Introduction to Multi-Agent Programming

15. Learning in Multi-Agent Systems (Part B)

Reinforcement Learning, Hierarchical Learning, Joint-Action Learners

Alexander Kleiner, Bernhard Nebel

Contents

Reinforcement Learning

- Learning from interaction with an external environment or other agents
- Goal-oriented learning
- Learning and making observations are interleaved
- Process is modeled as MDP or variants

Key Features of RL

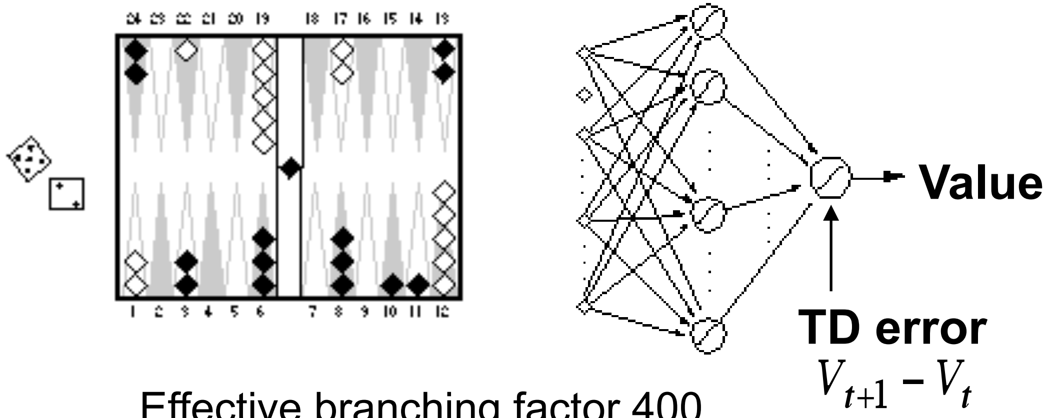
- Learner is not explicitly told which actions to take
- Possibility of delayed reward (sacrifice short-term gains for greater long-term gains)
- Model-free: Models are learned online, i.e., have not to be defined in advance!
- Trial-and-Error search
- The need to exploit and explore, i.e., to perform the best known action or any arbitrary action …

Some Notable RL Applications

- TD-Gammon: Tesauro
  - world's best backgammon program
- Elevator Control: Crites & Barto
  - high performance down-peak elevator controller
- Dynamic Channel Assignment: Singh & Bertsekas, Nie & Haykin
  - high performance assignment of radio channels to mobile telephone calls
- …

Some Notable RL Applications
TD-Gammon

Tesauro, 1992–1995



Action selection by 2–3 ply search

Effective branching factor 400

Start with a random network

Play very many games against self

Learn a value function from this simulated experience

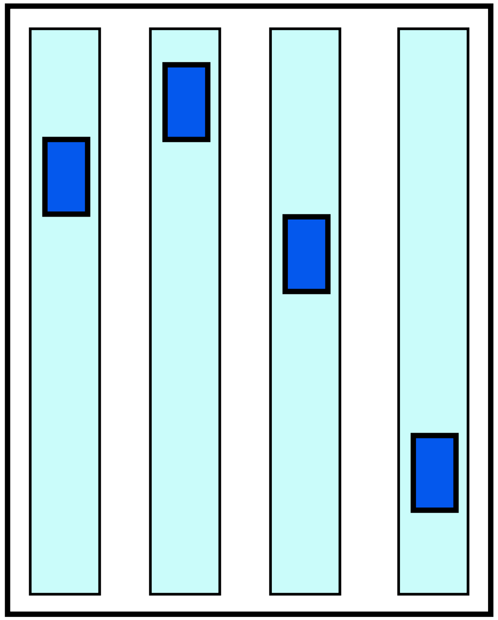
This produces arguably the best player in the world

Some Notable RL Applications

Elevator Dispatching

Crites and Barto, 1996

10 floors, 4 elevator cars



STATES: button states; positions, directions, and motion states of cars; passengers in cars & in halls

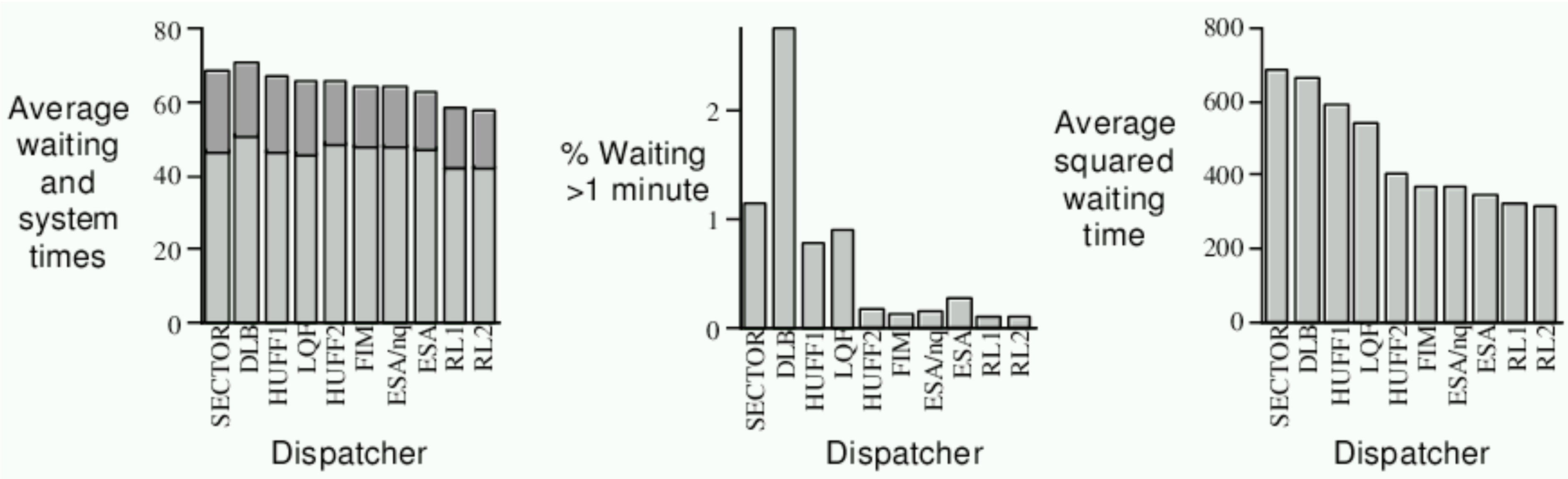
ACTIONS: stop at, or go by, next floor

REWARDS: roughly, –1 per time step for each person waiting

Conservatively about 10^{22} states

Some Notable RL Applications

Performance Comparison Elevator Dispatching



Q-Learning (1)

- Very common Reinforcement Leaning method
- Maintains a table of Q-values
  - Q(s,a) – "what is the outcome of action *a* executed in state *s*"?
- Since values are with respect to states and actions, no explicit transition model *T* needed
- Updates are performed with a step size parameter in order to prevent value overwriting when learning different traces
- Converges to the optimum Q-values with probability 1

Q-Learning (2)

- At time *t* the agent performs the following steps:
  - Observe the current state s_t
  - Select and perform action a_t
  - Observe the subsequent state s_{t+1}
  - Receive immediate payoff r_t
  - Adjust Q-value for state s_t

Q-Learning (3)
Update and Selection

- Update function:

$$Q_{k+1}(s_t, a_t) := (1-\alpha)\, Q_k(s_t, a_t) + \alpha \left[R(s_t, a_t) + \gamma \max_{a \in A} Q_k(s_{t+1}, a_{t+1}) \right]$$

- Where k denotes the version of the Q function, and α denotes a learning step size parameter that should decay over time

- Intuitively, actions can be selected by:

$$\pi(s_t) = \underset{a \in A}{argmax}\, Q(s_t, a)$$

Q-Learning (4)
Algorithm

Initialise $Q(s,a)$ arbitrary for all $s \in S$ and $a \in A$

Repeat

select best action a_t *with the greedy policy:*

$$a_t = \pi(s_t) = \underset{a \in A}{argmax}\, Q(s_t, a)$$

apply a_t *in the world and observe* s_{t+1} and immediate reward r_t*:*

$$s_t \rightarrow s_{t+1}$$

$$r_t$$

adapt the value function for state s_t

$$Q_{k+1}(s_t, a_t) := (1-\alpha)\, Q_k(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a \in A} Q_k(s_{t+1}, a_{t+1}) \right]$$

Until (Q_{k+1}-$Q_k < \varepsilon$) or (s is terminal)

The Exploration/Exploitation Dilemma

- Suppose you form estimates

$$Q_t(a) \approx Q^*(a)$$

action value estimates

- The greedy action at time t is:

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

$$a_t = a_t^* \Rightarrow \text{exploitation}$$

$$a_t \neq a_t^* \Rightarrow \text{exploration}$$

- You can't exploit all the time; you can't explore all the time
- You can never stop exploring; but you should always reduce exploring

e-Greedy Action Selection

- Greedy action selection:

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

- e-Greedy:

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

  – Continuously decrease of ε during each episode necessary!

→ the simplest way to try to balance exploration and exploitation

Eligibility Traces (1)

- Convergence speed of Q-Learning and other RL methods can be improved by eligibility traces
- Idea: simultaneous update of all *Q* values of states that have been visited within the current episode
- A whole trace can be updated from the effect of one step
- The influence of states on the past is controlled by the parameter λ
- Q-Learning with eligibility traces is denoted by $Q(\lambda)$

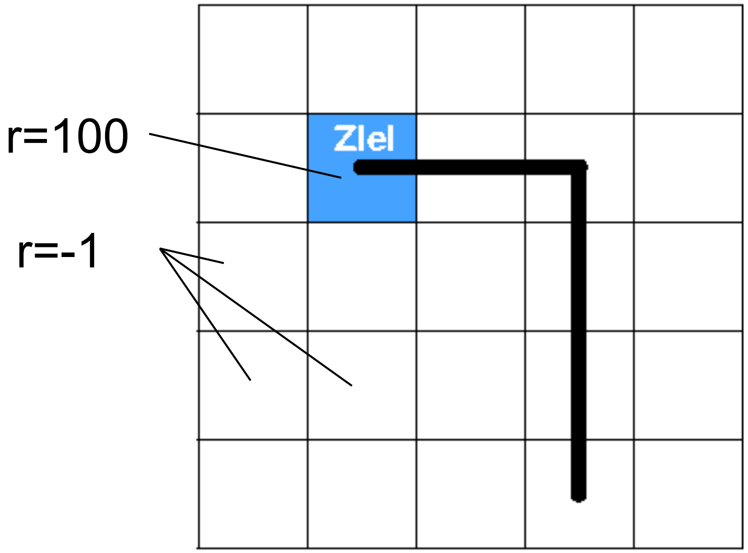
Eligibility Traces (2)

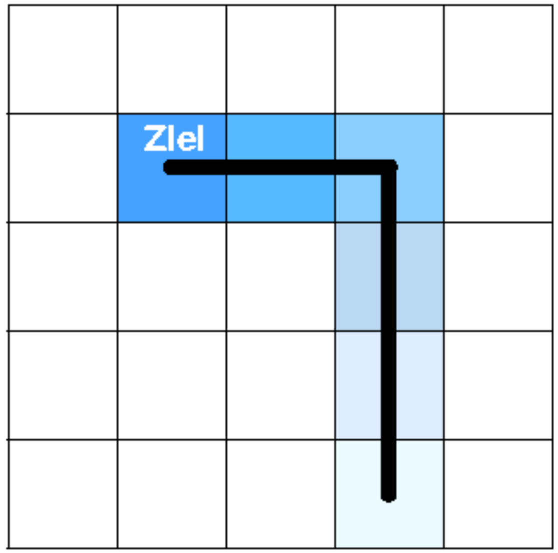
- An eligibility trace defines the state-action pair's responsibility for the current error in Q-values and is denoted by e(s, a)
  - e(s, a) is a scalar value and initialized with 0
- After observing state s and selecting action a, e(s,a) is updated for every Q value according to:

$$\forall \hat{s} \in \mathcal{S}\ \hat{a} \in \mathcal{A} \qquad e(\hat{s},\hat{a}) \quad \leftarrow \lambda\gamma e(\hat{s},\hat{a}) + \begin{cases} 1 & \text{if } \hat{s} = s \text{ and } \hat{a} = a \\ 0 & \text{otherwise} \end{cases}$$

- After each action execution, we update the whole Q-table by applying the standard update rule, however with step-size e(s,a)*ɑ instead of ɑ
- Note that this can be implemented mach faster by keeping all states visited during an episode in memory and applying the update to only those

Eligibility Traces (3)





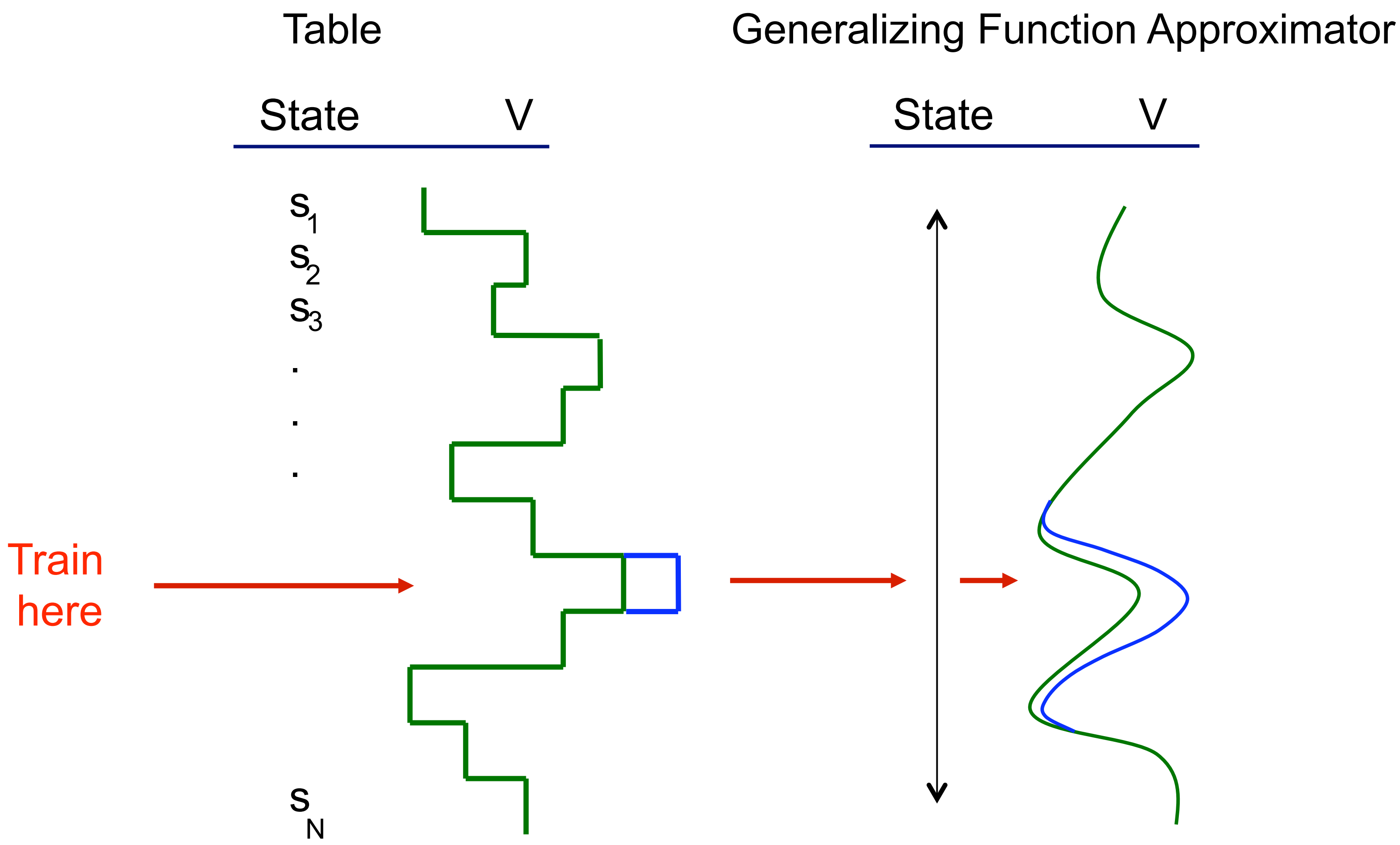
Normal Q-Learning:
Slow update, after each step only one *Q* value is updated

Learning with eligibility traces:
Updated all *Q* values of states that have been visited within the current episode

Function approximation
Motivation

- RL infeasible for many real applications due to curse of dimensionality: |S| too big.
  - Memory limit
  - Time for learning is limited, i.e. impossible to visit all states
- FA may provide a way to "lift the curse:"
  - Memory needed to capture regularity in environment may be << |S|
  - No need to sweep thru entire state space: train on N "plausible" samples and then generalize to similar samples drawn from the same distribution
- Commonly used with Reinforcement Learning:
  - Artificial Neuronal Networks (ANNs)
  - Tile Coding
- FA: Compact representations of S X A -> R, providing a mapping from action-state correlations to expected reward
- Note: RL convergence guarantees are all based on look-up table representation, and do not necessarily hold with function approximation!

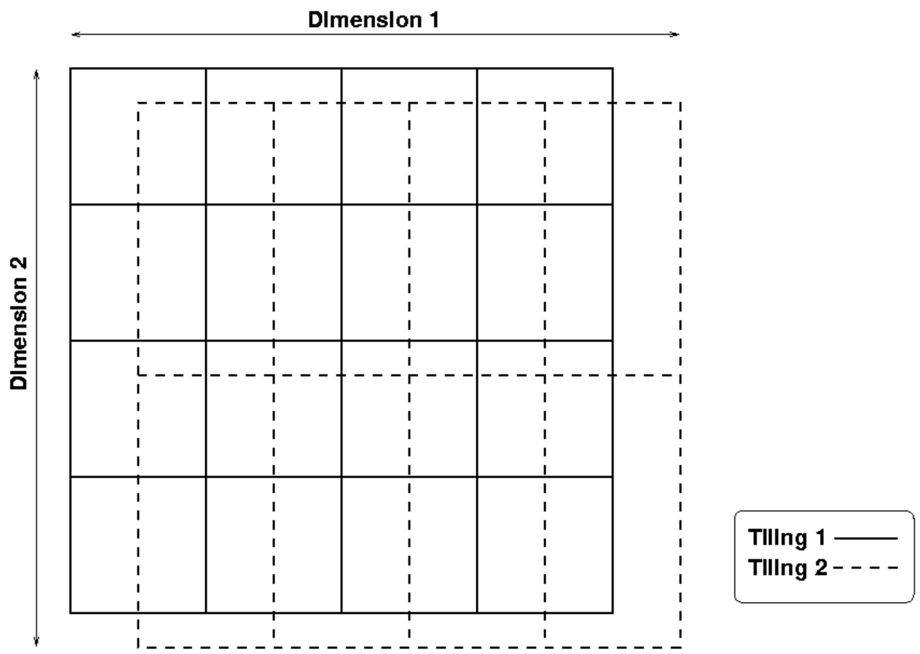
Function approximation
Example



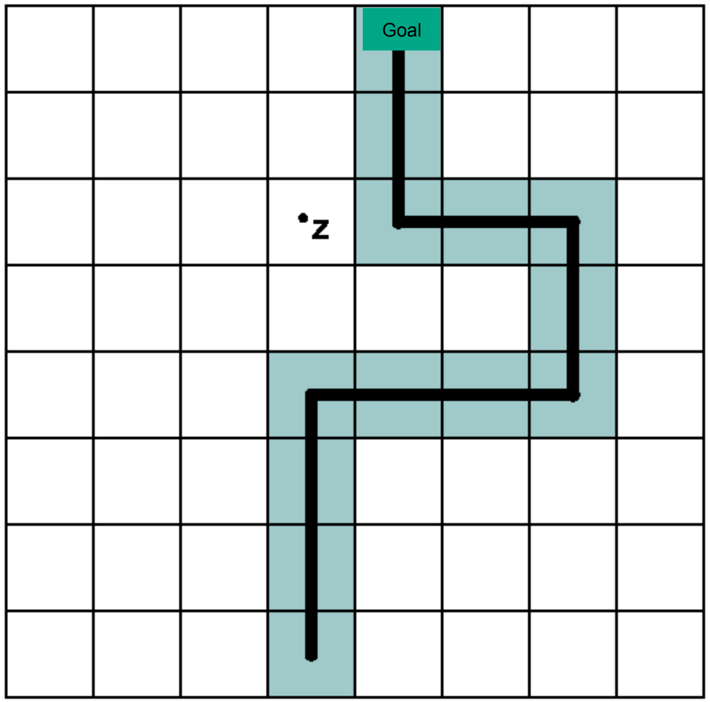
Function approximation

Tile Coding

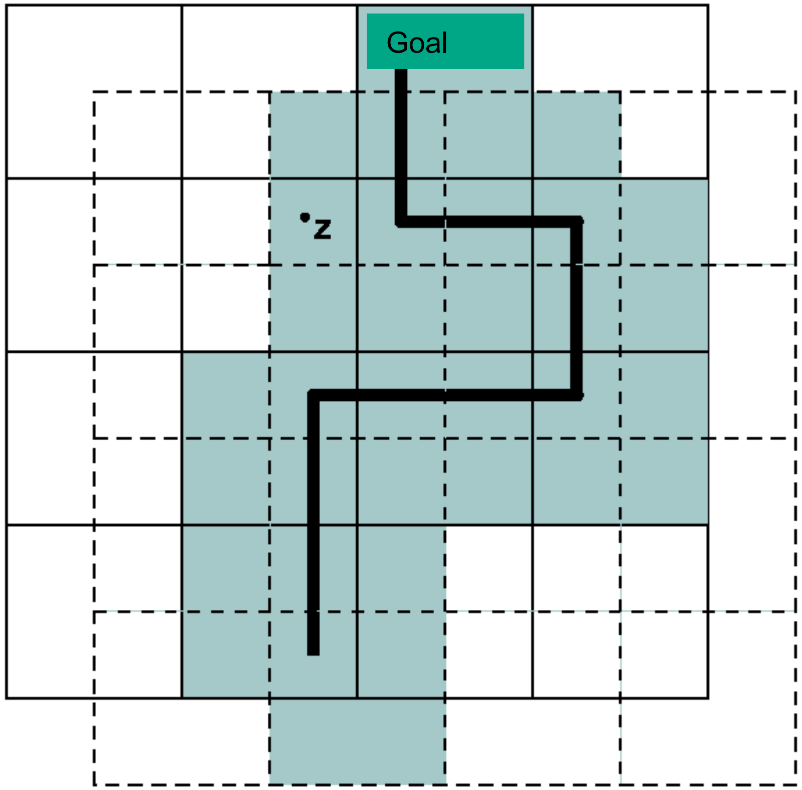
- Discretizations that differ in offset and size are overlaid with each other
- The values of each cell are weights
- Q(s,a) = Sum of the weights of all tiles activated by (s,a)



Look-up table vs. Tile Coding



Look-up table



Tiling with 2 discretizations

Tile Coding – Memory reduction

- Use many tilings with different offset
- Combine only correlating variables within a single tiling
  - Note variables are taken from the state **and** action vector
- Example:
  - 12 variables, 20 discretization intervals:
    - 20^{12} values in memory
  - Combining 4 correlating variables, each:
    - $3 * 20^4$ values in memory
  - 5 discretization intervals, but 24 tilings instead of 3:
    - $24 * 5^4 = 15000$ values in memory

Tile Coding vs. ANNs

- *Function approximation with tile coding*
  - is linear (good convergence behavior!)
  - Mostly explicit knowledge representation
    - Unlikely to overwrite already learned knowledge
    - Easier to visualize
  - Expert knowledge about correlations needed

- *Function approximation with ANNs*
  - Non-linear: convergence can be a problem
  - Implicit knowledge representation
    - Learned knowledge can be "deleted"
    - Unreadable by human beings
  - Automatic learning of correlation

Hierarchical Learning

- Simultaneous acting and learning on multiple layers of the hierarchy
- Basic idea:
  - Sub-tasks are modelled as single MDPs
  - Actions on higher layers initiate Sub-MDPs on lower layers
- However, MDP model requires actions to be executed within discrete time steps

→ Usage of Semi Markov Decision Processes (SMDPs)

SMDPs I

- In SMDPs, actions are allowed to continue for more than one time step
- SMDPs are an extension to MDPs by adding the time distribution F
  - *F* is defined by *p(t |s, a),* and returns the probability of reaching the next SMDP state after time *t*, when behavior *a* is taken in state *s*
  - Q-Learning has been extended for learning in SMDPs
  - The method is guaranteed to converge when similar conditions as for standard Q-Learning are met

SMDPs II

- The update rule for SMDP Q-Learning is defined by:

$$Q_{k+1}(s_t, a_t) := (1-\alpha) Q_k(s_t, a_t) + \alpha \left[r + \gamma^t \max_{a \in A} Q_k(s_{t+1}, a_{t+1}) \right]$$

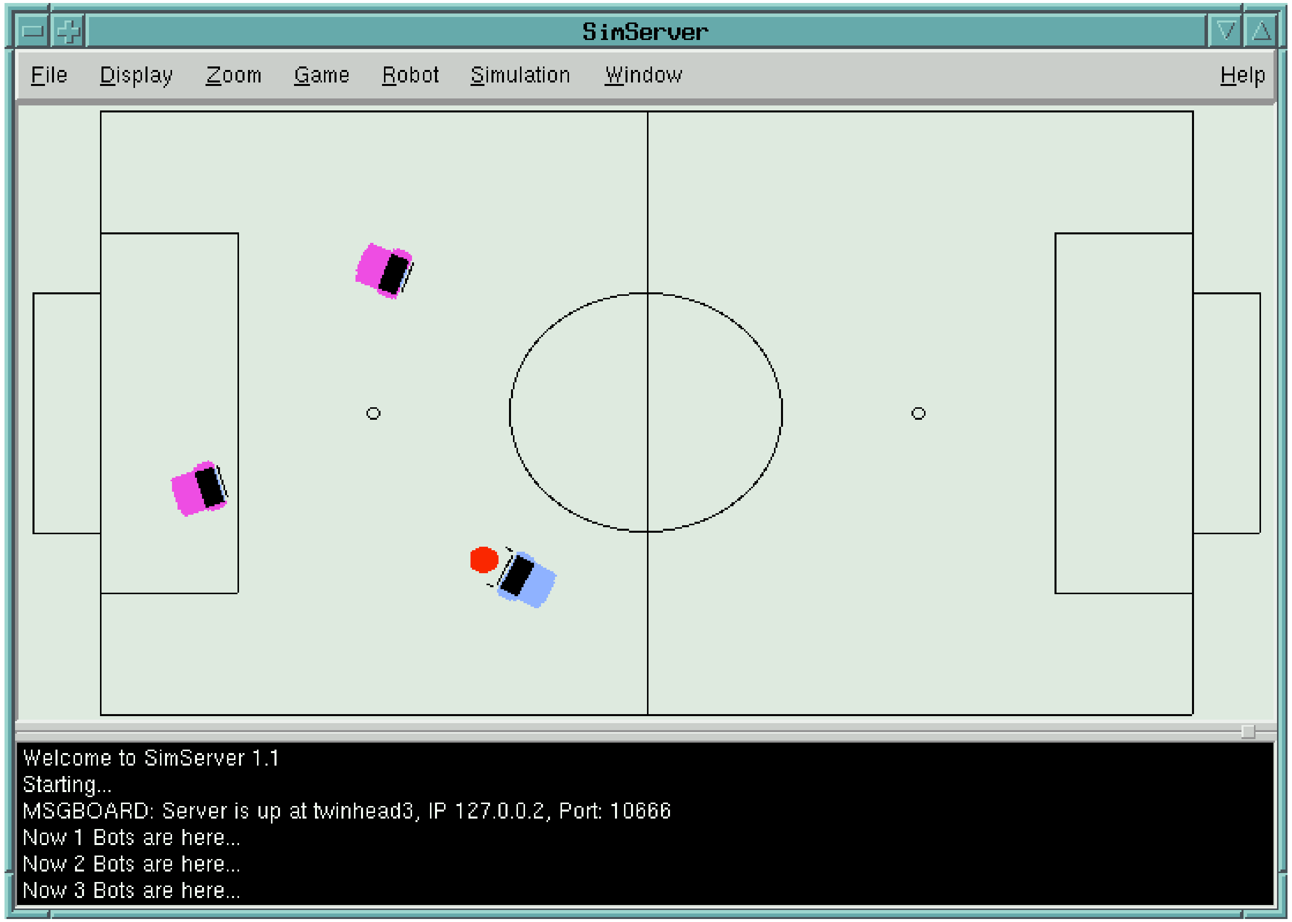
- Where *t* denotes the sampled time of executing the behavior and *r* its accumulated discounted reward received during execution
- Like the transition model *T*, the time model *F* is implicitly learned from experience online

Case Study: RL in robot soccer

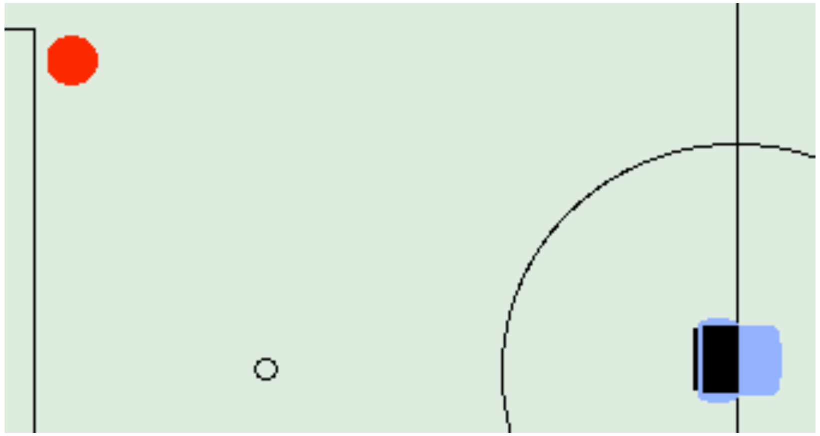
- World model generated at 100Hz from extracted position data, e.g., ball, player, and opponent position, …
- Stochastic actions: turn left/right, drive forward/backward, kick
- RL parameters: $\gamma=1.0$ (finite horizon), $\alpha=0.1$ (small since actions are very stochastic), $\varepsilon=0.05$ (small since traces are comparably long), $\lambda=0.8$ (typical value)
- World model serves as basis for the action selection
  - Shoot goal, dribbling, etc.
  - Actions/Behaviors are realized by modules that directly send commands to the motors
- Goals:
  - Learning of single behaviors
  - Learning of the action selection

Case Study: RL in robot soccer

Acceleration of learning with a simulator



Learning of behaviours
Example "ApproachBall" I



- State space: Angle and distance to ball, current translational velocity
- Actions: Setting of translational and rotational velocities

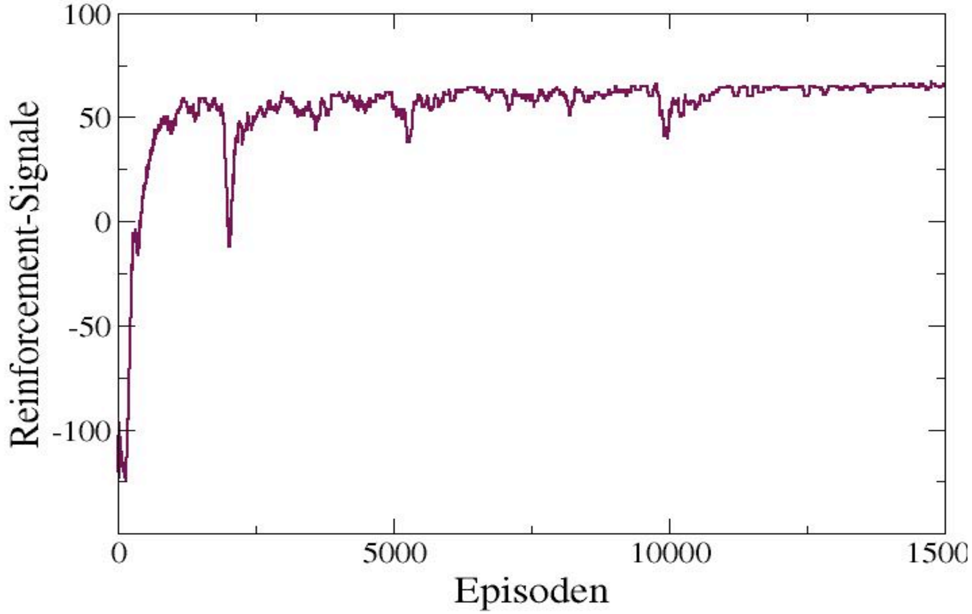
Learning of behaviours

Example "ApproachBall" II

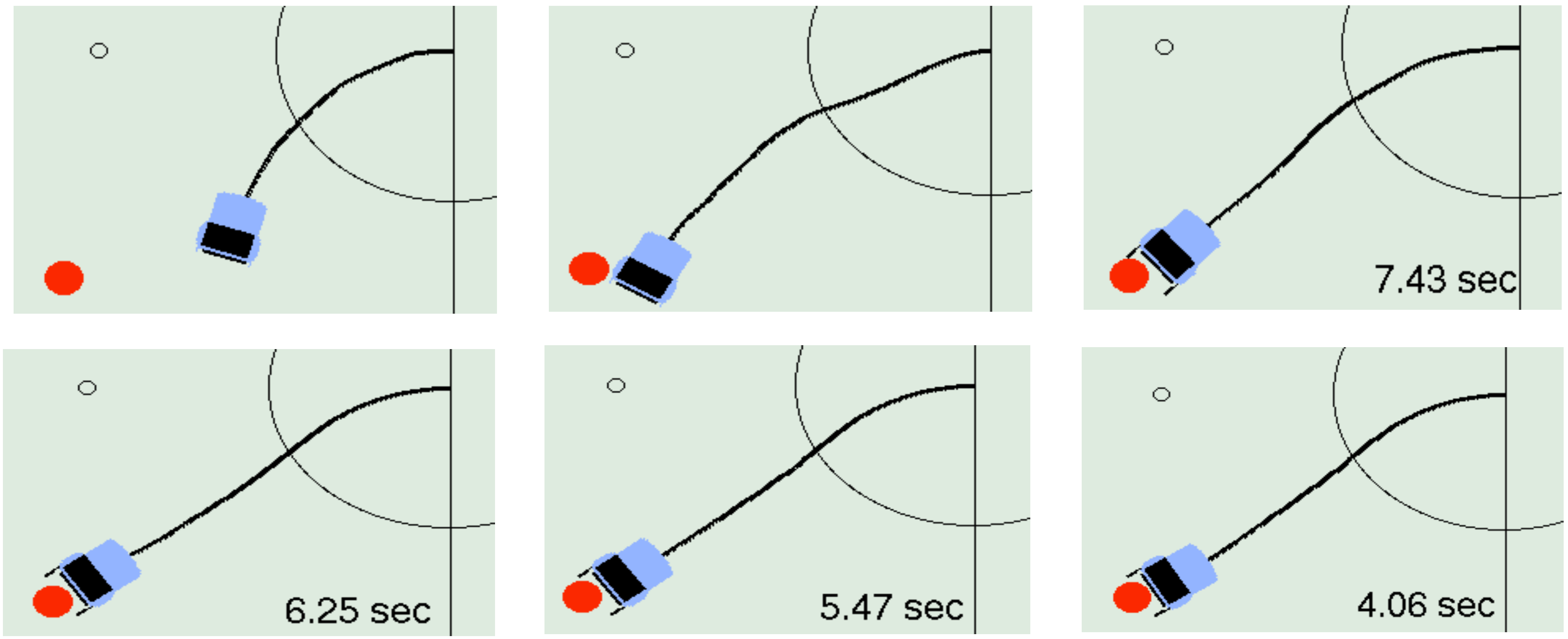
- Reward function:
  - Modelled as MDPs
  - +100: termination if the player touches the ball with reduced velocity of if stops close to and facing the ball
  - -100: termination if the ball is out of the robot's field of view or if the player kicks the ball away
  - -1: else

Learning performance

- *x-axis*
  - Time (# of episode)
- *y-axis:*
  - averaged rewards per episode (smoothed)
- Successful playing after 800 episodes



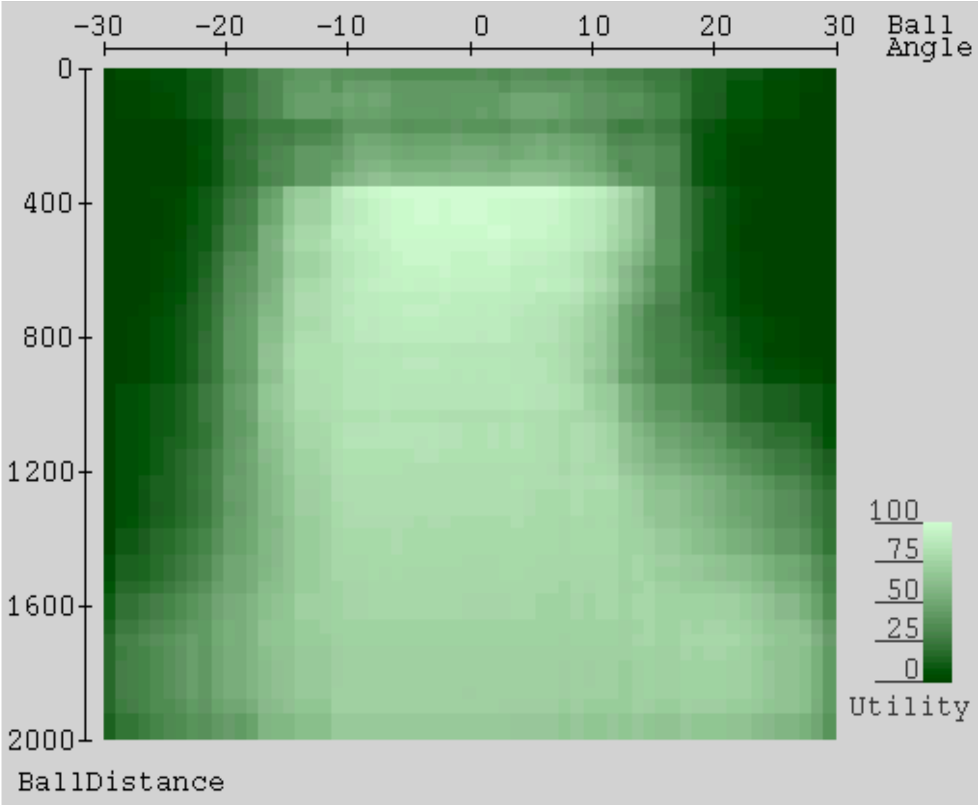
Learning after some steps



The behaviour after 10, 100, 500, 1000, 5000 and 15000 episodes

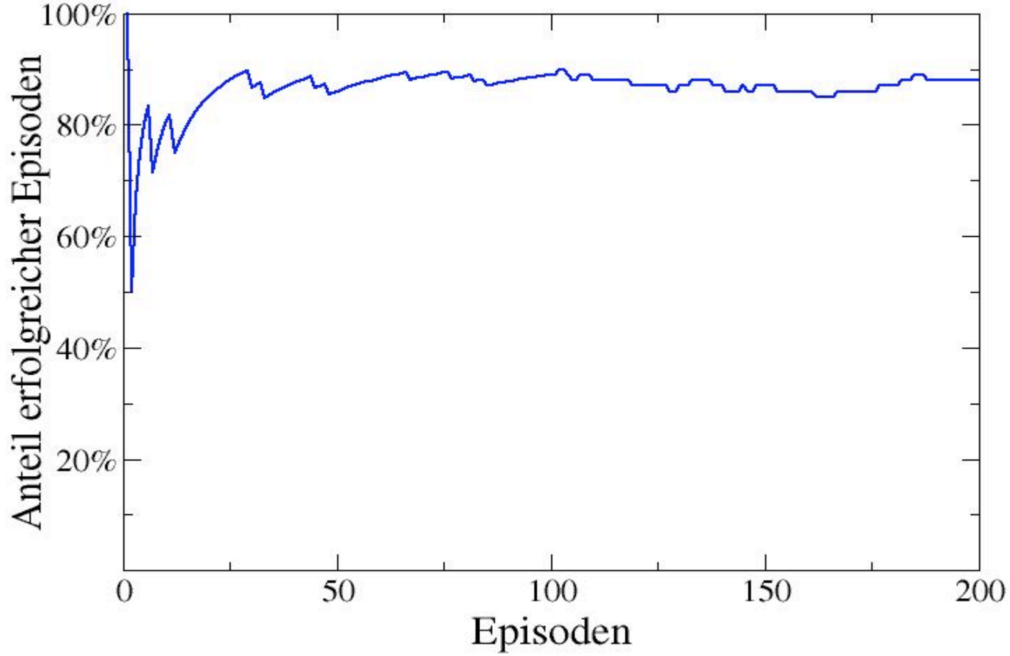
Visualization of the value function

- x-axis: Ball angle
- y-axis: Ball distance
- for a translational velocity of 1 m/s

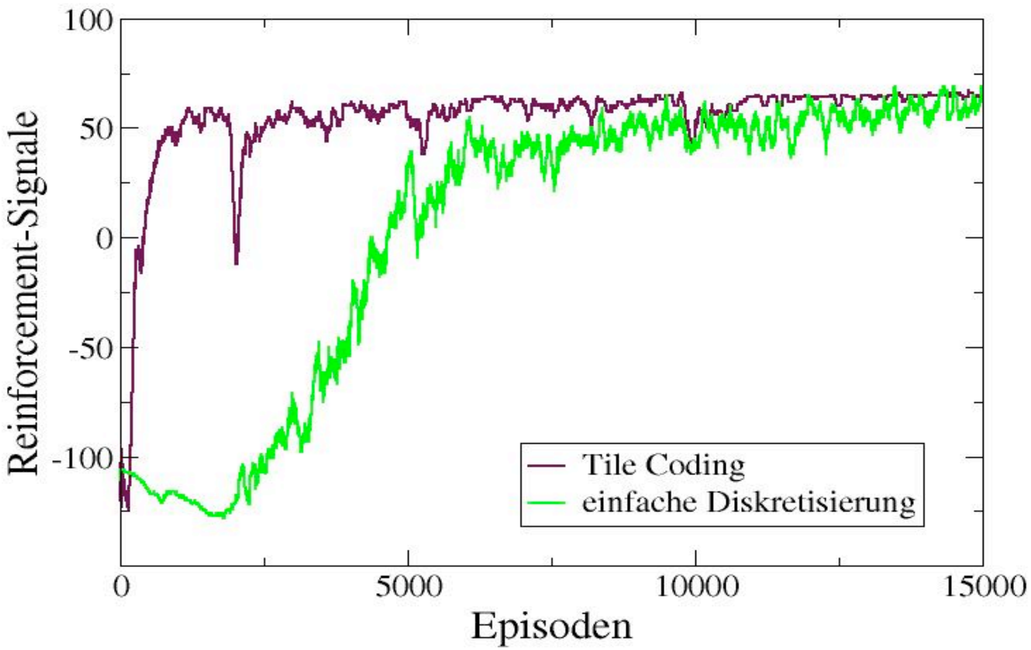


Transfer on the real robot platform

Total success rate of 88 %.

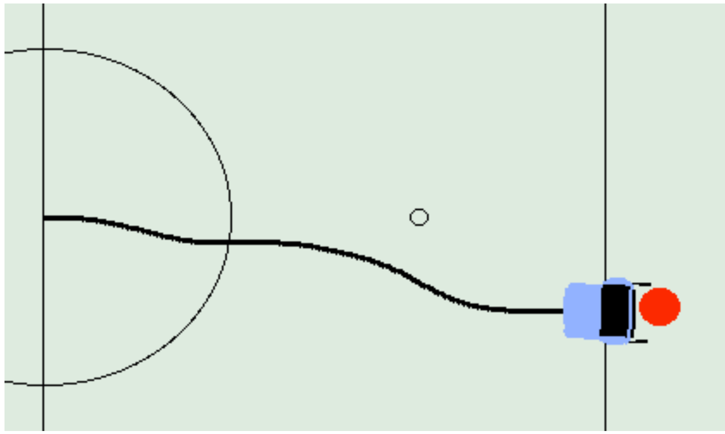


Comparing look-up table and tile coding based discretization

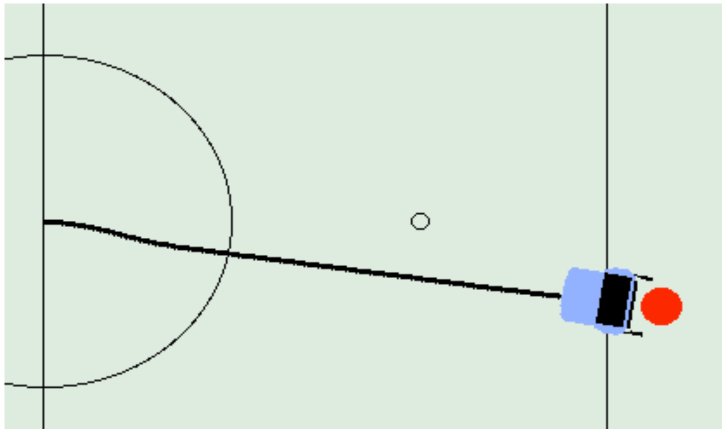


- Tile coding leads to more efficient learning

Comparing look-up table and tile coding based discretization



look-up table



tile coding

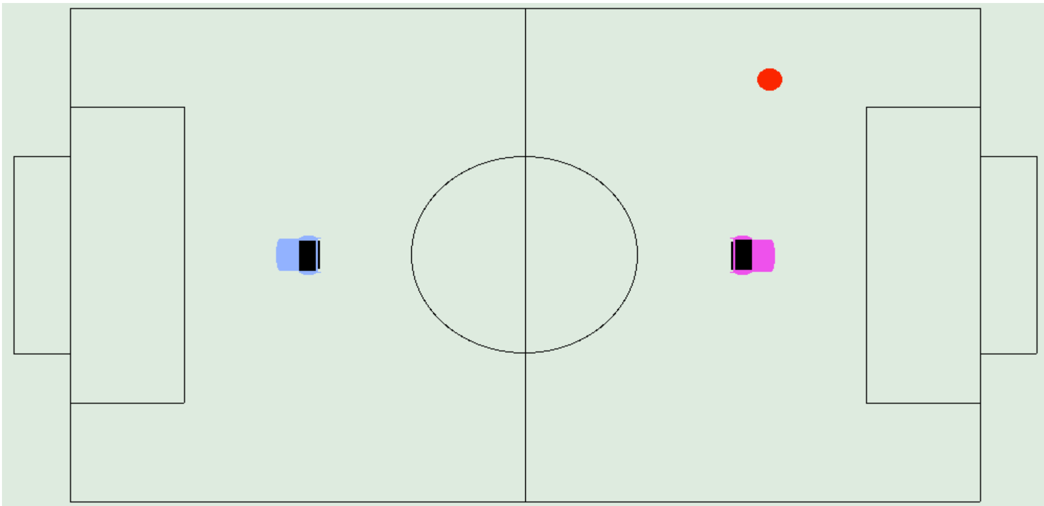
The resulting behaviour after learning:

Function approximation leads to smoother execution

Learning Action Selection

- With an appropriate set of trained behaviours, a complete soccer game can be played
- Trained behaviours:
  - SearchBall, ApproachBall, BumpAgainstBall, DribbleBall, ShootGoal, ShootAway, FreeFromStall
- Finally, the right selection of behaviours within different situations has to be learned

Example:
Playing against a hand-coded CS-Freiburg player (world champion 98/00/01)



- State space: Distance and angle to goal, ball, and opponent
- Actions: Selection of one of the listed behaviours

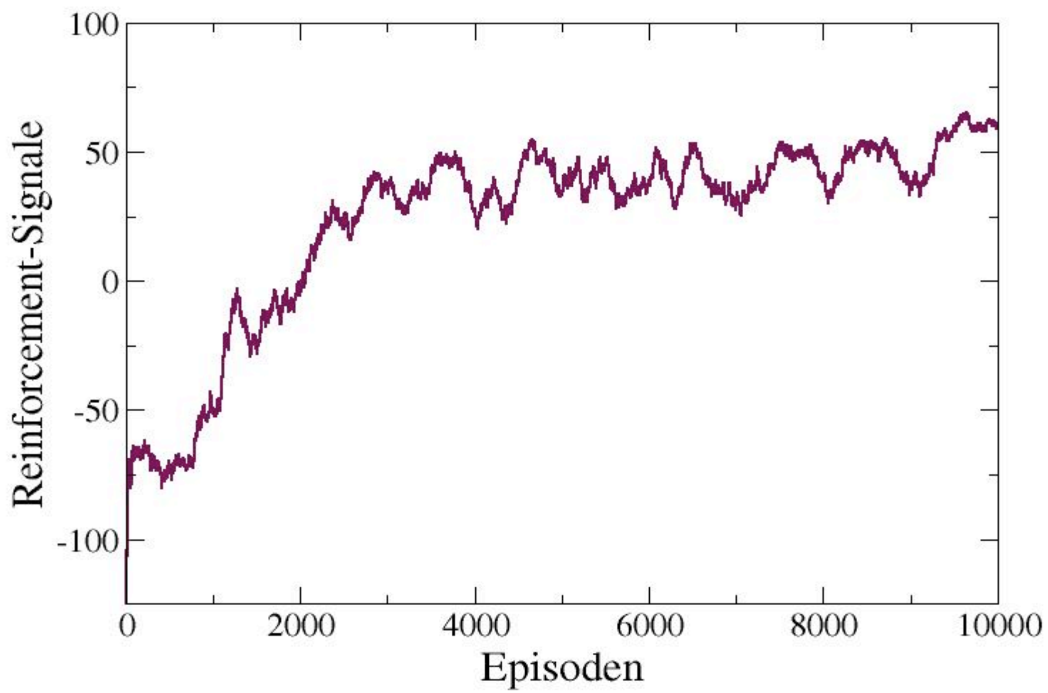
Example:

Playing against a hand-coded CS-Freiburg player (world champion 98/00/01)

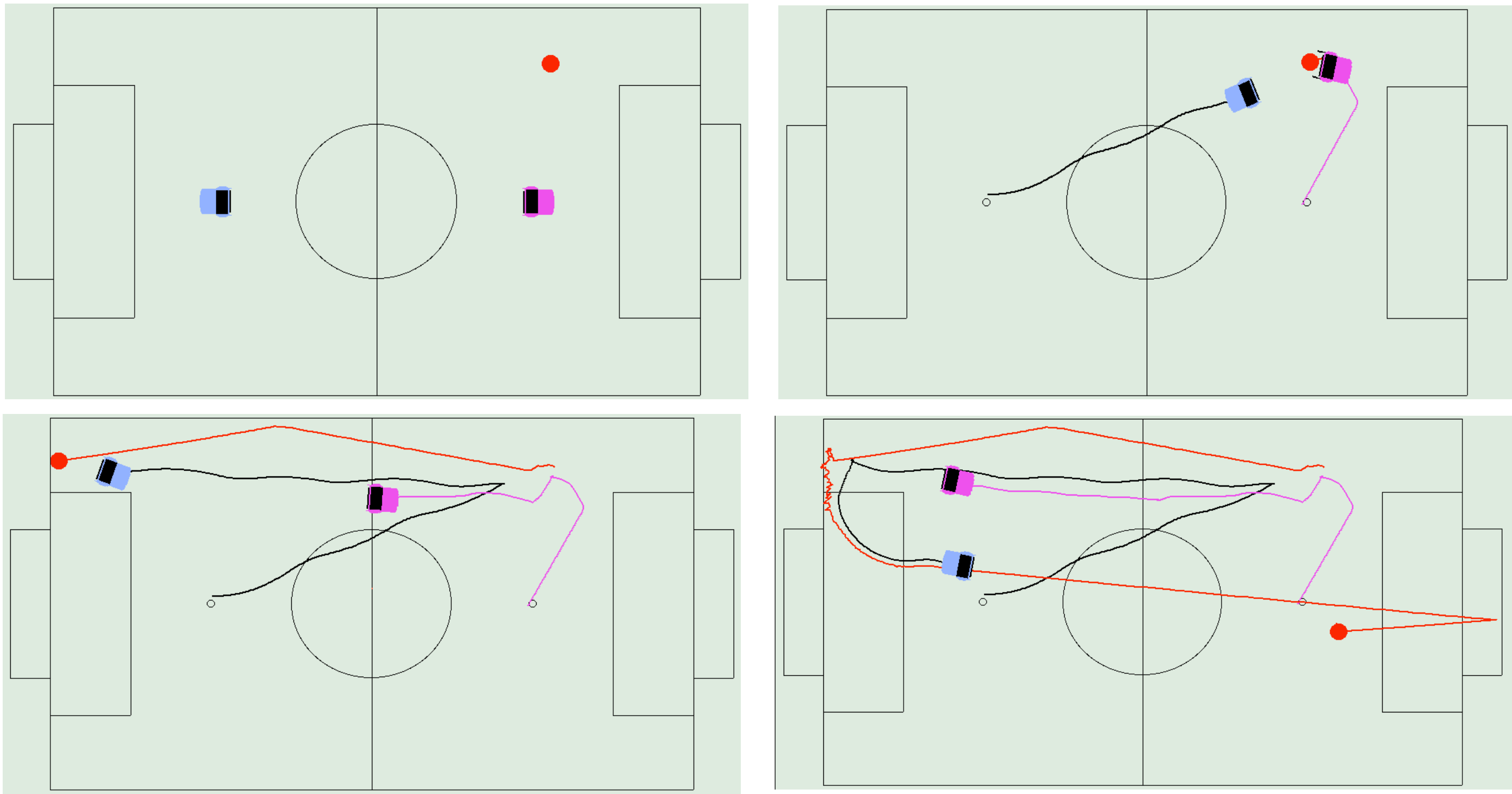
- Modelled as SMDPs
- Reward function:
  - +100 for each scored goal
  - -100 for each received goal
  - -1 for each passed second

Learning performance

- Learning on both layers
  - Successful play after 3500 episodes



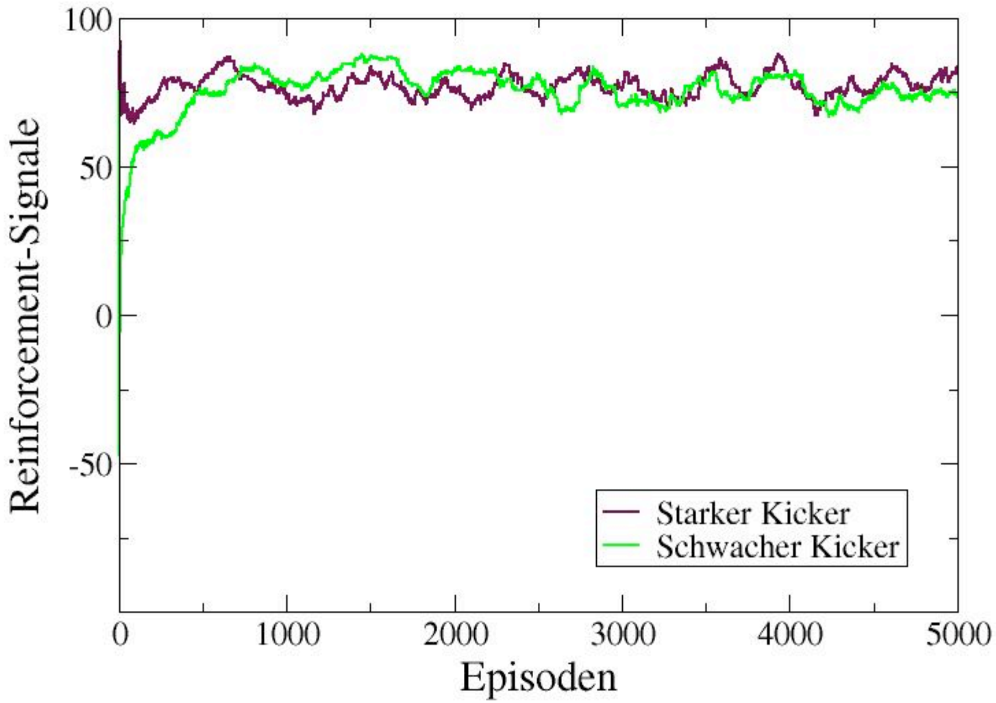
One example episode



Blue: Learner, Pink: Hard-coded

Adaption to sudden changes/defects

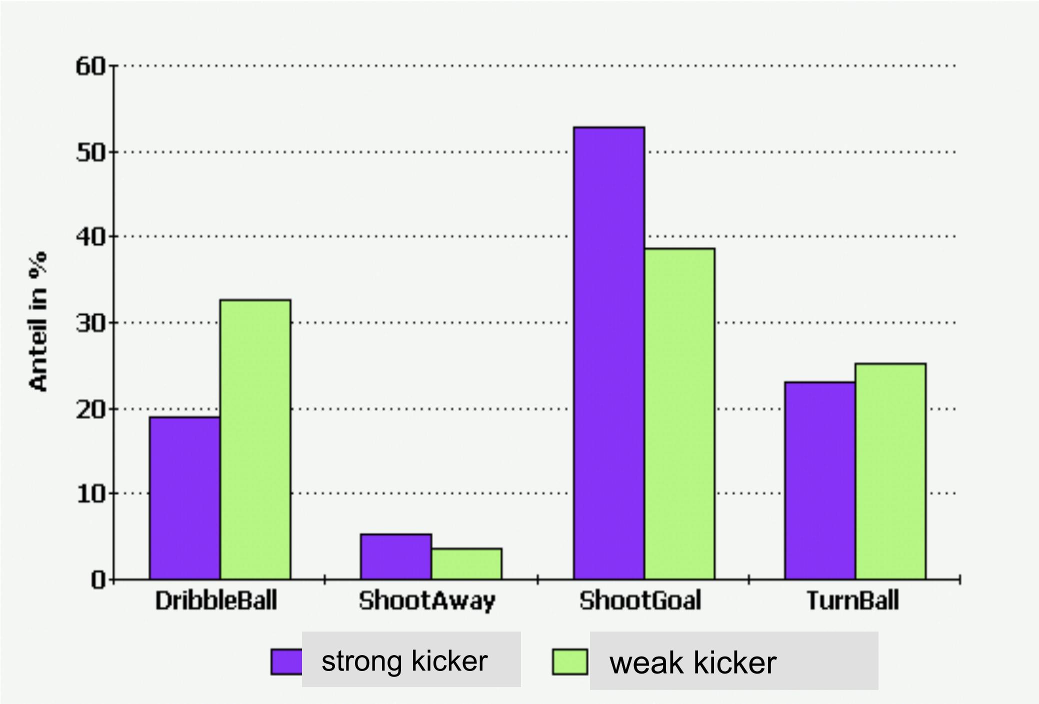
- Performance during continuous learning
  - once with the same (strong) kicking device (brown)
  - once with a replaced (weak) kicking device (green)
- The "weak" kicker curve increases



Adaption to sudden changes/defects

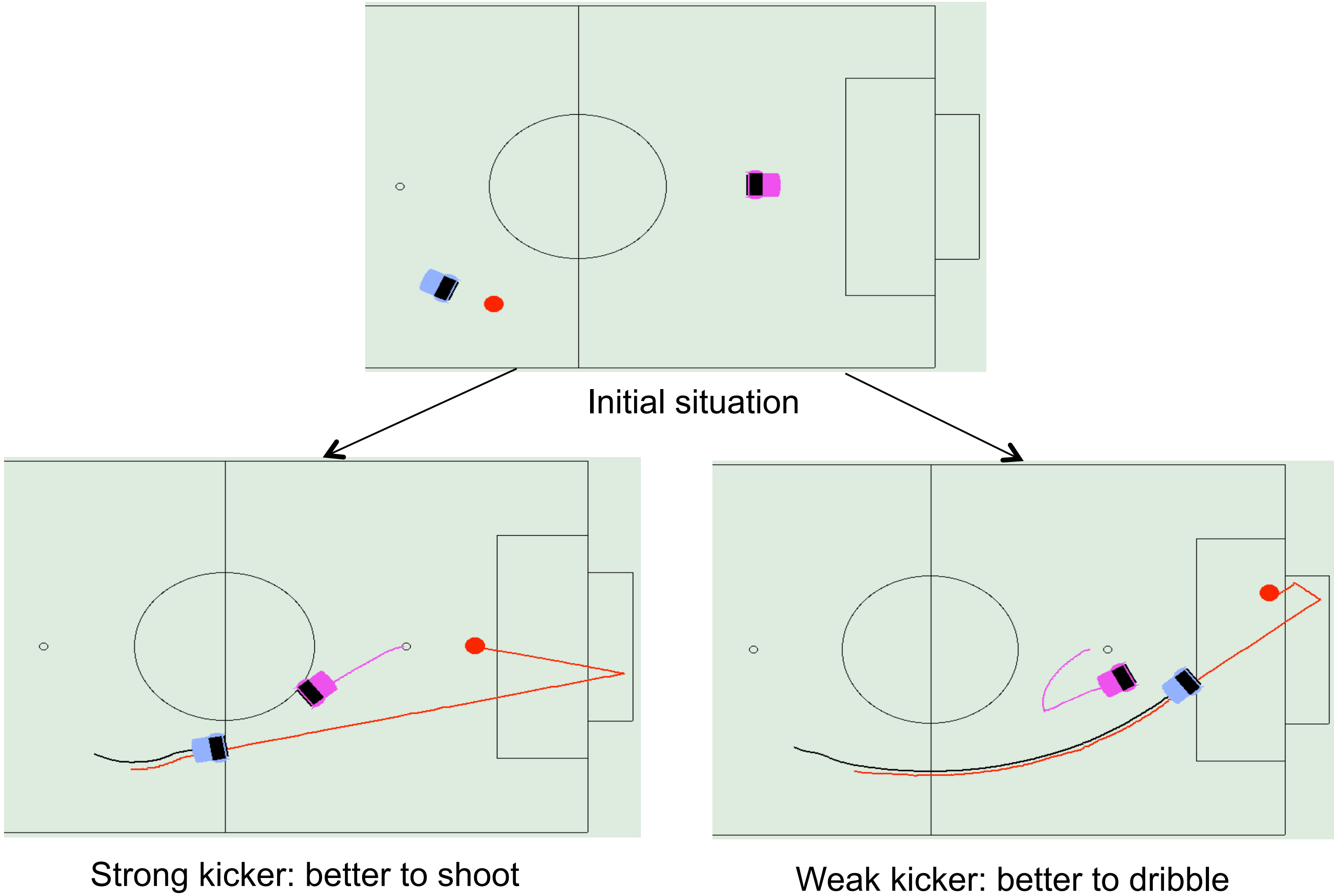
Selected behaviours during offensive

- The distribution of chosen behaviours changes...
  - The player with the weak kicker tends dribble more frequently
  - The player with the strong kicker prefers shooting behaviours



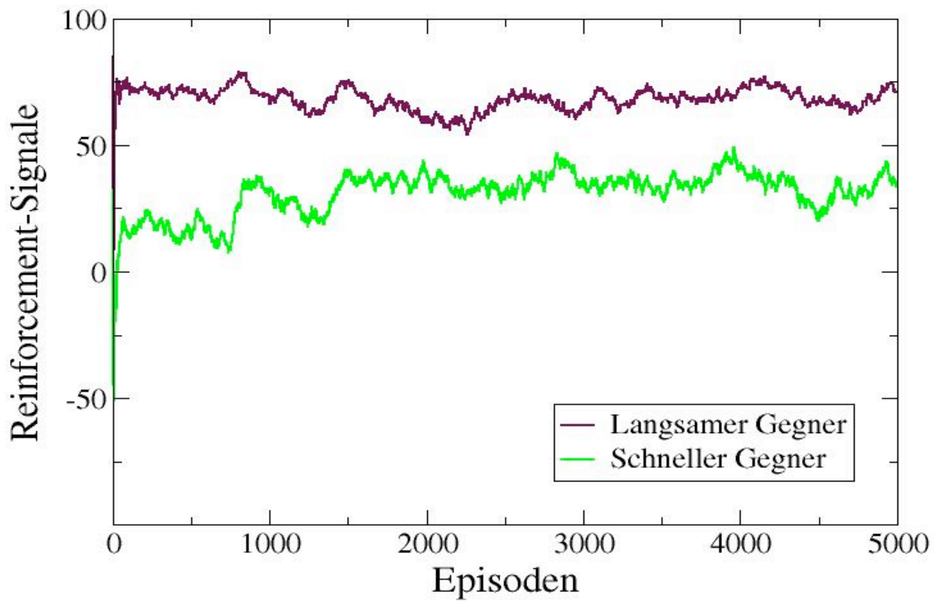
Adaption to sudden changes/defects

Behaviour with strong and weak kicker



Adaption to a different opponent

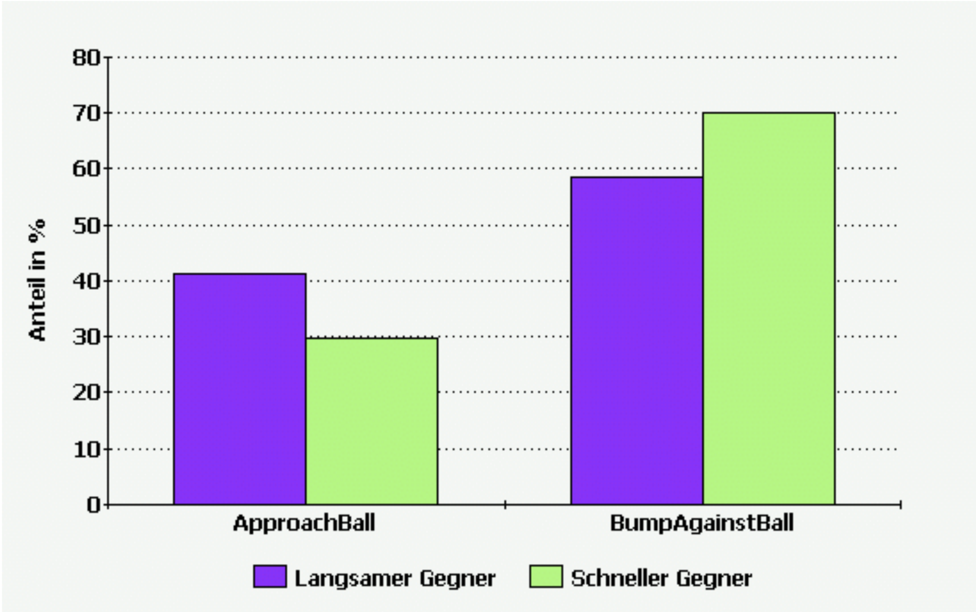
- Performance during continuous learning
  - once with the same (slow) opponent (brown)
  - once with a replaced (faster) opponent (green)
- The "faster" opponent curve increases



Adaption to a different opponent

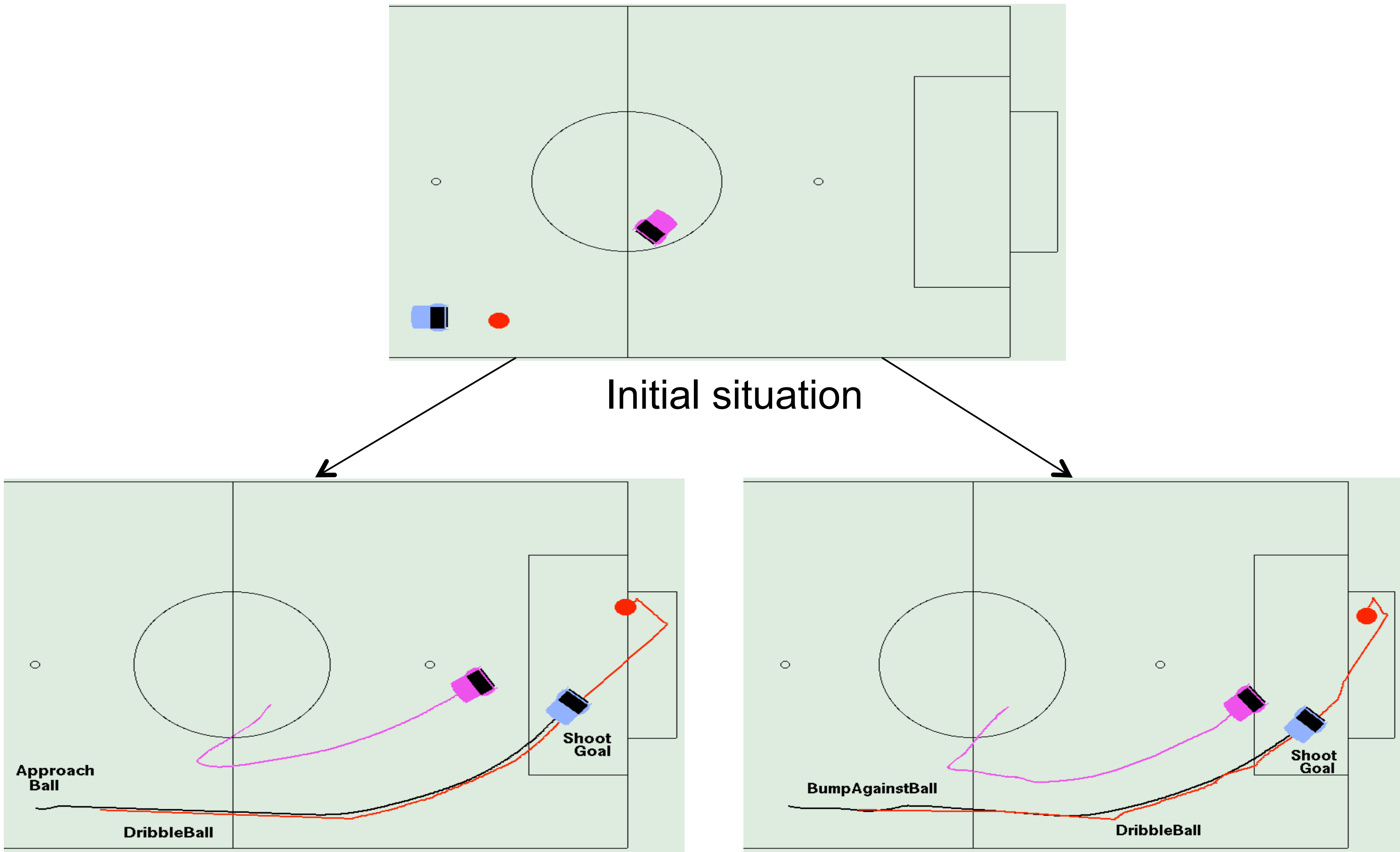
Selected behaviours during offensive

- The distribution of chosen behaviours changes again...
  - The player selects more often "BumpAgainstBall" in order to win time



Adaption to a different opponent

Behaviours against a slow and a fast opponent



Slow opponent

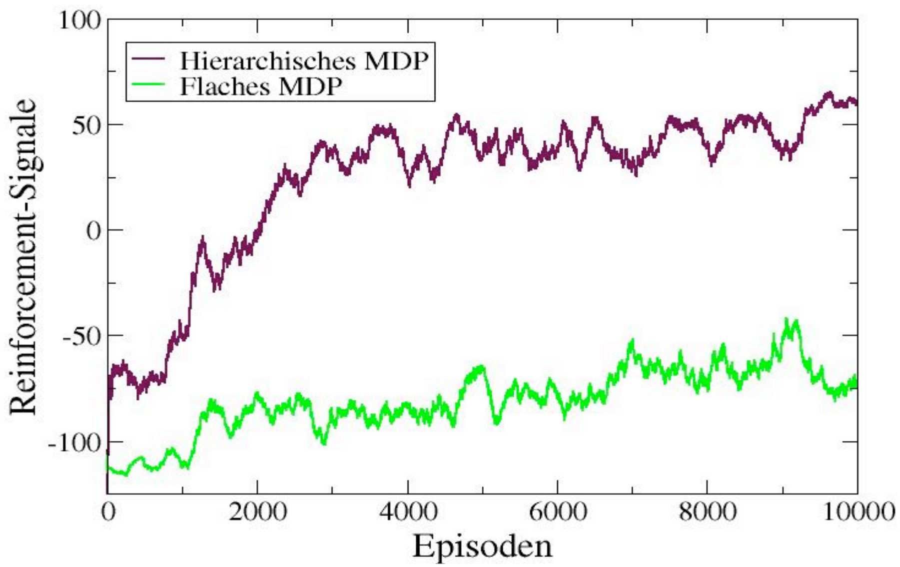
Fast Opponent

Some comments on adaption

- Re-learning takes automatically place without
  - user input to the system
  - the agent's knows nothing about the different concepts
  - no "performance gap" during to the re-learning

Hierarchical vs. Flat MDPs

- In the "flat" MDP we consider a single behaviour that takes as input all state variables
  - Learning takes much longer
  - Adaption unlikely ...



Transfer on the real robot platform

Achieved score

- Learner: 0.75 goals/minute
- CS-Freiburg player: 1.37 goals/minute
- Good result, but could still be improved...
  - Better (more realistic) simulation
  - Learning of additional skills
  - etc ...

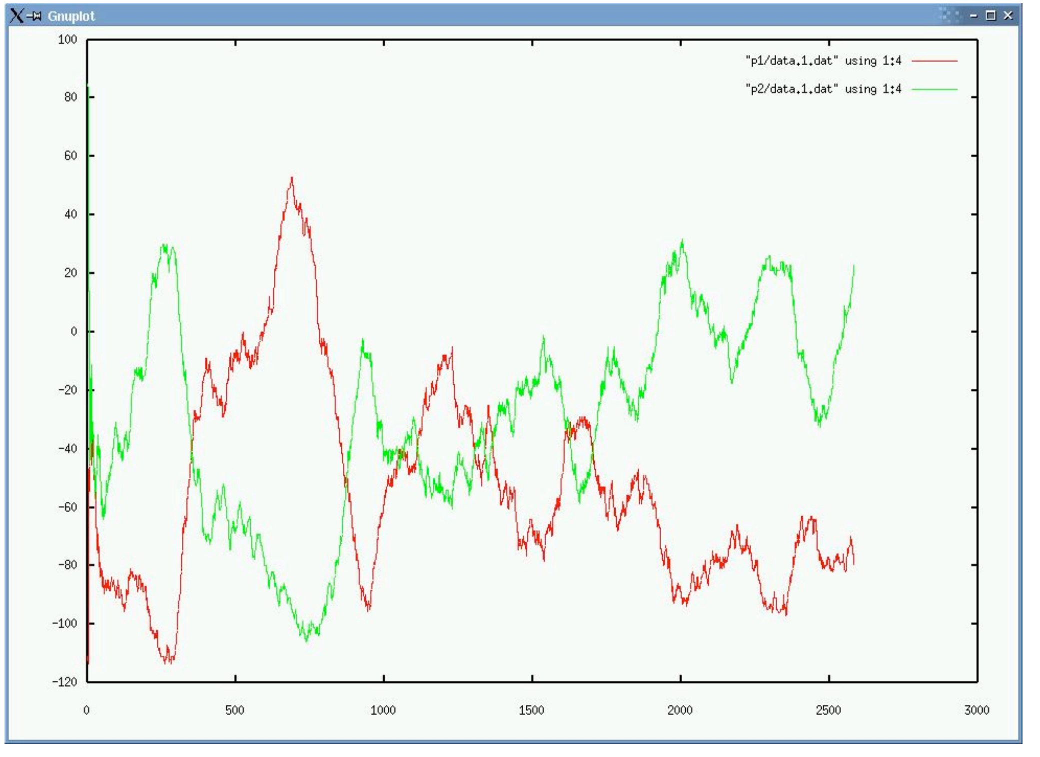
Video Result

Player executes *learned* behaviors and action selection

Multi-agent Learning *revised*

- So far we considered a relaxed version of the multi-agent learning problem:
  - Other agents were considered as stationary, i.e. executing a fixed policy
    - What if other agents are adapting to changes as well?
    - In this case we are facing a much more difficult learning problem with a moving target function
  - Furthermore, we did not consider multi-agent cooperation
    - Agents were choosing their actions greedily in that they maximized their individual reward
    - What if a team of agents shares a joint reward, e.g. scoring a goal in soccer together?

Example: Two robots learn playing soccer simultaneously



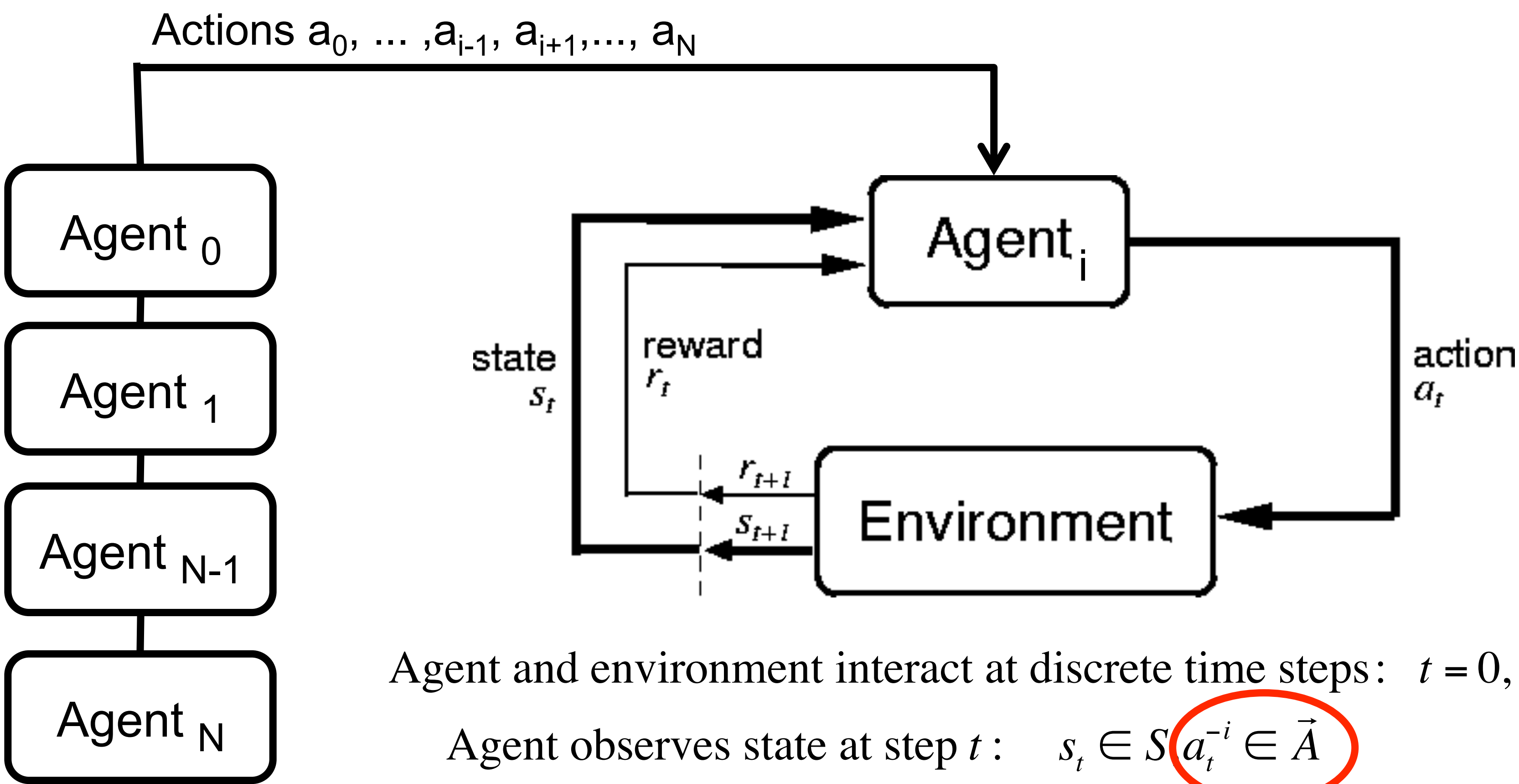
Multi-agent environments are non-stationary, thus violating the traditional assumption underlying single-agent learning approaches

Joint-Action Learners

Cooperation by learning joint-action values

- Consider the case that we have 2 offenders in the soccer game instead of one
  - The optimal policy depends on the joint action
  - For example, if robot A approaches the ball, the optimal action of robot B would be to do something else, e.g. going to the support position
- Solution: each agent learns a Q-Function of the joint action space: $Q(s,<a_1,a_2,\ldots,a_n>)$
- Observation or communication of actions performed by the team mates is required!

The Agent-Environment Interface for Joint-Action learners



Agent and environment interact at discrete time steps: $t = 0, 1, 2, \ldots$

Agent observes state at step t: $s_t \in S$ $a_t^{-i} \in \vec{A}$

produces action at step t: $a_t \in A(s_t)$

gets resulting reward: $r_{t+1} \in \Re$

and resulting next state: s_{t+1}

Joint-Action Learners

Opponent Modeling

- Maintain an explicit model of the opponents/team-mates for each state
- Q-values are updated for all possible joint actions at a given state
- Also here the key assumption is that the opponent is stationary
- Opponent modeling by counting frequencies of the joint actions they executed in the past
- Probability of joint action a_{-i}: $P(a'_{-i}) = \dfrac{C(a'_{-i})}{\sum_{a_{-i} \in A_{-i}} C(a_{-i})}$
- where $C(a_{-i})$ is the number of times the opponent has played action a_{-i}

Joint-Action learners

Opponent Modeling Q learning for agent i

(1) Let $\alpha_0 \in (0,1]$ be the initial learning rate, and ϵ be the initial exploration rate. Initialize $Q(s,\vec{a})$ arbitrarily, $C(s,a_{-i}) \leftarrow 0 \forall s \in S, \forall a_{-i} \in A_{-i}$, $n(s) \leftarrow 0 \forall s \in S$.

(2) Repeat,

(a) Observe state s, $n(s) \leftarrow n(s)+1$

(b) From state s select action a_i with probability $(1-\epsilon)$ by solving

$$\operatorname*{argmax}_{a_i} \sum_{a_{-i}} \frac{C(s,a_{-i})}{n(s)} Q(s,\langle a_i, a_{-i}\rangle),$$

and a random action with probability ϵ.

(c) Observing the opponent's action a_{-i}, the reward $R(s,a_i)$, and the next state s',

$$\begin{aligned} Q(s,\langle a_i,a_{-i}\rangle) &\leftarrow (1-\alpha)Q(s,\langle a_i,a_{-i}\rangle) + \alpha(R(s,a_i) + \gamma V(s') \\ C(s,a_{-i}) &\leftarrow C(s,a_{-i}) + 1 \end{aligned}$$

where

$$V(s') = \max_{a_i} \sum_{a_{-i}} \frac{C(s',a_{-i})}{n(s')} Q(s',\langle a_i', a_{-i}\rangle))$$

(d) Decay α and ϵ as per Q-learning.

Markov Games

- Also known as Stochastic Games or MMDPs
- Each state in a stochastic game can be considered as a matrix game* with payoff for player *i* of joint action *a* in state *s* determined by $R_i(s, a)$
- After playing the matrix game and receiving the payoffs, the players are transitioned to another state (or matrix game) determined by their joint action

* See slides from lecture 5: Game Theory

Minimax-Q

- Extension of traditional Q-Learning to zero-sum stochastic games
- Also here the the Q function is extended to maintain the value of joint actions
- Difference: The Q function is incrementally updated from the function $Value_i$
- $Value_i$ computes the expected payoff for player *i* if all players play the unique Nash equilibrium
- Using this computation, the Minimax-Q algorithm learns the player's part of the Nash equilibrium strategy

Summary

- RL can be used for learning online and model-free MDPs
  - In the past, different tasks, such as playing back gammon or robot soccer, have been solved surprisingly well
- However, it also suffers under the "curse of dimensionality", hence, success highly depends on an adequate representation or hierarchical decomposition
- Standard RL methods are in general not well suited for MA problems (but sometimes they work surprisingly well)
- The approach of Joint-Action learners allows to improve coordination among agents
- Stochastic games are a straightforward extension of MDPs and Game Theory
  - However, they assume that games are stationary and fully specified, enough computer power to compute equilibrium is available, and other agents are also game theorists…
  - ... which rarely holds in real applications

Literature

- Reinforcement Learning
  - R. S. Sutton, A. G. Barto, **Reinforcement Learning**, *MIT Press*, 1998, online at: http://www.cs.ualberta.ca/~sutton/book/the-book.html

- Hierarchical Q-Learning
  - A. Kleiner, M. Dietl, and B. Nebel. **Towards a Life-Long Learning Soccer Agent,** In *RoboCup 2002: Robot Soccer World Cup VI, (G. A. Kaminka, P. U. Lima, and R. Rojas, eds.), 2002, pp. 126-134.*

- Joint-Action learners
  - W. Uther, M. Veloso, **Adversarial reinforcement learning**, Tech. rep., Carnegie Mellon University, unpublished (1997).
  - C. Claus, C. Boutilier, **The dyanmics of reinforcement learning in co-operative multiagent systems**, in: *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA, 1998.
  - M. Bowling, M. Veloso, **Variable learning rate and the convergence of gradient dynamics**, in: *Proceedings of the Eighteenth International Con- ference on Machine Learning*, Williams College, 2001, pp. 27–34.