Reinforcement Learning

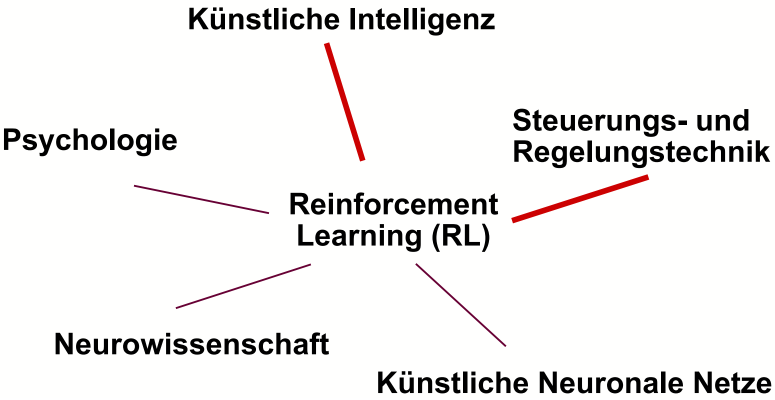
Algorithmic Learning 64-360, Part 13

Jianwei Zhang

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

06/07/2011

Introduction

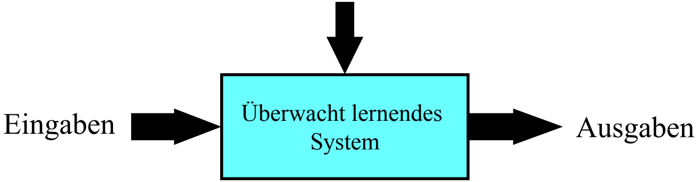


What is Reinforcement Learning?

- learning from interaction
- goal-oriented learning
- learning **by/from/during** interaction with an external environment
- learning "what to do" — how to map situations to actions — to maximize a numeric reward signal

Supervised Learning

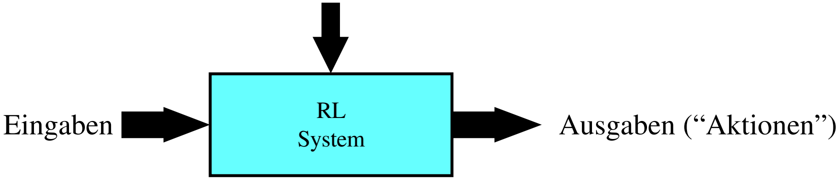
training data = desired (target) output



error = (target output – actual system output)

Reinforcement Learning

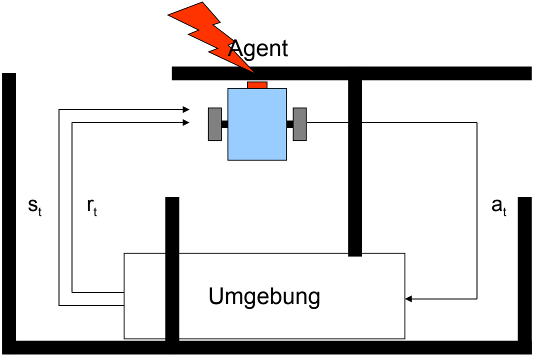
training information = evaluation ("rewards" / "penalties")



Goal: achieve as much *reward* as possible

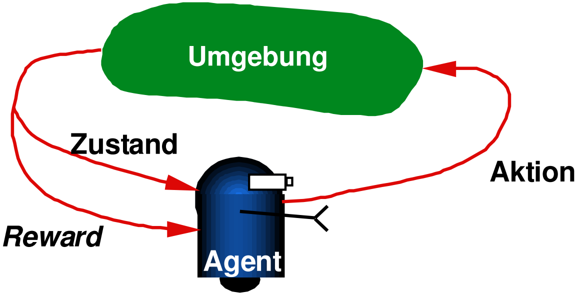
Reinforcement Learning

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

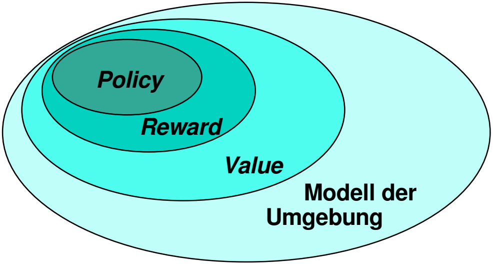


The complete agent

- chronologically situated
- constant learning and planning
- affects the environment
- environment is stochastic and uncertain

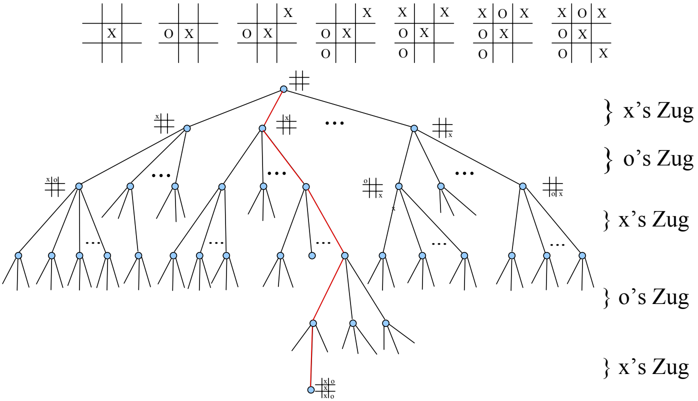


Elements of RL



- **policy:** what to do
- **reward:** what is good
- **value:** what is good because of expected reward
- **model:** what follows what

An Extended Example: Tic-tac-toe



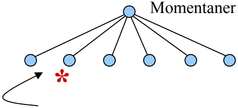
Requires an imperfect opponent: he / she makes mistakes

An RL-Approach

1. Erstelle eine Tabelle mit einem Eintrag pro Zustand:

Zustand	$V(s)$ – geschätzte Wahrscheinlichkeit für den Gewinn	
	.5	
	.5	
⋮	⋮	
	1	gewonnen
⋮	⋮	
	0	verloren
⋮	⋮	
	0	unentschieden

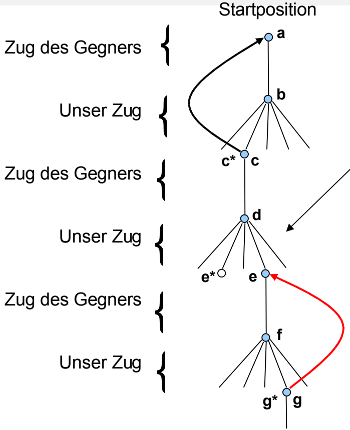
2. Jetzt spiele viele Spiele.
   Um einen Zug zu wählen,
   schaue einen Schritt nach vorne:



Nehme den nächsten Zustand mit der höchsten geschätzten Gewinnwahrscheinlichkeit — das höchste $V(s)$; ein ***greedy*** Zug.

Aber in 10% aller Fälle wähle einen zufälligen Zug; ein ***explorierender*** Zug.

RL-Learning Rule for Tic-tac-toe



Explorierender Zug

s – Zustand vor dem *greedy* Zug

s' – Zustand nach dem *greedy* Zug

Wir inkrementieren jedes $V(s)$ zu $V(s')$ – ein „***backup***"

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

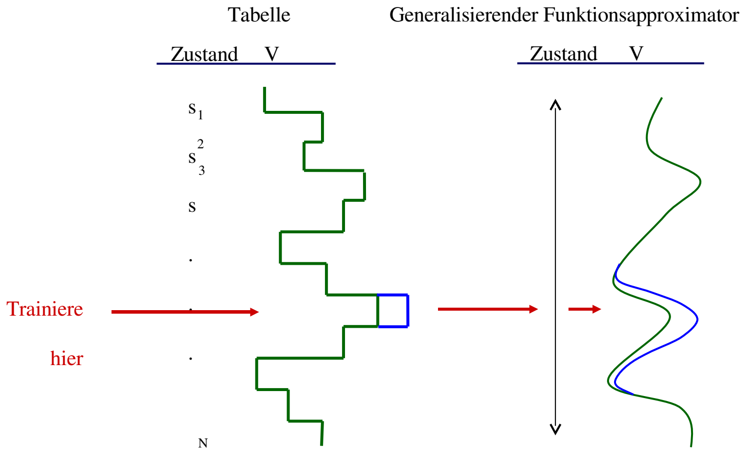
kleiner positiver Wert, z.B. $\alpha = 0.1$

der **„Schrittweitenparameter"**

Improving the Tic-tac-toe Player

- take notice of symmetries
  - representation / generalization
  - How can it fail?
- Do we need random moves"? Why?
  - Do we always need 10 %?
- Can we learn from random moves"?
- Can we learn offline?
  - Pre-learning by playing against oneself?
  - Using the learned models of the opponent?
- ...

e.g. Generalization



Why is Tic-tac-toe Simple?

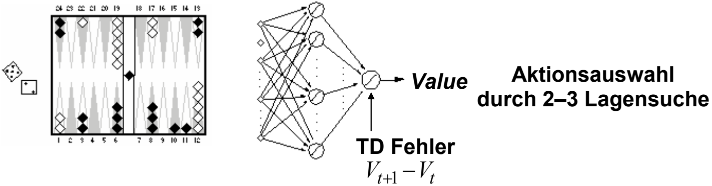
- finite, small number of states,
- deterministic (one-step look ahead)
- all states are recognizable
- ...

Some Important RL Applications

- **TD-Gammon:** Tesauro
  - world's best backgammon program
- **Elevator control:** Crites & Barto
  - High Performance "down-peak" elevator control
- **Warehouse management:** Van Roy, Bertsekas, Lee & Tsitsiklis
  - 10–15 % improvement compared to standard industry methods
- **Dynamic Channel Assignment:** Singh & Bertsekas, Nie & Haykin
  - high performance assignment of channels for mobile communication

TD-Gammon

Tesauro, 1992–1995



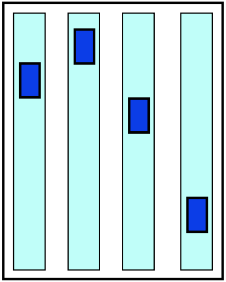
- Start with a randomly initialized network.
- Play many games against yourself.
- Learn a value function based on the simulated experience.

This probably makes the best players in the world.

Elevator Control

Crites and Barto, 1996,

10 floors, 4 cabins



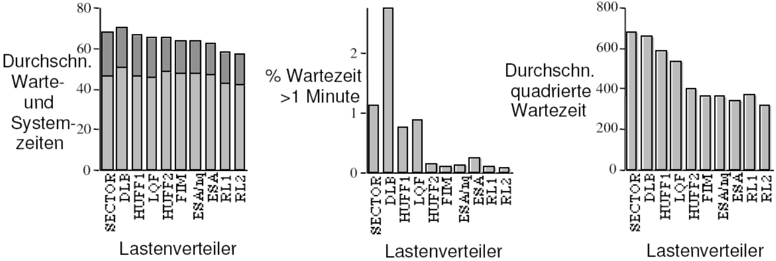
Zustände: Knopfzustände; Positionen, Richtungen, und Bewegungszustände der Kabinen; Personen in Kabinen & in Etagen

Aktionen: halte an X, oder fahre nach Y, nächste Etage

Rewards: geschätzt, –1 pro Zeitschritt für jede wartende Person

Conservative estimation: about 10^{22} states

Performance Comparison



RL Timeline

Trial-and-Error learning

Thorndike (Ψ) 1911

Minsky

Klopf

Barto et al.

Temporal-difference learning

Secondary reinforcement (Ψ)

Samuel

Holland

Witten

Sutton

Optimal control, value functions

Hamilton (Physics) 1800s

Shannon

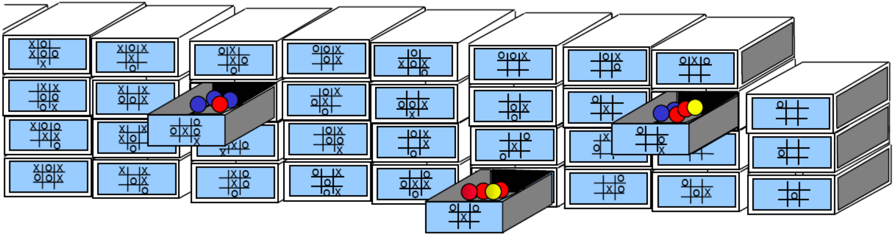
Bellman/Howard (OR)

Werbos

Watkins

MENACE (Michie 1961)

"Matchbox Educable Noughts and Crosses Engine"



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) (at least our view of it):
  - Non-Associative
  - Evaluating feedback

The *n*-Armed Bandit

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

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

  These are unknown **action values**.
  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 Problem

- Suppose values are estimated:
  $Q_t(a) \approx Q^*(a)$ **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, but also not exploit all the time
- Exploration should never be stopped, but it should be reduced

Action – *Value* Method

- Methods, that only consider the estimates for *action values*
  Suppose in the t-th game action a has been chosen k_a times, that produce the *rewards* $r_1, r_2, \ldots, r_a$, then

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

  "average *reward*"

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

ϵ-greedy Action Selection

- *greedy* Action selection

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

- ϵ-greedy Action selection:

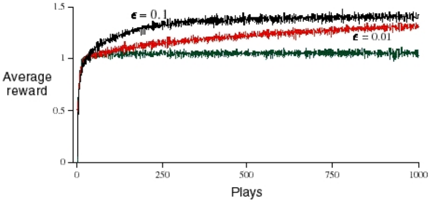
$$a_t = \begin{cases} a_t^* & \text{with probability} \quad 1-\epsilon \\ & \text{random action with probability} \quad \epsilon \end{cases}$$

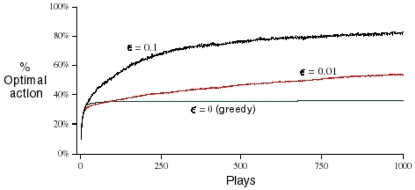
...the easiest way to handle *exploration* and *exploitation*.

10-armed Testing Environment

- $n = 10$ possible actions
- Every $Q^*(a)$ is chosen randomly from the normal distribution: $\eta(0, 1)$
- Every r_t is also normally distributed: $\eta(Q^*(a_t), 1)$
- 1000 games
- Repeat everything 2000 times and average the results.

ϵ-greedy Method for the 10-armed Testing Environment





Softmax Action selection

- *Softmax*-action selection method defines action probabilities with approximated values
- The most usual *softmax*-method uses a Gibbs- or a Bolzmann-distribution:
  Chose 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 the "temperature".

Binary Bandit-Task

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**:

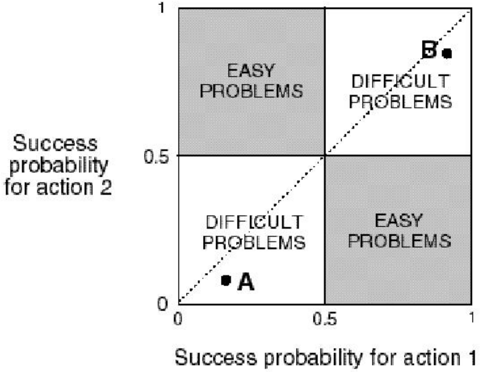
$$d_t = \begin{cases} a_t & \text{if} \quad success \\ \text{The other Action} & \text{if} \quad error \end{cases}$$

and choose always the action, that lead to the goal most often.

This is a **supervised algorithm**.
If works well for deterministic problems...

Random 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

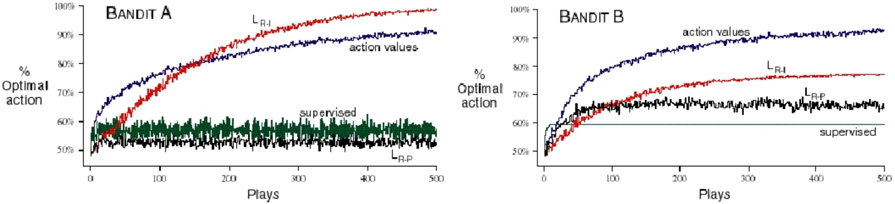
L_{R-P} (Linear, reward -penalty):

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 Implementation

Remember the evaluation-method for the average *rewards*:

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

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

can this be built incrementally (without saving all *rewards*)?

We could use the running average:

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

This is a common form for *update*-rules:

NewEstimation = *OldEstimation* + *Stepwidth [Value - OldEstimation]*

Non-Stationary Problems

Using Q_k as the average *reward* is adequate for a stationary problem, i.e. if no $Q^*(a)$ changes with time.

But not for a non-stationary problem.

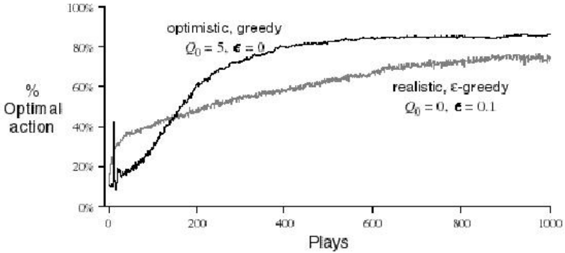
Better in case of a non-stationary problem is:

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

exponential, recency-weighted average

Optimistic Initial Values

- All previous methods depend on $Q_0(a)$, i.e., they are ***biased***.
- Given that we initialize the action-values **optimistically**, e.g. for the 10-armed testing environment: $Q_0(a) = 5$ for all a



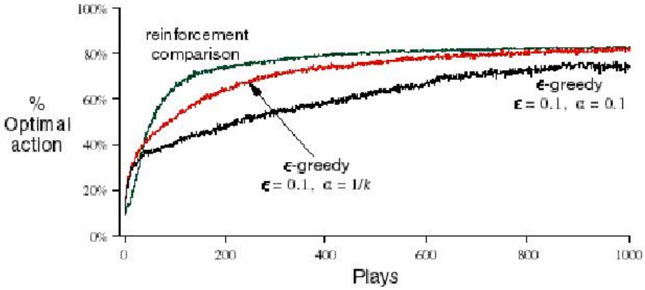
Reinforcement-Comparison

- Compare rewards with a 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.
- Preference 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* $\bar{r}_{t+1} = \bar{r}_t + \alpha\left[r_t - \bar{r}_t\right]$

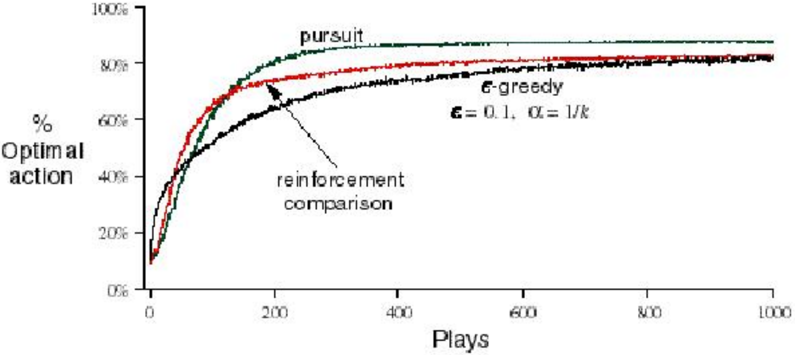
Performance of Reinforcement-Comparison-Methods



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 $a^*_{t+1} = \arg \max_a Q_{t+1}(a)$
- 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



Conclusions

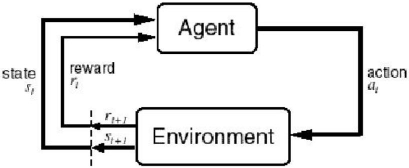
- These are all quite simple methods,
  - but they are complex enough - that we can build on them
  - Ideas for improvements:
  - estimation of uncertainties ... Interval estimation
  - approximation of *Bayes optimal solutions*
  - Gittens indices (classical solution for n-armed bandits for controlling *exploration* and *exploitation*)
- The complete RL problem has some approaches for a solution....

The Reinforcement-Learning Problem

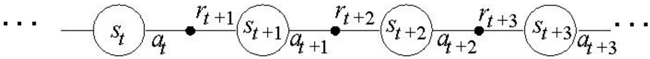
Description of the RL-Problem:

- Presentation of an idealized form of the RL problem which can be described theoretically.
- Introduction of the most important mathematical components: value-functions and Bellman-equation.
- Description of the trade-off between applicability and mathematical linguistic.

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*.

Degree of Abstraction

- Time steps do not need to be fixed intervals of real time.
- Actions can be *lowlevel* (e.g., Voltage of motors), or *highlevel* (e.g., take a job offer), "mental" (z.B., shift in focus of attention), etc.
- States can be *lowlevel* "perception", abstract, symbolic, memory-based, or subjective (e.g. the state of being surprised).
- An RL-agent is not comparable to a whole animal or robot, because the consist of multiple agents and other parts.
- The environment is not necessarily unknown to the agent, it is incompletely controllable.
- The *reward*-calculation is done in the environment, that the agent cannot modify arbitrarily.

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.

Returns

A 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.

Returns for Continuous Tasks

continuous tasks: Interaction has no episodes.

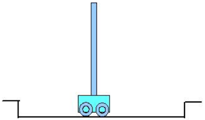
discounted *return* :

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

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

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

An example



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
$\Rightarrow$ *Return*	$=$	number of steps to failure

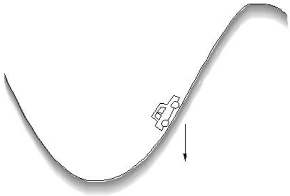
As **continuous task** with *discounted Return*:

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

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

A further example

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

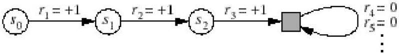


$$
\begin{aligned}
\textit{Reward} &= -1 \;\; \text{for each step where the top of the mountain is } \textbf{not} \text{ reached} \\
\textit{Return} &= -\text{number of steps before reaching the top of the mountain.}
\end{aligned}
$$

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.

The Markov Probability

- The "state" at time t includes all information that the agent has about its environment.
- The state can include instant perceptions, processed perceptions and structures, that are built on a sequence of perceptions.
- Ideally the state should conclude previous perceptions, to contain all "relevant" information; this means it should provide the **Markov Probability**:

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

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 a RL-task provides a Markov Probability, it is mainly a Markov decision process.
- If state and action spaces are finite, it is a finite MDP.
- To define a finite MDP, we need:
  - **state and action spaces**
  - one-step-"dynamic"defined by the **transition probabilities:**

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

  - ***reward* probabilities**:

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

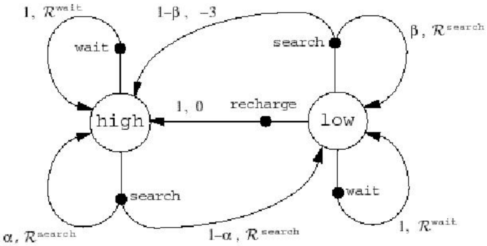
An example for a finite MDP

recycling-robot

- In each step the robot decides, whether it (1) actively searches for cans, (2) waiting for someone bringing a can, or (3) drives to the basis for recharge.
- Searching is better, but uses battery; if the batteries run empty during searching, it needs to be recovered (bad).
- Decisions are made based on the current battery level: `high`, `low`
- *reward* = number of collected cans.

Recycling-Robot MDP

$S = \{\texttt{high},\ \texttt{low}\}$
$A\,(\texttt{high}) = \{\texttt{search},\ \texttt{wait}\}$
$A\,(\texttt{low}) = \{\texttt{search},\ \texttt{wait},\ \texttt{recharge}\}$
$R^{\texttt{search}}$ = expected number of cans during search
$R^{\texttt{wait}}$ = expected number of cans during wait
$R^{\texttt{search}} > R^{\texttt{wait}}$



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 ***Policy*** $\boldsymbol{\pi}$:

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

- The **action value** of an action in a state under a ***policy*** $\boldsymbol{\pi}$ is the expected *return* beginning with this state, if this action is chosen and π is pursued afterwards. **Action Value for** ***Policy*** $\boldsymbol{\pi}$:

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

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:

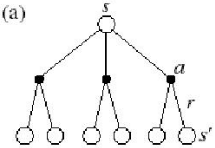
$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right]$$

More about the Bellman-Equation

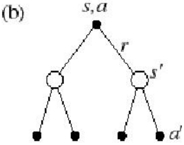
$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

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

Backup-Diagrams :



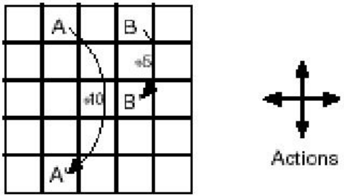
for V^{π}



for Q^{π}

Gridworld

- Actions: up , down , right , left ; deterministic.
- If the agent would leave the grid: no turn, but *reward* $= -1$.
- Other actions *reward* $= 0$, except actions that move the agent out of state A or B.

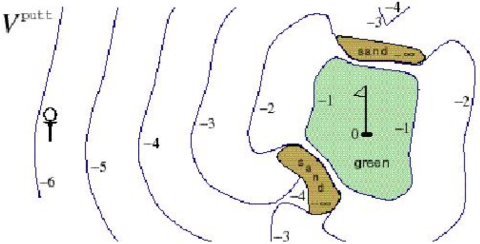


3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

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

Golf

- State is the position of the ball
- Reward is -1 for each swing until the ball is in the hole
- Value of a State?
- Actions: `putt` (use putter) `driver` (use driver)
- `putt` on the "green" area always successful (hole)



Optimal Value Function

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

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

- 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**:

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

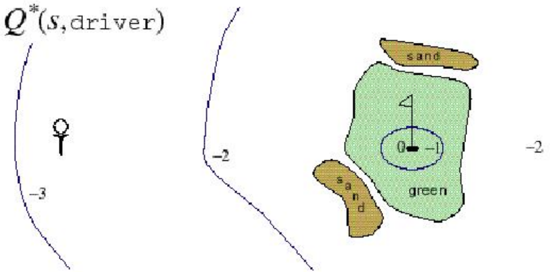
- Optimal *policies* also share the same ,**optimal action-value-function**:

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

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

Optimal Value-Function for 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`, if always the best action is chosen.

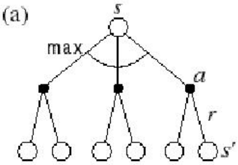


Optimal Bellman-Equation for V^*

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

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

The backup diagram:

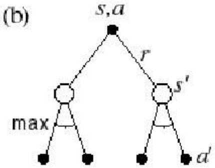


V^* is the unique solution of this system of nonlinear equations.

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:

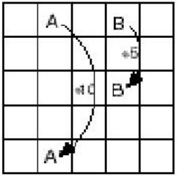


Q^* is the unique solution of this system of nonlinear equations.

Why Optimal State-Value Functions are Useful

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

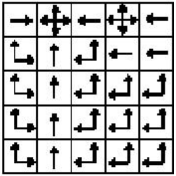
Therefore,given V^*, the (it one-step-ahead)-search produces optimal actions in the long time. e.g., in the gridworld:



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 be able to determine an optimal policy *policy* by solving the optimal Bellman-equation we need the following:
  - exact knowledge of the dynamics of the environment;
  - enough storage space and computation time;
  - the Markov probability
- How much space and time do we need?
  - polynomially with the number of states (with *dynamic programming*, later lecture)
  - 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 probability
- 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