Reinforcement Learning

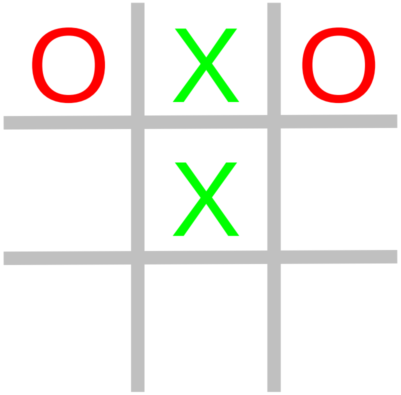
- Introduction
  - MENACE (Michie 1963)
- Formalization
  - Policies
  - Value Function
  - Q-Function
- Model-based Reinforcement Learning
  - Policy Iteration
  - Value Iteration
- Model-free Reinforcement Learning
  - Q-Learning
  - extensions
- Application Examples

Reinforcement Learning

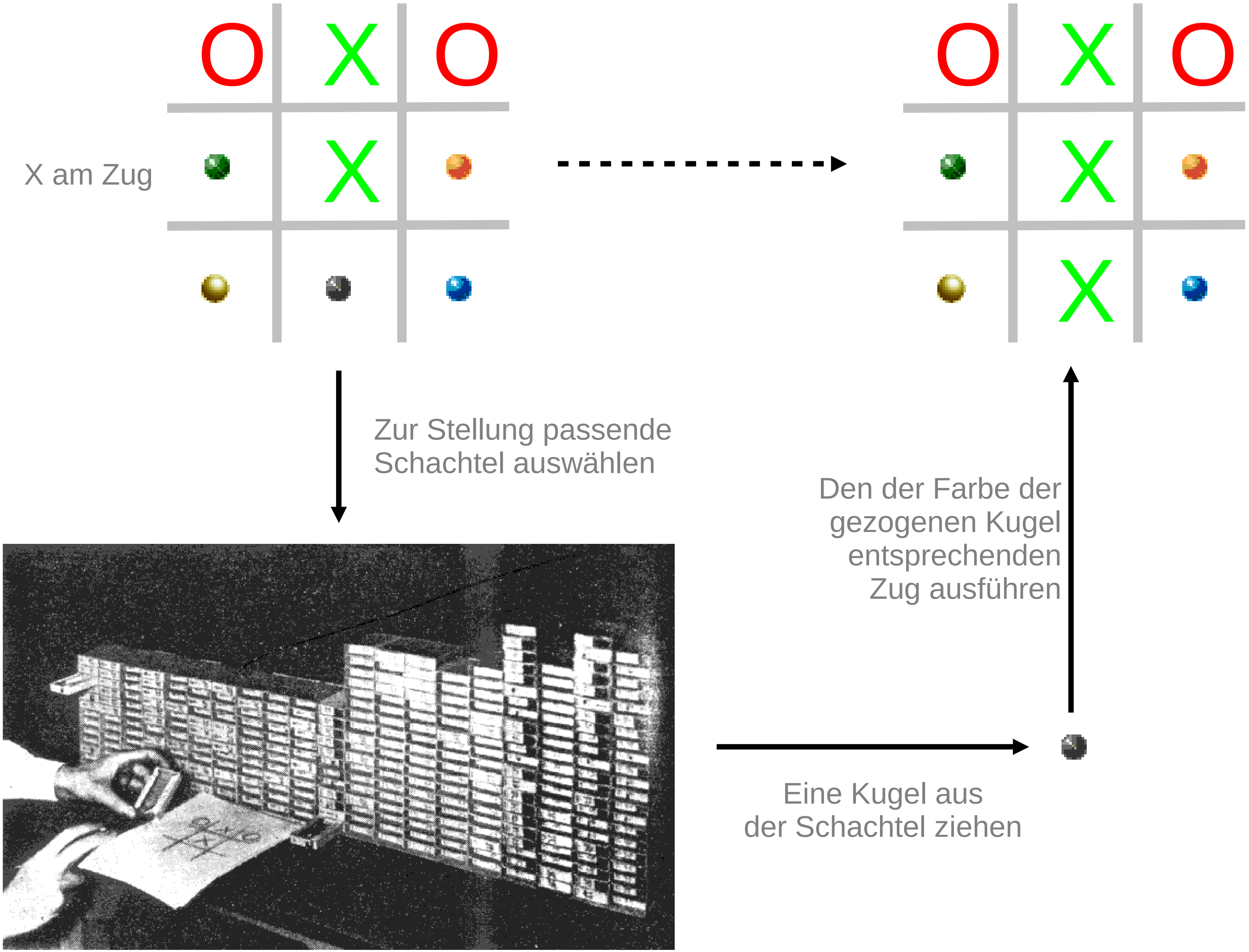
- Ziel:
  - Lernen von Bewertungsfunktionen durch Feedback (Reinforcement) der Umwelt (z.B. Spiel gewonnen/verloren).
- Anwendungen:
  - **Spiele:**
    - Tic-Tac-Toe: MENACE (Michie 1963)
    - Backgammon: TD-Gammon (Tesauro 1995)
    - Schach: KnightCap (Baxter et al. 2000)
  - **Andere:**
    - Elevator Dispatching
    - Robot Control
    - Job-Shop Scheduling

MENACE (Michie, 1963)

- Lernt Tic-Tac-Toe zu spielen



- Hardware:
  - 287 Zündholzschachteln (1 für jede Stellung)
  - Perlen in 9 verschiedenen Farbe (1 Farbe für jedes Feld)
- Spiel-Algorithmus:
  - Wähle Zündholzschachtel, die der Stellung entspricht
  - Ziehe zufällig eine der Perlen
  - Ziehe auf das Feld, das der Farbe der Perle entspricht
- Implementation: http://www.codeproject.com/KB/cpp/ccross.aspx


O X O
X am Zug
X
O X O
X
X
Zur Stellung passende
Schachtel auswählen
Den der Farbe der
gezogenen Kugel
entsprechenden
Zug ausführen
Eine Kugel aus
der Schachtel ziehen

Reinforcement Learning in MENACE

- Initialisierung
  - alle Züge sind gleich wahrscheinlich, i.e., jede Schachtel enthält gleich viele Perlen für alle möglichen Züge
- Lern-Algorithmus:
  - Spiel verloren → gezogene Perlen werden einbehalten (*negative reinforcement*)
  - Spiel gewonnen → eine Perle der gezogenen Farbe wird in verwendeten Schachteln hinzugefügt (*positive reinforcement*)
  - Spiel remis → Perlen werden zurückgelegt (keine Änderung)
- führt zu
  - Erhöhung der Wahrscheinlichkeit, daß ein erfolgreicher Zug wiederholt wird
  - Senkung der Wahrscheinlichkeit, daß ein nicht erfolgreicher Zug wiederholt wird

Credit Assignment Problem

- Delayed Reward
  - Der Lerner merkt erst am Ende eines Spiels, daß er verloren (oder gewonnen) hat
  - Der Lerner weiß aber nicht, welcher Zug den Verlust (oder Gewinn verursacht hat)
    - oft war der Fehler schon am Anfang des Spiels, und die letzten Züge waren gar nicht schlecht
- Lösung in Reinforcement Learning:
  - Alle Züge der Partie werden belohnt bzw. bestraft (Hinzufügen bzw. Entfernen von Perlen)
  - Durch zahlreiche Spiele konvergiert dieses Verfahren
    - schlechte Züge werden seltener positiv verstärkt werden
    - gute Züge werden öfter positiv verstärkt werden

Reinforcement Learning - Formalization

- Learning Scenario
  - a learning agent
  - S: a set of possible states
  - A: a set of possible actions
  - a state transition function $\delta: S \times A \rightarrow S$
  - a reward function $r: S \times A \rightarrow \mathbb{R}$
- Enviroment:
  - the agent repeatedly chooses an action according to some *policy* $\pi: S \rightarrow A$
  - this will put the agent into a new state according to δ
  - in some states the agent receives feedback from the environment (reinforcement)

- Markov property
  - rewards and state transitions only depend on last state
  - not on how you got into this state

MENACE - Formalization

- Framework
  - states = matchboxes
  - actions = moves/beads
  - policy = prefer actions with higher number of beads
  - reward = game won/ game lost
    - *delayed* reward: we don't know right away whether a move was good or bad

Learning Task

find a policy that maximizes the cumulative reward

- delayed reward
  - reward for actions may not come immediately (e.g., game playing)
  - modeled as: every state s_i gives a reward r_i, but most $r_i=0$
- goal: maximize cumulative reward $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$
  - reward from "now" until the end of time
  - immediate rewards are weighted higher, rewards further in the future are discounted (discount factor γ)
- training examples
  - generated by interacting with the environment (trial and error)
  - Note:
    - training examples are not supervised (s, a_{max})
    - training examples are of the form (s, a, r)

Optimal Policies and Value Functions

- Each policy can be assigned a value
  - the cumulative expected reward that the agent receives when it follows that policy

$$s_{t+1}=\delta(s_t,a_t)$$

$$V^{\pi}(s_t)=\sum_{i=0}^{\infty}\gamma^i r_{t+i}=r_t+\gamma r_{t+1}+\gamma^2 r_{t+2}\ldots=$$

$$=r_t+\gamma(r_{t+1}+\gamma r_{t+2}+\ldots)=r(s_t,a_t)+\gamma V^{\pi}(\delta(s_t,a_t))$$

- Optimal policy
  - the policy with the highest expected value for all states s

$$\pi^*=arg\,max_{\pi}V^{\pi}(s)$$

  - learning an optimal value function $V^*(s)$ yields an optimal policy

$$\pi^*(s)=arg\,max_a[r(s,a)+\gamma V^*(\delta(s,a))]$$

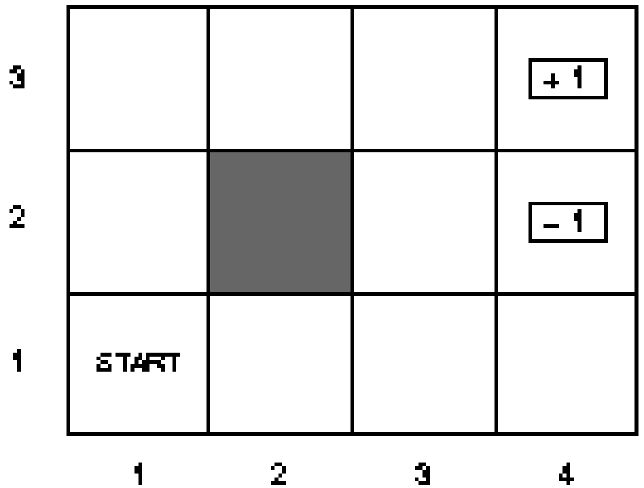
- We can try to learn the policy or the value function by starting with some function and iteratively improving it
  - policy iteration / value iteration

Unknown Actions and Rewards

- In many problems we might not know the effects of actions (δ) or the reward functions (r)
  - don't know which states are good
  - don't know which actions lead to which states
    - actions may also be indeterministic

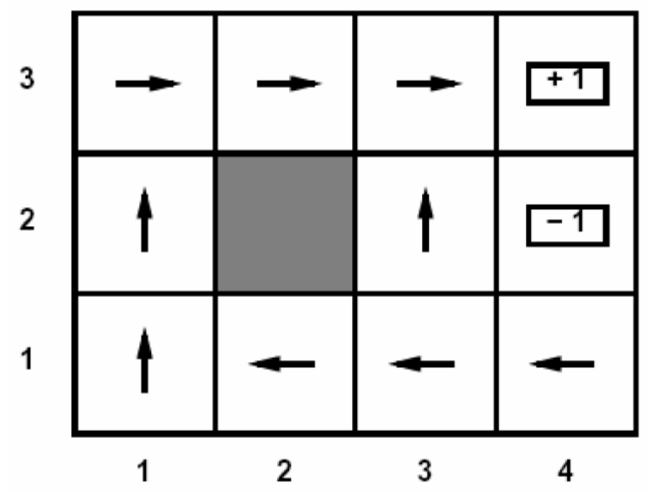
  → must try out actions to learn their effects

- Example:
  - learn to navigate in a simple tile world
  - Actions:
    - go left/right/up/down,
      - may fail (but we don't know how)
    - each action costs a small amount
  - Goal:
    - get to the upper right corner quickly
    - but don't fall into the pit below



Policy Evaluation

- Simplified task
  - we don't know δ
  - we don't know r
  - but we are given a policy π
    - i.e., we have a function that gives us an action in each state



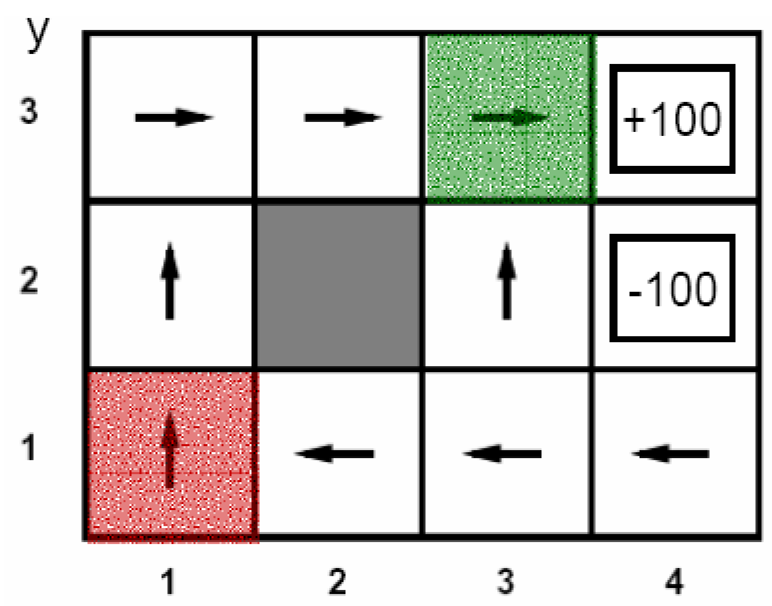
- Goal:
  - learn the value of each states
- Note:
  - here we have no choice about the actions to take
  - we just execute the policy and observe what happens

Policy Evaluation – Example

- Episodes:

(1,1) up -1	(1,1) up -1
(1,2) up -1	(1,2) up -1
(1,2) up -1	(1,3) right -1
(1,3) right -1	(2,3) right -1
(2,3) right -1	(3,3) right -1
(3,3) right -1	(3,2) up -1
(3,2) up -1	(4,2) exit -100
(3,3) right -1	(done)
(4,3) exit +100	
(done)	

Actions are indeterministic!



$\gamma = 1$,

$$V^{\pi}(1,1) \leftarrow (92 + -106)/2 = -7$$

$$V^{\pi}(3,3) \leftarrow (99 + 97 + -102)/3 = 31.3$$

Q-function

- the Q-function does not evaluate states, but evaluates state-action pairs
- The Q-function for a given policy π
  - is the cumulative reward for starting in s, applying action a, and, in the resulting state s', play according to π

$$Q^{\pi}(s,a) := r(s,a) + \gamma V^{\pi}(s') \qquad [s' = \delta(s,a)]$$

- For indeterministic actions:
  - The function δ does not map to a single successor action
  - but may be modeled as a probability distribution $\mathbf{P}(s'|s,a)$ over all possible successor states
  - the Q-function then needs to compute an expected value

$$Q^{\pi}(s,a) := r(s,a) + \gamma \sum_{s'} \boldsymbol{P}(s' \mid s,a) V^{\pi}(s')$$

- for the moment we stick with the deterministic case

Policy Iteration

- Policy Improvement Theorem
  - if it is true that selecting the first action in each state according to a policy π' and continuing with policy π is better than always following π then π' is a better policy than π

$$V^{\pi'}(s) \geq V^{\pi}(s) \Leftrightarrow Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s)$$

- Policy Improvement
  - always select the action that maximizes the Q-function of the current policy

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

- Policy Iteration
  - Interleave steps of policy evaluation with policy improvement

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

Value Iteration

- Policy Iteration works, but it involves frequent steps of policy evaluations
  - may be expensive
  - we have to run the agent several times before the estimates of V^π converge
- Value Iteration directly updates a value function $\hat{V}$

$$\hat{V}(s) \leftarrow \max_a Q^{\hat{V}}(s,a) = \max_a \left(r(s,a) + \gamma \hat{V}(s') \right)$$

- In practice, value iteration is much faster per iteration, but policy iteration takes fewer iterations.

Model-Free Reinforcement Learning

- Both, Value and Policy Iteration need the maximal Q-function for each action

$$Q(s,a) := r(s,a) + \gamma V(s') \qquad [s' = \delta(s,a)]$$

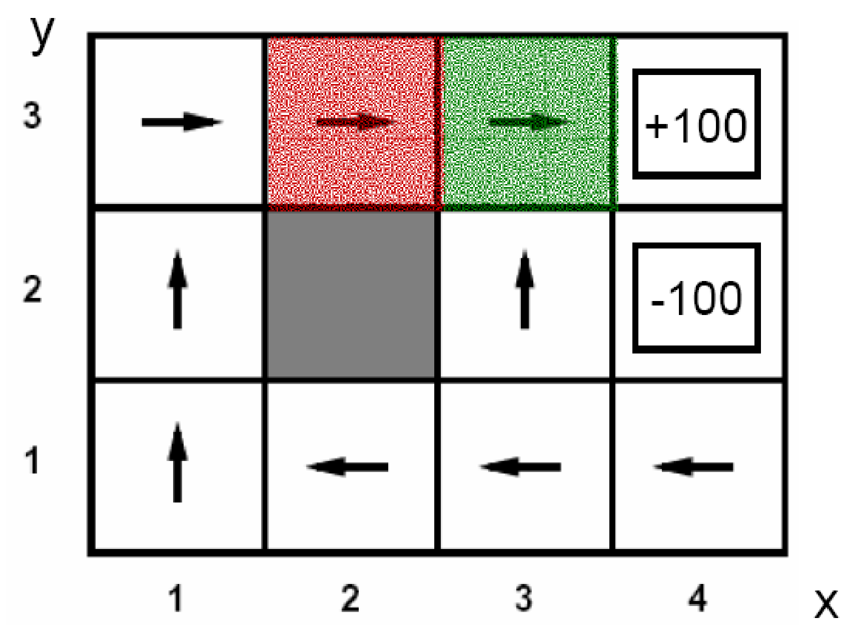
- BUT
  - for computing this maximum we need to know the functions r and δ
  - i.e., we need a model of the world
- Can we learn to act without having a model of the world?

Simple Approach: Learn the Model from Data

- Episodes:

(1,1) up -1
(1,2) up -1
(1,2) up -1
(1,3) right -1
(2,3) right -1
(3,3) right -1
(3,2) up -1
(3,3) right -1
(4,3) exit +100
(done)

(1,1) up -1
(1,2) up -1
(1,3) right -1
(2,3) right -1
(3,3) right -1
(3,2) up -1
(4,2) exit -100
(done)



$\gamma = 1$

$$\mathbf{P}((4,3) \mid (3,3), \text{right}) = 1/3$$

$$\mathbf{P}((3,3) \mid (2,3), \text{right}) = 2/2$$

But do we really need to learn the transition model?

Optimal Q-function

- the optimal Q-function is the cumulative reward for starting in s, applying action a, and, in the resulting state s', play optimally

$$Q(s,a) := r(s,a) + \gamma V^*(s') \qquad [s' = \delta(s,a)]$$

→ the optimal value function is the maximal Q-function over all possible actions in a state $V^*(s) = max_a Q(s,a)$

- *Bellman equation*: $Q(s,a) = r(s,a) + \gamma\, max_{a'} Q(s',a')$
  - the value of the Q-function for the current state s and an action a is the same as the sum of
    - the reward in the current state s for the chosen action a
    - the (discounted) value of the Q-function for the best action that I can play in the successor state s'

Better Approach: Directly Learning the Q-function

- Basic strategy:
  - start with some function $\hat{Q}$, and update it after each step
  - in MENACE: $\hat{Q}$ returns for each box s and each action a the number of beads in the box
- update rule:
  - the Bellman equation will in general not hold for $\hat{Q}$ i.e., the left side and the right side will be different

  → new value of $\hat{Q}(s,a)$ is a weighted sum of both sides
  - weighted by a learning rate α

$$\hat{Q}(s,a) \leftarrow (1-\alpha)\hat{Q}(s,a) + \alpha(r(s,a)+\gamma \max_{a'}\hat{Q}(s',a'))$$

$$\leftarrow \hat{Q}(s,a) + \alpha[r(s,a)+\gamma \max_{a'}\hat{Q}(s',a')-\hat{Q}(s,a)]$$¿

new Q-value for state s and action a

old Q-value for this state/action pair

predicted Q-value for state s' and action a'

Q-learning (Watkins, 1989)

1. initialize all $\hat{Q}(s,a)$ with 0
2. observe current state s
3. loop
   1. select an action a and execute it
   2. receive the immediate reward and observe the new state s'
   3. update the table entry

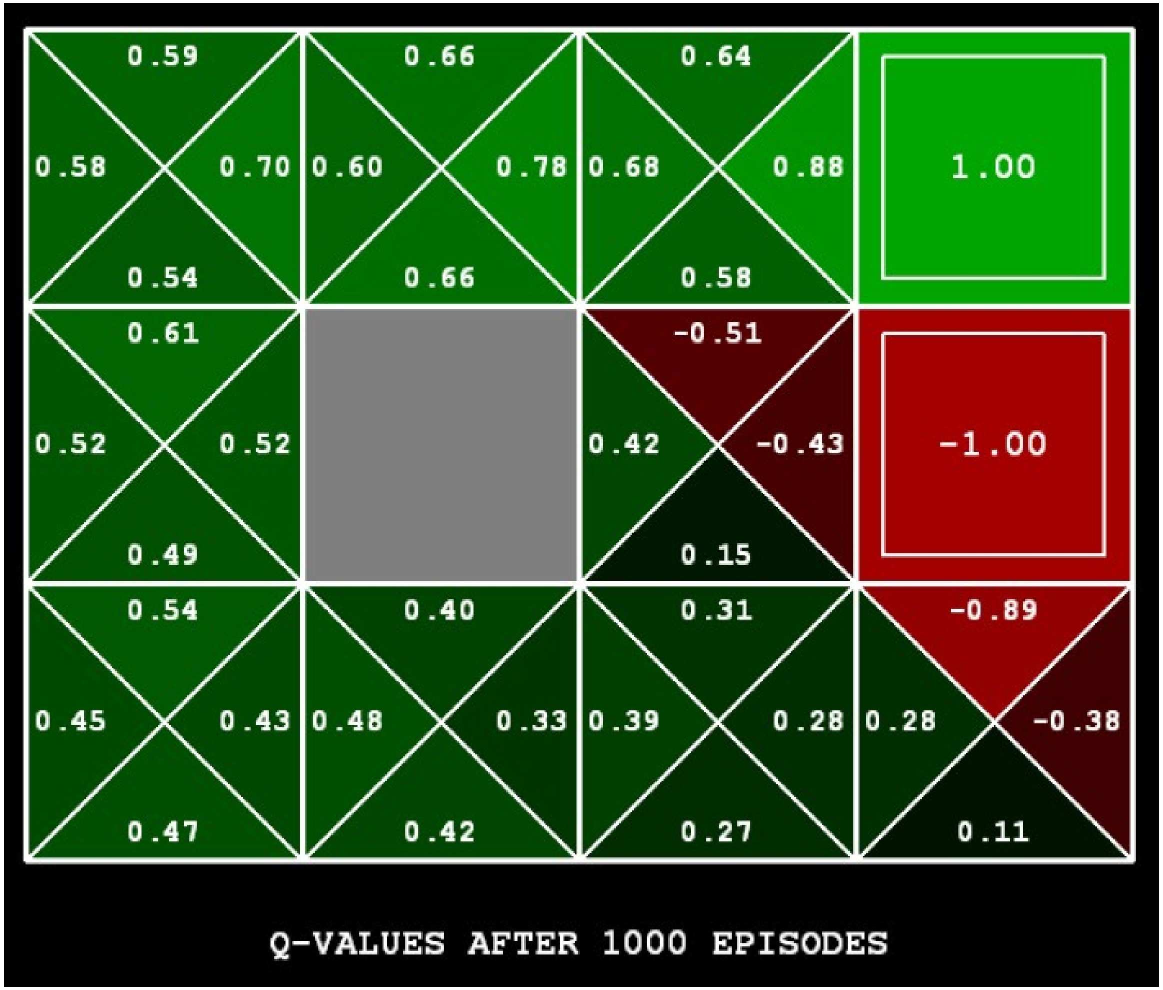
      $$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha\left[\underline{\left(r(s,a) + \gamma \max_{a'} \hat{Q}(s',a')\right)} - \underline{\hat{Q}(s,a)}\right]$$

   4. $s = s'$

Temporal Difference:
Difference between the estimate of the value of a state/action pair before and after performing the action.
→ Temporal Difference Learning

Example: Maze

- Q-Learning will produce the following values

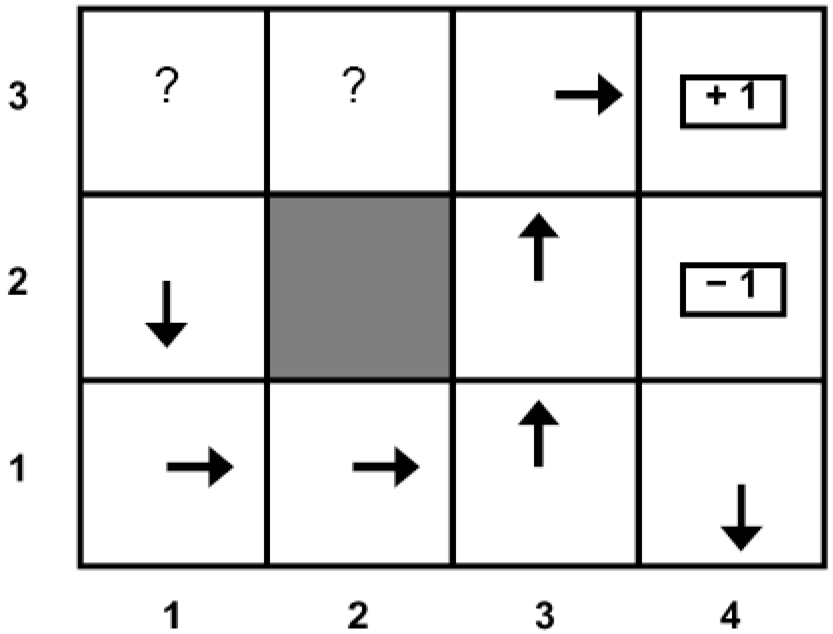


Miscellaneous

- Weight Decay:
  - α decreases over time, e.g. $\alpha = \frac{1}{1 + visits(s,a)}$
- Convergence:
  it can be shown that Q-learning converges
  - if every state/action pair is visited infinitely often
    - not very realistic for large state/action spaces
    - but it typically converges in practice under less restricting conditions
- Representation
  - in the simplest case, $\hat{Q}(s,a)$ is realized with a look-up table with one entry for each state/action pair
  - a better idea would be to have trainable function, so that experience in some part of the space can be generalized
  - special training algorithms for, e.g., neural networks exist

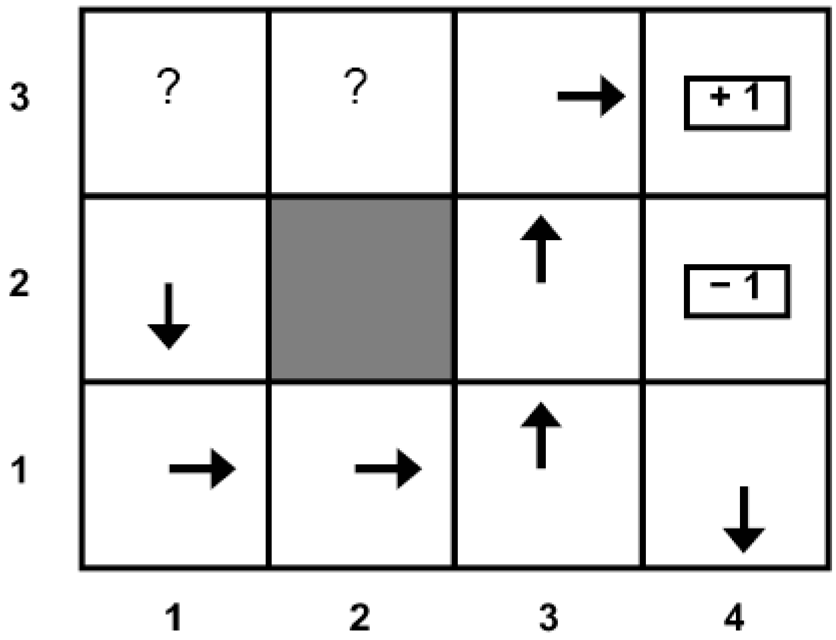
Exploration vs. Exploitation

- Imagine we find the lower path to the good exit first
- Some states will never be visited following this policy from (1,1)
- We'll keep re-using this policy because following it never collects the regions of the model we need to learn the optimal policy



Exploration vs. Exploitation

- Problem with following optimal policy for current model:
  - Never learn about better regions of the space if current policy neglects them
- Fundamental tradeoff: exploration vs. exploitation
  - Exploration: must take actions with suboptimal estimates to discover new rewards and increase eventual utility
  - Exploitation: once the true optimal policy is learned, exploration reduces utility
  - Systems must explore in the beginning and exploit in the limit



ε-greedy policies
- choose random action with probability ε, otherwise greedy
- reduce ε over time

SARSA

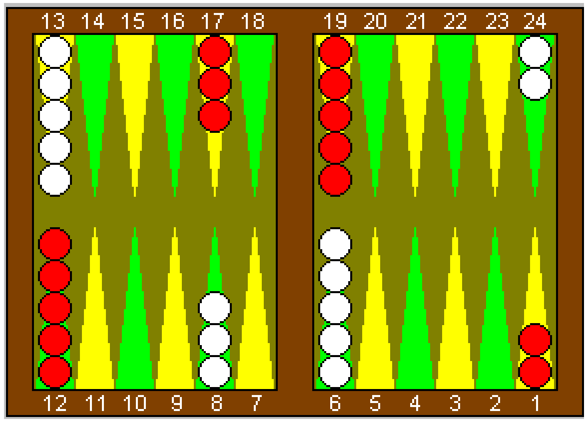
- performs *on-policy updates*
  - update rule assumes action a' is chosen according to current policy

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha [r(s,a) + \gamma \hat{Q}(s',a') - \hat{Q}(s,a)]$$

  - convergence if the policy gradually moves towards a policy that is greedy with respect to the current Q-function

TD-Gammon (Tesauro, 1995)

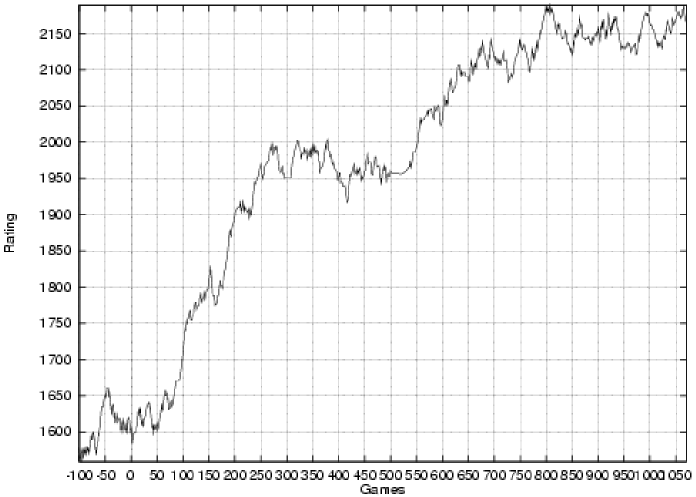
- weltmeisterliches Backgammon-Programm
  - Entwicklung von Anfänger zu einem weltmeisterlichen Spieler nach 1,500,000 Trainings-Spiele gegen sich selbst (!)
  - Verlor 1998 WM-Kampf über 100 Spiele knapp mit 8 Punkten
  - Führte zu Veränderungen in der Backgammon-Theorie und ist ein beliebter Trainings- und Analyse-Partner der Spitzenspieler



- Verbesserungen gegenüber MENACE:
  - Schnellere Konvergenz durch Temporal-Difference Learning
  - Neurales Netz statt Schachteln und Perlen erlaubt Generalisierung
  - Verwendung von Stellungsmerkmalen als Features

KnightCap (Baxter et al. 2000)

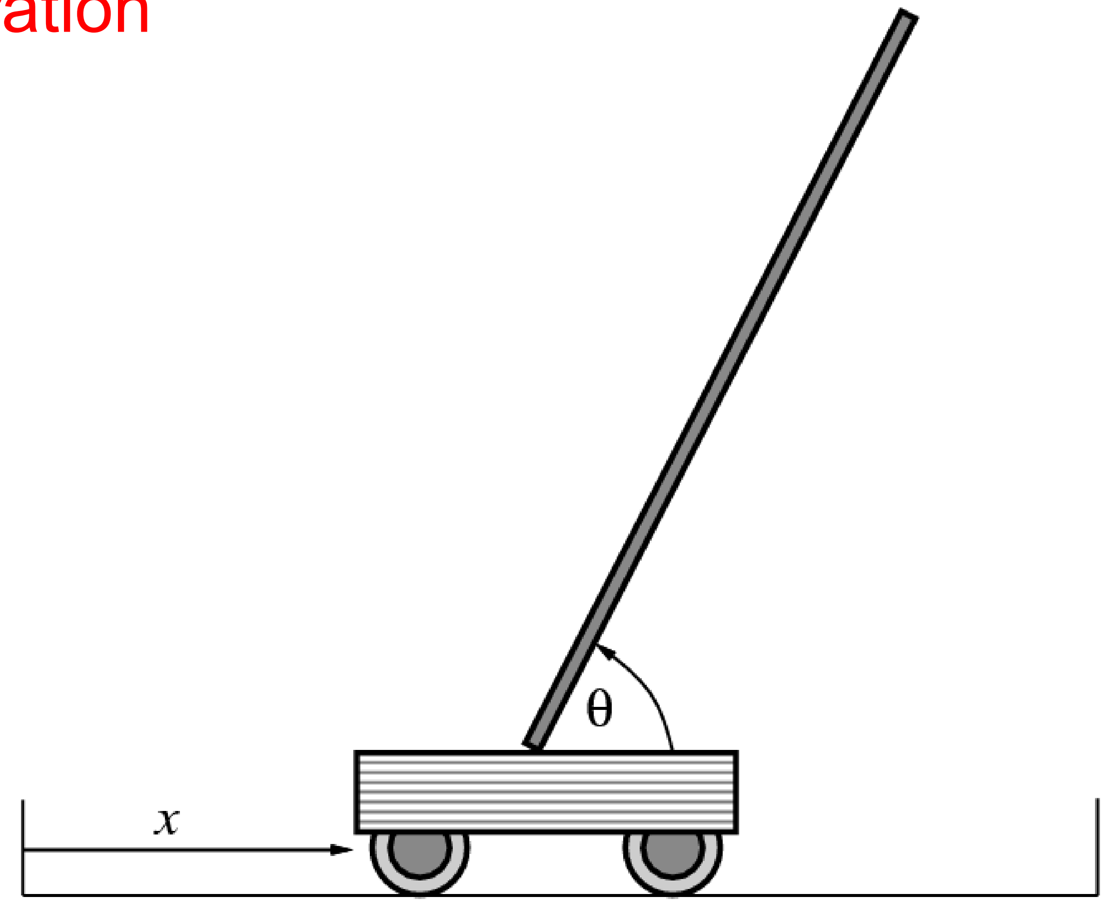
- Lernt meisterlich Schach zu spielen
  - Verbesserung von 1650 Elo (Anfänger) auf 2150 Elo (guter Club-Spieler) in nur ca. 1000 Spielen am Internet



- Verbesserungen gegenüber TD-Gammon:
  - Integration von TD-learning mit den tiefen Suchen, die für Schach erforderlich sind
  - Training durch Spielen gegen sich selbst → Training durch Spielen am Internet

Cart – Pole balancing

- Demonstration



http://www.bovine.net/~jlawson/hmc/pole/sane.html

Inverted Pendulum

- Demo

http://www.eecg.utoronto.ca/~aamodt/BAScThesis/

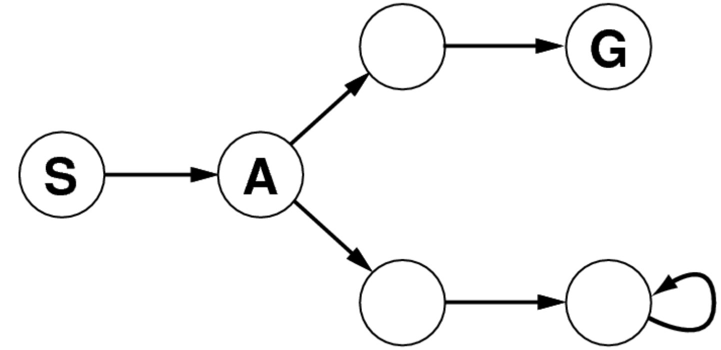
Reinforcement Learning Resources

- Book
  - On-line Textbook on Reinforcement learning
    - http://www.cs.ualberta.ca/~sutton/book/the-book.html
- More Demos
  - Grid world
    - http://thierry.masson.free.fr/IA/en/qlearning_applet.htm
  - Robot learns to crawl
    - http://www.applied-mathematics.net/qlearning/qlearning.html
- Reinforcement Learning Repository
  - tutorial articles, applications, more demos, etc.
    - http://www-anw.cs.umass.edu/rlr/
- RL-Glue (Open Source RL Programming framework)
    - http://glue.rl-community.org/

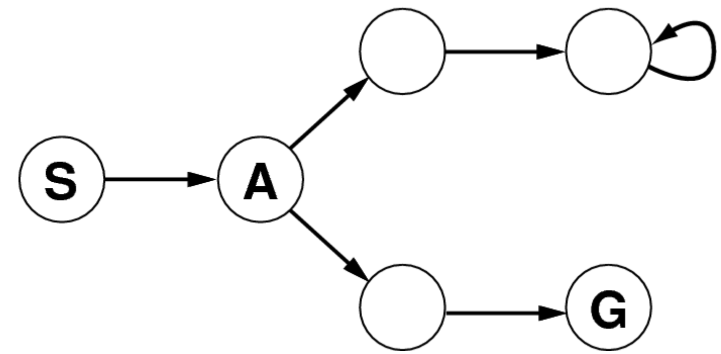
On-line Search Agents

- Off-line Search
  - find a complete solution before setting a foot in the real world
- On-line Search
  - interleaves computation of solution and action
  - good in (semi-)dynamic and stochastic domains
  - on-line versions of search algorithms can only expand the current node (because they are physically located there)
    - depth-first search and local methods are directly applicable
    - some techniques like random restarts etc. are not available
- On-line search is necessary for exploration problems
  - Example: constructing a map of an unknown building

Dead Ends & Adversary Argument



- No on-line agent is able to always avoid dead ends in all state spaces
  - dead-ends: cliffs, staircases, ...
- Example:
  - no agent that has visited **S** and **A** can can discriminate between the two choices



- Adversary argument:
  - imagine that an adversary constructs the state space while the agent explores it
  - and puts the goals and dead ends wherever it likes

→ We will assume that the search space is safely explorable
  - i.e., no dead-ends