Neural Reinforcment Learning for Control

Cotesys ROS Fall School

Martin Riedmiller
Machine Learning Lab
Albert-Ludwigs-Universitaet Freiburg

Martin.Riedmiller@informatik.uni-freiburg.de

Machine Learning Lab Freiburg - Real Life Projects

- Neural forecasting and trading systems. Customers: **Axel-Springer AG** since 1997, **HeLaBa** 1995-1999, **ABP** 2006
- Neural control systems. Motor control (1996-1998), Active damping system (2006) customers: German car companies
- Neural Slot Car Racer (Harting, Hannover Fair 2008, 2009)
- Machine Learning in autonomous robots (RoboCup, since 1999)

$\Rightarrow$ Learning components are embedded in large software systems

Machine Learning Lab Freiburg

'Future computer programs will contain a growing part of 'intelligent' software modules that are not conventionally programmed but that are learned either from data provided by the user, or from data that the program autonomously collects during its use.'

develop software systems, that learn from data or own experience

Machine Learning Lab Freiburg

'Future computer programs will contain a growing part of 'intelligent' software modules that are not conventionally programmed but that are learned either from data provided by the user, or from data that the program autonomously collects during its use.'

develop software systems, that learn from data or own experience

research on fast, robust, and (data-) efficient learning methods
for supervised, unsupervised and reinforcement learning scenarios

develop concepts for embedding learning components
in complex software solutions

Our goals of a learning controller

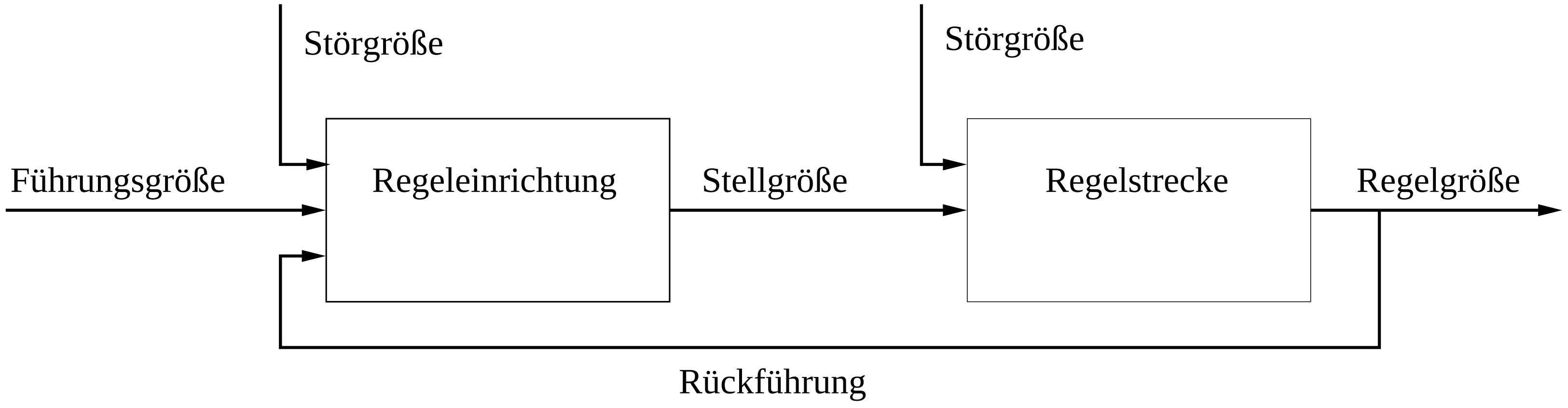
Autonomously learn a complex behaviour from scratch

- no control knowledge, no initial policy
- no process knowledge (no model)
- high quality

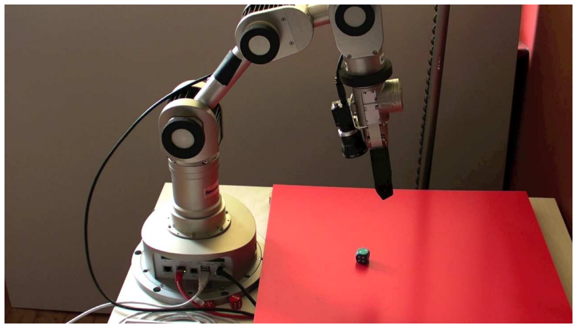
Reinforcement Learning: learning from reward and punishment

example: riding a bicycle

Closed loop control and typical applications



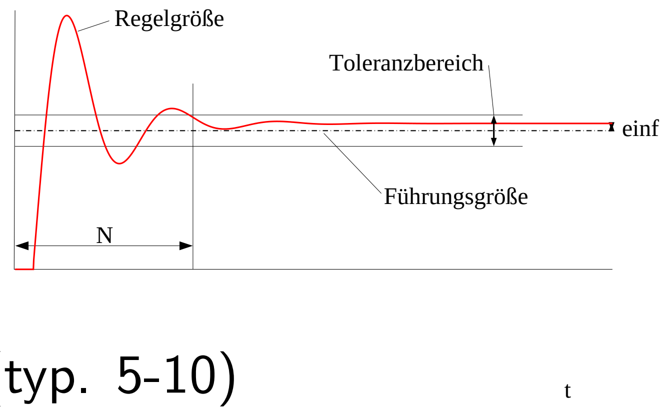
- temperature control
- position control
- active suspension control
- robot control
- . . .



General control loop implentation realized by open source software $CLS^2 (CLSquare)$

Reinforcement Learning (RL) for technical process control

Challenges:



- model free
- non-linear, noisy
- continous states, continuos actions
- considerable number of state variables (typ. 5-10)
- considerable trajectory length
- high quality control behaviour
- typical: set-point regulation $y \mapsto y^d$

Goals of the talk

- discuss value function based approaches as one methodology to tackle real world RL problems
- in particular, focus on Neural Fitted Q Iteration (NFQ) as an efficient, model-free method
- discuss practical issues in setting up the learning tasks

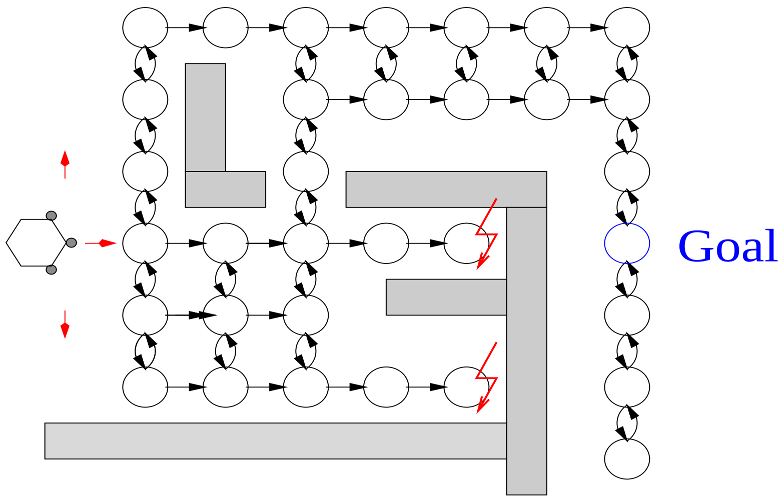
Overview

- Review: value function based Reinforcement Learning
- Choosing degrees of freedom: immediate costs, discounting, states, actions
- Efficient learning: Neural fitted Q iteration
- Drifting values problem: Hint-to-Goal and Min-Q heuristics
- Generation/ reuse of training data
- Improving Accuracy
- Continuous actions
- Case studies

I

Value Function based RL

Sequential Decision Making



Examples:
Chess, Checkers (Samuel, 1959), Backgammon (Tesauro, 92)
Cart-Pole-Balancing (AHC/ ACE (Barto, Sutton, Anderson, 1983)), Robotics and control, . . .

Three Steps

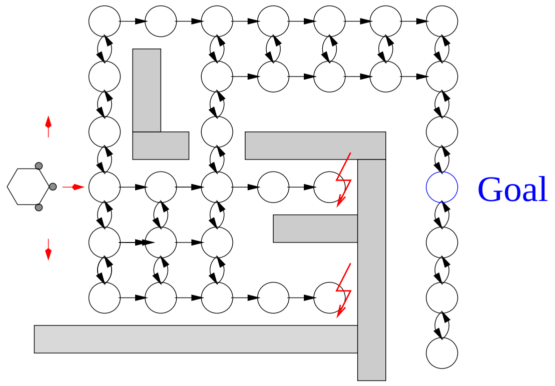
$\Rightarrow$Describe environment as a Markov Decision Process (MDP)

$\Rightarrow$Formulate learning task as a dynamic optimization problem

$\Rightarrow$Solve dynamic optimization problem by dynamic programming methods

1. Describing the environment

S: (finite) set of states
A: (finite) set of actions



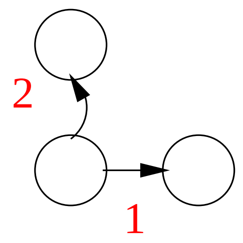
Behaviour of the environment (transition) model
$p : S \times S \times A \rightarrow [0, 1]$
$p(s', s, a)$ Probability distribution of transition

Markov property: Transition only depends on current state and action

$$Pr(s_{t+1}|s_t, a_t) = Pr(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \ldots)$$

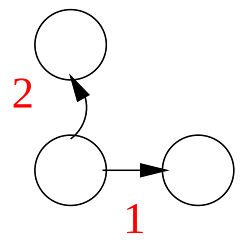
2. Formulation of the learning task

every transition emits transition costs, 'immediate costs', $c : S \times A \longrightarrow \Re$
(sometimes also called 'immediate reward', r)

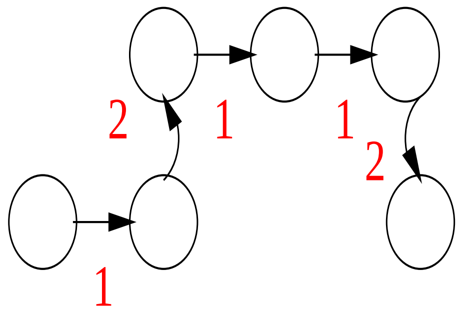


2. Formulation of the learning task

every transition emits transition costs, 'immediate costs', $c : S \times A \rightarrow \Re$ (sometimes also called 'immediate reward', r)



Now, an agent policy $\pi : S \rightarrow A$ can be evaluated (and judged):
Consider pathcosts:
$J^\pi(s) = \sum_t c(s_t, \pi(s_t)), s_0 = s$



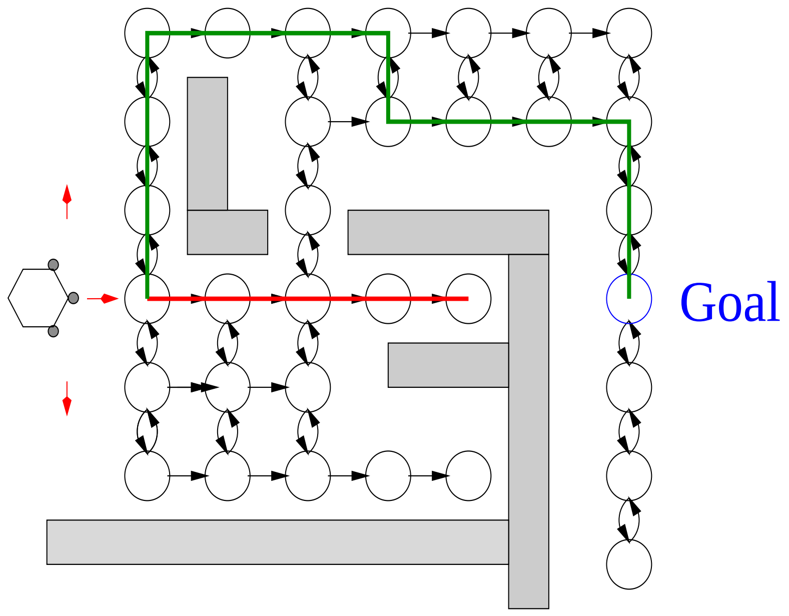
Wanted: optimal policy $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$
where $J^{\pi^*}(s) = \min_\pi \{\sum_t c(s_t, \pi(s_t)) | s_0 = s\}$

$\Rightarrow$Additive (path-)costs allow to consider *all* events during trajectories
$\Rightarrow$This formulation cares for the complete temporal behaviour of the system

Choice of immediate cost function $c(\cdot)$ specifies policy to be learned

Example:

$$c(s) = \begin{cases} 0 & , \quad \text{if } s \text{ success } (s \in Goal) \\ 1000 & , \quad \text{if } s \text{ failure } (s \in Failure) \\ 1 & , \quad else \end{cases}$$



$$J^{\pi}(s_{start}) = 12$$
$$J^{\pi}(s_{start}) = 1004$$

$\Rightarrow$specification of requested policy by $c(\cdot)$ is simple!

3. Solving the optimization problem

For the optimal path costs it is known that

$$J^*(s) = \min_a \{ \sum_{s' \in S} p(s, s', a) \, (c(s, a) + J^*(f(s, a)))\}$$

(Principle of Optimality (Bellman, 1959))

$\Rightarrow$Can we compute J^* (we will see why, soon)?

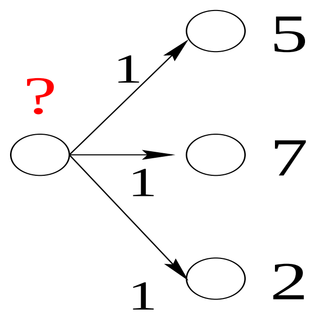
Computing J^*: the value iteration (VI) algorithm

Start with arbitrary $J_0(s)$
for all states $s: J_{k+1}(s) := \min_{a \in \mathcal{A}} \{c(s,a) + J_k(f(s,a))\}$

Computing J^*: the value iteration (VI) algorithm

Start with arbitrary $J_0(s)$

for all states s : $J_{k+1}(s) := \min_{a \in \mathcal{A}} \{c(s,a) + J_k(f(s,a))\}$

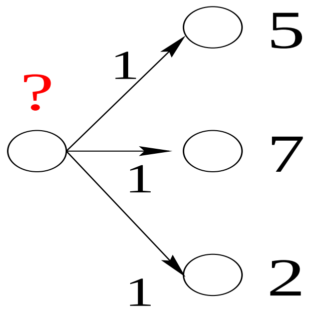


$\Rightarrow$

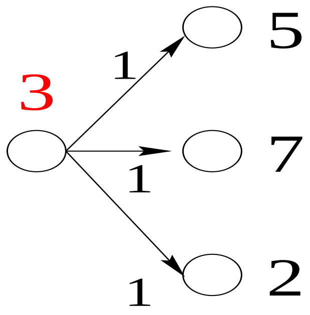
Computing J^*: the value iteration (VI) algorithm

Start with arbitrary $J_0(s)$

for all states s :$J_{k+1}(s) := \min_{a \in \mathcal{A}} \{c(s,a) + J_k(f(s,a))\}$



$\Rightarrow$



$\Rightarrow$Can be extended straight-forward to the stochastic case

Convergence of value iteration

Value iteration converges under certain assumptions, i.e. we have $lim_{k \to \infty} J_k = J^*$

$\Rightarrow$Discounted problems: $J^{\pi^*}(s) = \min_\pi \{\sum_t \gamma^t c(s_t, \pi(s_t)) | s_0 = s\}$
where $0 \leq \gamma < 1$ (contraction mapping)

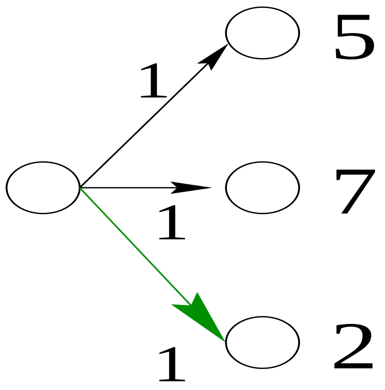
$\Rightarrow$Stochastic shortest path problems:

- there exists an absorbing terminal state with zero costs
- there exists a 'proper' policy (a policy that has a non-zero chance to finally reach the terminal state)
- every non-proper policy has infinite path costs for at least one state alternative, stronger assumption: $c(s, u) > 0$ if s is not a goal state

Ok, now we have J^*

⇒when J^* is known, then we also know an optimal policy:

$$\pi^*(s) \quad \in \quad \arg\min_{a\in\mathcal{A}}\{c(s,a) + J^*(f(s,a))\}$$



Reinforcement Learning

Problems of Value Iteration:

for all $s \in \mathcal{S}$:$J_{k+1}(s) = \min_{a \in \mathcal{A}}\{c(s,a) + J_k(f(s,a))\}$

problems:

- Size of S (Chess, robotics, . . .) $\Rightarrow$ learning time, storage?
- 'model' (transition behaviour) $f(s,a)$ or $p(s', s, a)$ must be known!

Reinforcement Learning

Problems of Value Iteration:

for all $s \in \mathcal{S}$:$J_{k+1}(s) = \min_{a \in \mathcal{A}}\{c(s,a) + J_k(f(s,a))\}$

problems:

- Size of S (Chess, robotics, . . .) $\Rightarrow$ learning time, storage?
- 'model' (transition behaviour) $f(s,a)$ or $p(s', s, a)$ must be known!

Reinforcement Learning is dynamic programming for very large state spaces and/ or model-free tasks

Important contributions - Overview

- Real Time Dynamic Programming
  (Barto, Sutton, Watkins, 1989)

- Model-free learning (Q-Learning,(Watkins, 1989))

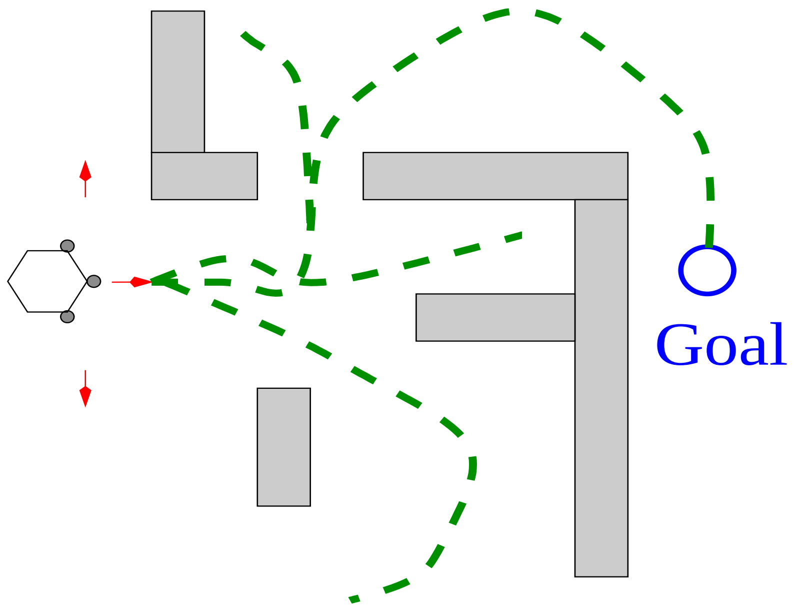
- neural representation of value function (or alternative function approximators)

Real Time Dynamic Programming (Barto, Sutton, Watkins, 1989)

Idea:

instead For all $s \in S$ now For some $s \in \mathcal{S}$ . . .

$\Rightarrow$concentrating on the 'relevant' parts of state space



Learning without a model: Q-Learning (Watkins, 1989)

Idea Represent expected path-costs as a function of state-action pairs:

$$Q^*(s,a) := \sum_{s' \in S} p(s', s, a)(c(s,a) + J^*(s'))$$

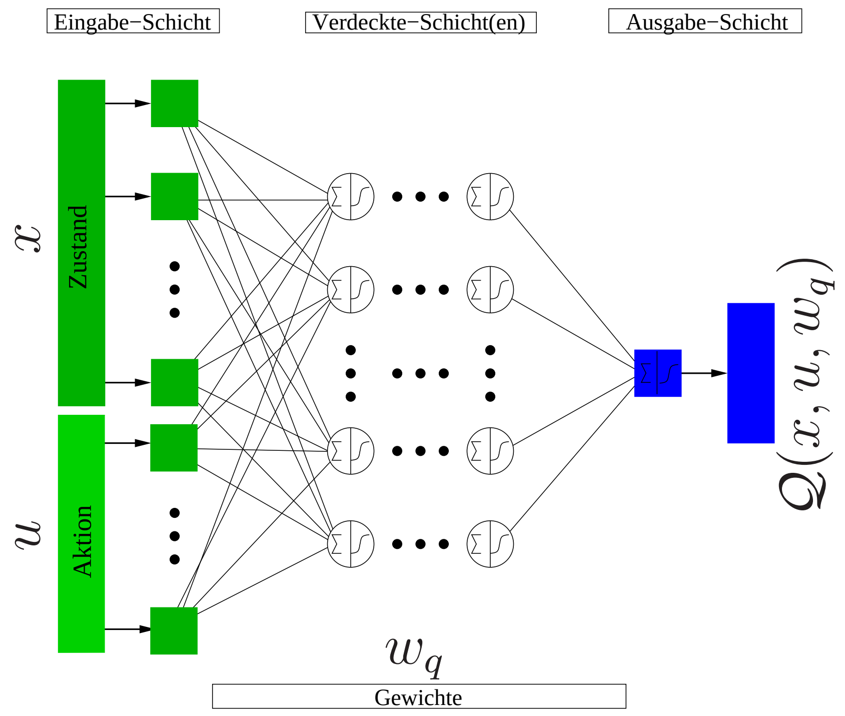
- optimal Q-function can be learned by sampling state transitions $s, a \rightarrow s'$ only (no transition model required)

$$Q_{k+1}(s,a) := (1-\alpha)\, Q_k(s,a) + \alpha\, (c(s,a) + \min_{a' \in \mathcal{A}(s')} Q_k(s', a'))$$

- also, action selection is model-free: $\pi^*(s) = \arg\min Q^*(s,a)$
- convergence properties similar to value iteration + every state-action pair has to be visited infinitely often
- Q-function is updated after every state transition

Representation of the path-costs in a function approximator

Idea: neural representation of value function (or alternative function approximators) (Neuro Dynamic Programming (Bertsekas, 1987))



⇒few parameters (here: weights) specify value function for a large state space ⇒learning by gradient descent:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial(\min_b Q(s',b) - c(s,a) - Q(s,a))^2}{\partial w_{ij}}$$

Consequences from theory for the learning controller setup

- Markov poperty: state information s must be 'rich enough' to fulfill the Markov property. If state information is not known, or cannot be measured, approximations might be used, e.g. $\dot{pos}(t) \approx \frac{pos(t)-pos(t-\triangle t)}{\triangle t}$. Since no explicit model is built, one has a lot of possibilities for the choice of the state information (e.g. might be redundant), as long as the main information is captured. A standard way might be to just use differences with different time lags.

- Existence of a proper policy: action sets must be 'rich enough', that a proper policy exists. However, an explicit proper policy need not to be known in advance

- Discounting: in principle, using a discount factor < 1 leads to a framework, where convergence conditions are more relaxed. However, it will also influence the type of optimal solution one gets. Using no discounting (i.e $\gamma = 1$) therefore might be closer to the intention with respect to the quality of the solution.

- immediate costs:
  - should be derived from the type of solution one desires, e.g. 'minimum-time'. Typically should not be misused as a local pointer to the goal, such as e.g. 'distance to the goal' - unless 'minimizing the sum of distances to the goal along the trajectory' is the type of solution one expects.
  - can be chosen very flexibly, e.g. no restriction to quadratic costs or to being diffentiable, ...
  - constraints: if hard constraints (e.g. damaging the robot) are violated, trajectory terminates and large terminal path costs are given. Terminal path costs should be larger than largest path costs of a trajectory leading to goal
  - concrete choice depends on the type of problem. In technical control problems, two types are of particular interest:
    a. goal states with termination property (TG: Termination at goal)
    b. goal states without termination property (NoTG: Non-termination at goal)

Immediate costs (I)

Goal states with termination property (TG)

Example mobile robot positioning: driving the robot to a certain position. By reaching the goal, the control task is finished.

Choice of immediate costs:

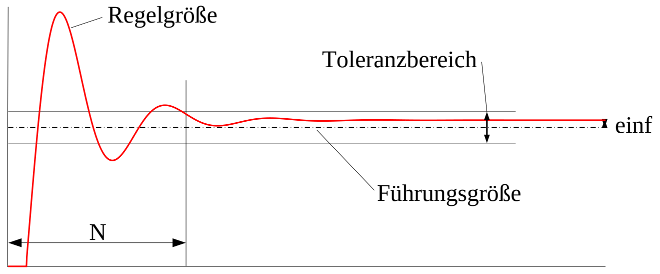
- for non-goal states: $c(s, a) > 0$
- if goal is reached terminal path costs of 0 arise. Trajectory terminates.
  $Q(s, a) \leftarrow c(s, a) + J(s^+) = c(s, a) + 0$.

Immediate costs (II)

Goal states with non termination property (NoTG)

Example robot arm positioning: move the finger tips to a certain position and keep them there, even under external disturbances. The control task continues forever.

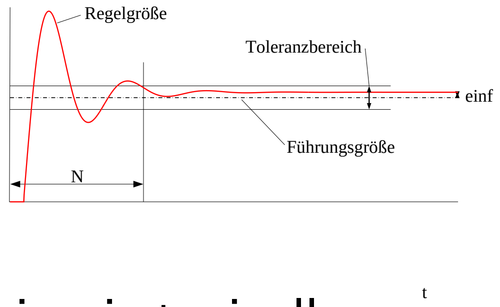
This is the typical situation faced by so-called 'set-point regulation problems', where a system output y should be controlled to reach and stay at a certain target value y^d



Choice of immediate costs:

- for non-goal states: $c(s,a) > 0$
- for goal states: $c(s,a) = 0$. No termination.
  $Q(s,a) \leftarrow c(s,a) + J(s^+) = 0 + \min_b Q(s^+,b)$.

Immediate costs (III)



- For set-point regulation problems, the goal region is typically chosen, that the target value is reached with some tolerance δ, i.e. $c(s,u) = 0 \Leftrightarrow y \in [y^d - \delta, y^d + \delta]$
- For non-discounted probelms, requires that a policy exists, that is able to keep the system in the goal area ('proper' policy). Policy need not to be known in advance. This influences the choice of the action set and the size of the goal region.
- Note: Control task is more difficult than in TG problems. Typically trade of between going fast to target region and keeping the system there.
- Care must be taken, if function approximation is used $\longrightarrow$ 'drifting values problem' (see later)

Immediate costs (IV)

A very general framework is to use constant costs outside the goal region, e.g.

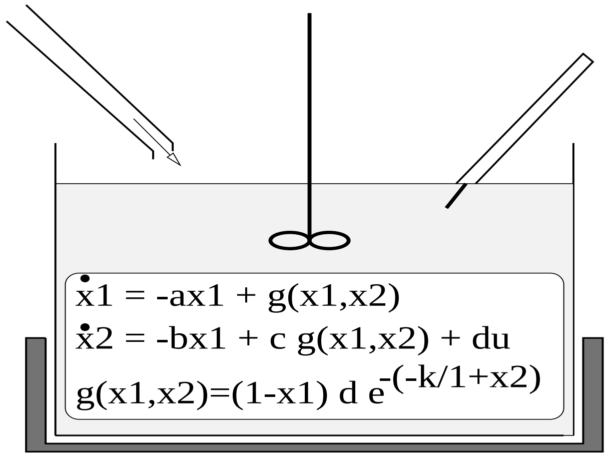
$$c(s,u) = 1, \text{ if } s \text{ not in goal}; c(s,u) = 0, else$$

This leads to a minimum time control policy, where the time outside the target region is minimized.

- very general formulation that requires no prior knowledge about the process behaviour
- in combination with neural networks with sigmoidal outputs, the immediate costs are scaled by a small factor, such that the expected pathcosts for successful policies are < 1, e.g. $c(s,u) = 0.01$

Control of a chemical reaction by heating/ cooling

control:	u external heating/ cooling
	x_1 concentration
measured:	x_2 temperature in reactor
goal:	keep a certain temperature
	to get a desired concentration



$\dot{x}1 = -ax1 + g(x1,x2)$
$\dot{x}2 = -bx1 + c\, g(x1,x2) + du$
$g(x1,x2)=(1-x1)\, d\, e^{-(-k/1+x2)}$

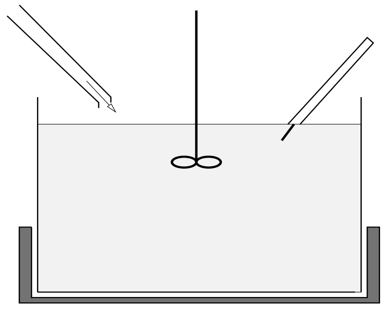
Analytical nonlinear controller

$$u = -k(y)\, y = -[C_0\, y + C_1\, y\, e^{-\frac{-\epsilon}{1+y}} + C_2\, y\, \frac{e^{-\frac{-\epsilon}{1+y}} - e^{-\frac{-\epsilon}{1+y_R}}}{y - y_R}]$$

Neural RL control

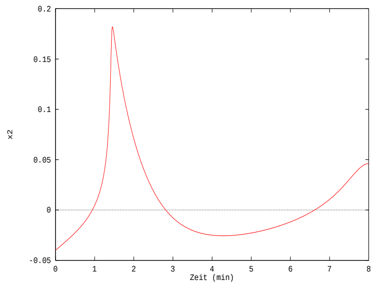
- Actions: $\mathcal{A} = \{-0.05, +0.05, 0\}$
- states: concentration x_1 temperature x_2
- neural value function: MLP, 3-20-1
- goal: $x_2 - x_2^d = \triangle x_2 \approx 0$
- Setpoint regulation: NoTG framework,

$$c(s,a) := \begin{cases} 0\,, |\triangle x_2| < \delta \\ 1\,, \text{else} \end{cases}$$

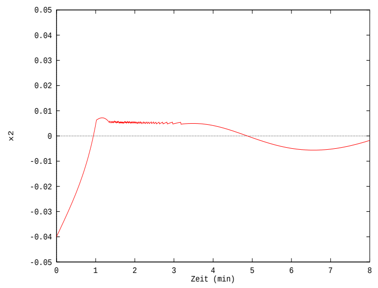


Learning

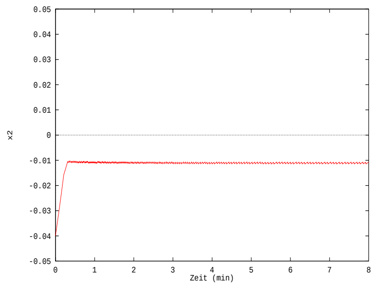
untrained



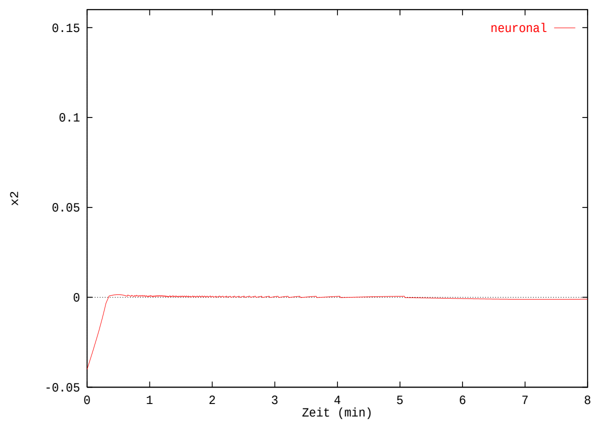
1000 episodes

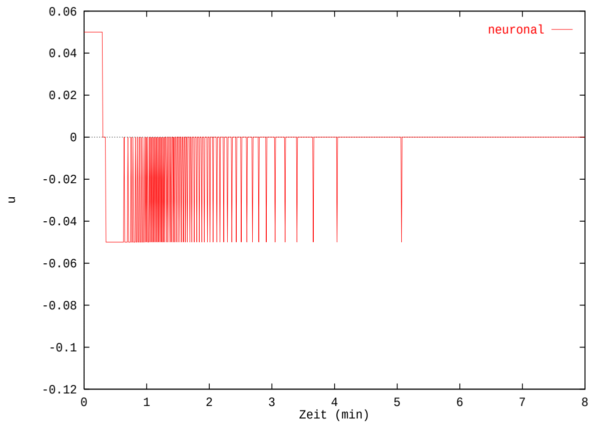


20000 episodes

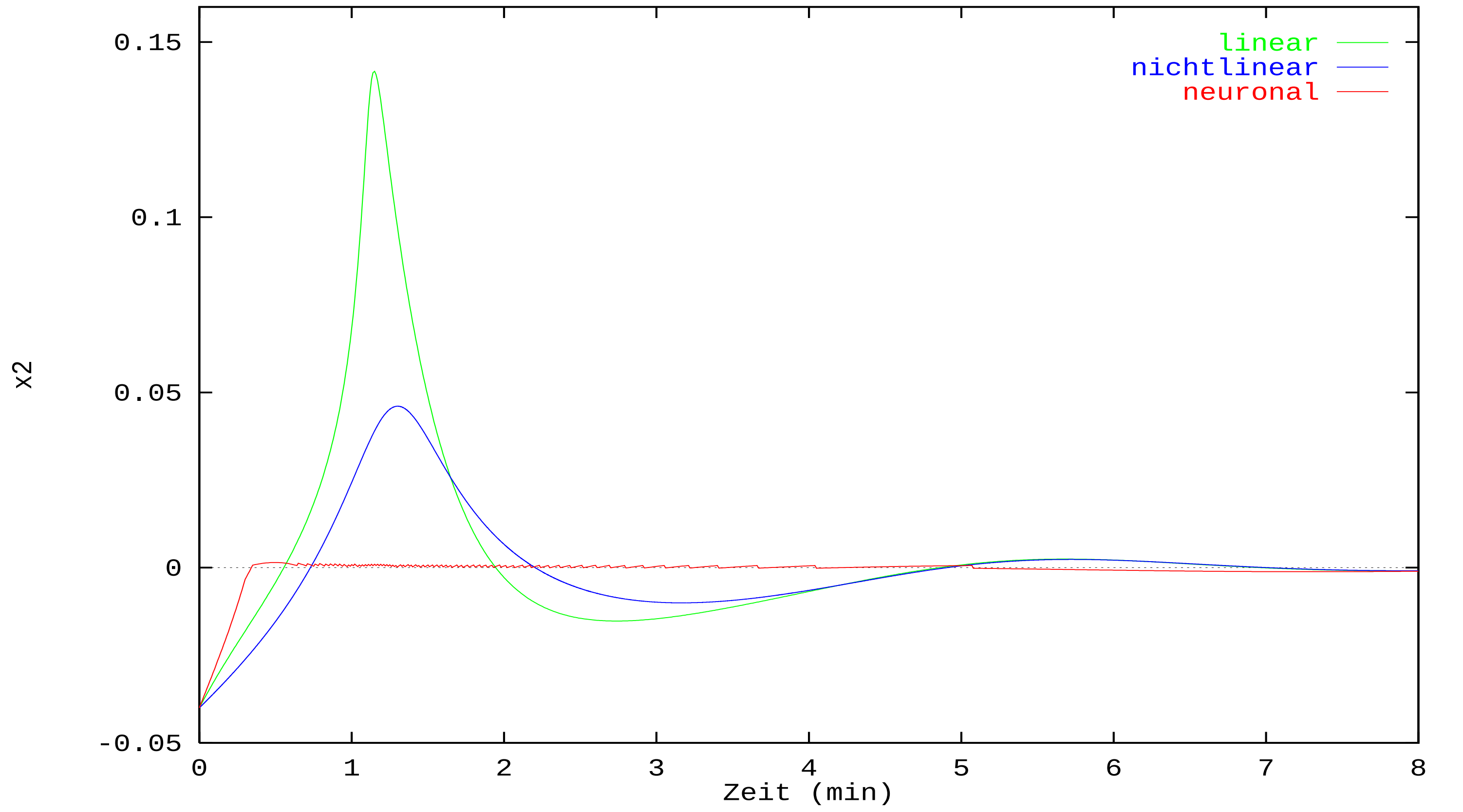


Final performance ($50\,000$ episodes):





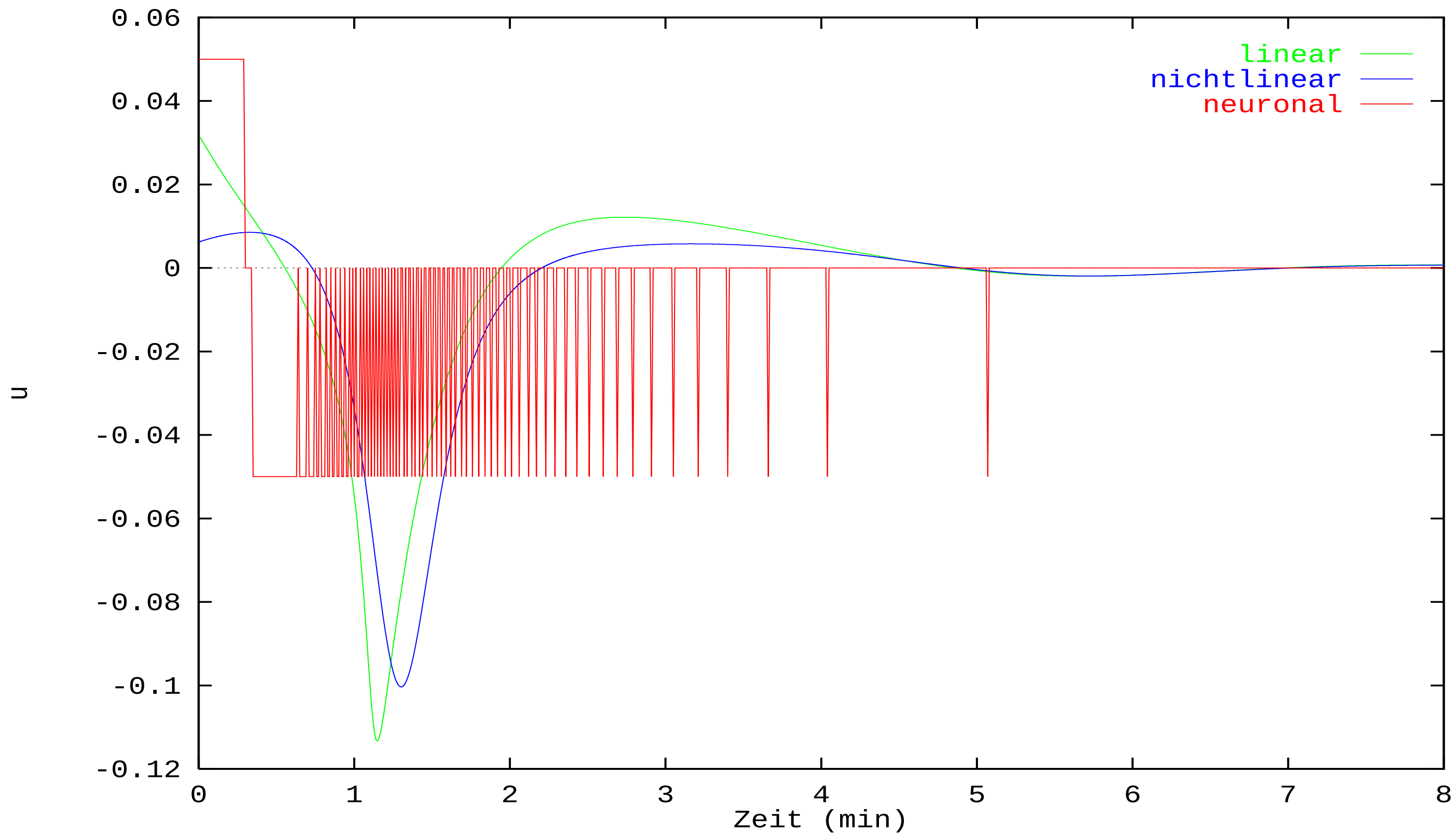
Comparison Analytical - Neural



green: linear, analytically derived
blue: non-linear, analytically derived

red: neural RL controller

Comparison Analytical - Neural. Policy



green: linear, analytically derived
blue: non-linear, analytically derived

red: neural RL controller

Intermediate summary

- needs no (analytical or whatsoever) prior model of the plant
- controller quality high
- high flexibilty to specify controller properties
- choice of states representation, actions, immediate costs, discounting is reasonably simple
- open questions:
  - lack of efficiency (often, $\approx$ 1E5 to 1E6 episodes required)
  - continuous actions

II

Neural Fitted Q Iteration (NFQ)

Neural Networks (Multilayer Perceptrons, MLPs) as Value Functions

+ good generalisation ('global' fitting), general function approximator

- can 'forget' values at arbitrary points (in contrast to a local scheme as a table, or a grid, or RBF network)

Idea: to prevent forgetting, store crucial information explicitly (in contrast to TD-learning)
Here, 'crucial information' means the transition tuples (s, a, s')

$\Rightarrow$ update neural value function at all transition tuples concurrently:
Neural Fitted Q Iteration (Riedmiller, 2005)

Idea in the spirit of 'Fitted Value Iteration' (Gordon, 1995) and 'Fitted Q Iteration' (Ormoneit, Sen, 2002; Ernst, 2005)

Neural Fitted Q Iteration NFQ (Riedmiller 2005)

$Q_k \in \mathcal{F}$: neural approximation of Q-value function at stage k
$(s_1, a_1, s'_1), \ldots, (s_N, a_N, s'_N)$ are sampled transitions

For each transition sample do

$$\hat{Q}(s_i, a_i) := c(s_i, a_i) + \gamma min_a Q_k(s'_i, a)$$

Compute next iterate Q_{k+1} by

$$Q_{k+1} = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} (f(s_i, a_i) - \hat{Q}(s_i, a_i))^2$$

Neural Fitted Q Iteration NFQ (Riedmiller 2005)

$Q_k \in \mathcal{F}$: neural approximation of Q-value function at stage k
$(s_1, a_1, s'_1), \ldots, (s_N, a_N, s'_N)$ are sampled transitions

For each transition sample do

$$\hat{Q}(s_i, a_i) := c(s_i, a_i) + \gamma min_a Q_k(s'_i, a)$$

Compute next iterate Q_{k+1} by

$$Q_{k+1} = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} (f(s_i, a_i) - \hat{Q}(s_i, a_i))^2$$

Reinforcement Learning becomes solving a series of supervised learning problems (offline!) on sampled transition data. Batch learning: Efficient and robust neural training methods exists (e.g. Rprop, (Riedmiller, 1992))

Neural Fitted Q Iteration

1. sample transitions $(s_i, a_i, s'_i, r_i), i = 1 \ldots N$
   - randomly
   - 'greedy sampling': doing actual trajectories greedily exploiting Q_k
   - according to some other policy
   - ...
2. compute training targets
   $\hat{Q}(s_i, a_i) := c(s_i, a_i) + \gamma min_a Q_k(s'_i, a), i = 1 \ldots N$
3. train MLP to approximate $\hat{Q}$ using Rprop (Riedmiller, 1992)
4. optionally use special extensions, e.g. additional training patterns ('hint-to-goal')

NFQ

- model-free: assumes no a priori known model of the plant
- highly efficient with respect to the number of data points collected
- applicable to real systems directly (see examples)
- Rprop is fast and very robust with respect to parameter choice. Of course, other supervised learning algorithms can be used
- Instead of MLPs, other regression methods can be used (e.g. Gaussian processes (Deisenroth, Rasmussen, Peters), Randomized trees (Ernst et.al): Fitted Q Iteration framework (Ernst et. al, 2005)

Neural Fitted Q Iteration

```
NFQ_main() {
input: a set of transition samples $D$; output: Q-value function $Q_N$
    k=0
    init_MLP() $\rightarrow Q_0$;
    Do {
        generate_pattern_set $P = \{(input^l, target^l), l = 1, \ldots, \#D\}$ where:
            $input^l = s^l, u^l$,
            $target^l = c(s^l, u^l, s'^l) + \gamma \, min_b Q_k(s'^l, b)$
        Rprop_training($P$) $\rightarrow Q_{k+1}$
        k:= k+1
    } While ($k < N$)
```

CartPole Regulation

'control a pole mounted on a cart avoiding failure of pole'
4 dim, constraints (boundaries of track). two actions $\pm 10N$.

'Classical' balancing benchmark: prevent pole from falling starting from upright position (avoidance task).

Here: regulator task: Regulate cart position with pole uprigtht, starting from various initial states
starting states: position $\pm 1.0m$, angle $\pm 20^o$.
target region: position $0 \pm 0.05m$, angle 0 ± 3^o

CartPole - NFQ results

NFQ (5-5-5-1-net), average over 20 experiments, data-acquisition: iterative-greedy (max episode length: 100)

First successful policy				
	episodes	cycles	interaction time	costs
average	197.3	14439.8	4m49s	319.1
Best policy found (within 500 episodes)				
	episodes	cycles	interaction time	costs
average	354.0	28821.1	9m 36s	132.9

Learning performance reference:

2 neural networks, online (modelbased): $> 100,000$ episodes (Riedmiller, 1996)

Q-table (50x50x50x50): $> 2,000,000$ episodes

Controller performance reference:

linear controller: avg. cycles out of target region: 405.2.

Pole (avoidance)

Avoidance task: balancing a pole, noise

2 dim, constraints (boundaries of track). two actions $\pm 50N$.

avoid failure $angle > 90^o$

Sampling: random trajectories from upright position

Success: Balancing for ≥ 3000 cycles

Pole (avoidance) - NFQ

NN: 3-5-5-1 (like in cart-pole).

# random episodes	successful learning trials
50	23/50 (46%)
100	44/50 (88 %)
150	48/50 (96 %)
200	50/50 (100 %)
300	50/50 (100 %)
400	50/50 (100 %)

Benchmark used for LSPI in (Lagoudakis, Parr, 2003)
Achieved $approx$ 99% success when using 1000 random episodes for training.

Acquisition of transition data

All information about system behaviour is captured in the transition tuples (s, a, s'). The information within the tuples is independent of the policy, with which it is collected.
Therefore, transistion data can be collected in various ways:

- randomly
- by greedily exploiting the current Q-function
- by sampling according to a prior policy
- by learning a different policy first. Example: learning to swing-up a pole. Reuse data to learn to balance
  learning to swing-up a cart-pole system. Reuse data to do suspension

Example: Control a robot to a target position

- actions: drive back and forth with different speeds
- task: reach and keep a position as fast as possible
- problem: be fast but avoid overshooting
- type of problem: set-point regulation $\Rightarrow$ NoTG
- learned directly on a real robot in less than 100 episodes

III

Practical aspects of NFQ

The 'Drifting Q-values' Problem

- in NoTG (no terminal goal) problems, no explicit terminal states exist with known pathcosts of 0 ($\Rightarrow$ 'no anchor')
- for all states (also for goal states) the same update rule is used:

$$Q(s,a) \leftarrow c(s,a) + J(s^+) = c(s,a) + \min_b Q(s^+, b)$$

  In the minimum case, $c(s,u) = 0$ and the Q-value is set to the path costs of its successor.
  In all other cases, the Q-value is increased by $c(s,u)$

- by the generalisation property of MLPs, all the values therefore tend to increase and finally, the Q-value of all states is ≈ 1.
- Two methods:
  - forcing the output values to be 0 at some a priori determined states by introducing artificial training patterns ('hint-to-goal' (HTG) heuristic)
  - actively driving all outputs back against 0 ('Qmin'-heuristic)

Drifting Q-values: Hint-to-Goal heuristic

- Motivation: We assume that a proper policy exists (not necessarily known), that is able to keep the process in the goal region, where each transition causes zero immediate costs.
- then, for some states within the goal region we know, that $J^*(s) = 0$.
- Often, states for which the above is true, can be easily guessed a priori (e.g. being in the centre of the goal area)
- Idea: generate artificial training patterns s, a, with potential target costs 0, and set their target value to 0

Drift of Q-Function - Qmin

HTG heuristic works well for many problems, but to define the 'hint-to-goal' input pattern, all state variable values and all actions must be known a priori.
Sometimes, this is not the case in a practical application.
Idea of Qmin heuristic:

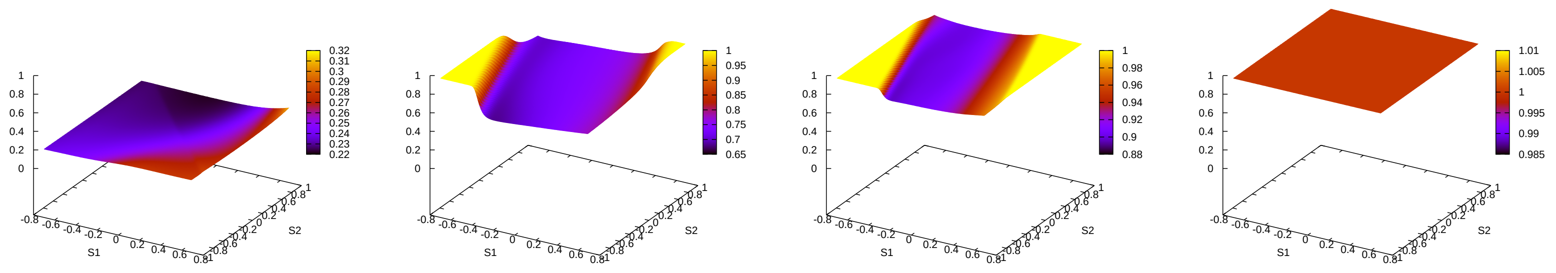
- minimum value of $\hat{Q}_{min}$ in training set is a estimate for recent drift in all Q-values
- correct the target values with this estimate:

$$Q^{t'} = Q^t - \hat{Q}_{min}$$

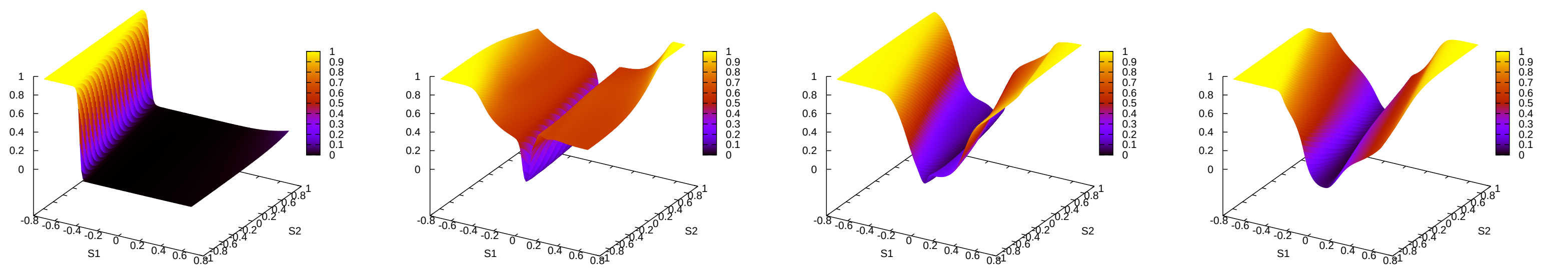
In practice, the Qmin heuristic has proven to be highly effective

Drifting Q-values: appyling anti-drift methods

Example Cart-Pole Balancing. No Anti-Drift:



With Anti-Drift (here: HTG heuristic)

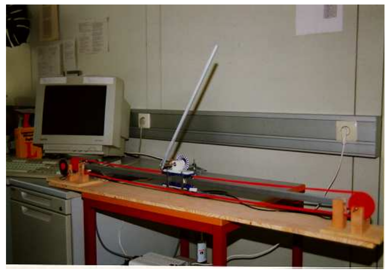


Example: Control of a real cart pole system

good: pole stands upright and cart is at desired position

bad: cart hits boundary

NFQ setup:



- actions $\pm 12V, 0V$
- HTG-heuristic at input state (0,0,0,0)
- 5-20-20-1 MLP
- speed and angular velocity approximated by differences

Special challenges:

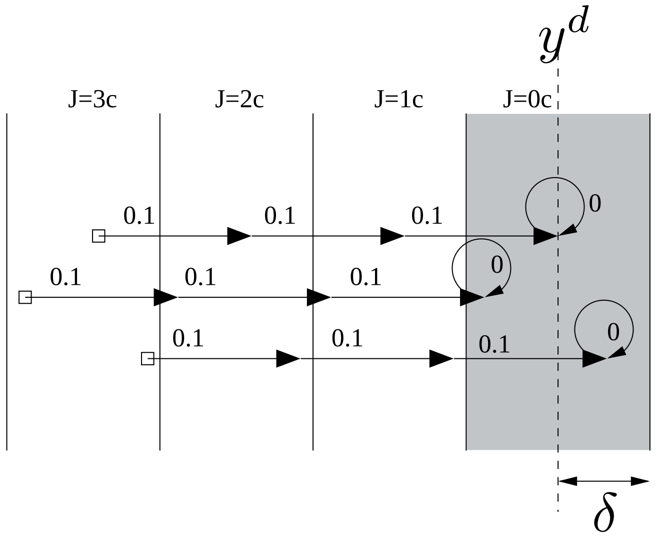
- learn directly on real system
- no model, no simulation
- learn without human interaction

Improving Accuracy

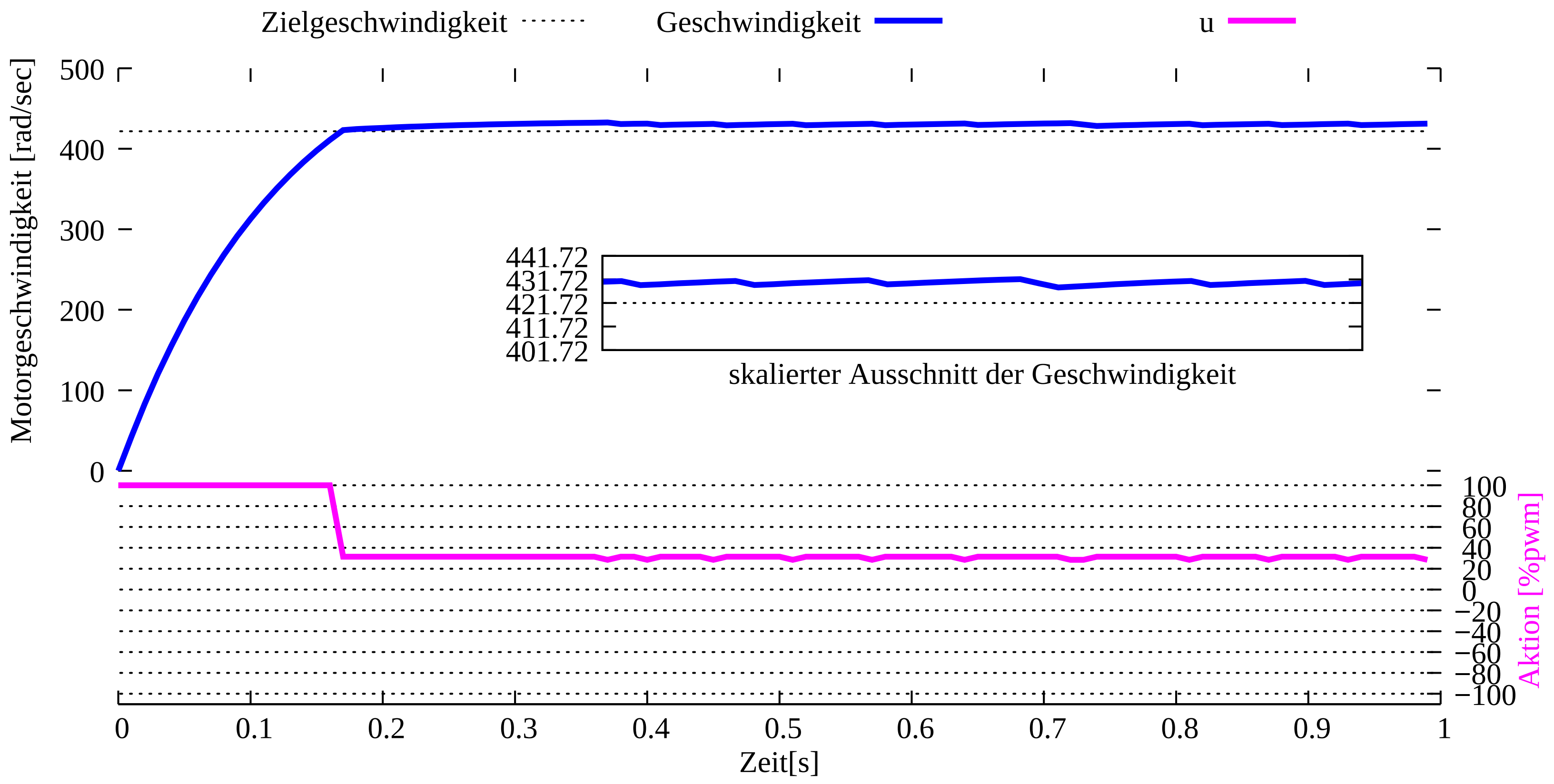
Problem: the current standard choice of the cost-function is looking for a policy, that keeps the plant output within a tolerated region $y^d \pm \delta$: $c(s,u) = 0 \Leftrightarrow y \in [y^d - \delta, y^d + \delta]$
trade-off choosing δ:

- large enough, such that system output can be kept within region
- as small as possible, such that accuracy is as good as possible

$\Rightarrow$ difficult to determine a priori.



Example: Learned set point regulation using standard cost function

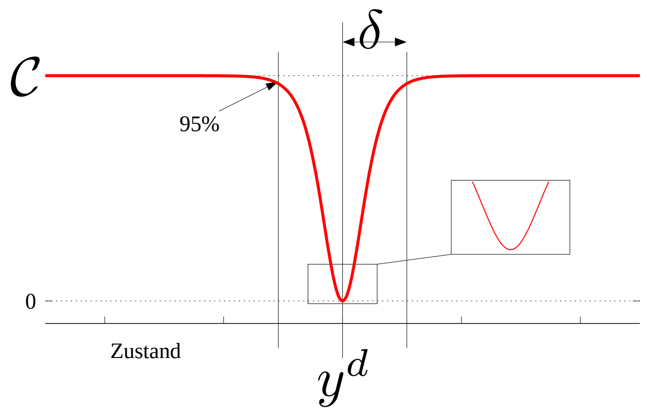


Example: Control to a single set point. Controller is optimal with respect to specified cost function.

Improving accuracy: alternative immediate cost formulation

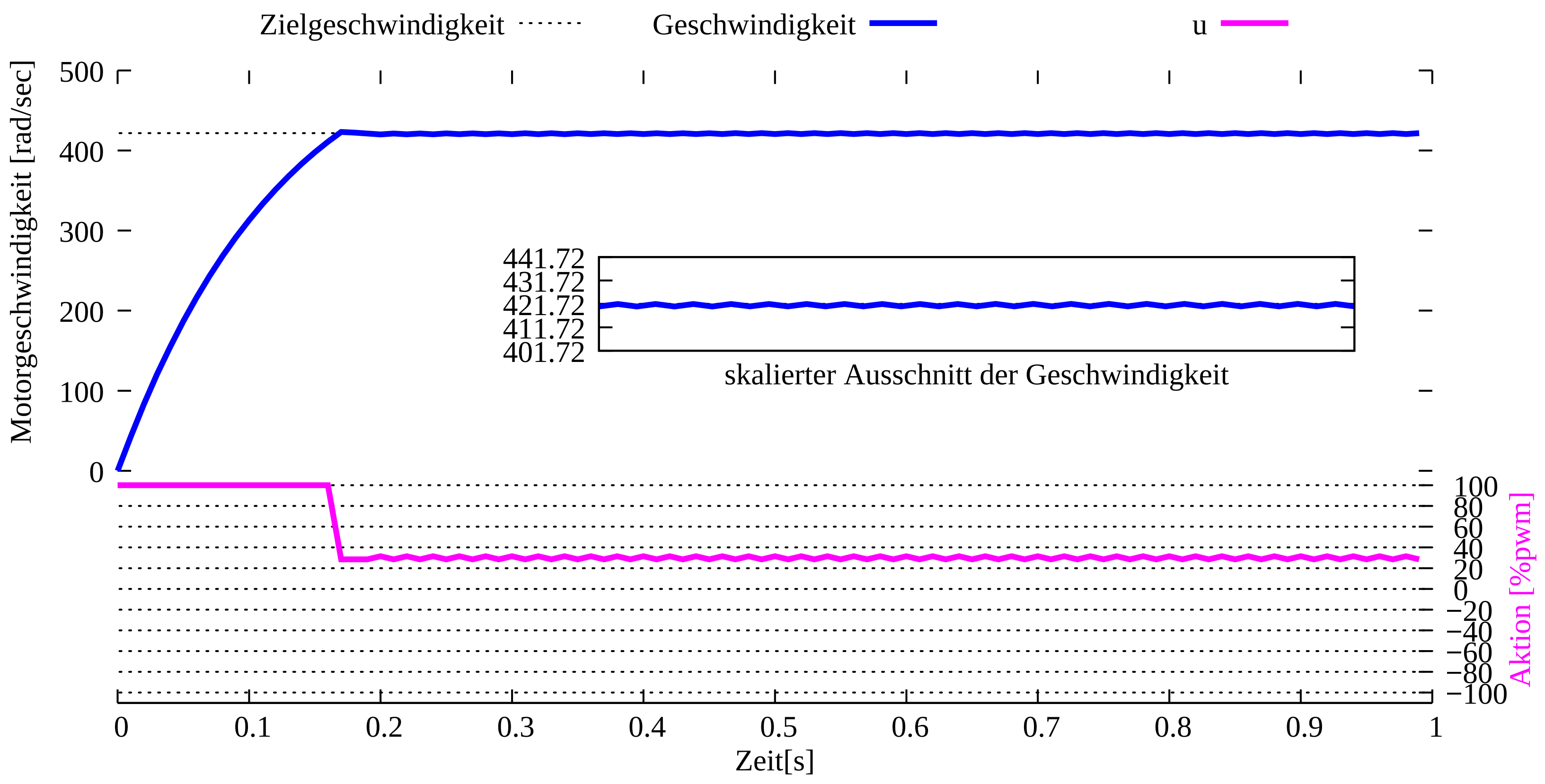
Idea: using a smoothed version of the immediate cost function

$$\begin{aligned} c(s,u) &= c(s) \\ &= \tanh^2(|s - s^d| * w) * \mathcal{C} \\ w &= \tanh^{-1}\left(\frac{\sqrt{(0.95)}}{\delta}\right) \end{aligned}$$



Note: might require slight discounting, since $c(s,u) = 0$ only if $y = y^d$

Example: Learned set point regulation using $tanh()^2$

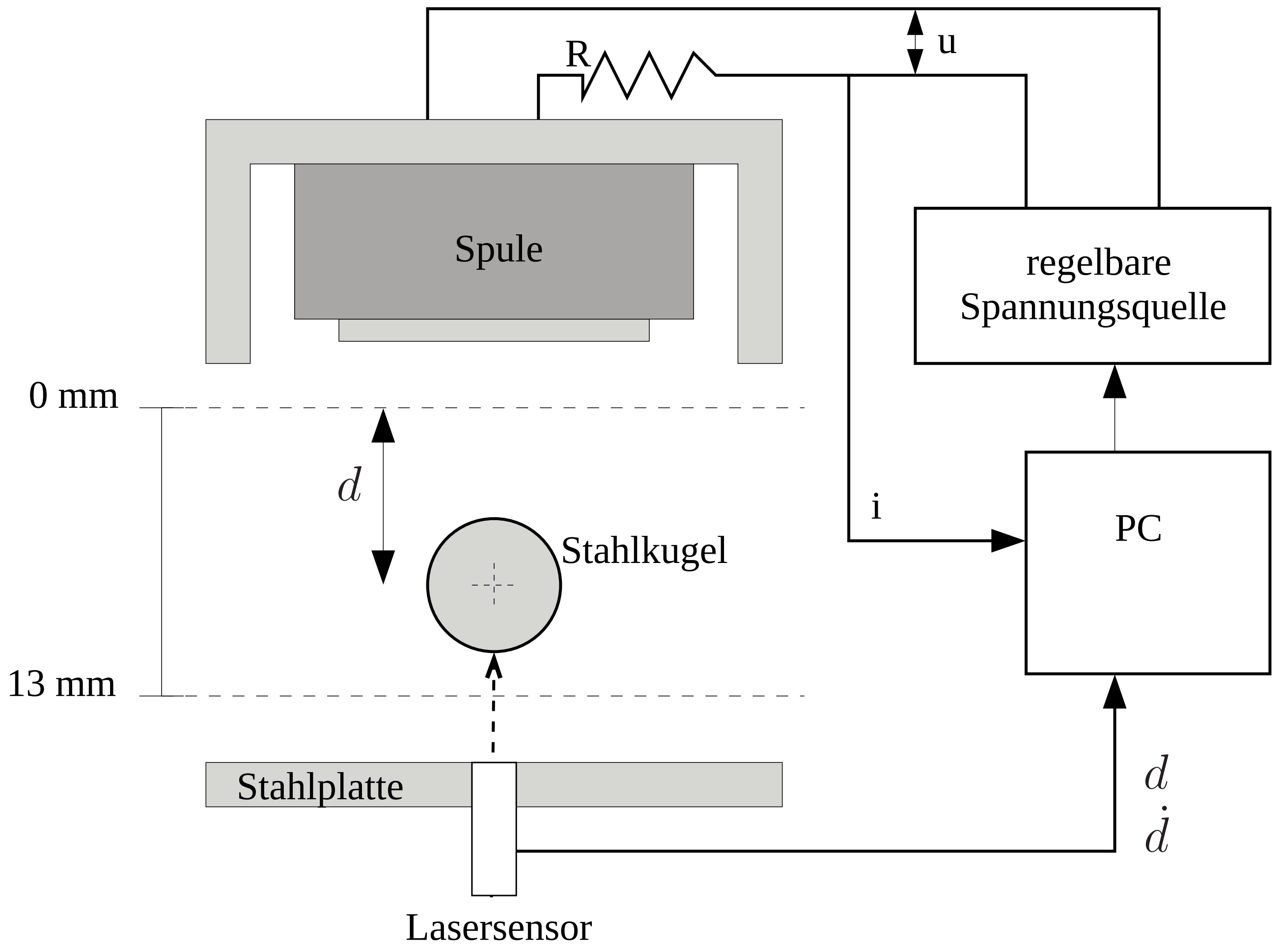


Example: Control to a single set point. Using $tanh()^2$, the control accuracy is significantly improved

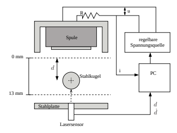
IV

Case studies

Magnetic Floating



Magnetic Floating: Learning task setup



State	4 dimensional	d	position ball
		$\dot{d}$	speed ball
		I	current
		$e = d_d - d$	error in pos
immediate costs	$tanh^2$	x^e	$(-,-,-,0)$
		μ	(0, 0, 0, 0.002)
		c	0.01
Actions	1 dimensional	u	voltage
Q-function	neuronal		5-15-20-1
policy-function	neuronal		4-15-1
Exploration	ϵ-greedy	ϵ	0.15

Magnetic Floating

- NFQCA training in max. 400 episodes with N= 160 ($\Delta t = 0.004s$)

  Demo (training with exploration)   (fast)

- After 90 episodes (<1min interaction)

  Demo

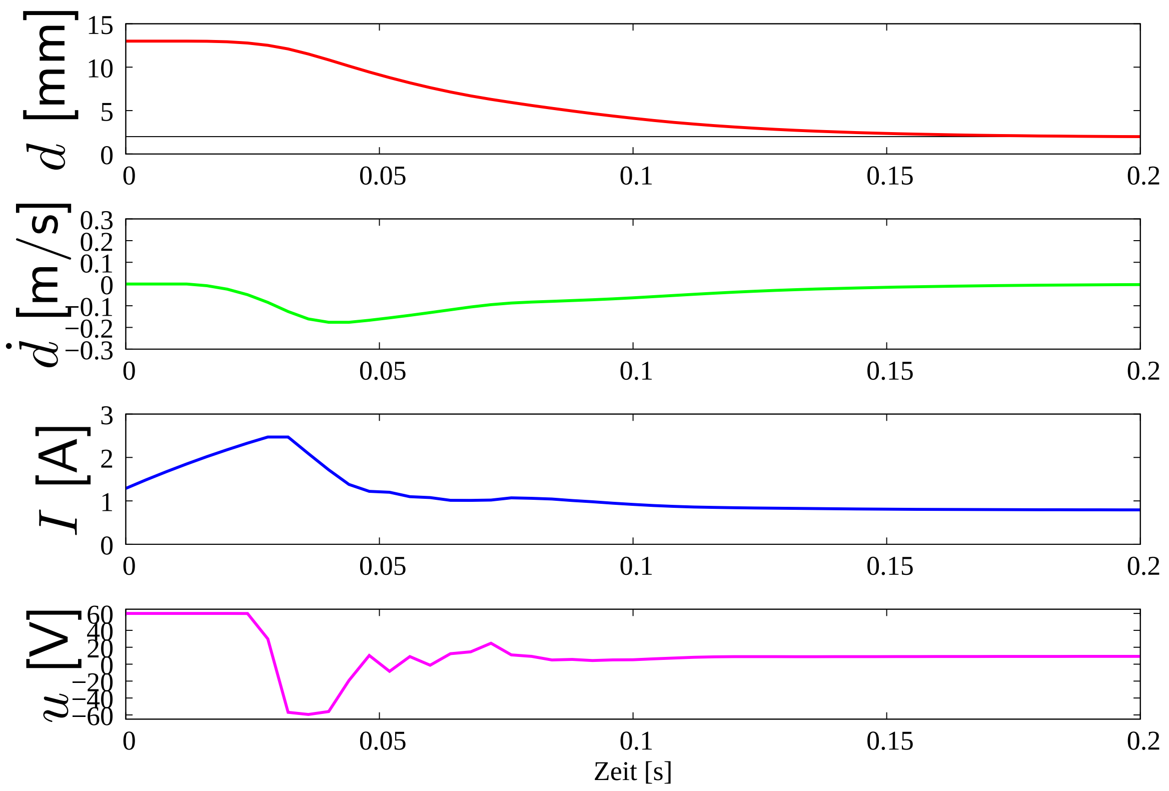
- tracking control

  Demo: tracking control

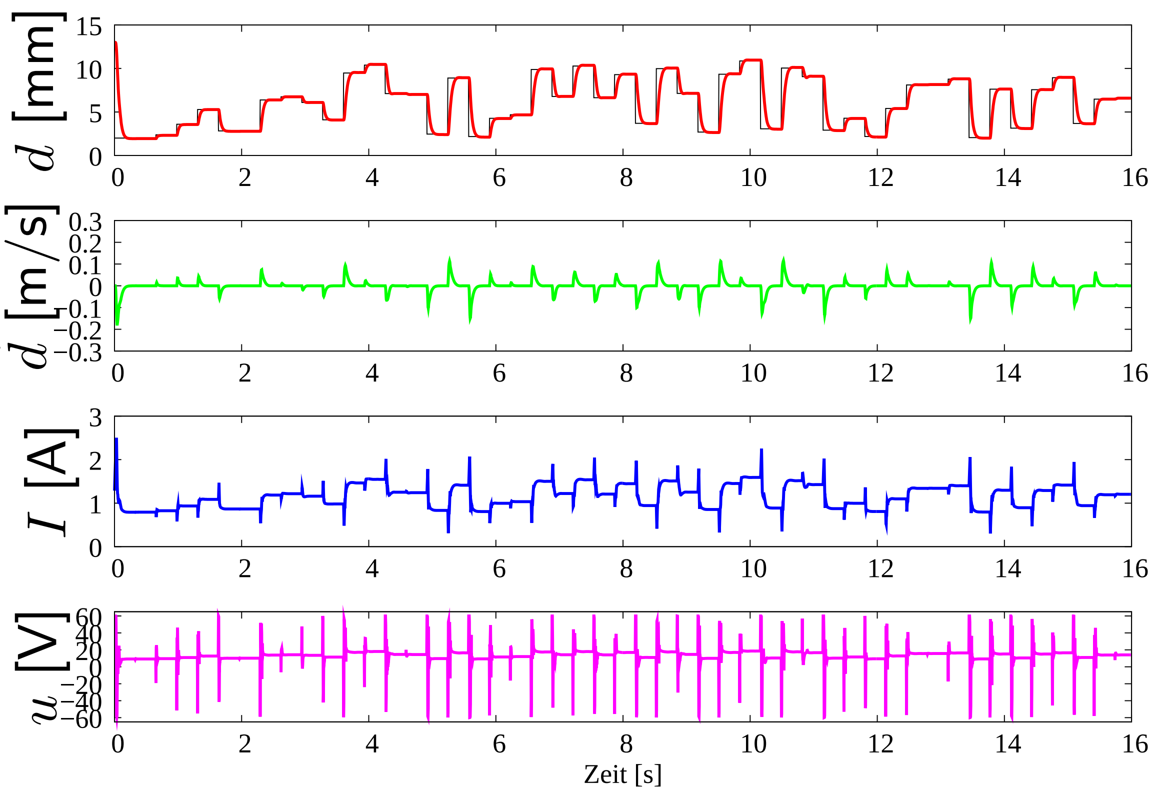
- tracking control possible

  Demo: tracking control

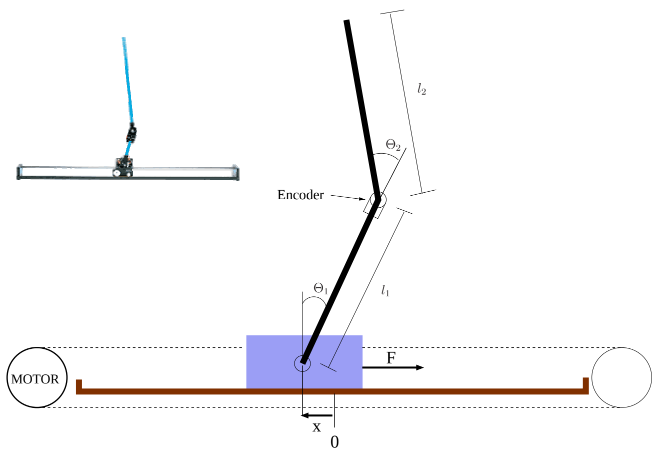
Magnetic Floating



Magnetic Floating



Double cart pole



x	position of cart	$\in$ [-0.3, 0.3]	[m]
$\dot{x}$	speed of cart		[m/s]
θ_1	angle pole 1	$\in$ [-0.25, 0.25]	[rad]
$\dot{\theta}_1$	angular velocity pole 1		[rad/s]
θ_2	angle pole 2	$\in$ [-0.25, 0.25]	[rad]
$\dot{\theta}_2$	angular velocity pole 2		[rad/s]
u	actions (voltage)	$\in$ [-6, 6]	[V]

Double cart pole

- max. 500 episodes   Demo
- after 136 episodes balances forever

  Demo: test trajectories   (fast)
- Best NFQCA controller after 248 episodes

  Demo: test trajectories   (fast) ,   Demo: single trajectory   (loop)   (slomo)
- Best NFQ controller with discrete actions after 364 episodes

  Demo: test trajectories

Further examples

- Aircraft Pitch : NFQCA Profil 1 NFQCA Profil 2 ; NFQ Profil 1 NFQ Profil 2
- Bus-Suspension : ohne Regelung NFQCA

Neuro Dribbling for a soccer robot

Task: 'Dribbling:' Moving to a target direction without loosing the ball.

state dim: 5 (rel. speed (x,y), rot. speed, angle to target direction, ballposession)
actions: 4 (pairs of target speeds in forward and sideward direction)
$\triangle t$: 33 ms

- offline sampling: random sampling (100 episodes) - NFQ (100 iterations) - greedy sampling (100 episodes) - NFQ (100 iterations)
- standard set-point cost function

$$c(s,u) = \begin{cases} 0 & , \quad \text{if } |\theta - target| < 5^o \text{ ('setpoint area')} \\ 0.01 & , \quad else \end{cases}$$

$Q(s,a) = 1,$ if ball is lost (failure)

Neuro Dribbling Results

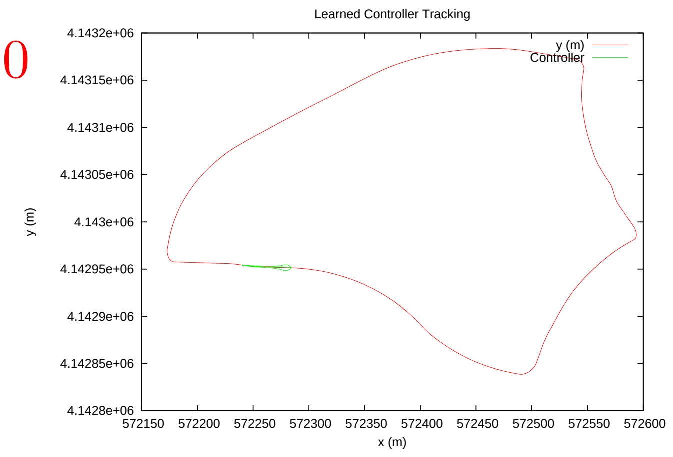
- decent human interaction in two phases of data sampling of about 15 minutes each.
- two offline NFQ learning phases of about 3 hours each
- used for dribbling in Brainstormer's World Champion Team RoboCup 2007
- won Technical Challenge Award RoboCup 2007

Learning to steer a real car (2006)

- task: smooth track following
- research stay at S. Thrun's lab, Stanford
- 6 dim. state: cross-track-error (cte) and time derivative of error, speed of rear wheels, heading error and yaw-rate matching, current steering wheel angle
- actions: changes in steering wheel angle
- $c(s,u) = \begin{cases} 0 & , \text{ if } |cte| < 0.05m \text{ and } u == 0 \\ 0.01 & , \; else \end{cases}$

  $Q(s,u) = 1, \text{if } |cte| > 0.5m$ (failure)

- embedded RL controller (parallel learning and control)



Neural steering wheel control - summary

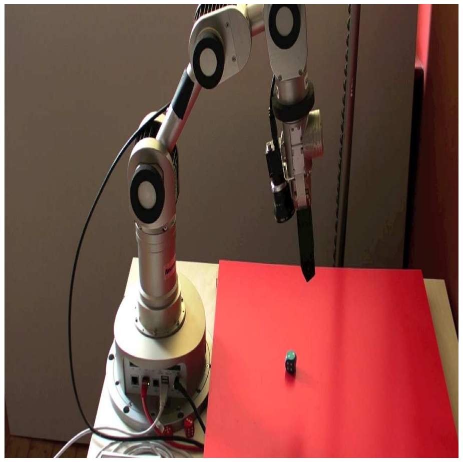
- no prior knowledge
- learned directly in real car (4 passengers)
- alternative fall-back controller during learning phases
- failure free and reasonably smooth steering in less than 20 minutes driving time

Video

RL in visual servoing

Task: Vision based closed loop control to grasp objects in arbitrary position.
No kinematic/ dynamic model of the arm is provided.
Camera mounted on hand.

states: joint pos and speed, object pos in camera.
actions: small joint movements

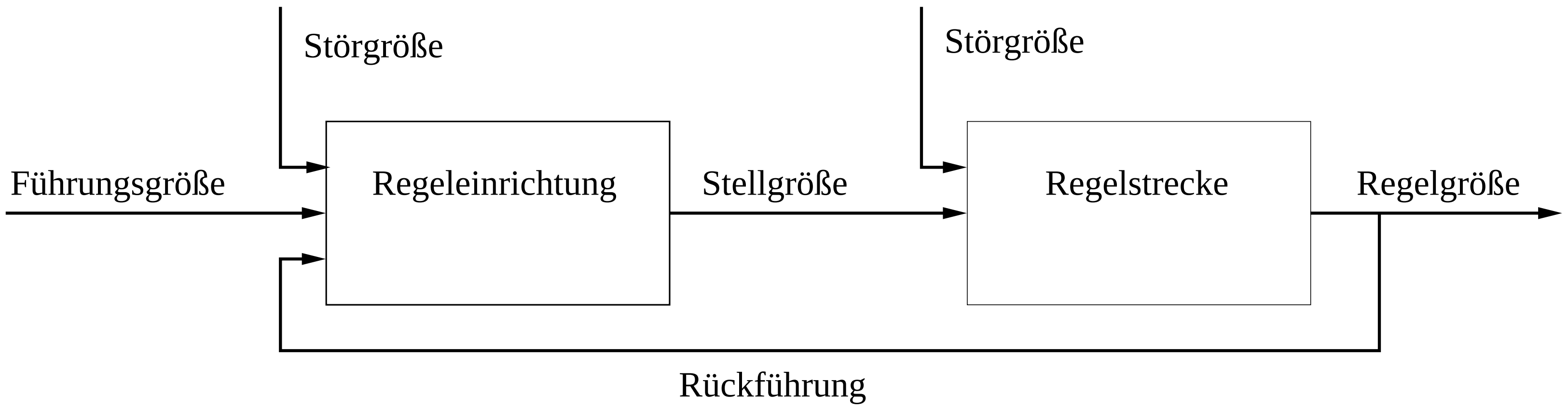


- about 300 trials (episodes)
- standard set-point cost function

$$c(s,u) = \begin{cases} 0 & , \text{ if object is in gripper} \\ 0.01 & , \ else \end{cases}$$

$Q(s,a) = 1,$ if object is out of camera (failure)

CLS^2 - closed loop simulation for learning controllers



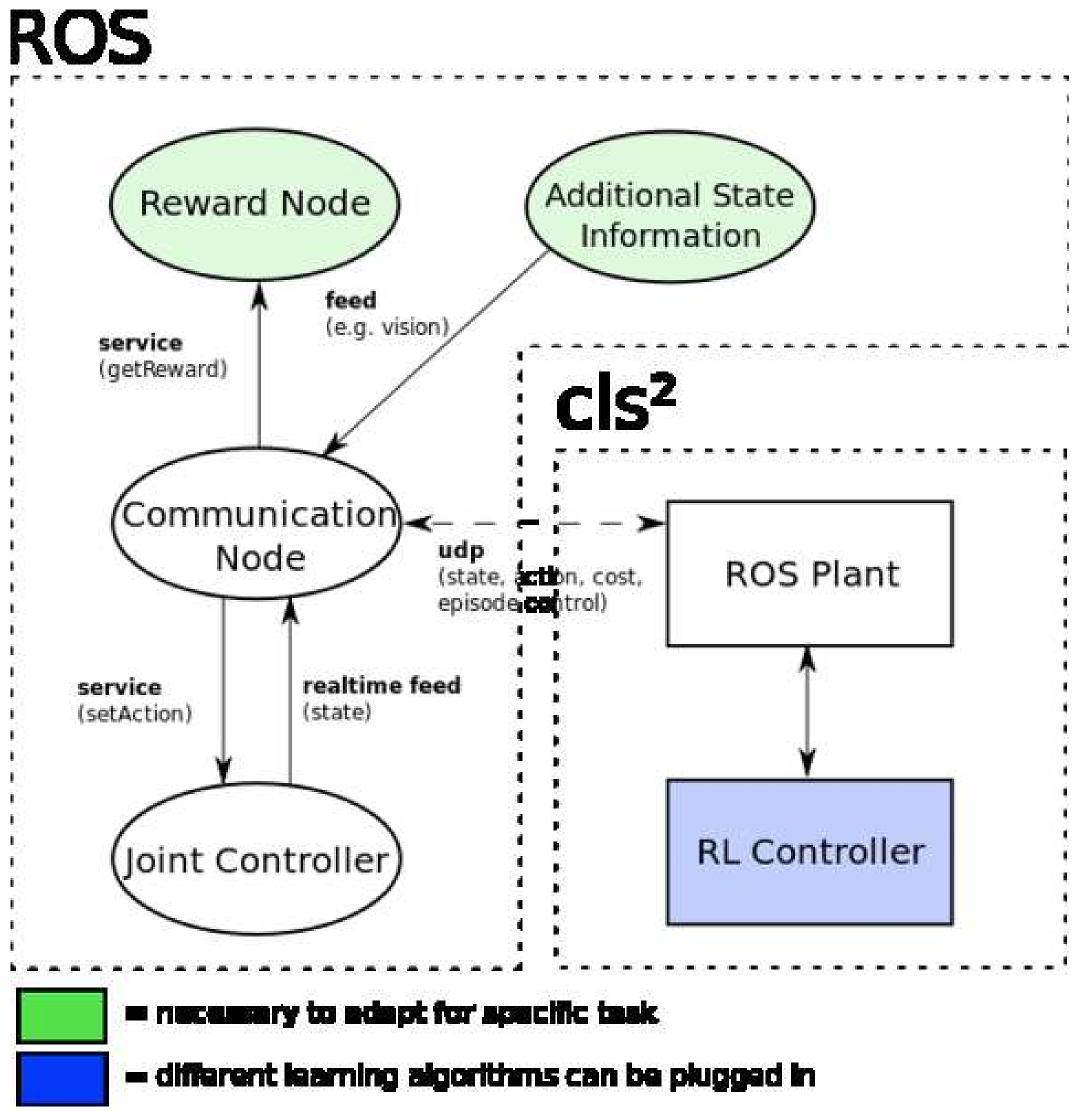
- open source
- many plants
- both simulated and real plants
- many controllers
- general modules: graphics, statistics, ...

CLSquare in ROS

- Goal: Solve highly dynamic control tasks with the PR2 using Reinforcement Learning
  - To achieve this, we need to embed a general RL framework into the ROS architecture.
  - Instead of implementing a complete RL framework inside ROS, we choose to make cls^2 compatible with it.

$\Rightarrow$The PR2 robot becomes just another plant (or MDP, respectively) within the CLS^2 framework ('action in, state out').

Implementation - Scheme


ROS
Reward Node
Additional State Information
service (getReward)
feed (e.g. vision)
Communication Node
udp (state, action, cost, episode control)
service (setAction)
realtime feed (state)
Joint Controller
cls²
ROS Plant
RL Controller
= necessary to adapt for specific task
= different learning algorithms can be plugged in

Implementation - Node Description

Joint Controller:

- runs with up to 1000 Hz
- is plugged into the pr2_controller_manager
- is implemented realtime safe
- can be exchanged online with other controllers
- can be configured to control different sets of joints
- publishes joint state information

Communication Node:

- handles the communication with cls^2 via UDP using a custom protocol
- receives actions from cls^2 and sends them to the joint controller
- collects state and reward information and sends them to cls^2

Implementation - Node Description

Reward Node:

- given state information, calculates the current reward and detects terminal state

Additional State Information:

- collects additional data from the PR2's sensors, e.g. visual information

Implementation - Node Description

ROS Plant:

- handles the communication with ROS
- sends action and episode control commands to ROS
- receives state information and reward from ROS and routes them to the RL Controller

RL Controller:

- the general cls^2 controller class
- here we can plug in arbitrary learning algorithms

Ongoing work

- Learning on abstract states (Stephan Timmer)
- Multi-agent learning (Scheduling, Thomas Gabel)
- RL in computer games (Thomas Gabel)
- Vision-based RL (Deep neural networks, Sascha Lange)

Summary

- (Neural) Fitted Q Iteration: highly data efficient RL learning method
- learning from scratch, model-free
- Multi-layer perceptrons are pretty robust, generalise well and can deal with large amounts of data
- provide useful gradient information
- sampling of transition data very flexible: randomly, with different policies, different time-scales ...
  reuse of data possible
- direct application to real plants possible
- PR2: currently implementing a closed control loop in ROS, that fits to our learning framework CLS^2