Evolution of Meta-parameters in Reinforcement Learning

Anders Eriksson

TRITA-NA-Eyynn

Abstract

A crucial issue in reinforcement learning applications is how to set meta-parameters, such as the learning rate and "temperature" for exploration, to match the demands of the task and the environment. In this thesis, a method to adjust meta-parameters of reinforcement learning by using a real-number genetic algorithm is proposed. Simulations of foraging tasks show that appropriate settings of meta-parameters, which are strongly dependent on each other, can be found by evolution. Furthermore, hardware experiments using Cyber Rodent robots verify that the meta-parameters evolved in simulation are helpful for learning in real hardware.

Evolution av meta-parametrar i reinforcement learning

Sammanfattning

Ett kritiskt problem i reinforcement learning-applikationer är hur man ska sätta metaparametrar, såsom inlärningshastighet och "temperatur" för utforskning, för att uppfylla kraven som ställs av uppgiften och miljön. I den här rapporten föreslås en metod för att justera metaparametrar i reinforcement learning genom att använda genetiska algoritmer baserade på flyttalsaritmetik. Simuleringar av uppgifter där målet är att finna föda visar att riktiga inställningar av metaparametrar, som har starka beroenden av varandra, kan hittas genom evolution. Dessutom verifierar experiment utförda med hjälp av Cyber Rodent-robotar att metaparametrar optimerade i mjukvara är användbara vid inlärning på hårdvaruplattform.

Acknowledgment

Contents

List of Figures

List of Tables

List of abbreviations

API	-	Application Program Interface
CMOS	-	Complementary Metal-Oxide Semiconductor
CR	-	Cyber Rodent
CREST	-	Core Research for Evolutional Science and Technology
CRmS	-	Cyber Rodent matlab Simulator
EA	-	Evolutionary Algorithm
eCos	-	Embedded Cygnus OS
ES	-	Evolutionary Strategy
GA	-	Genetic Algorithm
gcc	-	GNU Compiler Collection
GP	-	Genetic Programming
IR	-	Infra Red
LED	-	Light-Emitting Diode
MSE	-	Mean Square Error
RBF	-	Radial Basis Function
TD	-	Temporal Difference
ATR	-	Advanced Telecommunication Research Institute International

Chapter 1

Introduction

During lifetime, humans and animals develop skills and behaviors to be able to survive in the surrounding environment. In the early stage of an individual's life, basic movements as eye, head and arm movements are trained repeatedly. During this learning process, there are no clear instructions of what is right or wrong but instead the connection between motor control and sensory input creates information of cause and action. Humans and animals are able to utilize this information to learn how to perform various tasks necessary for survival. This is known as direct learning.

Reinforcement learning [1, 32] (RL) is a general, computational approach to solve the problem of an agent's direct learning from interaction with an environment. The reinforcement learning framework is today widely used both as a proposed model within the human and animal learning research [9] and as a tool when making adaptive or intelligent applications. There have been many successful implementations such as game playing programs [33], robotic control [7, 26] and resource allocation [31]. Still, creating reliable and stable applications using RL involves many problems.

A known issue is the setting of parameters that controls the process of learning, such as speed of learning, environment exploration and time scale of reward prediction. These parameters are called *meta-parameters* and are critical in the RL algorithm to achieve stability and learning convergence. The fact that these are usually heuristically determined by a human expert creates problems when making applications. The absence of a general automatic way to tune the meta-parameters introduces uncertainties about optimality and limitations to the algorithm's abilities to adapt to environmental changes.

Human experts often use a trial-and-error method to tune the meta-parameters. It is a difficult process and to obtain optimal meta-parameter settings without extensional investigations is impossible. Usually, the number of dimensions of the meta-parameter space is large, and problems as getting stuck in a local maximum and ignoring the importance of vital meta-parameters are common. Moreover, the meta-parameters themselves have a complicated dependency. Changing one probably implies that a number of other meta-parameters should be adjusted as compens-

ation. During development of an RL implementation, these kinds of dependencies become complex and are difficult to take into account.

Drastic changes in the environmental conditions completely deform the meta-parameter space. Specially, applying an application developed in a clean laboratory environment to systems like robotics performing in real life situations is not reasonable. Often, this kind of environmental changes involve a complete new meta-parameter setting and system redesign.

1.1   Previous Research

Attempts to generalize the tuning of meta-parameters has been ongoing research during the last decade without ground breaking, globally accepted methods. Theories applying risk minimization [35], Bayesian estimation [27] and attempts towards online adaptations as exploration control algorithms [17] have been proposed. However, most applications depend on heuristic search for the meta-parameter settings and are still based on hand tuning by human experts.

General issues of combining learning and evolution have been studied and evaluated in several works [28, 5]. Integration is possible in various ways including different restrictions and levels of efficiency. Approaches to use genetic algorithms (GAs) to optimize initial meta-parameter settings in RL have been investigated by Tatsuo Unemi [36]. The study considers a discrete environment using one step Q-learning [32] in simulation to optimize some of the meta-parameters in the RL algorithm. The focus is on the interaction between learning and evolution under different stabilities of the environment.

1.2   Problem Formulation and Goals

The aim of this study is to investigate how meta-parameters in a RL algorithm can be decided automatically using GA. The learning problem setup is based on biological restrictions, where a rodent like agent acting in an environment with continuous coordinates is considered. The Cyber Rodent robot, see chapter 3, has been used as a model for the agent.

The basic idea is to encode the meta-parameters of the RL algorithm as the agent's genes, and to take the meta-parameters of best-performing agents in the next generation. The specific questions of interest in this study are: 1) whether GA can successfully find appropriate meta-parameters subject to their mutual dependency, and 2) whether the meta-parameters optimized by GA in simulation can be helpful in real world implementations of RL.

1.3   Organization

This report is organized as follows: in chapter 2 the fundamental theories of RL and GA are covered including techniques used in recent research and theory important

to understand the content of this study. Chapter 3 describes the Cyber Rodent robot and the Cyber Rodent Matlab Simulator. The proposed method for automatic decision of meta-parameters is explained in chapter 4 and in chapter 5, the experiments and results used to verify it are presented. Finally, chapter 6 summarizes the report.

1.4   Cyber Rodent Project

The research presented in this report has been performed within the Cyber Rodent project at ATR, Advanced Telecommunication Research Institute International, located in Kyoto, Japan. ATR is an independent corporation that conducts both basic and advanced research in the field of telecommunications.

The Cyber Rodent project is carried out at department 3 within HIS, Human Informations Science Laboratories, directed by Dr. Mitsuo Kawato. The field of research is focused on human information processing and communication mechanisms and the main projects are done on speech processing and acquisition and computational neuroscience.

The Cyber Rodent project is directed by Ph.D. Kenji Doya, senior researcher at ATR and director of the meta-learning and neuro-modulation project at CREST[1]. CREST was initiated in Japan in 1995 to encourage basic research by invigorating the potential of universities, national laboratories, and other research institutions with the clear aim to build up the tangible foundation for the future directions of Japan's science and technology. The goal of the Cyber Rodent Project is to understand the adaptive mechanisms necessary for artificial agents that have the same fundamental constraints as biological agents, namely self-preservation and self-reproduction. Previous studies of RL assumed arbitrarily defined "rewards", but in the models of biological systems, rewards should be grounded as the mechanisms for self-preservation and self-reproduction. To this end a colony of autonomous robots that have the capabilities of finding and recharging from battery packs and copying programs through infrared communication ports has been developed. The purpose is to evolve learning algorithms for various behaviors, meta-learning algorithms for robust, efficient learning, communication methods for foraging and mating and evolutionary mechanisms supporting adaptive behaviors.

---

[1]The Core Research for Evolutional Science and Technology program.

Chapter 2

Background

This chapter covers the basic terminology and the theory necessary to understand the proposed method for automatic decision of meta-parameters presented in this report.

2.1 Reinforcement Learning

Reinforcement learning (RL) is a computational approach for decision-making that emphasizes the interaction between an agent and its surrounding environment. More specific, the RL problem is about how to "learn" how to connect situations to actions that maximize a numerical reward signal over time.

An important point is that RL is not supervised learning [25]. The learning agent does not rely on "true" or "correct" answers as in many other branches of machine learning, but learns behaviors that mirror goals embedded in a reward signal. By traversing the environment the agent collects information about the reward signal and learns how to make the correct connection between situations and actions. Because the immediate reward given as a response to an action does not reflect the cumulative reward that the agent will be able to get in the long run, the problem is challenging.

This section begins by explaining the concepts of reinforcement learning briefly, with the purpose of giving the reader an understanding of the nature of the problem and the basic structure of the method. Thereafter, properties of the RL problem and underlying theories are presented. From section 2.1.4, *Temporal difference learning*, specific methods for solving the problem are explained. These sections cover some of the most popular approaches used in RL applications today and also an important theory in understanding the following chapters of this report.

2.1.1 Basic Concepts

Figure 2.1 shows the basic structure of the RL model where an agent interacts with the environment through the three signals *state*, *action* and *reward*. The signals are discrete in time, $t = 1 \ldots T$, where the agent makes a decision in each step.
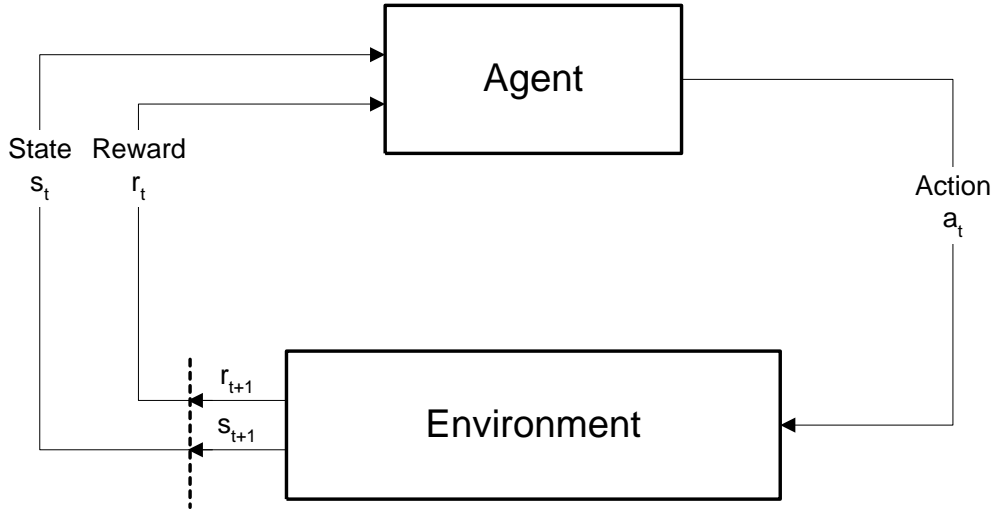
Figure 2.1. The reinforcement learning framework including the three major feature signals: action, state and reward.

State The state, $s_t \in \mathbf{S}$, is the agent's input from the environment at time t where $\mathbf{S}$ is the state space consisting of one or more parameters, discrete or continuous, that defines the different situations that the agent has to be able to respond to.

Action Based on the state signal, the agent makes the decision to take an action $a_t \in \mathbf{A}$ where $\mathbf{A}$ is the predefined action space, discrete or continuous.

Reward When taking an action at time t, the environment is effected and as an immediate response the agent receives a reward $r_{t+1} \in \mathbf{R}$ and a new state input, s_{t+1}.

Task A RL task is defined by a complete specification of the environment, including state and reward signals.

Policy The policy, $\pi(s,a)$, is the function that maps states to actions by defining probabilities for all actions in each state. The purpose of the RL algorithm is to derive an optimal policy, π^*, that maps states to actions in a way that maximizes the cumulative reward during the lifetime of the agent.

Value function The *state-value function* $V^\pi(s)$ (or *action-value function*, $Q\,\pi(a,s)$), is an estimation of the future expected reward associated with a state (or a state and an action) when following policy π. That is, when being in state s (or being in state s taking action a) the value function estimates how much reward the agent can expect to collect until the end of its lifetime following policy π. By knowing the true optimal value function V^* (Q^*), the agent gets indirect knowledge of the optimal policy, π^*.
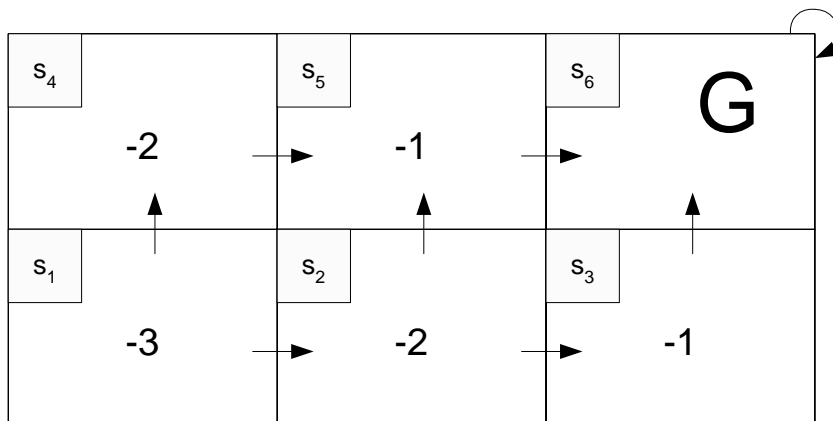
Figure 2.2. Example of a RL grid-world task. The figure shows the optimal policies to reach the goal square **G** and the associated value function.

Example The following simple example aims at giving a better understanding of the basic concepts and how a concrete RL problem can be approached. **Figure 2.2** shows a grid-world where the task is for an agent to find its way to a goal square G from any other square in the world. The state space consists of the squares in the grid-world, $s_1, s_2, \ldots, s_6$, where the state signal is the unique number of the current square. The action space consists of the actions go north, south, east and west where it is possible. The reward signal is -1 for each step taken by the agent until it reaches the goal square.

The set of optimal policies is given by the arrows in the figure (note that there is more than one way to reach the goal square). In each state, the decision is the best possible to achieve the maximum cumulative reward. The value function for this policy is given by the values in the squares. $\qquad\square$

2.1.2 The Markov Property

A RL problem is assumed to have the so called *Markov property*. This means that the information in the state signal received by the agent includes all that the agent needs to make a correct next decision. No information about earlier actions or states that led to the current position is needed. The dynamics of the environment can be defined as:

$$\Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\} \tag{2.1}$$

for all s', r, s_t and a_t. Usually in RL tasks, the Markov property is not completely fulfilled. The state signal often lacks information or precision to have Markov property, but it is still appropriate and necessary to regard is as approximatively Markov when solving RL problems.

A RL task satisfying the Markov property is also a *Markov decision process* (*MPD*). This means that given a state s and an action a there exists a probability of ending up in the next state according to:

$$P_{ss'}^a = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \tag{2.2}$$

Also, given a state s, an action a and the next state s', the expected next reward can be estimated as:

$$R_{ss'}^a = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\} \tag{2.3}$$

These two properties of the RL problem are important to keep in mind. First, the agent only reacts on the current information received from the environment. Secondly, when taking an action the next state is not fully predictable, but given by a probability distribution. Following the last statement, the predicted cumulative reward from a given state can not be a true number, but an expected value.

2.1.3   Optimal Value Function

Calculating the optimal value function, that indirectly gives knowledge about the optimal policy, is the most common approach to solve RL problems. As the optimal policy maximizes the cumulative reward over time, it is important to know exactly what cumulative reward means. The cumulative reward from time t until the end of the agent's lifetime is defined as:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T \tag{2.4}$$

where T is the final step in the learning process. RL tasks can be put into two categories, *episodic tasks* and *continuous tasks*, where the cumulative reward have somewhat different properties. Episodic tasks are tasks where the agent-environment interaction breaks naturally into subsequences. These include a *terminal state* where the task ends and resets to the initial setup. An example of a episodic task is a maze task, where the agent is repositioned when reaching the exit during the learning process.

The second category of tasks, the continuous tasks, has no terminal state. Instead the agent is allowed to interact with the environment without resettings. In this case the definition of the cumulative reward in equation 2.4 involves a problem if $T \rightarrow \infty$. It would imply that the cumulative reward also could be infinite, making it impossible to maximize. Therefore, a *discounting factor* is introduced to limit the sum:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.5}$$

where γ is $1 > \gamma > 0$. This is the first meta-parameter introduced and is called the *discount rate*. It simply decides to what degree the future reward will be taken into account in the present time step.

Knowing how to limit the future reward sum, the value function can be defined more formally. First, a value function is always defined with respect to a specific policy. A value function describes the expected future reward when being in a state or when being in a state and taking an action. Obviously, the future reward is dependent on the future actions, and it is therefore necessary to associate the value function to a policy. $V^\pi(s)$ is defined to be the expected reward when being in state s and following policy π. For MPDs the definition is:

$$
\begin{aligned}
V^\pi(s) &= E_\pi\{R_t|s_t = s\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\right\}
\end{aligned}
\tag{2.6}
$$

This function is called the *state-value function* for policy π. The definition of the value function for a state and an action is similar:

$$
\begin{aligned}
Q^\pi(s,a) &= E_\pi\{R_t|s_t = s, a_t = a\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s, a_t = a\right\}
\end{aligned}
\tag{2.7}
$$

and is called the *action-value function* for policy π.

There are several different ways to estimate the value function in a RL problem. However, they all utilize a fundamental recursive property of the value function. There exists a relation between the value of the current state s and the value of all its possible successor states s', knowing the current policy π, for all value functions. This relation is:

$$
\begin{aligned}
V^\pi(s) &= E_\pi\{R_t|s_t = s\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\right\} \\
&= E_\pi\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s\right\} \\
&= \sum_a \pi(s,a)\sum_{s'} P_{ss'}^a\left[R_{ss'}^a + \gamma E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s'\right\}\right]
\end{aligned}
\tag{2.8}
$$

and the resulting equation is known as the *Bellman equation*. The Bellman equation says that the value of a state s, given a policy π, is the discounted value of the next state s' plus the expected reward given during the state transition. The next state is formulated as an average over all possible successor states weighted by their probability of occurring.

Knowing the policy π, V^π is the unique solution to the Bellman equation. When solving a RL problem, the aim is to find the optimal policy π^* and therefore also the optimal value function V^*. It is defined as:

$$
V^*(s) = \max_\pi V^\pi(s)
\tag{2.9}
$$

for all states $s \in \mathbf{S}$. Equally, the optimal action-value function is defined as:

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) \tag{2.10}$$

for all states $s \in \mathbf{S}$ and all actions $a \in \mathbf{A}$. As the action-value function is dependent on the value of the next state, the definition of $Q^*(s, a)$ can be written in terms of $V^*(s)$ according to:

$$Q^*(s, a) = E\left\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\right\} \tag{2.11}$$

Now, the Bellman equation can be rewritten to suit the goal of finding the optimal value function and policy:

$$
\begin{aligned}
V^*(s) &= \max_{a} Q^{\pi^*}(s, a) \\
&= \max_{a} E_{\pi^*}\{R_t | s_t = s, a_t = a\} \\
&= \max_{a \in A(s)} E_{\pi^*}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \\
&= \max_{a \in A(s)} E_{\pi^*}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a\right\} \\
&= \max_{a \in A(s)} E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\} \\
&= \max_{a \in A(s)} \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V^*(s')\right] \tag{2.12}
\end{aligned}
$$

and by rewriting the last two equations

$$
\begin{aligned}
Q^*(s, a) &= E\{r_{t+1} + \gamma \max_{a} Q^*(s_{t+1}), a' | s_t = s, a_t = a\} \\
&= \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma \max_{a} Q^*(s', a')\right] \tag{2.13}
\end{aligned}
$$

Equation 2.12 or 2.13 could theoretically be used to find the optimal policy for correct stated RL tasks. However, to calculate the optimal policy in this way for not purely trivial tasks is not possible even by means of computers. The problems usually contain too much information to be solved in reasonable time. Instead, RL is about approximating the optimal policy, where the Bellman equation lies as a foundation of the methods.

Example Consider the same task as in the example in **section 2.1.1**. Let the states be $s_1, s_2, \ldots, s_6$ and the actions be *north*, *south*, *east* and *west* be n, s, e and w respectively, see **Figure 2.3**.

First, lets formulate the Bellman equation for a given policy $\pi(s, a) = 1/|a|$. This would result in the six linear equations:

$$V^{\pi}(s_1) = 1/2(-1 + \gamma V^{\pi}(s_2)) + 1/2(-1 + \gamma V^{\pi}(s_4))$$

9
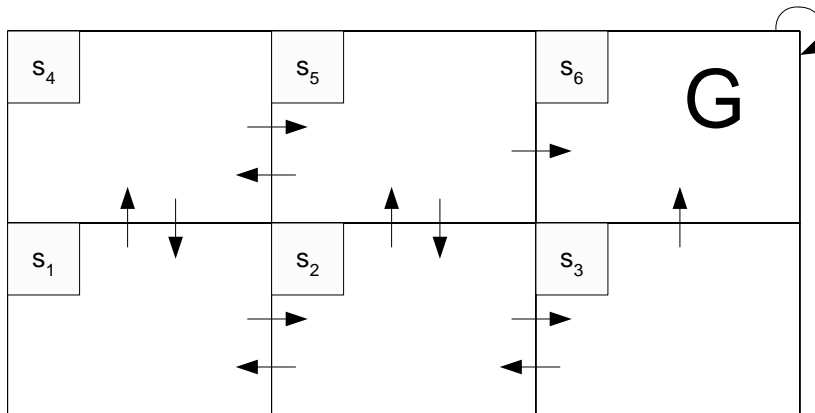
Figure 2.3. Grid-world task including the complete state and action spaces. **G** defines the goal state.

$$
\begin{aligned}
&= 1/2(-2 + \gamma(V^\pi(s_2) + V^\pi(s_4))) \\
V^\pi(s_2) &= 1/3(-3 + \gamma(V^\pi(s_1) + V^\pi(s_5)) + V^\pi(s_3))) \\
V^\pi(s_3) &= 1/2(-1 + \gamma(V^\pi(s_2) + V^\pi(s_6))) \\
V^\pi(s_4) &= 1/2(-2 + \gamma(V^\pi(s_1) + V^\pi(s_5))) \\
V^\pi(s_5) &= 1/3(-2 + \gamma(V^\pi(s_4) + V^\pi(s_2) + V^\pi(s_6))) \\
V^\pi(s_6) &= 0
\end{aligned}
$$

The unique solution V^π can be computed using known methods for solving linear equation systems.

In the optimal Bellman equation case the policy is not given. Considering the same problem, the first of the six optimal Bellman equations is:

$$
V^*(s_1) = \max \left\{ \begin{array}{l} P^e_{s_1 s_2}[R^e_{s_1 s_2} + \gamma V^*(s_2)] + P^e_{s_1 s_4}[R^e_{s_1 s_4} + \gamma V^*(s_4)] \\ P^n_{s_1 s_2}[R^n_{s_1 s_2} + \gamma V^*(s_2)] + P^n_{s_1 s_4}[R^n_{s_1 s_4} + \gamma V^*(s_4)] \end{array} \right\} \quad (2.14)
$$

The five equations that are left out have the same shape, i.e. maximizing over all possible actions for each state. The discount rate γ and the environment dynamics parameters $P^a_{s_b s_c}$ and $R^e_{s_1 s_2}$ for all possible actions $a \in \mathbf{A}(s)$, corresponding to all valid state transition pairs $s_b s_c$, have to be known to be able to solve the equation system.

Note that in the grid-world case, for example $P^e_{s_1 s_4}$ is probably 0, as many other state transition probabilities (choosing action east in state s_1 will make the agent end up in state s_2). In the general RL case though, these transitions are non-deterministic, and the complexity of the equation system increases.

As conclusion, the RL problem can be solved using the optimal Bellman equation. However, there are in practice at least three necessary assumptions that are almost never true simultaneously:

1. That knowledge of the dynamics of the environment is at hand.

2. That enough computational resources to derive the solution are available.

3. That the task has Markov property.

$\square$

2.1.4 Temporal Difference Learning

To solve not purely trivial RL tasks, the value function has to be estimated rather than determined exactly. As stated in the previous section these kinds of problems are not feasible to solve completely. There exist several classes of methods to approach this problem. *Dynamic programming* methods [3] are well developed from a mathematical point of view but need a complete model of the environment to solve the problem. *Monte Carlo* methods [32] do not have this restriction but are not suited for incremental step-by-step approaches. The third group of methods, *Temporal difference* methods, can be seen as a combination of the two above mentioned groups, able to handle step-by-step implementation without the need of extensive information about the environment.

Temporal difference learning is an incremental way to estimate the value function, given a policy. While the agent moves in the environment, it gains experience about the properties of the reward signal and improves its current guess of the true value function. In general, incremental algorithms that estimate a target function use an old estimation and update it with an error derived from the experiences taken in the last step. This can be written as:

$$New\_estimate \leftarrow Old\_estimate + \overbrace{(Target - Old\_estimate)}^{error\ estimate} \tag{2.15}$$

Suppose that the target is the cumulative reward from each state in a reinforcement learning problem. This signal can be noisy, giving wrong information about the true target function. Also, in a non-stationary environment where the target function changes over time, this equation will make wrong estimations. The solution is to make smaller steps towards the target. Thus, equation 2.15 is rewritten as:

$$New\_estimate \leftarrow Old\_estimate + \alpha(Target - Old\_estimate) \tag{2.16}$$

where α is a step-size parameter, $1 > \alpha > 0$. The step-size parameter α makes the estimation approach the target function slowly, filtering noise. Moreover, recent information about the target function is weighted higher than information from the past, enabling the equation to handle non-stationary tasks. The step-size parameter α is the second meta-parameter that is introduced, called the *learning rate*, and it controls how fast the estimation approaches the true value function.

The influence of α on the distribution of reward within the algorithm is shown by equation 2.17. Let the estimation and the target function at time t be Q_t and r_t respectively.

$$Q_t \quad = \quad Q_{t-1} + \alpha[r_t - Q_{t-1}]$$

11

$$\begin{aligned}
&= \quad \alpha r_t + (1-\alpha)Q_{t-1} \\
&= \quad \alpha r_t + (1-\alpha)\alpha r_{t-1} + (1-\alpha)^2 Q_{t-2} \\
&= \quad (1-\alpha)^t Q_0 + \sum_{i=0}^{t} \alpha(1-\alpha)^{t-i} r_i
\end{aligned} \tag{2.17}$$

Note that the weight $(1-\alpha)^{t-i}$ of the target function input r_i depends on how many time steps ago the input was received. This property enables the incremental updating rule to adjust to non-stationary environments.

When estimating the value function in RL problems, the target function is the future cumulative reward and from equation 2.16 follows:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)] \tag{2.18}$$

This formulation implies that the cumulative reward R_t is known which is not the case until the end of the episode. In temporal difference methods this problem is solved by predicting the cumulative reward in each step. Recall from equation 2.8, the Bellman equation, that:

$$\begin{aligned}
V^\pi(s) &= \quad E_\pi \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s \right\} \\
&= \quad E_\pi \left\{ r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s \right\}
\end{aligned} \tag{2.19}$$

The true value of $V^\pi(s_{t+1})$ in equation 2.19 is not known, instead the current estimation $V_t(s_{t+1})$ is used as an approximation to predict the future cumulative reward. From this follows the basic temporal difference updating rule known as *TD(0)*:

$$V(s_t) \leftarrow V(s_t) + \alpha \underbrace{[\overbrace{r_{t+1} + \gamma V(s_{t+1})}^{reward\ prediction} - V(s_t)]}_{TD-error} \tag{2.20}$$

Algorithm *TD(0)* explains the procedure of how to use equation 2.20 to estimate V^π for a given policy π.

Algorithm *TD(0)*
1.   Initialize $V(s)$ arbitrarily, let π be the policy to be evaluated
2.   **repeat** (for each episode)
3.   Initialize s
4.   **repeat** (for each step of episode)
5. $a \leftarrow$ action given by π for s
6.   Take action a; observe reward, r, and next state, s'
7. $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
8. $s \leftarrow s'$
9.   **until** s is terminal

2.1.5 Eligibility Traces

Eligibility traces is a method that can be combined with numerous RL algorithms to improve the efficiency of the algorithms by increasing the error information to the states. The method is not applicable in all RL tasks but can in many cases be a crucial component to make the learning procedure converge within reasonable time. An example of a method that uses eligibility traces is $TD(\lambda)$. It is an extension of $TD(0)$, introduced in section 2.1.4, where λ is a parameter that controls the use of the eligibility traces.

The fundamentals of eligibility traces is about how to predict rewards. In $TD(0)$, the reward prediction is made based on one step in the environment, using the direct reward and the current value of the next state to estimate the cumulative reward:

$$E\{R_t\} = r_t + \gamma V_t(s_{t+1})$$

The basic concept of eligibility traces is to make prediction by means of more than one step into the future. If $E\{R_t^{[n]}\}$ is the estimation of the cumulative reward n steps into the future, then:

$$
\begin{aligned}
E\{R_t^{[1]}\} &= r_{t+1} + \gamma V_t(s_{t+1}) \\
E\{R_t^{[2]}\} &= r_{t+1} + \gamma(r_{t+2} + \gamma V_t(s_{t+2})) \\
&= r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2}) \\
E\{R_t^{[n]}\} &= r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})
\end{aligned}
\tag{2.21}
$$

Usually, equation 2.21 is not used as stated here. The direct usage of this reward prediction would imply that the agent has to take n steps into the future before being able to update $V_t(s)$. Instead, the $TD(\lambda)$ method calculates the TD-error δ_t at time t and back-propagates it to previously visited states. For each state, δ_t is scaled according to the time since that state was visited.

In this case, each state is associated with its own eligibility trace, $e_t(s)$. This trace is exponentially decayed in every step, and updated when visited according to:

$$
e_t(s) = \begin{cases} \gamma \lambda e_t(s) & \text{if } s \neq s_t \\ \gamma \lambda e_t(s) + 1 & \text{if } s = s_t \end{cases}
\tag{2.22}
$$

for all non-terminal states s, where λ is the trace *decay factor* and the third meta-parameter introduced. Obviously, λ controls the size of the traces and therefore also how far back in the past the TD-error is back-propagated. **Figure 2.4** illustrates the concept of traces and **Algorithm** *TD(λ)* describes how to combine eligibility traces with $TD(0)$.

Eligibility traces occur in more than one version considering the trace updating method. Equation 2.22 describes so called *accumulating traces*, that are used in **Algorithm** *TD(λ)*. In this case each trace is increased by 1 when visited. Another
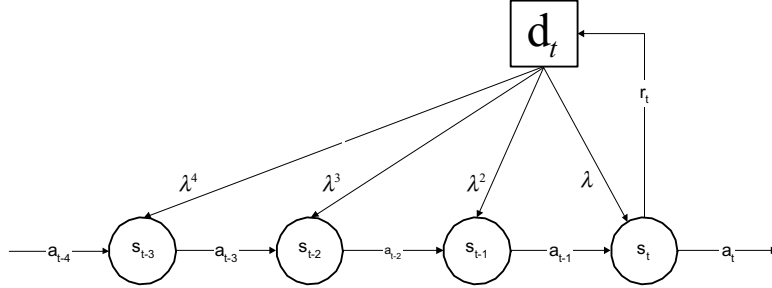
Figure 2.4. Conceptual scheme of eligibility traces in the TD(λ) algorithm. The TD-error at time t is back-propagated to previous visited states.

way to update the eligibility trace of a state when visited could be to reset it to 1, according to:

$$e_t(s) = \begin{cases} \gamma \lambda e_t(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases} \tag{2.23}$$

for all non-terminal states s. This kind of trace updating method is called *replacing traces*. Even though the difference between replacing traces and accumulating traces seems to be small, the effect on the time to convergence can be significant [32]. **Figure 2.5** shows the difference between the trace of a single state when using accumulating and replacing traces respectively.

2.1.6 State Space Exploration

So far methods for estimating the value function for a given policy has been discussed. In RL algorithms however, the goal is to find the optimal policy. The main issue in this process is how to select actions, i.e. how to explore the state space while estimating the value function. The problem is related to deciding the so called *exploration-exploitation* rate for reasons that will become clear in this section.

Algorithm *TD(λ)*
1.   Initialize $V(s)$ arbitrarily and $e(s) = 0$ for all $s \in \mathbf{S}$
2.   **repeat** (for each episode)
3.   Initialize s
4.   **repeat** (for each step of episode)
5. $a \leftarrow$ action given by π for s
6.   Take action a; observe reward, r, and next state, s'
7. $\delta \leftarrow r + \gamma V(s') - V(s)$
8. $e(s) \leftarrow e(s) + 1$
9.   **for** all s
10.   **do** $sV(s) \leftarrow V(s) + \alpha \delta e(s)$
11. $se(s) \leftarrow \gamma \lambda e(s)$
12.   **until** s is terminal
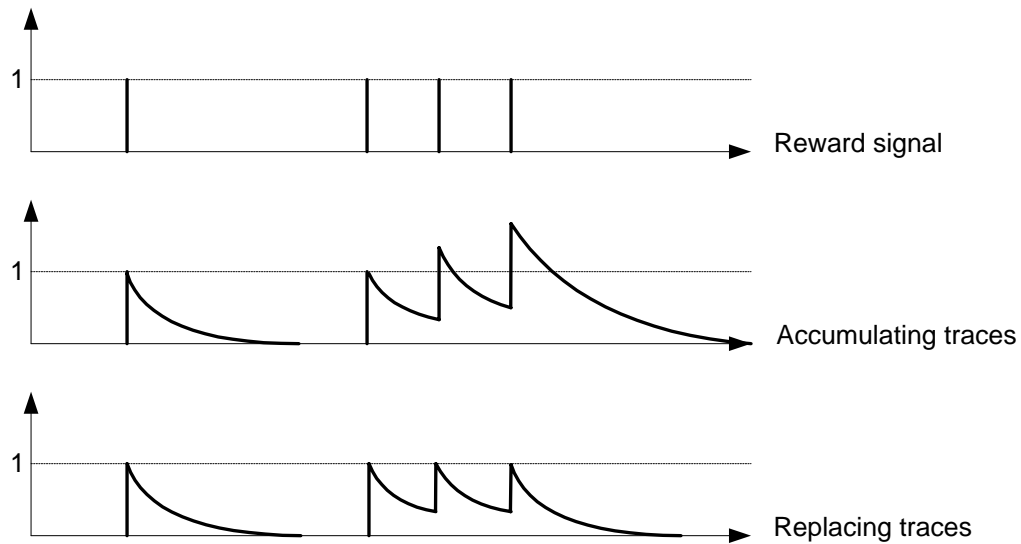
Figure 2.5. Eligibility trace decay diagrams for accumulation and replacing traces.

Taking action a in state s which is expected to lead to state s' having the greatest value according to the current estimation of the value function, is called a *greedy* action. When selecting a greedy action, it is said that the agent is *exploiting*, using its current knowledge to maximize the future reward. On the contrary, when taking a non-greedy action, it is said that the agent is *exploring*, searching the state space in hope to find better paths and to gain higher reward in the long run.

Example   The problem of trade-off between exploration and exploitation can be illustrated by the simple reinforcement learning problem known as *the k-armed bandit*

15

problem. The agent is in an environment consisting of k one-armed bandit gambling machines with h number of free pulls. When playing bandit i, the pay-off is 1 or 0 according to the underlying probability p_i. All payoffs are independent and the probabilities p_i, $i = 1, \ldots, k$, are unknown. To maximize the final payoff, what strategy should the agent have? How long should the agent explore the state space before deciding which machine is the best and only use that one? $\square$

A popular way to control the amount of exploitation and exploration is called *ϵ-greedy*. The method is based on taking greedy actions by default, but with probability ϵ taking random actions according to:

$$a_t = \begin{cases} \arg\max_a Q(s_t, a) & \text{with probability } (1 - \epsilon) \\ \text{rand}(a \in A(s_t)) & \text{with probability } \epsilon \end{cases} \tag{2.24}$$

Some versions of the strategy start out with a high value of ϵ, decreasing it during the learning to approach complete greedy action selection in the end.

This method is straight forward, easy to implement and time efficient with respect to computer calculations. The drawback is its naive exploration, that doesn't use the existing information available in the value function. Another action selection method, called *Softmax*, does utilize the current value of all actions when making a decision. The method scales the probability of choosing an action by the current estimated value. Usually, this is done using a Boltzmann distribution. When being in state s_t with the possible actions a_j, $j = 1, 2, \ldots, m$, the probability of choosing action a_i is:

$$\Pr\{a_i\} = \frac{e^{Q(s_t, a_i)/\tau}}{\sum_{j=1}^{m} e^{Q(s_t, a_j)/\tau}} \tag{2.25}$$

where τ is the so-called *temperature*. The temperature controls to what extent the values of the actions are being considered. As the temperature approaches infinity, the action selection becomes random, without taking the action-values into account:

$$\Pr\{a_i\} = \lim_{\tau \to \infty} \frac{e^{Q(s_t, a_i)/\tau}}{\sum_{j=1}^{m} e^{Q(s_t, a_j)/\tau}} = 1/m \tag{2.26}$$

As the temperature is decreased, the values of the actions are taken more and more into account. Finally, for $\lim_{\tau \to 0}$ actions are selected completely greedy:

$$a_t = \arg\max_a Q(s_t, a) \tag{2.27}$$

As in ϵ-greedy action selection methods, it is common to start out using a high value of τ and decrease it during the learning process. This can be dealt with in many ways. Whether a linear or exponential method is used, it raises the need for an additional parameter that controls the speed of temperature reduction. In the softmax case it is called temperature decrease factor, τ_{df}.

The two parameters, τ and τ_{df} (ϵ and ϵ_{df}), are important meta-parameters in the reinforcement learning framework that highly influence the algorithm's convergence time.

16

2.1.7 Generalization

Previous learning problems considered in this chapter have dealt with state spaces possible to represent as a table. Except for cases where the number of states is few, this means huge memory and computational requirements. If dealing with continuous input signals, the table state space representation is not applicable at all. The issue of generalization is how to use a compact representation of the learning information to transfer knowledge to a larger subset, including states never visited.

The generalization method used is *function approximation.* The aim is to approximate the value function as good as possible based on the received instances of the reward signal. In incremental reinforcement learning tasks, function approximation is a supervised learning problem where fields as neural networks, pattern recognition and statistical curve fitting can be used.

When estimating the value function V_t using function approximation, V_t is not represented as a table but as a parameterized function using the parameter vector $\vec{\theta}_t$. The estimation V_t is totally dependent on $\vec{\theta}_t$. For example, $\vec{\theta}_t$ could be the weights in an artificial network that represents the value function estimation. Most function approximation methods use the mean square error (MSE) as a measurement of the correctness of the approximation. The solution is derived by minimizing this error. In value function approximation, where the target is the true value function V^π using the approximation V_t parameterized by $\vec{\theta}_t$, the MSE is:

$$MSE(\vec{\theta}_t) = \sum_{s \in S} P(s)(V^\pi(s) - V_t(s))^2 \tag{2.28}$$

where P is a distribution weighting the errors in different states. For example, in an incremental RL method, the distribution is determined by the number of times each state is visited.

A group of methods used to minimize the MSE well suited for RL algorithms is the group of *gradient-descent* methods. In these methods, the target function is approached by updating the feature vector $\vec{\theta}_t$ in the direction of the negative gradient of the MSE. Assuming that a true value of V^π is received in each step, the gradient-descent update of $\vec{\theta}_t$ would be:

$$\begin{aligned}
\vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha \nabla_{\vec{\theta}_t}(V^\pi(s_t) - V_t(s_t))^2 \\
&= \vec{\theta}_t - \alpha(V^\pi(s_t) - V_t(s_t))\nabla_{\vec{\theta}_t} V_t(s_t)
\end{aligned} \tag{2.29}$$

Recall that $V_t(s_t)$ is a function completely dependent on $\vec{\theta}_t$, i.e. $V_t(s_t) = f(\vec{\theta}_t)$ for some function f. Therefore, $\nabla_{\vec{\theta}_t} V_t(s_t)$ becomes the gradient of f with respect to $\vec{\theta}_t$ and determines the direction in which the error increases most. By taking steps in the opposite direction, the gradient-descent methods seek to minimize the error. Using sufficiently small value of the step-size parameter α, these approximation methods are guaranteed to converge to a local optimum [32].

The true value of the target function $V^\pi(s_t)$ is not known and has to be estimated. When using the $TD(\lambda)$ updating rule to estimate the error, equation 2.29 can be written as:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \delta_t \vec{e}_t \tag{2.30}$$

where δ_t is the TD error

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{2.31}$$

and $\vec{e}_t$ is a vector of eligibility traces, one for each feature in $\vec{\theta}_t$. This trace vector is update in each step according to:

$$\vec{e}_t = \gamma \lambda \vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t) \tag{2.32}$$

Algorithm *GradientDescentTD(λ)* summarizes how to use gradient-descent TD(λ).

Algorithm *GradientDescentTD(λ)*
1. Initialize $\vec{\theta}$ arbitrarily and $\vec{e} = 0$
2. **repeat** (for each episode)
3. $s \leftarrow$ initial state of episode
4.   **repeat** (for each step of episode)
5. $a \leftarrow$ action given by π for s
6.   Take action a; observe reward, r, and next state, s'
7. $\delta \leftarrow r + \gamma V(s') - V(s)$
8. $\vec{e} \leftarrow \gamma \lambda \vec{e} + \nabla_{\vec{\theta}} V(s)$
9. $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
10. $s \leftarrow s'$
11.   **until** s is terminal

An important point to make is when using the gradient-descent TD updating rule for function approximation in the general case, there is no guarantee that the solution will converge even to a local optimum. However, if the function $f(\vec{\theta}_t)$ that approximates V_t is linear and the step parameter α fulfills necessary restrictions, the gradient-descent TD(λ) algorithm has been proven to converge to the global optimum $\vec{\theta}^*$ [34].

In the linear gradient-descent function approximation case there is a feature vector $\vec{\phi}_s = (\phi_s(1), \phi_s(2), \ldots, \phi_s(n))$ associated to each state having the same number of components as $\vec{\theta}_t$. The value function approximation becomes:

$$V_t(s) = \vec{\theta}_t \vec{\phi}_s = \sum_{i=1}^{n} \theta_t(i) \phi_s(i) \tag{2.33}$$

and the gradient of the value function approximation in the linear case becomes very simple:

$$\nabla_{\vec{\theta}_t} V_t(s_t) = \nabla_{\vec{\theta}_t} \left(\sum_{i=1}^{n} \theta_t(i) \phi_s(i) \right) = \phi_s \tag{2.34}$$

18

The vector $\vec{\phi}_s$ is typically generated from functions. Considering approximation over continuous state spaces, these functions are often so called *radial basis functions* (RBF). They can have different shapes, though the most used function is the Gaussian function:

$$\phi_s(i) = e^{\left(-\frac{||s-c_i||^2}{2\sigma_i^2}\right)} \tag{2.35}$$

where c_i is the center and σ_i is the width parameter (in this case, the standard deviation of the Gaussian function). The number of basis functions and the setting of the two parameters c_i and σ_i are important and have to be chosen appropriate to fit the task and the state space. These parameters can be seen as a part of the set of meta-parameters, controlling the properties of convergence of the algorithm.

One advantage when using radial basis functions, such as the Gaussian function, is that it is possible to linearly approximate smooth continuous value functions. **Figure 2.6** shows Gaussian radial basis functions in the one-dimensional case.
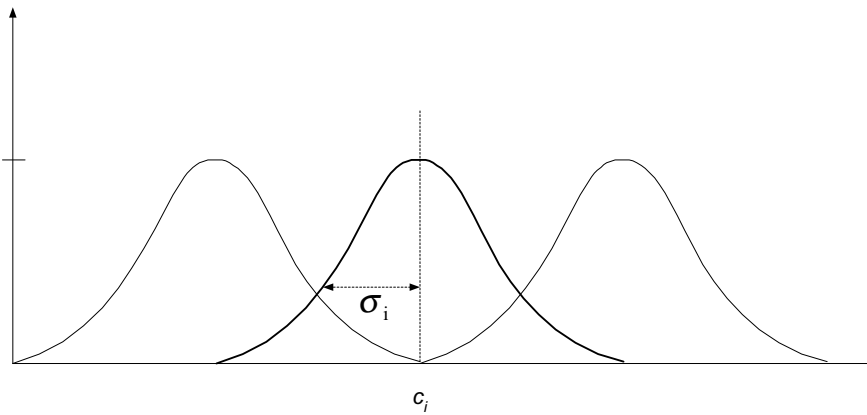
Figure 2.6. The Radial Basis Function (RBF), specifying the position c_i and the width σ_i in the one-dimensional case.

When using Gaussian radial basis functions to approximate the value function as described here, the approximation model is a *radial basis function network*, (RBFN). **Figure 2.7** shows the scheme of this type of network, assuming that the action space is discrete.
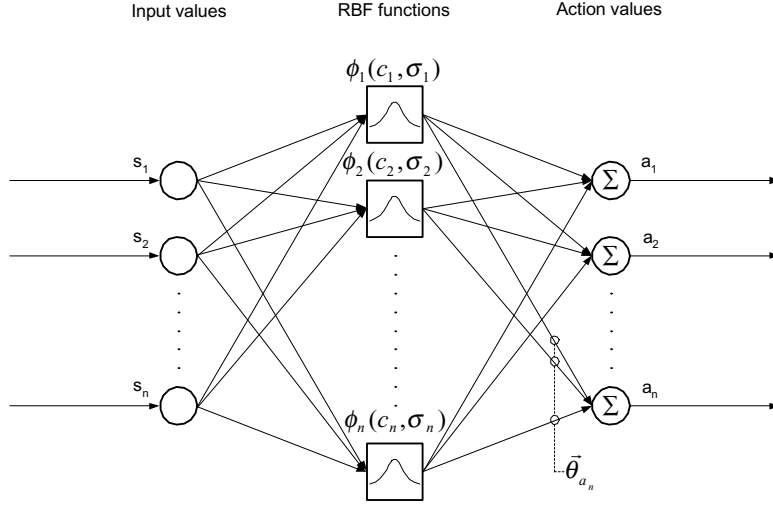
Figure 2.7. Radial basis function network (RBFN).

RL control (handling of the entire learning process) using function approximation is achieved first when action-selection and policy-improvement schemes are included. In this case, it is preferable to approximate the action-value function, $Q_t \approx Q^\pi$, as a parameterized function based on the parameter vector $\vec{\theta}_t$. By redefining equation 2.30 the general gradient-descent update rule is:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \delta_t \vec{e}_t \tag{2.36}$$

where

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}) - Q_t(s_t) \tag{2.37}$$

and

$$\vec{e}_t = \gamma \lambda \vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t) \tag{2.38}$$

and it is called *gradient-descent Sarsa(λ)*. The complete RL control is explained in **Algorithm** *Sarsa(λ)*. The features in the algorithm are Gaussian radial basis functions, $\phi_s(i) = e^{\left(-\frac{||s-c_i||^2}{2\sigma_i^2}\right)}$, and the action selection is based on a softmax function with exponential decreasing temperature.

Algorithm *Sarsa(λ)*
1.   Initialize $\vec{\theta}$ arbitrarily, $\vec{e} = 0$, $\tau_{df} \in [0, 1]$ and τ appropriate
2.   **repeat** (for each episode)
3. $s \leftarrow$ initial state of episode
4.   **for** all $a \in A(s)$
5. $F_a \leftarrow$ set of features in a
6. $Q_a \leftarrow \sum_{i \in F_a} \theta_a(i)\phi(i)$
7. $a \leftarrow a_i$ with probability $\frac{e^{Q(s,a_i)/\tau}}{\sum_{j=1}^{m} e^{Q(s,a_j)/\tau}}$
8.   **repeat** (for each step of episode)
9. $\vec{e} \leftarrow \gamma\lambda\vec{e}$
10.   **for** all $i \in F_a$
11. $e(i) \leftarrow e(i) + 1$ (accumulation traces)
12. $e(i) \leftarrow 1$ (replacing traces)
13.   Take action a; observe reward, r, and next state, s'
14. $\delta \leftarrow r - Q_a$
15.   **for** all $a \in A(s)$
16. $F_a \leftarrow$ set of features in a
17. $Q_a \leftarrow \sum_{i \in F_a} \theta_a(i)\phi(i)$
18. $a' \leftarrow a_i$ with probability $\frac{e^{Q(s,a_i)/\tau}}{\sum_{j=1}^{m} e^{Q(s,a_j)/\tau}}$
19. $\delta \leftarrow \delta + \gamma Q_{a'}$
20. $\vec{\theta}_a \leftarrow \vec{\theta}_a + \alpha\delta\vec{e}$
21. $a \leftarrow a'$
22. $\tau \leftarrow \tau_{df}\tau$
23.   **until** s is terminal

2.2   Genetic Algorithms

Genetic algorithms (GA) [14, 15] are computer-based optimization methods that fall under the category *Evolutionary algorithms* (EA). Common for all EA is that they use basic mechanisms from evolutionary theories as the foundation of their structures. The main feature of these methods is based on principles of natural selection and evolvements by "survival of the fittest", stated by Charles Darwin in *The Origin of Species*. Important methods in addition to GA included in the category of EA are *Genetic programming* (GP) [21] and *Evolutionary Strategies* (ES) [4].

Optimization in general is about finding an instance $\vec{x}^* = (x_1, x_2, \ldots, x_n) \in M$ used in the system under consideration, such that some criterion of interest $f : M \rightarrow R$ is maximized. That is:

$$f(\vec{x}^*) \rightarrow max(f) \tag{2.39}$$

where

$$f(\vec{x}) < f(\vec{x}^*) \tag{2.40}$$

for all $\vec{x} \in M$. In GA, the so called *objective function* f can be given as an analytic expression or from a real-world system of any complexity. The latter representation source motivates the use of GA. GA are robust, general optimization methods that are easy to apply to objective functions that are non-linear, non-differential and noisy with a diffuse representation. However, in problem categories where specific optimization methods have been developed, the GA framework usually performs worse both in terms of speed and accuracy than existing methods.

Examples of areas where GA have been successfully applied are:

- Numerical function approximation, especially in discontinuous, multi-modal and noisy cases [6]

- Combinatorial optimization, including problems as *the traveling salesperson* [12] and *bin packing* [19]

- Machine learning, where the most significant topic is *classifier systems* [10]

This section begins by defining some basic concepts of GA and explaining the general framework. From section 2.2.2 topics important for this report are considered more carefully.

2.2.1 Basic Concepts

The fundamental structure of the GA is based on evolution of populations. A population, consisting of individuals that each one represents a solution hypothesis, is evaluated and recombined to form the next population. As the evolution cycle continues, the hypotheses will eventually become similar and a near optimal solution will be reached.

In early stages of GA research, the individual representation used was so called binary coding, which has a direct translation of the biological model. Lately, steps have been made towards other, in some cases more effective representations. In this introductory section, the binary representation will be emphasized to explain basic terms and dynamics of the GA framework. **Figure 2.8** shows the structure of a GA population. Below follows explanations of some the fundamental terms.

Gene is the functional entity, value space for the parts of the genome. Typically, a gene is representing some parameter x_i in the GA implementation.

Genome specifies the species in biology, it is constructed of all existing genes.

Genotype is the genome of a specific individual.

Individual is the instance of a complete set of parameters that can be applied to the objective function.

Population represent the set of competing individuals.
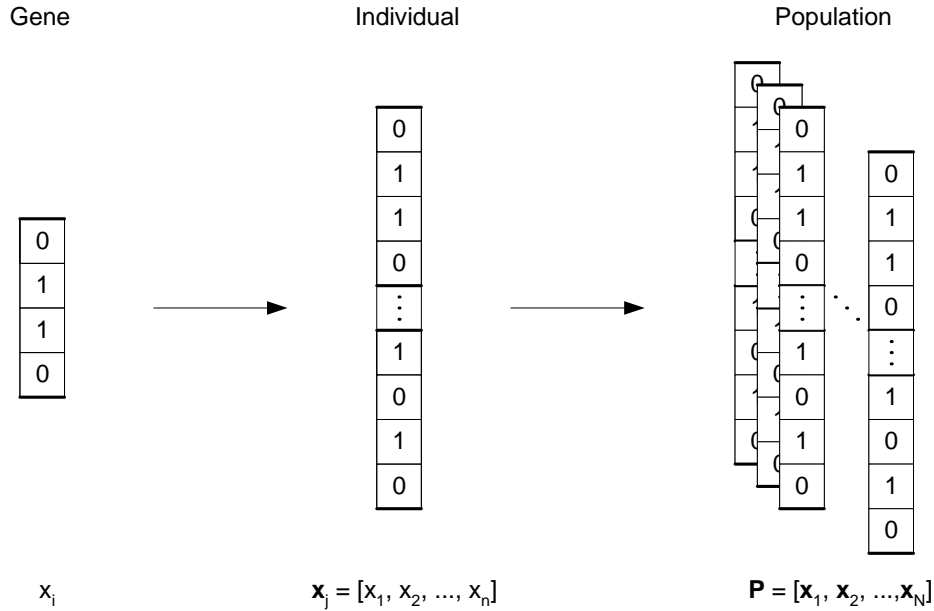
Figure 2.8. The components of a GA population. The smallest component is the gene. One individual consists of several genes and several of individuals create a population. In the GA, a gene codes a parameter in the objective function, an individual is a solution hypothesis and a population is the collection of hypothesis exploring the search space.

Evaluation is the procedure when the information representing the individual is used to execute the objective function.

Fitness is the scaled value representing the outcome of an evaluation.

Selection is the driving force of GA. In the selection, the individuals that will be the parents of the next generation are chosen based on the fitness values.

Genetic operators are used to combine or modify parents to create new individuals.

In the general case, the GA scheme begins by specifying and arbitrary initializing the individuals of an initial population. Each of these individuals are evaluated as input to the objective function, from which a fitness value is calculated and associated with the individual. The best performing individuals are chosen in the selection mechanism to be parents. By applying genetic operators to these individuals the population of successors is created. The genetic operators exchange information between the parents and create individuals that explore the search space. The cycle of evolution proceeds until some criterion of termination is fulfilled. The major components of GA are summarized in **Algorithm** *GeneticAlgorithmOverview*.

Algorithm *GeneticAlgorithmOverview*
1. $t \leftarrow 0$
2. Initialize $P(t)$ arbitrarily
3. **repeat**
4.   Evaluate $P(t)$
5. $P'(t) \leftarrow$ Select $(P(t) \cup Q)$
6. $P''(t) \leftarrow$ Genetic_operators $(P'(t))$
7. $P(t+1) \leftarrow P''(t)$
8. $t \leftarrow t+1$
9. **until** termination

Example The following simple example explains how to use GA to optimize a problem called the *Brachystrochrone* problem, see **Figure 2.9**. The objective is to construct an interpolated track by deciding the heights $x_1, x_2, \ldots, x_n$ to optimize the traveling time from the start point to the end point for a frictionless mass. In this example, the genes specify the parameters x_i and an individual consists of the vector $\vec{x} = x_1, x_2, \ldots, x_n$. Each parameter x_i is coded as a binary string s_i, where for example x_1 would be interpreted as $x_1 = 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 25$. This value can also be scaled to fit the representation of the problem from range $[0, 2^k - 1]$ to range $[a, b]$ according to $x = a + \frac{b-a}{s^k - 1} \sum_{s_i} 2^{k-i}$.

A population P of N strings $S_1, S_2, \ldots, S_N$ is arbitrary initialized, setting each bit of the strings to 0 or 1 randomly. For each individual $k = 1, 2, \ldots, N$, the traveling time $T(S_k)$ and the corresponding fitness value $f(S_k)$ are computed. In this simple case, the probability $p(S_k)$ of selecting individual k is $f(S_k)/\sum_N f(S_n)$ and the set of parents is generated by randomly sample N strings from P according to $p(S_k)$.

Finally, the individuals of the next generation are created by applying genetic operators such as the crossover operator and the mutation operator, explained in section 2.2.4. Next evolution cycle is executed using the new individuals unless some termination criterion is satisfied. Usually, a measure of the similarity of the individuals is defined, and the termination criterion is fulfilled at a specific threshold. □

2.2.2 Coding

The original coding of the individual representation in GA, close connected to genes of biological creatures, is the binary coding scheme used in the example in section 2.2.1. It is argued that this kind of coding has theoretical advantages over coding schemes using symbols with higher cardinality [13]. Moreover, the binary representation offers functionality as effective coding of *if-else* rules for decision problems [25].

However, in the case when the genes of the individuals represent numerical parameters, the choice of binary coding is not obvious. Several empirical studies have
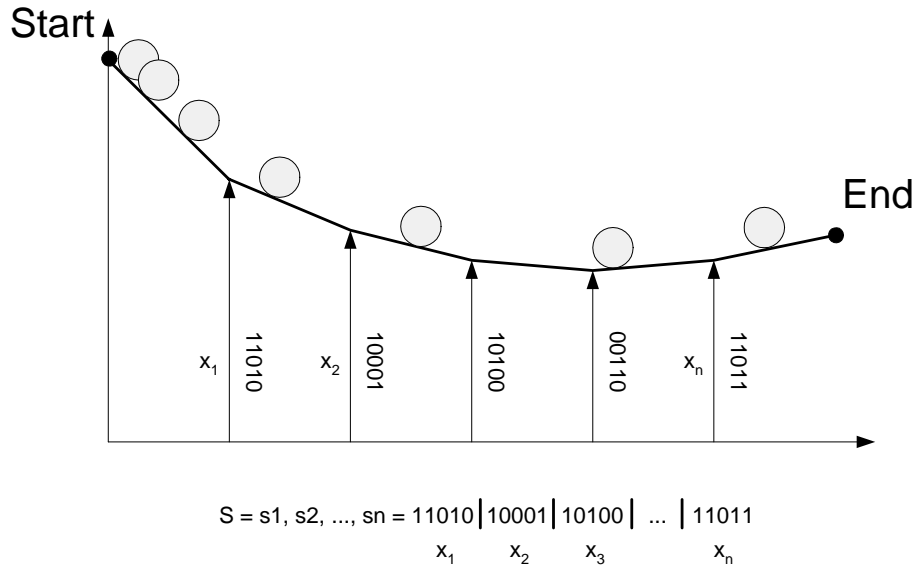
Figure 2.9. The Brachystrochrone problem. The task is to optimize the height values to minimize the traveling time of the friction-less mass from the start point to the end point. The parameters $x_1, x_2, \ldots, x_n$ are coded by binary strings.

argued that integer or floating-point number representations can perform better in these cases [8]. That is, instead of having bit strings as genes, they are coded as integers or floating-point numbers with specified precision, which are manipulated using arithmetic functions. In this case the interpretation of the genetic operators becomes more meaningful, the effect of the operators is more obvious and it is easier to make the operators more problem specific. Studies comparing binary and floating-point representations have shown that floating-point methods give faster, more consistent and more accurate results in problems with numerical parameters [18]

2.2.3   Selection

The selection of individuals to create the successive generation, called parents, is an important part of a GA. The properties of the selection method highly controls the time of convergence and the amount of exploration used in the search for the optimal solution. The selection procedure utilizes the fitness value of the individuals. There are several schemes used including roulette wheel selection, scaling techniques, tournament selection and ranking methods [13, 24].

A common way to select parents is to assign a probability P_i to each individual i based on its fitness value. A series of N random numbers is generated and matched against the cumulative probability, $C_j = \sum_{i=1}^{j} P_i$, of the population. Individual j is selected if $C_{j-1} < U(0, 1) < C_j$.

25

There are several ways to assign the probabilities P_i to the individuals. In the example in section 2.2.1, the roulette wheel probability assignment method is used, defined as:

$$\Pr[\text{chose individual } i] = \frac{F_i}{\sum_{j=1}^{N} F_j} \tag{2.41}$$

where F_i is the fitness of individual i and N is the population size.

A more general group of selection methods, that only requires the objective function and allows both minimization and negativity is the group of ranking methods. They assign probability P_i based on the rank of solution i when all solutions are sorted. For example, the normalized geometric ranking selection assigns P_i for each individual according to:

$$\Pr[\text{chose individual } i] = q'(1-q)^{r-1} \tag{2.42}$$
$$q' = \frac{q}{1-(1-q)^N} \tag{2.43}$$

where q is the probability of selecting the best individual (predefined), r is the rank of the individual (1 is best) and N is the size of the population.

2.2.4   Genetic Operators

As the last step in an evolution cycle in the GA framework, a subset of the selected parents are recombined and mutated to produce the generation of successors. The operators correspond to mating processes in biological evolution. In GA, the genetic operators are used to exchange information between individuals to explore new areas in the parameter search space.

There exist three major classes of genetic operators:

- Crossover, takes two parental individuals, combines them and creates two child individuals.

- Mutation, takes one parental individual, mutates it (changes one or more genes) and creates one child individual.

- Reproduction, replaces the parental individual having the worst fitness value with the best parental individual.

All operator-classes include several versions that create new individuals in slightly different ways. The choice of which to include in a GA implementation is based on the specific problem.

The crossover operator has different definition depending on the choice of individual coding. In binary representation the basic crossover operator is known as *one-point crossover* and in floating-point representation *uniform crossover*. The definition of the basic floating-point crossover has a clear geometric interpretation while it is more diffuse in the binary case. **Figure 2.10** shows the effect of the two crossover operators when applied to individuals with two genes.

Binary

Parent 1 x_1 x_2 | 0 | 1 | 1 | 1 | 0 | 0 |

Parent 2 x_1 x_2 | 1 | 1 | 0 | 1 | 0 | 1 |

Crossover point

Child 1 x_1 x_2 | 1 | 1 | 1 | 1 | 0 | 0 |

Child 2 x_1 x_2 | 0 | 1 | 0 | 1 | 0 | 1 |

Floating point

x_1

Parent 1

Parent 2

x_2

x_1

Child 1

Child 2

x_2
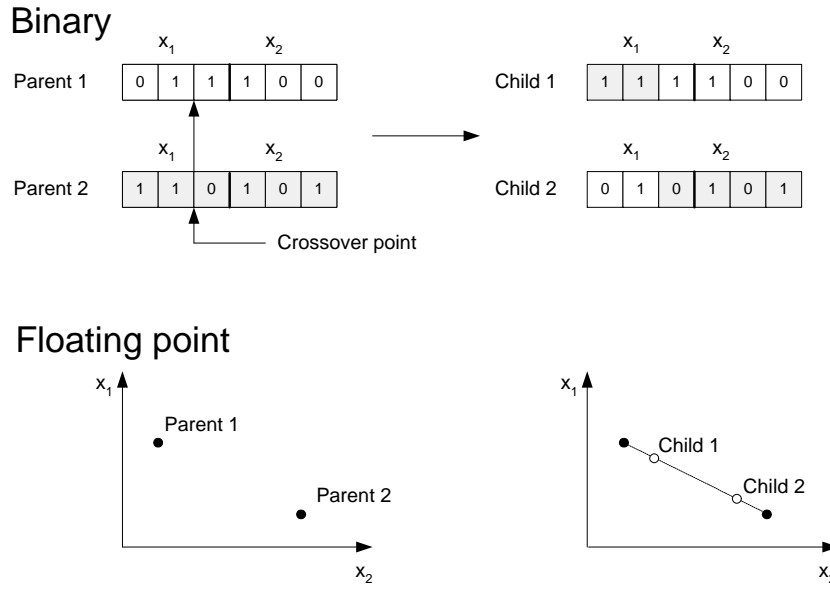
Figure 2.10. The crossover operator in the binary and the floating-point case for individuals with two genes. In the binary case, a crossover point is chosen randomly, where a change of sub-strings are done to create the child individuals. In the floating-point case, the children are created along the interpolation line between the parents according to a uniform distribution.

As in the case of the crossover operator, the definition of the mutation operator differs when using binary and floating-point representations. **Figure 2.11** shows the effect of the *one-bit mutation* operator in the binary case and the *uniform mutation* operator in the floating-point case.

In addition to the standard genetic operators, problem specific operators can sometimes improve the GA performance significantly. For example, when using GA to learn rules in a control task the addition could be rule specializing operators, operating on the rules directly instead of on their representations.

Binary
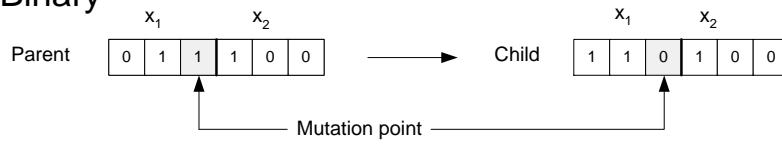
Floating point
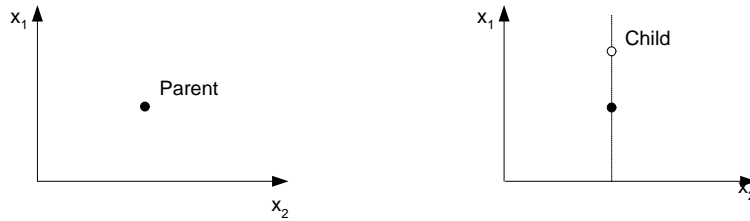
Figure 2.11. The mutation operator in the binary and the floating-point case for individuals with two genes. In the binary case, a random bit is inverted. In the floating-point case, a random parameter is changed according to a uniform distribution.

28

Chapter 3

The Cyber Rodent Robot

The Cyber Rodent (CR) robot is a two wheel driven mobile robot as shown in **Figure 3.1**. It is developed primarily to serve the needs of the Cyber Rodent Project, see section 1.4. The aim is to have a robot that has the same basic constraints as a real rodent animal, with similar physical restrictions. The size, the sensory input, the communication ability and the need of power supply mirror the biological model.

Figure 3.1. CR robot and battery pack.

3.1 Hardware Specifications

The CR robot is 250 mm long and weighs 1.7 kg. It is driven by two wheels positioned at the rear and the front of the robot and it rests on a padding that slides on the ground surface when moving. The engine of the robot is able to give the wheels

a maximum angular velocity of 1.3 m/s in both direction. In addition to normal movement, the CR robot can move in wheelie mode, having the back plate towards the ground.

The CR robot has claws in the front for capturing of portable battery packs. With a magnet that can be activated on will, the robot is able to transport battery packs and recharge.

For communication and interaction with the environment, the CR robot is equipped with:

- Omni-directional CMOS camera.

- IR range sensor.

- Seven IR proximity sensors.

- 3-axis acceleration sensor.

- 2-axis gyro sensor.

- Red, green and blue LED (light-emitting diodes) for visual signaling.

- Audio speaker and two microphones for acoustic communication.

- Infrared port to communicate with a near-by-agent.

- Wireless LAN card and USB port to communicate with the host computer.

The proximity sensors have an accurate range of $70 - 300$ mm. Five are in the front of the robot, one behind and one under the robot pointing downwards. The proximity sensor under the robot is used when the robot moves in wheelie mode. The range sensor is pointed straight forward, used for length measurements in the range of $100 - 800$ mm.

The CR robot has a Hitachi SH-4 cpu with 32 MB memory and an additional FPGA graphic cpu for video capturing and image processing at 30 Hz. Programs can be uploaded through the wireless LAN or the USB port and the CR robots can exchange programs through the IR port.

3.2   Software Specifications

The CR runs an operating system named *eCos* [29] (Embedded Cygnus OS). It is an open source, configurable and portable embedded real time operating system originally developed by Cygnus (now RedHat). The operating system is not a complete Unix variant, although it provides a mostly-complete C [30] library.

The programs running on the CR are written in the C programming language and cross complied on a stationary computer using a modified version of gcc (GNU Compiler Collection) [11].

There is an API (Application Program Interface) to handle robot specific tasks. It provides functions to control:

- Camera settings and picture capturing.

- CR movement.

- Basic multi thread handling.

- Output printing and support.

3.3   The Cyber Rodent Simulator

In addition to the CR robot, a simulator has been developed to enable faster experiments without the same computational restrictions and hardware strains, named the Cyber Rodent Matlab Simulator (CRmS). It is implemented in the Matlab language [22] for easy integration with complex calculations and RL algorithm implementations.

CRmS allows dynamic constructions of environments including walls, boxes and battery packs together with a single CR. The physical engine supports realistic movement and collision response to walls and boxes. Battery packs are not included in the collision framework and are treated as either non-physical objects that can be run over by the CR or be seen as food supplies that disappear in case of contact. The battery packs can be added and moved during execution.

The simulator implements the vision and proximity sensors. The vision sensor can be configured in terms of range of sight in millimeters and field of vision in degrees. Only battery packs can be detected by the vision sensor. The proximity sensors can be used either in realistic mode, giving range values using a noise model similar to real noise in the hardware system, or in true mode giving the correct readings without range limits.

CRmS supports multiple possibilities for visualization. The main screen shows the environment including the possibility to visualize the proximity sensor readings and to print messages. The vision screen displays a flat plane projection of the CR's vision sensor. **Figure 3.2** shows a screen shot of a typical environment setup.

The CRmS API is included in appendix A.

Plotting the readings from distance sensors 1, 3 and 5, robot sensor mode
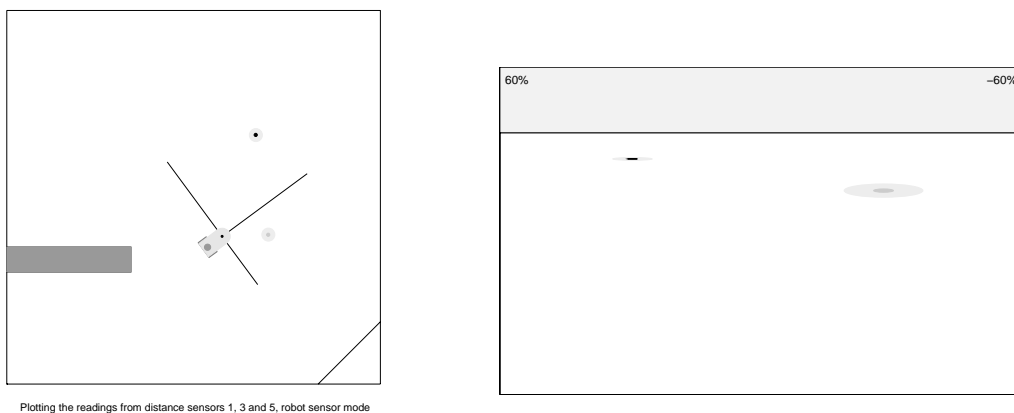
Figure 3.2. View of the Cyber Rodent Matlab Simulator (CRmS), showing the main and the vision screen including visualization of the sensor readings.

Chapter 4

Automatic Decision of Meta-parameters

The method for automatic decision of meta-parameters presented in this study is based on a combination of RL and GA. Using GA to optimize the initial meta-parameter settings gives the possibility to obtain a general, easy-tuned framework for acquiring optimal meta-parameters. These properties make the method interesting as fundamental problems that occur in previous used meta-parameter tuning techniques as human hand tuning and statistical prediction methods are excluded.

Previous studies combining RL and GA have shown promising results in the case of acquiring good meta-parameter settings and effective policy learning in various tasks and environments. Stefano Nolfi and Dario Floreano argues that learning "helps and guides" evolution in the general case of finding an optimal behavior [28]. Given that the fitness surface is flat with a high spike at the optimal Q-value combination, a GA that seeks to find a policy by directly changing Q-values has a small probability of finding a good combination of Q-values. The search is not better than a complete random search algorithm. When learning is involved, the else flat fitness surface becomes smoother, including more information of the position of the optimal solution. From Q-value combinations near the spike in the fitness surface (near optimal Q-values), it is probable that an individual that utilizes learning finds a good Q-value combination at some point during its lifetime and therefore receives some of the fitness of the spike.

In [36], Tatsuo Unemi investigates the interaction between evolution and learning by optimizing initial meta-parameters in a RL algorithm. By studying resulting behaviors in both fixed environments and environments changing from generation to generation, it has been shown that near optimal meta-parameters are possible to find. The RL algorithm used was based on look-up-table Q-learning [32] and all experiments were carried out in simulation.

The method proposed in this study aims to find meta-parameters of a RL algorithm that is able to handle a continuous state space and can be implemented on a real robot. The purpose is to use it as a tool to study dependences between meta-parameters in different environmental setups and to investigate how meta-parameters optimized in simulation can be applied to real robots.

4.1 Proposed Method

Learning and evolution are complementary mechanisms for acquiring adaptive behaviors, within the lifetime of an agent and over generations of agents. There are a number of basic design issues that have to be considered:

1. What to be learned and what to be evolved: the policy, the initial parameters of the policy, the meta-parameters or even the learning algorithm?

2. What type of evolutionary scheme to be used: Lamarcian, where the parameters considered are improved both by evolution and learning, or Darwinian, where the parameters considered by evolution are not effected by learning?

3. How to evaluate the fitness: average lifetime reward, final performance, and/or learning speed?

4. Centralized, synchronous evolution or distributed, asynchronous evolution. The former is standard in simulation where all individuals are evaluated before selection and the procedure is serial. The latter may be more advantageous in hardware implementation.

5. How to combine simulation and hardware experiments?

In the method proposed in this study the individuals of a GA represent the meta-parameters. These can be optimized to get the best agent performance and to minimize the learning time. The combination of learning and evolution is carried out in the following way:

1. Each agent learns by $Sarsa(\lambda)$ algorithm, and the learning rate α and the temperature decrease factor τ_{df} are evolved. The aim is to tune meta-parameters in a real robot learning application that demands a RL algorithm that can handle a continuous state space. $Sarsa(\lambda)$ provides efficient learning control in combination with state space generalization.

2. The policy is reset in every generation, i.e. the Darwinian approach is used. This reflects the evolution process of genes in nature and is also necessary to be able to investigate the mutual relation between meta-parameters.

3. The fitness value is evaluated by the individual performance after a given learning period. Compared to the case where the fitness value is evaluated during the entire learning period, this method results in more distinct fitness values between individuals with different learned policies. It also gives the opportunity to study how meta-parameters are optimized to fit the start of this period of fitness evaluation.

4. The best individuals are selected at the end of each generation in favor to distributed, asynchronous selection. The experiments consist of environmental setups containing a single CR which is best supported by the centralized, synchronous selection.

5. The optimal values of meta-parameters generated by GA are used to learn a capturing behavior in real environments.

An overview of this method is presented in **Figure 4.1**.
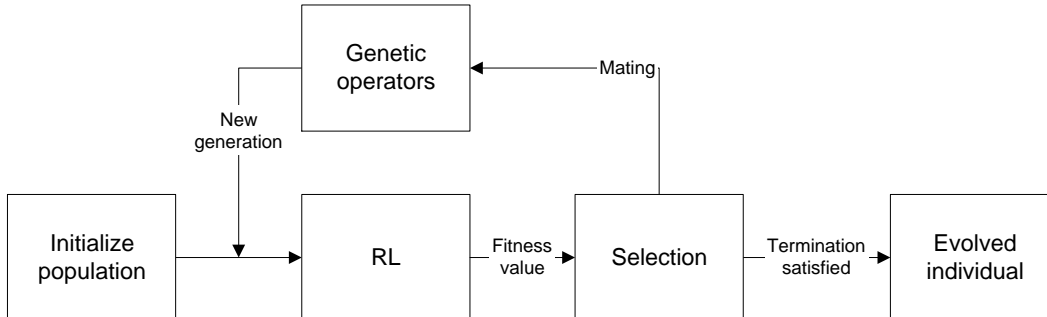
Figure 4.1. Overview of the system for automatic decision of meta-parameters.

4.2   Task and Reward

In order to investigate the method of automatic decision of meta-parameters the RL task considered is capturing of battery packs. A CR has to learn how to approach and capture battery packs marked with LED. With different environmental setups, the aim is to study the performance of the proposed method and properties and dependencies of the meta-parameters themselves.

The task reproduces fundamental learning situations for biological individuals where target position information and basic movements have to be associated. It gives the opportunity to study a RL problem with delayed reward, which increases the importance of the meta-parameter settings. The relative simplicity of the task makes the learning procedure less time consuming, which is of the highest importance, and gives transparent results that are easy to analyze.

The reward signal consists of two parts. The agent receives a reward of 1 when reaching and capturing a battery pack. This is seen as the "true" reward which an individual experiences when finding food. As a compliment, the agent also receives a small reward (up to 2% of the true reward) when a battery pack is in the center of its view. This auxiliary reward is interpreted as if the agent becomes excited when seeing food. It speeds up the learning process and makes it more stable, giving more reliable results. The reward signal is defined as:

$$reward = \begin{cases} 0 & |v| > 0.2\pi \\ \frac{0.1}{\pi}(0.2\pi - |v|) & |v| \le 0.2\pi \\ 1 & \text{reaching battery} \end{cases} \tag{4.1}$$

where v is the angle to the battery pack in radians.

4.3 Reinforcement Learning Model

This section describes the RL implementation as to the task of capturing battery packs. Specifications of the state and action spaces and control structures are followed by the initial settings used in the experiments.

4.3.1 Input and Output Space

The agent receives the continuous value of the angle v to the closest battery pack as state input. Note that the closest battery is chosen by selecting the battery pack with the largest projection size in the camera and the actual distance to the battery pack is never used. The proximity sensory information is only used for an innate behavior to avoid walls. The angle to the battery pack is scaled to the interval $[-0.5, 0.5]$ where negative angle values are to the right in the direction of the agent. The field of view is limited to the interval $[-90, 90]$ degrees. The camera of the CR is omni directional but the field of view is partially blocked by the body of the robot.

When the agent loses sight of the battery pack, the input is the extreme value, -0.5, if the last input was less than zero and 0.5 otherwise. This introduces two hidden states (states where information needed for determining the next correct action is missing [23]) in the RL algorithm.

The action space consists of seven discrete actions, $A = 7$, all for moving. The velocities of the wheels for action number a are calculated as follows:

$$
\begin{aligned}
\omega_{LeftWheel}(a) &= \omega_{Const} * \tfrac{a-1}{A-1}, \\
\omega_{RightWheel}(a) &= \omega_{Const} * \tfrac{A-a)}{A-1}
\end{aligned}
\tag{4.2}
$$

where ω_{Const} is the constant angular velocity. In this way, the action space spans from turning left to turning right, the straightforward action included. The time step of the action selection and the resulting length of movement has a major impact on the RL algorithm convergence properties and the performance and the shape of the optimal policy. Changing the step length changes basic probabilities of the act of capturing battery packs which demands new meta-parameter settings.

4.3.2 Value Function Approximation

To derive a near optimal policy, an estimation of the action-value function $Q(s, a)$ is updated during the learning process. As the state space is continuous, the action-value function has to be approximated. This is realized by a RBFN, see section 2.1.7, using Gaussian basis functions. The action-value in state s taking action a is defined as:

$$
Q(s, a) = \sum_{i=1}^{n} \phi(i, s)\theta(i, a)
\tag{4.3}
$$

where $\phi(i, s)$ is the feature of the Gaussian basis function i in state s defined as:

$$\phi(i, s) = \frac{e^{\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right)}}{\sum_{j=1}^{N} e^{\left(-\frac{||s - c_j||^2}{2\sigma_j^2}\right)}} \tag{4.4}$$

where c_i is the center of the function, σ_i is the width of the function and N is the total number of basis functions in the network. For a schematic view of this kind of network, see **Figure 2.7**.

The network consists of 19 normalized Gaussian basis functions $\phi(i, s)$ scattered uniformly in the one-dimensional input space at the positions $\vec{c} = -0.45, -0.4, \ldots,$ 0.45 with $\sigma_i = 0.05$, $i = 1, 2, \ldots, 19$. Two additional features have been added to handle the situations when the agent loses sight of the battery pack, at position -0.5 and 0.5. The center of the Gaussian basis functions and the position of the added features handling the hidden states are called representative states. **Figure 4.2** shows the basis functions in the input space.
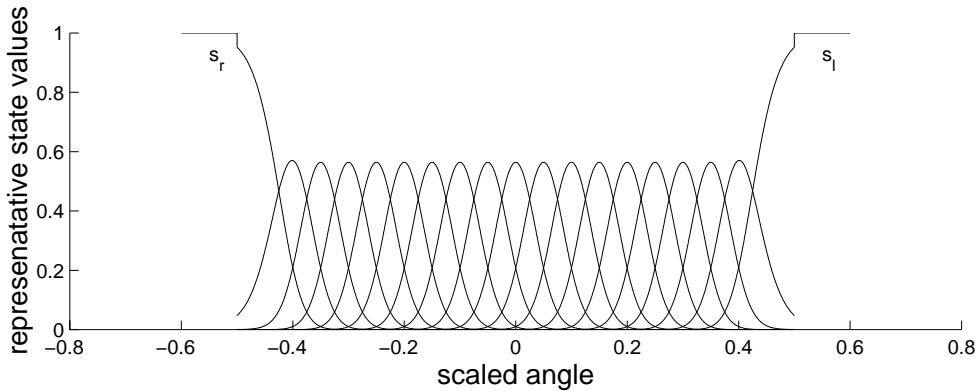
Figure 4.2. Uniformly scattered normalized RBFs in the input space. The two additional states s_l and s_r cover the situation when the agent loses sight of the battery pack.

The positions and widths of the RBFs have been decided after empirical studies of the learning performance in a simple task setup. The number of RBFs and the values c_i and σ_i are important for the resolution of the function approximation, the convergence properties of the RL algorithm and the settings of other meta-parameters.

4.3.3 Learning Control

The weights $\theta(i, a)$ in the RBFN is learned using a control algorithm based on Sarsa(λ), see **Algorithm** *Sarsa(λ)*. It means that the estimation of the action-value function is based on the gradient-descent temporal difference algorithm, see **Algorithm** *GradientDescentTD(λ)*, based on the equations 2.30, 2.31 and 2.32.

There is an eligibility trace associated to each representative state and action, totally $21 * 7$ traces. The traces are implemented as accumulating traces. Considering the small number of actions and the fact that all the representative states are effected when updating the trace of an action, the difference from replacing traces is significant. As the probability of choosing the same action multiple times in a row is high, the associated trace of the action in the accumulating trace case will easily exceed the trace limit of one in the replacing trace case. Moreover, as the function approximation method uses Gaussian basis functions, the entire state space is effected to some extend at each trace update, reinforcing the difference between the methods.

The state-space exploration is controlled by softmax action selection using the Boltzmann distribution. An action a in a state s is chosen with a probability according to:

$$\Pr(a) = \frac{e^{Q(s,a)/\tau}}{\sum_{j=1}^{A} e^{Q(s,a_j)/\tau}} \tag{4.5}$$

where $A = 7$ is the total number of actions. The temperature τ is initially set to 10 and exponentially decreased in each basic step with the temperature decrease factor τ_{df}.

$$\tau := \tau_{df}\tau \tag{4.6}$$

The lifetime of the agents, the environmental setup and the choice of episode or continuous based learning varies in the different experiments and are specified in chapter 5.

4.3.4 Parameter Settings

The initial settings of the RL algorithm are summarized in **Table 4.1**. The learning rate α and the temperature decrease factor τ_{df} are considered for evolution in chapter 5 and are not included here.

Parameter	Name	Value
λ	Trace decay factor	0.2
γ	Discount rate	0.999
A	Number of actions	7
c	RBF position	$[-0.45, -0.4, \ldots, 0.45]$
σ	RBF width	$[0.05]$
$\vec{\theta}_{init}$	Initial RBFN weights	$\in [0, 0.05]$
τ_{init}	Initial temperature	10
δt	Time step	0.075 sec

Table 4.1. Reinforcement learning initial settings.

4.4   Evolutionary Scheme

This section describes the GA implementation used to optimize meta-parameters in the RL. In the first subsection the individual coding, objective function and selection method are explained. The genetic operators used are described in section 4.4.2 and section 4.4.3 summarize the initial settings of the GA.

4.4.1   Individual Coding and Selection

The genes of the individuals of the GA consist of the meta-parameters, $\vec{X} = (\alpha, \tau_{df})$, that are being considered for evolution. Each meta-parameter is coded as a floating-point number with computer precision.

The RL algorithm represents the objective function. The individuals are evaluated by being the input of the RL module and are associated with a fitness value based on the learning performance. Two kinds of experiments are carried out in this study, where the fitness values of all individuals are calculated based on the two following performance measurements respectively:

1. Number of steps used to reach the battery pack when the agent is placed in predefined positions. The measurement is carried out after the learning process.

2. Reward accumulated by the agent during the last part of learning.

In the selection process the normalized geometric ranking method, see equation 2.43, is used to provide each individual with a probability P_i. N parents are selected independently, where each individual can be selected more than one time, based on the probabilities P_i.

4.4.2   Genetic Operators

The genetic operators used to create a new generation of individuals consist of several versions of the crossover, mutation and reproduction operators. Taking experience from previous work by Houck [16] as a starting point, the number of times each operator is executed is derived through trials to fit this the task of this study. **Table 4.2** defines the names of the operators and the number of times they are applied to each generation.

The mutation and crossover operators are implemented as follows. Let $\vec{X} = (x_1, x_2)$ and $\vec{Y} = (y_1, y_2)$ be parental individuals consisting of two floating-point numbers and $[a_i, b_i]$ be the bounds of any parameter x_i or y_i.

The following mutation operators are applied to one parent $\vec{X}$ producing one child $\vec{X}'$.

Type	Name	Number
Mutation	Uniform	5
	Non uniform	4
	Multi non uniform	6
	Boundary mutation	2
Crossover	Simple	2
	Arithmetic	2
	Heuristic	2
Reproduction	Simple	5

Table 4.2. Genetic operators and their parameters.

Uniform mutation operator

$$x'_i = \begin{cases} U(a_i, b_i) & \text{if } i = j \\ x_i & \text{otherwise} \end{cases} \tag{4.7}$$

where j is 1 or 2 randomly and U is a uniform distribution. The operator randomly selects one parameter of $\vec{X}$ and sets it to a random value within the bounds of the parameters, see **Figure 4.3(a)**.

Non-uniform mutation operator

$$x'_i = \begin{cases} x_i + (b_i - x_i)f(G) & \text{if } i = j \text{ and } r < 0.5 \\ x_i - (a_i - x_i)f(G) & \text{if } i = j \text{ and } r \geq 0.5 \\ x_i & \text{otherwise} \end{cases} \tag{4.8}$$

where j is 1 or 2 randomly, $r = U(0, 1)$, G is the current generation and f is a function creating a non-uniform distribution based on G. The operator selects one parameter randomly and sets it to a new value based on its value and the current generation G. The *multi non-uniform* mutation operator applies the non-uniform operator to both parameters in $\vec{X}$, see **Figure 4.3(b)**.

Boundary mutation operator

$$x'_i = \begin{cases} a_i & \text{if } i = j \text{ and } r < 0.5 \\ b_i & \text{if } i = j \text{ and } r \geq 0.5 \\ x_i & \text{otherwise} \end{cases} \tag{4.9}$$

where j is 1 or 2 randomly and $r = U(0, 1)$. The operator randomly selects one parameter and sets it to one of its bound values, see **Figure 4.3(c)**.

The following crossover operators create two children $\vec{X}'$ and $\vec{Y}'$ from two parents $\vec{X}$ and $\vec{Y}$.

Simple crossover operator

$$x_i' = \begin{cases} x_i & \text{if } i < r \\ y_i & \text{otherwise} \end{cases} \tag{4.10}$$

$$y_i' = \begin{cases} y_i & \text{if } i < r \\ x_i & \text{otherwise} \end{cases} \tag{4.11}$$

where r is 1 or 2 randomly. The operator changes parameters between the parents if $r = 2$, see **Figure 4.3(d)**. The

Arithmetic crossover operator

$$\vec{X} = r\vec{X} + (1 - r)\vec{Y} \tag{4.12}$$
$$\vec{Y} = r\vec{Y} + (1 - r)\vec{X} \tag{4.13}$$

where $r = U(0, 1)$. The operator produces children on the interpolation line between the parents, see **Figure 4.3(e)**.

Heuristic crossover operator

$$\vec{X} = \vec{X} + r(\vec{X} - \vec{Y}) \tag{4.14}$$
$$\vec{Y} = \vec{X} \tag{4.15}$$

where $r = U(0, 1)$ and $\vec{X}$ has higher fitness value than $\vec{Y}$. The operator produces one child on the extrapolation from the parents, not necessarily between them, see **Figure 4.3(f)**. Therefore, an inspection of the child $\vec{X}'$ is necessary. If it is created outside the predefined search space, the operator is applied again to the original parents. This procedure is repeated maximum t times to ensure halting.

4.4.3 Parameter Settings

The initial settings of the GA are summarized in **Table 4.3**.

Parameter	Name	Value
$s(P)$	Population size	40
q	Choosing best individual probability	0.08
$[a_\alpha, b_\alpha]$	Search space of α	[0.001, 0.2]
$[a_{\tau_{fd}}, b_{\tau_{fd}}]$	Search space of τ_{df}	[0.9, 1]

Table 4.3. Genetic algorithm initial settings.
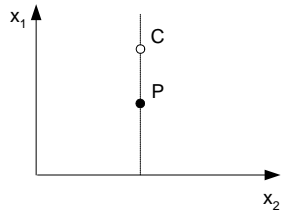
(a) Uniform and non-uniform mutation operator.
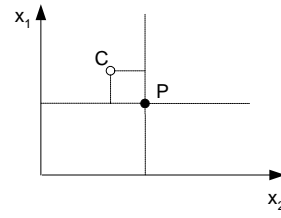
(b) Multi-non-uniform mutation operator.
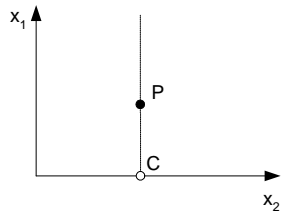
(c) Boundary mutation operator.
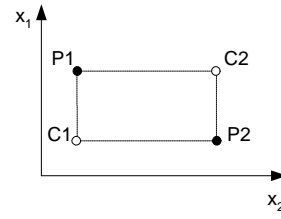
(d) Simple crossover operator.
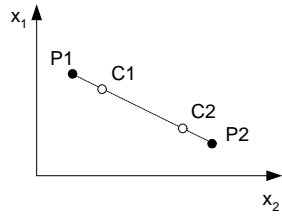
(e) Arithmetic crossover operator.
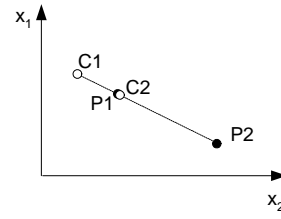
(f) Heuristic crossover operator.

Figure 4.3. Geometric representation of genetic operators.

Chapter 5

Experiments and Results

Most experiments in this study made to verify the proposed method and to investigate the properties of the meta-parameters have been done using the CRmS. Theoretical results are achieved fast and accurately, avoiding wear of expensive hardware equipment. The implementation is straight forward including mathematical functions and matrix manipulation in the Matlab programming language. The angular velocity of the agent, ω_{Const}, is set to 1300 mm/s and the time step used in the simulator is 75 ms. Therefore, each action taken by the agent will result in a movement of 50 mm. All the following time measurements refer to simulation time.

A topic of interest in this study is to investigate how meta-parameters evolved in a simulation environment can be applied to a real robot. Therefore, the RL algorithm has also been implemented in the CR robot.

5.1 Single Food Capturing Task

In the first experiment, the agent is placed in a simple environment containing a single battery pack without any obstacles. The aim of the experiment is to evaluate the proposed method in a restricted and clean environment and to investigate the mutual dependency of the two meta-parameters α and τ_{df}. The learning process is episode based, where the initial position of the agent is systematically varied. In the initial stage of learning the agent starts each episode at a short distance from the battery pack (300 mm). To assure a smooth and complete exploration of the state space (i.e. a uniform distribution P when minimizing MSE function approximation, see section 2.1.7), the angle of the agent to the battery pack is given from a uniform distribution in the interval $[-90, 90]$ degrees. This means that the agent always has the battery in sight at the start of each episode in the initial part of learning. Each episode lasts for 0.525 seconds and the entire initial stage lasts for 75 seconds. In the late stage of learning, the position and orientation of the agent relative to the battery pack are randomly selected and the episode length is increased to 3.75 seconds. The late stage lasts for 100 seconds. The fitness value is calculated when learning is completed based on the performance of

the derived policy. The performance is evaluated by letting the agent capture 18 battery packs from predefined positions. The single food environmently setup and learning conditions are shown in **Figure 5.1**.
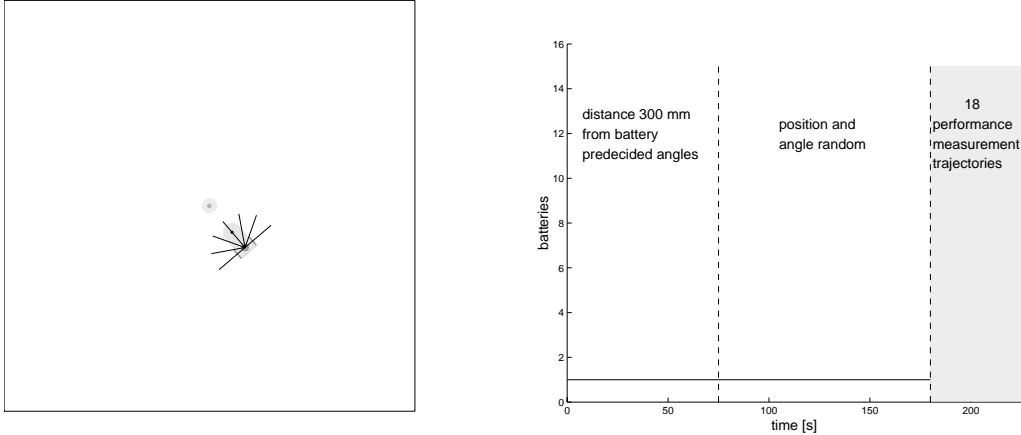
(a) Single food environmental setup. In the first stage of learning, the agent is alone with one battery pack and positioned 300 mm from the battery pack oriented according to the plotted lines.

(b) Single food learning conditions. Position and orientation values are predefined in the first stage of learning and random in the late stage. The fitness value is calculated after the learning.

Figure 5.1. Single food environmental setup and learning conditions.

The meta-parameters under consideration are the learning rate α and the temperature decrease factor τ_{df}. **Figure 5.2** shows the evolved values of the meta-parameters from five simulations. The experiments resulting in the two evolved values to the right in the figure have an additional term in the fitness function that includes the learning time, which explains the difference in the evolved values. The population size is 40 and after 20 to 30 generations the individuals fulfills the termination criterion. The thick line is the MSE fitting of the results. It shows a strong dependence between the meta-parameters. As the algorithm cools off early (small τ_{df}), the step size towards the estimated value function must be large (large α) to be able to make a clear value function decision before the policy becomes greedy. The resulting policy in this case is more often somewhat distorted, making the CR approach the battery packs in a curved path. Compared to the optimal policy, i.e. approaching the battery packs straight, these distorted policies consume more time to reach the battery packs and in some special cases even miss some of them. When the algorithm is allowed to cool off slowly, the step size can be smaller and the estimation of the value function is more correct. The average fitness (connected line) and the standard deviation (dashed lines) of the last populations show the performance of the evolved solutions. The mean fitness value increases as τ_{df}

decreases, until a breaking point is reached. As the lifetime of the agent in the RL is limited, a too high temperature decrease factor implies that the agent won't have cooled off when the learning ends, still exploring sub optimal actions. However, the reliability of the solutions (convergence to similar policies) increases as τ_{df} decreases, due to increased statistical basis.
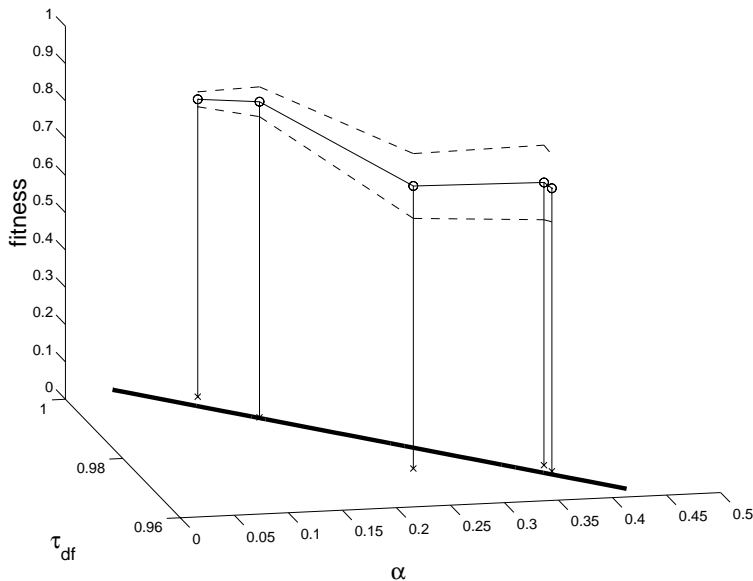
Figure 5.2. Relation between α and τ_{df} in the single food case. The average fitness and fitness standard deviation of the last populations show the performance of the solutions.

5.2 Multiple Food Capturing Task

In the second experiment the learning process is not episode based. The agent is placed in an environment containing multiple battery packs. The battery packs are randomly placed in the environment within predefined boxes to prevent the battery packs from being placed upon or too close to each other. The agent moves in the environment without being repositioned using a built-in behavior to avoid walls. If the agent gets closer than 250 mm to any wall, action 1 or 7 is used and the learning example is not considered. The number of battery packs decreases linearly from 15 to 1 during the agent's lifetime. In the last 98 seconds of the agent's lifetime, the fitness value is calculated directly from the collected reward. In the different experiments, the lifetime of the agents spans from 450 seconds to 1150 seconds. The multiple food environmental setup and learning conditions are shown in **Figure 5.3**.

Figure 5.4 shows results from an experiment where α and τ_{df} have been evolved for individuals with a lifetime of 900 seconds. In this task the agents need more time
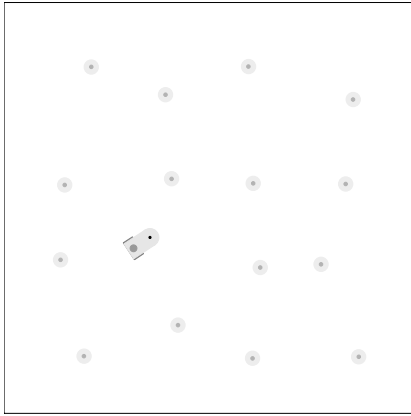
(a) Multiple food environmental setup.
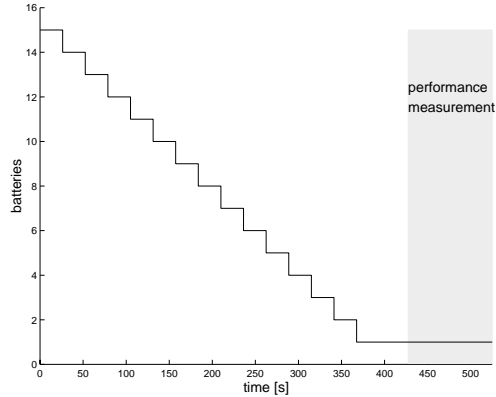
(b) Multiple food learning conditions for the case of lifetime = 525 seconds.

Figure 5.3. Multiple food environmental setup and learning conditions.

to find the correct policy compared to the single food capturing task in section 5.1 due to lower hit rate, increased noise and non-uniform state space feedback. With a lifetime of 900 seconds, the learned policies using the evolved meta-parameters are stable and show good straight approaching behaviors. **Figure 5.4(a)** shows the average reward curve for the initial and the last populations. Also, the temperature τ of the evolved individual and the number of battery packs during the learning are presented. Expected is that the best solution should be to wait as long as possible before cooling off (before τ approaches zero), i.e. reach a near greedy behavior just before the beginning of the fitness measurement. However, the solution suggests that it is better to cool off when there are still eight to ten battery packs left in the environment and the agent has about 375 seconds left until the start of the fitness measurement. This result shows that the learning process in this case is very sensitive at the time of policy decision (when τ approaches zero and the agent uses the learned policy fully). It turns out that searching the environment randomly by making big turns among many battery packs gives better results than almost following the optimal policy among few batteries. If the greedy actions give low hit rate they loose value. That is, to be able to confirm a policy the agent needs reward feedback with a good probability to catch battery packs.

Figure 5.4(b) shows the course of evolution. The connected lines shows the mean value of the two meta-parameters and the dashed lines show the standard deviations for each generation.

The relation between the meta-parameters is not as clear as in the case of the single food experiment in section 5.1 due to the more complicated task which creates a less significant fitness surface. Still, **Figure 5.5** nicely shows the dependencies of

46
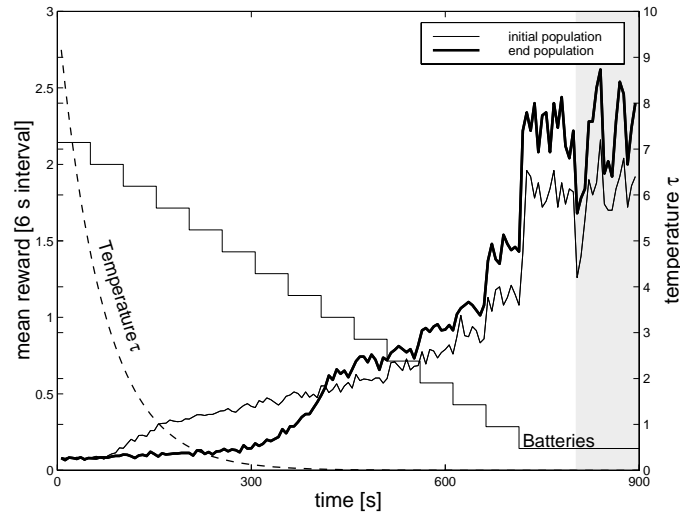
(a) Reward curve for initial and last populations, battery level, fitness measure time interval and temperature curve of the best individual.
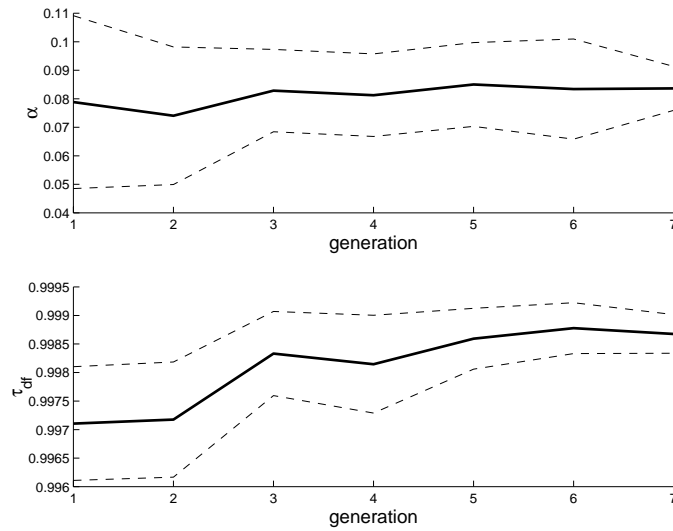
(b) The course of evolution showing the evolvements of mean and standard deviation values for α and τ_{df}.

Figure 5.4. Close up on the first and last population and the evolution of meta-parameters for individuals with a lifetime of 900 s.

the two meta-parameters α and τ_{df} and the environmental settings for eight evolved individuals with different lifetimes.
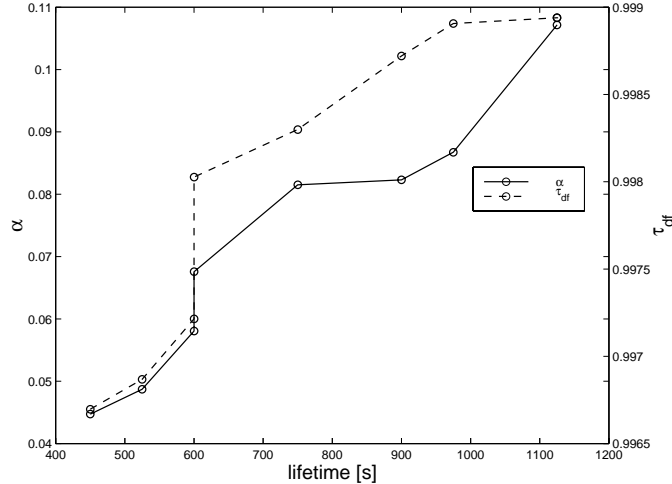
Figure 5.5. Evolved individuals from eight experiments in the multiple food task.

5.3   Hardware Implementation

In order to investigate how meta-parameters optimized in simulation perform in a real hardware system, the RL algorithm is implemented on the CR robot in the third experiment. The environment and the learning conditions are identical to the multiple food task in section 5.2 but carried out in a real world room. The initial values of the experiment are collected from the second simulation in **Figure 5.5**, i.e. the lifetime of the agent is 525 seconds and the meta-parameters $\alpha = 0.049$ and $\tau_{df} = 0.997$.

Due to the difference in computational power and physical conditions between the simulation and hardware platforms, some changes are done in the hardware porting. In the simulator, all calculations are carried out between taking actions. If the movement in the hardware implementation is going to be smooth, the calculations have to be done during movement. As a compensation for calculation delay, the step size has been increased to 500 ms and ω_{Const} has been decreased to 300 mm/s, resulting in the step length of 75 mm. Also, the battery packs simply disappear (move) when the agent reaches them in simulation. As this is not possible to achieve in the real hardware implementation, the agent moves backwards and turns randomly after capturing battery packs. Except for changes in the learning algorithm itself, some slow mathematical functions on the CR are optimized. The trigonometric functions $f(x) = \sin(x)$ and $f(x) = \cos(x)$ and the exponential function $f(x) = e^x$ are approximated by linear interpolated function based on look-up-tables.

Figure 5.6 shows value functions during the learning process in simulation and hardware implementation. Note that action number four is the straightforward action and the edge states, marked as circles, handle the case when no battery packs are visible. **Figures 5.6(d)** and **5.6(e)** show the final learned policies and that

48

there are two major differences between the simulation and the hardware version. First, the straightforward action is used in a wider range of states in the hardware solution. This is related to the hardware CR design. The CR has two "claws" to collect battery packs, see **Figure 3.1**, that makes the CR able to capture battery packs going straight forward even when they are situated slightly to the sides. The second difference is the behavior learned when there are no battery packs visible. In the simulation solution, the CR turns around to search for the battery packs where as in the hardware implementation, the agent uses the straightforward action. The reason is the vision range of the CR. It is shorter on the real robot compared to the simulation agent. The real robot agent relies on the wall avoidance behavior to change direction when reaching walls. If it instead would have used a turning action, it could rotate forever without finding anything.

The course of learning is similar in the two cases. The temperature decreases in the same way and time of decision occurs almost simultaneously. However, the period of decision is more critical in the case of real robot learning. The reasons are noise and time delays not present in the simulation environment.

This experiment shows a successful implementation of the RL algorithm on a real robot using the same meta-parameters as in simulation. However, even in a simple task as the one used in this experiment, the implementation on the real robot has to be modified.
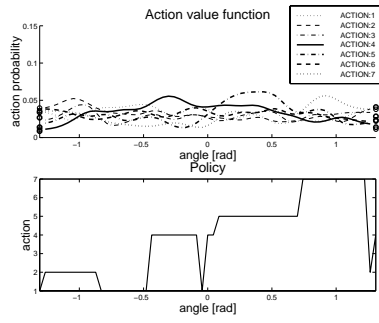
(a) Action-value functions learned after 98 s in simulation.
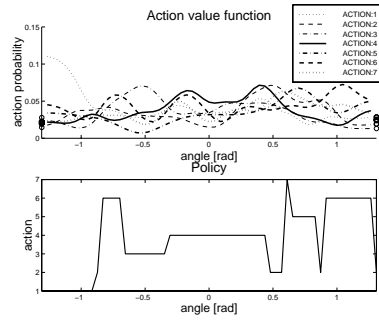
(b) Action-value functions learned after 98 s in hardware implementation.
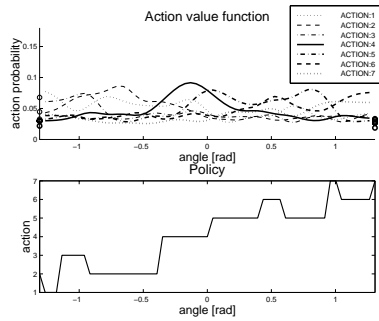
(c) Action-value functions learned after 247 s in simulation.
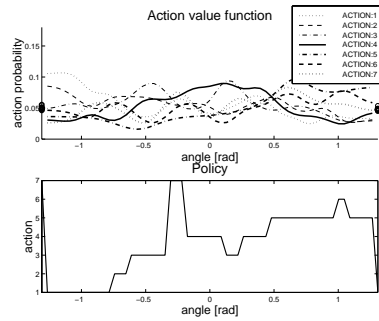
(d) Action-value functions learned after 247 s in hardware implementation.
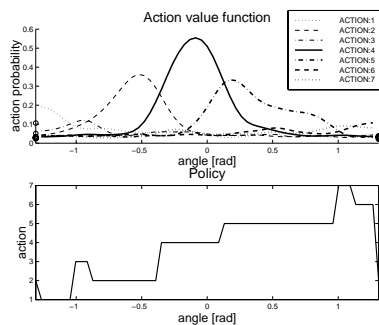
(e) Action-value functions and learned policy (after 525 s) in simulation.
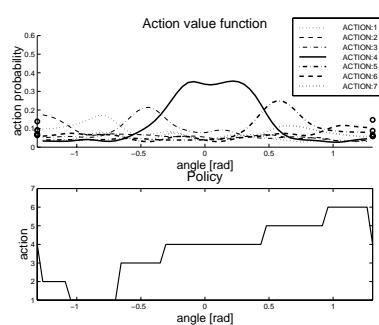
(f) Action-value functions and learned policy (after 525 s) in hardware implementation.

Figure 5.6. Process of learning using the optimized meta-parameters α and τ_{df}, in CRmS and on the CR robot. The associated policies are shown in the second column. All time measurements are given in simulator time. Sub-figures (**e**) and (**f**) show the final evolved results.

Chapter 6

Conclusions

This thesis presented an evolutionary approach to optimize meta-parameters in a RL algorithm. By combining RL and GA, it has been shown that near optimal meta-parameter settings can be found.

The method has been used to show the relation between the meta-parameters learning rate α and temperature decrease factor τ_{df}. The relation is clear in simple environments but deforms and becomes less significant when the environmental conditions change. The results show an illustrative example of how unintuitive meta-parameter settings can occur already when optimizing only two meta-parameters under relatively simple learning conditions. Finally, meta-parameters optimized in simulation has been successfully implemented on the Cyber Rodent, involving some algorithm changes.

The results strengthen the need of an automatic framework for meta-parameter settings in RL to achieve optimal performance and adaptive RL algorithms. The relation between the meta-parameters is difficult to follow and are highly environment dependent. The global optimum of the meta-parameter settings is often difficult to find by logical reasoning, making human hand tuning unwanted.

In future extensions, the aim is to include more meta-parameters to the optimization framework. The meta-parameters of highest interest are the discount rate γ and the trace decay factor λ, but also the initial temperature τ_{init} and radial basis function meta-parameters should eventually be included. An important improvement to consider is ways to accelerate the speed of the current method. At the moment, it suffers from high time consumption and the method is not applicable in complicated RL tasks performed by real robots.

References

[1] Barto A. G., *Reinforcement Learning.* In M. A. Arbib (Ed.), The handbook of Brain Theory and Neural Networks, 804-809, Cambridge, MA, MIT Press, 1995.

[2] Beasley D., Bull D. R. & Martin R. R., An Overview of Genetic Algorithms, Part 1&2. *University Computing*, 15(2), 58-69, 1993.

[3] Bertsekas D. P. & Tsitsiklis J. N., *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[4] Bäck T. & Schwefel H., *Evolutionary Computation: An Overview.* IEEE Press, Piscataway NJ, 1996.

[5] Cristianini N., Evolution and Learning: an Epistemological Perspective. *Evolution and Learning: an Epistemological Perspective*, Axiomathes n.3, 428-437, 1995.

[6] DeJong K., *The Analysis and Behaviour of a Class of Genetic Adaptive Systems.* PhD thesis.

[7] DeJong G. & Spong M. W., Swinging up the Acrobot: An Example of Intelligent Control. *Proceedings of the American Control Conference*, 2158-2162, 1994.

[8] Davis L., *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[9] Doya K., Metalearning and Neuromodulation. *Neural Networks*, 15(4/5), 2002.

[10] Forest S. & Mayer-Kress G., Genetic Algorithms, Nonlinear Dynamic Systems, and Models of International Security. In Davis L., editor, *Handbook of Genetic Algorithms*, 124-132, Morgan Kaufmann, 1989.

[11] GNU Project, *http://gcc.gnu.org/*

[12] Goldberg D. E., Alleles, loci, and TSP. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, 154-159, Lawrence Erlbaum Associates, 1985.

[13] Goldberg D. E., *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[14] Holland J. H., Outline for a Logical Theory of Adaptive Systems. *Journal of the Association for Computing Machinery*, 3, 297-314, 1962.

[15] Holland J. H., *Adaptaion in natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, MI, 1975.

[16] Houck C. R., Joines J. A. &Kay M. G., *A Genetic Algorithm for Function Optimization: A Matlab Implementation.* NCSU-IE TR 95-09, 1995.

[17] Iskii S., Yoshida W. & Yoshimoto J., Control of Exploitation-exploration Meta-parameter in Reinforcement Learning. *Neural Networks*, 15(4/5), 2002.

[18] Janiokow C. Z. & Michalewicz Z., An Exprimental Comparision of Binary and Floating Point Representions in Genetic Algorithms. *Proceedings of the Fourth International Conference on Genetic Algorithms*, 31-36, Morgan Kaufmann, 1991.

[19] Juliff K., Using a Multi Chromosome Genetic Algorithm to Pack a Truck. Technical Report RMIT CS TR 92-2, Royal Melbourne Institute of Technology, 1992.

[20] Kaelbling L. P., Littman M. L. & Moore A. W., Reinforcemnet Learning: A Survey. *Journal of Artificial Intelligence Research 4*, 237-285, 1996.

[21] Koza John R., *Genetic Programming.* MIT Press, 1992.

[22] MathWorks Inc, The, *http://www.mathworks.com.* 1994-2003.

[23] McCallum A. K. & Andrew R., Hidden State and Reinforcement Learning with Instance-Based State Identification. *IEEE Transations on Systems, Man and Cybernetics (Special issue on Robot Learning)*, 1996.

[24] Michalewich Z., *Genetic Algorithms + Data Structures = Evaluation Programs.* Springer Verlag, 1994.

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

[26] Morimoto J. & Kenji D., Aquisition of Stand-up Behaviour by a Real Robot Using Hierarchical Reinforcement Learning. *Robotics and Autonomous Systems*, 36, 37-51, 2001.

[27] Neal R. M., *Bayesian Learning for Neural Networks.* Springer Verlag, 1996.

[28] Nolfi S. and Floreano D., Learning and Evolution. *Autonomous Robots*, 7(1): 89-113, 1999.

[29] Red Hat Inc, *http://sources.redhat.com/ecos/*

[30] Ritchie D. R., The Development of the C Language. *Second History of Programming Languages conference*, Cambridge, Mass., April, 1993.

[31] Singh S. & Bertsekas D., Reinforcement Learning for Dynamic Channel Allocation in Cellular Telephone Systems. In M. C. Mozer, M. I. Jordan & T. Petsche (Eds.), *Advances in Neural Information Processing Systems 9*, 974-980, Cambridge, MA, USA, MIT Press, 1997.

[32] Sutton R. S. & Barto A. G., *Reinforcement Learning: An Introduction.* Cambridge, MA, MIT Press, 1998.

[33] Tesauro G. J., TD-gammon, a Self-teaching Backgammon Program, Achieves Master-level Play. *Neural Computation*, 6(2), 215-219, 1994.

[34] Tsitsiklis J. N. & Van Roy B., An Analysis of Aemporal-difference Learning with Function Approximation. *IEEE Transactions on Automatic Control*, 1997.

[35] Vapnic V. N., *The Nature of Statistical Learnign Theory.* Springer Verlag, 2 edition, 2000.

[36] Unemi T., Nagayoshi M., Hirayama N., Nade T., Yano K. & Masujima Y., Evolutionary Differentiation of Learning Abilities - A Case Study on Optimizing Parameter Values in Q-learning by Genetic Algorithm. *Artificial Life IV*, 331-336, MIT Press, 1994.

Appendix A

CRmS API

```
********************************************************************************
* CRmS API
********************************************************************************
This is the user API for the Cyber Rodent matlab Simulator (CRmS).

Simulator info:
This is a 2-dimensional simulator for the Cyber Rodent robot. It allows
one Cyber Rodent equipped with six approx sensors and one visual sensor.
The environment can be build using walls, boxes and batteries. The
coordinate system of the environment has a horizontal x-axle and a
vertical y-axle. Zero radians is defined to be in the positive x-axle
direction.

When using the simulator notice that the first return value is always
an error flag, i.e. non-zero if an error occured during call to function.

API syntax:
[return values] = function_name(arguments)
Description
arg:    argument descriptions
return: return description

API argument syntax:
[argument]           -> argument is optional
argument(dimension) -> argument is a matrix (: means variable dimension)
```

```
***************************
Function declareations
***************************
INIT FUNCTION
[err] = sim_cr()

GET FUNCTIONS
[err, length x, lenth y]                                = sim_cr_get_dim()
[err, position x, position y, angle]                    = sim_cr_get_pos()
[err, lenght, width, body length, head length, wheel radius]
                    = sim_cr_get_cr_measures()
[err, velocity left wheel, velocity right wheel]        = sim_cr_get_vel()
[err, color(1, 3)]                                      = sim_cr_get_led()
[err, angles(:), distances(:), colors(:), length]      = sim_cr_get_vision()
[err, angle noise, distance noise]
                    = sim_cr_get_vision_noise()
[err, angle range, distance range]
                    = sim_cr_get_vision_range()
[err, distance(:)]
                    = sim_cr_get_sensor([sensors(:)])
[err, distance noise]
                    = sim_cr_get_sensor_noise()
[err, range(1, 2)]
                    = sim_cr_get_sensor_range()
[err, time, steps]                                      = sim_cr_get_time()


SET FUNCTIONS
[err] = sim_cr_set_dim(length x, length y, [reward], [color(1, 3)])
[err] = sim_cr_set_cr(position x, position y, rotation angle)
[err] = sim_cr_set_vel(velocity left wheel, velocity right wheel)
[err] = sim_cr_set_led([color(1, 3)])
[err] = sim_cr_set_vision_noise(angle noise, [distance noise])
[err] = sim_cr_set_vision_range(angle range, [distance range])
[err] = sim_cr_set_sensor_noise(distance noise)
[err] = sim_cr_set_sensor_range(range(1, 2))
[err] = sim_cr_set_walls_reward(reward)
[err] = sim_cr_set_batteries_reward(reward, [color(1, 3)])
[err] = sim_cr_set_step_reward(reward)
[err] = sim_cr_set_bat_reach_mode(mode)
[err] = sim_cr_set_speed(speed)
[err] = sim_cr_set_mesg(string, [window])
[err] = sim_cr_set_bg_color(color(1, 3))
[err] = sim_cr_set_cr_color(body color(1, 3), [head color(1, 3)],
```

```
                        [wheel color(1, 3)], [led color(1, 3)], [camera color(1, 3)])

   ADD FUNCTIONS
   [err]                  = sim_cr_add_wall(first point x, first point y,
                            second point x, second point y, [reward], [color(1, 3)])
   [err]                  = sim_cr_add_box(middle point x , middle point y,
                            length x, length y, rotation angle, [reward], [color(1, 3)])
   [err, battery number] = sim_cr_add_battery(middle point x, middle point y,
                            [reward], [led color(1, 3)], [radius], [outer color(1, 3)])


   SHOW/HIDE FUNCTIONS
   [err] = sim_cr_show_sensor_dist(mode, [color(1, 3)])
   [err] = sim_cr_hide_sensor_dist()

   UPDATE FUNCTIONS
   [err]                                      = sim_cr_draw()
   [err]                                      = sim_cr_draw_vision()
   [err, reward, object type, object color(1, 3)] = sim_cr_move()
   [err]                                      = sim_cr_move_battery(position x,
                          position y, battery handle)




   ***************************
   * Function descriptions
   ***************************
   INIT FUNCTION

   ------------------------------------------------------------
   [err] = SIM_CR()
   Initializes the simulator object and sets default values. This function
   must be called before calling any other simulator functions.
   arg:   none
   return: error flag, zero if no error occured during call to function,
           non-zero othervise




   GET FUNCTIONS

   ------------------------------------------------------------
   [err, length x, lenth y] = SIM_CR_GET_DIM()
   Gets the dimention of the play ground.
```

```
arg:      none
return: error flag         zero if no error occured during call to
                             function, non-zero othervise
length x          length of the playground along the x axle
length y          length of the playground along the y axle
---------------------------------------------------------
[err, position x, position y, angle] = SIM_CR_GET_POS()
Gets the position and angle of the Cyber Rodent.
arg:      none
return: error flag        zero if no error occured during call to function,
                            non-zero othervise
position x        Cyber Rodents x coordinate
position y        Cyber Rodents y coordinate
angle             Cyber Rodents angle, [-pi pi], 0 along the x axle
---------------------------------------------------------
[err, lenght, width, body length, head length, wheel radius] =
SIM_CR_GET_CR_MEASURES()
Gets the measures of the Cyber Rodents parts
arg:      none
return: error flag         zero if no error occured during call to function,
                            non-zero othervise
lenght            Cyber Rodents total length
width             Cyber Rodents width
body length
head length
wheel radius

---------------------------------------------------------
[err, velocity left wheel, velocity right wheel] = SIM_CR_GET_VEL()
Gets the velocities of the wheels.
arg:      none
return: error flag zero if no error occured during call to function,
        non-zero othervise
velocity left wheel
velocity right wheel

---------------------------------------------------------
[err, color(1, 3)] = SIM_CR_GET_LED()
Gets the current color of the led ontop of the Cyber Rodent.
arg:      none
return: error flag         zero if no error occured during call to function,
                            non-zero othervise
color             current color of the led, [r g b],  0<= r,g,b <=1
---------------------------------------------------------
[err, angles(:), distances(:), colors(:), length] = SIM_CR_GET_VISION()
Gets angles, distances and colors of all batteries visable to the Cyber Rodent.
```

All values are calculated with current angle and distance noise rates.

```
arg:    none
return: error flag      zero if no error occured during call to function,
                        non-zero othervise
angles          a vector containing angles to all visable batteries
distances       a vector containing distances to all visable batteries
colors          a vector containing color vectors of all visable batteries,
                        [r g b],  0<= r,g,b <=1
length          number of visable batteries, equal to the length of
                        above vectors
------------------------------------------------------------
[err, angle noise, distance noise] = SIM_CR_GET_VISION_NOISE()
Gets the noise for the angle and distance values obtained using the vision sensor.
arg:    none
return: error flag      zero if no error occured during call to function,
                        non-zero othervise
angle noise     noise rate for the angles to the batteries
distance noise  noise rate for the distances to the batteries
------------------------------------------------------------
[err, angle range, distance range] = SIM_CR_GET_VISION_RANGE()
Gets how many radians form zero (straight ahead) the Cyber Rodent can detect
batteries and the maximum distance the Cyber Rodent can detect a battery pack.
arg:    none
return: error flag      zero if no error occured during call to function,
                        non-zero othervise
angle range     radians from zero (straight ahead) the vision sensor can
                        detect batteries
distance range  maximum distance the vision sensor can detect a battery
                        pack.
------------------------------------------------------------
[err, distance(:)] = SIM_CR_GET_SENSOR(sensors(:))
Gets approx. sensor distance readings from walls and boxes, not from batteries.
All values within approx. sensor range are calculated with current approx sensor
noise. Values under lower range gets the value of lower range and values above
upper range gets random value between upper range and upper range*2.
Only values from sensors defined by the vector sensor(:) are returned. This
function must be called to update sensor values if sim_cr_show_sensor_dist()
has been called to plot sensor readings using sim_cr_draw(). Only the values
from the sensors defined by sensors(:) will be plotted. This function can be
called with no arguments to initialize the look-up-table
arg:    [sensors]       vector of size (1,:) containing approx. sensor number
                        to get readings from. Front right sensor-front left
                        sensor has numbers 1-5. Rear sensors number is 6
return: error flag      zero if no error occured during call to function,
```

```
                        non-zero othervise
distances(:)      vector of size (1,:) containing the distance readings
                        from approx. sensors given by sensors(:)
------------------------------------------------------------
[err, distance noise] = SIM_CR_GET_SENSOR_NOISE()
Gets the noise for the distance values obtained using the approx. sensors.
arg:    none
return: error flag      zero if no error occured during call to function,
                        non-zero othervise
distance noise   noise rate for the distances to walls and boxes in the
                        environment
------------------------------------------------------------
[err, range(1, 2)] = SIM_CR_GET_SENSOR_RANGE()
Gets the range of the approx sensors
arg:    none
return: error flag      zero if no error occured during call to function,
                        non-zero othervise
range            vector of size (1, 2). range(1) is lower value able to
                        be measured and range(2) is upper value  able to be
                        measured and
------------------------------------------------------------
[err, time, steps] = SIM_CR_GET_TIME()
Gets the estimated simulator time since first sim_cr_move()-call and number of steps
since first sim_cr_move()-call.
arg:    none
return: error flag      zero if no error occured during call to function,
                        non-zero othervise
time             estimated simulator time since first step
steps            total number of steps taken




SET FUNCTIONS

------------------------------------------------------------
[err] = SIM_CR_SET_DIM(length x, length y, [reward], [color(1, 3)])
Sets the dimensions of the playground. This functuion must be called before
adding any components to the environment.
arg:    length x        playground length along x axle
length y        playground length along y axle
[reward]        reward gained when hitting the outer walls of the
                        environment, default 0
[color]         color of the walls, [r g b],  0<= r,g,b <=1.
                        default [0 0 0].
```

```
return: error flag      zero if no error occured during call to function,
                        non-zero othervise

----------------------------------------------------------
[err] = SIM_CR_SET_CR(position x, position y, rotation angle)
Sets the Cyber Rodent in the environment. Only one Cyber Roend can be set.
This must be done before calling sim_cr_draw, sim_cr_draw_vision or sim_cr_move.
arg:    position x      the x coordinate of the Cyber Rodents position
position y      the y coordinate of the Cyber Rodents position
rotation angle   the initial angle of the Cyber Rodent
return: error flag      zero if no error occured during call to function,
                        non-zero othervise

----------------------------------------------------------
[err] = SIM_CR_SET_VEL(velocity left wheel, velocity right wheel)
Sets the velocity of the wheels in mm/sec limited by abs(1300)
arg:    velocity left wheel
velocity right wheel
return: error flag      zero if no error occured during call to function,
                        non-zero othervise

----------------------------------------------------------
[err] = SIM_CR_SET_LED([color(1, 3)])
Sets the color of the led ontop of the Cyber Rodent. If color(1, 3) ids not
defined the led is turned off.
arg:    [color]         color to give the led, [r g b],  0<= r,g,b <=1
return: error flag      zero if no error occured during call to function,
                        non-zero othervise

----------------------------------------------------------
[err] = SIM_CR_SET_VISION_NOISE(angle noise, [distance noise])
Sets the nosie of angles and distances to batteries obtained using the vision
sensor.
arg:    angle noise     default 0.
[distance noise] default 0.
return: error flag      zero if no error occured during call to function,
                        non-zero othervise

----------------------------------------------------------
[err] = SIM_CR_SET_VISION_RANGE(angle range, [distance range])
Sets how many radians form zero (straight ahead) the Cyber Rodent can detect
batteries and the maximum distance the Cyber Rodent can detect a battery pack.
arg:    angle range     range in radians from 0 (straight ahead) for vision.
                        default pi/4
[distance range] maximum distance in mm the Cyber Rodent can detect a
                        battery pack. default is 3000.
return: error flag      zero if no error occured during call to function,
                        non-zero othervise

----------------------------------------------------------
```

```
[err] = SIM_CR_SET_SENSOR_NOISE(distance noise)
Sets the noise for the distance values obtained using the approx. sensors.
arg:    distance noise   noise rate for the distances to walls and boxes in the
                         environment. default is 0.
return: error flag       zero if no error occured during call to function,
                         non-zero othervise
------------------------------------------------------------
[err] = SIM_CR_SET_SENSOR_RANGE(range(1, 2))
Sets the range of the approx sensors.
arg:    range            vector of size (1, 2). range(1) is lower value able to
                         be measured and range(2) is upper value  able to be
                         measured and. default is [70, 350]
return: error flag       zero if no error occured during call to function,
                         non-zero othervise
------------------------------------------------------------
[err] = SIM_CR_SET_WALLS_REWARD(reward)
Sets the reward of all currently added walls in the environment.
arg:    reward
return: error flag       zero if no error occured during call to function,
                         non-zero othervise
------------------------------------------------------------
[err] = SIM_CR_SET_BATTERIES_REWARD(reward, [color(1, 3)])
Sets the reward of all currently added batteries in the environment. If
color(1, 3) is defined only the batteries with this color will have the reward.
arg:    reward
return: error flag       zero if no error occured during call to function,
                         non-zero othervise
------------------------------------------------------------
[err] = SIM_CR_SET_STEP_REWARD(reward)
Sets the reward given from sim_cr_move() for taking a step.
arg:    reward
return: error flag       zero if no error occured during call to function,
                         non-zero othervise
------------------------------------------------------------
[err] = SIM_CR_SET_BAT_REACH_MODE(mode)
Sets the battery reach mode. In mode = 1 the batteries will disappear when
reaching them. In mode = 2 the batteries will stay when reaching them.
arg:    mode             value 1 or 2. default is 1
return: error flag       zero if no error occured during call to function,
                         non-zero othervise
------------------------------------------------------------
[err] = SIM_CR_SET_SPEED(speed)
Sets how many times normal speed the simulator should run in. Speed = 1 gives
normal (real) Cyber Rodent speed. OBS! Note that increasing the speed increases
```

the approximation in the simulator.

```
arg:     speed              how many times normal speed the simulator should run in.
                            range is [0, 20] default is 1.
return: error flag         zero if no error occured during call to function,
                            non-zero othervise
```

[err] = SIM_CR_SET_MESG(string, [window])
Sets a message in the window specified by window. Sim_cr_draw has to be called
before calling sim_cr_set_mesg.

```
arg:     string             The string to be printed
[window]          Specifies the vindow to print the message in. window=1
                            is main window (upper view), window=2 is vision window
                            (vision view). default is 1
return: error flag         zero if no error occured during call to function,
                            non-zero othervise
```

[err] = SIM_CR_SET_BG_COLOR(color)
Sets the background color of the playground

```
arg:     color              background color, [r g b],  0<= r,g,b <=1
return: error flag         zero if no error occured during call to function,
                            non-zero othervise
```

[err] = SIM_CR_SET_CR_COLOR(body color(1, 3), [head color(1, 3)],
[wheel color(1, 3)], [led color(1, 3)], [camera color(1, 3)], [approx color(1, 3)])
Sets the color of the parts of the Cyber Rodent.
Sets the background color of the playground

```
arg:     body color         color of the main body, [r g b],  0<= r,g,b <=1
[headcolor]        color of the head, [r g b],  0<= r,g,b <=1
[wheel color]     color of the wheels, [r g b],  0<= r,g,b <=1
[led color]       default color of the led when turned off, [r g b],
                            0<= r,g,b <=1
[camera color]    camera color, [r g b],  0<= r,g,b <=1
[approx color]    approx. sensors color, [r g b],  0<= r,g,b <=1
return: error flag         zero if no error occured during call to function,
                            non-zero othervise
```

ADD FUNCTIONS

[err] = SIM_CR_ADD_WALL(first point x, first point y, second point x,
second point y, [reward], [color(1, 3)])
Adds a wall to the environment.

```
arg:     first point x     the x coordinate of wall point 1
```

```
first point y      the y coordinate of wall point 1
second point x     the x coordinate of wall point 2
second point y     the y coordinate of wall point 2
[reward]           reward given when hitting the wall. default is 0.
[color]            color of the wall, [r g b],  0<= r,g,b <=1. default is
                              [0 0 0].
return: error flag       zero if no error occured during call to function,
                         non-zero othervise


------------------------------------------------------------
[err] = SIM_CR_ADD_BOX(middle point x , middle point y, length x, length y,
[rotation angle], [reward], [color(1, 3)])
Adds a box to the environment. Also adds four walls to make ciollition detection.
arg:    middle point x   the x coordinate of the boxs position
middle point y   the y coordinate of the boxs position
length x         the length along the x axle before rotating the box
length y         the length along the y axle before rotating the box
[rotation angle] how many radians the box should be rotatted. default is 0.
[reward]         reward given when hitting the box. default is 0.
[color]          color of the box, [r g b],  0<= r,g,b <=1. default is
                              [0 0 0].
return: error flag       zero if no error occured during call to function,
                         non-zero othervise
------------------------------------------------------------
[err, battery number] = SIM_CR_ADD_BATTERY(middle point x, middle point y,
[reward], [led color(1, 3)], [radius], [outer color(1, 3)])
Adds a battery to the environment and returns a handle to the battery (integer)
which can be used for moving the battery.
arg:    middle point x   the x coordinate of the batterys position
middle point y   the y coordinate of the batterys position
[reward]         reward given when reaching the battery. default is 0.
[led color]      color of the led of the battery, [r g b],  0<= r,g,b <=1.
                              default is [0 1 0].
[radius]         the radius of the battery. default is 52.5.
[outer color]    color of the battery surounding the led, [r g b],
                              0<= r,g,b <=1.
 default is [0.93 0.93 0.93].
return: error flag       zero if no error occured during call to function,
                         non-zero othervise
battery number   a handle to this battery which can be used for moving
                              the battery
```

SHOW/HIDE FUNCTIONS

[err] = SIM_CR_SHOW_SENSOR_DIST(mode, [color(1, 3)])
Enables the simulator to plot the sensor readings (done by calling sim_cr_draw()).
To update the readings (the plot) sim_cr_get_sensor() must be called. Mode = 1
shows the distances the Cyber Rodent reads. Mode = 2 shows the true readings.
arg: mode   1 or 2. default is 1.
[color]   color of the sensor plotting
return: error flag   zero if no error occured during call to function,
                        non-zero othervise

[err] = SIM_CR_HIDE_SENSOR_DIST()
Hides the plot of the sensor readings
arg:
return: error flag   zero if no error occured during call to function,
                        non-zero othervise

UPDATE FUNCTIONS

[err] = SIM_CR_DRAW()
Draws the environment and the Cyber Rodent.
arg: none
return: error flag   zero if no error occured during call to function,
                        non-zero othervise

[err] = SIM_CR_DRAW_VISION()
Draws the batteries visable to the Cyber Rodent. Sim_cr_get_vision must be called
to update the vision information
arg: none
return: error flag   zero if no error occured during call to function,
                        non-zero othervise

[err, reward, object type, object color(1, 3)] = SIM_CR_MOVE()
Updates the Cyber Rodents position based on the current velocity of the wheels.
Detects collisions and reaching of batteries.
arg: reward   reward givent taking last step
object type   the type of object the Cyber Rodent hit if a collision
                        ocurred. 1 = wall (box), 2 = battery
object color   the color of object the Cyber Rodent hit if a collision
                        ocurred. [r g b],   0<= r,g,b <=1.
return: error flag   zero if no error occured during call to function,

non-zero othervise

[err] = SIM_CR_MOVE_BATTERY(position x, position y, battery handle)
Moves battery (battery handle) to the desired position (position x, position y)
arg: position x   new x position
position y   new y position
battery handle the handle to the battery
return: error flag   zero if no error occured during call to function,
                         non-zero othervise