Reinforcement Learning (1)

Machine Learning 64-360, Part II

Norman Hendrich

University of Hamburg
MIN Faculty, Dept. of Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
hendrich@informatik.uni-hamburg.de

17/06/2015

Schedule

Reinforcement-Learning: a set of learning problems and diverse algorithms and approaches to solve the problems.

- 17/06/2015 Introduction, MDP
- 22/06/2015 Value Functions, Bellmann Equation
- 24/06/2015 Monte-Carlo, TD(λ)
- 29/06/2015 Function Approximation
- 01/07/2015 Function Approximation
- 06/07/2015 Inverse-RL, Apprenticeship Learning
- 08/07/2015 Applications in Robotics, Wrap-Up

Recommended Literature

- S. Sutton and A. G. Barto, *Reinforcement Learning, an Introduction*, MIT Press, 1998
  http://webdocs.cs.ualberta.ca/˜sutton/book/ebook/
- C. Szepesvari, *Algorithms for Reinforcement Learning*, Morgan & Claypool Publishers,
  http://www.ualberta.ca/˜szepesva/papers/RLAlgsInMDPs.pdf
- Kaelbling, Littman, and A. Moore, *Reinforcement learning: a survey*, JAIR 4:237-285, 1996
- D.P. Bertsekas and J.N. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, 1996 (theory!)
- several papers to be added later

Context

operations research

artificial Intelligence (planning)

reinforcement Learning (RL)

psychology

automation and control (robotics)

neuroscience

artificial neural networks

What is Reinforcement Learning?

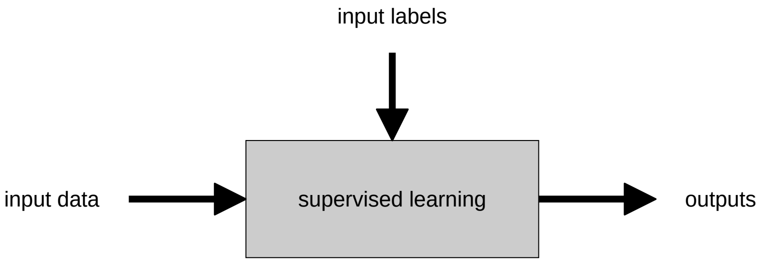
the term usually refers to the problem/setting, rather than a particular algorithm:

- learning **from/during** interaction with an external environment
- learning "what to do" — how to map situations to actions — to maximize a numeric reward signal
- learning about delayed rewards
- learning about structure, continuous learning
- goal-oriented learning

- in-between supervised and unsupervised learning
- applications in many areas

Supervised Learning

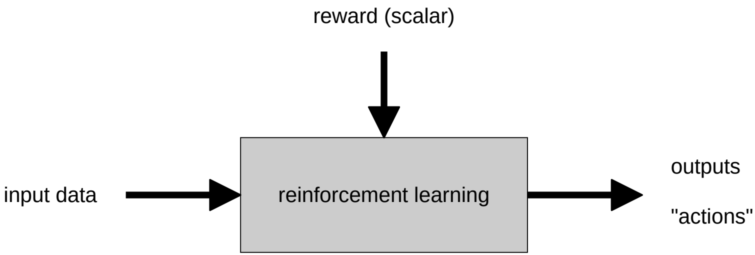
training data = inputs + desired (target) outputs



error = (target output – actual system output)

Reinforcement Learning

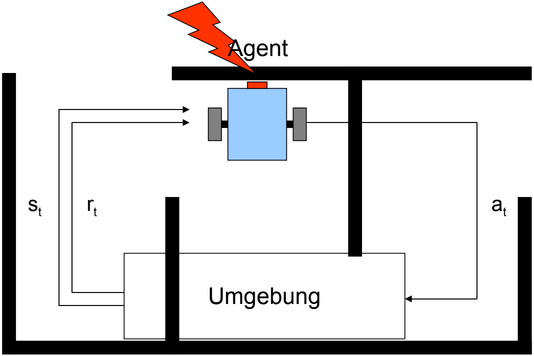
training information = evaluation ("rewards" / "penalties")



no way to directly calculate an error
instead: try to achieve as much *reward* as possible

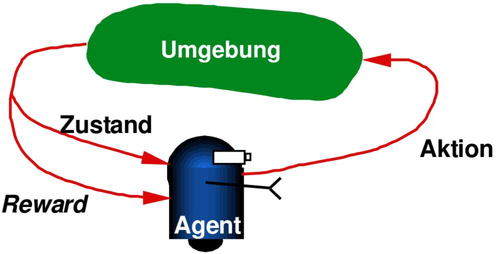
Reinforcement Learning

- goal: act „successfully“ in the environment
- this implies: maximize the sequence of rewards R_t

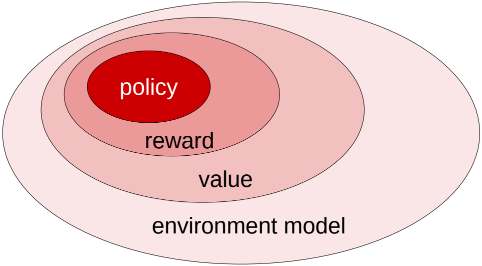


The agent

- continuous learning and planning
- affects the environment
- with or without a model of the environment
- environment may be stochastic and uncertain

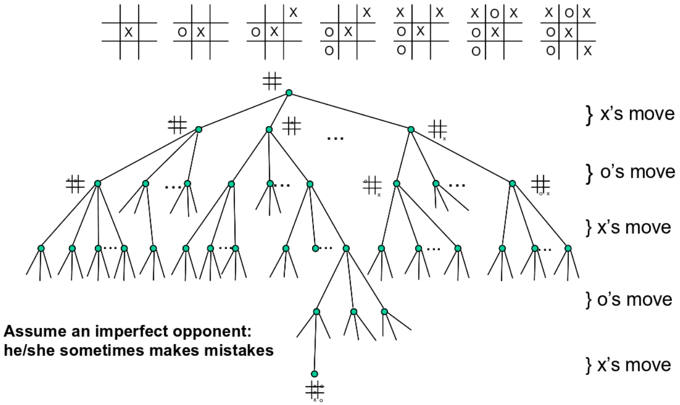


Elements of RL



- **policy:** what to do
- **reward:** what is good (immediately)
- **value:** estimate the expected reward (long-run)
- **model:** how does the environment work?

Example: playing Tic-tac-toe



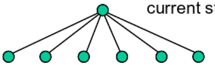
winning requires an imperfect opponent: he/she makes mistakes

RL-approach for Tic-tac-toe

1. Make a table with one entry per state:

State	$V(s)$ – estimated probability of winning	
	.5	?
	.5	?
⋮		
	1	win
⋮		
	0	loss
⋮		
	0	draw

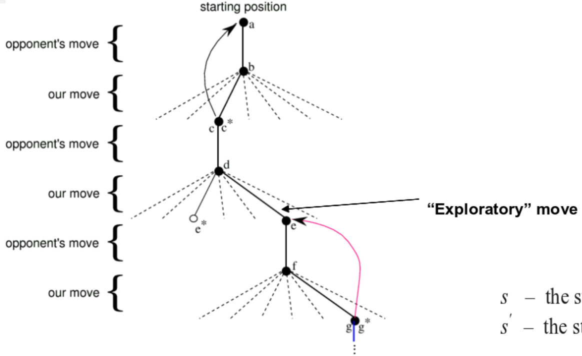
2. Now play lots of games. To pick our moves, look ahead one step:



* Just pick the next state with the highest estimated prob. of winning — the largest $V(s)$; a ***greedy*** move.

But 10% of the time pick a move at random; an ***exploratory move***.

RL-learning rule for Tic-tac-toe



s – the state before our greedy move
s' – the state after our greedy move

We increment each $V(s)$ toward $V(s')$ – a ***backup***:

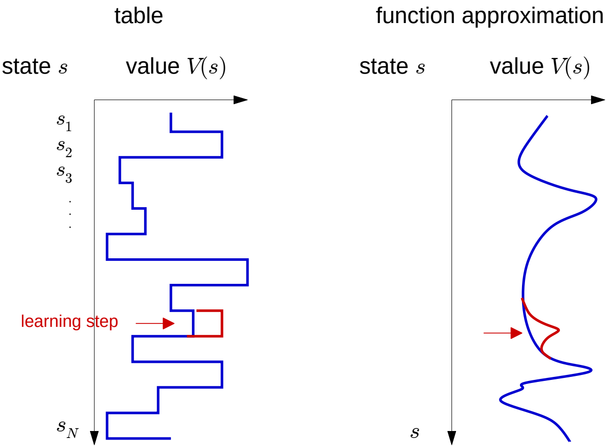
$$V(s) \leftarrow V(s) + \alpha\left[V(s') - V(s)\right]$$

a small positive fraction, e.g., $\alpha = .1$
the ***step-size parameter***

Improving the Tic-tac-toe player

- take notice of symmetries
  - in theory, much smaller state-space
  - representation / generalization
  - will it work? how can it fail?
- what can we learn from random moves?
- do we need random moves?
  - do we always need 10 %?
- can we learn offline?
  - pre-learning by playing against oneself?
  - using the learned models of the opponent?
- ...

The role of generalization



Why is Tic-tac-toe simple?

- discrete state space
- small number of states
- deterministic actions
- the agent has complete information about the game, all states are recognizable

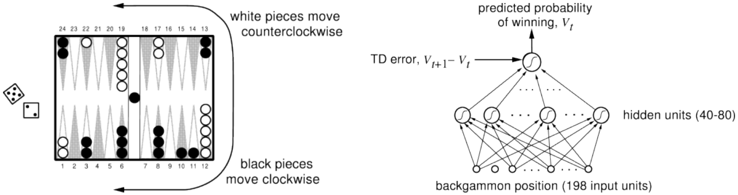
Similar approach in this lecture:

- we will look at toy examples mostly
- real applications will be (a lot) more complex
- but using the same principles

Example RL applications

- TD-Gammon: (Tesauro 1996)
  - fully know state space, but probabilistic element
  - at the time, world's best backgammon program/player
- elevator control: Crites & Barto
  - high performance "down-peak" elevator control
  - finite but very large state-space
- warehouse management: Van Roy, Bertsekas, Lee & Tsitsiklis
  - approximate the extremely large state space
  - 10–15 % improvement compared to standard industry methods
- dynamic channel assignment: Singh & Bertsekas, Nie & Haykin
  - efficient assignment of channels for mobile communication

TD-Gammon



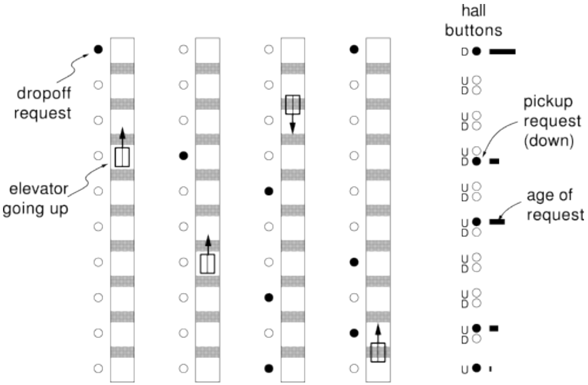
Tesauro 1992-1995:

- start with a randomly initialized network,
- play many games against yourself,
- learn a value function based on the simulated experience.

- at the time, one of the best players in the world

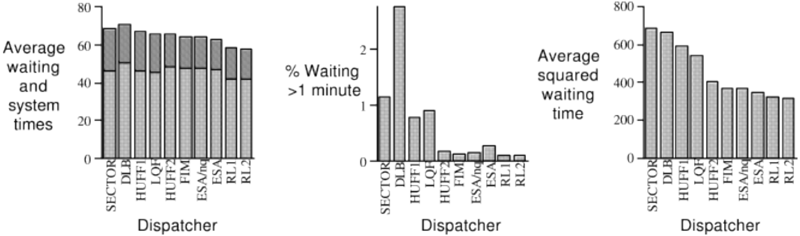
Elevator control

Crites and Barto 1996: 10 floors, 4 cabins



conservative estimation: about 10^{22} states

Elevator control performance



- RL approaches vs. state-of-the-art planning algorithms
- simple reward function: sum of waiting times

Evaluating feedback

- **evaluate** actions instead of instructing the correct action.
- pure evaluating feedback only depends on the chosen action.
  pure instructing feedback does not depend on the chosen action at all.
- supervised learning is instructive; optimization is evaluating.
- **associative** vs. **non-associative:**
  - associative inputs are mapped to outputs; learn the best output **for each** input.
  - non-associative: "learn" (find) the best output.
- *n*-armed bandit (slot machine) in the context of RL:
  - non-associative
  - evaluating feedback

The *n*-armed bandit

- choose one of n actions a repeatedly;
  each selection is called **game**.
- after each game a_t a reward r_t is obtained, where:

  $$E \langle r_t | a_t \rangle = Q^*(a_t)$$

  These are unknown **action values**.
  The distribution of r_t just depends on a_t.
- the goal is to maximize the long-term reward, e.g. over 1000 games. To solve the task of the *n*-armed bandit,

  a set of actions have to be **explored**
  and the best of them will be **exploited**.

The exploration/exploitation dilemma

- our learner estimates the value of its actions:
  $Q_t(a) \approx Q^*(a)$ **Estimation of Action Values**
- the *greedy*-action for time t is:

$$
\begin{aligned}
a_t^* &= \arg\max_a Q_t(a) \\
a_t &= a_t^* \Rightarrow \textit{exploitation} \\
a_t &\neq a_t^* \Rightarrow \textit{exploration}
\end{aligned}
$$

- you cannot explore all the time (many wasted actions)
- but also not exploit all the time (no more learning)
- exploration should never be stopped, but it may be reduced over time (when the agent has learned enough)

General action-value methods

- the name for learning methods that only consider the estimates for *action values*.
- suppose in the t-th game action a has been chosen k_a times, and the agent received *rewards* $r_1, r_2, \ldots, r_{r_a}$, then

$$Q_t(a) = \frac{r_1 + r_2 + \cdots + r_{k_a}}{k_a}$$

is the average reward.

- and in stationary environments:

$$\lim_{k_a \to \infty} Q_t(a) = Q^*(a)$$

ϵ-greedy action selection

- *greedy* action selection

$$a_t = a_t^* = \arg\max_a Q_t(a)$$

- ϵ-greedy action selection:

$$a_t = \begin{cases} a_t^* & \text{with probability} \quad 1 - \epsilon \\ & \text{random action with probability} \quad \epsilon \end{cases}$$

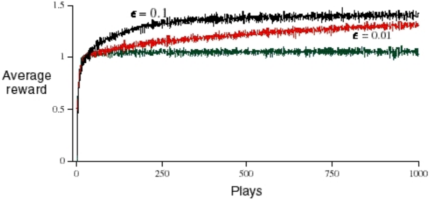
...the easiest way to combine *exploration* and *exploitation*.

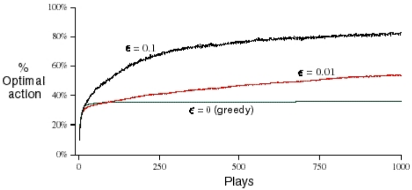
Example: 10-armed bandit

- $n = 10$ possible actions
- every $Q^*(a)$ is chosen randomly from the normal distribution: $\mathcal{N}(0, 1)$
- every r_t is also normally distributed: $\mathcal{N}(Q^*(a_t), 1)$

- play a number of games (here: 1000 games)
- repeat everything 2000 times and average the results:

ϵ-greedy method for the 10-armed bandit example





- the greedy agent is stuck very soon
- higher ϵ implies more learning, and finds good actions faster,
- lower ϵ eventually reaches higher rewards (why?)

Softmax action selection

- *softmax-action selection* method approximates action probabilities
- the most common *softmax*-method uses a Gibbs- or a Bolzmann-distribution:
  choose action a in game t with probability

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$$

  where τ is a control parameter, the *temperature*
- high τ: all actions almost equally probable
- $\tau \rightarrow 0$: only the best action has high probability

Example: binary bandit

Assume there are only **two** actions:$a_t = 1$ or $a_t = 2$ and only **two** *rewards* : $r_t = success$ or $r_t = error$

Then we could define a **goal-** or **target-action**:

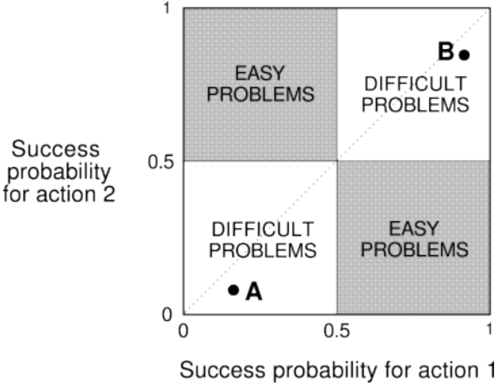
$$d_t = \begin{cases} a_t & \text{if} \quad success \\ \text{the other action} & \text{if} \quad error \end{cases}$$

and choose always the action that leads to the goal most often. This is a **supervised algorithm**.

If works well for deterministic problems...

Binary bandit task space

The space of all possible binary bandit-tasks:



Linear learning automata

Let be $\pi_t(a) = Pr\{a_1 = a\}$ the only parameter to be adapted:

L_{R-I} (Linear, reward -inaction):

on **success:** $\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha(1 - \pi_t(a_t)) \quad 0 < \alpha < 1$

on **failure:** no change

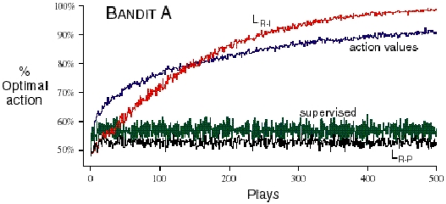
L_{R-P} (Linear, reward -penalty):

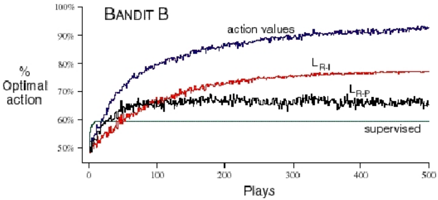
on **success:** $\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha(1 - \pi_t(a_t)) \quad 0 < \alpha < 1$

on **failure:** $\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha(0 - \pi_t(a_t)) \quad 0 < \alpha < 1$

- after each update the other probabilities get updated in a way that the sum of all probabilities is 1.

Performance of the binary bandit-tasks A and B





Incremental calculation of the average reward

Remember the definition of the *average rewards*:

The average of the k first *ewards* is (neglecting the dependency on a):

$$Q_k = \frac{r_1 + r_2 + \cdots + r_k}{k}$$

problem: we need to keep all previously received rewards...

The *running average* trick is more memory efficient:

$$Q_{k+1} = Q_k + \frac{1}{k+1}\left[r_{k+1} - Q_k\right]$$

Note: this is a common form for *update*-rules:

NewEstimation = *OldEstimation* + *Stepsize* · *[Value - OldEstimation]*

Non-stationary problems

Using Q_k as the average reward is adequate for a stationary problem, i.e. if none of the $Q^*(a)$ changes over time.

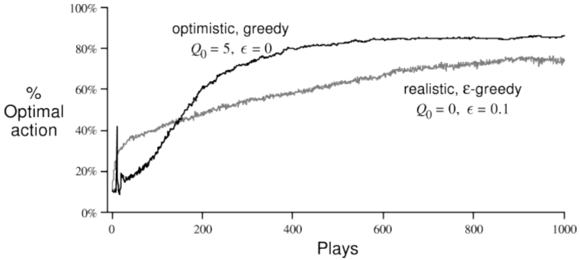
But in the case of a non-stationary problem, this is better:

$$\begin{aligned} Q_{k+1} = & \quad Q_k + \alpha \left[r_{k+1} - Q_k \right] \quad \textit{for constant } \alpha, 0 < \alpha \leq 1 \\ = & \quad (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha (1-\alpha)^{k-i} r_i \end{aligned}$$

the exponential, recency-weighted average

Optimistic initial values

- all previous methods depend on $Q_0(a)$, i.e., they are *biased*.
- initialize the action-values **optimistically**, e.g. for the 10-armed testing environment: $Q_0(a) = 5$ for all a
- this enforces exploration during the first few iterations (until the values have stabilized):



Reinforcement-comparison

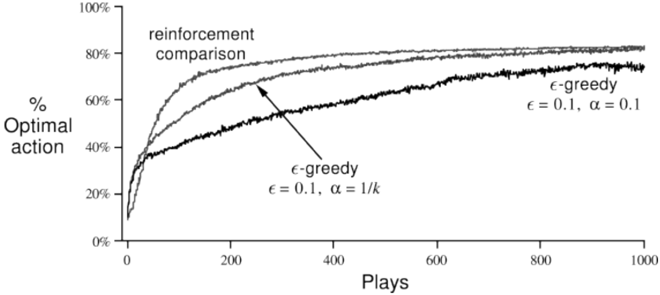
- compare rewards with a known *reference-reward* $\bar{r}_t$, e.g. the average of all possible rewards
- strengthen or weaken the chosen action depending on $r_t - \bar{r}_t$.

- let $p_t(a)$ be the **preference** for action a.
- The preferences determine the action-probabilities, e.g. by a Gibbs-distribution:

$$\pi_t(a) = Pr\{a_t = a\} = \frac{e^{p_t(a)}}{\sum_{b=1}^{n} e^{p_t(b)}}$$

- then: $p_{t+1}(a_t) = p_t(a) + \beta \left[r_t - \bar{r}_t \right]$ *and* $\bar{r}_{t+1} = \bar{r}_t + \alpha \left[r_t - \bar{r}_t \right]$

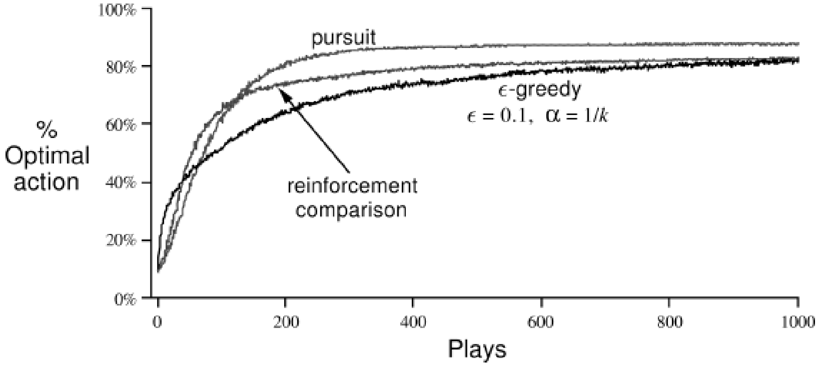
Reinforcement-comparison example



Pursuit methods

- incorporate both estimations of action values as well as action preferences.
- "Pursue" always the *greedy*-action, i.e. make the *greedy*-action more probable in the action selection.
- Update the action values after the t-th game to obtain Q_{t+1}.
- The new greedy-action is $a^*_{t+1} = \arg \underset{a}{max}\, Q_{t+1}(a)$
- Then: $\pi_{t+1}(a^*_{t+1}) = \pi_t(a^*_{t+1}) + \beta \left[1 - \pi_t(a^*_{t+1})\right]$
  and the probabilities of the other actions are reduced to keep their sum 1.

Performance of a Pursuit-Method



Summary

- a class of problems in-between supervised and un-supervised learning
- agent takes actions, receivces rewards
- goal is to maximize accumulated reward over time

- n-armed bandit problems illustrate action-selection
- so far, independent of states

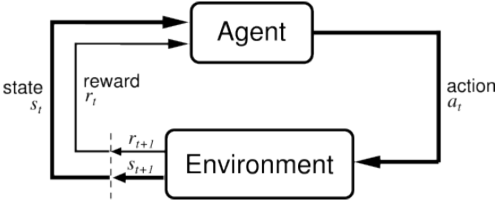
- exploitation-exploration dilemma
- ϵ-greedy and softmax action selection
- comparison of RL approach with supervised learning

The Reinforcement-Learning problem

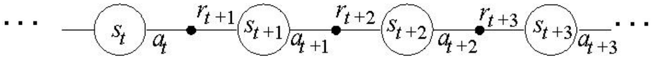
formalization of the RL problem: Markov Decision Process (MDP)

- an idealized and very general form of the RL problem with precise mathematical definition and theory
- interaction between agent and environment
- *state*- and *action*-spaces
- state transitions and rewards
- goal is to maximize the return: accumulated reward
- Markov assumption: behaviour only depends on current state, not on history
- idea of value-functions and relation to policies
- Bellman equation

The learning agent in an environment



agent and environment interact at discrete times:	$t = 0,1,2 \ldots K$
agent observed state at the time t:	$s_t \in S$
executes action at the time t:	$a_t \in A(s_t)$
obtains *reward*:	$r_{t+1} \in \mathcal{R}$
and the following state:	s_{t+1}



The agent learns a *policy*

policy at time t, π_t :

mapping of states to action-probabilities
$\pi_t(s, a) =$ probability, that $a_t = a$ if $s_t = s$

- Reinforcement learning methods describe how an agent updates its *policy* as a result of its experience.
- The overall goal of the agent is to maximize the long-term sum of *rewards*.

Modeling approach and abstraction

- time steps do not need to be fixed intervals of real time.
- actions can be *low-level* (e.g., voltage of motors), or *high-level* (e.g., take a job offer), "mental" (z.B., shift in focus of attention), etc.
- states can be *low-level* "perception", abstract, symbolic, memory-based, or subjective (e.g. the state of being surprised).
- the environment is not necessarily unknown to the agent, but it is incompletely controllable.
- the *reward*-calculation is done in the environment, and outside of control of the agent.

Goals and rewards

- Is a scalar *reward* signal an adequate description for a goal?
  – perhaps not, but it is surprisingly flexible.
- A goal should describe **what** we want to achieve and not **how** we want to achieve it.
- A goal must be beyond the control of the agent – therefore outside the agent itself.
- The agent needs to be able to measure success:
  - explicit;
  - frequently during its lifetime.

Accumulated rewards or *return*

the sequence of rewards after time t is:

$$r_{t+1}, r_{t+2}, r_{t+3}, \ldots$$

What do we want to maximize?

In general, we want to maximize the **expected *return***, $E\{R_t\}$ at each time step t.

Episodic task : Interaction splits in episodes,
e.g. a game round,
passes through a labyrinth

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

where T is a final time where a final state is reached and the episode ends.

Return for continuous tasks

- **continuous tasks**: no final/terminal state
  - the interaction has no episodes
  - naive sum of all rewards may diverge

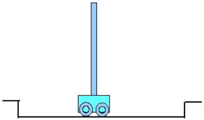
- **discounted return**:

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

where $\gamma, 0 \leq \gamma \leq 1$, is the *discount rate*.

- „nearsighted“ $0 \leftarrow \gamma \rightarrow 1$ „farsighted“

Example: pole balancing



Avoid **Failure**: the pole turns over a critical angle or the waggon reaches the end of the track

As an **episodic task** where episodes end on failure:

Reward	=	+1 for every step before failure
⇒ *Return*	=	number of steps to failure

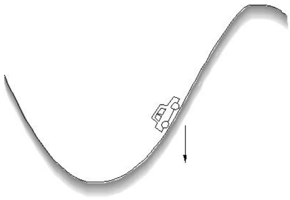
As **continuous task** with *discounted Return*:

Reward	=	−1 on failure; 0 otherwise
⇒ *Return*	=	$-\gamma^k$, for k steps before failure

In both cases, the return is maximized by avoiding failure as long as possible.

Example: mountain car

Drive as fast as possible to the top of the mountain.

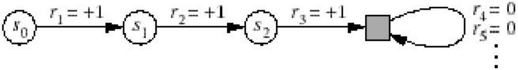


$$\begin{aligned} \textit{Reward} &= -1 \text{ for each step where the top of the mountain is } \textbf{not} \text{ reached} \\ \textit{Return} &= -\text{number of steps before reaching the top of the mountain.} \end{aligned}$$

The *return* is maximized by minimizing the number of steps to reach the top of the mountain.

Unified notation

- In episodic tasks, we number the time steps of each episode starting with zero.
- In general, we do not differentiate between episodes. We write $s(t)$ instead of $s(t,j)$ for the state at time t in episode j.
- Consider the end of each episode as an absorbing state that always returns a **reward** of 0:



- We summarize all cases:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where γ can only be 1 if an absorbing state is reached.

Markov assumption

- the state s_t at time t includes all information that the agent has (and needs) about its environment.
- the state can include instant perceptions, processed perceptions and structures or features that are built on a sequence of perceptions.
- but the behaviour of the environment does *not* depend on the history of the agent-environment interaction. The current state contains all "relevant" information, this is equivalent to the *Markov property*:

$$Pr\left\{s_{t+1} = s', r_{t+1} = r \middle| s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right\} = \\ Pr\left\{s_{t+1} = s', r_{t+1} = r \middle| s_t, a_t\right\}$$

For all s', r,and *histories* $s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0$.

Markov decision processes

- if the Markov proporty holds for a given RL-task, it is called a Markov Decision Process (MDP)
- if state and action spaces are finite, it is a finite MDP.
- to define a finite MDP, we need:
  - **state and action spaces**
  - environment "dynamics" defined by the **transition probabilities:**

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

  - **reward probabilities**:

$$R^a_{ss'} = E\left\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right\} \forall s, s' \in S, a \in A(s).$$

Markov decision process

MDP: a five-tuple (S, A, P, R, γ), where

- S is a set of states s,
- A is a set of actions, where $A(s)$ is the finite set of actions available in state s,
- $P^a_{s,s'}$ is the probability that action a in state s at time t will lead to state s' at time $t+1$,
- $R^a_{s,s'}$ is the immediate reward received after transition from state s to state s' at time t,
- the transition and reward probabilities only depend on the current state s, but not on the history of the system,
- $\gamma \in [0, 1]$ is the discount factor used for calculating the return.

- most basic algorithms assume that the sets S and A are finite.

Recycling-robot: toy example for a finite MDP

Consider a robot designed to collect empty cans:

- reward = number of collected cans.
- at each time step the robot decides, whether it
  1. actively searches for cans,
  2. waits for someone bringing a can, or,
  3. drives to the basis for recharge.
- searching is better, but uses battery; if the batteries runs empty during searching, the robot needs to be recovered (bad).
- decisions are made based on the current battery level: $\{\mathit{high}, \mathit{low}\}$.

Recycling-robot MDP

state space: $S = \{high, low\}$

action space depends on the states:

$A(high) = \{search, wait\}$,
$A(low) = \{search, wait, recharge\}$

rewards depends on the actions:

$R^{\mathtt{search}}$ = expected number of cans during search,
$R^{\mathtt{wait}}$ = expected number of cans during wait,
assuming $R^{\mathtt{search}} > R^{\mathtt{wait}}$

dynamics $P^a_{ss'}$ depends on two parameters $\{\alpha, \beta\}$:

α: probability of the battery keeping high value
β: probability of the battery keeping low value

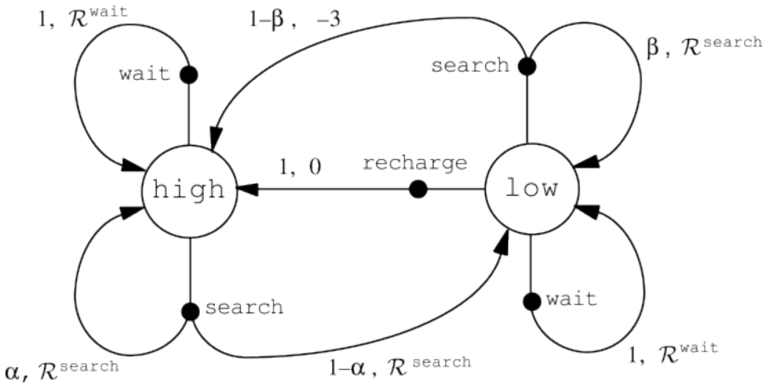
Recycling-robot transition table

Table 3.1 Transition probabilities and expected rewards for the finite MDP of the recycling robot example.

s	s'	a	$\mathcal{P}^a_{ss'}$	$\mathcal{R}^a_{ss'}$
high	high	search	α	$\mathcal{R}^{\mathtt{search}}$
high	low	search	$1-\alpha$	$\mathcal{R}^{\mathtt{search}}$
low	high	search	$1-\beta$	-3
low	low	search	β	$\mathcal{R}^{\mathtt{search}}$
high	high	wait	1	$\mathcal{R}^{\mathtt{wait}}$
high	low	wait	0	$\mathcal{R}^{\mathtt{wait}}$
low	high	wait	0	$\mathcal{R}^{\mathtt{wait}}$
low	low	wait	1	$\mathcal{R}^{\mathtt{wait}}$
low	high	recharge	1	0
low	low	recharge	0	0

Note: There is a row for each possible combination of current state, s, next state, s', and action possible in the current state, $a \in \mathcal{A}(s)$.

Recycling-robot transition graph



α, β: probability of battery keeping its level during searching

e.g., *low-search-high* implies running out of battery, reward -3 because then the operator needs to recover and recharge the robot.

Value Function

- the **value of a state** is the expected *return* beginning with this state; depends on the *policy* of the agent:

  state-value-function for policy π:

$$V^\pi(s) = E_\pi\left\{R_t | s_t = s\right\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\}$$

- the **action value** of an action in a state under a *policy* π is the expected *return* beginning with this state, if this action is chosen and π is pursued afterwards.

  action-value-function for policy π:

$$Q^\pi(s,a) = E_\pi\left\{R_t | s_t = s, a_t = a\right\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\}$$

The Bellman-Equation for policy π

Basic Idea:

$$\begin{aligned} R_t = & \quad r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots \\ = & \quad r_{t+1} + \gamma \left(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \dots \right) \\ = & \quad r_{t+1} + \gamma R_{t+1} \end{aligned}$$

Thus:

$$\begin{aligned} V^\pi(s) = & \quad E_\pi \left\{ R_t | s_t = s \right\} \\ = & \quad E_\pi \left\{ r_{t+1} + \gamma V(s_{t+1}) | s_t = s \right\} \end{aligned}$$

Or, without expectation operator:

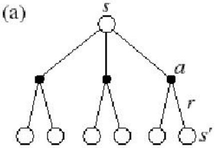
$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right]$$

More about the Bellman-Equation

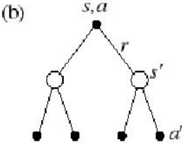
$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

These are a set of (linear) equations, one for each state. The value-function for π is an unique solution.

Backup-Diagrams :



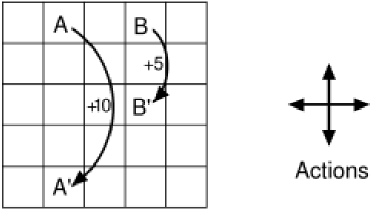
for V^{π}



for Q^{π}

Example: Gridworld I

- actions: up, down, right, left; deterministic.
- if the agent would leave the grid: no motion, but *reward* $= -1$.
- other actions *reward* $= 0$, except actions that move the agent out of state A or B (reward 10 or 5).



(a)

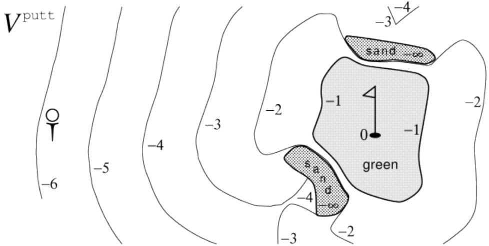
3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

(b)

state-value-function for the uniform random policy; $\gamma = 0.9$

Example: Golf

- state is the position of the ball
- reward is -1 for each swing until the ball is in the hole
- two actions: `putt` (use putter) `driver` (use driver)
- `putt` on the "green" area is always successful (hole)
- sketch of the state value function $V(s)$:



Optimal Value Function

- For finite MDPs, the *policies* can be **partially ordered**

$$\pi \geq \pi' \quad \textit{if} \quad V^{\pi}(s) \geq V^{\pi'}(s)\ \forall s \in S$$

- There is always at least one (maybe more) *policies* that are better than or equal all others. This is an **optimal *policy***. We call it π^*.
- Optimal *policies* share the same ‚**optimal state-value-function**:

$$V^*(s) = \max_{\pi} V^{\pi}(s)\ \forall s \in S$$

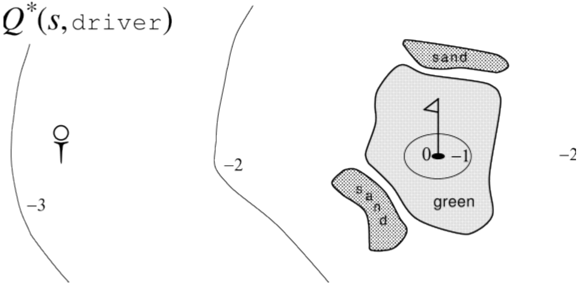
- Optimal *policies* also share the same ‚**optimal action-value-function**:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)\ \forall s \in S \textit{ and } a \in A(s)$$

This is the expected *return* after choosing action *a* in state *s* an continuing to pursue an optimal *policy*.

Example: Golf

- we can strike the ball further with the `driver` than with the `putter`, but with less accuracy.
- $Q^*(s, \texttt{driver})$ gives the values for the choice of the `driver` at the given start position, and afterwards always the best action is chosen.

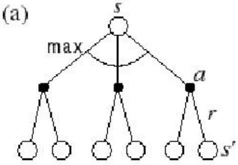


Optimal Bellman-Equation for $V^*(s)$

The value of a state under an optimal *policy* is equal to the expected *returns* for choosing the best actions from now on.

$$\begin{aligned} V^*(s) &= \max_{a \in A(s)} Q^{\pi^*}(s,a) \\ &= \max_{a \in A(s)} E\left\{ r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a \right\} \\ &= \max_{a \in A(s)} \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s') \right] \end{aligned}$$

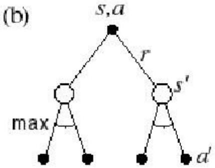
V^* is the unique solution of this system of nonlinear equations.
The corresponding backup diagram:



Optimal Bellman-Equation for Q^*

$$
\begin{aligned}
Q^*(s,a) &= E\left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a \right\} \\
&= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a') \right]
\end{aligned}
$$

The backup diagram:

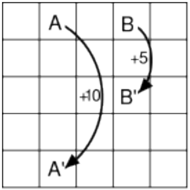


Q^* is the unique solution of this system of nonlinear equations.

Why optimal state-value functions are useful

A *policy* that is *greedy* with respect to V^* is an optimal *policy*!

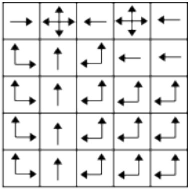
Therefore, given V^*, the (one-step-ahead)-search produces optimal action sequences. In the gridworld example:



a) gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

b) V^*



c) π^*

What about Optimal Action-Values Functions?

Given Q^*, the agent does not need to perform the *one-step-ahead*-search:

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

Solving the optimal Bellman-Equation

- to determine an optimal policy π^* by solving the optimal Bellman-equation we need the following:
  - knowledge of the dynamics of the environment ($P^a_{ss'}$),
  - enough storage space and computation time,
  - the Markov property must hold.
- how much space and time do we need?
  - polynomially with the number of states (with *dynamic programming*, see below)
  - BUT, usually the number of states is very large (e.g., backgammon has about 10^{20} states).
- we usually have to resort to approximations.
- many RL methods can be understood as an approximate solution to the optimal Bellman equation.

Summary

- agent-environment interaction
  - states
  - actions
  - rewards
- policy: stochastic action selection rule
- return: the function of the rewards that the agent tries to maximize
- episodic and continuing tasks
- Markov assumption (Markov property)
- MDP or Markov decision process
  - transition probabilities
  - expected rewards

Summary (cont.)

- **Value functions**
  - state-value function for a *policy*
  - action-value function for a *policy*
  - optimal state-value function
  - optimal action-value function
- optimal *policies*
- Bellman-equation
- the need for approximation