# Reinforcement Learning II

**Andrea Bonarini**

Artificial Intelligence and Robotics Lab
Department of Electronics and Information
Politecnico di Milano

E-mail: bonarini@elet.polimi.it
URL:http://www.dei.polimi.it/people/bonarini

## Learning from interaction

R. S. Sutton, A. G. Barto, 1998, **Reinforcement learning: An introduction** (available on line)

Reinforcement learning is learning from interaction with an environment to achieve a goal, despite the **uncertainty** about the environment.

Agent's **actions affect the environment**, and so its options and opportunities, including the possibility to learn further.

A correct choice requires taking into account **indirect, delayed consequences** of actions, which often cannot be predicted appropriately, due to uncertainty and poor information about the environment.

The agent knows when it has reached its own **goals**.

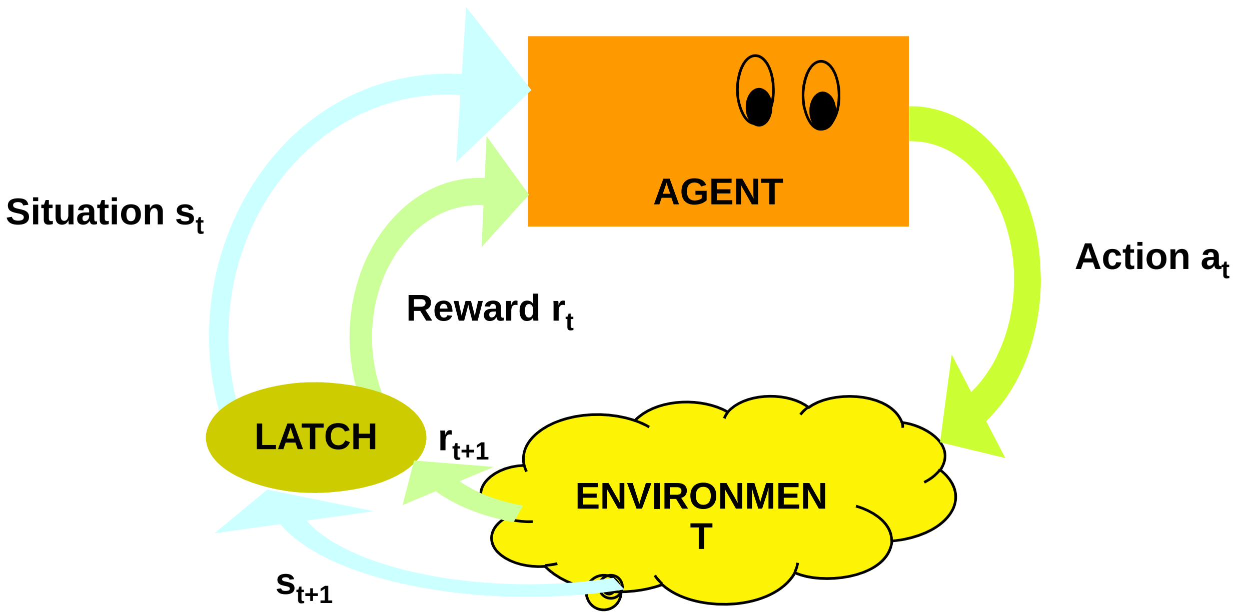
The  agent can use its **experience** to improve.

## The Reinforcement Learning Framework

The learner and decison-maker is called the **agent**

The thing it interacts with is called the **environment**

The agent performs **actions** to the environment and receives a (partial) description of the **situation**, and a **reward**



## Interaction

The agent interacts with the environment at each of a series of time steps (**discrete time**) t= 0, 1, 2,….

At each time step, the agent receives a description of the situation $s_t \in \boldsymbol{S}$, and selects an action $a_t \in \boldsymbol{A}(s_t)$ among those possible in situation $s_t$.

One time step later, in part possibly as consequence of its action, the agent receives a reward $r_{t+1}$ and finds itself in a new situation $s_{t+1}$.

At each time step the agent implements a mapping from situations to probabilities of selecting each possible action. This mapping is called the agent's policy $\pi_t$, where $\pi_t(s,a)$ is the probability that $a_t = a \in \boldsymbol{A}(s)$ if $s_t = s$.

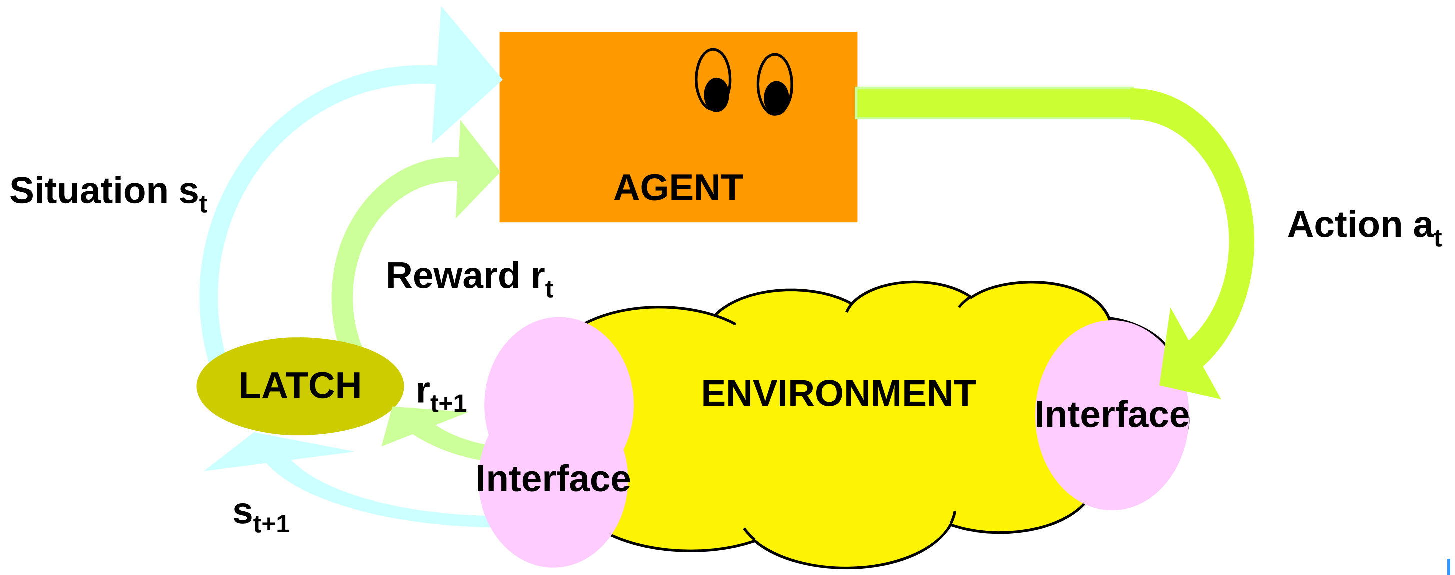
Reinforcement learning methods specify how the agent changes its policy as a result of its experiences in interacting with its environment

## Generic framework

Time steps do not need to be time intervals, but could be also discrete time points where a decision is required, or a discretization of continuous time as it always happen.

Actions can be low level control values for an engine, as well as “have a lunch”, or “buy ENEL bonds”.

Situations can be sensor readings, their interpretations or even more abstract descriptions.



## Interfaces and lost information

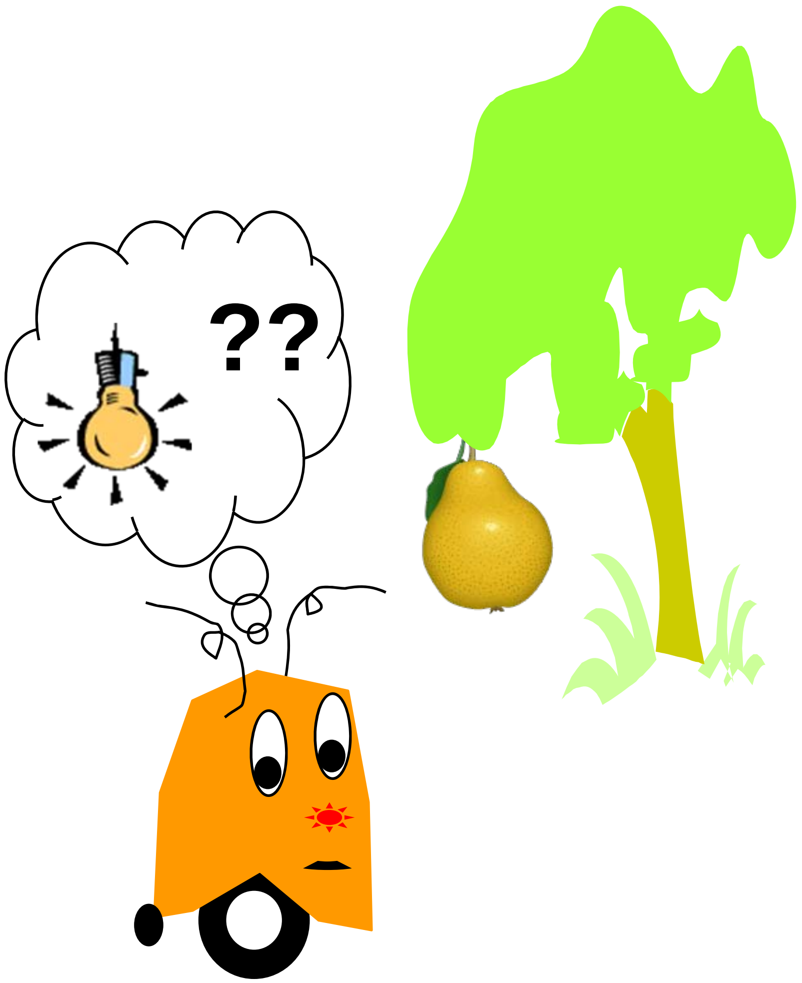
Every time we have interfaces we may lose some information, since interfaces translate some signal into something possibly different from the expected associated information .

E.g.:

a camera interfaces the world to the visual interpretation system, and introduces errors and approximation due to discretization in pixels, time of acquisition, sensor quality, settings, …

A visual interpretation system may provide a world description affected by errors and approximation due to interpretation algorithms, lack of infromation from sensors, discretization, ….

The difference between the situation description received by the learner and the reality may reduce the learning quality.



## Goal and reward

The reward signal is the way to communicate to the agent *what* we want to achieve, not *how* to achieve it. If we know “how” we don’t need any RL.

Reinforcement learning optimizes the reward received in the long run. It is a designer’s responsibility to define reward signals really related to the goal, and informative enough to drive learning.

Often, reward is computed directly from “sensors”: problems analogous to those mentioned for situation perception may arise

## Markovianity

Given the history of an agent

$H_t=\{s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{t-1}, a_{t-1}, r_t, s_t\}$

a situation is a **state** (i.e., it completely describes the system) iff

$\Pr\{s_{t+1}= s, r_{t+1}= r \mid s_t, a_t\} = \Pr\{s_{t+1}= s, r_{t+1}= r \mid H_t, a_t\}$

for all $t \geq 0$, $s \in \boldsymbol{S}$, $r \in \boldsymbol{R}$, $a \in A(s_t)$, and all possible $H_t$ .

In this case the environment is said to have the

*Markov property*

In other words: the state reached by performing an action at time t from a given state, and the received reward, only depends on the action and the state at time t, and not on any other element which coud be possibly undefined

## Markov Decision Process

When the environment has the Markov property, it and its interface define a ***Markov Decision Process*** (***MDP***)

If an MDP has a finite number of states, and a finite number of actions are available from each state, then it is a ***finite MDP***.

For finite MDPs we can represent the dynamics of the system by:

- the state transition probabilities

  $P^a_{ss'} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$

- the expected reward

  $R^a_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}$

If these are known we say that we have a reinforcement learning problem under conditions of *complete information*

## Does perfection exist?

Unfortunately, in real world we cannot work under conditions of *complete information* for many reasons:

- sensors are not enough (nor good enough) to capture all the information about the environment
- sensors translate signals to other signals, often introducing noise and approximation
- …

For instance, let us consider a vision system that should learn to recognize a target …

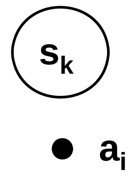
Anyway, RL algorithms work reasonably well also when we have incomplete information, of course, to some extent…

## State transition graphs

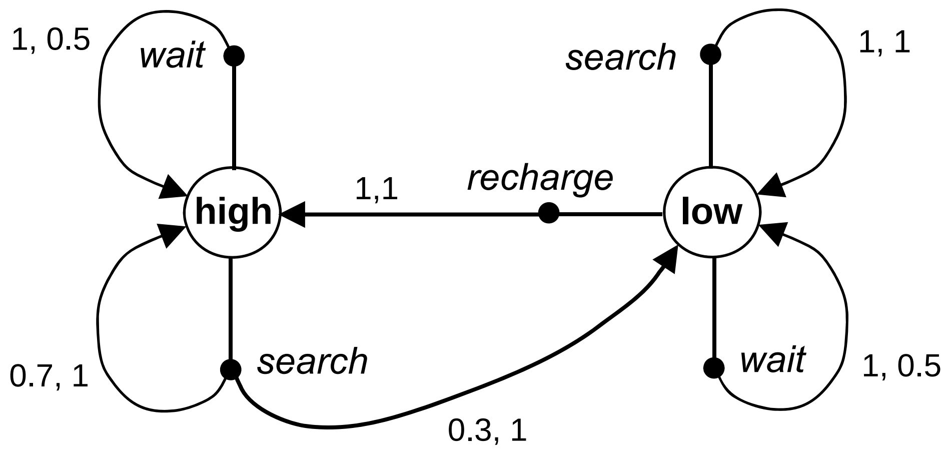
Useful tool to represent an MDP.

2 types of nodes:

- **state nodes** represent states
- **action nodes** represent actions



If we start from a state and take an action we follow the link to the corresponding node. From there, the environment will bring us to the state nodes connected with arrows to the action node, with the probabilities and rewards marked on the arrows



## Dynamic Programming

Dynamic programming methods can be used to estimate value functions and action functions.

Dynamic programming methods are guaranteed to converge when complete information is available, i.e., when we have a perfect model of the environment. This is not true in most real world cases, but dynamic programming techniques may obtain good results anyway.

The most important problem, apart the need for the model, is the required computational effort, which is polynomial in the number of states and actions. Anyway, this is better than exhaustive search which is exponential.

## Dynamic Programming: Policy Iteration I

```
1. Initialization
    $\pi \leftarrow$ an arbitrary policy
    $V \leftarrow$ an arbitrary function : $\mathbf{S} \rightarrow \Re$
    $\theta \leftarrow$ a small positive number
2. Policy evaluation
    Repeat
        $\Delta \leftarrow 0$
        For each $s \in \mathbf{S}$:
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V(s')]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    until $\Delta < \theta$
3. Policy improvement (cont.)
```

Probability to get in state s' from state s, by taking action a

## Dynamic Programming: Policy Iteration II

*3. Policy improvement (cont'd)*

```
policy-stable $\leftarrow$ true
    For each $s \in \mathbf{S}$:
        $b \leftarrow \pi(s)$
        $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V(s')]$
    If $b \neq \pi(s)$ then policy-stable $\leftarrow$ false
    If policy-stable then stop else go to 2
```

Notice that, whenever a policy is changed, all the policy evaluation (step 2) has to be performed again.

## Dynamic Programming: Value Iteration

```
1. Initialization
    $\pi \leftarrow$ an arbitrary policy
    $V \leftarrow$ an arbitrary function : $\mathbf{S} \rightarrow \Re$ (e.g., $V(s)=0$ for all $s$)
    $\theta \leftarrow$ a small positive number
2. State evaluation
    Repeat
        $\Delta \leftarrow 0$
        For each $s \in \mathbf{S}$:
            $v \leftarrow V(s)$
            $V(s) \leftarrow \max_a \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V(s')]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    until $\Delta < \theta$
3. Output
    Output a deterministic policy such that $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V(s')]$
```

## Bootstrapping

One of the key ideas of DP is that of *looking ahead* along a state transition and *using the current value estimates at the ending states to improve the value estimate for the starting state*.

Thus, estimates are built on the basis of other estimates. This is called the **bootstrapping property** and will be used later also for other RL methods.

## Co-convergence

The second idea of DP is that there are two simultaneous, interacting processes, one making the value function consistent with the current policy (the policy evaluation step) and the other making the policy greedy  with respect to the current value function (the policy improvement step).

In policy iteration the two steps alternate, each completing before the other begins.

In value iteration, only a single iteration of policy evaluation is performed between each policy improvement step.

The result is the same, over many iterations.

The results are obtained from the simultanous convergence of the two processes: *co-convergence*.

## Monte Carlo methods

Monte Carlo methods are used to estimate action-value functions when incomplete model is available to generate some experience-samples.

They are guaranteed to converge as the number of visits to each state goes to infinity.

The estimates for each state are independent.

The computational burden to estimate the value of a single state is independent of the number of the states: **interesting methods when the values of only a subset of the states are required**.

## A Monte Carlo algorithm

```
1. Initialization
    For each $s \in \mathbf{S}$ and $a \in A(s)$:
        Rewards (s, a)= empty list    ; a list of all rewards for s,a
        $\pi(s) \leftarrow$ an arbitrary policy
        $Q(s,a) \leftarrow$ an arbitrary action-value function : $\mathbf{S}, A(s) \rightarrow \Re$
2. Value improvement step
    Generate a trial using $\pi$
    For each pair s,a in the trial:
        $r \leftarrow$ the reward due to the first occurrence of s, a
        append r to Rewards(s,a)
        $Q(s,a)$ = average (Rewards(s,a))
3. Policy improvement step
    For each s in the trial:
        $\pi(s) = \text{argmax}_a Q(s,a)$
Go to 2.
```

## Temporal Difference methods

Learning without a model (as MC methods), by updating estimates based in part on other estimates (as DP).

Learning is driven by the temporal differences in prediction.

$$\Delta V_t(s_t) = \alpha \, [r_{t+1} + \gamma \, V_t(s_{t+1}) - V_t(s_t)]$$

$V_t(s_{t+1})$ is taken as an estimate of the correct one

The target is here $r_{t+1} + \gamma \, V_t(s_{t+1})$

Suitable in dynamic environments for online learning.

They have been proven to converge in the mean for any fixed policy $\pi$ for sufficiently small step size, and with probability 1 for step size decreasing in time.

## SARSA: On-policy TD control

*On-policy means that it evaluates and improves the same policy used to control.*

*SARSA uses past state and action, reward, and future state and action.*

*Algorithm:*

```
   $Q \leftarrow$ an arbitrary action-value function :
  For each trial:
      Initialize s
      Chose a from s using a policy based on Q
      Repeat for each step in the trial:
        Take action a, observe r and s'
        Choose a' from s' using a policy based on Q
        $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
        $a \leftarrow a'$
      until s is terminal
```

## Q-learning: Off-policy TD control

*Off-policy means that evaluates and improves a policy (estimation policy) different from that used to control (behavior policy). This makes it possible, e.g., to learn an optimal policy by experiencing another one, simpler to define*

*Algorithm:*

```
$Q(s,a) \leftarrow$ an arbitrary action-value function : $\mathbf{S}$, $A(s) \rightarrow \Re$
For each trial:
    Initialize s
    Repeat for each step in the trial:
      Choose a from s using a policy based on Q
      Take action a, observe r and s'
      $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
      $s \leftarrow s'$
      until s is terminal
```

## Eligibility traces

An eligibility trace is a temporary record of the occurrence of an event such as a state visit or an action selection.

The trace marks the memory parameters associated with the event as *eligible* for undergoing learning changes.

When reinforcement occurs, the eligible states or actions are assigned credit for it.

It is a way to distribute reinforcement not only to the current state or action, but also to those that brought the system in the current state.

Useful in case of sparse (delayed) reinforcement.

## What is an eligibility trace?

Let's denote the trace for state s at time t as $e_t(s) \in \Re$

On each step the traces for each state decay by $\gamma\lambda$ and the trace of the visited state is incremented by 1:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$

where $\gamma$ is the usual discount parameter and $0 \leq \lambda \leq 1$

This is called an *accumulating* trace.

In many problems a *replace* trace may reduce the learning time:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases}$$

## TD($\lambda$)

The TD error used in the TD methods becomes here:

$$\Delta V_t(s) = \alpha \, [r_{t+1} + \gamma \, V_t(s_{t+1}) - V_t(s_t)] \, e_t(s)$$

As we will see in the algorithm, values of ALL the states are updated using the TD error (the term in square brackets), which varies from time to time, but is the same for each state. Different states are assigned credit proportionally to their eligibility.

## TD($\lambda$)

```
Initialize $V(s)$ arbitrarily and  $e(s) = 0$, for all $s$
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        $a \leftarrow$ action selected for $s$ using a policy $\pi$  based on $Q$
        Take action $a$, observe $r$ and  the next state $s'$
        $\delta \leftarrow r + \gamma V(s') - V(s)$
        $e(s) \leftarrow e(s) + 1$
        For all $s$:
          $V(s) \leftarrow V(s) + \alpha \delta e(s)$
          $e(s) \leftarrow \gamma\lambda e(s)$
        $s \leftarrow s'$
    until $s$ is terminal
```

## Q($\lambda$)

```
Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all $s,a$
Repeat (for each episode):
    Initialize $s, a$
    Repeat (for each step of episode):
        Take action $a$, observe $r,s'$
        Choose $a'$ from $s'$, using a policy based on $Q$
        $a^* \leftarrow \text{argmax}_b\, Q(s',b)$ (if $a'$ ties for the max, then $a^* \leftarrow a'$ )
        $\delta \leftarrow r + \gamma\, Q(s',a^*) - Q(s,a)$
        $e(s,a) \leftarrow e(s,a) + 1$
        For all $s,a$:
          $Q(s,a) \leftarrow Q(s,a) + \alpha\, \delta\, e(s,a)$
          If $a' = a^*$ then $e(s,a) \leftarrow \gamma\lambda\, e(s,a)$ else $e(s,a) \leftarrow 0$
        $s \leftarrow s'$; $a \leftarrow a'$
    until $s$ is terminal
```