Machine Learning I
Reinforcement Learning

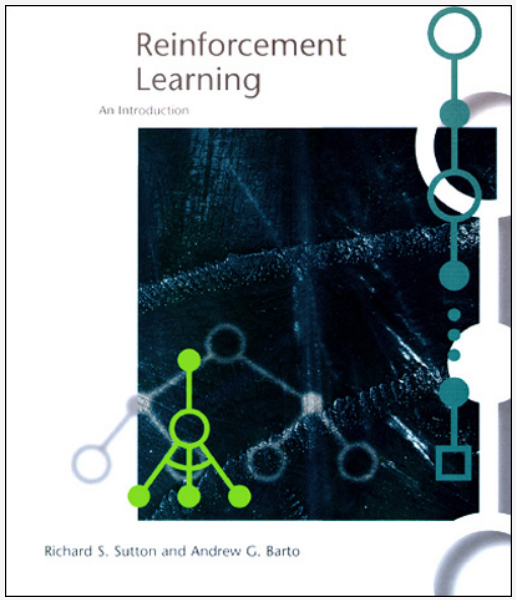
Thomas Rückstieß

Technische Universität München

December 17/18, 2009

Literature

- Book: Reinforcement Learning: An Introduction
  Sutton & Barto (free online version: google "sutton barto book")



- Survey Paper Reinforcement learning: a survey
  Kaelbling, Littman & Moore

What is Reinforcement Learning?

What is it not?

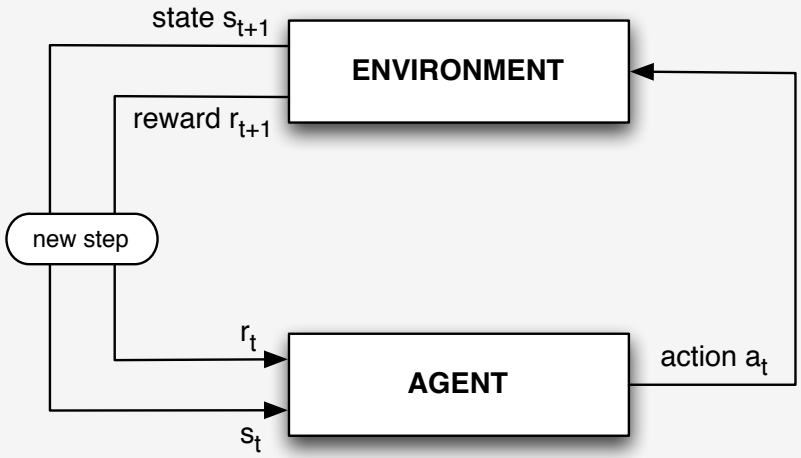
Supervised Learning For a given input, the learner is also provided with the desired output (target). Learn mapping from input to output.

Unsupervised Learning The goal is to build a model of the inputs (e.g. for clustering, outlier detection, compression, ...) without any feedback.

Reinforcement Learning

- is more interactive
- does not receive the target
- does receive evaluative feedback (rewards)
- is more a problem description than a single method

Problem Statement



Definition (agent, environment, state, action, reward)

An agent interacts with an environment at discrete time steps $t = 0, 1, 2, \ldots$ At each time step t, the agent receives state $s_t \in \mathcal{S}$ from the environment. It then chooses to execute action $a_t \in \mathcal{A}(s_t)$ where $\mathcal{A}(s_t)$ is the set of available actions in s_t. At the next time step, it receives the immediate reward $r_{t+1} \in \mathbb{R}$ and finds itself in state s_{t+1}.

Problem Statement (comments)

This is a very general definition:

environment does not have to be deterministic: transitions from s to s' with action a can be probabilistic, described by $\mathcal{P}^a_{ss'}$.

rewards only have the condition of being scalar values. They can be given sparsely (reward only at goal) or after each timestep. They can be probabilistic and the expected reward when transitioning from s to s' with action a is described by $\mathcal{R}^a_{ss'}$.

time steps don't have to refer to fixed intervals of real time, they can just be successive stages of decision-making.

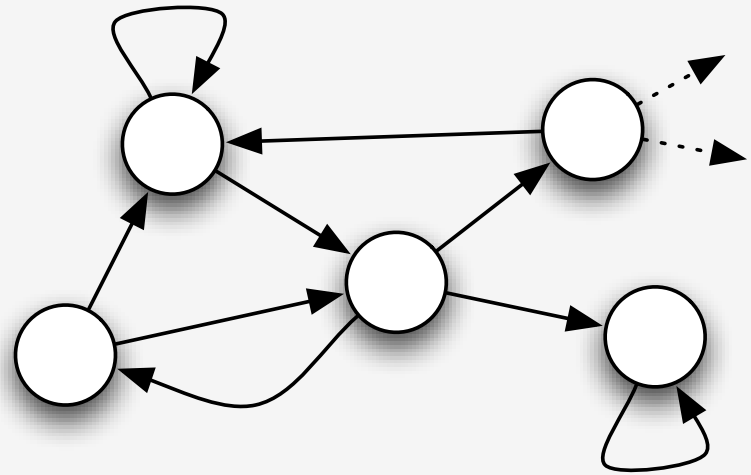
states can be actual states of a machine (e.g. on/off) or abstract states of some defined problem. An agent could be in a state of *"I don't know what this object is"*.

actions can be low-level motor controls of a robot or high-level decisions of a planner, such as *"going to lunch"*.

General Assumptions

For now, we assume the following:

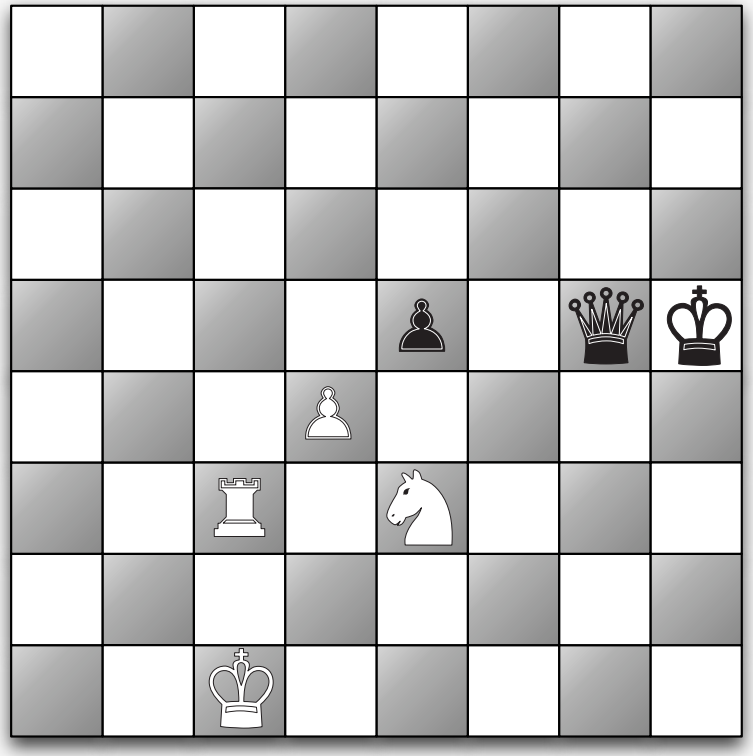
- Both states and actions are discrete and finite.
- Our problem fulfills the Markov property (MDP)
  - the current state information summarizes all relevant information from the past (e.g. chess, cannonball)
  - the next state is only determined by the last state and the last action, not the entire history
  - the environment has a stationary $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$.



Example: Chess

Imagine an agent playing one of the players in a chess game.

- What are the states? actions? rewards?
- Is the environment probabilistic or deterministic?
- What is a good move for black?



Pawn		1
Bishop		3
Knight		3
Rook		5
Queen		9

Example: Chess

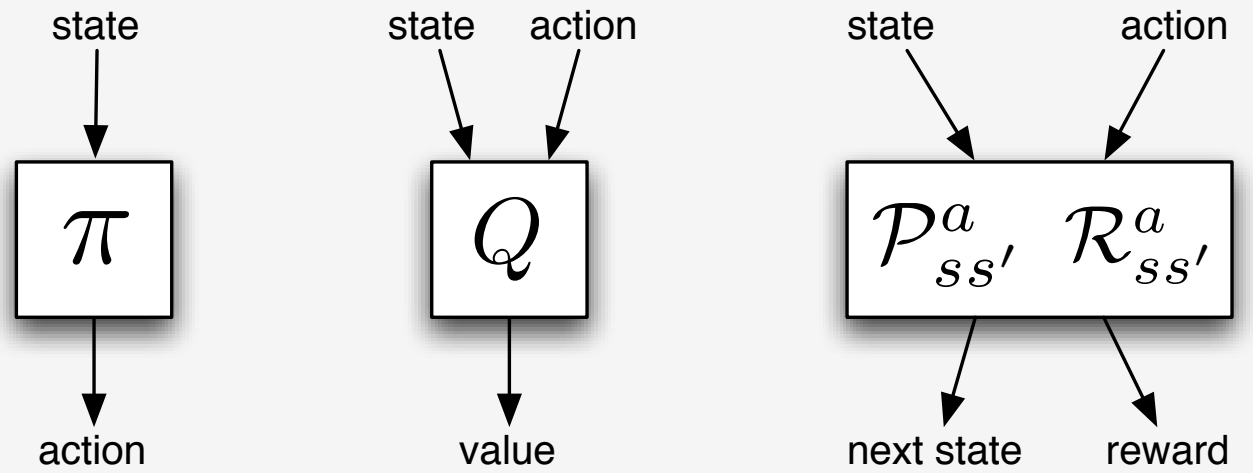
state one legal combination of pieces on the board ($\approx 10^{50}$ possible states)

actions all possible legal moves in a certain state

rewards end of game ($1 = \text{win}, -1 = \text{loss}$)
but: heuristics for taking pieces and certain positions

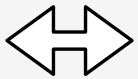
environment chess itself is deterministic, but since one agent plays against an opponent, it depends on the opponent. other player can be considered probabilistic.

Different Types of RL



Direct RL **Value-based RL** **Model-based RL**

data is cheap
computation is limited
e.g. embedded systems



data is expensive
computation doesn't matter
e.g. medical trials

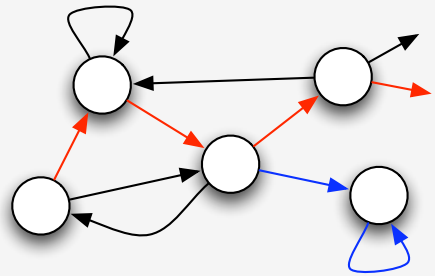
Value-Based RL – Overview

- Returns R_t
- Policy π
- Value Functions $V^\pi(s)$ and $Q^\pi(s,a)$
- Optimal Policy π^* and Optimal Value Functions V^* and Q^*
- Policy Iteration
  - Policy Evaluation $\pi \longrightarrow Q^\pi$
  - Policy Improvement $Q^\pi \longrightarrow \pi', \quad \pi' \geq \pi$
- Monte Carlo Algorithm
- Temporal Difference RL
  - SARSA
  - Q-Learning
- Eligibility Traces

Returns

What is the goal of the agent? What should we optimize?

- Change agent's behavior in a way that it selects the actions that maximize sum of future rewards: $R = r_{t+1} + r_{t+2} + r_{t+3} + \cdots$
- Problem of infinite sums in continuous settings (blue path) $\Rightarrow$ use discounting



Definition (return, discounting factor)

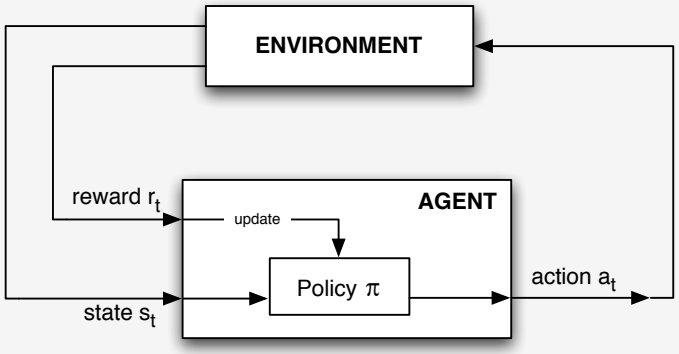
The return R_t is defined as the (discounted) sum of rewards starting at time t+1.

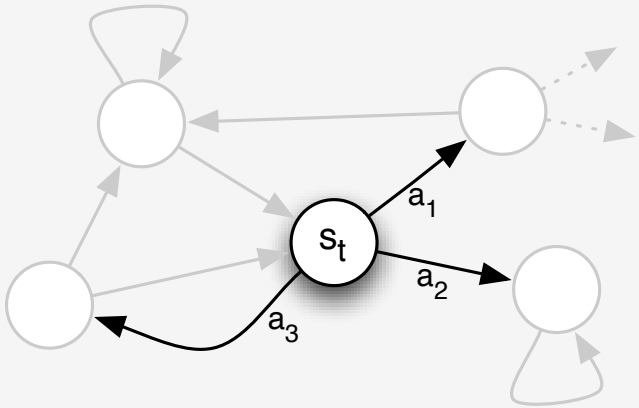
$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

where $0 < \gamma \leq 1$ is the discounting factor which ensures that recent rewards are more important than older ones. For continuous tasks, where $T = \infty$, the discounting factor must be strictly less than 1.

Policy

The agent's behavior is formally described by its policy. It returns an action for each state.





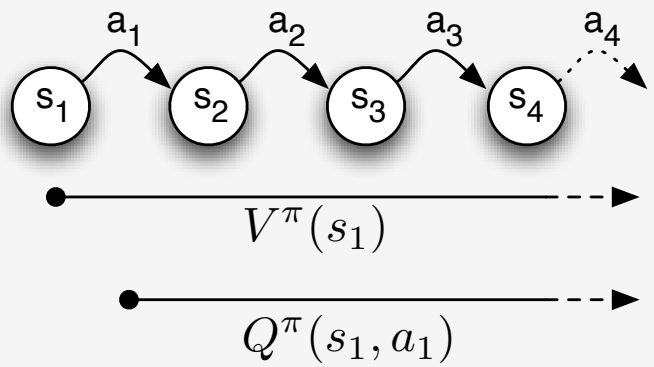
Definition (policy)

An agent's policy is its mapping from observed states to probabilities of choosing an action. $\pi(s_t, a_t) : \mathcal{S} \times \mathcal{A}(s_t) \mapsto [0, 1], \quad \pi(s_t, a_t) = p(a_t|s_t)$.

If the policy is deterministic: $\pi(s, a) = 0 \quad \forall a \in \mathcal{A}(s) \setminus \{a'\}$, and $\pi(s, a') = 1$ then we write $\pi(s) = a'$.

Value Functions

- We need to make use of the returns R_t to change the agent's behavior. One way to do this: value functions.
- A value function describes, how *good* it is for the agent, to be in a certain state, or how *good* it is to perform a certain action in a state.
- *good*ness is defined as the expected future return
- Value functions depend on the policy π of the agent
- 2 variants: we can start in a state (state-value function V^π) or during an action (action-value function Q^π).



Value Functions

Definition (state-value function)

A state-value function $V^\pi(s)$ with respect to policy π is a function of a state s, which describes the expected future return R_t if the agent starts in state s and follows policy π thereafter:

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

Definition (action-value function)

An action-value function $Q^\pi(s,a)$ with respect to policy π is a function of a state s and action a, which describes the expected future return R_t if the agent starts in state s, takes action a and follows policy π thereafter:

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

Value Functions

We will concentrate on action-value functions. From an action-value function, we can always get the state-values $V^\pi(s) = Q^\pi(s, \pi(s))$.

Action values can be represented in a Q-Table.

$Q^\pi(s,a)$	a_1	a_2	a_3
s_1	2.0	-0.5	0.1
s_2	-1.0	3.35	0.0
s_3	0.3	0.3	-0.3
s_4	10.4	0.5	0.5

The values can be estimated from experience: Follow π and keep seperate return estimates for each state-action pair $\Rightarrow$ Policy Evaluation

Optimal Policies & Optimal Value Functions

Definition (Ranking of Policies)

A policy π is considered "better" than π' if the expected returns of π are higher than or equal to policy π's for **all** states and actions:

$$Q^{\pi}(s,a) \geq Q^{\pi'}(s,a) \quad \forall\, s \in \mathcal{S}, a \in \mathcal{A}(s)$$

- The best policy (or policies) is denoted π^* (optimal policy).
- The matching value function $Q^{\pi^*}(s,a) = Q^*(s,a)$ is known as the optimal value function and it is a unique solution to a reinforcement learning problem.
- Our goal: finding Q^*

Policy Evaluation

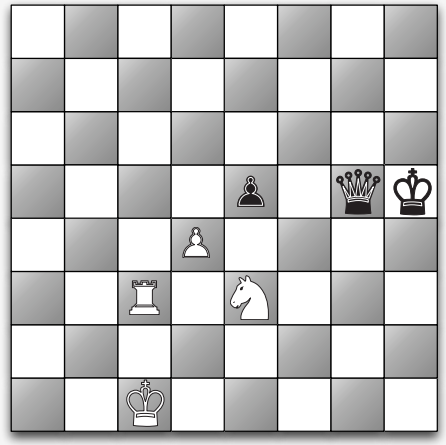
If we have a policy π, how do we get the corresponding action value function Q^π? In short: $\pi \longrightarrow Q^\pi$

1. Start in random state with random action
2. Follow π
3. Keep history of all visited (s, a) pairs and received returns R

Average over received returns following a (s, a) pair converges to $Q(s, a)$ value $\Rightarrow$ Learning from experience (Monte Carlo)

Example

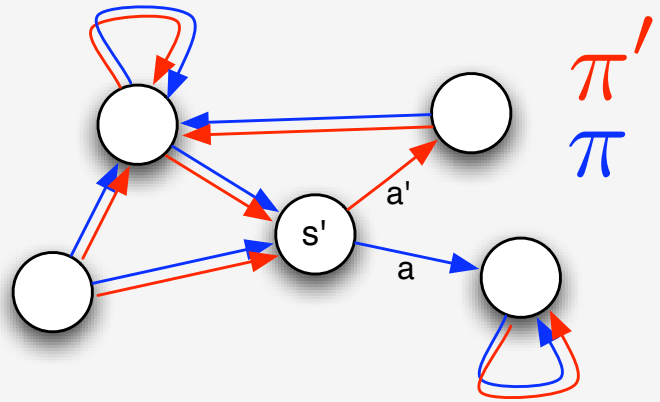
Assume we play many chess games and encounter (s, a) a total of 5 times. After finishing these games, we received the rewards $1, 1, 1, -1, 1$. $R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$ but because all except the last reward of this game are 0, $R_t = r_T$ and we get $Q^\pi(s, a) = \frac{1+1+1-1+1}{5} = 0.6$. With more games containing (s, a) the approximation improves.



Policy Improvement

Now, we want to improve a (deterministic) policy π.

- Assume π' identical to π except for state s': π selects a and π' selects a' (deterministic policies): $\pi(s') = a, \pi'(s') = a'$
- $Q^\pi(s, \pi'(s)) = Q^\pi(s, \pi(s)) \quad \forall s \in \mathcal{S} \setminus \{s'\}, a \in \mathcal{A}(s)$
- if $Q^\pi(s', \pi'(s)) \geq Q^\pi(s', \pi(s))$ then $Q^{\pi'}(s, a) \geq Q^\pi(s, a)$ $\forall s, a$ and therefore π' is better than π (or at least equal).



This is called the Policy Improvement Theorem.

Policy Improvement

All we need to do is find a π' that improves Q in one or more states. We can construct a policy that does exactly that:

$$\pi'(s) = \arg\max_a Q^\pi(s, a)$$

This greedy policy uses the current Q-table to execute the currently most promising action in each state. By above definition,

$$Q^{\pi'}(s, a) \geq Q^\pi(s, a) \quad \forall s, a$$

and therefore

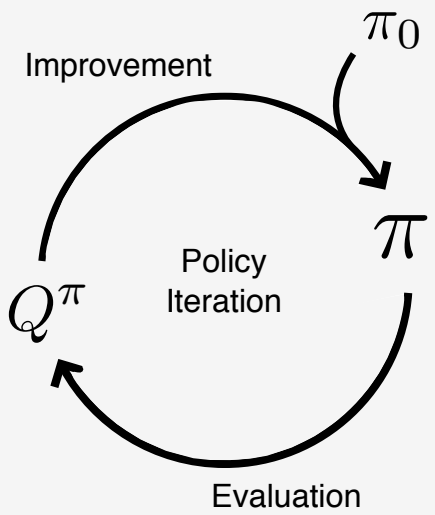
$$\pi' \geq \pi$$

Policy Iteration

We can evaluate a policy ($\pi \rightarrow Q^\pi$) and we can find a better policy by taking the greedy policy of the last value function ($Q^\pi \rightarrow \pi'$ with $\pi' \geq \pi$).

This leads to the policy iteration cycle:

$$\pi_0 \xrightarrow{E} Q^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} Q^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} Q^*$$



Monte Carlo Reinforcement Learning

Monte-Carlo algorithm

1. initialize π, Q arbitrarily, returns$(s, a) \leftarrow$ empty list
2. generate an episode starting at random (s, a) and follow π
3. for each (s, a) pair in episode
   - add R to returns(s, a)
   - $Q(s, a) \leftarrow \langle$ returns$(s, a) \rangle$, $\quad \langle \cdot \rangle$ sample average
4. for each s in episode
   - $\pi(s) \leftarrow \arg\max_a Q(s, a)$
5. goto 2

Example: Black Jack

state sum of own cards $c \in [4, 21]$
useable ace $u \in \{\text{yes}, \text{no}\}$
dealer's card $d \in \{A, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$s = (c, u, d)$

action hit or stick

reward $+1$ or -1 on win or loss, 0 otherwise.

episode one game

return same as reward (if $\gamma = 1$)

policy strategy of playing: $\pi(s) \mapsto \{\text{hit}, \text{stick}\}$
with $s = (c, u, d)$

Q table table of states $\times$ actions containing the Q values

Some Remarks

- If the transition probabilities $\mathcal{P}^a_{ss'}$ and expected rewards $\mathcal{R}^a_{ss'}$ are known, Dynamic Programming can be used. We skipped this part, but you can read it in Sutton & Barto's Book, Chapter 4.
- Often, exploring starts are not possible (real life experiences). Instead, we can use an exploring policy (called an ϵ-greedy policy)

$$\pi(s) = \begin{cases} \arg\max_a Q(s,a) & \text{if } r \geq \epsilon \\ \text{random action from } \mathcal{A}(s) & \text{if } r < \epsilon \end{cases}$$

with small ϵ and a random number $r \in [0,1]$, drawn at each time step.

Full Backups vs. Bootstraping

Dynamic Programming full model dynamics known, use bootstrapping

Monte Carlo RL collect samples from real experience, full backups

Temporal Differences collect samples and bootstrap

What is bootstrapping?

Figure: Muenchhausen pulled himself out of the swamp on his own hair.

$\Rightarrow$ In our case: We use an approximation of Q to improve Q.

Temporal Difference Learning

- In Monte-Carlo, the targets for the Q updates are the returns R_t
- We have to wait until the end of the episode to make an update
- Use approximation to R_t by bootstrapping

$$
\begin{aligned}
Q^\pi(s,a) &= E_\pi\{R_t \mid s_t = s, a_t = a\} && (1)\\
&= E_\pi\left\{\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} && (2)\\
&= E_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a\right\} && (3)\\
&= E_\pi\left\{r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a\right\} && (4)
\end{aligned}
$$

This is called the Bellman Equation for Q^π.

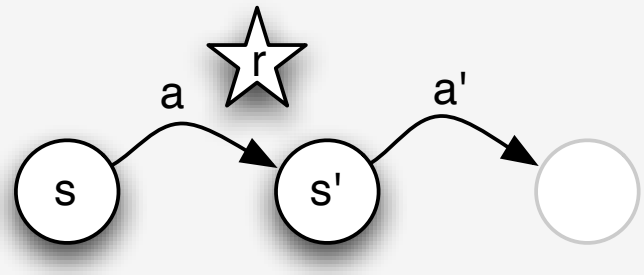
SARSA

Let's bootstrap and use the approximation for R_t

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t) \right)$$

This is our new update rule, and the algorithm is called SARSA. For an update, we need

- current state s_t and action a_t,
- the resulting reward r_{t+1},
- next state s_{t+1} and action a_{t+1}



Q-Learning (Watkins, 1989)

- SARSA was on-policy (we used the real trajectory)
- Q-Learning is an off-policy method
- We can pick arbitrary (state, action, reward)-triples to approximate the Q-function

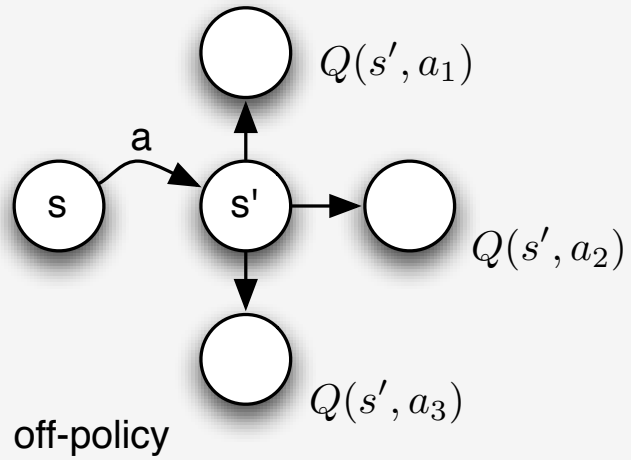
Q-Learning Update Rule

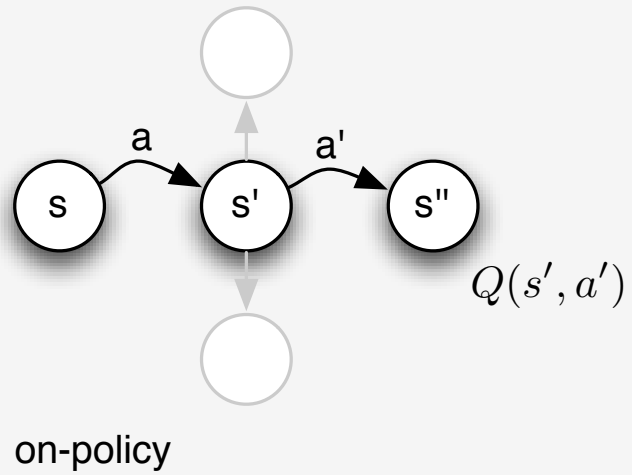
$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_a Q^\pi(s_{t+1}, a) - Q^\pi(s_t, a_t) \right)$$

Q-Learning vs. SARSA

Q-Learning Update Rule (top and left)
SARSA Update Rule (bottom and right)

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha(r_{t+1} + \gamma \cdot \max_a Q^\pi(s_{t+1}, a) - Q^\pi(s_t, a_t))$$
$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha(r_{t+1} + \gamma \cdot Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t))$$





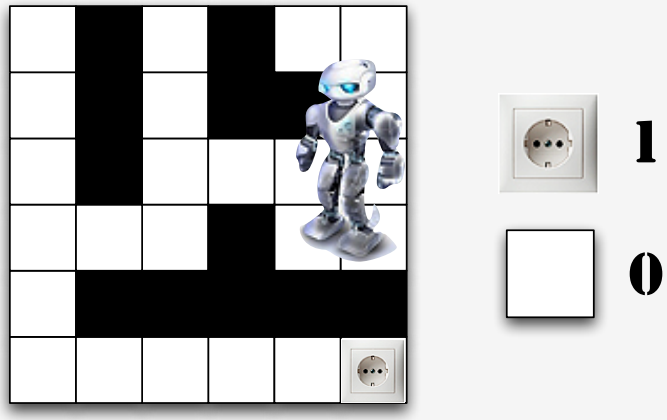
Q-Learning Algorithm

1. initialize π, Q arbitrarily
2. choose a using ϵ-greedy policy

$$a = \pi(s) = \begin{cases} \arg\max_a Q(s,a) & \text{if random}(0,1) \geq \epsilon \\ \text{random action from } \mathcal{A}(s) & \text{else} \end{cases}$$

3. execute action a, observe r, s'
4. $Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma max_a Q(s',a) - Q(s,a)\right)$
5. $s \leftarrow s'$
6. goto 2

Eligibility Traces



- How do the Q-values change when the robot walks from field to field? $(\gamma = 1, \alpha = 0.1)$

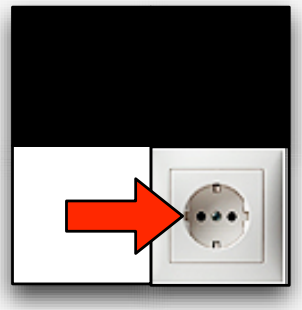
$$\Rightarrow Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + 0.1 \cdot (0 + 1 \cdot Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t))$$

$$\Rightarrow Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + 0.1 \cdot Q^\pi(s_{t+1}, a_{t+1}) - 0.1 \cdot Q^\pi(s_t, a_t)$$

$$\Rightarrow Q^\pi(s_t, a_t) \leftarrow 0.9 \cdot Q^\pi(s_t, a_t) + 0.1 \cdot Q^\pi(s_{t+1}, a_{t+1})$$

- Answer: not much, we just stir around a bit in the Q-table. If we initialize the table with 0, then absolutely nothing happens.

Eligibility Traces



- How do the Q-values change when the robot finds the power outlet ($\gamma = 1, \alpha = 0.1$)

$$\Rightarrow Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + 0.1 \cdot (1 + 1 \cdot Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t))$$

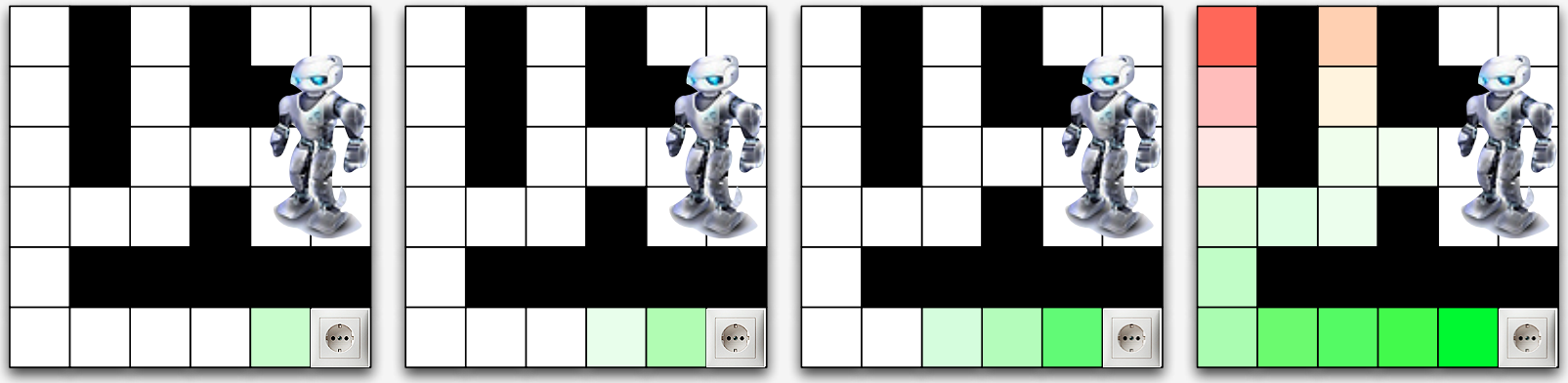
$$\Rightarrow Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + 0.1 + 0.1 \cdot Q^\pi(s_{t+1}, a_{t+1}) - 0.1 \cdot Q^\pi(s_t, a_t)$$

$$\Rightarrow Q^\pi(s_t, a_t) \leftarrow 0.9 \cdot Q^\pi(s_t, a_t) + 0.1 \cdot Q^\pi(s_{t+1}, a_{t+1}) + 0.1.$$

- Answer: if Q-table was initialized with 0, then the state-action pair that lead to the goal now has a value of 0.1.

Eligibility Traces

Q-values (here actually summarized as V-Values) after 1, 2, 3, many runs



- for each run (episode) we move backwards from the goal to the start by one state-action pair.
- this is slow!
- Eligibility Traces: remember where you came from. Use the trace (track, route, path, ...) to speed up convergence.

Eligibility Traces

How does it work?

- Each state-action pair (s, a) receives a counter $e_t(s, a)$, that keeps track of its eligibility at time t (i.e. how recently it was visited).
- Initially, all the counters are set to 0 (no state-action pair has been visited).
- Whenever a state-action pair (s, a) is visited, we add $+1$ to its eligibility.
- All eligibilities decay over time, with each timestep we multiply them with $\gamma\lambda$, where $0 \leq \lambda \leq 1$ as trace decay parameter.

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

- Main Difference: After one step, we update all Q-values for all possible state-action transitions but multiply them with their eligibility $e_t(s, a)$

SARSA(λ)

Remember SARSA(0) update rule?

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha(r_{t+1} + \gamma \cdot Q^\pi(s_{t+1}, a_{t+1}) - \ Q^\pi(s_t, a_t))$$

To save space, we subsitute

$$\delta_t := r_{t+1} + \gamma Q_t^\pi(s_{t+1}, a_{t+1}) - Q_t^\pi(s_t, a_t)$$

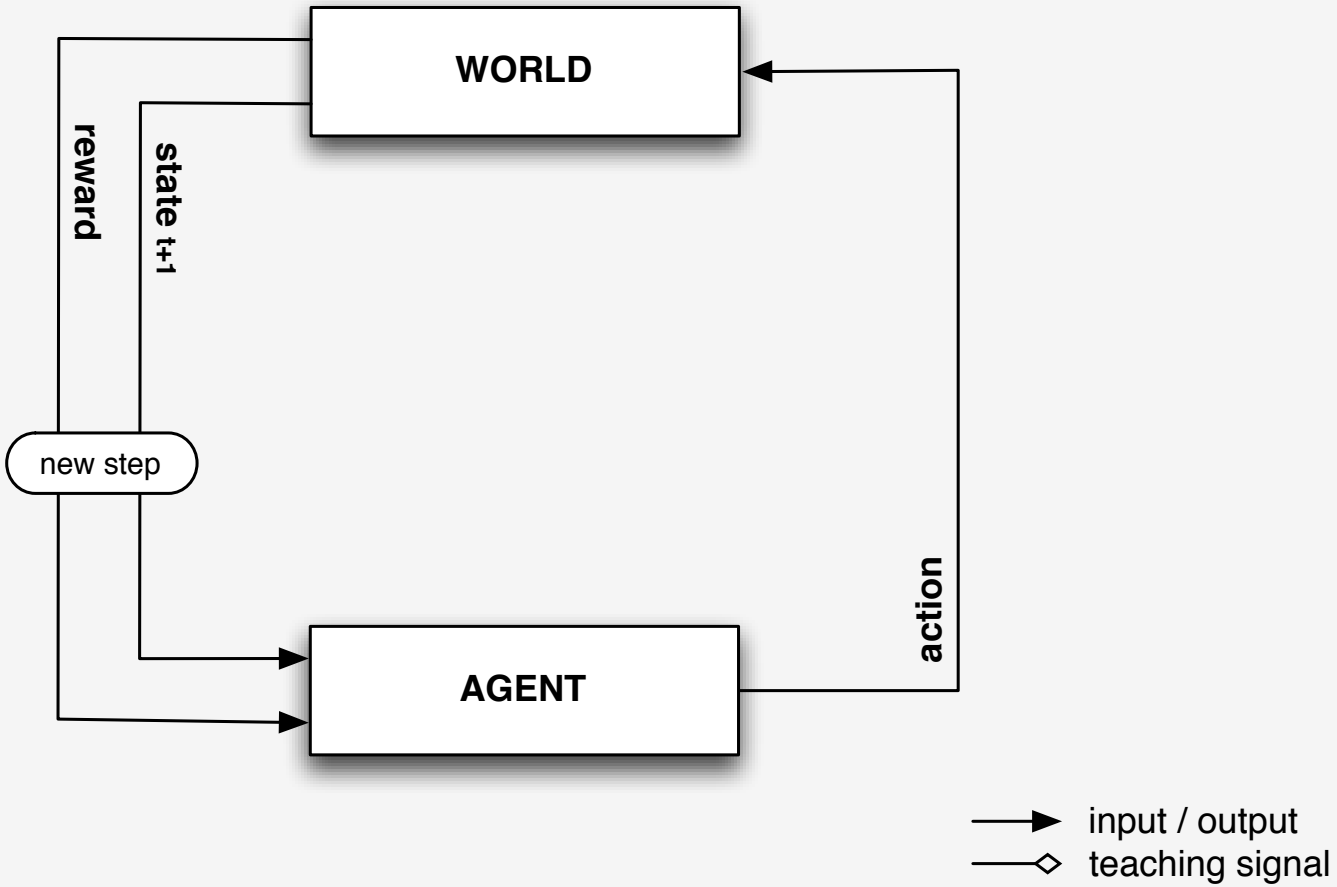
The new SARSA(λ) update is then

$$Q^\pi(s, a) \leftarrow Q^\pi(s, a) + \alpha\, \delta_t\, e_t(s, a)$$

for all s, a.

$\Rightarrow$ There is a similar extension for $Q(\lambda)$.

Planning / Model-Based RL / Dyna-Q



Planning / Model-Based RL / Dyna-Q