V(s') \right)$$

for all $s \in S$, and making V consistent with the current policy with policy evaluation (see [SB18, Section 4.1]). We use Code 2 below.

```
% Initialize the policy
pi = ones(model.stateCount, 1);
for i = 1:maxit
 % Policy evaluation
 v = policyEvaluation(pi,model,...
 maxit,10^(-10),v);
 % The greedy policy associated to v
 pi_ = policyFromV(v,model);
 % Exit early if pi is not changing anymore
 if sum(abs(pi_ - pi)) == 0
     break;
 end
 pi=pi_;
end
```

Code 2: Policy iteration algorithm.

Observe that we use the previously estimated value function as the starting point of the policy evaluation algorithm, which speeds up the algorithm (we can expect the updated value function to be close to the old one), as suggested in [SB18, Section 4.3]. Code 1.2 below implements "in-place" policy evaluation for both deterministic and stochastic policies, which is guaranteed to converge for similar reasons as for value iteration.

```
function v = policyEvaluation(pi,model,...
maxit,tolerance,varargin)
% varargin is meant to be a value function
% to be used as a starting point
if nargin == 5
    v= varargin{1};
else
    % Otherwise, initialise v trivially
    v = zeros(model.stateCount, 1);
end
for i = 1:maxit
    % Used to test convergence
    v_comparison=v;
    % Tests whether pi is deterministic,
    % updates it accordingly
    % using the Bellman update
    if size(pi,2)==1
        for s = 1:model.stateCount
          v(s)= model.P( s, :, pi(s) )*...
          (model.R(s,pi(s))+ model.gamma*v);
        end
    else
        for s = 1:model.stateCount
            v_(s)=0;
            for a = 1:4
              v_(s)=v_(s) + ...
              pi(s,a)*model.P( s, :, a)*...
              (model.R(s,a)+ model.gamma*v);
            end
            v(s)=v_(s);
        end
    end
    % exit early if v is not changing much
    if max(abs(v_comparison-v))<tolerance
        break;
    end
end
```

Code 3: Policy evaluation algorithm.

The convergence of the policy iteration algorithm is then a consequence of the policy improvement theorem (see [SB18, Section 4.2]). On the 'gridworld' model, it happens after only 7 iterations[3] for a tolerance of 10^{-7}. The resulting optimal policy is (as expected) the same as for the value iteration algorithm, and so is the associated approximate value function (up to the tolerance of 10^{-7}); see Figure 1 again.

The model's small state and action spaces make such simple DP methods well-suited to the task. For bigger S and A (*e.g.* in higher dimensional problems), asynchronous DP methods (which do not sweep through every state in each iteration, see [SB18, 4.5]) might work better.

[3] But each global iteration involves calling the policy evaluation algorithm, which runs itself for $50 - 150$ iterations.

2 Model-free methods

2.1 Sarsa

Code 4 showcases the key lines of our implementation of *Sarsa*[4], an on-policy, model-free temporal-difference method (see [SB18, Section 6.4]).

```
for i = 1:maxeps
 % The cumulated reward for that episode
 total_R=0;
 % Decrease epsilon
 epsilon = initialEpsilon/...
 (i^epsilonDeclineCoeff);
 % Decrease alpha
 alpha = initialAlpha/(i^alphaDeclineCoeff);
 % Start episode
 s = model.startState;
 a = EGreedyActionFromQ(Q(s,:),epsilon);
 for j = 1:maxit
   % Observe new state s_ and reward r
   s_ = categoricalSample(model.P(s,:,a));
   r = model.R(s,a);
   total_R=total_R+model.gamma^(j-1)*r;
   % Pick new action a_
   a_ = EGreedyActionFromQ(Q(s_,:),epsilon);
   % Update Q(s,a)
   Q(s,a) = Q(s,a) + ...
   alpha*(r+model.gamma*Q(s_,a_)-Q(s,a) );
   % Actualize current state and new action
   s = s_;  a = a_;
   % Stop if end state reached
   if s == model.stateCount
     break;
   end
 end
    R_history(i)=total_R;
end
```

Code 4: Sarsa algorithm, with options to let ϵ and α decrease over time.

The function 'EGreedyActionFromQ' takes as arguments $\epsilon > 0$ and $Q(s,-) : A \to \mathbb{R}$ for some $s \in S$, and samples an action a from A with probability $1 - \epsilon + \frac{\epsilon}{|A|}$ if $a = \text{argmax}_{a' \in A} Q(s,a')$ and $\frac{\epsilon}{|A|}$ otherwise - an ϵ-greedy choice.

Sarsa learns the optimal ϵ-soft policy using the update rule (see [SB18, Section 6.4])

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R(s,a) + \gamma Q(s',a') - Q(s,a) \right),$$

where all actions (including a') are selected using the ϵ-greedy policy associated to the current action-value function Q. Our code returns both the ϵ-greedy policy π_ϵ learned by the algorithm and the deterministic greedy policy π associated to Q, as well as their value functions V_ϵ and V.

Running it for 10′000 episodes with $\alpha = 0.5$, $\epsilon = 0.1$ and at most 20 steps per episode on the 'smallworld' model yields the value function V_ϵ illustrated in Figure 3.

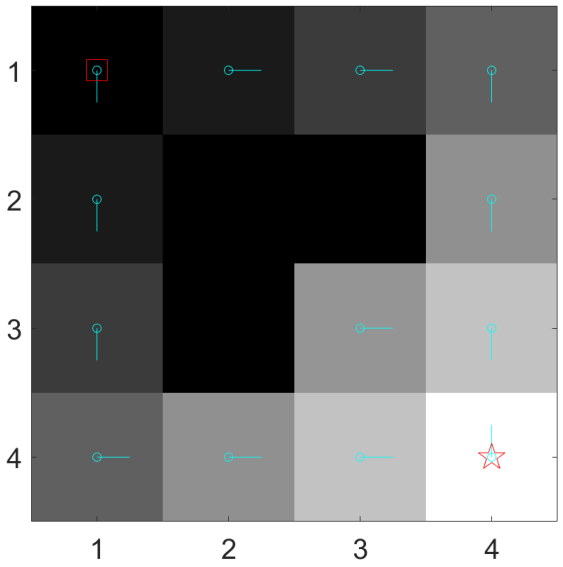

Figure 3: Value function of the ϵ-greedy policy π_ϵ obtained after running Sarsa for 10′000 episodes (with $\alpha = 0.5$ and $\epsilon = 0.1$). The directions displayed correspond to the associated greedy deterministic policy.

Note that Sarsa optimizes an ϵ-greedy policy; it need not be equal to the ϵ-soft version of the optimal deterministic policy (as we will see in Subsection 2.3). Having $\epsilon_k \xrightarrow{k \to \infty} 0$ (where k is the episode) lets Sarsa converge towards the optimal policy as long as each state-action pair is visited infinitely many times (see [SB18]). Observe that this condition is not satisfied[5] for $\epsilon_k := \epsilon_0 / k$ (which our code allows us to implement), as the asymptotic probability of visiting certain states is in $\mathcal{O}(\epsilon^n)$ with $n > 1$: the expected number of visits would be finite.

[4] Specifically its ϵ-greedy version. See Appendix B for the complete code.

[5] Unlike what is suggested on [SB18, page 129].

In practice, even for constant $\epsilon > 0$, some states in more complicated models are visited infrequently enough (either due to being pitfalls, or too far from the "ϵ-optimal" path) that the associated Q suggest very poor actions, even after many episodes, as can be seen in Figure 4.

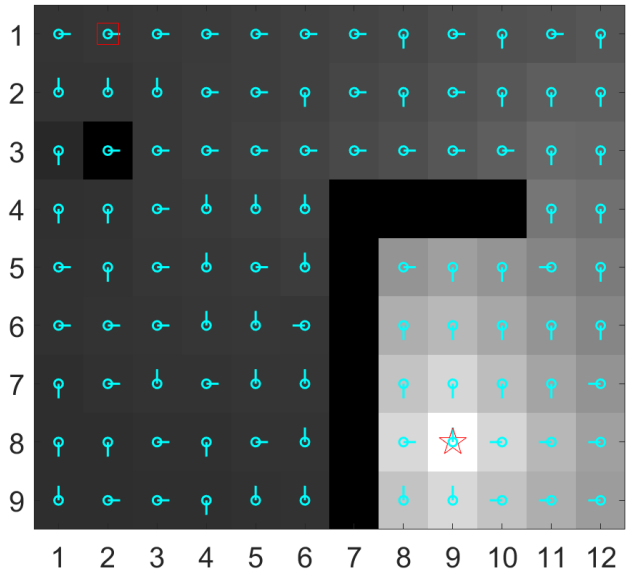

Figure 4: Deterministic greedy policy associated to the Q action-value function computed with Sarsa after 10′000 episodes for $\alpha = 0.5$, $\epsilon = 0.1$ and at most 40 steps per episode. Observe that the policy is optimal close to the fastest paths but very suboptimal in rarely visited regions (lower left corner).

This might not be a problem: depending on the context, the agent does not necessarily need to know how to act optimally in rarely encountered states far from any good path.

2.2 Q-learning

Unlike Sarsa, *Q-learning* is an *off-policy* temporal-difference method, with the following update rule (see [WD92] or [SB18, Section 6.5]):

$$Q(s,a) \leftarrow Q(s,a)+\alpha(R(s,a)+\gamma \max_{a' \in A} Q(s',a')-Q(s,a)),$$

where successive actions a are selected using the ϵ-greedy policy associated to Q. Its target is the deterministic optimal policy, while its behaviour policy is an ϵ-soft policy. Code 5 shows the key lines of our implementation.

```
for i = 1:maxeps
  % Epsilon decreases
  epsilon = initialEpsilon/...
  (i^epsilonDeclineCoeff);
  % Alpha decreases
  alpha = InitialAlpha/...
  (i^alphaDeclineCoeff);
  % Start the episode
  s = model.startState;
  % The accumulated reward for the episode
  total_R=0;
  for j = 1:maxit
    % We choose and take action a,
    % observe new state s_ and reward r
    a = EGreedyActionFromQ( ...
    Q(s,:),epsilon);
    s_ = categoricalSample(...
    model.P(s,:,a));
    r = model.R(s,a);
    total_R = total_R+model.gamma^(j-1)*r;
    % What would have been the greedy
    % action to take in state s_
    % (a 0-greedy action)
    a_ = EGreedyActionFromQ(Q(s_,:),0);
    % Update Q(s,a)
    Q(s,a) = Q(s,a) + alpha*...
    (r+model.gamma*Q(s_,a_)-Q(s,a) );
    % Actualize current state
    s = s_;
    % Stop if end state reached
    if s == model.stateCount
      break;
    end
  end
    % Records the episode's total reward
    R_history(i)=total_R;
end
```

Code 5: Q-learning algorithm, with options to let ϵ and α decrease over time.

Running it on the 'smallworld' model, we get visually indistinguishable results from those obtained with Sarsa (see Figure 3), though differences appear on more complicated tasks - see Subsection 2.3, where we also compare the effects of different values of α (both for Sarsa and Q-learning).

2.3 Comparison between Sarsa and Q-learning

We test Sarsa and Q-learning on the more complicated 'cliffworld' model, as well as the on-policy version of the *Expected Sarsa* algorithm, which should outperform both in online performance according to [SB18, Section 6.6]. It is similar to the Q-learning algorithm, except that the update is

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[R(s,a) + ... \\ \gamma \mathbb{E}_{a' \sim \pi_\epsilon(-|s')}[Q(s',a')] - Q(s,a)\Big],$$

where π_ϵ is the ϵ-greedy policy associated to Q and successive actions $a \in A$ are sampled from π_ϵ. It eliminates the variance associated to the choice of the action a' in the Sarsa update.

In Figure 5, we plot for Sarsa, Q-learning and Expected Sarsa the sum of discounted rewards $R = \sum_{t=1}^{T} \gamma^{t-1} r_t$ for each episode, averaged over 500 runs[6]. We see that Sarsa and Expected Sarsa behave somewhat similarly (Expected Sarsa being very slightly better), and outperform on average Q-learning.

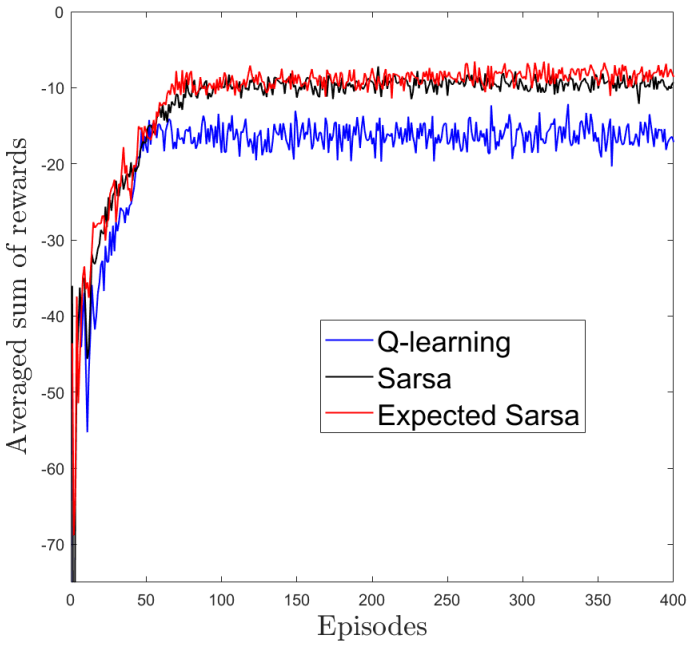

Figure 5: Sum of rewards per episode averaged over 500 runs on the 'cliffworld' model for Sarsa, Q-learning and Expected Sarsa, with $\alpha = 0.5$, $\epsilon = 0.1$, discount rate $\gamma = 0.9$ and at most 30 steps per episode.

[6]Each with a different random seed.

This is because Q-learning learns the optimal deterministic policy and uses an ϵ-soft version of it as its behaviour policy. This means that it learns the shortest path close to "the cliff", and moves with some added randomness to guarantee exploration, which makes the agent likely to "fall" (see Figure 6).

Oppositely, Sarsa and Expected Sarsa learn the optimal ϵ-soft policy, which is different from an ϵ-soft version of the optimal deterministic policy; it involves learning a safer path, further away from the cliff (see Figure 6).

These differences are also illustrated in Figure 7, which shows the (non-averaged) sum of rewards per episode for a rather typical run of Sarsa and Q-learning. We see that Q-learning outperforms Sarsa on most episodes due to taking a shorter path, but that its average performance is penalized by more frequent catastrophic falls. Expected Sarsa behaves similarly in that regard to Sarsa, as it has the same behaviour and target policy, but learns it in a way that reduces the variance of the updates.

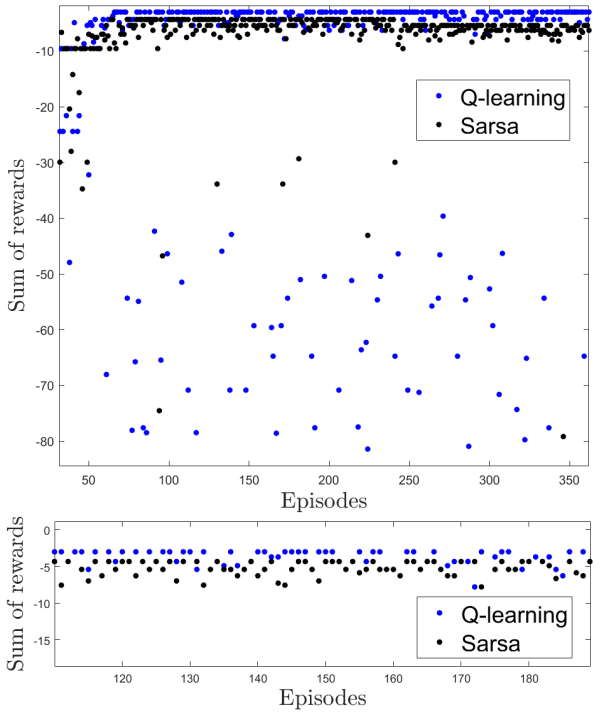

Figure 7: Sum of discounted rewards per episode (for a single run) on the 'cliffworld' model for Sarsa and Q-learning, with $\alpha = 0.5$, $\epsilon = 0.1$, discount rate $\gamma = 0.9$ and at most 30 steps per episode. Bottom figure is a detail of the top one.

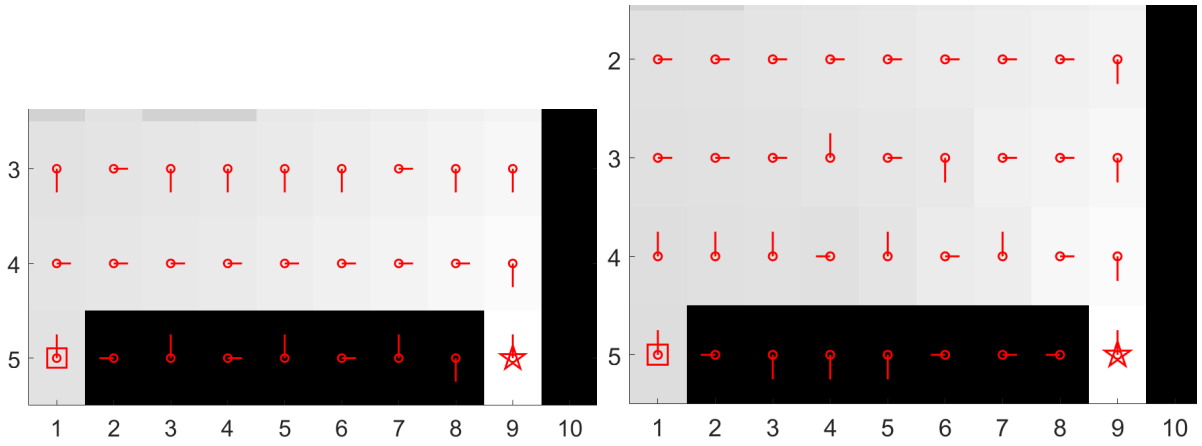

Figure 6: Details of the greedy deterministic policies on the 'cliffworld' model associated to the final action-value function Q after 400 episodes for Q-learning (left) and Sarsa (right) for $\alpha = 0.5$, $\epsilon = 0.1$, and at most 30 steps per episode. Observe that Sarsa encourages taking a longer, albeit safer path.

On the other hand, it means that the policy $\pi_{\text{Q-learning}}$ learnt by Q-learning is closer to the optimal deterministic policy π_* than the deterministic, greedy versions[7] π_{Sarsa} and $\pi_{\text{Ex. Sarsa}}$ of the policies learnt by Sarsa and Expected Sarsa: their average value functions after 400 episodes[8] are $V_{\pi_{\text{Q-learning}}}(s_0) = -3.03$, $V_{\pi_{\text{Sarsa}}}(s_0) = -5.84$ and $V_{\text{Ex. Sarsa}}(s_0) = -4.35$. This means that while Sarsa (and even more so Expected Sarsa) outperform Q-learning when operating online, their offline performance is worse. Which is best depends on context: Sarsa (or Expected Sarsa) might be better when training a fragile robot to walk, while Q-learning could be better when consequence-free training is possible (*e.g.* when teaching a program how to play a video game).

As mentioned in Subsection 2.1, Sarsa and Expected Sarsa could be made to converge towards π_* by gradually decreasing ϵ (which should improve offline performance), but it is hard to find a decreasing schedule for ϵ fast enough that convergence happens after a reasonable number of episodes, yet slow enough that the level of exploration remains sufficient for the action-value to keep updating (at least on states close to the optimal path) and eventually converge to the action-value function of π_*.

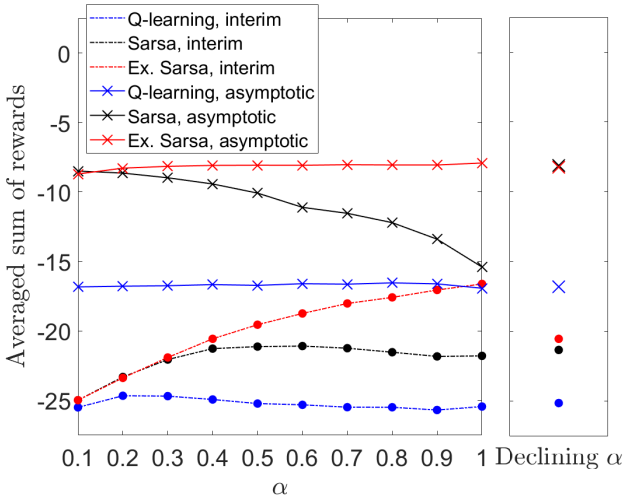

Figure 8: Average over 1000 runs (respectively, 10 runs) on the 'cliffworld' model of the average sum of discounted rewards over the first 100 episodes (respectively, the first 10′000 episodes) for various values of constant α and decreasing $\alpha = k^{-\frac{1}{3}}$, where k is the episode. The maximum number of steps per episode is 30, and we let $\epsilon = 0.1$ and $\gamma = 0.9$.

For a more in-depth comparison, we also test the impact of different values of the parameter $\alpha > 0$ on the short-term and asymptotic performances of all three algorithms, in the spirit of [HLGAP21, Figure 6.3]. Specifically, we compute for $\alpha \in \{0.1, 0.2, \ldots, 0.9, 1\}$ and for each algorithm the average over 1000 runs of the aver-

[7] Obtained directly using the action-value function Q.

[8] Averaged over 500 runs with $\alpha = 0.5$, $\epsilon = 0.1$ and at most 30 steps, with discount rate $\gamma = 0.9$.

age over the first 100 episodes of the sum of discounted rewards, which we call the *interim performance* (following [SB18]), as well as the average over 10 runs of the average over the first $10'000$ episodes of the sum of discounted rewards, which we call the *asymptotic performance*. The results are displayed in Figure 8.

The general trends match those from [vSvHWW09][9], though the details differ slightly because they compute undiscounted rewards (and possibly a different maximal number of steps per episode). Q-learning performs poorly (online, as discussed before) and is not very affected by the choice of α, while Sarsa's interim performance increases with α up to a certain point (as it allows for faster learning), but stagnates beyond that. As large α increase the variance of the Sarsa update, which might cause Q to diverge, Sarsa's asymptotic performance decreases as α increases. Expected Sarsa's lower variance update allows it to avoid this hurdle, and learn quickly with large α without adverse consequences[10].

We also compute the same quantities for decreasing $\alpha = k^{-\frac{1}{3}}$, where k is the episode; though it does not allow us to improve upon the best choices of constant α, a more subtle decreasing schedule for α might (see *e.g.* [DB12]).

References

[Ban22] S. Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3, 1922.

[DB12] W. Dabney and A. Barto. Adaptive step-size for online temporal difference learning. *AAAI*, 2012.

[HLGAP21] J. M. Hernández-Lobato, A. Garriga-Alonso, and R. Pinsler. Reinforcement learning and decision making lectures. University of Cambridge, MLMI MPhil, `https://www.vle.cam.ac.uk/course/view.php?id=121351&sectionid=3219391`, 2020-2021.

[SB18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction, 2nd edition.* MIT Press, 2018.

[vSvHWW09] H. van Seijen, H. van Hasselt, S. Whiteson, and M. Wiering. A theoretical and empirical analysis of Expected Sarsa. *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 2009.

[WD92] C. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8, 1992.

[9] From which Figure 6.3 from [SB18] is adapted.

[10] Its asymptotic performance is unaffected by α simply because convergence happens well before $10'000$ episodes for all 10 values of α. The same is true for Q-learning.

A Proof of convergence for the value iteration algorithm

Let $V \in \mathbb{R}^S$ be a value function[11], and let $L : \mathbb{R}^S \to \mathbb{R}^S$ be defined as

$$L(V)(s) := \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left(R(s,a) + \gamma V(s') \right).$$

Notice that in the value iteration algorithm, $V_{k+1} = L(V_k)$. Let $V_1, V_2 \in \mathbb{R}^S$, $s \in S$ and $a_i := \text{argmax}_{a \in A} \sum_{s' \in S} P(s'|s,a) \left(R(s,a) + \gamma V_i(s') \right)$. Then

$$\begin{aligned} L(V_1)(s) &= \sum_{s' \in S} P(s'|s,a_1) \left(R(s,a_1) + \gamma V_1(s') \right) \geq \sum_{s' \in S} P(s'|s,a_2) \left(R(s,a_2) + \gamma V_1(s') \right) \\ &\geq \sum_{s' \in S} P(s'|s,a_2) \left(R(s,a_2) + \gamma (V_2(s') - ||V_1 - V_2||_\infty) \right) = L(V_2)(s) - \gamma ||V_1 - V_2||_\infty, \end{aligned}$$

where $|| - ||_\infty : \mathbb{R}^S \to \mathbb{R}_+$ is the infinity norm, and symmetrically $L(V_2)(s) \geq L(V_1)(s) - \gamma ||V_1 - V_2||_\infty$, from which we can deduce that

$$||L(V_1) - L(V_2)||_\infty \leq \gamma ||V_1 - V_2||_\infty.$$

Hence L is a γ-Lipschitz continuous application. If $0 < \gamma < 1$, the Banach fixed-point theorem (see [Ban22]) guarantees that L admits a unique fixed point $V_* \in \mathbb{R}^S$, and that

$$L^k(V_0) \xrightarrow{k \to \infty} V_*$$

for any $V_0 \in \mathbb{R}^S$. As

$$L(V_*) = V_*$$

is precisely the Bellman optimality condition (see [SB18]), this concludes the proof. Similar arguments can be used when considering the policy evaluation algorithm or the "in-place" version of both algorithms.

[11] We identify the set of functions $\{V : S \to \mathbb{R}\}$ to $\mathbb{R}^S$, and assume as in the rest of the text that S and A are finite, and that the discount rate γ is strictly smaller than 1. Additional conditions must be added otherwise.

B Code

B.1 Main algorithms

B.1.1 Value Iteration

```
% Implements the "in-place" version of the value iteration algorithm
function [v, pi] = valueIteration(model, maxit,tolerance)

% initialize the value function and the policy
v = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
for i = 1:maxit
    % Used for convergence criterion
    v_comparison = v;
    % Perform the Bellman update for each state
    for s = 1:model.stateCount
        Qs=zeros(4,1);
        for a=1:4
            Qs(a)= model.P( s, :, a )* (model.R(s,a)+ model.gamma*v);
        end
        [best_Qs_a,~]=max(Qs);
        v(s)=best_Qs_a;
    end

    % Exit early if v is not changing much anymore
    if max(abs(v-v_comparison))<tolerance
        break;
    end
end
pi=policyFromV(v,model);
% If maxit was very small, the true value function associated to pi
% can be quite different from the function v (which was not a true value function)
% used to define pi. For maxit> ~20, this is superfluous
v=policyEvaluation(pi,model,10000,tolerance,v);
end
```

Code 6: "In-place" version of the value iteration algorithm.

B.1.2 Policy Evaluation

```
% Computes iteratively the value function associated to the provided policy
% using the "in-place" algorithm
% Works for both deterministic and non-deterministic policies
function v = policyEvaluation(pi,model,maxit,tolerance,varargin)
% The last optional argument is meant to be a value function
% to be used as a starting point by the algorithm
if nargin == 5
    v= varargin{1};
else
    % otherwise, initialise v trivially
    v = zeros(model.stateCount, 1);
end
for i = 1:maxit
    % used to test convergence
    v_comparison=v;
    % tests whether pi is deterministic or not, updates it accordingly
    % using the Bellman update
    if size(pi,2)==1
        for s = 1:model.stateCount
            v(s)= model.P( s, :, pi(s) )* (model.R(s,pi(s))+ model.gamma*v);
        end
    else
        for s = 1:model.stateCount
            v_(s)=0;
            for a = 1:4
                v_(s)=v_(s) + pi(s,a)*model.P( s, :, a)* (model.R(s,a)+ model.gamma*v);
            end
            v(s)=v_(s);
        end
    end
    % exit early if v is not changing much anymore
    if max(abs(v_comparison-v))<tolerance
        break;
    end
end
end
```

Code 7: Policy evaluation algorithm - works for both deterministic and stochastic policies. An initial value for the value function 'v' can be passed as an optional argument.

B.1.3 Policy Iteration

```
function [v, pi] = policyIteration(model, maxit)
% Initialize the value function
v = zeros(model.stateCount, 1);
% Initialize the policy and the new value function
pi = ones(model.stateCount, 1);
for i = 1:maxit
    % Perform policy evaluation to find the value function associated to pi
    % As an optional argument, we use the previous value function as the
    % starting point of the policyEvaluation algorithm to speed up
    % convergence
    v = policyEvaluation(pi,model,maxit,10^(-10),v);
    % The greedy policy associated to v
    pi_ = policyFromV(v,model);
    % Exit early if pi is not changing anymore
    if sum(abs(pi_ - pi)) ==0
        break;
    end
    pi=pi_;
end
end
```

Code 8: "In-place" version of the policy iteration algorithm.

B.1.4 Sarsa

```
% pi is the greedy deterministic policy associated to the final action-value
% function Q obtained and piEpsilon is the epsilon-greedy function associated to it
%  (where epsilon is the value of epsilon used for the last episode, see below)

% piEpsilon is a (model.statecount,4) vector,
% where piEpsilon(s,a) = probability of doing a in state s

% v is the value function associated to pi, and vEpsilon the value function
% associated to piEpsilon

% Epsilon and alpha decrease for each episode for epsilonDeclineCoeff>0
% (respectively alphaDeclineCoeff>0)
function [v, pi,vEpsilon,piEpsilon,R_history] = sarsa(model, maxit, maxeps,...
initialAlpha,alphaDeclineCoeff,initialEpsilon,epsilonDeclineCoeff)
% Initialize the value function
Q = zeros(model.stateCount, 4);
% Tracks the number of times each state-action pair has been tested (used
% for a modified version of the sarsa algorithm)
Q_number_visits = zeros(model.stateCount, 4);
```

```
% The cumulative reward for each episode
R_history = zeros(maxeps,1);
for i = 1:maxeps
    % The cumulated reward for that episode
    total_R=0;
    % Epsilon decreases for epsilonDeclineCoeff > 0
    epsilon = initialEpsilon/(i^epsilonDeclineCoeff);
    % Alpha decreases for alphaDeclineCoeff > 0
    alpha = initialAlpha/(i^alphaDeclineCoeff);
    % Every time we reset the episode, start at the given startState
    s = model.startState;
    a = EGreedyActionFromQ(Q(s,:),epsilon);
    for j = 1:maxit
        % We take action a, observe new state s_ and reward r
        s_ = categoricalSample(model.P(s,:,a));
        r = model.R(s,a);
        total_R=total_R+model.gamma^(j-1)*r;
        % Pick new action a_
        a_ = EGreedyActionFromQ(Q(s_,:),epsilon);
        % Update Q(s,a)
        Q(s,a) = Q(s,a) + alpha*(r+model.gamma*Q(s_,a_)-Q(s,a) );
        Q_number_visits(s,a)=Q_number_visits(s,a)+1;
        % Actualize current state and new action
        s = s_;
        a = a_;
        % Stop if end state reached
        if s == model.stateCount
            break;
        end
    end
    R_history(i)=total_R;
end
pi= greedyPolicyFromQ(Q);
piEpsilon = EGreedyPolicyFromQ(Q,epsilon);
% We compute v and vEpsilon independently - we could use Q, but if the
% number of episodes is very small, Q might be an extremely poor approximation
% of the true state-action value functions of pi and piEpsilon
v = policyEvaluation(pi,model,10000,10^(-9));
vEpsilon = policyEvaluation(piEpsilon,model,10000,10^(-9));
end
```

Code 9: Sarsa algorithm, with options to let ϵ and α decrease over time.

B.1.5 Q-learning

```
function [v, pi, R_history] = qLearning(model, maxit, maxeps,initialAlpha,...
alphaDeclineCoeff,initialEpsilon,epsilonDeclineCoeff)

% Initialize the value function
Q = zeros(model.stateCount, 4);
% The cumulative reward for each episode
R_history = zeros(maxeps,1);

for i = 1:maxeps
    % Epsilon decreases for epsilonDeclineCoeff > 0
    epsilon = initialEpsilon/(i^epsilonDeclineCoeff);
    % Alpha decreases for AlphaDeclineCoeff > 0
    alpha = initialAlpha/(i^alphaDeclineCoeff);
    % Every time we reset the episode, start at the given startState
    s = model.startState;
    % The accumulated reward for that episode
    total_R=0;

    for j = 1:maxit
        % We choose and take action a, observe new state s_ and reward r
        a = EGreedyActionFromQ(Q(s,:),epsilon);
        s_ = categoricalSample(model.P(s,:,a));
        r = model.R(s,a);
        total_R = total_R + model.gamma^(j-1)*r;
        % What would have been the greedy action to take in state s_
        % (a 0-greedy action)
        a_ = EGreedyActionFromQ(Q(s_,:),0);
        % Update Q(s,a)
        Q(s,a) = Q(s,a) + alpha*(r+model.gamma*Q(s_,a_)-Q(s,a) );
        % Actualize current state
        s = s_;
        % Stop if end state reached
        if s == model.stateCount
            break;
        end
    end
    % Records the episode's cumulative reward
    R_history(i)=total_R;
end
pi= greedyPolicyFromQ(Q);
% We compute v - we could use Q, but if the number of episodes was small,
% Q might be an extremely poor approximation
% of the true state-action value functions of pi
v = policyEvaluation(pi,model,10000,10^(-9));
end
```

Code 10: Q-learning algorithm, with options to let ϵ and α decrease over time.

B.1.6 Expected Sarsa

```
% pi is the greedy deterministic policy associated to the final
% action-value function Q obtained and piEpsilon is the epsilon-greedy
% function associated to it (where epsilon is the value of epsilon used for
% the last episode, see beloz)

% piEpsilon is a model.statecount x 4 vector, where piEpsilon(s,a) =
% probability of doing a in state s

% v is the value function associated to pi, and vEpsilon the value function
% associated to piEpsilon

% Epsilon and alpha decrease for each episode for epsilonDeclineCoeff>0
% (respectively alphaDeclineCoeff>0)

function [v, pi,vEpsilon,piEpsilon,R_history] = expectedSarsa(model, maxit,...
maxeps, initialAlpha, alphaDeclineCoeff, initialEpsilon, epsilonDeclineCoeff)

% Initialize the value function
Q = zeros(model.stateCount, 4);
% The cumulative reward for each episode
R_history = zeros(maxeps,1);

for i = 1:maxeps
    % The cumulated reward for that episode
    total_R=0;
    % Epsilon decreases for epsilonDeclineCoeff > 0
    epsilon = initialEpsilon/(i^epsilonDeclineCoeff);
    % Alpha decreases for alphaDeclineCoeff > 0
    alpha = initialAlpha/(i^alphaDeclineCoeff);
    % Every time we reset the episode, start at the given startState
    s = model.startState;

    for j = 1:maxit
        % We choose and take action a, observe new state s_ and reward r
        a = EGreedyActionFromQ(Q(s,:),epsilon);
        s_ = categoricalSample(model.P(s,:,a));
        r = model.R(s,a);
        total_R = total_R + model.gamma^(j-1)*r;

        % Update Q(s,a)
        probas=epsilon/4*ones(4,1);
        [~,best_a]=max(Q(s_,:));
        probas(best_a)=probas(best_a)+1-epsilon;
        expectancy = Q(s_,:)*probas;
        Q(s,a) = Q(s,a) + alpha*(r+model.gamma*expectancy-Q(s,a) );
        % Actualize current state and new action
        s = s_;
```

```
        % Stop if end state reached
        if s == model.stateCount
            break;
        end
    end
    R_history(i)=total_R;

end
pi= greedyPolicyFromQ(Q);
piEpsilon = EGreedyPolicyFromQ(Q,epsilon);
% We compute v and vEpsilon independently - we could use Q, but if the
% number of episodes was small, Q might be an extremely poor approximation
% of the true state-action value functions of pi and piEpsilon
v = policyEvaluation(pi,model,10000,10^(-9));
vEpsilon = policyEvaluation(piEpsilon,model,10000,10^(-9));
end
```

Code 11: Expected Sarsa algorithm, with options to let ϵ and α decrease over time.

B.2 Useful functions

```
% Samples from a categorical distribution defined by a
% (potentially unnormalized) vector of probabilities
function a = categoricalSample(probas)
x = sum(probas)*rand();
a = 1;
t = probas(1);
while t < x
  a = a+1;
  t = t+probas(a);
end
end
```

Code 12: Draws from a categorical distribution.

```
% yields the Epsilon-greedy action associated to action-state function Q(s,-)
% Qs should be a vector of size 4
function a = EGreedyActionFromQ(Qs,epsilon)
    probas=epsilon/4*ones(4,1);
    [~,best_a]=max(Qs);
    probas(best_a)=probas(best_a)+1-epsilon;
    a=categoricalSample(probas);
end
```

Code 13: Samples an ϵ-greedy action from an action-value function Q (or chooses the greedy action, if $\epsilon = 0$).

```
% yields the greedy policy pi associated to Q (assuming
% that Q is a proper state-action value function)
function piEpsilon = EGreedyPolicyFromQ(Q,epsilon)
    piEpsilon = epsilon/4*ones(size(Q,1), 4);
    for s=1:size(Q,1)
        [~,best_a]=max(Q(s,:));
        piEpsilon(s,best_a)=piEpsilon(s,best_a)+1-epsilon;
    end
end
```

Code 14: Returns the ϵ-greedy policy associated to an action-value function Q.

```
% yields the greedy policy pi associated to Q (assuming
% that Q is a proper state-action value function)
function pi = greedyPolicyFromQ(Q)
    pi = ones(size(Q,1), 1);
    for s=1:size(Q,1)
        [~,best_a]=max(Q(s,:));
        pi(s)=best_a;
    end
end
```

Code 15: Returns the deterministic greedy policy associated to an action-value function Q.

```
% yields the greedy policy pi associated to the value function v (assuming
% that v is a proper value function)
function pi = policyFromV(v,model)
    pi = ones(model.stateCount, 1);
    for s=1:model.stateCount
        Qs=zeros(4,1);
        for a=1:4
            % Q(s,a) of the policy being defined
            Qs(a)= model.P( s, :, a )* (model.R(s,a)+ model.gamma*v);
        end
        [~,best_a]=max(Qs);
        pi(s)=best_a;
    end
end
```

Code 16: Returns the deterministic greedy policy associated to a value function V.

B.3 Example of experimental code

```
% asymptotic performance
cliffworld
maxit=30;
maxeps=10000;
initialAlphas=[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1];
initialEpsilon=0.1;
alphaDeclineCoeff=0;
epsilonDeclineCoeff=0;
R_sarsa_average_10000=zeros(10,1);
R_QLearning_average_10000=zeros(10,1);
R_expected_average_10000=zeros(10,1);
N=10;

for j=1:10
for i=1:N
    rng(i)
    [v_qLearning, pi_qLearning,R_qLearning] = qLearning(model, maxit, maxeps,...
    initialAlphas(j),alphaDeclineCoeff,initialEpsilon,epsilonDeclineCoeff);
    [v_sarsa, pi_sarsa,vEpsilon_sarsa,piEpsilon_sarsa,R_sarsa] = ...
    sarsa(model, maxit, maxeps,initialAlphas(j),alphaDeclineCoeff,...
    initialEpsilon,epsilonDeclineCoeff);
    [v_expected, pi_expected,vEpsilon_expected,piEpsilon_expected,R_expected] =...
    expectedSarsa(model, maxit, maxeps,initialAlphas(j),...
    alphaDeclineCoeff,initialEpsilon,epsilonDeclineCoeff);
    R_QLearning_average_10000(j)=R_QLearning_average_10000(j)+mean(R_qLearning)/N;
    R_sarsa_average_10000(j)=R_sarsa_average_10000(j)+mean(R_sarsa)/N;
    R_expected_average_10000(j)=R_expected_average_10000(j)+mean(R_expected)/N;
end
end
```

Code 17: Code used to obtain the data in Figure 8.