Overview

Reinforcement Learning

- Problem: Agent acts and gets sometimes feedback. It tries to find an optimal policy, what actions in what situations maximizes its rewards. In contrast to Markov decision problems (MDPs) transition model und rewards are initially often unknown.
- Typical for all creatures (but rewards given by emotions and pain), also in games

Reinforcement Learning (RL)

- Learning from rewards instead of supervised learning (SL)
  - e.g. in chess: SL from grandmaster games (chess positions labeled with the (nearly) correct move) vs. RL from the reward at the end of the game (+1, ½, 0)
  - Similar to concept of reward in Markow decision processes (MDP): Maximize the expected sum of rewards
    - Difference of RL to MDP: The agent is not given the MDP problem, it is *in* the MDP
      - Imagine playing a new game whose rules you don‘t know: after 100 moves you are told „you lose“
  - Main advantage of RL (compared to SL): Often simple to supply labeled examples
    - e.g. with simulations (like self play for games)
  - Main problem of RL: Sparse rewards
    - Intermediate rewards make learning much easier (e.g. scores in Atari games)

Categorization of RL Approaches (1)

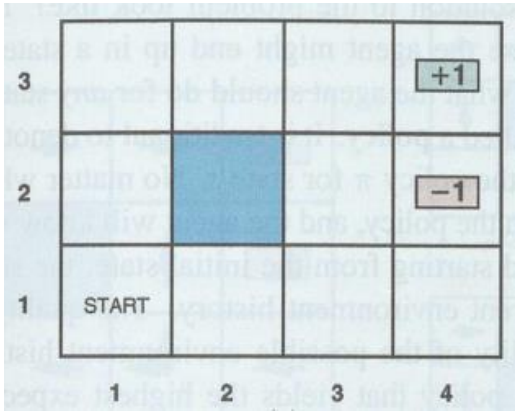
- **Model-based reinforcement learning:**
  - Agent uses at **transition model** of the environment e.g. for state estimation
    - Transition model may be initially unknown
  - Agent often learns a **utility function** U(s): sum of rewards from state s onward
- **Model-free reinforcement learning:**
  - Without use of a transition model
  - Agent learns a direct representation how to act:
    - **Action-utility learning** like **Q-learning**, where the agent learns a **Q-function** Q(s,a) denoting the sum of rewards from state s onward if action a is taken
    - **Policy search:** Agent learns a policy π (s), i.e. rules that map from states to actions

Categorization of RL Approaches (2)

- **Passive reinforcement learning:**
  - The policy of the agent is fixed and the task is to learn the utility of the states
    - Similar to policy evaluation in policy iteration of Markov decison problems
- **Active reinforcement learning:**
  - The agent must also figure out what to do
  - Principal issue is exploration
- **Apprenticeship learning:**
  - Training a learning agent using demonstration rather than reward signals

Passive Reinforcement Learning

- Given:
  - Fully observable environment
  - Agent has a fixed policy π (s)
- Sought:
  - Agent learns the utility function $U^{\pi}(s)$: Expected total (discounted) reward in state s
- Difference to policy evaluation task of MPDs
  - Agent knows neither transition model P(s‘|s,a) nor reward function R(s,a,s‘)
    - Total reward given at the end of sequence (as sum of rewards)
- Typical trials:



$$(1,1) \xrightarrow[Up]{-.04} (1,2) \xrightarrow[Up]{-.04} (1,3) \xrightarrow[Right]{-.04} (1,2) \xrightarrow[Up]{-.04} (1,3) \xrightarrow[Right]{-.04} (2,3) \xrightarrow[Right]{-.04} (3,3) \xrightarrow[Right]{+1} (4,3)$$

$$(1,1) \xrightarrow[Up]{-.04} (1,2) \xrightarrow[Up]{-.04} (1,3) \xrightarrow[Right]{-.04} (2,3) \xrightarrow[Right]{-.04} (3,3) \xrightarrow[Right]{-.04} (3,2) \xrightarrow[Up]{-.04} (3,3) \xrightarrow[Right]{+1} (4,3)$$

$$(1,1) \xrightarrow[Up]{-.04} (1,2) \xrightarrow[Up]{-.04} (1,3) \xrightarrow[Right]{-.04} (2,3) \xrightarrow[Right]{-.04} (3,3) \xrightarrow[Right]{-.04} (3,2) \xrightarrow[Up]{-1} (4,2)$$

Basic Considerations

- Transition model can be learned simply by couting state changes (with fixed policy), e.g.
  - Transition from (1,2) to (1,3): 4/4: 1.0
  - Transition from (3,3) to (4,3): 2/4: 0.5
  - Transition from (3,3) to (3,2): 2/4: 0.5

$$(1,1) \xrightarrow[Up]{-.04} (1,2) \xrightarrow[Up]{-.04} (1,3) \xrightarrow[Right]{-.04} (1,2) \xrightarrow[Up]{-.04} (1,3) \xrightarrow[Right]{-.04} (2,3) \xrightarrow[Right]{-.04} (3,3) \xrightarrow[Right]{+1} (4,3)$$

$$(1,1) \xrightarrow[Up]{-.04} (1,2) \xrightarrow[Up]{-.04} (1,3) \xrightarrow[Right]{-.04} (2,3) \xrightarrow[Right]{-.04} (3,3) \xrightarrow[Right]{-.04} (3,2) \xrightarrow[Up]{-.04} (3,3) \xrightarrow[Right]{+1} (4,3)$$

$$(1,1) \xrightarrow[Up]{-.04} (1,2) \xrightarrow[Up]{-.04} (1,3) \xrightarrow[Right]{-.04} (2,3) \xrightarrow[Right]{-.04} (3,3) \xrightarrow[Right]{-.04} (3,2) \xrightarrow[Up]{-1} (4,2)$$

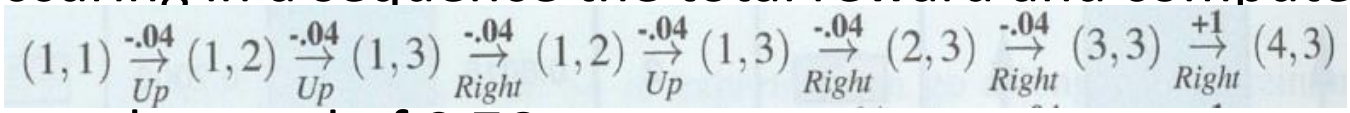
- Reward in a state is direct reward in state and future rewards in path:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right]$$

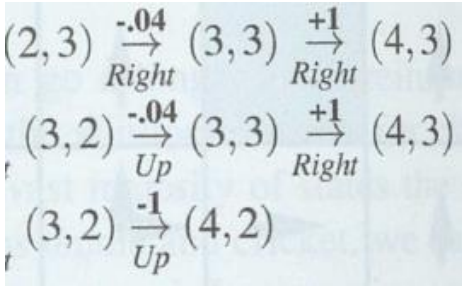
  - $R(S_t, \pi(S_t), S_{t+1})$ is reward received when action $\pi(S_t)$ is taken in state S_t and reaches S_{t+1}

Direct Utility Estimation

- Simply count for each state occuring in a sequence the total reward and compute a running average, e.g. from



  - State (1,1): 1 sample with total reward of 0.76
  - State (1,2): 2 samples with 0.8 and 0.88
  - State (1,3): 2 samples with 0.84 and 0.92
- Critic:
  - Slow convergence, because the algorithm ignores connection between states, e.g.
    - For state (3,2) the utility is -0.02 = (0.96-1.00)/2
    - For state (3,3) the utility is 1 = (1+1)/2
    - But from state (3,2), state (3,3) can be reached



Adaptive Dynamic Programming (ADP)

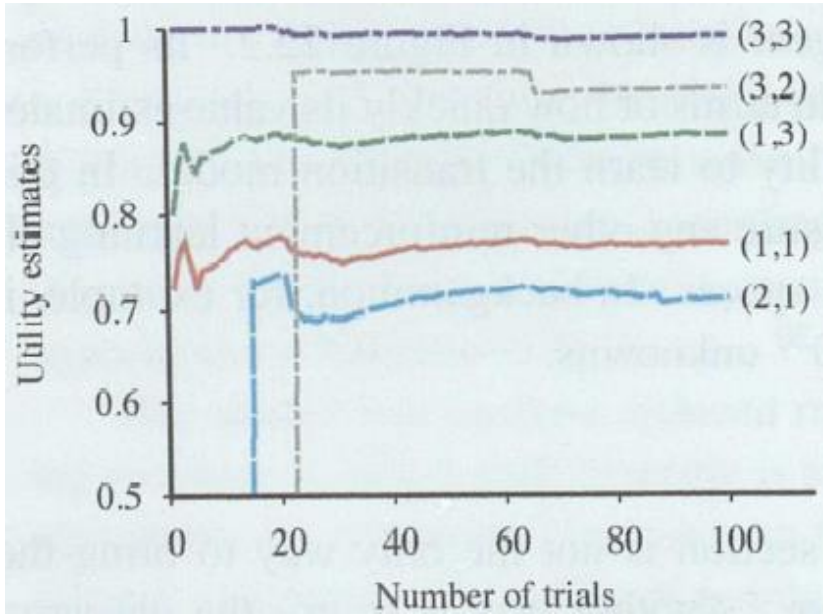
- Exploit, that utility values obey Bellman equations for a fixed policy:

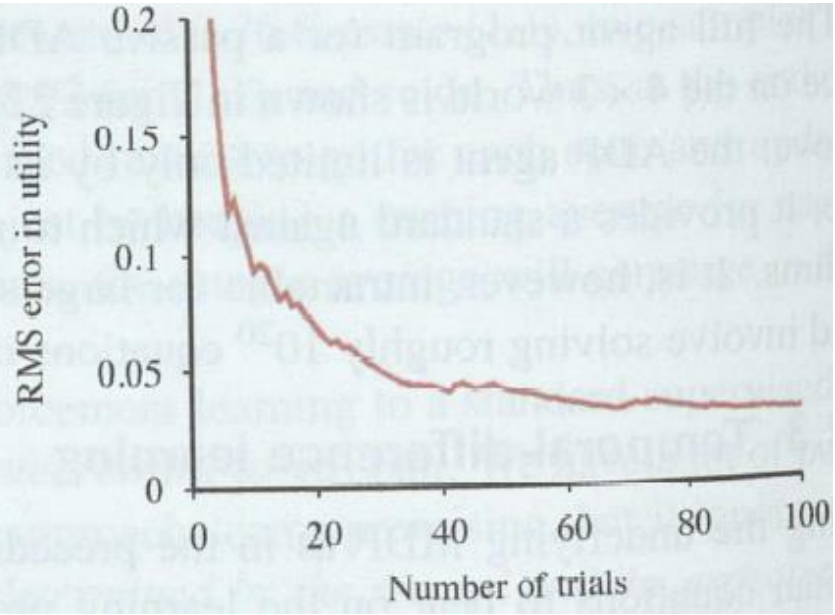
$$U_i(s) = \sum_{s'} P(s' \mid s, \pi_i(s))[R(s, \pi_i(s), s') + \gamma U_i(s')]$$

- Learn the transition model from the state transfers
- Use the observed rewards
- Solve the Bellman equations (but might be intractable in large state spaces)

Experiments with 0-100 trials:

- Left: Utilities for some states
- Right: the root-mean-square error for state (1,1)



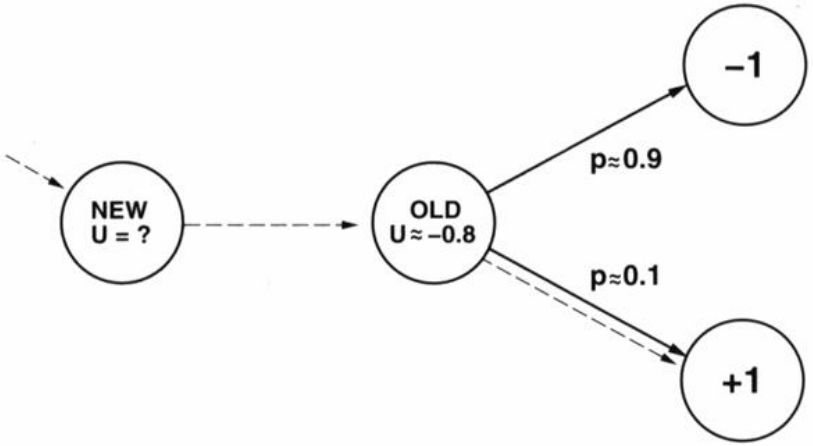


Temporal Difference (TD) Learning

- Combination of advantages of direct and ADP algorithms: Simplicity and consideration of transitions between states
- Example:
  - Utitility of state OLD is -0.8
  - Utilitity of state NEW should be estimated by its direct reward (0) and the utility of the successor OLD (-0.8) and not by the final outcome (dashed line; resulting in +1 as in direct utility estimation)
- General update formula:

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha\,[R(s,\pi(s),s') + \gamma\, U^{\pi}(s') - U^{\pi}(s)]$$

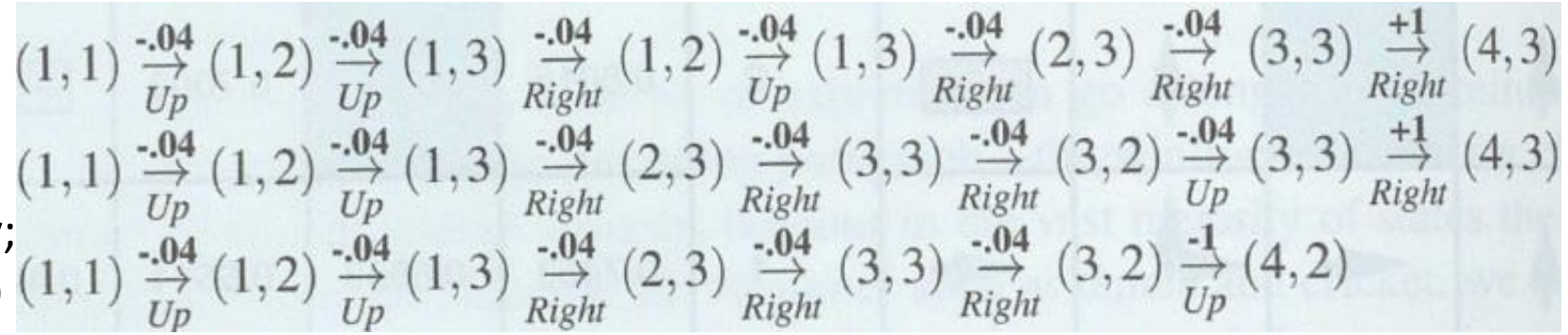
  - α: Learning rate parameter (decreases over time)



Example for TD Learner

Update Formula:

$U^{\pi}(s) \leftarrow U^{\pi}(s) +$
$\alpha\,[R(s,\pi(s),s') + \gamma\, U^{\pi}(s') - U^{\pi}(s)];$
set $\gamma = 1$; $\alpha = 0.9$ (constant for simplicity;
but $\alpha = 1$ for first visit of state)



First sequence (backwards):

(3,3): 1

(2,3): 0.96

(1,3): 0.92

(1.2): 0.88

(1,3): 0.92+0.9(-0.04+0.88-0.92) = 0.92-0.9*0.08 = 0.848

(1,2): 0.88+0.9(-0.04+0.848-0.88) = 0.88-0.9*0.072 = 0.8152

(1,1): 0.7752

Second sequence:

(3,3): 1 + 0.9(1 - 1) = 1

(3,2): 0.96

(3,3): 1 + 0.9(-0.04 + 0.96 - 1) = 1-0.9*0.08 = 0.928

(2.3): 0.888

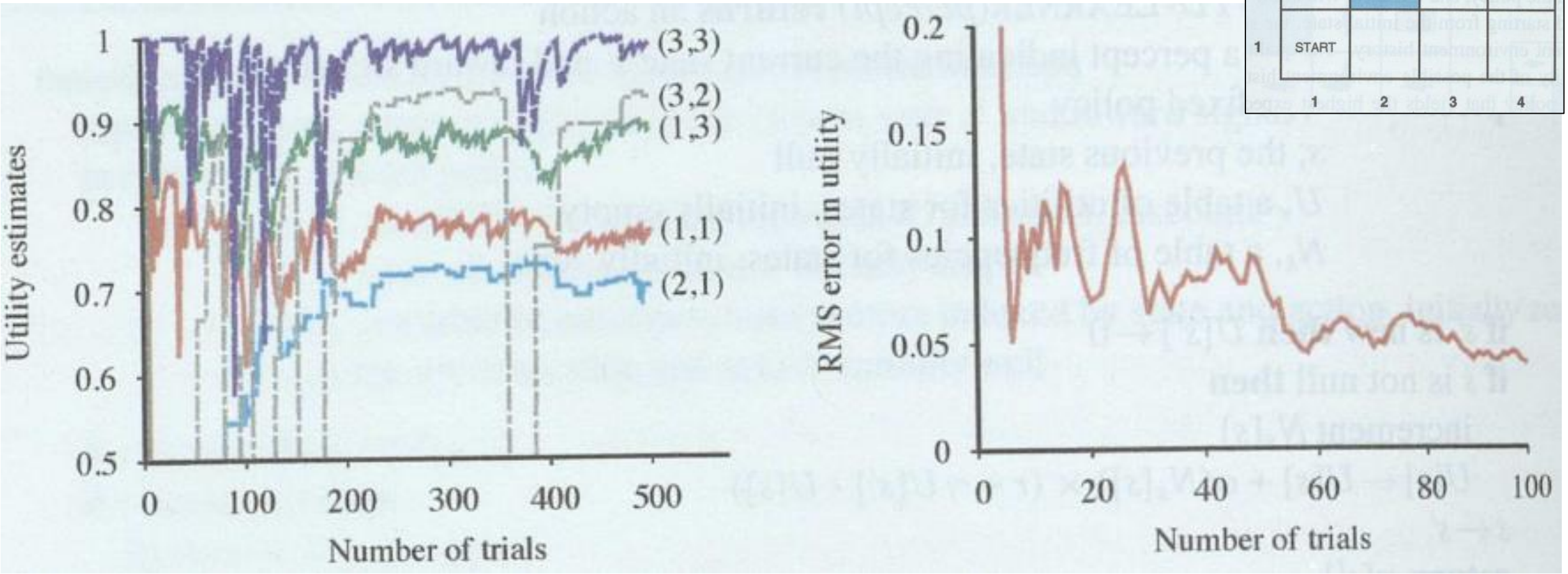
(1,3): 0.848 + 0.9(-0.04 + 0.848 - 0.888) =
0.848 - 0.9*0.08 = 0.776

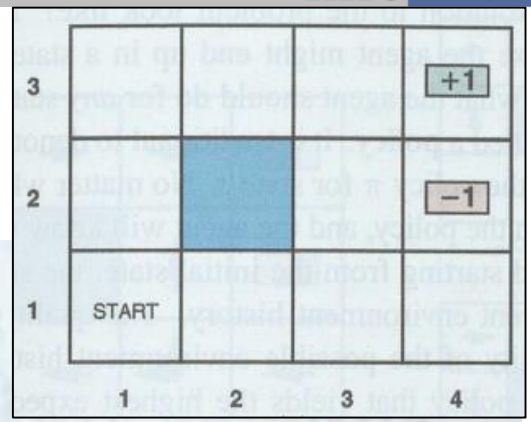
(1,2): 0.8152 + 0.9(-0.04 + 0.776 - 0.8152) =
0.8152 - 0.9*0.0792 = 0.74392

(1,1): 0.7752 + 0.9(-0.04 + 0.74392 - 0.7752) =
0.7752 - 0.9*0.07128 = 0.711048

TD Learning Curves for 4 x 3 World

- Note the changes of utility values for e.g. states (3,3) and (3,2)!
  - ➢ Convergence only with decreasing α





Algorithm for Passive TD Learner

- The algorithm iterates over each episode and each step in an episode
- Note: $\alpha(n)$ is here a function (e.g. 60/(59+n) in last slide)

function PASSIVE-TD-LEARNER(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: π, a fixed policy
    s, the previous state, initially null
    U, a table of utilities for states, initially empty
    N_s, a table of frequencies for states, initially zero

  if s' is new **then** $U[s'] \leftarrow 0$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s]) \times (r + \gamma U[s'] - U[s])$
  $s \leftarrow s'$
  return $\pi[s']$

Properties of Passive TD Learner

- Update formula: $U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(N_s[s])\,[R(s,\pi(s),s') + \gamma\, U^{\pi}(s') - U^{\pi}(s)]$
  - The term $[R(s,\pi(s),s') + \gamma\, U^{\pi}(s') - U^{\pi}(s)]$ is effectively an error signal and the update is intended to reduce the error
  - Update involves only the observed successor state s‘, not the other possible successors, but in the long run, the value converges to an equilibrium, if α decreases over time
    - TD makes only only a single adjustment per observation
    - TD learns slower than ADP agent and shows higher variability
    - But is simpler and requires less computation per observation
    - Does not need a transition model to perform its updates
    - TD can be viewed as a crude but efficient approximation of ADP
- TD can be extended by „pseudoexperiences“ based on its current transition model simulating transitions that might happen
  - In a similar vein ADP can be extended to bound the number of adjustments

Active Reinforcement Learning

- Main issue: Decision what action to take
  - Need for exploration of so far untried actions, otherwise risk of sticking in rather bad local maxima
  - Balancing current best optimal policy against potentially better unknown policy
    - Depends on the number of trials:
      - If infinite, than **GLIE** strategy: „greedy in the limit of infinite exploration“

Active Reinforcement Learning

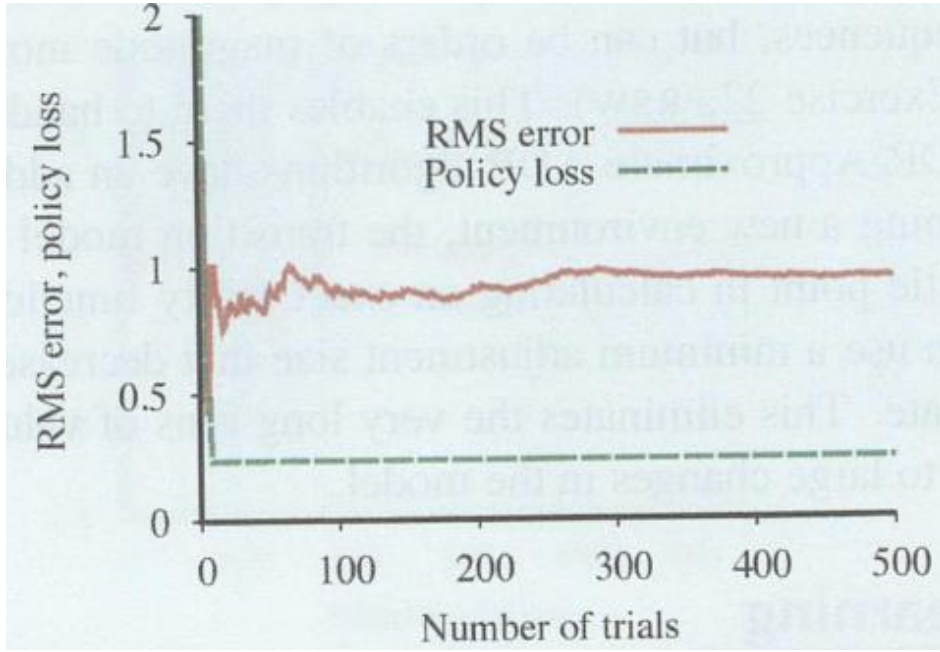
- Update equation for exploration with optimistic estimate:

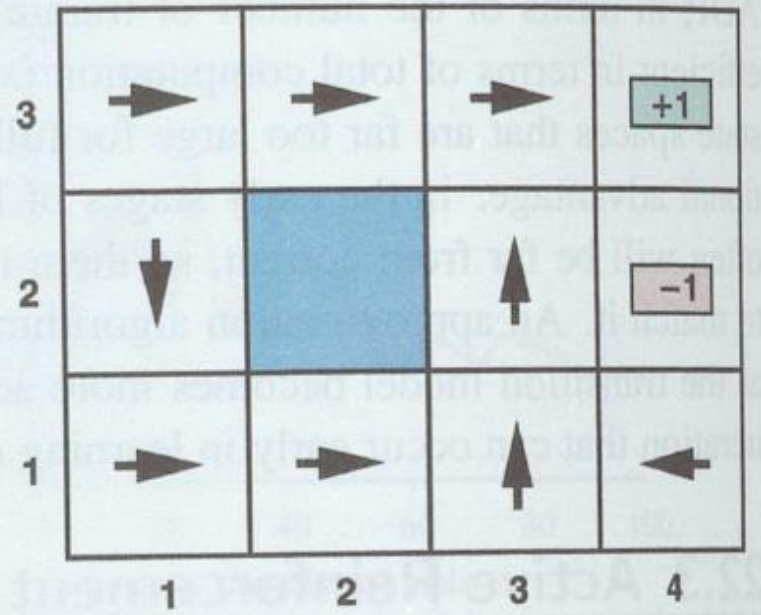
  $U^+(s) \leftarrow \max_a f\left(\sum_{s'} P(s'|s,a)\,[R(s,a,s') + \gamma\, U^+(s')],\ N(s,a)\right)$

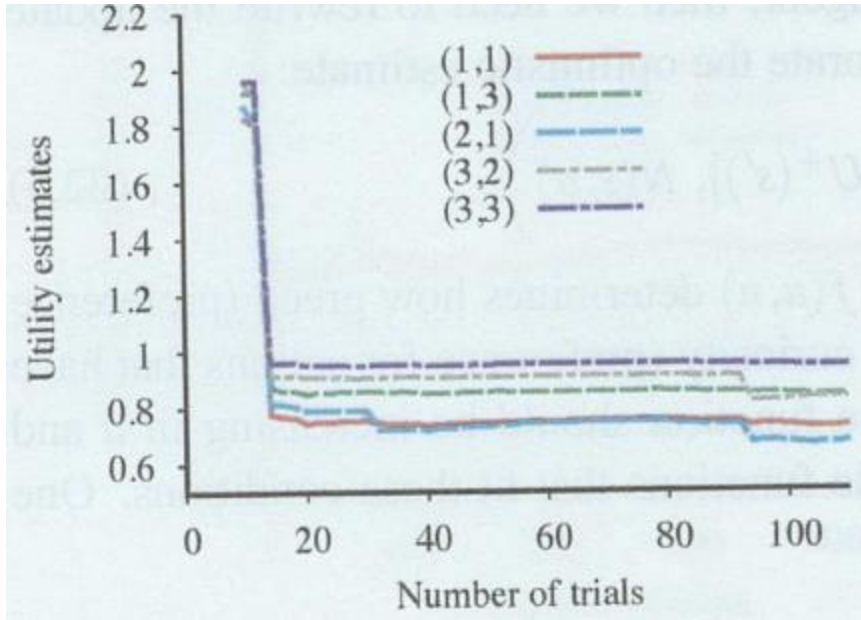
  - With function f(u,n) balancing known utility u against curiosity, i.e preference of actions with low count n, that have been tried rarely
    - e.g. f(u,n) = *if* $n<N_e$ *then* R^+ *else* u
      - R^+ optimistic estimate of best possible reward
      - N_e fixed threshold
  - $U^+(s')$ denotes not just the known utility of state s', but an optimistic estimate
    - e.g. even if state s' has been visited and has low utility U(s'), some unexplored action sequences of s' might lead to a high optimistic estimate of $U^+(s')$

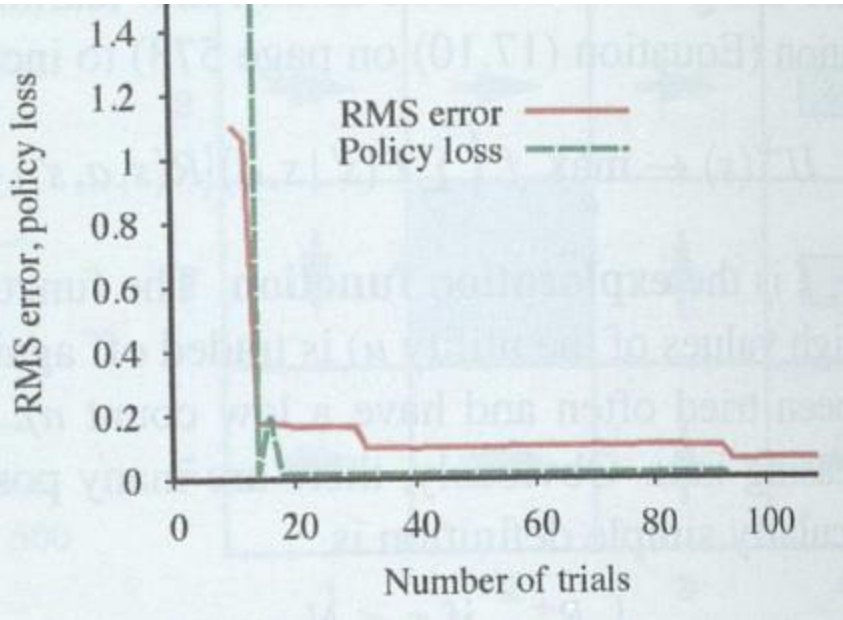
Effect of Exploration for Active Reinforcement Learning

- Above: Example for a run with a greedy strategy without exploration. Agent is stuck in a suboptimal policy (above right)
- Below: Strategy with exploration. After 18 trials the agent has found the optimal strategy and sticks to it after 20 trials (even if utility estimates still have a small error)









Safe Exploration

- In a simulation, negative rewards serve only to improve the model of the world
- In the real world, many actions are irreversible, leading in dead ends or even injury or death
- How to avoid them?
  - Robust control theory: Assume worst case when computinig utilities
    - Might lead to avoid any unknown action
  - Prior knowledge about risks advantageous
    - Also from observation of other agents
  - For autonomous vehicles
    - A special „safety“ policy takes over control when state is estimated as „unsafe“

Temporal-Difference Q-Learning

- Temporal difference learning would need a transition model to balance exploration and current maximal reward
  - Can be avoided by **Q-learning** method
    - Learns an action-utility function Q(s,a) instead of a utility function U(s)
    - $Q(s,a) = \sum_{s'} P(s'|s,a) [R(s,a,s') + \gamma \max_{a'} Q(s',a')]$
    - Resulting TD update function:
      - $Q(s,a) \leftarrow Q(s,a) + \alpha [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
      - Does not contain a transition model
- Q-Learning has a close relative: **SARSA** (State, Action, Reward, State, Action)
  - While Q-Learning needs to know the best action in the follow-up state $\max_{a'} Q(s',a')$
  - SARSA waits until the action is taken and backs up the Q-value for that action:
    - $Q(s,a) \leftarrow Q(s,a) + \alpha [R(s,a,s') + \gamma Q(s',a') - Q(s,a)]$
  - When exploration yields a negative reward, SARSA penalizes the action, Q-Learning not

Exploratory Q-learning Agent

- Update the Q-value in each round
- Chooses the best action balancing Q-value and curiosity with the f-function

function Q-LEARNING-AGENT(*percept*) **returns** an action

inputs: *percept*, a percept indicating the current state s' and reward signal r

persistent: Q, a table of action values indexed by state and action, initially zero

N_{sa}, a table of frequencies for state–action pairs, initially zero

s, a, the previous state and action, initially null

if s is not null **then**

increment $N_{sa}[s,a]$

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

$s,a \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$

return a

Generalization in Reinforcement Learning

- Utility- and Q-functions need a large tabular data structure with an output value for each state or state-action pair
  - Too much for most real-world problems (even backgammon has about 10^{20} states)
- Generalization: **Function approximation** for utility- or Q-function similar to **evaluation function** in games
  - Example: Weighted linear combination of features f_1 .. f_n:
    - $\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$
    - Reinforcement learning algorithm learns the values for the n parameters θ
    - that make $\hat{U}_\theta$ a good approximation of the true utility function U
  - Approximate utility function can be combined with limited look-ahead
- Advantages:
  - Representation of utility or Q-functions for large state spaces
  - Inductive generalization from visited states to unvisited states

Approximating Direct Utility Estimation (1)

- Goal: Compute utility estimates for each state *separately* by a (linear) function with appropriate features
- Example in 4 x 3 world: features are just the x- and y-position the states:
  - $\hat{U}_\theta(x,y) = \theta_0 + \theta_1 x + \theta_2 y$
  - e.g. if $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$ then $\hat{U}_\theta(1,1) = 0.8$
- Given a collection of trials, we can find the best fit (minimizing the squared error) by using linear regression

Approximating Direct Utility Estimation (2)

- Alternate: Learn $\hat{U}_\theta(x,y) = \theta_0 + \theta_1 x + \theta_2 y$ from each trial by updating the parameters (online learning)
  - Write an error function and compute its gradient with respect to the parameters
  - Error: $E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2$ with $u_j(s)$ observed total reward from state s onward in j^{th} trial
  - Gradient update:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha [u_j(s) - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_j}$$ (**Widrow-Hoff rule** or **delta rule**)

  - Delta rule results in three simple update rules for the three parameters:
    - $\theta_0 \leftarrow \theta_0 + \alpha [u_j(s) - \hat{U}_\theta(s)]$
    - $\theta_1 \leftarrow \theta_1 + \alpha [u_j(s) - \hat{U}_\theta(s)]\ x$
    - $\theta_2 \leftarrow \theta_2 + \alpha [u_j(s) - \hat{U}_\theta(s)]\ y$
  - Exxample continued: $u_j(1,1) = 0.4$ and $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$ with $\hat{U}_\theta(1,1) = 0.8$
    - Update for each parameter of $\hat{U}_\theta = (0.5, 0.2, 0.1)$ by $(-\alpha\ 0.4)$ which reduces the error
    - Changes also the values of $\hat{U}_\theta$ for all other states too (generalisation)!

Discussion of Approximating Direct Utility Estimation

- If we replace the 4 x 3 world by a 10 x 10 world with a reward of +1 in (10,10), the update works quite well and is very efficient
- However, if the reward of +1 is placed in state (5,5), the system is not able to learn the utility with a linear function approximator
- Solution (similar to SVMs): Choose appropriate features
  - e.g. add a feature to f_1 = x and f_2 = y like $f_3 = \sqrt{(x - xg)^2 + (y - y_g)^2}$
  - with (x_g , y_g) = coordinates of the goal, i.e. (5,5)

Approximating Temporal-Difference Learning

- In a similar way, parameters for temporal difference learners can be adjusted to reduce the „error“ (difference between successive states):
- For utility estimation with TD learner:

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a,s') + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

- For Q-Value estimation with TD:

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a,s') + \gamma \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)] \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

- Problems with inclusion of active learning and non-linear functions such as neural networks:
  - Parameters can go off to infinity, even for simple cases
  - **Catastrophic forgetting**: Learned capabilities might be generalized in such a way, that they become brittle and are „forgotten“ by later updates
    - Solution: **Experience replay** from former experiences
- Function approximation for learning an environment model (e.g. state transitions) useful
  - Allows the agent to do a look-ahead search to improve its decisions and do simulations

Deep Reinforcement Learning

- Representing utilities and Q-values with linear functions might be very difficult, since the feature construction is not obvious in the general case
- Alternate: Represention of utilities and Q-values with non-linear functions learned by neural networks
  - **Deep Reinforcement Learning** achieved very significant results
    - Playing a range fo video games at an expert level
    - Beating the world champion at GO
    - Training robots to perform complex tasks
  - Problems:
    - Often difficult to get good performance
    - Even successful trained systems may behave very unpredictably if the environment differs even a little from the training data

Reward Shaping

- Real world environments often have very sparse rewards
  - e.g. a robot soccer agent may send millions of motor control commands to its various joints before conceding a goal
    - Working out, what was wrong, is difficult: „Credit assignment“ problem
- Common method, also used for animal training, is **reward shaping**
  - Supply the agent with additional rewards called **pseudorewards**, for making progress
    - e.g. pseudorewards for advancing or contacting the ball
    - Can speed-up learning enormously
      - Risk of maximizing the pseudorewards rather than the true rewards, e.g. standing next to the ball and vibrating causes many contacts with the ball
  - New reward function R‘: **R‘(s,a,s‘) = R(s,a, s‘) + γ Φ(s‘) - Φ(s)**
    - with Φ constructed to reflect any desirable aspects fo the state

Hierarchical Reinforcement Learning

- Long action sequences are difficult to learn. It is much easier by breaking them hierarchically in smaller pieces
  - Example: Scoring a goal in soccer:
    - Obtaining ball possession, passing to a teammate, receiving ball from teammate, running or dribbling toward goal and shooting
      - Multiple ways of obtaining ball possession or shooting
        - Broken down further till the lower-level motor behaviors
- A hierarchical reinforcement learning agent begins with a partial program that outlines a hierarchical structure for the agent‘s behavior
  - Partial-programming language extends ordinary programming language by adding primitives for unspecified choices that must be filled by learning
  - For each high level action, a separate reward should be given

Policy Search (1)

- Remember: A policy π is a function, that maps states to actions
- We are interested in parametrized representations of π
  - For example π could be represented by a collection of parametrized Q-functions:

    $\pi(s) = \text{argmax}_a \hat{Q}_\theta(s,a)$
  - Each Q-function could be linear or nonlinear like a deep neural network
- Policy search adjust the parameters to improve the policy
  - If policy is represented by Q-functions, policy search learns Q-functions
    - Different to Q-Learning, since Q-Learning approximates the optimal Q-function Q*
    - while policy search approximates the optimal policy, which might use different Q-functions (e.g. $\hat{Q}_\theta(s,a) = Q^*(s,a) / 100$ is sufficient for an optimal policy)

Policy Search (2)

- $\pi(s) = \text{argmax}_a \hat{Q}_\theta(s,a) \Rightarrow$ policy is discontinuous due to maximum operator
- Policy search methods often use a differentiable stochastic policy representation $\pi_\theta(s,a)$ which specifies the probability of selecting action a in state s, e.g. the softmax function with a parameter $\beta > 0$ modulating the softness of softmax (high β implies a kind of „hardmax“)

$$\pi_\theta(s,a) = \frac{e^{\beta \hat{Q}_\theta(s,a)}}{\sum_{a'} e^{\beta \hat{Q}_\theta(s,a')}}$$

- For a deterministic policy and a deterministic environment we can follow the empirical gradient with hill-climbing converging to a (local) optimum in policy space
- For a non-deterministic environment (or policy) we would need (too) many trials due to randomness
  - Needs techniques to reduce variability, e.g. by generating trials with a simulator

Apprenticeship Learning

- Some domains are too complex to define a good reward function
  - e.g. reward function for self-driving car:
    - Components: Duration to get to destination, not too fast (risk, speeding tickets), fuel consumption, avoid jostling or accelerating passenger, but slam on brakes in an emergency, and so on
    - Omitting a factor can result in extreme values during learning
- Solution approaches:
  - Extensive testing in simulations and modifying the reward function
  - Seek additional sources of information for reward function, e.g. from expert human driver
    - **Apprenticeship learning** studies the process of learning how to behave well given observations of expert behavior

Apprenticeship and Inverse Reinforcement Learning

- Two approaches for apprenticeship learning
  - **Imitation learning:** Apply supervised learning in the observed state-action pairs to learn a policy π (s)
    - Some success in robotics, but suffer from the problem of brittleness
    - Learner does not understand, why an action should be performed
  - **Inverse reinforcement learning:** Try to work out from observed behavior (state action pairs) what reward function the expert is maximizing
    - Learning rewards by observing a policy rather than learning a policy by observing reward
    - Many inverse RL algorithms in literature
      - One of the simplest is called **feature matching**

Feature Matching

- Feature matching assumes that the reward function can be written as a weighted linear combination of features:
  - $R_\theta(s,a,s') = \sum_{i=1}^{n} \theta_i f_i(s,a,s') = \theta \cdot f$
    - For example, the features in the driving domain could include speed, speed in excess of speed limit, accelaration, proximity to nearest obstacle etc.
  - Thus, from a given policy, the expected features can be derived
  - We need to find values for the parameters θ_i such that the feature expectations of the policy induced by the parameter values match those of the expert policy on the observed trajectories.
    - This can be done by an iterative search for an optimal policy, that is close to the policy of the expert compared by the resulting feature expectations

Applications of Reinforcement Learning

- Game Playing: Very successful
  - Alpha Go combined learning both a value function and a Q-function that guided search by predictin which moves are worth exploring
- Robot Control:
  - e.g. radio-controlled helicopter flight