Reinforcement Learning

Introduction: Framework, concepts and definitions

Mario Martin

Mario Martin - CS-UPC

February 11, 2026

What is reinforcement learning?: RL Framework

Some Literature

“So saying, they handcuffed him, and carried him away to the regiment. There he was made to wheel about to the right, to the left, to draw his rammer, to return his rammer, to present, to fire, to march, and they gave him thirty blows with a cane; the next day he performed his exercise a little better, and they gave him but twenty; the day following he came off with ten, and was looked upon as a young fellow of surprising genius by all his comrades."

Candide: or, Optimism.
Voltaire (1759)

Reinforcement Learning concept

Main characteristics of RL:

Agent-like learning:

1. Goal is learning a behavior (*policy*), not a class
2. No example dataset
3. Grounded learning: Agent is *actively* collecting data in the environment

Informal definition

Learning about, from, and while interacting with an environment to achieve a goal (learning a behavior).

Reinforcement Learning concept

Main characteristics of RL:
Agent-like learning:

1. Goal is learning a behavior (*policy*), not a class
2. No example dataset
3. Grounded learning: Agent is *actively* collecting data in the environment

Informal definition

Learning about, from, and while interacting with an environment to achieve a goal (learning a behavior).

Reinforcement Learning concept

Main characteristics of RL:

Agent-like learning:

1. Goal is learning a behavior (*policy*), not a class
2. No example dataset
3. Grounded learning: Agent is *actively* collecting data in the environment

Informal definition

Learning about, from, and while interacting with an environment to achieve a goal (learning a behavior).

Reinforcement Learning concept

Main characteristics of RL:

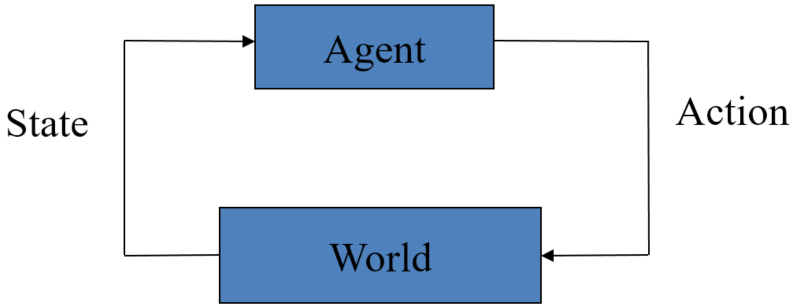
Agent-like learning:

1. Goal is learning a behavior (*policy*), not a class
2. No example dataset
3. Grounded learning: Agent is *actively* collecting data in the environment

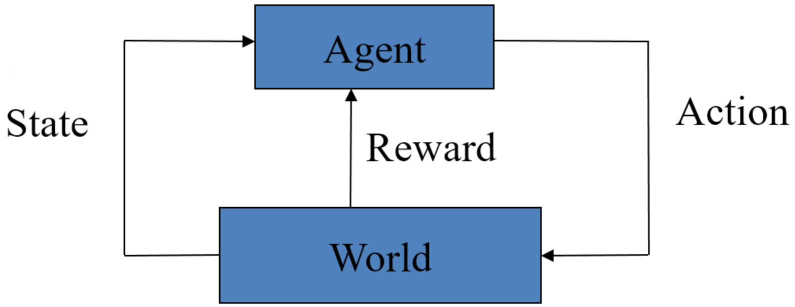
Informal definition

Learning about, from, and while interacting with an environment to achieve a goal (learning a behavior).

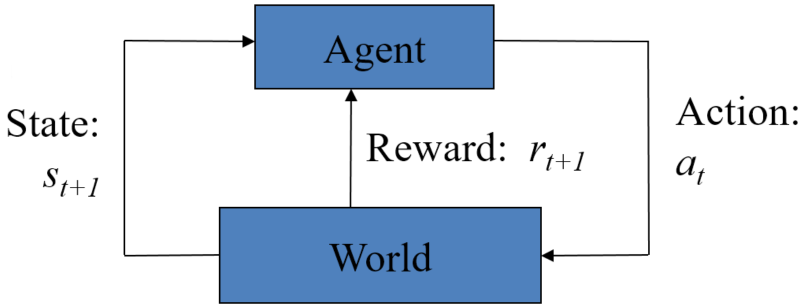
RL Framework



RL Framework



RL Framework



RL Framework

Why use reward instead of examples?:

1. Usually it's easy to define a reward function (not always).
2. You don't need to know the goal behavior to train an agent (in contrast to supervised learning).
3. Behavior is grounded and efficient (optimal in some cases) given perceptual system and possible actions of the agent.

Reward assumption

All goals can be formalized as the outcome of maximizing a cumulative reward

See position in Reward is enough (Silver et al. 21) and On the Expressivity of Markov Reward (Abel et al. 21)

RL Framework

Why use reward instead of examples?:

1. Usually it's easy to define a reward function (not always).
2. You don't need to know the goal behavior to train an agent (in contrast to supervised learning).
3. Behavior is grounded and efficient (optimal in some cases) given perceptual system and possible actions of the agent.

Reward assumption

All goals can be formalized as the outcome of maximizing a cumulative reward

See position in Reward is enough (Silver et al. 21) and On the Expressivity of Markov Reward (Abel et al. 21)

RL Framework

Why use reward instead of examples?:

1. Usually it's easy to define a reward function (not always).
2. You don't need to know the goal behavior to train an agent (in contrast to supervised learning).
3. Behavior is grounded and efficient (optimal in some cases) given perceptual system and possible actions of the agent.

Reward assumption

All goals can be formalized as the outcome of maximizing a cumulative reward

See position in Reward is enough (Silver et al. 21) and On the Expressivity of Markov Reward (Abel et al. 21)

RL Characteristics

What makes reinforcement learning harder than other machine learning paradigms?

- Feedback is not the *right action*, but a *sparse* scalar value (*reward* function).
- Relevant feedback is delayed, not instantaneous.
- Time really matters (sequential, non i.i.d. data).
- Environment can be stochastic and uncertain.

RL Definition

Informal definition

Learning about, from, and while interacting with an environment to achieve a goal (learning a behavior).

read as

Formal definition

Learning a mapping from situations to actions to maximize long-term reward, without using a model of the world.

RL Definition

Informal definition

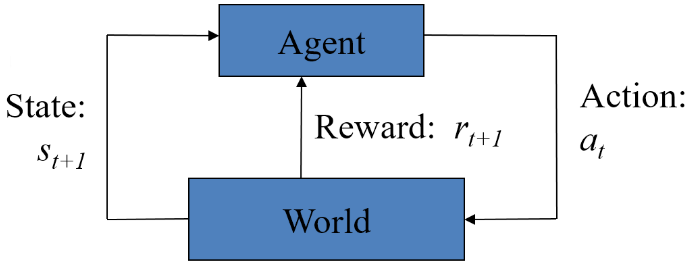
Learning about, from, and while interacting with an environment to achieve a goal (learning a behavior).

read as

Formal definition

Learning a mapping from situations to actions to maximize long-term reward, without using a model of the world.

RL Framework

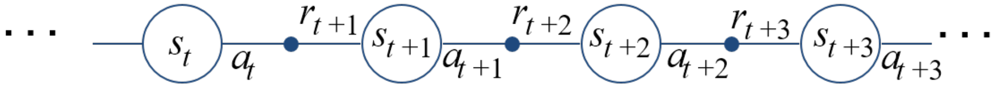


Agent and environment interact at discrete time steps: $t = 0,\ 1,\ 2,\ \ldots$

- Agent observes state at step t: $s_t \in S$
- produces action at step t: $a_t \in A(s_t)$
- gets resulting reward: $r_{t+1} \in \mathbb{R}$, and resulting next state: s_{t+1}

RL Framework

Snapshot of a trial of the agent:



RL Framework: MDP process

- RL Problem can be formulated as a Markov Decision Process (MDP): a tuple $< S, A, P, R >$ where
  - S: Finite set of states
  - A: Finite set of actions
  - P: Transition Probabilities (Markov property):

$$P^a_{ss'} = \Pr\{s_{t+1} = s' \mid s_t = s,\ a_t = a\}\,\forall s, s' \in S,\ a \in A(s).$$

  - R: Reward Probabilities:

$$R^a_s = \mathbb{E}\{r_{t+1} \mid s_t = s,\ a_t = a, s_{t+1} = s'\}\,\forall s, s' \in S,\ a \in A(s).$$

- Some constraints can be relaxed later:
  - Markov property (fully vs. partial observability)
  - Infinite (or continuous) sets of actions and states

RL Framework: MDP process

- RL Problem can be formulated as a Markov Decision Process (MDP): a tuple $< S, A, P, R >$ where
  - S: Finite set of states
  - A: Finite set of actions
  - P: Transition Probabilities (Markov property):

$$P^a_{ss'} = \Pr\{s_{t+1} = s' \mid s_t = s,\ a_t = a\} \forall s, s' \in S,\ a \in A(s).$$

  - R: Reward Probabilities:

$$R^a_s = \mathbb{E}\{r_{t+1} \mid s_t = s,\ a_t = a, s_{t+1} = s'\} \forall s, s' \in S,\ a \in A(s).$$

- Some constraints can be relaxed later:
  - Markov property (fully vs. partial observability)
  - Infinite (or continuous) sets of actions and states

RL elements:

RL Elements

In RL are **key** the following elements:

1. **Policy**: What to do.
2. **Model**: What follows what. Dynamics of the environment.
3. **Reward**: What is good
4. **Value function**: What is good because it *predicts reward*.

Policy

- A **policy** is the agent's behavior
- It is a map from current state to action to execute:

$$\pi : s \in \mathcal{S} \longrightarrow a \in \mathcal{A}$$

- *Policy* could be deterministic:

$$a = \pi(s)$$

- ... or stochastic:

$$\pi(a|s) = \mathbb{P}[A_t = s|S_t = s]$$

Model

- A **model** predicts next state and reward
- Allows modeling of stochastic environments with probability transition functions:
  - $\mathcal{P}$ predicts the next state

$$\mathcal{T}(s, a, s') = P^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- *Usually not known by the agent*

Rewards

- **Immediate reward** r_t is a *scalar* feedback value that depends on the current state r_t given that current state is S_t.
- **Reward function** $\mathcal{R}$ determines (immediate) reward r_t at each step of the agent's life

$$\mathcal{R}(s) = \mathbb{E}[r_t | S_t = s]$$

- It can be very sparse and does not evaluate of the goodness of the last action but the goodness of the whole chain of actions (*trajectory*).
- Sometimes written in the *equivalent* form $r(s, a)$:

$$\mathcal{R}(s, a) = \mathbb{E}[r_{t+1} | S_t = s, A_t = a]$$

- Difference in reward in post-action or pre-action execution

Rewards: examples

- Fly stunt manoeuvres in a helicopter
  - +ve reward for following desired trajectory
  - -ve reward for crashing
- Defeat the world champion at Go
  - +ve/-ve reward for winning/losing a game
- Make a humanoid robot walk
  - +ve reward for forward motion
  - -ve reward for falling over
- Play Atari games better than humans
  - +ve reward for increasing/decreasing score

[Kinds of experiences]

- Agents will learn from experiences that in this case are sequences of actions
- Interaction of the agent with the environment can be for organized in two different ways:
  - **Trials (or episodic learning)**: The agent has a final state after which he receive the reward. In some cases it has to be achieved after a limited maximum time H. After he arrives to the goal state (or surpass the maximum time allowed), a new *trial* is started.
  - **Non-ending tasks**: The agent has no limit in time or it has not a clear *final state*. Learning by trials can be also simulated with non-ending tasks by adding random extra-transitions from goal state to initial states.

Long-term reward

Formal definition of RL

Learning a mapping from situations to actions to maximize **long-term reward**, without using a model of the world.

- The agent's job is to maximise cumulative reward over an episode
- Long term reward must be defined in terms of the goal of the agent
- Definition of long-term reward must be derived from local rewards

Long-term Return

First intuitive definition of long-term reward:

Finite horizon undiscounted return

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_H = \sum_{k=0}^{H} r_{t+k+1}$$

Problem: Optimal policy depends on horizon H and becomes no-stationary

- Sum of rewards obtained in trajectory
- H because we want a limit in the sum of rewards to compare trajectories

Long-term Return

Infinite horizon *discounted* return

The return R_t is the total discounted reward from time-step t.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

- The discount $\gamma \in [0, 1]$ is the present value of future rewards. Usually very close to 1.
- The value of receiving reward r after $k + 1$ time-steps is $\gamma^k r$.
- This values immediate reward above delayed reward: γ close to 0 leads to *myopic* evaluation γ close to 1 leads to *far-sighted* evaluation

Long-term Return

$$\begin{tikzcd}
 & S_2 & & \\
S_1 \arrow[ur, "\text{a}=A_1,\ \textbf{r=5}"] \arrow[dr, "\text{a}=A_2,\ \text{r}=0"] & & & \\
 & S_3 \arrow[r, "\text{a}=A_1, A_2,\ \text{r}=0"] & S_4 \arrow[r, "\text{a}=A_1, A_2,\ \textbf{r=10}"] & S_5
\end{tikzcd}$$

$$\gamma = 0.7$$

Long-term Return

- Infinite horizon *discounted* return is limited by:

$$R_t \leq \sum_{k=0}^{\infty} \gamma^k r_{max} = \frac{r_{max}}{1-\gamma}$$

- So, also useful for learning non-ending tasks, because addition is unlimited.
- Greedy policies are stationary
- Elegant and convenient recursive definition (see Bellman eqs. later)
- Choice of reward function and maximization of Long-term Return should lead to *desired* behavior.

Long-term Return

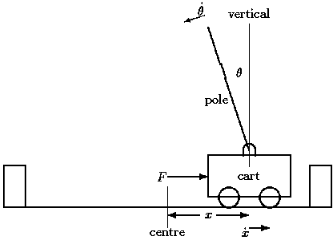
- Infinite horizon *discounted* return is limited by:

$$R_t \leq \sum_{k=0}^{\infty} \gamma^k r_{max} = \frac{r_{max}}{1-\gamma}$$

- So, also useful for learning non-ending tasks, because addition is unlimited.
- Greedy policies are stationary
- Elegant and convenient recursive definition (see Bellman eqs. later)
- Choice of reward function and maximization of Long-term Return should lead to *desired* behavior.

Long-term Return examples

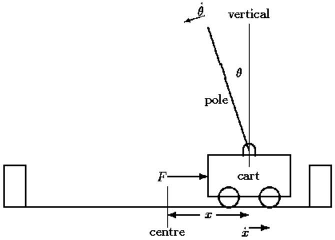
- Pole balancing example:



- Episodic learning.
- Three possible actions: $\{-F, 0, F\}$
- Sate is defined by $(x, \dot{x}, \theta, \dot{\theta})$
- Markovian problem because $(x', \dot{x'}, \theta', \dot{\theta}') = F(x, \dot{x}, \theta, \dot{\theta})$
- Goal: $|\theta|$ bellow a threshold (similar to a Segway problem)

Long-term Return examples

Reward definition:

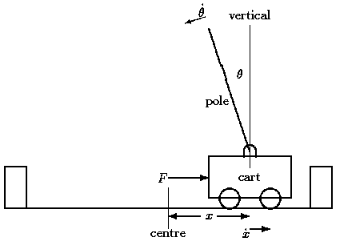


Case 1: $\gamma = 1, r = 1$ for each step except $r = 0$ when pole falls. $\Longrightarrow R =$ number of time steps before failure

- Return is maximized by avoiding failure for as long as possible.

Long-term Return examples

Reward definition:

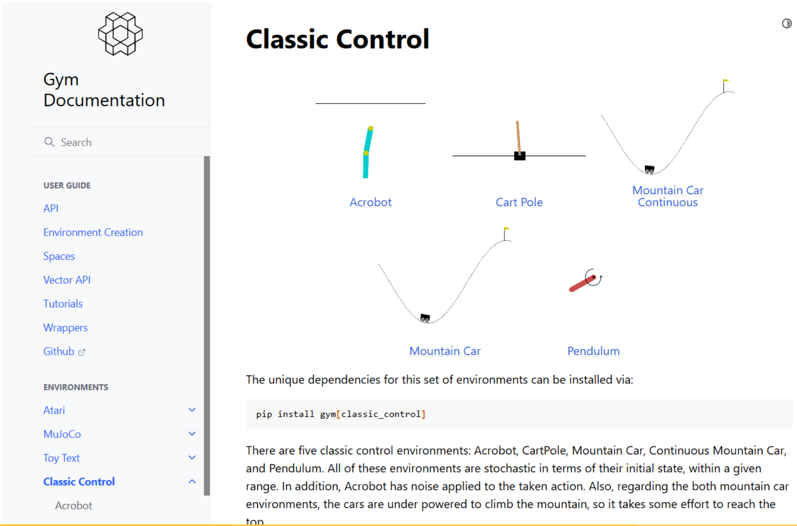


Case 2: $\gamma < 1$, $r = 0$ for each step, and $r = -1$ when pole falls. $\Longrightarrow R = -\gamma^k$ for k time steps before failure

- Return is maximized by avoiding failure for as long as possible.

Long-term Return examples

Other examples from OpenAI gym[1]:



Gym Documentation

Search

USER GUIDE

API

Environment Creation

Spaces

Vector API

Tutorials

Wrappers

Github

ENVIRONMENTS

Atari

MuJoCo

Toy Text

Classic Control

Acrobot

Classic Control

The unique dependencies for this set of environments can be installed via:

```
pip install gym[classic_control]
```

There are five classic control environments: Acrobot, CartPole, Mountain Car, Continuous Mountain Car, and Pendulum. All of these environments are stochastic in terms of their initial state, within a given range. In addition, Acrobot has noise applied to the taken action. Also, regarding the both mountain car environments, the cars are under powered to climb the mountain, so it takes some effort to reach the top

Value functions

Value function

- Value function is a prediction of future reward
- Used to evaluate goodness/badness of states
- Depends on the agents policy...
- ... and is used to select between actions

state-value function $V^\pi(s)$

$V^\pi(s)$ is defined as the expected return starting from state s, and then following policy π

$$V^\pi(s) = \mathbb{E}_\pi[R_t|S_t = s] = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\}$$

Q-Value function

action-value function $Q^\pi(s,a)$

$Q^\pi(s,a)$ is the expected return starting from state s, taking action a, and then following policy π

$$Q^\pi(s,a) = \mathbb{E}_\pi[R_t | S_t = s, A_t = a] = \mathbb{E}_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\}$$

Bellman expectation equation

The value function can be decomposed into two parts:

- immediate reward r_{t+1}
- discounted value of successor state $\gamma V^\pi(S_{t+1})$

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_t|S_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots |S_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \ldots)|S_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma R_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(S_{t+1})|S_t = s] \end{aligned}$$

Bellman Expectation Equation for V^π

So, the state-value function can again be decomposed recursively into immediate reward plus discounted value of successor state,

Bellman equation for state-value function

$$V_\pi(s) = \mathbb{E}^\pi[r_{t+1} + \gamma V^\pi(S_{t+1}) | S_t = s]$$

Equivalent expression without the expectation operator:

$$V^\pi(s) = \sum_{s'} P_{ss'}^{\pi(s)} \left[R(s') + \gamma V^\pi(s')\right]$$

Bellman Expectation Equation for Q^π

The action-value function can similarly be decomposed:

$$\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}_\pi[R_t | S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[\ \underbrace{r_{t+1}}_{\text{because } a} \ + \underbrace{\gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots}_{\text{following } \pi} | S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \ldots) | S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma R_{t+1} | S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(S_{t+1}) | S_t = s, A_t = a]
\end{aligned}$$

Bellman Expectation Equation for Q^π

Notice that:

$$V^\pi(S_t) = Q^\pi(S_t, \pi(S_t))$$

So,

Bellman equation for state-action value function

$$Q^\pi(s, a) = \mathbb{E}_\pi[r_{t+1} + \gamma Q^\pi(S_{t+1}, \pi(S_{t+1}))|S_t = s, A_t = a]$$

Bellman Expectation Equation for Q^π

Notice that:

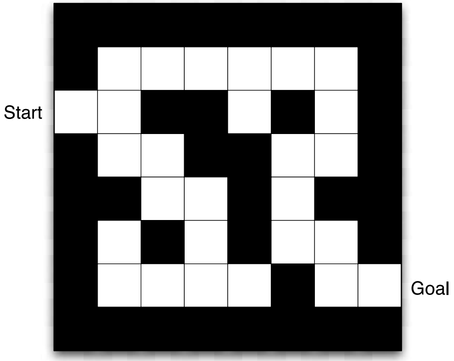
$$V^\pi(S_t) = Q^\pi(S_t, \pi(S_t))$$

So,

Bellman equation for state-action value function

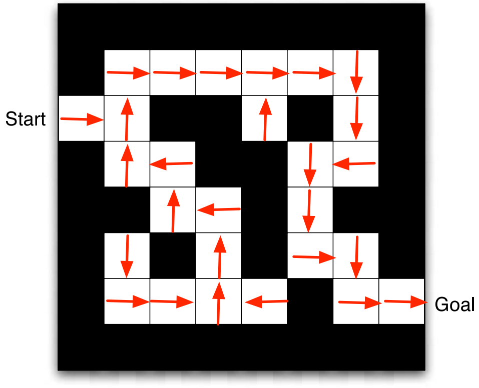
$$Q^\pi(s,a) = \mathbb{E}_\pi[r_{t+1} + \gamma Q^\pi(S_{t+1}, \pi(S_{t+1}))|S_t = s, A_t = a]$$

Maze example



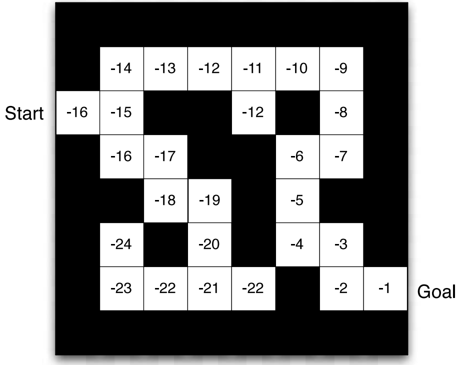
- Rewards: -1 per time-step
- Actions: N, S, W, E
- States: Agent's location

Maze example: policy



- Arrows represent policy $\pi(s)$ for each state s

Maze example: value function



- Numbers represent $V^{\pi}(s)$ for each state s, for $\gamma = 1$

Policy evaluation (1)

- Given π, **policy evaluation** methods obtain V^π (*same procedures can be used to compute* Q^π).
- First method: Algebraic solution using Bellman equations in matrix form

$$\begin{aligned} V^\pi(s) &= R(s,\pi(s)) + \gamma \sum_{s'} P_{ss'}^{\pi(s)} V^\pi(s') \\ V^\pi &= R + \gamma P^\pi V^\pi \\ V^\pi - \gamma P^\pi V^\pi &= R \\ (I - \gamma P^\pi) V^\pi &= R \end{aligned}$$

Algebraic solution

$$V^\pi = (I - \gamma P^\pi)^{-1} R$$

- Computational cost is $O(n^3)$ where n is the number of states

Policy evaluation (1)

- Given π, **policy evaluation** methods obtain V^π (*same procedures can be used to compute* Q^π).
- First method: Algebraic solution using Bellman equations in matrix form

$$\begin{aligned} V^\pi(s) &= R(s,\pi(s)) + \gamma \sum_{s'} P_{ss'}^{\pi(s)} V^\pi(s') \\ V^\pi &= R + \gamma P^\pi V^\pi \\ V^\pi - \gamma P^\pi V^\pi &= R \\ (I - \gamma P^\pi) V^\pi &= R \end{aligned}$$

Algebraic solution

$$V^\pi = (I - \gamma P^\pi)^{-1} R$$

- Computational cost is $O(n^3)$ where n is the number of states

Policy evaluation (2)

- Second method: iterative value policy evaluation
  - Given arbitrary V as estimation of V^π, we can tell the error using Bellman equations:

$$error = \max_{s \in S} \left| V(s) - \sum_{s'} P_{ss'}^{\pi(s)} \left[R(s') + \gamma V(s') \right] \right|$$

  - Consider to apply iteratively Bellman equations to update V for all states (*Bellman operator*)

$$V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)} \left[R(s') + \gamma V(s') \right]$$

  - Convergence can be proved: applying Bellman operator, error is reduced by a γ factor (*contraction*)
  - So, apply updates of Bellman operator until convergence.
  - Solution is a fixed point of the application of this operator

Policy evaluation (2)

Iterative value policy evaluation

```
Given $\pi$, the policy to be evaluated, initialize $V(s) = 0 \;\; \forall s \in S$
repeat
    $\Delta \leftarrow 0$
    for each $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)} \left[ R(s') + \gamma V(s') \right]$
        $\Delta \leftarrow \max(\Delta, |v - V(S)|)$
    end for
until $\Delta < \theta$ (a small threshold)
```

Policy evaluation (2)

- Value iteration converges to optimal value: $V \to V^{\pi}$
- Update of all states using the Bellman equation

$$V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)} \left[R(s') + \gamma V(s') \right]$$

  is called also the Bellman operator
- It can be proved that iterative application of the Bellman operator is max-norm **contraction** that ends in a fixed point
- The **fixed point** is exactly the solution V^{π}

Optimal Policies

Relationship between value functions and policies

- We can define a partial ordering of policies $\leq$ in the following way:

$$\pi' \leq \pi \iff V^{\pi'}(s) \leq V^{\pi}(s) \ \ \forall s$$

[Remember that $V^{\pi}(s) \equiv Q^{\pi}(s, \pi(s))$]

- Under this ordering, we can prove that:
  - There exists at least an optimal policy (π^*)
  - Could be not unique
  - In the set of optimal policies some are deterministic
  - All share the same value function

Relationship between value functions and policies

We say a policy π is **greedy** when:

$$\pi(S_t) = \arg\max_{a \in A} \ \mathbb{E}_\pi[R_{t+1}]$$

if value states are estimations of R_t, then in greedy policies:

$$\pi(s) = \arg\max_{a \in A} Q^\pi(s, a)$$

It is easy to see that **the optimal policy is greedy**.

Relationship between value functions and policies

Infinite number of actions (f.i. continuous space of actions)

Implementations of:

$$\pi(S_t) = \arg\max_{a \in A} \mathbb{E}_\pi[R_{t+1}]$$

is **easy** when we have a **finite number of actions**. When we have an infinite number of actions like in case of continuous space of actions (parametrized actions), computation is harder!

Finding Policies: Model based methods

Finding policies

- Knowing that optimal policy is greedy...
- ... and using recursive Bellman equations
- we can apply *Dynamic Programming* (DP) techniques to find the optimal policies
- Main methods to find optimal policies using DP
  - Policy iteration (PI)
  - Value iteration (VI)
- *Model based method*: In these methods, knowledge of the model is assumed.

Finding policies

- Knowing that optimal policy is greedy...
- ... and using recursive Bellman equations
- we can apply *Dynamic Programming* (DP) techniques to find the optimal policies
- Main methods to find optimal policies using DP
  - Policy iteration (PI)
  - Value iteration (VI)
- *Model based method*: In these methods, knowledge of the model is assumed.

Finding policies: Policy iteration

- A policy π can be improved iif

$$\exists\ s \in S, a \in A \text{ such that } Q^{\pi}(s,a) > Q^{\pi}(s,\pi(s))$$

- Obvious. In this case, π is not optimal and can be improved setting $\pi(s) = a$
- Simple idea for the algorithm:
  1. Start from random policy π
  2. Compute Q^{π}
  3. Check for each state if the policy can be improved (and improve it)
  4. If policy cannot be improved, stop. In other case repeat from 2.

Finding policies: Policy iteration

Policy Iteration (PI)

```
Initialize $\pi, \forall s \in S$ to a random action $a \in \mathcal{A}(s)$, arbitrarily
repeat
    $\pi' \leftarrow \pi$
    Compute $Q^{\pi}$ for all states using a policy evaluation method
    for each state $s$ do
        $\pi(s) \leftarrow \arg\max_{a \in A} Q(s,a)$
    end for
until $\pi(s) = \pi'(s) \ \ \forall s$
```

Finding policies: Policy iteration

Theorem

Policy iteration is *guaranteed to converge* and at convergence, the current policy and its value function are the *optimal policy* and the optimal value function!

- At each iteration the policy improves. This means:
  - that a given policy can be encountered at most once (so number of iterations is bounded) ...
  - ... and the number of possible policies is finite ($|A|^{|S|}$), so it must stop at some point (usually in polynomial time).
  - At end, the policy cannot be improved. That means that the policy is optimal (because there are not suboptimal policies that cannot be improved)

Finding policies: Asynchronous versions

- PI requires exhaustive sweeps of the entire state set.
- Asynchronous PI does not use complete sweeps.
- Pick a state at *random* and apply the appropriate backups for Q and π. Repeat until convergence criterion is meet:
- Still need lots of computation, but does not get locked into hopelessly long sweeps