Reinforcement Learning in Motor Control

Lasse Scherffig
University of Osnabrück

August 21, 2002

Abstract

In the field of machine learning, reinforcement learning constitutes the idea of enabling machines to learn how to deal with a given environment by means of direct exploration. As opposed to supervised learning in reinforcement learning no explicit examples of how to act are presented to the learning agent. Instead a learner tries out actions and observes reinforcement signals that give an account of an improvement in the state of the environment. This Bachelorthesis first gives a short overview of the theoretical foundations of reinforcement learning and then presents ways of applying the theory of temporal difference learning to a robotic task in a simulated environment. Besides regular Q-Learning the usage of artificial neural networks for reinforcement learning is discussed.

Contents

1 What is Reinforcement Learning? 4

2 The Theory of Reinforcement Learning 4
2.1 Basic Concepts 4
2.1.1 Reinforcement Signals 4
2.1.2 Agent and Environment 5
2.2 Formal Definitions 5
2.2.1 Terms and Functions 5
2.2.2 The Goal of Reinforcement Learning 6
2.3 Markov Decision Processes 7
2.3.1 The Markov Property 7
2.3.2 Markov Decision Processes 8
2.3.3 Action and Action-Value Functions for Markov Decision Processes 8
2.4 Possible Solutions 9
2.4.1 Generalized Policy Iteration 10
2.4.2 Dynamic Programming 11
2.4.3 Monte Carlo Methods 11
2.4.4 Temporal Difference Learning 12

3 From Theory to Realworld Tasks: Algorithms for Learning and Approximating Action Values 12
3.1 Watkins Q-Learning 12
3.1.1 Formulas 12
3.1.2 Action Selection 13
3.1.3 The Algorithm 14
3.1.4 Convergence of Q-Learning 14
3.1.5 Shortcomings and Problems 15
3.2 Q-Learning with Artificial Neural Networks 16
3.2.1 Artificial Neural Networks 16
3.2.2 Using Neural Networks to Approximate Q 16
3.3 Coding of Data 17
3.4 Modular Learning 18

4 Into Action: Learning to Solve the Inverted Pendulum Problem 18
4.1 Motivation 18
4.2 Physics and Simulation 19
4.2.1 Derivation of the Differential Equations 19
4.2.2 Simulation and Control 22
4.3 Tasks 22
4.4 Architectures 23

4.5 Balancing 23
4.5.1 Regular Q-Learning in the Balancing Task 24
4.5.2 Q-Learning with Neural Networks in the Balancing Task 26
4.5.3 Q-Learning versus Q-Learning with Neural Networks . 27
4.6 Full Control 27
4.6.1 Regular Q-Learning in the Full Control Task 27
4.6.2 Q-Learning with Neural Networks in the Full Control Task 27
4.6.3 Modular Learning in the Full Control Task 29

5 Summary **32**

1 What is Reinforcement Learning?

Most techniques of machine learning are forms of "supervised learning" or "learning with a teacher". They require a set of training data having the form of examples. In motor control tasks this approach would need explicit examples that tell the learning agent which action to take in a certain situation. If situations or states of the environment are called s and the actions the learner can take are called a, examples of the form $\langle s, a \rangle$ - pairs of states and actions to take in those states - would be needed. The teacher thus would already have to know how to act in (at least) some stereotypical situations and mediate this knowledge to the learning agent.

However, it is not very likely that natural organisms learn to deal with new environments using some sort of supervised learning. We rather would expect (at least in many cases) a biological organism to learn to deal with new circumstances by "trial and error". Given an unknown situation, the organism could try *some* action and observe the following state of the environment. If the outcome of the action just taken would improve the situation for this organism in some way, we would expect it to try the same action in the same situation again. This approach - learning by trial and error - is the core idea of reinforcement learning.

In neurobiology and neuropsychology the term used for exactly this kind of learning - "trial and error learning" or "learning the association between behavior and a reward" - is *operant conditioning* [Kandel et al., 2000, p.1242].

2 The Theory of Reinforcement Learning

2.1 Basic Concepts

2.1.1 Reinforcement Signals

The most important concept in reinforcement learning is the idea of *reinforcement signals* or *rewards*. Trial and error can only yield useful results, when it is possible to decide if the last action taken was an error or not. The learner needs some kind of evaluative feedback, to be able to detect an improvement (or the opposite) in the situation following an action. Thus, solving a problem using reinforcement learning implies that there is an *evaluation function*, a function that is able to map states of the environment to some sort of numerical scale.

This evaluation function often is called a *critic* [Barto, 1995/1, p.804]. Reinforcement learning hence is not a form of supervised learning, but also not mere unsupervised learning (as self organizing maps, etc.). It rather can be placed somewhere between those approaches, while the critic in reinforcement learning holds some of the role of the teacher in supervised learning.

As we saw, for supervised learning in motor control, training data would have the form $\langle s, a \rangle$. As opposed to that, reinforcement learning in motor control is based on training data consisting of a state of the environment, an action taken by the agent and the immediately received reward: $\langle \langle s, a \rangle, r \rangle$. Of course this implies that the goal the learning agent is intended to reach can be comprised by the reinforcement signal - a demand that has to be met by the designer of the evaluation function.

2.1.2 Agent and Environment

Any agent in artificial intelligence can be specified by defining its percepts, actions, goals and environment [Russel and Norvig, 1995]. This also holds for agents that do reinforcement learning.

All reinforcement learning agents (independent of the algorithms used, etc.) have two features in common:

1 Their percepts include the reward signal that evaluates the current situation.

2 Their goal is to maximize cumulative rewards, that is to reach the optimal sum of rewards over time.

In motor control, as well as in many other cases, the learning task is an *associative* task. The agent seeks to learn an association between situations and actions to be taken given the environment is in this situation. Generally in associative reinforcement learning the percepts of the learning agent also include information about the environment - the state of the system. In the following only the associative case of reinforcement learning will be covered.

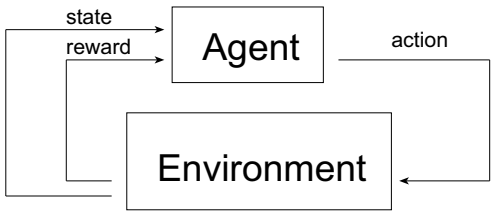

Figure 1: The agent-environment interface for associative reinforcement learning.

2.2 Formal Definitions

2.2.1 Terms and Functions

To deal with reinforcement learning on a theoretical foundation the ideas behind it have to be defined formally. The basic concepts in the theory of reinforcement learning are:

- The agent can take actions a from the set of all possible actions A. The action taken at timestep t is named a_t.
- The agent can observe the environment, that is it can observe the current state s from the set of all possible states S. The state at timestep t is named s_t.
- The agent receives rewards $r \in \mathbb{R}$ that somehow measure the "goodness" of the current state s. Again, the reward received at timestep t is named r_t.
- To choose actions, the agent follows a *control policy*, a mapping from states to actions $\pi : S \mapsto A$. The policy determines which action a the learning agent will take, given the environment is in situation s.

Based on the concepts defined above, there are two more important terms *value functions* and *action-value functions*.

- Value functions - $V^\pi(s)$ - represent the value of states $s \in S$, assuming that the agent follows a given policy π.
- Action-value functions - $Q^\pi(s, a)$ - represent the value of taking action $a \in A$ in state $s \in S$, assuming that the agent follows policy π.

2.2.2 The Goal of Reinforcement Learning

In these terms the general goal of any associative reinforcement learning agent is to find an *optimal* control policy. This optimal policy is normally named π^*. But what constitutes an optimal policy? Since the only measure the learning agent can rely on here is the reinforcement signal, an optimal policy can be considered every policy that maximizes the received rewards.

Thus, as mentioned above, the general goal of any reinforcement learning agent is to gain maximal rewards. This general goal however - maximizing received rewards - can be used to make a reinforcement learning agent achieve any specific goal that can be described in terms of the evaluation function.

Possibly a final goal of a learning agent may take a very long sequence of steps to be reached. Because of that it is useful not only to focus on the immediate reward following the current state, but also involve all rewards following that state and all future states. Thus, the learning agent should try to maximize the expected sum of all rewards it will receive in the future - it should maximize *cumulative rewards*.

But a reinforcement learning process, especially in control tasks, might take an infinite long time. This unfortunately might always lead to an infinitely high reward for any action. To deal with this problem, rewards can be *discounted*. That is, future rewards that are more than one timestep away

from the current state are not fully taken into account but multiplied with a discount factor that approaches zero for very distant time steps. Using a discount factor, the term the agent should maximize is the *cumulated discounted reward*[1]:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1}$$

The factor γ is called the *discount rate* and it should hold that $0 \leq \gamma \leq 1$. Since the role of future rewards is discounted by the factor γ^k, the closer γ gets to one the more the agent will take the far future into account, while a γ close to zero will make the agent only rely on rewards rather close in time.

Generally a reinforcement learning agent is searching an optimal policy from the set of all possible policies. During this search the reward signal is reminiscent of a heuristic function that guides the agent on its search through the policy space [Ballard, 1999, p.230].

2.3 Markov Decision Processes

2.3.1 The Markov Property

As we have seen, a control policy is a function mapping states of the environment to actions. But why can we assume that it is sufficient to observe the current state of the system to be able to decide which action to take?

In fact, generally we can *not* assume that. There may be environments in which the *whole* sequence of past states and actions that lead to the current state must be known in order to be able to decide optimally.

Learning in such environments, however, would be hard if not impossible. To be able to learn efficiently, we need an environment in which the state signal "summarizes past sensations compactly, yet in such a way that all relevant information is retained" [Sutton and Barto, 1998, p.62]. Knowledge of the *current state* hence should be enough to decide which action to take next. We do not want to be forced to keep track of the *whole* sequence of states and actions that occurred before the current state.

This claim to the environment (or the state signal) is called *Markov property* [2]. Formally a system fulfills the Markov property if the probability that the state s_{t+1} following the current state s_t is a certain state s' and the observed reward r_{t+1} is a certain reward r solely depends on the current state and the action taken by the agent.

$$Pr\left\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\right\} \tag{2}$$

[1] In some publications the term *return* is used instead of reward.

[2] The Markov property sometimes is called *independence of path* property.

2.3.2 Markov Decision Processes

If a reinforcement learning process takes place in an environment that fulfills the Markov property it is called a *Markov Decision Process* (MDP). Moreover, if the sets of actions and states A and S are finite it is called a *finite Markov Decision Process*. For finite Markov Decision Processes the probability of reaching state s' from state s by choosing action a can be formally defined as:

$$P^a_{ss'} = Pr\left\{s_{t+1} = s' \mid s_t = s, a_t = a\right\} \tag{3}$$

Similarly the expected reward of the state s' following state s after choosing action a is given by:

$$R^a_{ss'} = E\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{4}$$

Most of the theory in reinforcement learning only deals with finite Markov Decision Processes. In fact, general theorems proving convergence of reinforcement learning processes only exist within the framework of finite MDPs. But, as we will see later, often it is useful to apply algorithms developed for finite Markov Decision Processes to problems dealing with continuous state or action spaces or systems that lack the Markov property.

2.3.3 Action and Action-Value Functions for Markov Decision Processes

Assuming that the given environment satisfies the Markov property, it is possible to give a formal definition of action and action-value functions.

The value of a state s while following policy π is still given by the expected (and discounted) reward following s but assuming that the state signal is a Markov signal, this expectation merely depends on the current state.

$$\begin{aligned} V^\pi(s) &= E_\pi\left\{R_t | s_t = s\right\} \\ &= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \end{aligned} \tag{5}$$

Similarly, the value of taking action a in state s in a finite Markov Decision Process merely depends on the current state, the chosen action and the resulting state.

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\left\{R_t | s_t = s, a_t = a\right\} \\ &= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \end{aligned} \tag{6}$$

Considering the fact that the policy the reinforcement learning agent is searching for is the optimal policy π^*, it is useful to have a look at the action and action-value function for this optimal policy. The value function for π^* is called V^* while its action-value function is called Q^*.

Since the optimal policy is the policy maximizing cumulative discounted rewards, it holds that

$$V^*(s) = \max_\pi V^\pi(s)$$
$$Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

This yields the following equations:

$$\begin{aligned} V^*(s) &= \max_a E\left\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right\} \\ &= \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^*(s')\right] \end{aligned} \tag{7}$$

This means that the value of a state under an optimal policy is the maximal reward that can be received immediately plus the discounted value of the resulting state.

For the action-value function of π^* holds essentially the same:

$$\begin{aligned} Q^*(s,a) &= E\left\{r_{t+1} + \gamma \max_a Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\right\} \\ &= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma \max_a Q^*(s', a')\right] \end{aligned} \tag{8}$$

2.4 Possible Solutions

Since the 1950s there is research dealing with the problem of optimal control in Markov environments [Sutton and Barto, 1998, p.16]. While the first solutions where the purely mathematical approaches of Dynamic Programming, currently it is possible to distinguish three different general approaches to deal with Markov Decision Processes:

- Dynamic Programming (DP)
- Monte Carlo Methods (MCM)
- Temporal Difference Learning (TD-Learning)

All of them have their advantages and shortcomings. The only approach enabling an agent to learn by direct interaction with minimal knowledge of the environment in minimal time, however, is the last one. Therefore, Dynamic Programming and Monte Carlo Methods will only be treated superficially while the algorithms presented later solely come from the field of Temporal Difference Learning.

Before introducing these three techniques it is useful to introduce the general concept behind all of them: The concept of Generalized Policy Iteration.

2.4.1 Generalized Policy Iteration

One feature the three general techniques mentioned above have in common is the idea of *Generalized Policy Iteration*.

Recall that the goal of all those approaches was finding an optimal policy π^*. Originally the agent does not know this policy and thus will probably follow another one. Therefore, the agent will have to *improve* its current policy whenever this is possible.

Improving the policy, however, is only possible if the agent is able to *evaluate* policies - if it can decide which of two given policies is better than the other.

Thus, two processes are needed: *Policy evaluation*[3] and *policy improvement*. It is possible to do both processes simultaneously and interacting. This technique constitutes the idea of Generalized Policy Iteration. It is a very general idea and the basis of most reinforcement learning techniques. In fact, all reinforcement learning problems can be described in terms of Generalized Policy Iteration [Sutton and Barto, 1998].

At each incremental step of GPI, the current state is evaluated using the current policy. That means that the current state is evaluated using the expected reward under the policy currently followed. This process, policy evaluation, makes "the value function consistent with the current policy" [Sutton and Barto, 1998, p.105]. Then the current policy is improved using the current value function. This, for instance, can be done by choosing actions in a *greedy* manner with respect to the current value function. That is, by always choosing actions that maximize the expected reward for the future.

During the Generalized Policy Iteration process on the one hand the value function is shifted towards the actual value function of the current policy, while on the other hand the current policy is shifted towards a greedy policy. When this process converges, the resulting policy must necessarily be the optimal policy π^* and the value function must be the corresponding value function V^*. This holds because both processes only can stabilize if

- the value function is the value function of the current policy; and
- the current policy is greedy with respect to the current value function.

This implies that the resulting policy must be greedy with respect to its *own* value function, which s just a reformulation of (7). Thus, the resulting policy must fulfill those equations and therefore it must be an optimal policy π^*.

[3]Since policy evaluation means predicting the value of a given state under a certain policy, sometimes policy evaluation is referred to as the *prediction problem*.

2.4.2 Dynamic Programming

If the reinforcement learning agent possesses a complete model of its environment, it is easy to do policy evaluation. Assuming that the environment constitutes a finite Markov Decision Process the value of a state, given a policy π is simply $V^{\pi}(s)$, which can be mathematically computed here because of the complete model. Therefore, Dynamic Programming can be defined as a Generalized Policy Iteration process:

- Policy evaluation: To evaluate a state under a policy π, mathematically compute the immediate reward and all further states and rewards using the model of the environment.
- Policy improvement: Improve the current policy by choosing actions greedy with respect to the current value function.

A very important feature of Dynamic Programming is that here the value of a state is calculated using estimated values that are based on estimated values. That is, the value of a state involves the values of states the agent expects the environment to take in the future. This feature - estimating on the basis of estimates - is called *bootstrapping*.

2.4.3 Monte Carlo Methods

While dynamic programming provides a mathematical solution to the reinforcement learning problem, it suffers from the necessity of having a complete model of its environment as well as from the (possibly) enormous computational expenses needed to calculate all future states, actions and rewards. If a reinforcement learning agent is intended to learn to control its environment *without* any knowledge about it besides the state and reward signals then other approaches are needed.

A first idea here could be to do policy evaluation by just testing what rewards certain states yield under the given policy. This is the basic idea behind a class of algorithms that are called *Monte Carlo Methods*.

Following the scheme of Generalized Policy Iteration, Monte Carlo Methods can generally be defined as follows:

- Policy evaluation: To evaluate a state under a policy π, generate a number of sample episodes starting in that state and average the rewards gained.
- Policy improvement: Improve the current policy by choosing actions greedy with respect to the current value function.

One could say that the major drawback of this approach is its biggest advantage as well: It does not involve bootstrapping. Here the only basis for

estimating the value of a state is an averaged value composed of actually observed rewards. No further estimations are needed. This is an advantage because algorithms that do not involve bootstrapping have proven to be more robust against violations of the Markov property [Sutton and Barto, 1998, p.130]. On the other hand this is an important disadvantage because many very long sample episodes may be needed to provide useful estimations for values of states.

2.4.4 Temporal Difference Learning

If on the one hand it is possible to estimate the value of a state on the basis of other estimates and on the other hand we can compute the value of a state by observation, can't we combine both ideas into a single method? The term *Temporal Difference Learning* summarizes approaches that do bootstrapping as well as learning through interaction, that therefore integrate ideas from Dynamic Programming and Monte Carlo Methods.

In terms of Generalized Policy Iteration, Temporal Difference Learning can be defined this way:

- Policy evaluation: To evaluate a state under a policy π, use the *observed* immediate reward and the *estimated* future rewards.
- Policy improvement: Improve the current policy by choosing actions greedy with respect to the current value function.

Here the estimated future rewards needed for policy evaluation can be obtained from the current value function. During the learning process the value of the current state is estimated using the observed immediate reward and the output of the current value function for the immediately following state.

This approach involves learning from direct interaction since the only signal used to learn from is the immediate reward - no model of the environment is needed. It also involves bootstrapping since the value function learned so far is used to estimate the value of successor states. How this is done algorithmically is shown in detail in the next section.

3 From Theory to Realworld Tasks: Algorithms for Learning and Approximating Action Values

3.1 Watkins Q-Learning

3.1.1 Formulas

Within the setting of finite Markov Decision Processes it is possible to formulate the value functions and action-value functions recursively:

$$V^{\pi}(s) \quad = \quad E_{\pi}\left\{R_t \mid s_t = s\right\}$$

$$
\begin{aligned}
&= E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\} \\
&= E_\pi \left\{ r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s \right\} \qquad (9)
\end{aligned}
$$

$$
\begin{aligned}
Q^\pi(s,a) &= E_\pi \left\{ R_t \mid s_t = s, a_t = a \right\} \\
&= E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\} \\
&= E_\pi \left\{ r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a \right\} \qquad (10)
\end{aligned}
$$

Using those recursive formulas for V and Q now it is possible to do exactly what constitutes the TD-learning approach: evaluating states (or state-action pairs) on the basis of the immediate reward r_{t+1} and the estimated value of the succeeding state s_{t+t}. In motor control it is advisable to learn Q instead of V, which is exactly what the algorithm *Q-learning* does.

Q-Learning is a rather simple but useful algorithm and was first suggested by Watkins in 1989 [Watkins, 1989]. It is still one of the most important algorithms in reinforcement learning.

For Q-Learning the Q function is stored in a table. Each state-action pair has one entry in this table. This entry represents the Q value for the state-action pair.

During learning table entries are updated using the recursive formula (10) for Q. Of course storing each state-action pair in a table is only possible if the number of states and actions is limited, which fortunately is one property of finite Markov Decision Processes.

Using the notation of Generalized Policy Iteration, a formulation of Q-Learning is:

- Policy evaluation: To evaluate a state s use the observed immediate reward and the estimated value of the successor state using the stored table. Then Update the stored action-value in the table using those values.
- Policy improvement: Improve the current policy by choosing actions greedy with respect to the current action-value function Q.

3.1.2 Action Selection

For all reinforcement learning techniques described so far it was stated that they do policy improvement by choosing actions in a greedy manner. This, however, is not always the best choice and it is worth to think of other possibilities.

During the learning process it might be extremely important for success that the learning agent from time to time does not take the action that

seems optimal at that moment but another seemingly suboptimal one. This would prevent the learner from getting stuck in local minima. Thus, there is a conflict between the need to always take the seemingly best action to get optimal rewards and the need for some kind of random exploration of the policy space. This conflict is generally named the "conflict between exploration and exploitation" [Barto, 1995/1].

A widely used solution here is not to take the seemingly best action with probability one, but to take it with a probability of $1 - \varepsilon$ and to choose a random action with probability ε. This approach is called ε-greedy instead of greedy.

It has been shown that ε-greedy strategies often perform much better than pure greedy strategies, especially in long learning processes [Sutton and Barto, 1998, p.29].

3.1.3 The Algorithm

The algorithm yielded by the idea of Q-Learning is very simple and can be implemented quite easy:

```
Initialize Q(s,a) arbitrarily
Observe starting state s
Repeat:
Choose a from s using policy derived from Q (e.g.  greedy)
Take action a, observe r, s'
Q(s,a) ← r + γ max_a Q(s',a)
s ← s'
```

Note that the update rule here is:

$$Q(s,a) \leftarrow r + \gamma \max_a Q(s',a) \tag{11}$$

It may be advisable to use a stepsize parameter α for the update rule, which yields the same algorithm with the modified update rule:

$$Q(s,a_t) \leftarrow Q(s,a_t) + \alpha \left[r + \gamma \max_{a_{t+1}} Q(s',a_{t+1}) - Q(s,a_t) \right] \tag{12}$$

The simple update rule (11) can be derived from (12) if $\alpha = 1$.

3.1.4 Convergence of Q-Learning

Is is possible to prove convergence of Q-Learning under certain assumptions:

- The environment has to be a finite Markov Decision Process.
- Immediate rewards may not be infinitely high.

- Every state-action pair must be visited infinitely often.

The idea behind the proof is that if every state-action pair is visited infinitely often, we can define a *full interval* as an interval in which each state-action pair is visited once. It is possible to prove mathematically that during a full interval the largest error in the table is reduced by γ. After infinitely many visits of each state-action pair there have been infinitely many full intervals. Therefore the largest error in the table must necessarily approach zero[4].

3.1.5 Shortcomings and Problems

Q-Learning is a rather simple and quite efficient reinforcement learning algorithm. However, it suffers from several shortcomings mostly based on the demand that the environment has to be a finite Markov Decision Process.

In the real world the Markov property is not always exactly met and - even more important in dealing with physical systems - state (and sometimes action) spaces are almost always continuous and not discrete. Thus, even dealing with physical systems that fulfill the Markov property it is very likely that the given Markov Decision Process is not finite.

Furthermore, even if the state space is finite it may be extremely large, depending on the number of input dimensions. Generally the state space grows exponentially with the number of input dimensions, a feature often referred to as *the curse of dimensionality*.

The question of how to deal with continuous input or action spaces or high dimensional spaces is still subject of current research. There are no general results yet, however, for many tasks there are interesting approaches.

First the number of input dimensions can be scaled down by some sort of intelligent coding of data such as tile coding, hashing or coarse coding [Sutton and Barto, 1998, p.202ff].

Secondly a widely used approach is storing the Q function using a parameterized function instead of a discrete table. Here any general function approximator can be used and many of them have been. For instance, radial basis function networks, linear gradient descent methods or artificial neural networks have been successfully used within the setting of Q-Learning or similar algorithms.

The next section takes up this idea, showing how to use nonlinear feed forward neural networks for approximating the Q function.

[4]More information on this proof can be obtained from [Mitchell, 1997, p.377ff]. The proof was first published in [Watkins and Dayan, 1992].

3.2 Q-Learning with Artificial Neural Networks

3.2.1 Artificial Neural Networks

Artificial neural networks and especially feed forward neural networks are well studied function approximators. The general idea behind artificial neural networks is storing a function in parameterized form: in the form of weighted connections between nonlinear numerical units, called neurons.

It is possible to train feed forward neural networks to approximate any given function using supervised learning. To do this, input patterns are presented to the network and the output of the network resulting from this input pattern is computed. Then the output is compared to the desired output associated to the input pattern[5]. The error between actual and desired output is used to adjust the weighted connections of the network in order to be reduced. The general technique used here is nonlinear gradient descent, the most basic algorithm for this is known as *backpropagation*. In most cases the error signal for standard backpropagation is the mean squared error[6].

3.2.2 Using Neural Networks to Approximate Q

As mentioned above, one solution to the problems Q-Learning suffers from could be using a neural network instead of a table to represent Q.

This approach can be used to do both: To reduce memory usage by storing the parameters defining the network instead of the table and for applying Q-Learning to continuous state or action spaces. However, in most cases input and action spaces are discretized by some form of coding of data even if neural networks are used.

The general idea enabling us to use artificial neural networks for reinforcement learning is using the update rule for learning Q as the error signal for backpropagation. The neural network then can take the role of the table in Q-Learning: It can output estimated Q values for states and actions as well as incrementally learn the value of state-action pairs.

This of course constitutes a from of supervised learning. Doing supervised learning here, although reinforcement learning is not a form of supervised learning, is possible since the output of the network for the current state can be compared to the reward actually received after selecting an action.

The resulting algorithm is very similar to regular Q-Learning:

[5]Recall that in supervised learning for each input pattern in the training data, there is an output pattern that belongs to it.

[6]More detailed information on neural network learning and backpropagation can be found in [Hammer, 1999], [Zell, 2000], [Ballard, 1999] and many more publications.

Initialize network with random small weights
Observe starting state s
Repeat:
Choose a from s using policy derived from Q (e.g. greedy)
Take action a, observe r, s'
Update Q by backpropagating the error signal given by
$E = Q(s,a) - r + \gamma \max_a Q(s',a)$
$s \leftarrow s'$

An important point here is that dropping the table representation of Q means dropping the security of guaranteed convergence. Reinforcement learning using function approximation thus can only be seen as an abstraction of the technique of Q-Learning to non Markov tasks. Convergence theorems only exist for the discrete, tabular case. The algorithms derived from the theory of finite Markov Decision Processes *can* be transferred to slightly different domains, but then there are no theoretical claims on performance, convergence and usefulness.

As experiments have shown, this approach can be very successful. Maybe the most famous experiment here is TD-Gammon by Gerald Tesauro, who used reinforcement learning with neural networks to create a system that learned to play Backgammon. After a long training sequence, playing against itself, the system has shown to be able to beat human world champions in this game [Tesauro, 1995].

3.3 Coding of Data

No matter how Q-Learning is done - using a table, neural networks or other means - one of the most important questions is the question how to code data. As in any machine learning problem, successful learning crucially depends on the representation of input and output spaces.

In Q-Learning dealing with the curse of dimensionality can be done in several ways. Tile coding can be used, a technique where each the input vector is mapped to an index number. Very large input spaces can be scaled down by some sort of hashing [Sutton and Barto, 1998, p.207] or the input space may be separated into discrete entities using domain knowledge about the environment.

All these approaches can be used in Q-Learning with artificial neural networks too. Using neural networks, however, there are additionally many possible ways of representing Q. For discrete and rather small action spaces, it is often useful to use one distinct neural network for the Q value of each possible action [Wengerek, 1996, p.26]. One could also include the action into the input pattern and use only one neural network at all. Or even do not map states to Q values, but rather use a neural network to map states to actions [Ritter et al., 1990, p.127ff].

There are no general results on the question of how to represent states and actions for any kind of reinforcement learning. Domain knowledge enabling the designer of a reinforcement learning agent to present states or actions in a way that suits the environment thus can be extremely important. Also empirical experiments with different learning architectures and different ways of representing input and output data may be very useful.

3.4 Modular Learning

For tasks requiring complex interaction with the environment it may be advisable to separate the problem into subtasks of smaller complexity. Several forms of modular or hierarchical learning strategies are possible. The general approach here is to design learning agents for simple tasks and a control structure that decides which agent should deal with the current state of the environment. The control structure in this setting can hand the state signal to one of the learners, obtain the action this agent proposes, execute it and mediate the reward signal back to the agent.

If the designer of the learning system possesses enough domain knowledge, the control structure can be designed in a way that suits the environmental conditions. Otherwise the control structure itself can be a reinforcement learner that learns which states should be mediated to which learning agent.

4 Into Action: Learning to Solve the Inverted Pendulum Problem

4.1 Motivation

Consider a pendulum that is attached to a cart as shown in Figure 2. The cart can move freely in x direction.

The goal of a reinforcement learning agent is to keep the pendulum in upright position or to bring it into upright position and keep it there. This problem is often referred to as the *inverted pendulum* or *pole balancing* problem.

It is a standard problem in control theory. The most widely used techniques to control such a cart are based on the theory of fuzzy control. Sometimes supervised learning is used to train neural networks to take the role of such a fuzzy controller [Liebscher, 2000]. Of course also reinforcement learning techniques have been applied to this problem.

Two facts make the inverted pendulum problem a very well suited problem for reinforcement learning:

- As many dynamical physical systems an *idealized* cart and pole perfectly fulfill the Markov property.

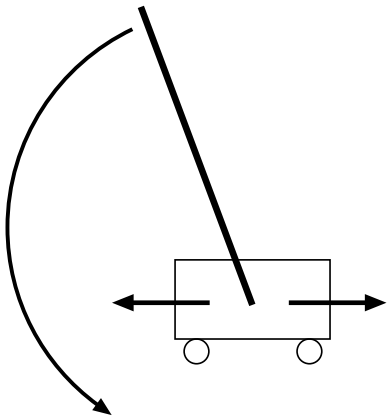

Figure 2: Pendulum and cart.

- It is impossible to mathematically compute general solutions of the nonlinear equations describing the physics of the pole cart system. Thus, it is impossible to calculate an optimal control force at an arbitrary time step.

The pole balancing problem can be seen as an abstraction of the general problem of balancing. It is of relevance for robotic tasks and also a question of interest for computational models of biological organisms. For instance problems very similar to the inverted pendulum problem have to be solved by robots and biological organisms that walk upright [Ritter et al., 1990, p.130].

4.2 Physics and Simulation

As stated above, there is no trivial solution for the differential equation describing the behavior of the pendulum. But given a state of the system, it is possible to calculate its succeeding state by means of *interpolation*. Therefore, simulation of the pendulum can be done iteratively, calculating each new state from the preceding one[7].

4.2.1 Derivation of the Differential Equations

Pendulum The force influencing the behavior of the pendulum is the force F_T acting tangentially to the circular motion. It is given by the mass of the pendulum, the gravity and the sine of the current angle of the pendulum.

$$F_T = -m \cdot g \cdot \sin(\varphi) \tag{13}$$

[7]For detailed information on the formulas used here and the dynamics of the system please see [Bergman and Schaefer, 1943]

Where m is the mass of the pendulum, g is the acceleration due to gravity and φ is the angle of the pendulum.

Now the torque (or moment of force) M_P of the pendulum can be computed:

$$M_P = F_T \cdot \frac{l}{2} = \frac{-m \cdot g \cdot l}{2} \cdot \sin(\varphi) \tag{14}$$

With l denoting the length of the pendulum (Note that $\frac{l}{2}$ is the distance between the axis of rotation and the barycenter B of the pendulum).

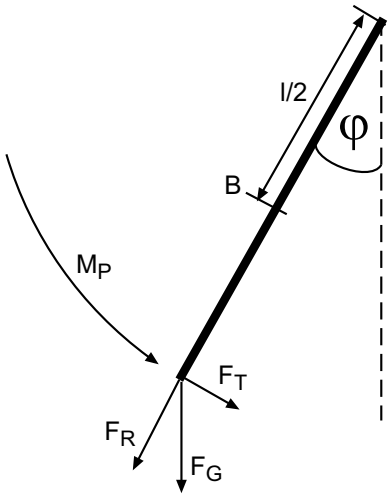

Figure 3: The pendulum and the important variables describing it

Cart The second force that is important for the behavior of the pendulum is the force F_C caused by acceleration of the cart. It is given by:

$$F_C = -m \cdot \ddot{x} \cdot \cos(\varphi) \tag{15}$$

Where $\ddot{x}$ is the current acceleration of the cart.

Given this force, it is possible to calculate the torque M_C that results from the acceleration of the cart:

$$M_C = F_C \cdot \frac{l}{2} = \frac{-m \cdot \ddot{x} \cdot l}{2} \cdot \cos(\varphi) \tag{16}$$

Of course the pendulum is retarded by several sorts of friction, caused by air and the bearing. For simulation purposes it is sufficient to assume a constant friction fc acting against the angular velocity.[8] This friction yields a torque M_{Fr}.

$$M_{Fr} = -\dot{\varphi} \cdot fc \tag{17}$$

[8]This of course is a rather rough approximation, but it should not have *qualitative* effects on the dynamics of the system.

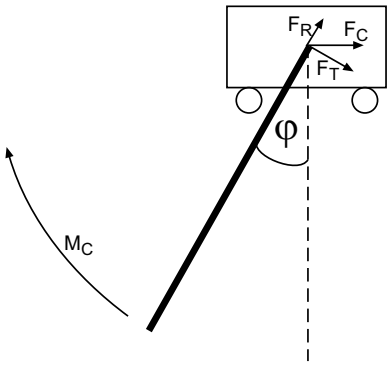

Figure 4: The cart and its important variables

The Total System The resulting torque M_{res} of the system can be computed by summing up all those torques. It hence is:

$$M_{res} = M_P + M_C + M_{Fr} \tag{18}$$

To compute the angular acceleration of the pendulum using the torque, the moment of inertia J of the pole is needed. If the axis of rotation lies in the barycenter of the pole, then the moment of inertia J_B is given by:

$$J_B = \frac{1}{12} \cdot m \cdot l^2$$

Since the axis of rotation here is at one end of the pole, the moment of inertia J has to be computed using the parallel-axis theorem:

$$J = J_B + m \cdot r^2$$

Where r is the distance from the barycenter to the axis of rotation. For the pendulum this distance is $\frac{l}{2}$. This yields the following equation:

$$J = \frac{1}{12} \cdot m \cdot l^2 + m \cdot \left(\frac{l}{2}\right)^2 = \frac{1}{3} \cdot m \cdot l^2 \tag{19}$$

The relation between torque, moment of inertia and angular acceleration is given by:

$$M_{res} = J \cdot \ddot{\varphi} \tag{20}$$

Solving this equation yields a resulting angular acceleration $\ddot{\varphi}$:

$$\ddot{\varphi} = \frac{M_{res}}{J} = \frac{-\frac{m \cdot g \cdot l}{2} \cdot \sin(\varphi) - \frac{m \cdot \ddot{x} \cdot l}{2} \cdot \cos(\varphi) - \dot{\varphi} \cdot fc}{\frac{1}{3} \cdot m \cdot l^2} \tag{21}$$

Angle and Angular Velocity Assuming that the angular acceleration at moment t is not altered during a constant time step dt, it is possible to compute the new angle and angular velocity at moment $t + dt$. This, as stated above, constitutes an interpolation of the behavior of the pendulum and only yields valid results if dt is small enough to neglect changes in the angular velocity during dt. However, if dt is chosen rather small, this interpolation gives results that are sufficient for the simulation.

$$\varphi_{t+dt} = \varphi_t + \dot{\varphi}_t \cdot dt + \ddot{\varphi} \cdot dt^2 \tag{22}$$

$$\dot{\varphi}_{t+dt} = \dot{\varphi}_t + \ddot{\varphi} \cdot dt \tag{23}$$

Position and Velocity of the Cart Similarly, the new position x_{t+dt} and new velocity $\dot{x}_{t+dt}$ of the cart at moment $t + dt$ can be computed by:

$$x_{t+dt} = x_t + \dot{x}_t \cdot dt + \ddot{x}_t \cdot dt^2 \tag{24}$$

$$\dot{x}_{t+dt} = \dot{x}_t + \ddot{x}_t \cdot dt \tag{25}$$

4.2.2 Simulation and Control

The whole simulation process now consists of calculating the interpolated values for the new state variables of the system using the current state, a constant interval in time dt and a given acceleration of the cart $\ddot{x}$ during dt. This acceleration is the control value that can be altered by the reinforcement learning agent in order to solve the inverted pendulum problem.

For the actual simulation used here the mass of the pendulum is 0.1 kg, its length is 0.2 m, the length of the intervals in time dt is 0.005 s, the acceleration due to gravity g is $9.81 \cdot \frac{m}{s^2}$ and the friction constant fc is 0.001.

4.3 Tasks

Now a learning agent intended to solve the pole balancing problem in the simulation can be defined. This definition depends on the exact task the agent has to solve. While it is possible to think of various tasks, only a some special tasks will be treated here:

- Balancing: The pendulum is initialized in a near-upright position; the agent has to learn to keep it upright. The simulation is reset whenever a critical angle is passed[9]
- Full control: The pendulum is initialized arbitrarily; the agent has to learn to bring it into upright position and keep it there.

[9]The balancing task is the task that is most widely used in reinforcement learning experiments. In fact, the author did not find any paper trying to solve other tasks for an inverted pendulum with reinforcement learning.

While in a real world environment the position of the cart would be important since it cannot move infinitely long into one direction, here its position will not be taken into account - assuming an infinite long track the cart can move on.

4.4 Architectures

Depending on the algorithm - regular Q-Learning, Q-Learning with neural networks or modular learning - and depending on the way states and actions are coded one can think of lots of architectures that might solve the tasks defined above. The learning architectures used here just constitute some ideas of dealing with the problem.

All these architectures can be seen as learning agents and hence can be described in terms of their percepts, actions, goals and environments. Here all of them share the same actions, goals and percepts while they might differ in their environment, the algorithm they apply and the way input data is coded.

- Actions: For all reinforcement learning agents used here the set of actions - accelerations of the cart - was discrete. It consisted of three accelerations: $-10\frac{m}{s^2}$, $0\frac{m}{s^2}$ and $10\frac{m}{s^2}$. All learning agents shared the same action selection strategy, which was an ε-greedy strategy with $\varepsilon = 0.01$.

- Goals: As for any reinforcement learning agent the goal here was to maximize discounted cumulative rewards. Since the position of the cart was not taken into account here it was sufficient to use a reward function that is maximal when the angle φ of the pendulum reaches $0°$. A good function having that feature is $-\varphi^2$.

- Percepts: Since the task generally was bringing the pendulum into upright position, the state variables of the system that were important for reaching the goal were the angle φ and the angular velocity $\dot{\varphi}$. These made up the state signal.

In the following some architectures dealing with those tasks are described and their performance in dealing with them is shown and discussed.

4.5 Balancing

For the balancing task the environment was set up in a way that each time a critical angle of $-90°$ or $90°$ was passed the simulation was reset to an (randomly chosen) angle between $-20°$ and $20°$.

For both algorithms the input space consisting of the angle and the angular velocity was coded using binary tile coding. That is, the two dimensional input space was mapped onto a two dimensional grid, each place in the grid

- called tile - having one unique index number n. This index number was used by the algorithms to determine the current state.

4.5.1 Regular Q-Learning in the Balancing Task

For regular Q-Learning in the balancing task, the input space was discretized as described above. The input value φ was partitioned into 4 parts between $-\pi$ and π, which is the interval between the critical angles. The angular velocity $\dot{\varphi}$ was partitioned into 20 parts, from $-20\frac{m}{s}$ to $20\frac{m}{s}$. Thus the input space consisted of 80 possible states. Since the agent could choose from 3 different actions this design yielded a table consisting of 80×3 entries. For the discount parameter γ a value of 0.7 was used, while different stepsize parameters α were tested.

Using the simple Q-Learning algorithm without a stepsize parameter (or rather with a step size parameter of $\alpha = 1$), that is using update rule (11), learning seemed to be very fast and efficient, at least on the short run. However, extending the task in time revealed another picture. As figure 5 shows, controlling the pendulum worked most of the time but was highly instable. This probably was due to the fact that the table representation of Q was altered too much during the learning process, forcing the agent to "jump over" good minima.

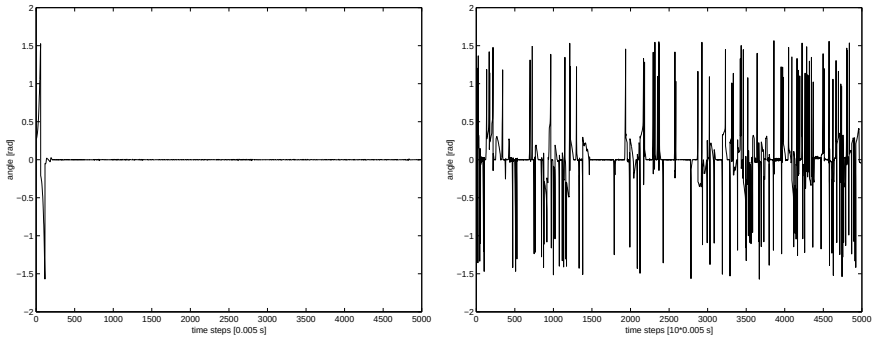

Figure 5: Q-Learning in the balancing task. While the algorithm learns extremely fast in the beginning, it stays instable during longer learning sequences. Here 5000 and 50000 steps were simulated, $\alpha = 1$.

The Role of Step Sizes A simple solution to this problem seemed to be altering the step size parameter α. Experimenting with smaller values here showed remarkable results. For this simple learning task, very small changes in the table representation of Q seemed to be necessary to enable the agent to learn efficiently while controlling the pendulum with high stability. Figure 6 gives an impression on the role of α in Q-Learning in the balancing task.

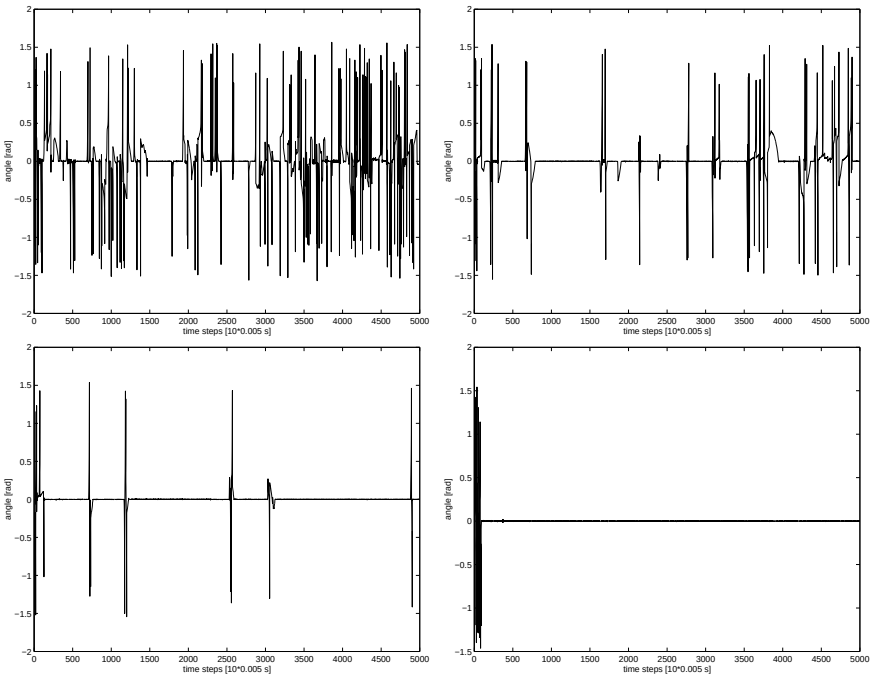

Figure 6: Q-Learning in the balancing task. Stable control largely depends on the value of α. Here 50000 steps with $\alpha = 1$, $\alpha = 0.75$, $\alpha = 0.5$ and $\alpha = 0.001$ were simulated.

4.5.2 Q-Learning with Neural Networks in the Balancing Task

For using neural network learning in the balancing task again the input space was discretized - in exactly the same way as it was done for regular Q-Learning. This lead to a neural network with 80 input dimensions. Since states were coded binary, each state was mapped to an input vector consisting of 79 zeros and a one at the index position of the current state.

Because feed forward neural networks are of far higher computational complexity than the table used for Q-Learning, here rather small networks were used. There was one network for each action, thus coding the expected Q-value for that action. Each network consisted of 80 input neurons, 5 neurons in a hidden layer and a single output neuron.

All layers were fully connected to the succeeding layer (that is, each neuron in layer n had one connection to each neuron in layer $n+1$). For the neurons in the hidden layer the nonlinear *logistic* activation function[10]

$$f(x) = \frac{1}{1+e^{-x}} \tag{26}$$

was used. The output neuron had the identity as linear activation function. The learning algorithm for the network was standard backpropagation. For the update rule, which constituted the error signal for backpropagation a discount value of $\gamma = 0.7$ and a stepsize value of $\alpha = 1$ was used.

Learning with this architecture was slower than regular Q-Learning in the same task. But, as figure 7 indicates, once the balancing behavior was learned control was very stable.

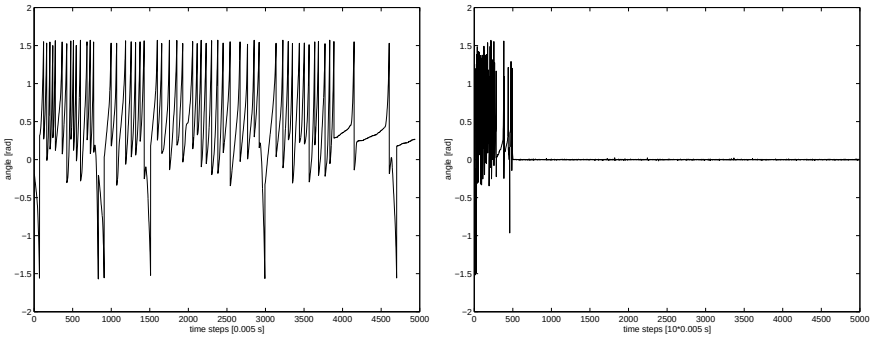

Figure 7: Q-Learning with neural networks in the balancing task. Learning with neural networks is slower than regular Q-Learning, but very stable. Again 5000 and 50000 steps were simulated.

[10]This function is often called *sigmoid* activation function.

4.5.3 Q-Learning versus Q-Learning with Neural Networks

While on a first impression it seemed that neural networks are learning slower but yielding more stable control than regular Q-Learning does, this idea cannot be held up.

It rather seems that in a simple learning task, as the balancing task is, the much lower complexity of regular Q-Learning easily beats the generalization and nonlinearity features of neural network learning. However, success in learning with Watkins Q-Learning, even in this simple environment, seems to be crucially depending on the stepsize used in the update rule for Q. Whereas learning with neural networks yielded satisfactory results without the need to do experiments with different step sizes.

4.6 Full Control

The design for the full control task differed from the design for the balancing task merely in the fact that here the simulation was not stopped or reset at all.

Again binary tile coding was used to code the input space. Actions and evaluation function stayed the same.

4.6.1 Regular Q-Learning in the Full Control Task

To use regular Q-Learning here, input values were partitioned into 8 parts between -2π and 2π. As above the angular velocity was partitioned into 20 parts between $-20\frac{m}{s}$ and $20\frac{m}{s}$. Therefore, the table to represent Q consisted of 160×3 entries. Again a discount value of $\gamma = 0.7$ and different stepsizes were used.

Even if very long learning sequences were simulated, reinforcement learning agents using regular Q-Learning were not able to solve that problem. The process ended up in a local minimum that was only slightly better than no control at all. This result seemed to be rather independent of stepsizes.

Thus, it seems likely that the simple representation of Q as a table is generally unable to cope with the complex nonlinear dynamics of a free moving pendulum.

4.6.2 Q-Learning with Neural Networks in the Full Control Task

Q-Learning with neural networks for the full control task also was set up in almost the same way as for the balancing task. Since the input space was partitioned in exactly the way as it was done for regular Q-Learning, neural networks with 160 input dimension were used.

Again for each action one network coding and learning the Q value for this action was used. In this task, however, different network architectures were tested. It did not seem to be unlikely that bigger networks might rather

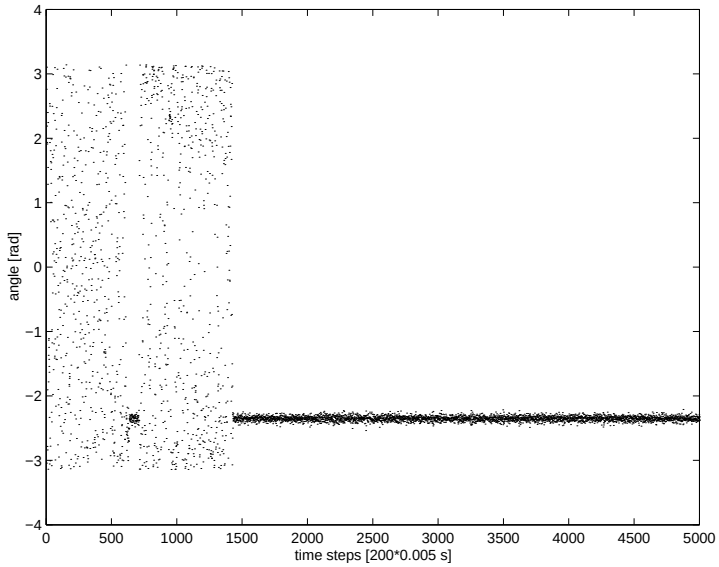

Figure 8: Regular Q-Learning in the full control task. The algorithm is only able to find a (rather bad) local minimum. Here one million steps with $\alpha = 0.001$ were simulated.

be able to deal with the nonlinearities of the pendulum. Thus, three network architectures of different internal structure and complexity were tested:

- Three networks with 160 input neurons, five hidden neurons and one output neuron.
- Three networks with 160 input neurons, a layer with 20 hidden neurons, a layer with 5 hidden neurons and one output neuron.
- Three networks with 160 input neurons, 30 hidden neurons and one output neuron.

All network architectures yielded basically the same results: No minimum in the angle φ could be reached - the learning agents failed in trying to bring the pendulum into upright position.

Probably this is because the behavior required to solve the task is contradictory: On one hand sometimes the pendulum has to be moved against its angular velocity and angle in order to keep it upright (balancing). On the other hand sometimes a more complex behavior is necessary, involving acceleration of the pendulum with its angular velocity and angle (upswing). Learning both contradicting behaviors at the same time may be impossible with feed forward neural networks.

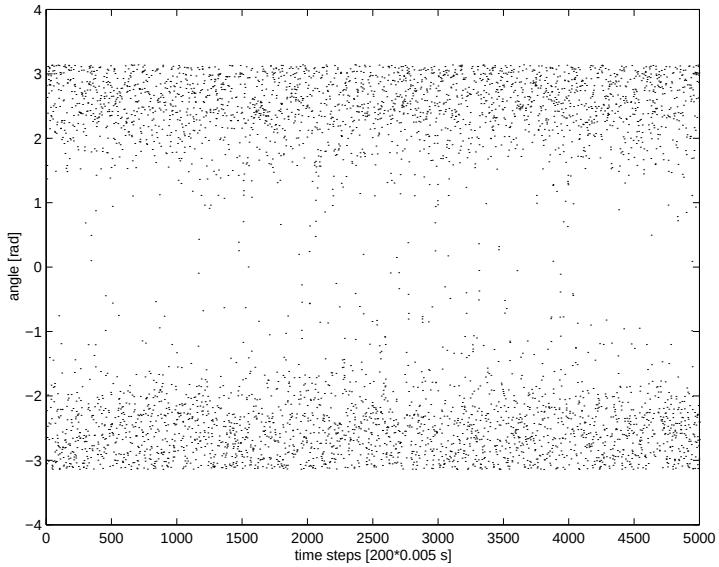

Figure 9: Q-Learning with neural networks in the full control task. The algorithm creates rather random movement, not being able to reach any minimum. Here one million steps were simulated using three networks with 160 input neurons, 30 hidden neurons and one output neuron. The results for the other architectures look essentially the same.

4.6.3 Modular Learning in the Full Control Task

The problems arising for the full control task seem to be due to an important feature: The learning agent has to learn two essentially different behaviors. This seemingly makes the full control task a perfect candidate for modular learning. Moreover, knowledge of the physical system constituted by cart and pendulum suggests a simple modularization of the task: It seems to be sufficient to have one learning agent for the upswing component and one for balancing. The control structure that is then needed just has to mediate each state in which the angle is below horizontal to the upswing agent and all other states to the balancing agent.

Modular Q-Learning First a design with two Q-Learning agents - one for each subtask - was tested. Again, input states were coded using tile coding. For the upswing agent angles smaller than $-\pi$ and bigger than π were partitioned into 4 parts, as well as angles between $-\pi$ and π for the balancing agent. Angular velocities were partitioned into 20 parts between $-20\frac{m}{s}$ and $20\frac{m}{s}$. Thus, both learning agents were designed in the same way as the learner used to solve the balancing task described above and consisted of a table with 80×3 entries.

The results looked almost the same as in applying regular Q-Learning

to the full control task. After very long training sequences only a bad local minimum was reached. Changing the stepsize parameter did not improve the situation in any way - the problem seemed to be principally not solvable by this algorithm. Probably, even the upswing part of the full control task constitutes too complex nonlinear dynamics for tabular-based Q-Learning. Of course without being able to bring the pendulum up balancing it could not be learned.

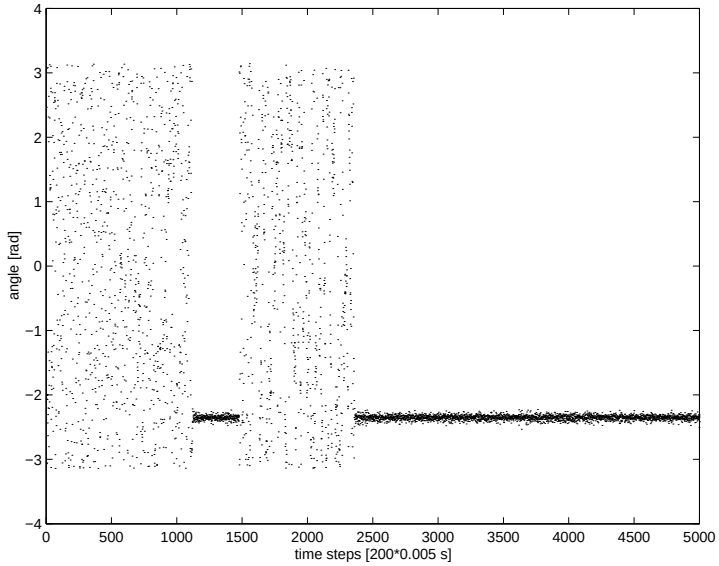

Figure 10: Modular Q-Learning in the full control task. One million steps were simulated, $\alpha = 0.001$.

Modular Q-Learning with Neural Networks The last architecture tested here consisted of two learning agents using Q-Learning with neural networks and the control structure just used. Input states were coded as in the modular Q-Learning design, thus yielding 80 possible states for each learning agent. The networks used were the same as used in the balancing task: Three networks - one for each action - having 80 input, five hidden and one output neuron.

After a first test the results seemed to be sufficient. After a relatively long period, the agents were able to reach a good minimum (Figure 11). However, further experiments often showed results as seen in the full control task with neural network based Q-Learning (Figure 9). Thus, this algorithm did not *necessarily* converge.

A closer look on its performance shows that the learning agent was able to solve the swinging up part of the task and as the experiments on the balancing task have shown learning the balancing component of the task

should not be a problem. Hence the question arises why is this modular learning architecture not able to generally solve the full control task? This is probably due to two reasons:

Learning to balance for an agent that uses neural network Q-Learning needs a rather long time (Figure 7). In this task, the main time of training is used for the upswing part. Even after successful swinging up, the pendulum soon returns to an angle below the horizontal line. Thus, this learning setting is not really suited to learn the balancing component of the task.

The more important reason lies in the physical properties of the simulation: After swinging up, the pendulum reaches rather high angular velocities. Given that the actions that might stop the pendulum on its way up are accelerations of the cart of $10\frac{m}{s^2}$ and $-10\frac{m}{s^2}$, in many cases it will be impossible to stop the pendulum in upright position. This problem could only be solved if either one single learning agent would learn the full control task (since the agent used for upswing in modular learning cannot know which angular velocities are to be preferred and thus will often use velocities that are to high) or if angular velocities would somehow be included into the reward function. The latter approach would need more expert knowledge on the relation of angular velocities, angles and the accelerations of the cart.

Generally the modular learning system tested here *can* solve the full control task but it does not necessarily do so. Here it would be advisable to experiment with further architectures. But probably a lot more knowledge of the dynamics of the physical system would be needed to create a perfect learning agent for this task.

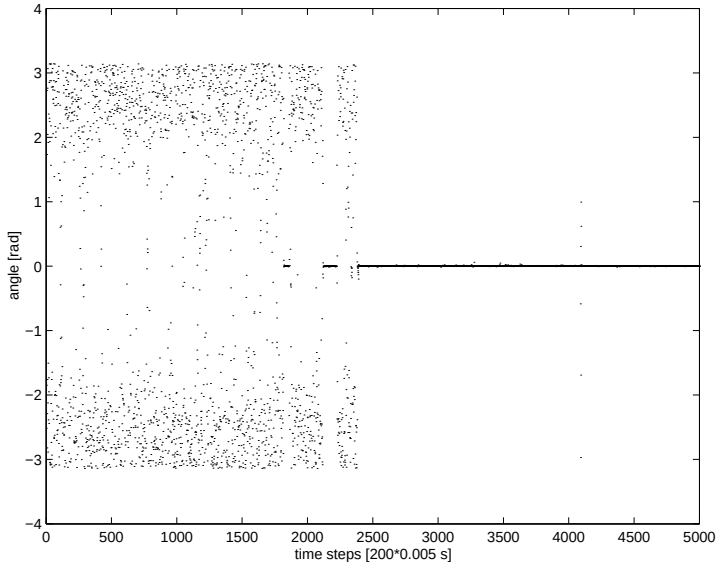

Figure 11: Modular Q-Learning with neural networks in the full control task. A good minimum could be reached. One million steps were simulated.

5 Summary

The results of the work on reinforcement learning done here can be summarized as follows:

There is a well understood theory of how to do learning by trial and error in control tasks. However, this theory uses a foundation that requires the special setting of environments having the Markov property and finite state and action spaces.

For many tasks it might be useful - or even necessary - to use the algorithms derived from these theoretical foundations and to modify them. Extending them with features such as function approximation or generalization may improve their functionality and enable the designer to use reinforcement learning in a much wider variety of environments and tasks.

The major drawback of this idea is that at the moment there are no *general* claims on how the different modified Temporal Difference Learning algorithms perform. The experiments shown here - as well as some shown in other publications - indicate that Q-Learning with neural networks and other approaches of that sort *may* yield far better results than regular Q-Learning but they do not always.

Reinforcement learning as learning by trial and error is intended to be done with minimal domain knowledge. As shown here it is quite useful in simple domains. However, for more complex problems coding of data and design of the reward function crucially determine the outcome of the learning process and crucially depend on the expert knowledge that was used to design them. If the goal is to solve a control task by using reinforcement learning techniques, it seems to be advisable not only to test different architectures using different algorithms and different forms of representing input and output spaces; but usage of domain knowledge seems to be extremely important to guarantee success.

Reinforcement Learning and Cognitive Science These results yield some interesting (though speculative) considerations for cognitive science.

From the information processing perspective, which is the perspective of classical cognitive science, reinforcement learning could be seen as a computational model of simple learning mechanisms, such as operant conditioning[11].

Just as more complex learning using reinforcement learning techniques seems to require a sophisticated framework and complex reward functions, this may also hold for biological organisms. In biological learning this framework may largely be established by anatomical and physiological constraints that were created by means of evolution. Thus, even from the purely func-

[11] Recall that operant conditioning in neurobiology denotes "learning the association between behavior and a reward"[Kandel et al., 2000, p.1242]

tionalist point of view, simple reinforcement learning can probably only be used to model simple learning processes. While using reinforcement learning techniques in connection with other approaches might provide interesting tools for modelling more complex biological learning. Again this only makes sense from a functionalist point of view. The theory of reinforcement learning probably is at least as far away from biological learning as artificial neural networks are from biological neural networks.

References

[Ballard, 1999] D. H. Ballard, *An Introduction to Natural Computation*, MIT Press, 1999

[Barto, 1995/1] A. G. Barto, *Reinforcement Learning*, in *The Handbook of Brain Theory and Neural Networks*, MIT Press, 1995

[Barto, 1995/2] A. G. Barto, *Reinforcement Learning in Motor Control*, in *The Handbook of Brain Theory and Neural Networks*, MIT Press, 1995

[Bergman and Schaefer, 1943] L. Bergmann and C. Schaefer, *Lehrbuch der Experimentalphysik*, Bd. 1, de Gruyter, 1943

[Hammer, 1999] B. Hammer, *Neuronale Netze: Vorlesung im WS 99/00*, http://www-lehre.inf.uos.de/ nn/nn.ps, 1999

[Kandel et al., 2000] E. R. Kandel, J. H. Schwartz and T. M. Jessel, *Principles of Neural Science*, McGraw-Hill, 2000

[Liebscher, 2000] R. Liebscher, *Beleg Neuronale Netze*, http://iweb1.informatik.htw-dresden.de/ iwe/Belege/Liebscher/, 2000

[Mitchell, 1997] T. M. Mitchell, *Machine Learning*, McGraw-Hill, 1997

[Ritter et al., 1990] H. Ritter, T. Martinez and K. Schulten, *Neuronale Netze: Eine Einfürung in die Neuroinformatik selbstorganisierter Netzwerke*, Addison-Wesley, 1990

[Russel and Norvig, 1995] S. Russel and P. Norvig, *Artificial Intelligence - A Modern Approach*, Prentice-Hall, 1995

[Sutton and Barto, 1998] R. S. Sutton and A. G. Barto, *Reinforcement Learning*, MIT Press, 1998

[Sutton, 1999] R. S. Sutton, *Reinforcement Learning*, in *The MIT Encyclopedia of Cognitive Science*, MIT Press, 1999

[Tesauro, 1995] G. Tesauro, *Temporal Difference Learning and TD-Gammon*, in *Communications of the ACM*, March 1995 / Vol. 38, No. 3, 1995

[Watkins, 1989] , C. J. C. H. Watkins, *Learning from Delayed Rewards*, Ph.D. Thesis Cambridge University, 1989

[Watkins and Dayan, 1992] C. J. C. H. Watkins and P. Dayan, *Q-Learning*, in *Machine Learning 8:279-292*, 1992

[Wengerek, 1996] T. Wengerek, *Reinforcement-Lernen in der Robotik*, Infix, 1996

[Zell, 2000] A. Zell, *Simulation Neuronaler Netze*, Oldenbourg Verlag, 2000