JOINT ADVANCED STUDENT SCHOOL 2008, ST. PETERSBURG

HIERARCHICAL REINFORCEMENT LEARNING

Final Report by

Dipl.-Ing. (BA) Susanne Schlötzer

born on 29.05.1982
address:
Itzgrund 46
95512 Neudrossenfeld
Tel.: 089 66560780

Institute of
AUTOMATIC CONTROL ENGINEERING
Technische Universität München
Univ.-Prof. Dr.-Ing../Univ. Tokio Martin Buss
Univ.-Prof. Dr.-Ing. Sandra Hirche
Univ.-Prof. Dr.-Ing.. Olaf Stursberg

Supervisor: Dipl.-Ing. Tina Paschedag
Submitted: 03.04.2008

Contents

1 Introduction

Since the late 1980s, extensive research has been performed in the domain of reinforcement learning. Reinforcement learning is an approach to machine intelligence and promises significant progress in complex control tasks that are so far unsolved. Modern approaches to trial-and-error learning have been founded during the last 20 years. Nevertheless dynamic programming, that forms a mathematical basis to optimal control theory, has already been invented by the US mathematician Richard Bellman in the 1950s. The main problem of dynamic programming is the computational effort connected to solving complex control problems and therefore restricts the number of applications. According to Sutton et al. [22], the computational effort even grows exponentially with the number of state variables. Respectively, it has shown that flat reinforcement learning cannot be transferred to real-life problems in many cases where reasonable learning speed is required. Therefore hierarchical approaches to reinforcement learning have been developed recently, promising a solution to complex machine learning systems.

This report is focused on a special approach to hierarchical reinforcement learning. The approach is referred to as task decomposition by sub-goals and has been proposed by Morimoto and Doya in 2001 [15]. A hierarchical architecture is used in order to learn a stand-up task of a three-link, two-joint robot. The stand-up task is subject to a high-dimensional state-space, e.g. flat architectures would induce high computational effort. In order to speed up the learning process, learning is performed in an upper and lower level. Sub-goals are learned in an abstract low-dimensional state-space in the upper level and local trajectories of the actual non-linear control law are learned in the lower level.

In section 2, the basics of a reinforcement learning problem are introduced. A detailed description of flat reinforcement learning can be found in Sutton et al. [22], Alpaydin [1] or Harmon et al. [7].

In section 3, commonalities of hierarchical approaches in reinforcement learning will be outlined and the notation of Semi-Markov Decision Processes is introduced.

In section 4, the previously mentioned hierarchical approach to learning of a stand-up task of a robot by Morimoto et al. [15] is presented. A detailed discussion of the algorithms and the advantages and disadvantages of the approach follows.

In section 5, further prominent approaches to hierarchical reinforcement learning are shortly presented. The options formalism (Sutton et al. [21]), Hierarchies of Abstract Machines HAMS (Parr et al. [16]), the MAXQ- (Dietterich [4]) and the HEXQ- (Hengst [8], Hengst [9]) framework are included in this section.

Finally, in section 6 the results are summarized and a prospect on future research in the domain of hierarchical reinforcement learning is given.

2 Basics of Reinforcement Learning

2.1 Learning Agent

In terms of reinforcement learning (RL), the learner is denoted as agent. The agent interacts with a (dynamic) environment. It receives a possibly delayed reward or penalty from the environment when taking an action that modifies the state of the environment. Thereby the agent learns by trial-and-error runs to achieve a specific task. This is in contrast to learning with a teacher, also denoted as supervised learning, where the teacher indicates the best action to be taken next in advance. In RL, there is a critic instead of a teacher, e.g. there is no information in advance which action to be taken next. A further difference is that there is only scarce feedback, often not before the final goal is achieved or failed. Delayed feedback complicates the agent's learning process and has brought forth new algorithms explicitly accounting for that delay. In Figure 2.1, the general structure of a RL problem is depicted:

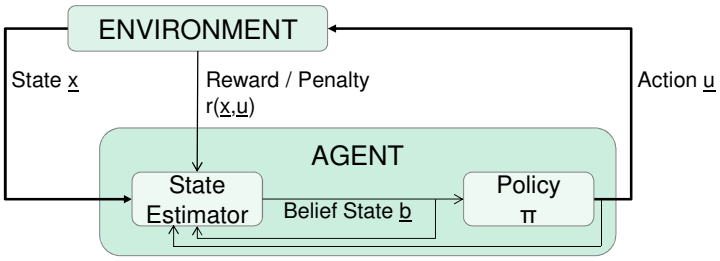


Figure 2.1: Interaction of the Agent with the Environment

In general, the agent has no prior knowledge of the model of the environment. The block diagram shows the case where the environment is only partially observable by the agent. This is the more general case. Thus a state estimator has to be implemented in addition which would not be required in case of a fully observable environment. Alpaydin [1] indicates that the belief state $\underline{b}$ is described by a probability distribution over states of the environment given the initial belief state and the past history of observations and actions of the agent. The agent has to keep an internal belief state $\underline{b}$ of the environment to keep the process Markov. In the further progress of this report, it is assumed that the environment is fully observable by the agent. The actual states $\underline{x}$ can serve directly as input to the policy π. The agent's policy π is a mapping from states of the environment to actions. The action u_t to be taken in each state x_t is defined by the policy π [1]:

$$u_t = \pi(x_t) \tag{2.1}$$

Whenever the agent takes an action u_t the state of the environment is changed and the agent receives a reward or penalty as return from the environment. The reward function $r(\underline{x},\underline{u})$ is used to evaluate whether an action has been good or bad. The single parts of the RL problem will be described in more detail in the following sections.

2.2 Markov Decision Process

Although RL may also be used for processes where interaction with the environment cannot be modelled as Markov Decision Process (MDP), most of modern RL theory refers to finite MDPs. In control tasks it is common to assume that the state and action spaces are finite. It is characteristic for a MDP that the probability of observing the next state x' and the reward r only depends on the last action u_t and the last observed state x_t. The mathematical formulation of the Markov property is given by the following equality of probability distributions [22]:

$$p\{x_{t+1} = x', r_{t+1} = r \mid x_t, u_t, x_{t-1}, u_{t-1}, \dots, x_0, u_0\} \stackrel{\text{def}}{=} p\{x_{t+1} = x', r_{t+1} = r \mid x_t, u_t\} \tag{2.2}$$

From equation (2.2) the term "one-step dynamics" of an environment that has Markov property becomes obvious. The property of one-step dynamics is frequently exploited by iterating RL algorithms. In Hutter [10] it is stated that there exist self-optimizing policies for ergodic MDPs. In this context ergodicity means that there exists a policy under which every state is visited infinitely often with probability 1. Because of the strong mathematical foundation of MDPs it is highly interesting if a RL problem can be modelled as MDP.

A simple example of a finite and deterministic MDP is shown in Figure 2.2.

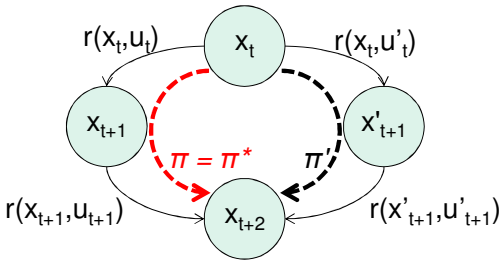


Figure 2.2: Finite and deterministic Markov Decision Process

Starting in state x_t, there are two options for the agent: taking action u_t or action u'_t. In this simple example, the reward and successor state are deterministic when taking any of both possible actions in the current state x_t. In the more general case state-transitions are non-deterministic and the next state is x_{t+1} with a probability of $P(x_{t+1}|x_t,u_t)$ when taking action u_t in state x_t. At the transition from one state to the next, the agent observes an immediate reward or penalty $r(x_t,u_t)$, once more only considering the deterministic case in this simple example. An episode is finished when the terminal state is reached after taking a sequence of actions or if a termination condition is reached. In state x_t there are two possible policies that can be followed to reach the terminal state x_{t+2}. Following the optimal policy π^* provides a greater expected return than following the policy π'. In RL,

the agent learns to maximize the cumulative expected reward and therefore will learn to follow the policy π^* after a sufficient training period. In this example the optimal policy π^* maps the state x_t to the action u_t while the policy π' maps the state x_t to the action u'_t.

2.3 Reward Function

The definition of appropriate reward functions is essential to ensure that the learning agent will solve the problem efficiently.

Basic concepts of defining reward functions depending on the current task are listed in Harmon et al. [7]. The following classes of reward functions are frequently used in RL:

1. Pure delayed reward class of functions
2. Minimum time to goal class of functions
3. Optimal behaviour class of functions

The pure delayed reward class of functions can be characterized as follows: There is only non-zero reinforcement at the terminal state. The agent receives a reward (positive sign of the scalar reinforcement) if the system reaches the desired terminal state. If the system reaches a terminal state which should be avoided the agent receives a penalty (negative sign of the scalar reinforcement function).

In the minimum time to goal class of functions the reward function is set to -1 for all state transitions except for the transition to the terminal state where the reinforcement is zero. Therefore the agent learns to find the shortest trajectory to the terminal state.

The third class of reward functions listed here aims at generating an optimal behaviour in complex scenarios. The reward function does not necessarily have to be maximized, but, depending on the concrete problem to solve, has to be minimized or a saddle-point, a maxmin or a minmax of the reward function has to be found.

A combination of different classes of reward function is also possible. Special care is required to define appropriate reward functions in the context of hierarchical RL. According to Barto et al. [2], RL engineers can introduce pseudo-reward functions, or auxiliary reward function, to specify sub-tasks that have to be learned by an agent in hierarchical frameworks without having to specify the policy itself to solve the sub-tasks. By seeking to maximize these additional rewards the agent will learn to achieve the sub-goals.

2.4 Function Estimation

2.4.1 State-Value Function

The essence of RL is to derive the optimal value function for a specific problem efficiently. The state-value function $V(\underline{x}_t)$ maps states to state-values. In RL, a state-value is defined as the sum of expected and possibly discounted future rewards the actor receives from the critic when starting in that state and terminating when the final state is

reached. Since the expected future rewards when the agent is in a given state $\underline{x}_t$ depend on the course of actions the agent will take, the value of a state is always linked to a policy π that is obeyed by the agent [22]:

$$V^{\pi}(\underline{x}_t) = E_{\pi}\left[\sum_{i=1}^{\infty} \gamma^{i-1} \cdot r_{t+i}\right] \tag{2.3}$$

, where

$V^{\pi}(\underline{x}_t)$: is the value of the state $\underline{x}_t$ when following the policy π thereafter,

$E_{\pi}[\ldots]$: is the expected value given that the agent follows the policy π,

γ: is the discount rate, where $0 \leq \gamma < 1$,

r_t: is the reward function.

Equation (2.3) refers to an infinite-horizon model. In an episodic or finite horizon model, the sequence of actions the agent can take is limited. Assuming that the sequence limit of the finite-horizon model is reached after N steps, equation (2.3) can be truncated after N steps. The discount rate γ serves two purposes: For infinite-horizon models the introduction of the discount rate ensures that the sum of expected future rewards keeps finite. This is important since the time period for solving a task is only limited if the state-value function is bounded. The second purpose of the discount rate is to give different weight to rewards that are nearer or further in the future. By setting the discount rate to zero, only immediate rewards count and rewards further in the future cancel out. On the other hand, the closer the discount rate is to 1, the greater is the contribution of rewards in the far future to the current state-value and the agent becomes more far-sighted.

RL aims at approximating the optimal value function $V^*(\underline{x}_t)$, starting from a poor initial estimate $V(\underline{x}_t)$:

$$V^*(\underline{x}_t) = \max_{\pi} V^{\pi}(\underline{x}_t), \quad \forall \underline{x}_t. \tag{2.4}$$

Although there may be more than one optimal policy that maximizes the value of each state, there is only one optimal state-value function $V^*(\underline{x}_t)$ they have all in common.

2.4.2 Action-Value Function

In the context of control, Q-functions frequently replace state-value functions. Q-functions are also denoted as action-value functions since they perform a mapping from state-action pairs to values instead of a mapping from states to state-values. In accordance to equation (2.3), the action-value-function of an infinite-horizon model can be formulated as:

$$Q^{\pi}(\underline{x}_t, \underline{u}_t) = E_{\pi}\left[\sum_{i=1}^{\infty} \gamma^{i-1} \cdot r_{t+i}\right] \tag{2.5}$$

, where $Q^{\pi}(\underline{x}_t, \underline{u}_t)$ is the action-value function when the agent takes an action $\underline{u}_t$ in state $\underline{x}_t$ and follows the policy π thereafter. Thus, in contrast to the state-value function one is rather interested in how good it is for the agent to take an action $\underline{u}_t$ when in state $\underline{x}_t$ than

how good it is for the agent to be merely in state $\underline{x}_t$. Both the state-value function $V^{\pi}(\underline{x}_t)$ and the action-value function $Q^{\pi}(\underline{x}_t,\underline{u}_t)$ can be represented as parameterized functions whose parameters have to be adjusted during learning. It is referred to Sutton et al. [22] to get a comprehensive introduction to function approximation of the state-value and action-value function. According to Dietterich [4] it is far more efficient to use function approximation techniques instead of tabular representations of the value functions.

During the learning process, the optimal action-value function $Q^*(\underline{x}_t,\underline{u}_t)$ shall be approximated and in analogy to equation (2.4), the optimal state-value function is defined as:

$$Q^*(\underline{x}_t,\underline{u}_t) = \max_{\pi} Q^{\pi}(\underline{x}_t,\underline{u}_t)\ , \quad \forall \underline{x}_t, \forall \underline{u}_t, \tag{2.6}$$

Finally, the relation between the optimal state-value function and the optimal Q-function results in the Bellman optimality equation [1]

$$\begin{aligned} V^*(\underline{x}_t) &= \max_{\underline{u}_t} Q^*(\underline{x}_t,\underline{u}_t) \\ &= \max_{\underline{u}_t} E_{\pi^*}\left[\sum_{i=1}^{\infty} \gamma^{i-1}\cdot r_{t+i}\right] \\ &= \max_{\underline{u}_t} E\left[r_{t+1} + \gamma \cdot V^*(\underline{x}_{t+1})\right] \end{aligned} \tag{2.7}$$

An N-state MDP results in a system of N equations, e.g. for each state one equation, with N unknowns. Although a great variety of methods exist for solving those equation systems, so far the computational effort for solving RL problems that have to be formulated in a high-dimensional state-space is too high to be of practical relevance. Therefore, recently hierarchical architectures have been developed in RL that can deal better with the dimensionality of complex problems than flat architectures.

3 Hierarchical Architectures

3.1 Survey of Hierarchical Concepts

The main reason for introducing hierarchical architectures in RL is to bridge the gap between theoretical considerations to machine intelligence and practical application to real-world problems. Hierarchical RL is especially interesting for large-scale problems where the performance of flat RL is too poor with regard to learning speed and computational effort. Complex machine-learning systems are described in high-dimensional state-spaces. However, RL methods scale badly with the size of state-spaces. As mentioned before, computational effort even grows exponentially with the number of state variables [22] and convergence to the optimal policy takes prohibitively long time under practical considerations. Note that this statement refers to solution methods that are based on dynamic programming invented by Bellman. Flat RL methods also scale badly with the size of action-spaces and the length of the sequence of actions to reach the terminal state. Hierarchical RL is used to accelerate the learning of new problems in case of online-learning, for example online-planning and online-adaption to the environment, where the performance of the agent during the learning period matters. Besides faster learning, an economic aspect in favour of some hierarchical RL methods is the possibility to reuse learned policies in the lower levels of the hierarchy [4]. Once an optimal policy is derived for a sub-task, the hierarchical architecture should allow transferring the learned policy to different parent tasks.

In Miyamoto et al. [13], multiple time scales, macro-actions and sub-goals are listed to construct higher levels, accounting for multiple temporal scales and/or multiple levels and/or multiple spatial scales of large tasks to be learned.

A complex problem, like driving a car, can be divided into multiple time scales where several primitive actions are executed before terminating a sub-task. Temporally extended sub-problems may be learned more efficiently by introducing a hierarchical structure if there is at least near independence of some state variables from others. Irrelevant aspects of the state-space can be ignored when solving a concrete sub-task.

Hierarchical approaches that apply macro-actions introduce abstraction by comprising a sequence of actions in one macro that can be invoked as if it were a primitive action. Multiple macros can be interleaved, forming several levels of hierarchies. According to [2], the main drawback of macros is that they can define only open-loop control policies. A generalization of the idea of macros leads to the options-formalism presented in section 5.1, which can be used to implement closed-loop partial policies.

The decomposition of a task by sub-goals involves that high-level policies identify sub-goals that follow the achievement of the overall goal. In principal, design engineers

might introduce some predisposed knowledge into higher levels, e.g. by defining fixed sub-goals instead of learning them, and thereby facilitate the agent's learning for concrete tasks. However, the more general approach is that the agent also learns the definition of sub-goals in the upper level. In the lower levels of the hierarchy sub-policies are learned to reach the sub-goals defined in an upper level. Quite often separate specialized learning agents are used for different sub-tasks. The hierarchical approach to acquire stand-up behaviour for a robot presented in section 4 is based on this scheme of decomposing a task by sub-goals.

3.2 Semi-Markov Decision Process

Semi-Markov decision processes (SMDPs) serve as theoretical basis for many hierarchical RL approaches developed during the last decay. In these hierarchical approaches temporally-extended and abstract actions need to be modelled. SMDPs may be considered as generalization of MDPs. Sutton et al. [22] define a SMDP as a continuous-time decision problem that is treated as a discrete-time system, where the system makes discrete jumps from one time at which it has to make a decision to the next. In contrast to MDPs, which were presented in section 2.2, SMDPs also account for the amount of time that passes between sequential decision stages. Therefore a real-valued variable for continuous-time SMDPs or an integer-valued variable for discrete-time SMDPs is introduced that holds the stochastically-distributed amount of time that passes between two decisions. The probability to make a transition from state x_t to state x_{t+1} has to be extended by a random time variable τ. According to Barto et al. [2], the resulting joint probability $P(x_{t+1},\tau|x_t,u_t)$ defines the probability that a transition from state x_t to state x_{t+1} occurs after τ time steps when action u_t is executed.

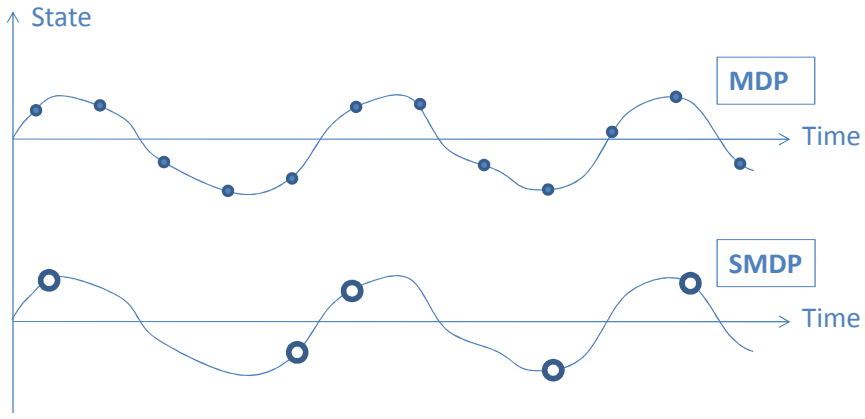


Figure 3.1: Comparison of a Markov Decision Process and a Semi-Markov Decision Process

In the top of Figure 3.1 a discrete-time MDP is shown and in the bottom of Figure 3.1 a continuous-time, discrete-event SMDP is shown. If the system is regarded from an upper level of the hierarchy, it remains in each state for a random waiting time. An instantaneous transition to the next state occurs after the waiting time has passed by. The lowest level of the hierarchy can be understood as temporally-extended action.

4 Task Decomposition by Sub-goals

4.1 Stand-up Task of a Robot

A hierarchical approach, that has been proposed by Morimoto and Doya in 2001 [15], to achieve stand-up behaviour of a robot is presented and analyzed in this and the following sub-sections. The learning of the complex task, which is originally formulated in a high-dimensional state-space, is achieved by decomposition of the task by sub-goals. Figure 4.1 shows a two-joint, three-link robot that shall acquire stand-up behaviour using hierarchical RL:

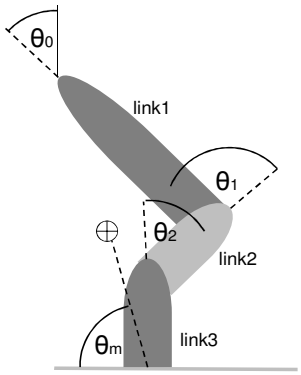


Figure 4.1: Two-joint, three-link robot

The angle θ_0 denotes the pitch angle, θ_1 the hip joint angle, θ_2 the knee joint angle and θ_m is the angle of line from the centre of mass of the robot to the centre of the robot foot. Note that the robot foot is not connected with the ground which enhances the system's dynamics significantly. There exists no static solution to the control problem. In order to extract non-linear control laws for the two servo-motors located at the robot joints, a RL algorithm has to be found that can deal with the high-dimensional state-space and provides a reasonable learning speed to allow for online-adaptation to changes in the environment. The hierarchical approach proposed by Morimoto et al. reduces the dimensions of the state-space in the upper level to speed up the learning of a sequence of sub-goals. A complete sequence of learned sub-goals with which the stand-up task could be achieved is depicted in the top of Figure 4.2. The state variable vector chosen for the upper level is $\underline{X}(T) = (\theta_m, \theta_1, \theta_2)$. It can be verified with the help of simulations that the choice of the state variables in the upper level has been appropriate to achieve the overall goal. A sub-goal $\underline{U}_T$ defined by the agent in the upper level and that shall be learned by an agent in the lower level is composed of the angles θ_0, θ_1 and θ_2. In the lower level, several agents optimized for learning of different local trajectories are used. In contrast to learning in the upper level, the learning of concrete sub-goals has to be

performed in a high-dimensional state-space. The physical state variable vector that is used in the lower level is given by $\underline{x}(t) = (\theta_0, \theta_1, \theta_2, \dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2)$. Local stand-up trajectories learned by agents in the lower level are depicted in the bottom of Figure 4.2. Morimoto et al. indicate that approximately 750 trials are required in simulations to learn the stand-up task.

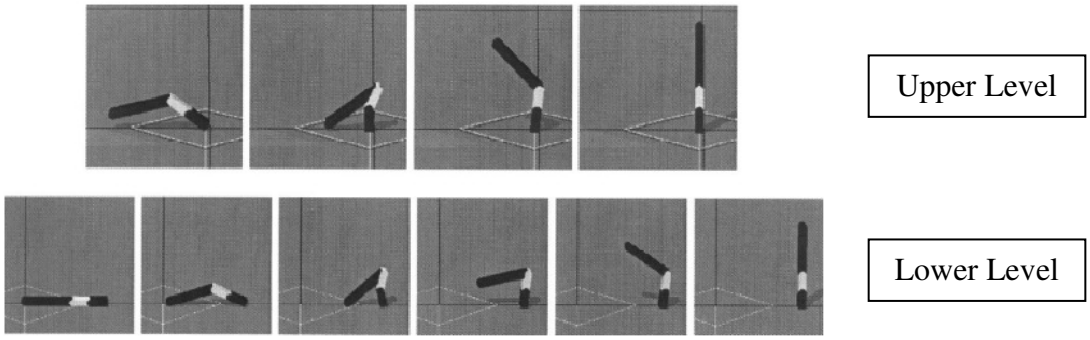


Figure 4.2: Episode of a successful Stand-up Trial (taken from Morimoto et al. [15])

The general scheme of the hierarchical RL approach is outlined in Figure 4.3. Sequences of sub-goals have to be explored globally in an abstract state-space in order to prevent convergence to local optima. The notation abstract state-space here also implies that the state-space is low-dimensional by coarsely discretizing it and thereby fastening the global search for sub-goals. Depending on the definition of the concrete sub-goal in the upper level, it is switched between different learning modules in the lower level. In contrast to the abstract state-space in the upper level, the physical state-space in the lower level is not discretized. Thus, the agents' exploration in the lower level has to be performed in a high-dimensional state-space. However, as the sub-goal to be learned has already been defined in a sense of global optimality in the upper level, it is sufficient if the agents in the lower level restrict their exploration to local areas in the high-dimensional state-space. The output of the lower level learning modules are the desired control signals derived from a non-linear feedback control law.

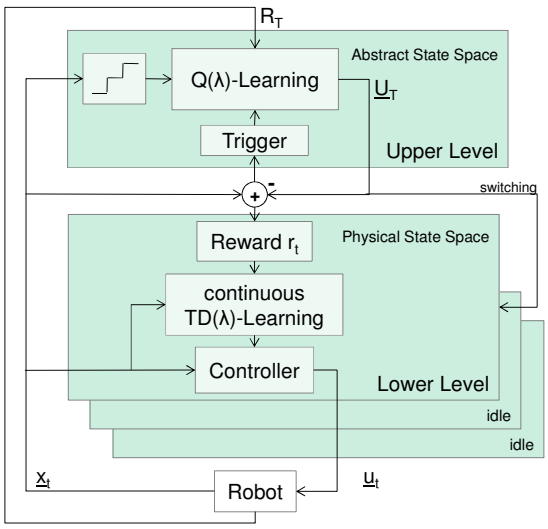


Figure 4.3: Hierarchical architecture to solve the non-linear control problem

By introducing a hierarchical architecture as shown in Figure 4.3, the non-linear control problem in an originally high-dimensional state-space is simplified. While the overall problem to be learned by the upper level agent in the abstract state-space is still non-linear, the local trajectories of the control law to be learned by the lower level agents in the physical state-space are nearly-linear.

There exists a great variety of dynamic programming methods to approximate the optimal value function. As shown in Figure 4.3, for the given example of a stand-up task, Morimoto et al. have decided to use Q(λ)-learning in the upper level and a continuous version of TD(λ)-learning with the actor-critic method in the lower level. In order to ensure fast convergence to an optimal policy, appropriate reward functions R_T and r_t have to be defined in the upper and lower level. The next three sections are dedicated to the definition of reward functions and the description of both RL algorithms as well as the motivation for using these methods.

4.2 Reward Functions in the Upper and Lower Level

The definition of appropriate reward functions in the upper and lower level is essential to achieve good performance in learning the stand-up task. The number of episodes to learn the task shall be as small as possible. It becomes obvious from equation (4.1) to (4.5) that the definition of appropriate reward functions is highly task-specific and experimentation-sensitive.

The reward function in the upper level R_T is composed of a main reward R_{main} and a supplementary reward R_{sub} [15]:

$$R_T = R_{main} + R_{sub} \tag{4.1}$$

$$R_{main} = \begin{cases} 1 & , on\ success\ of\ stand-up \\ 0 & , on\ failure \end{cases} \tag{4.2}$$

$$R_{sub} = \begin{cases} 1 & , final\ goal\ achieved \\ 0.25 \cdot \left(\frac{Y}{L} + 1\right) & , sub-goal\ achieved \\ 0 & , on\ failure \end{cases} \tag{4.3}$$

, where the fraction of Y over L indicates the normalized height of the robot head (L) with regard to the total length of the robot (L). The introduction of R_{sub} is essential to prevent that there is only delayed reward in the upper level when the whole stand-up task is achieved. The agent in the upper level has to define sub-goals that are achievable by the agents in the lower level within a reasonable learning period. The supplementary reward encourages the agent in the upper level to define sub-goals which the lower level agents can already achieve in an early stage of learning.

Similar effort is required when defining the reward functions in the lower level [15]:

$$r\left(\underline{\theta}, \underline{\tilde{\theta}}\right) = E\left[-\frac{\left\|\underline{\theta} - \underline{\tilde{\theta}}\right\|^2}{s_\theta^2}\right] - 1 \quad , during\ control \tag{4.4}$$

$$r(t) = \begin{cases} E\left[-\frac{\left\|\underline{\dot{\theta}} - \underline{\tilde{\dot{\theta}}}\right\|^2}{s_{\dot{\theta}}^2}\right] & , sub-goal\ achieved \\ -1.5 & , fall\ down \end{cases} \quad (4.5)$$

With:

$r\left(\underline{\theta}, \underline{\tilde{\theta}}\right)$ is the reward function during control,

$r(t)$ is the additional reward function at the end of control,

$\underline{\theta} = (\theta_0, \theta_1, \theta_2)$: is the vector of observed angles,

$\underline{\tilde{\theta}} = \left(\tilde{\theta}_0, \tilde{\theta}_1, \tilde{\theta}_2\right)$: is the vector of desired angles according to the sub-goal U that is pre-defined by the agent in the upper level,

$\underline{\dot{\theta}} = \left(\dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2\right)$ is the vector of observed angular velocities,

$\underline{\tilde{\dot{\theta}}} = \left(\tilde{\dot{\theta}}_0, \tilde{\dot{\theta}}_1, \tilde{\dot{\theta}}_2\right)$: is the vector of desired angular velocities that are set by the memory of successful trials,

s_θ is a normalization factor in units of degree to set the width of the reward function $r\left(\underline{\theta}, \underline{\tilde{\theta}}\right)$ and

$s_{\dot{\theta}}$ is a normalization factor in units of degree per second to set the width of the reward function r(t).

According to equation (4.4), agents in the lower level receive a penalty during control if there is a deviation between the observed angles and the angles specified by the agent in the upper level. As agents try to maximize the expected cumulative reward, a local trajectory that is straight-forward and requires minimum control effort will be learned. The additional reward r(t) given at the end of the control period in the lower level does also account for the system dynamics, considering that there is no static solution to the stand-up task. It is likely that many simulation runs were required to derive and tune reward functions that provide a reasonable learning speed.

4.3 Learning in the Upper Level

4.3.1 Choice of Actions in the Upper Level

In this approach to learning of sub-goals in the upper level, the policy used to select actions is not the same as the policy learned by Q-learning to define sub-goals. Sutton et al. [22] define methods like Q-learning as off-policy methods, e.g. the policy used to select actions may differ from the policy learned by the learning algorithm. Actions in the upper level are chosen following an exploration strategy to ensure convergence to the global optimum, but the selection of actions might be sub-optimal with regard to the current action-value-function. In this section the choice of actions will be presented before Q(λ)-learning will be introduced in the next section.

An action $\underline{u}_t$ is chosen at time t with a probability distributed according to a Boltzmann distribution:

$$P(\underline{u}_t|\underline{x}_t) = \frac{E[\beta \cdot Q(\underline{x}_t, \underline{u}_t)]}{\sum_{\underline{b}\in A(\underline{x})} E[\beta \cdot Q(\underline{x}_t, \underline{b})]} \tag{4.6}$$

, where

$P(\underline{u}_t|\underline{x}_t)$: is the probability of choosing action $\underline{u}_t$ in state $\underline{x}_t$,

β: is the exploration factor, or inverse temperature,

$A(\underline{x})$: is the set of possible actions at state $\underline{x}$.

Equation (4.6) is also called softmax function [1] as it ensures a soft policy with non-zero probability of choosing any action $\underline{u}\in A$ in state $\underline{x}\in X$. In general, there is no perfect knowledge of the environment concerning RL problems. At the beginning of learning, the whole state-space has to be explored to query a model of the environment. After sufficient exploration of the action-space, e.g. one is already close to the global optimum, one can start with exploitation. Exploitation in this context means that presumably the best actions are taken more frequently than rather selecting actions randomly. An increase of the exploration factor β in time is used to perform the transition from exploration to exploitation. The choice of actions is almost uniformly randomly distributed for small values of β. By increasing β the value of the Q-function becomes more and more important when choosing an action and thus one already exploits instead of solely explores after a certain learning period. A disadvantage of the Boltzmann exploration scheme presented here is that it needs to calculate all discrete action-values. Computational costs increase exponentially with the number of action-variables [11], prohibiting to use this scheme for large action-spaces. Nevertheless, this scheme of selecting actions in the upper level is justified for the hierarchical RL approach considered here since the state-space in the upper level is coarsely discretized.

4.3.2 Q(λ)-Learning

Morimoto et al. use Peng's Q(λ)-learning method [17] in order to learn sub-goals in the upper level. Q(λ)-learning is a model-free RL algorithm that makes the application to real-life problems where a concrete model of the environment is generally not known in advance often more easy. In [2] it is mentioned that it is beneficiary to use Q(λ)-learning, which estimates action-values, for discrete-event systems since it does not require a one-step ahead search to determine optimal actions. Thus, the Q(λ)-learning algorithm is particularly suited for the learning of sub-goals in the upper level. The Q(λ)-learning algorithm combines one-step Q-learning, e.g. Q(0)-learning, with the return of temporal-differences for general λ in an incremental way. The decay parameter λ will be explained later in this section in the context of eligibility traces. Since Q-learning is one of the most frequently used algorithms in RL and also many hierarchical RL algorithms are based on slightly adapted versions of Q-learning, the basic formulation of Q-learning will be presented here.

One-step Q-learning:

The standard approach of one-step Q-learning is obtained by setting λ to zero. The update rule for Q(0)-learning is:

$$Q(\underline{x},\underline{u}) \leftarrow Q(\underline{x},\underline{u}) + \alpha \cdot \left[r + \gamma \cdot \max_{\underline{u}'} Q(\underline{x}',\underline{u}') - Q(\underline{x},\underline{u}) \right] \qquad (4.7)$$

, where α is the learning rate, $\underline{x}'$ is the next state and $\underline{u}'$ is the next action. One-step Q-learning converges to the optimal action-value function Q* if the learning rate α is gradually decreased in time [1]. However, if the agent shall keep on learning infinitely and the environment changes during the considered control period, it might be preferable to set the learning rate to a constant factor as Morimoto et al. have done for learning the stand-up task. Q-learning belongs to the category of off-policy temporal-difference (TD) control algorithms [22], where the convergence to Q* is independent of the policy being followed. The difference between the current action-value Q(x,u) and the sampled backup of the next step is reduced during several episodes. The procedure of one-step Q-learning will be demonstrated with the help of a simple example. The underlying deterministic MDP is shown in Figure 4.4. The immediate reward *r* for taken an action *u* in state *x* is indicated next to the arrow of transition between two states.

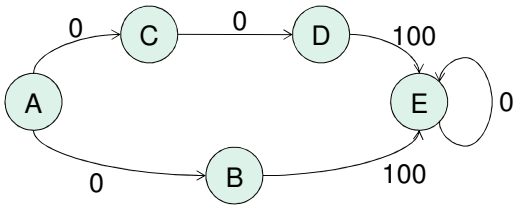
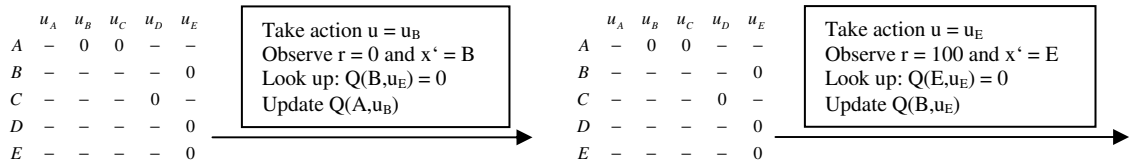
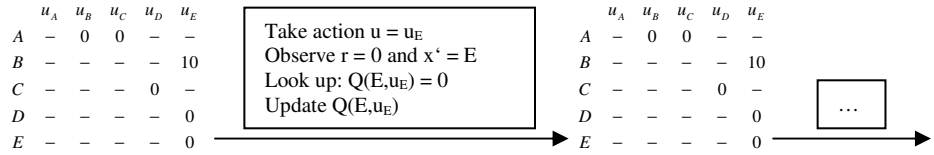


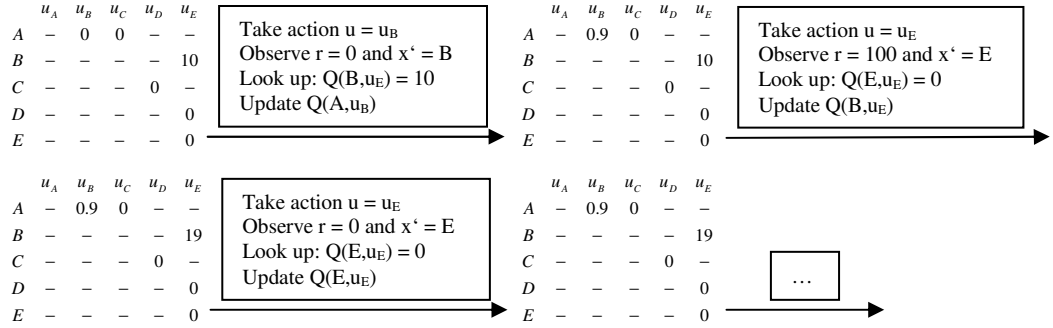
Figure 4.4: Simple MDP with Reward Assignment

The RL problem can be described as follows: The initial state of the environment is given by state A. An agent shall learn the best policy to reach the goal state E. Non-zero reward is only given at the transition from state B to E or at the transition from state D to E. Starting in state A, the agent can choose arbitrarily between two actions – one that causes a transition to state B while the other causes a transition to state C. In order to illustrate the update of Q-values during learning, a Q-table is introduced. In the following numerical example, the learning rate α is set to 0.1 and the discount rate γ is set to 0.9. The Q-values of the table are initialized arbitrarily, for example by zero. Following the current exploration strategy, assume that the agent chooses to take an action that causes the transition from state A to state B. The update of Q-values for the current episode starting from state A to the terminal state E is given by:





Now the next episode is considered, starting again in state A. By chance, the exploration strategy assesses once more to take action u_B in state A.



As long as an exploration strategy is followed instead of an exploitation strategy in order to choose actions, there will also be episodes that visit the state sequence A-C-D-E instead of A-B-E. After a sufficient period of learning, the state-action values of the Q-table will converge to the Q*-values:

	u_A	u_B	u_C	u_D	u_E
A	–	90	81	–	–
B	–	–	–	–	100
C	–	–	–	90	–
D	–	–	–	–	100
E	–	–	–	–	0

Later, the agent will follow an optimal policy π*. The agent, following the optimal policy π*(A), will choose action u_B instead of action u_c when being in the initial state A since $Q^*(A,u_B) > Q^*(A,u_C)$. This simple example also illustrates the influence of the discount rate γ. By setting γ to zero, only immediate rewards count to the Q-value to be updated and here it will become indifferent for the agent whether to choose action u_B or action u_C when being in state A. By choosing γ different from zero, expected future rewards will contribute to the current Q-value. The closer γ is to 1, the less important it is how far in the future the expected rewards are. If γ equals 1, the action-values $Q(A,u_B)$ and $Q(A,u_C)$ will converge to the same Q*-value, e.g. 100.

The pseudo-code of the Q(λ)-algorithm actually implemented in the upper level, that is more complex than the simple one-step Q-learning algorithm presented here, can be found in the appendix of this report. In contrast to one-step Q-learning, Peng et al. [17] indicate that Q(λ)-learning is experimentation-sensitive if $\lambda \neq 0$. On the other hand Q(λ)-learning is less experimentation-sensitive than the actor-critic method used for learning in the lower level that will be presented in section 4.4.2. Unlike for one-step Q-learning, convergence to the optimal action-value function Q* is not guaranteed for Q(λ)-learning if an action selection policy is chosen that selects actions arbitrarily in any state.

Eligibility traces:

Q(λ)-learning is a special form of temporal-difference methods that can be combined with eligibility traces. The handling of delayed reward is accomplished by introducing eligibility traces into RL algorithms. Therefore, the combination with eligibility traces frequently results in a more efficient way of learning by propagating information rapidly. The decay parameter λ, $0 \leq \lambda \leq 1$, is a measure for the decay of the trace over time. Peng's Q(λ)-algorithm makes use of accumulating activity traces. An activity trace refers to a trace for a state-action pair ($\underline{x}$,$\underline{u}$) and thus slightly differs from standard eligibility traces which just refer to traces for states. The following considerations are designated to eligibility traces, but can immediately be transferred to activity traces. The accumulating eligibility trace is defined as [20]:

$$e_{t+1}(\underline{x}) = \begin{cases} \gamma \cdot \lambda \cdot e_t(\underline{x}) & if\ \underline{x} \neq \underline{x}_t \\ \gamma \cdot \lambda \cdot e_t(\underline{x}) + 1 & if\ \underline{x} = \underline{x}_t \end{cases} \tag{4.8}$$

, where $e_t(\underline{x})$ is the trace for state $\underline{x}$ at time t. The decay of the trace depends both on the decay parameter λ and the discount rate γ. A memory variable, that reflects the temporary record of the occurrence of an event, is introduced for each state. The value of the memory variable is incremented every time the state is visited and decays gradually over time. The exponential decay of an eligibility trace over time is shown in Figure 4.5. Sutton et al. [22] indicate that the credit assigned to prior events depends on how recently and how frequently an event has occurred. Therefore eligibility traces bridge the gap between events and training information.

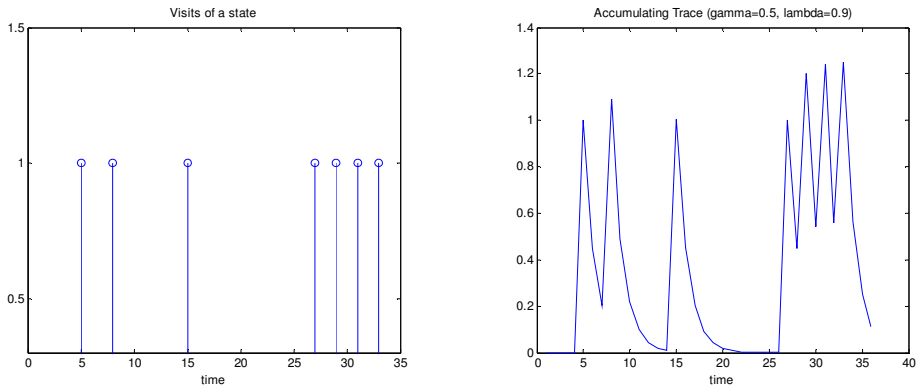


Figure 4.5: Accumulating Eligibility Trace

Concerning control problems, a separate short-time memory, or activity trace, has to be implemented for each state-action pair. In order to fasten the learning of sub-goals in the upper level, Morimoto et al. have chosen a decay rate that is close to 1. However, Singh et al. [20] state that the use of replacing eligibility traces instead of accumulating eligibility traces could make learning faster, more reliable and less parameter-sensitive. Their investigations refer to the family of TD(λ)-algorithms in general and not to Peng's Q(λ)-learning explicitly.

4.4 Learning in the Lower Level

4.4.1 Choice of Actions in the Lower Level

The choice of actions in the upper level follows a Boltzmann distribution. In the lower level, a continuous action-space has to be explored. Kimura [11] states that the computational costs for the Boltzmann exploration as presented in section 4.3.1 increase exponentially with the number of action variables. Therefore a different exploration strategy is required than the one used in the upper level. According to Kretchmar et al. [12] radial basis functions (RBFs) are local function approximators that are particularly suited to represent gradual-continuous transitions in the functions to be approximated. The usage of normalized radial basis functions (NRBFs) in RL is better suited than the usage of RBFs since they do not drop-off as fast at the edge of the state-space as RBFs. The general form of a Gaussian-shaped RBF is given by [12]:

$$RBF(\underline{x}) = w \cdot e^{-\frac{\|\underline{c}-\underline{x}\|^2}{\sigma^2}} \tag{4.9}$$

where,

w: is a weight factor assessing the maximum value of the RBF,

σ: is a width factor of the RBF and

$\underline{c}$: is the centre vector of the basis function.

Correspondingly, the NRBF can be expressed as:

$$NRBF_j(\underline{x}) = w_j \cdot \frac{e^{-\frac{\|\underline{c}_j-\underline{x}\|^2}{\sigma_j^2}}}{\sum_i \left[e^{-\frac{\|\underline{c}_i-\underline{x}\|^2}{\sigma_i^2}} \right]} \tag{4.10}$$

The benefit of using NRBFs instead of flat action selection schemes like the Boltzmann exploration used for the upper level is that actions can be selected in high-dimensional continuous action-spaces at reasonable computational effort. To conclude, at least three reasons can be listed in favour of using NRBFs in the lower level of the hierarchical RL approach:

1. NRBFs are particularly suited to approximate continuous functions like local trajectories. Any quantization introduced in the lower level would result in a worse control law than the one obtained with the learned continuous control function.
2. NRBFs are particularly suited to approximate non-linear functions. Although only a local part of the non-linear control function is considered in the lower level, there still remains non-linearity in the local trajectory at some degree.
3. A drawback of NRBFs, e.g. they tend to converge to local optima, is overcome as the global search has already been performed in the upper level of the

hierarchy. Thus, a presumably optimal sub-goal is already given to the agent in the lower level, ensuring that the convergence to the next optimum will solve the sub-problem.

4.4.2 Continuous TD(λ)-Learning with the Actor-Critic Method

Q(λ)-learning is not well-suited for complex control tasks in high-dimensional continuous action-spaces. As stated before, a continuous formulation of the learning algorithms is preferable to prevent approximation errors that occur when discretizing the control problem. Therefore, better performance is expected from continuous TD(λ)-learning with the actor-critic method to learn local trajectories of the control law in the high-dimensional state-space. In Figure 4.6 the actor-critic method is illustrated where the critic learns the state-value function $V(\underline{x}_t)$ and the actor learns the control command sequence $\underline{f}(\underline{x}_t)$.

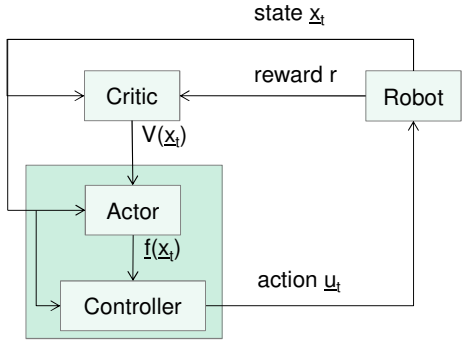


Figure 4.6: Schematic of the Actor-Critic Method

There is a separate memory to represent the policy learned by the actor independent of the value function learned by the critic. A temporal-difference error is generated by the critic that influences the learning of the actor. The optimal value-function $V^*(\underline{x})$ and the nonlinear feedback control law $\underline{f}(\underline{x}_t)$ are approximated with NRBFs:

$$\dot{\underline{w}} = \alpha \cdot \delta_t \cdot \underline{g}_t \tag{4.11}$$

, with the parameter vector $\underline{w}$ of NRBFs, the learning rate α and the continuous equivalent to Bellman's residual, e.g. the Hamiltonian δ_t:

$$\delta_t = r_t - \frac{1}{\tau} \cdot V(\underline{x}_t) + \frac{dV(\underline{x}_t)}{dt} \tag{4.12}$$

, where τ is a time factor. In case of an update of the critic, the vector $\underline{g}_t$ stands for the eligibility traces of the NRBFs. In case of an update of the actor, the vector $\underline{g}_t$ stands for NRBFs weighted with a noise term for exploration. A positive Hamiltonian ($\delta_t>0$) causes an increase in past values of $\dot{\underline{w}}$; a negative Hamiltonian ($\delta_t<0$) causes a decrease in past values of $\dot{\underline{w}}$ respectively. Coulom [3] indicates that the gradient ascent of the Hamiltonian can be used to extend the parametric optimization of policies to infinite horizon and stochastic optimal control problems in case of the actor-critic method. Regarding critic-only methods, the optimization is restricted to finite horizon and deterministic RL problems.

The following benefits of applying an actor-critic method in RL can be listed according to Sutton et al. [22]: The computational costs for selecting actions are minimal. This is especially important in case of a continuous action-space in the lower level, where learning with the Q(λ)-method would become infeasible in practise. The policy, e.g. the nonlinear feedback control law, is stored separately from the value function and an extensive computation of state-action values is not required for each action selection. A further advantage of actor-critic methods is that application-specific constraints on the set of allowed policies can be imposed more easily by separating the actor from the critic.

4.5 Evaluation

Morimoto et al. [15] indicate that the learning speed and the success rate of solving the stand-up task of a two-joint, three-link robot can be increased significantly by introducing the hierarchical RL approach presented in section 4. The efficiency of the task decomposition by sub-goals has been proofed by a series of simulations. In the next step, the simulation results were successfully transferred to a real robot acquiring stand-up behaviour. The selection of RL algorithms in the upper and lower level is well-suited for the given task. On the one hand computational costs are minimized as far as possible to keep the overall learning period short, on the other hand a continuous RL algorithm is applied in the lower level to derive an optimal control law for the actuators. There exists no formal convergence proof for the hierarchical RL approach presented here. Policies might become sub-optimal when combining different sub-problems that have been learned. Therefore, Morimoto et al. stress that it is important to choose proper state-variables in the upper level in order to set up the low-dimensional state-space. Designer knowledge is required at many stages: parameter tuning, definition of appropriate reward functions, selection of appropriate state-variables in the upper level of the hierarchy and definition of an appropriate action step size $\Delta\underline{X}$ in the upper level. A-priori information about the problem to be learned can be included more easily compared to flat architectures due to the abstraction that is introduced by the hierarchy.

This is consistent with a research paper by Wyatt [23], who has investigated issues in putting RL onto robots, and indicates that application-specific approaches to RL are required concerning the solution of real-robot tasks. A systematic assembly of robot behaviour and designer knowledge is desired. The approach to learning of the stand-up task of a robot introduces some expert knowledge in the way the hierarchy is set up and the reward functions are designed. It has become obvious that hierarchical architectures facilitate the introduction of designer knowledge in RL without becoming a form of supervised learning.

A further matching statement on hierarchical approaches that are based on task decomposition by sub-goals can be found in Skelly [18]. The hypothesis that complex tasks can be learned faster by decomposing the overall task into smaller sub-tasks consists of three assumptions. One assumption that has to be treated very carefully and possibly does not always hold true is that complex tasks can be decomposed into a set of sub-tasks at all. The next assumption is that the learning speed is enhanced for smaller sub-tasks compared to large main-tasks. However, the agent's learning period for a

conceptually difficult sub-task may even be longer than for a rather simple task in the high-dimensional state-space. Nevertheless, many implementations of hierarchical approaches have shown that the assumption of speeding up learning by task decomposition is justified from an experimentation-driven point of view. The last assumption is that the original task can be solved by combining the solutions of different sub-tasks. Any statement on this assumption requires a closer look at the concrete algorithms that are used. Concerning the hierarchical RL approach of Morimoto et al., the sequential combination of learned local trajectories actually solves the stand-up task and provides better performance than non-hierarchical approaches. It is stressed once more that the performance of hierarchical approaches in real-life is strongly task-dependent and a large amount of further investigation is required each time the algorithms are transferred to new problems.

Interesting aspects in hierarchical RL not covered so far by the approach of Morimoto et al. are the automatic selection of proper state variables in the upper level by learning and an autonomous adjustment of the action step size $\Delta\underline{X}$ in the upper level. It is expected that this kind of generalization will be quite difficult to achieve. Similar approaches to hierarchical RL have been developed that are less application-specific. They provide reusability of the modules in the lower level for the learning of several sub-tasks. Compositional Q-learning proposed by Singh [19] and nested Q-learning proposed by Digney [5] can be listed. However, learning is expected to be slower with any of these approaches than with an approach optimized for solving a specific task.

5 Further Approaches in Hierarchical RL

5.1 Options Formalism

In the following, four further hierarchical RL approaches will shortly be outlined that are frequently referred to in literature. The first approach to be presented is the options-formalism proposed by Sutton et al. [21]. It is one of the most theoretically sound approaches to hierarchical RL presented so far. Options are considered as generalization of primitive actions in order to include courses of actions that are temporally extended. According to [21], a Markov option o can be described by a triplet

$$o = \langle I, \pi, \beta \rangle \tag{5.1}$$

, where

$I \subseteq X$	is the input set of states in which an option o may be started,
$\pi: \mathrm{X} \times \mathrm{A} \rightarrow [0,1]$	is the policy followed during an option o and
$\beta: \mathrm{X} \rightarrow [0,1]$	is the probability of terminating in each state.

The agent cannot select the next option before the current option terminates. The idea behind options in hierarchical RL is closely linked to the theory of SMDPs that has been presented in section 3.2. The flat policy π, e.g. a non-hierarchical policy over actions, is replaced by a semi-Markov policy μ over options in order to define hierarchical options. A SMDP chooses among a set of options $O(\underline{x})$ and executes each option to termination, viewing options as indivisible units.

Intra-option learning is an extension to SMDP option learning. This extension has also been proposed by Sutton et al. [21] in 1998. In contrast to option learning, intra-option learning does not require the execution of an option to termination. This method learns from small fragments of experience within an option. The benefit from intra-option learning compared to standard option learning is that an offline-policy can be applied and temporal-difference methods allow learning about an option before the option terminates. Different values of options and models of options can be learned even simultaneously from the same experience. Thus, the speed of learning may be increased when implementing an intra-option learning method. Furthermore, there exists a formal convergence proof for intra-option value learning. In many other hierarchical RL approaches convergence to the optimal value function cannot be proofed theoretically at all.

5.2 Hierarchies of Abstract Machines (HAMs)

Another approach to hierarchical RL is Hierarchies of Abstract Machines (HAMs) that has been proposed by Parr et al. [16] in 1998. Prior knowledge of the RL problem can be introduced by generating hierarchical structures of partially specified machines. A complex problem is solved by decomposing the original problem into sub-problems, solving the sub-problems independently and finally recombining the sub-solutions. Sub-problems are solved by lower level machines and the switching between lower level machines is realized by non-deterministic finite state machines in the upper levels of the hierarchy. The admissible set of actions is constrained depending on the current hierarchical level. An analogy to hybrid control methods can be seen in here: A supervisor in the higher level switches to a new regulator in the lower level if its state enters a set of boundary states.

In order to apply the theory of HAMs in the context of RL, a variation of Q-learning called HAMQ-learning has been developed. The agent's update rule for HAMQ-learning can be found in [16]. There also exists a convergence proof for HAMQ-learning, given that an appropriate exploration strategy is applied. The RL approach seems to have a significant drawback as it uses an extended lookup-table for the action-value function. The usage of lookup-tables should be prevented when complex problems have to be solved efficiently. Barto et al. [2] indicate that the HAMs approach differs from the options-formalism by restricting the class of realizable policies for complex MDPs, while the options-formalism expands the action choices. The theory of SMDPs can also be found in HAMs: Sub-tasks to be solved in the lower level can still be modelled as finite MDPs, but the tasks in the higher levels of the hierarchy have to be modelled as finite SMDPs. Parr et al. have demonstrated for a dedicated navigation task that the time to converge to an optimal policy can be reduced by a factor of 4 by applying HAMs instead of applying policy iteration to the flat model. Note that this numerical example does not explicitly refer to RL but rather to a general solution of a complex MDP. However, it has been shown that the reduction of computational costs by applying HAMs in the context of RL is even of greater significance.

5.3 MAXQ Framework

Dietterich [4] has introduced the MAXQ decomposition of the value function to hierarchical RL. In the MAXQ-framework a hierarchy of SMDPs is set up whose solutions can be learned simultaneously. Task graphs are used to represent the structure of MAXQ-hierarchies. Regarding these task graphs, the link to structuring problems in programming languages by introducing sub-routines becomes obvious. The work of programmers in RL is facilitated if complex problems are already represented in the MAXQ framework. A task graph of a MAXQ-hierarchy comprises two types of nodes. The first type of node is a MAX node that either denotes primitive actions or sub-tasks. The second type of node is a Q node that denotes an action that can be performed to achieve its parent's sub-task, where the parent of a Q node is a MAX node. An example of a task graph for a RL problem where a taxi driver has to pick-up a client from a

source address and to deliver the client to a destination address is shown in Figure 5.1. The whole taxi problem is a core MDP that can be decomposed into a set of sub-tasks.

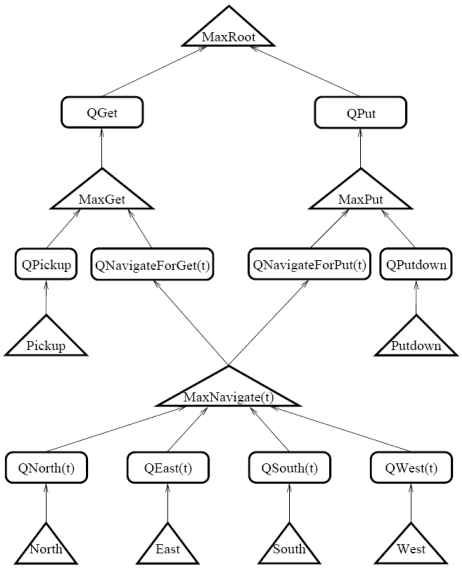


Figure 5.1: Example of a MAXQ Task Graph (taken from Dietterich [4])

At the beginning of the learning phase, MAX nodes can invoke their child nodes in an arbitrary order and even multiple times before completing their sub-tasks. At the end of the learning phase, a policy for each MAX node has been determined that assesses the order in which the child nodes have to be called. The formal decomposition of the value function requires that it is distinguished between Q nodes and MAX nodes, whose values either depend (in case of Q nodes) or not depend (in case of MAX nodes) on the context of performing their sub-task. Dietterich claims that the value function of any hierarchical policy can be represented with the MAXQ decomposition as long as the hierarchy is consistent with the MAXQ hierarchy. Once more Q-learning forms the basis of the learning algorithm that is extended here to MAXQ-0 learning. This short overview does not aim at presenting the algorithms of the MAXQ-framework. A summary of the main formulas of the MAXQ-framework can be found in [2] and the complete algorithms, including some convergence proofs, can be found in [4].

The hierarchy itself cannot be learned with the MAXQ-framework. This also holds true for all the other hierarchical RL approaches presented so far. In the next section an approach will be presented that is capable to learn the hierarchy of a task autonomously.

5.4 HEXQ Framework

The main advantage of the HEXQ-framework (Hengst [8], [9]) is the ability to abstract tasks dynamically. This differs from other approaches where the designer has to assess the components in each level of the hierarchy in advance. An agent learns in the HEXQ-framework to construct a hierarchical architecture of a task autonomously and thereby saves time and storage to solve the task. HEXQ addresses to problems, like maze grid problems and simple navigation tasks, where it is possible to isolate state variables and

partition the state-space into blocks. The decomposition of a task into sub-tasks comprises the following steps:

- The agent's interaction with the environment follows an exploration strategy.
- State variables are sorted according to their frequency of change.
- A new hierarchical level is set up for each state variable, beginning with sub-tasks in the lowest level of the hierarchy.
- Exits of a sub-task are defined by searching for state-actions pairs that change the value of other state variables or that change the probability distribution of possible next states and rewards. State-action pairs that cause the termination of the problem are also defined as exit.
- The agent identifies reusable sub-tasks.
- The agent tries to abstract sub-tasks at the next level, depending on the pre-specified degree of coarseness of the task hierarchy.

Barto et al. [2] point out that Hengst's approach to learning task hierarchies will fail to solve complex tasks where different state variables depend on each other. Thus, it is rather unlikely that HEXQ can be used to build a task hierarchy for a control problem as complex as controlling parts of a humanoid robot. Since there is a great demand for automating the abstraction of a task ad hoc and without requiring additional designer knowledge, RL researchers are still looking for efficient methods to introduce dynamic abstraction.

6 Summary

Similar to flat RL approaches, there also exists a great variety of hierarchical RL approaches. This report has been focused on one concrete hierarchical RL approach called task decomposition by sub-goals as proposed by Morimoto et al. [15]. The complex stand-up task of a three-link, two-joint robot can be learned at reasonable speed with this approach. The time to achieve the stand-up behaviour of a robot is reduced by learning sub-goals in the upper level of the hierarchy and local trajectories of a non-linear control law in the lower level of the hierarchy. However, this approach does not satisfy the following requirements for an ideal hierarchical RL method listed in [4]: 1) reusability of learned policies in the lower level modules; 2) non-hierarchical execution of sub-tasks to ensure convergence to the optimal policy; 3) autonomous learning of the task hierarchy without requiring designer knowledge in advance. Nevertheless, as discussed in section 4.5, researchers who have specialized on transferring RL theory to real applications stress on the opportunity of introducing designer knowledge into hierarchical architectures. Prior knowledge that is already available should also be used to accomplish a complex task. Deriving a general hierarchical RL approach that will work for many different control problems without any application-specific investigations and optimization is presently not considered to be realizable. Morimoto's approach to task decomposition by sub-goals presented in section 4 fulfils the most important issues in hierarchical RL, e.g. the learning speed is improved, efficient function approximation techniques are applied and a solution to the given stand-up task is found robustly.

Although theoretic convergence proofs are generally more difficult to derive for hierarchical RL approaches than for flat RL approaches, or sometimes even do not exist at all, recent research more and more focuses on introducing hierarchical architectures in RL. This is because of the desire to step from theoretical investigations and simple applications to more complex systems where it has shown that flat RL approaches are of no practical relevance. While general hierarchical RL methods have been outlined in section 5, the concrete accomplishment of hierarchical architectures is often application-dependent. Hierarchical RL approaches may be optimized for different purposes. The most important aspect is surely computational efficiency in presence of high-dimensional state-spaces, leading to an improved learning speed. Hierarchical architectures also introduce a level of abstraction that eases the implementation of complex controllers. Optimization may also refer to the reusability of learning modules in the lower levels of the hierarchy. Thus, the development costs can be reduced and the practical importance of hierarchical RL is stressed once more.

Although RL is already widely spread in the domain of computer science and game simulators nowadays, (hierarchical) RL can hardly be found in industry so far for solving complex control problems. There is notable progress in improving RL techniques to be applied to more complex real-life problems by introducing hierarchical

architectures. However, the step beyond test applications to solving large control tasks by hierarchical RL in industry on a grand scale is still missing. Issues of recent research in the domain of RL are dynamic abstraction to learn task hierarchies automatically and the development of cooperative hierarchical RL algorithms, for example see Ghavamzadeh et al. [6], to support efficient coordination of multiple agents.

Appendix

Pseudo-code of the **Q(λ)-learning algorithm** according to Peng et al. [17]:

1) Initialization: $Q(\underline{x},\underline{u}) = 0$ and $e(\underline{x},\underline{u}) = 0$ for all $\underline{x}$ and $\underline{u}$

2) For each episode repeat:

a) $\underline{x}_t \leftarrow$ current state

b) Choose an action $\underline{u}_t$ according to current exploration policy.

c) Carry out action $\underline{u}_t$ in the world. Observe the short-term reward r_t and the new state $\underline{x}_{t+1}$.

d) $\delta_t' \leftarrow r_t + \gamma \cdot V(\underline{x}_{t+1}) - Q(\underline{x}_t,\underline{u}_t)$

e) $\delta_t \leftarrow r_t + \gamma \cdot V(\underline{x}_{t+1}) - V(\underline{x}_t)$

f) For all state-action pairs $(\underline{x},\underline{u})$:

- $e(\underline{x},\underline{u}) \leftarrow \gamma \cdot \lambda \cdot e(\underline{x},\underline{u})$
- $Q(\underline{x},\underline{u}) \leftarrow Q(\underline{x},\underline{u}) + \alpha \cdot e(\underline{x},\underline{u})\ \delta_t$

g) $Q(\underline{x}_t,\underline{u}_t) \leftarrow Q(\underline{x}_t,\underline{u}_t) + \alpha \cdot \delta_t'$

h) $e(\underline{x}_t,\underline{u}_t) \leftarrow e(\underline{x}_t,\underline{u}_t) + 1$

where $V(\underline{x}_t) = \max_{\underline{u}} Q(\underline{x}_t,\underline{u}_t)$

List of Figures

Bibliography

[1] E. Alpaydin. (2004). Introduction to Machine Learning. The MIT Press, Cambridge, Massachusetts.

[2] A.G. Barto, S. Mahadevan. (2003). Recent Advances in Hierarchical Reinforcement Learning. Kluwer Academic Publishers, Discrete Event Dynamic Systems: Theory and Application, vol. 13, pp. 41-77.

[3] R. Coulom. (2003). A Model-Based Actor-Critic Algorithm in Continuous Time and Space. In Proceedings of the Sixth European Workshop on Reinforcement Learning, edited by A. Dutech and F. Garcia, Nancy, France.

[4] T.G. Dietterich. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. Journal of Artificial Intelligence Research, vol. 13, pp. 227-303.

[5] B.L. Digney. (1998). Learning hierarchical control structures for multiple tasks and changing environments. In Proceedings of the Fifth Conference of the Simulation of Adaptive Behavior, MIT Press, Cambridge, Massachusetts, pp. 321-330.

[6] M. Ghavamzadeh, S. Mahadevan, R. Makar. (2006). Hierarchical Multi-Agent Reinforcement Learning. Kluwer Academic Publishers, Journal of Machine Learning Research (JMLR-2007), vol. 8, pp. 2629-2669.

[7] M. Harmon, S. Harmon. (1997). Reinforcement Learning. A Tutorial. Defense Technical Information Center (DTIC), Accession Number: ADA323194, Proxy URL http://handle.dtic.mil/100.2/ADA323194 (date: 2008-03-14).

[8] B. Hengst. (2002). Discovering Hierarchy in Reinforcement Learning with HEXQ. In Proceedings of the 19th International Conference on Machine Learning, UNSW Sydney, Australia, ICML 2002, pp 243-250.

[9] B. Hengst. (2004). Model Approximation for HEXQ Hierarchical Reinforcement Learning. 15th European Conference on Machine Learning (ECML), ©Springer-Verlag, Pisa, Italy.

[10] M. Hutter. (2005). Universal Artificial Intelligence. Sequential Descisions Based on Algorithmic Probability. Springer-Verlag, Berlin, Heidelberg.

[11] H. Kimura. (2007). Reinforcement Learning in Multi-dimensional State-Action Space using Random Rectangular Coarse Coding and Gibbs Sampling. In SICE Annual Conference 2007, Takamatsu, Japan, pp. 2754-2761.

[12] R.M. Kretchmar, C.W. Anderson. (1997). Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning. In Proceedings of the International Conference on Neural Networks (ICNN'97).

[13] H. Miyamoto, J. Morimoto, K. Doya, M. Kawato. (2004). Reinforcement learning with via-point representation. Elsevier Science, Neural Networks, vol. 17, pp. 299-305.

[15] J. Morimoto, K. Doya. (2001). Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. Elsevier Science, Robotics and Autonomous Systems, vol. 36, pp. 37-51.
[16] R. Parr, S.J. Russell. (1998). Reinforcement learning with hierarchies of machines. In Advances of Neural Information Processing Systems 10 (NIPS-10), MIT Press, Cambridge, Massachusetts, pp. 1043-1049.
[17] J. Peng, R.J. Williams. (1996). Incremental Multi-Step Q-Learning. In Recent Advances of Reinforcement Learning, edited by L.P. Kaelbling, Kluwer Academic Publishers, Boston, pp. 283-290.
[18] M.M. Skelly. (2004). Hierarchical Reinforcement Learning with Function Approximation for Adaptive Control. Dissertation at the Case Western Reserve University, Department of Electrical Engineering and Computer Science, May 2004.
[19] S.P. Singh. (1992). Transfer of learning by composing solutions of elemental sequential tasks. Machine Learning 8, pp. 323-339.
[20] S.P. Singh, R.S. Sutton. (1996). Reinforcement Learning with Replacing Eligibility Traces. In Recent Advances of Reinforcement Learning, edited by L.P. Kaelbling, Kluwer Academic Publishers, Boston, pp. 123-158.
[21] R.S. Sutton, D. Precup, S.P. Singh. (1998). Intra-option learning about temporally abstract actions. In Proceedings of the 15th International Conference on Machine Learning, Morgan Kaufmann, pp. 556-564.
[22] R.S. Sutton, A.G. Barto. (1998). Reinforcement Learning: An Introduction. The MIT Press, Cambridge, Massachusetts, 3rd printing, 2000.
[23] J. Wyatt. (1996). Issues in putting reinforcement learning onto robots. DAI Research Paper, No. 784, University of Edinburgh.