# AIC/RL — Part III — Monte-Carlo and TD Learning

## The Code

In the previous exercise, you implemented model-based algorithms to find optimal policies for a given Markov Decision Process, which is represented in the class MarkovDecisionProcess. In this exercise, which considers **model-free** reinforcement learning, you will explicitly program agents which do not have direct access to the underlying MDP. They can however, interact with the MDP through the environment by executing actions and observing the results.

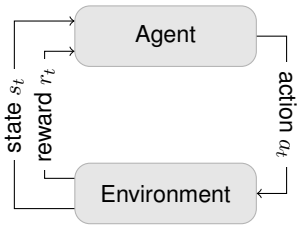


Figure 1: Agent-environment interface

We now describe the generic script experiment_episodic.py, which takes care of the interactions between the agent and the environment, as specified by the "agent-environment interface". We then present the Agent interface, which all agents must implement.

### Function `doEpisodes()`

The function doEpisodes in experiment_episodic.py is self-documenting; see the comments inside. It runs a certain amount of episodes. Each episode consists of an agent interacting with the environment until a terminal state is reached. Below is the basic structure (which differs from the full implementation) of this script:

```
# Implementation of the agent-environment interface
def doEpisodes(env,agent,n_episodes,...):

    for episode in range(n_episodes): # Run some episodes

        agent.newEpisode()                  # Tell agent episode is starting
        env.reset()                         # Reset the environment

        while not env.isFinished()          # Terminal state ends episode

            observation = env.getObservation()      # Get an observation
            agent.integrateObservation(observation) # Tell agent about it

            action = agent.getAction()              # Ask agent for action
            env.performAction(action)               # Tell environment about it

            reward = env.getReward()                # Get a reward
            agent.giveReward(reward)                # Tell agent about it
```

The doEpisodes function returns the learning curve, which is a chronological list of episode returns. It can be plotted with plotLearningCurve(...)

### Class `Agent`

The Agent class specified the agent interface, i.e. the function that each agent should implement. The most important functions are:

- __init__ (self, num_states, num_actions), which initializes an agent and lets it know how many states and actions are possible in the environment with which it will interact.
- newEpisode(self), which lets the agent know that a new episode is about to start.
- integrateObservation(self, obs), which is called at every discrete time step during an episode. It tells the agent about the current state (passed as an observation).
- getAction(self) should return the next action the agent would like to perform in the state it just observed. It corresponds to an agent's policy.
- giveReward(self), this gives a reward to the agent. This is called after each action the agent performs.

# 1 Monte Carlo Learning: state-values $V(s)$

Implement an agent that learns state-values $V(s)$ for a random policy. Please use the skeleton code in agent_montecarlo_statevalues.py. Note that for Monte Carlo the estimate of $V(s)$ is computed as

$$V^{\pi}(s) = average(Returns(s)), \tag{1}$$

which means your agent will probably have to store an (initially empty) list of returns for each state $s$.

**Q1** The accuracy of the estimates depends on the number of episodes the agent has seen. Compare the estimated values after 10, 100, 1000 episodes with the values acquired through policy evalution with dynamic programming for the same policy and environment.

**Q2** Change the size of the grid. How does this influence the convergence speed?

# 2 Monte Carlo Learning: state/action-values $Q(s, a)$

Implement an agent that learns state-values $Q(s, a)$ for a random policy. Remember that in MC, $Q$-values are implemented as $Q^{\pi}(s, a) = average(Returns(s, a))$, so you will have to adapt your lists which store the returns.

We recommend to copy the code from agent_montecarlo_statevalues.py into a new file called agent_montecarlo_stateactionvalues.py, because the code is quite similar.

# 3 Improving the policy: $\epsilon$-greedy exploration

Now it's time for your agent to improve its behavior, i.e. to go from exploration to exploitation. First, in the file agent_montecarlo_stateactionvalues.py, implement an $\epsilon$-greedy policy which does the following:

$$\pi(s) = \begin{cases} \text{random action (exploration)} & \text{with probability } \epsilon \\ \operatorname{argmax}_a Q(s, a) \text{ (exploitation)} & \text{with probability 1-}\epsilon \end{cases} \tag{2}$$

Note that to implement this, only the policy in the function getAction must be adapted.

**Q3** Compare the learning curves for $\epsilon = 1$, $\epsilon = 0.5$, and $\epsilon = 0.1$. What differences do you see, and why do they arise?

**Q4** Once $\epsilon$-greedy exploration works nicely, implement $\epsilon$ decay, by initially setting $\epsilon = 1$, and then decreasing it after each episode with a decay factor, e.g. $\epsilon \leftarrow 0.99\epsilon$. What happens if you decay epsilon too soon? Inspecting learning curves may help to understand.

**Q5** Discuss if $\epsilon$ is higher for yourself or your neighbour, at the current time in your lives. And who has the higher decay factor? Do you know someone whose $\epsilon$ may be a bit too high? And someone whose $\epsilon$ is too low?


## 4 Stochasticity in the environment: bumping your head

Now that are agent is able to learn optimal policies, let's change some properties of the environment. As we shall see, the exact same agent is able to learn optimal policies for all of them, even if it does not have a model.

The maze environment `EnvMaze` can be initialized with different levels of stochasticity, see the documentation of its __init__ function. If the argument stochasticity =0.0 (the default), the the environment is deterministic. Also, there is an argument bump_penalty which penalizes the agent for bumping into the wall. Its default is 0, but try setting it to -10.

Increase the size of the grid (e.g. to $8 \times 8$, and have your agent learn the $Q$-values and optimal policies. Do so for combinations of stochasticity =$\{0.0, 0.1, 0.5\}$ and bump_penalty=$\{0, -10\}$.

**Q6** Describe the difference in the policies. How is the agent adapting to the different environments?

## 5 Temporal Difference Learning

Implement TD learning, by implementing the equations from the lectures. If you want, you may skip $V$-value learning, and go directly to $Q$-values, i.e. the SARSA algorithm.

Hint: Since TD learning changes the values in place, there is no need to have lists for storing $Returns(s, a)$

We recommend to copy the code from agent_montecarlo_stateactionvalues.py into a new file called agent_td.py, because some parts of the code are similar.

**Q7** Compare the Monte-Carlo and TD learning agents using the script experiment_episodic_compare_agents.py. Who is learning quicker? You may need to increase the number of states in the environment to notice an effect.